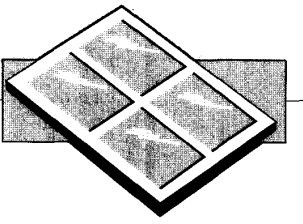


Shifting Register Windows

Shifting register windows is a new register windowing method that attempts to overcome some of the difficulties of traditional fixed- and variable-sized schemes. Using fewer register elements than a seven-window Sparc organization, shifting register windows more than halves spill/refill memory traffic, and reduces visible spill/refill cycles by an order of magnitude. In addition, shifting register windows, a scheme based on fast hardware stack and register-memory dribbling, has a very short register bus length. It also zeros registers as they are being allocated, making a common initialization unnecessary.

Gordon Russell

Paul Shaw

University of Strathclyde

The use of registers has grown considerably since the accumulators of von Neumann's 1945 machine. Index registers appeared in 1951[1] as part of the Manchester University Digital Computing Machine, followed in 1956 by general-purpose registers (within the Pegasus computer from Ferranti).[2] General-purpose registers hold commonly accessed data, such as local variables, pointers, parameters, and return values. They cannot, however, hold heap-based variables or other aliased data.[3]

One problem with the use of general-purpose registers is in the overhead incurred over subroutine calls, where register contents must be saved to memory and restored on return. Hennessy and Patterson[3] show that this overhead equates to between 5 and 40 percent of all data memory references. The common solution is to use many on-chip registers.

Designers can use either software or hardware to manage large register files. In architectures where all general-purpose registers are viewed as a single register file, software techniques[4,5] attempt to maintain values in registers over subroutine calls by using global program knowledge. To gain this global knowledge, the software allocates registers at link time.

Hardware management strategies center around register windows. This approach splits the register file into several banks, with a bank allocated on each call and deallocated on return. The on-chip banks take the form of a circular buffer: when requesting a bank that would mean that a previously allocated bank gets overwritten, the processor saves the information the requested bank contains to memory (window overflow). On returning to a previously saved register bank, the processor loads that bank from memory (window underflow).

Software techniques for maintaining values in registers help keep the hardware simple. However:

- Linking for a windowed register file is faster, and dynamic linking is easier to support.
- In the software solution, having more directly addressable registers requires more instruction bits to identify operands.
- Adding registers in a windowed architecture is transparent to the instruction set (and the user), while adding to a nonwindowed system is not.

Note that register windows cannot readily replace *all* processor registers, since globally accessible registers will still be required (such as program counter, user stack pointer, and window overflow stack pointer). Although floating-point registers can be windowed, current architectures typically leave these registers global.

Register windowing divides itself into two general sub-classes: fixed- and variable-sized. In a fixed-sized register windowing scheme, the hard-

0272-1732/93/0800-0028$03.00 © 1993 IEEE

ware designer defines the number of registers per bank, whereas in a variable-sized scheme, software specifies the bank's size at allocation time.

Fixed-sized register windows. Both the Sparc chip set[6] and the RISC II[7] use fixed-sized register windows. For these processors, the active window (that is, the currently accessible block of registers) splits into three parts: *in, local,* and *out,* with each holding eight registers. The *local* part contains registers accessible only while that window is active, *out* holds parameters to be passed to subroutines, and *in* holds the current subroutine's parameters as supplied by the parent. Whenever a new window is created, the *out* registers of the current window become the *in* registers of the new window. Deallocating the new window undoes this mapping.

Figure 1 shows three fixed-sized windows. Each column represents the parts accessible from any one window. Parts lying on the same row are directly mapped onto one another. For example, the *out* part of window 2 is mapped directly onto the *in* part of window 3. The underlying register file appears on the right.

Increasing the register file size increases internal bus capacitance. Provided that the number of windows stays small, this does not appear to affect processor cycle time.[5] However, with many on-chip banks, cycle time will certainly be affected, suggesting an upper limit on design scalability. Fixed-sized windows offer no flexibility in the number of parameters passed or locals declared. If the number of parameters exceeds the size of the *in* register part, the remaining parameters must be held in memory. Alternatively, if some registers within a bank are unused within a subroutine, window overflow/underflow will involve redundant memory transfers.

Variable-sized register windows. Figure 2 shows an organization supporting variable-sized register banks. Here, a global register stores the current window position. Its value is added to every register reference, then passed to a decoder, which selects the desired register.

The only instruction used in controlling the windows is a shift. On a subroutine call, the parent shifts the current window pointer to select the first parameter to be passed (that is, a position after the parent's local variables). A negative shift undoes this step on return from the subroutine. Once called, a subroutine can access registers from the current window pointer onwards. The Am29000[8] provides similar support for its register file.

Although this scheme supports variable-sized windows, it includes an overhead of an addition on each register access. The problem of scalability identified with fixed-sized windows remains unsolved.

Design goals for a better windowing mechanism. Each of the existing register management systems described offers comparative advantages. For example, variable-sized windows promise flexibility at the cost of performance, while the fixed scheme provides better performance at the expense of flexibility.

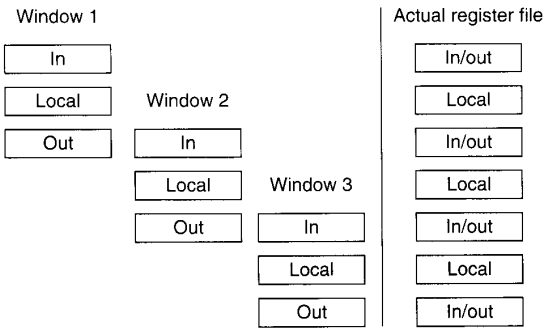


Figure 1. Three fixed-sized register window banks.

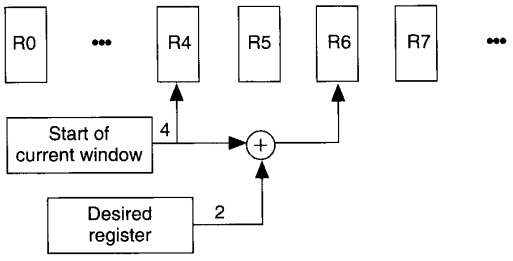


Figure 2. Organization of variable-sized register windows.

A new windowing model, shifting register windows, improves on existing register management schemes,

- the new design contains the flexibility of variable-sized register windows;
- currently accessible registers (those contained within the active window) should be the registers closest to the arithmetic logic unit, thus minimizing signal propagation times;
- the design should be without register-access overhead, such as that incurred by the addition in variable-sized windows;
- the return address for a subroutine should be stored on-chip to support fast call-return cycles;
- the design should be scalable; and so adding more on-chip space for register storage does not adversely affect access times or logical complexity.

Shifting register windows

Multiple accesses to the same register within a variable-sized windowing scheme each require an addition, even if the current window position remains unchanged between

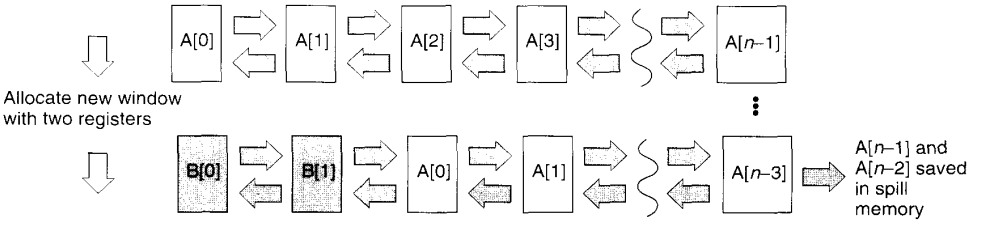


Figure 3. Register file arranged as a shifter.

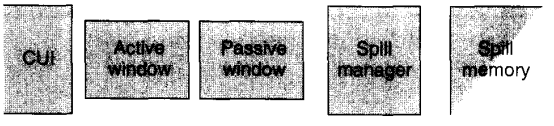


Figure 4. The register windowing system.

accesses. Keeping the active window pointer fixed and instead moving the contents of the register file avoids this situation. We can obtain this functionality by using a shifter.

Figure 3 shows a register file A[0...($n-1$)] arranged as an n cell shifter. It also shows the allocation of a new window B, which contains two elements. The program allocated this window by performing two right shifts, and will later deallocate it with two left shifts (thus losing its contents). After allocation, registers B [0,1] occupy the same physical cells as A [0,1] did before allocation. Therefore, the active window always resides in the leftmost cells. Processor designers should place this area as close to the ALU as possible.

When the processor allocates registers by shifting, it loses information contained within the rightmost register cells. To avoid this, shifted register contents should be stored in memory. In Figure 3, registers A[($n-2$),($n-1$)] have been saved. On deallocation, register contents residing in memory should be returned to the register file. This implies that the processor performs memory accesses in step with shifting, which requires the processor to stall until the accesses have completed.

Allowing the shifter to expand and contract as necessary will alleviate this stalling problem. In this way, the shifter's elasticity can absorb some elements that would have been spilled. Whenever we enlarge the shifter this way, a secondary system migrates those elements that caused the growth into memory. A similar mechanism operates when the shifter is not full. (This can occur when the processor has moved elements to memory and deallocated the registers.) The migration process stops when the shifter's capacity returns to normal, or in the case of contraction, when there are no more elements stored in memory.

An elastic shifter is constructed from n elastic cells. To the left of each cell lies a shadow cell. Both cells can hold one register element. The processor injects left or right shifting requests into the leftmost cell of the shifter, where they propagate between elastic cells. Acknowledgments propagate right to left in response to requests. During a shift, the processor uses the shadow cells as intermediate storage. Requests persist until acknowledged or cancelled. Also, at any time both left and right shift requests may exist along the length of the shifter (although only one type of request may exist between any two neighboring elastic cells). Depending on the type of requests, the shifter can appear to contain from 0 (a left shift persists for every elastic cell in the shifter) to $2n$ registers (a right shift persists for every elastic cell in the shifter). When the shifter contains no outstanding requests, we consider it stable.

Because requests must propagate from cell to cell, the elastic shifter takes longer to stabilize than the nonelastic variety. Without global knowledge of persisting requests, an elastic shifter must stabilize before the processor can predict the location of a register.

Overview. Figure 4 shows the organization of shifting register windows within a processor. Depicted are the five main entities:

- The *control unit interface (CUI)* takes register allocation/deallocation instructions from the control unit and converts these to simple shifting commands. The CUI then transfers these commands to the active and passive windows.
- The *active window* is a synchronous shifter comprised of a user-accessible set of registers (accessed by buses that span its length). Its synchronous nature allows it to stabilize quickly.
- The *passive window* forms an elastic shifter of non-user-accessible cells. It receives shifting commands from the CUI and buffers data transfers between the active window and spill manager. Because the processor never accesses the passive window directly, register buses are limited to those cells in the active window.
- The *spill manager* responds to requests generated by the rightmost cell of the passive window. When a re-

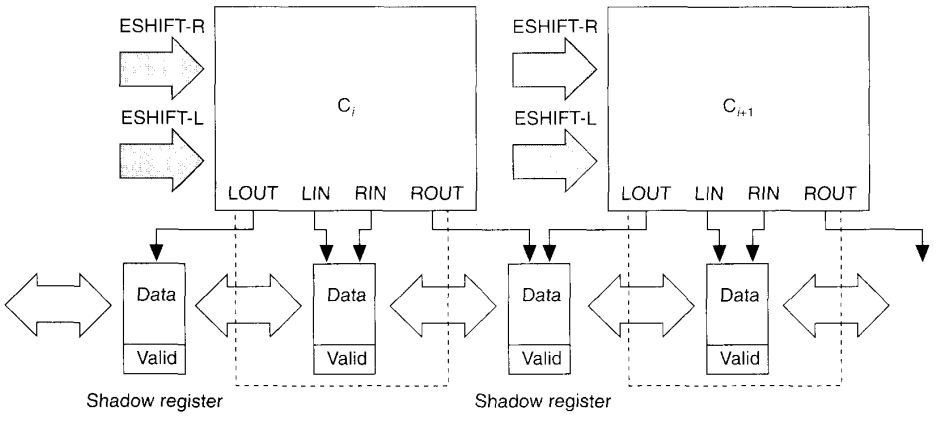


Figure 5. An elastic register cell and its neighbors.

quest to shift right arrives, the spill manager transfers the data to be shifted into memory. The opposite transfer occurs on a left shift. The spill manager stores spilled registers in a last in, first out manner. To minimize processor stalls, the spill manager undertakes memory transfers only when free memory cycles are available. To promote these free memory cycles, an instruction cache should be included in the processor.

- *Spill memory* holds register contents transferred by the spill manager during right shifts. Depending on decisions made during the processor design stage, accesses to spill memory can be normal cached accesses, or go directly to main memory.

The complete shifter. In shifting register windows, the active window contains a synchronous cells. The passive window contains b elastic cells, with a shadow cell to the left of each (thus having a maximum capacity of $2b$ data items). The shifter can shift data between the last element of the active window and the first shadow cell of the passive window.

During a right shift, the CUI instructs the active window to shift to the right. Once the active window has finished shifting, the data shifted off the end of the active window resides in the first shadow cell of the passive window. A right shift request then goes to the passive window from the CUI. Once that request is successfully acknowledged, the CUI can deliver new commands to the active window. A request to shift left goes to the passive window; when that is positively acknowledged the active window also shifts left.

In both shifting scenarios, the passive window need not stabilize before the CUI sends it subsequent shift requests. Previous requests propagate along the passive window independently from current requests.

A single register cell. The three types of cell used in constructing the overall model are known as synchronous, elastic, and shadow. All hold the same type of information:

- A data part, which holds either a program-related variable or a subroutine return address, and
- A valid bit, indicating whether the data part is currently in use.

Synchronous cell. Two global control lines are connected to each synchronous register cell. SHIFT-R signals a cell to shift out its contents to its right-hand neighbor, and shift in the contents of its left-hand neighbor. SHIFT-L signals the inverse operation.

Elastic and shadow cells. The elastic cell is more complex, using handshake lines to communicate shifting actions between itself and neighboring cells in the shifter. Each pair of neighboring elastic cells shares an intervening shadow cell through which data to be shifted communicates. Figure 5 shows this organization. Ebergen and Gingras[9] describe a similar use of shadow cells. In the stable state, the shifter holds all data within elastic cells, with the shadow cells remaining empty.

Figure 5 shows two asynchronous handshake channels (ESHIFT-R and ESHIFT-L) between each pair of elastic cells. As Figure 6 (next page) shows, each of these channels is comprised of three wires: a request line (REQ) and two acknowledgment lines (OK and FAIL). All requests must come from the left. On any one channel, a request must be acknowledged before the cell can issue a subsequent request. A request succeeds if OK is acknowledged and fails if FAIL is acknowledged. Channel ESHIFT-R (elastic shift right) controls right shifts, and ESHIFT-L controls those to the left. A

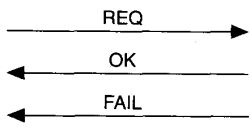


Figure 6. Composition of a handshake channel.

request to a cell fails if that cell is currently issuing the same class of request to its right-hand neighbor.

A pair of elastic cells operates as follows during ESHIFT-R and ESHIFT-L requests:

- An ESHIFT-R request made from C_i to C_{i+1} instructs C_{i+1} to first move its data to its right shadow cell, then move the contents of its left shadow cell into itself. The request succeeds if this can be done, or fails otherwise.
- An ESHIFT-L request made from C_i to C_{i+1} instructs C_{i+1} to move its data into its left shadow cell. If this can be done, the request succeeds, and C_i moves data from its right shadow cell into itself; otherwise the request fails.

Four signals control the movement of data between shadow and elastic cells: ROUT, RIN, LOUT, and LIN. ROUT latches data from the signalling elastic cell to its right-hand shadow cell; RIN latches in the opposite direction. LOUT and LIN control the symmetric operations for the left-hand shadow cell.

Each elastic cell contains a state variable *status* that takes on one of three values: IDLE, DO-ESHIFT-R, or DO-ESHIFT-L. This defines what the cell should be doing when receiving no requests from its left neighbor. The state of a cell is defined as the value of its status variable.

In the IDLE state C_i does nothing. In the DO-ESHIFT-R state, C_i makes an ESHIFT-R request to C_{i+1}. If successful, C_i assumes an IDLE state. Otherwise, its state remains unchanged, and the ESHIFT-R request will be tried again later. A similar action takes place in the DO-ESHIFT-L state. If previously in the IDLE state, C_i enters state DO-ESHIFT-R when an ESHIFT-R request from C_{i-1} succeeds. Similarly, C_i enters state DO-ESHIFT-L when a ESHIFT-L request from C_{i-1} succeeds.

Requests issued by C_i fail when the state of C_{i+1} matches the type of request. That is, if C_{i+1} is in state DO-ESHIFT-R and receives an ESHIFT-R request from C_i, the request fails and the state of C_i remains unchanged. Likewise, ESHIFT-L requests issued by C_i fail when C_{i+1} is in state DO-ESHIFT-L.

If the state of C_{i+1} does not match the request from C_i, optimizations occur. For instance, if C_{i+1} is in state DO-ESHIFT-R and an ESHIFT-L request comes from C_i, the request succeeds, the previous register transfers are undone, and C_{i+1} assumes the IDLE state. Similar actions occur if C_{i+1} is in state DO-ESHIFT-L and an ESHIFT-R request comes from C_i.

The spill manager. The spill manager is an augmented elastic cell. The interface to its left cell remains unchanged, with a shadow cell present between itself and its neighbor. The cell's right-hand interface is not to another elastic cell, but to spill memory.

The manager can exist in one of three states: idle, DO-ESHIFT-R, and DO-ESHIFT-L. During the idle state, the manager waits for requests from its left-hand neighbor.

When in the IDLE state, reception of an ESHIFT-R request causes the spill manager to examine the valid bit of its internal data. If the data is invalid, the request succeeds, the state remains unchanged, and the transfer from shadow cell to spill manager takes place. If the data is valid, the request fails, and the spill manager assumes state DO-ESHIFT-R.

ESHIFT-L requests received in the idle state succeed if the data valid bit is set or the spill memory is empty. (The amount of information stored in spill memory could be held within a counter.) Successful ESHIFT-L requests result in the normal data transfer from manager to shadow cell, and invalidation of the manager's valid bit. Whether the request succeeds or not, the spill manager sets the state to DO-ESHIFT-L if the spill memory is not empty.

State DO-ESHIFT-L indicates that a data item previously saved in the spill memory should be reloaded (and the valid bit set). DO-ESHIFT-R indicates the opposite action; data should be moved from the manager to spill memory (and the valid bit cleared). The memory transfers are delayed until both a free memory bus cycle becomes available and a hysteresis condition is met. When spilling, having the last h elastic cells of the passive window in state DO-ESHIFT-R creates the hysteresis condition. Refilling requires that these h cells be in state DO-ESHIFT-L. The hysteresis reduces thrashing (continual loading and storing of the same data elements). When the memory transfer completes, the spill manager returns to the idle state. During spill-memory transfers, requests are unsuccessful.

Whenever the spill manager's state matches the input request, that request will fail. With the state and request opposing, requests succeed provided the spill-memory transfer has not started. Receiving ESHIFT-L in state DO-ESHIFT-R places the manager in state idle. The manager then transfers its internal data to the shadow cell, and acknowledges the request as successful. Receiving ESHIFT-R in DO-ESHIFT-L sets the state to idle, and initiates the required shadow to manage transfer.

If the spill manager is starved of free bus cycles, the passive window can become either completely full or completely empty. In both cases, the processor cycles while it waits for a positive acknowledgment from the passive window. This increases the number of free bus cycles, allowing the spill manager to make the necessary memory transfers. Transfers made while the processor is stalled are called *forced* memory transfers. On a traditional windowed machine (such as Sparc),

all memory transfers to handle register spills and refill are forced: the processor can do no useful work during this time.

Typically, when the passive window contains $b - h$ data items or less, the spill manager attempts to pull data from memory. Likewise, when it contains $b + h$ items or more, the spill manager attempts to store data.

Register interfacing. This proposed interface to shifting register windows (ignoring supervisory instructions) consists of four instructions: ALLOC, DEALLOC, CALL, and RETURN. The ALLOC and DEALLOC instructions allow the machine to acquire and relinquish registers, while the CALL and RETURN instructions use the registers to store and restore the program counter. It should be possible to allocate and deallocate several registers per CPU cycle. If the number of registers that can be allocated/deallocated per cycle exceeds the number of registers requested, ALLOC and DEALLOC will complete within a single cycle. Otherwise these instructions take two or more cycles to complete. Figure 7 describes the operation of each of these instructions.

Three primitives are referenced, namely In, Out, and Signal. The In and Out operations correspond to right and left shifts, and Signal corresponds to a hardware trap. Reg is the register file of a dimension equal to the size of the visible window, addressed from 0 upwards. In takes two parameters: the data to be inserted and the valid bit state. Out requires only one parameter: where to put the data element produced by the shift left (which in this case is either to store it in the program counter, or to throw it away).

Start-up and context switching. In a multitasking system, the kernel handles process loading, saving, and initialization. All process registers and selected internal state variables must therefore be readable and writable. The passive window does, however, complicate the kernel, because the processor cannot directly access data items within this window. Instead, the contents of the active and passive windows must be flushed to spill memory on a context save, and restored again when the task is reloaded.

Turning off hysteresis and performing $a + b + 1$ right shifts flushes the registers to memory. When all elastic cells and the spill manager are in the idle state (as indicated by a global line), the flushing is complete. (To allow this global line sufficient time to stabilize, it should be possible to suspend the operation of all elastic cells. In a synchronous implementation, this could be done by removing their clocks at the source. Elastic cells are only stopped and started during a context switch.) The processor selects data elements saved during a previous context switch by setting the internal variable that points to the spill area. Performing $a + b + 1$ left shifts loads the new context's data elements into the register file. The processor then switches hysteresis back on. The reloaded process resumes execution as soon as a elements have been reloaded and propagated along the passive window into the active window. Any remaining transfers will continue to occur on free bus cycles under the control of the spill manager.

```
ALLOC(num)
    FOR i := 1..num
        IN(0,TRUE)
DEALLOC(num)
    FOR i := 1..num
        IF (reg[0].valid)
            OUT(null)
        ELSE
            SIGNAL(STACK_EMPTY)
CALL(location)
    IN(PC,TRUE)
    JUMP location
RETURN(num)
    DEALLOC(num)
    IF (reg[0].valid)
        OUT(PC)
    ELSE
        SIGNAL(STACK_EMPTY)
```

Figure 7. User-level interface.

If a process is beginning for the first time, the processor initializes the register file by setting the spill count to zero. Global lines then can be used to invalidate all valid bits and set the states of all elastic cells (and the spill manager) to IDLE.

With a shifting register windowing implementation, the overhead of context switching depends on the number of register saves and restores made by the spill manager during the switch. This overhead is no larger than that of a similarly sized (in terms of the total number of register holding elements) traditional fixed- or variable-sized scheme.

Implementation. We have designed an implementation of shifting register windows that is synchronous, thus allowing simple interfacing to a synchronous processor. Around 50 gates are required in each elastic cell to control handshaking. Analysis suggests allocation/deallocation of a register occurs in approximately 15 gate times.

Performance

The main performance benefit of shifting register windows arises from the improved access speed of the register contents, due to short bus lengths. With a 128-element register file, transferring to a shifting register window implementation with a active window of 16 elements could result in an eightfold decrease in bus length. Predicting the performance impact of short buses depends on a large number of implementation-specific factors. However, the amount of spill-memory accesses and register management related processor stalls is independent of implementation, and can be

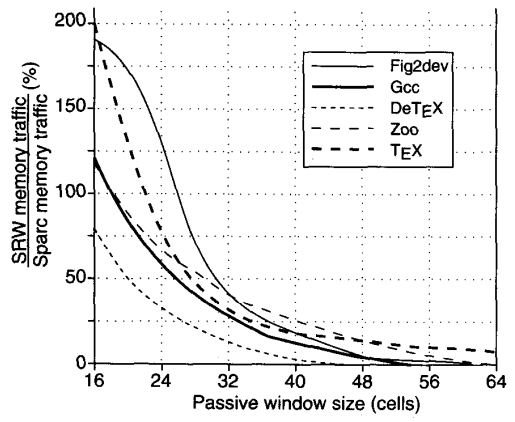


Figure 8. Shifting register window's memory accesses.

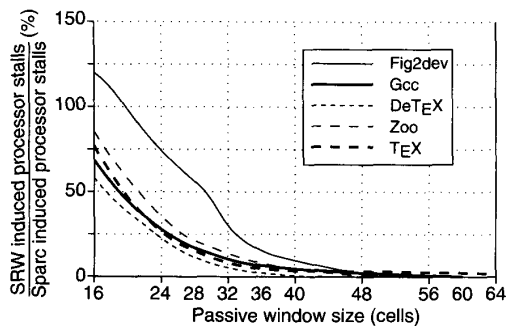


Figure 9. Shifting register window's processor stall cycles.

modeled unambiguously.

To predict memory accesses caused by spilling and restoring registers and processor stalls introduced due to register management, we constructed a simulator. This simulator traced Sparc binaries using Shadow,[10] and modelled instruction and data caches, Sparc fixed-sized windows, and shifting register windows. This simulation allowed for comparisons between shifting register windows and the Sparc windowing system, and demonstrated the effect that shifting register windows has on memory bus contention. Note that the results are for one process running to completion, with no context switches.

In the simulation, we used an 8-Kbyte instruction cache and a write-back, 4-Kbyte data cache. Both caches were direct mapped, with a block size of 16 bytes. The simulation assumed that the Sparc had seven windows, with a spill/refill of one window costing 60 cycles.[3]

For the shifting register windows simulation, we used a 16-cell active window. The spill manager had a hysteresis of eight elastic cells, and bypassed the data cache when accessing spill memory. We assumed that a spill/refill of a single register required four cycles, and that six registers could be allocated per CPU cycle.

To reduce contention between cache-miss induced memory accesses and spill-memory accesses, our simulation used a pipeline look-ahead. This warns the spill manager of forthcoming data cache accesses using information gleaned from the pipeline. Such a warning prevents the spill manager from initiating spill memory accesses. Our simulation assumed a look-ahead of two cycles.

The benchmarks used were Tex and Gcc (used throughout Hennessy and Patterson[3]), Detex, zoo, and fig2dev. We varied the passive window size from 16 to 64 elastic cells. We produced two graphs, detailing ratios of shifting register windows against Sparc windows for spill/refill memory accesses (Figure 8) and processor stalls (Figure 9). Processor stalls include memory collisions between the spill manager and caches, spills when the passive window is completely full, and refills when the passive window is completely empty. Associated with the refill is the time taken for the last register loaded to propagate to the active window. In practice, we found this propagation time to be negligible.

From Figure 8, shifting register windows appears to generate less memory traffic than the Sparc windowing system for passive window sizes over approximately 24. Maintaining low memory traffic is a goal in multimaster systems.

Figure 9 shows that, for passive window sizes over 16, processor stalls for shifting register windows typically are less than that incurred by the Sparc. Reducing processor stalls increases instruction throughput.

The hysteresis value chosen in the simulation was a balance between two performance-related factors: forced spill-memory accesses and cache collisions. A low hysteresis value results in fewer forced memory accesses at the expense of increased cache collisions. The converse situation pertains for a high hysteresis value. In terms of circuit area and signal propagation times, a low hysteresis value is desirable. In a processor design, the exact figure chosen will also depend on cache miss rates, look-ahead distance, and the speed of external memory.

THE PROPOSED SHIFTING REGISTER WINDOWING mechanism substantially improves on existing schemes. The main benefits are minimization of register bus length, a reduction in spill/fill overhead, and automatic zeroing of local registers.

These improvements come at the cost of added control circuitry and increased time taken to allocate local registers:

allocation time is proportional to the number of registers requested. Processor designers should strive to achieve sufficient allocation rates such that their designs can handle the majority of register allocations in a single cycle. Since the passive window is the slowest part of the design, using two passive windows will improve allocation rates, with the active window communicating to each in turn. The performance gain comes at the expense of a more complex, two-port spill manager.

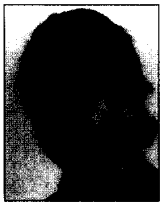
We hope that the ideas presented in this article will both increase the performance of register-based processors and encourage further research into register windowing paradigms. We are actively investigating the implications of shifting register windows for architectures based on multiple stacks. Each stack uses a small shifting register window. Initial analysis shows that the problems demonstrated with shifting register windows (such as multicycle allocation times) are avoided using this approach.

Acknowledgment

We thank Paul Cockshott and Robert Lambert for providing useful feedback on earlier drafts of this article.

References

1. T. Kilburn, "The Manchester University Digital Computing Machine," *Charles Babbage Institute Reprint Series for the History of Computing, Vol. 14,* M.R. Williams and M. Campbell-Kelly, eds., MIT Press, Cambridge, Mass., 1989, pp. 138-141.
2. D.P. Seiwiorek, C.G. Bell, and A. Newell, *Computer Structures: Principles and Examples*, McGraw-Hill International, Singapore, Malaysia, 1982.
3. J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Palo Alto, Calif., 1990.
4. S. Richardson and M. Ganapathi, "Code Optimization Across Procedures," *Computer*, Vol. 22, No. 2, Feb. 1989, pp. 42-50.
5. D.W. Wall, "Register Windows vs. Register Allocation," *Proc. SIGPLAN '88 Conf. on Programming Language Design and Implementation*, ACM Press, New York, Vol. 23, No. 7, June 1988, pp. 67-78..
6. B. Glass, "Sparc Revealed," *Byte*, Vol. 16, No. 4, Apr. 1991. pp. 295-302.
7. D.A. Patterson and C.H. Séquin, "A VLSI RISC," *Computer*, Vol. 15, No. 9, Sept. 1982, pp. 8-21.
8. *Am29000 User's Manual*, Advanced Micro Devices, Sunnyvale, Calif., 1987.
9. J.C. Ebergen and S. Gingras, "An Asynchronous Stack with Constant Response Time," Tech. Report CS-93-11, Computer Science Dept., Univ. of Waterloo, Waterloo, Canada, Aug. 1992.
10. *Introduction to Shadow,* Sun Microsystems, Mountain View, Calif., Apr. 1992.



Gordon Russell is a PhD research student in the Computer Science Department of Strathclyde University, where he is examining networked persistent object-oriented systems. Other interests include processor design, portable computer systems, and dynamic recompilation methods.

Russell's first degree was a BSc in computer science and electronics, also from Strathclyde. Currently, he is a student member of the IEEE, British Computer Society, and Institution of Electrical Engineers.

Paul Shaw currently works on the automatic compilation of Occam to digital logic and on the simulation of physical systems on field programmable gate arrays. His interests include computer architecture, asynchronous systems, cellular automata, and highly parallel machines.

Shaw received his BSc degree in computer science and electronics from the University of Strathclyde in 1990, where he is now studying as a PhD student.

Direct any comments regarding this article to Gordon Russell, University of Strathclyde, Department of Computer Science, 26 Richmond Street, Glasgow, Scotland; gor@cs.strath.ac.uk.