Real-time Resource Allocators in Network Processors using FIFOs

Dale Parson
Agere Systems
dparson@agere.com

Abstract

Embedded systems must perform many activities while satisfying real-time constraints. Network processors, for example, must interact with packet streams at the high speed of communication networks. One class of activity requiring a real-time mechanism is dynamic resource management. A network processor may need to allocate and recover TCP ephemeral port numbers, regions of memory, entries in a connection table, or other application resources, each within fixed time. Unfortunately, most software algorithms for resource management do not satisfy fixed time constraints. In addition, the intrinsic overhead of software execution adds unwanted latency to resource management. Finally, in a multithreaded processor, software must employ costly spin locks or complex hardware mutexes to avoid conflicting concurrent access to a resource pool shared among contexts. This paper presents a means for using a hardware FIFO as a real-time pool for resource identifiers. A network processor initializes a FIFO with identifiers representing resources. Once real-time processing is under way, a processor context allocates a resource by dequeueing the identifier at the FIFO's head. A processor recovers a resource by returning the identifier to the FIFO's tail. Allocation and recovery are single-instruction, atomic operations that operate within the upper time bound of a bus transaction.

Keywords: allocation, FIFO, network processor, queue, real-time, resource management[1]

1. The hardware queue-based resource management mechanism reported in Section 3 of this paper has been submitted to the U.S. Patent Office in a patent application by Agere Systems Inc. and the author.

1. Network processor resource management

1.1 Time-constrained resource management

Resource management in an embedded processor system is the allocation, use, and recovery of resource identifiers. These identifiers reside in a resource pool when their corresponding resources are not in use. Resource allocation consists of removing an identifier from its pool, mapping that identifier to the resource, and using the resource. Resource recovery consists of mapping the resource to its identifier and returning the identifier to its pool.

Memory is a classic example of a managed resource. A typical memory manager performs memory allocation by partitioning a contiguous region of memory into fixed or variable size pieces that it can supply to application processing requests for memory to house application data structures [1]. Unfortunately, most software memory managers do not operate in fixed time. In order to achieve maximum flexibility for varying-size application requests, and to achieve maximum use of available storage, the partitioning and coalescing of memory regions take unbounded, varying time that corresponds with the varying partitions and sub-partitions of memory. Such memory managers are not suitable for real-time allocation and recovery.

The problem of unconstrained time for resource management is largely a matter of *search* [2]. For example, if a memory manager must search a collection of varying-size memory partitions according to some criterion — best fit, or region with additional contiguous area if growth is required, etc. — execution occurs within the unfixed range of time required to perform this search. Likewise, recovery may need to search according to similar structuring criteria. Real-time allocation and recovery must avoid search.

Part of a means to avoid search is to utilize a pool of fixed-size resources. To continue with the

memory management example, if it is possible to define a constant size for all memory partitions in a pool corresponding to some application data structure such as a row in a table (e.g., a fixed-size array of C structs, where each struct is a row in the table), then it is possible to preallocate a table of N rows before real-time processing begins, and then allocate a row at a time during real-time processing by incrementing a counter and using the value of that counter as an index into the table each time the application needs a new row.

Fixing the size of application data structures within static arrays fixes memory costs, and in cases where application memory demands may vary greatly in different program runs, every execution of a program incurs the maximum, worst-case overhead of static memory allocation. This approach trades space for predictable time. It is important to note, however, that this trade-off is typically much worse for multitasking run-time environments that may contain processes with a wide mix of memory requirements than it is for a real-time embedded system such as network processing. A network processing program must guarantee availability of the number of memory-resident structures needed to meet its run-time storage requirements for application data. The only way to guarantee run-time availability is to reserve necessary memory upon program initialization, without sharing a dedicated region of memory for other uses. Fixed, O(1) time complexity goes hand in hand with fixed, O(1) memory complexity. This is not to say that a program must fit unrelated data into a single pool of structures, wasting space for unused fields. There can be a different pool (array) for each real-time data structure, each housing a fixed number of elements of fixed size.

Fixing the size of entries in a resource pool is only a partial solution because an application can return resources to a pool in an order different from the allocation order. There must be some management structure beyond a simple counter to track the remaining variability after varying size is eliminated, which is variability in order of recovery and re-allocation.

Two data structures with the required O(1) allocation and recovery characteristics are *LIFOs* (last-in first-out data structures, a.k.a. *stacks*) and *FIFOs* (last-in first-out data structures, a.k.a. *queues*), often implemented themselves as contiguous fixed-size regions of memory [3]. The remainder of this section illustrates several uses of integer identifiers managed by a FIFO pool in network processing applications.

Table 1 shows a very basic application of a FIFO of integers, a collection of pointers to fixed-size partitions of a memory region as already discussed. The concrete application driving this example is thread-local storage in Agere's APP550 network processor [4,5]. There are 64 hardware contexts in the classification engine of this multithreaded processor. Each classification engine context is responsible for examining a packet and determining what to do with the packet (e.g., discard, route to an outgoing network, and/or modify packet contents). A classifier context produces an integer *classification* based on the contents of its packet and optionally on information extracted from earlier, related packets and stored in memory. Upon completion of its work, the classifier passes this classification integer along with the packet to other APP550 processor engines that perform the work of discard, routing, transmit scheduling, packet modification and other non-classification jobs that use the classification result.

Table 1: Memory pointers before allocation

pointer to memory region 0 (e.g., 0x1000)
pointer to memory region 1 (e.g., 0x1100)
additional pointers . . .
pointer to memory region N (e.g., 0x1N00)

Each classifier context in the APP550 houses sixteen general purpose, 32-bit registers accessible only within that context, and sixteen additional 32-bit registers configured as a stack for storing function-local variables within the context. For many applications this register set provides sufficient thread-local storage, but in cases where it does not, it is necessary to spill registers to thread-local RAM in order to reuse the registers. To support this scenario the network processor initializes a software FIFO with the set of pointers to memory regions in Table 1, each of which can be used as a single context's thread-local storage. During real-time packet processing the arrival of a packet activates its classification context via hardware, and one of the first steps of a context requiring thread-local RAM is to retrieve a memory pointer from Table 1 and store it in a register. The context uses the register as a base address for register spills and loads. When the context has completed classification of its packet, the context returns the pointer to the FIFO, dispatches its packet and classification result to other APP550 engines, and pauses until activated by another

arriving packet.

For this application of resource management, an O(1) FIFO or LIFO is adequate. A given context can use any partition of the thread-local storage pool, as long as it is the sole user of that partition. The context releases its pointer back into the pool when it is done classifying its packet. Note that although the FIFO of Table 1 is initialized in ascending order 0x1000, 0x1100, ..., 0x1N00, contexts may return memory pointers to the pool in a different sequence. As long as a context is the sole user of a partition of thread-local storage, the order of return and subsequent reallocation of pointers in the FIFO does not matter. What matters is O(1) allocation and recovery time that is atomic. Contexts must avoid time indeterminacies and concurrent manipulation of the FIFO.

1.2 Order-sensitive network port management

An application of a resource pool that is sensitive to order is allocation and recovery of TCP/ UDP port numbers in Network Address-Port Translation (NAPT) [6-8]. A typical NAPT box acts as a gateway between a private IP network and the public Internet. A private enterprise network may contain a large number of private IP addresses that are valid only within the private network domain. When a private network host attempts to make a TCP connection or send a UDP datagram to a host on the public network, the IP source address of the private host is not valid for the public Internet. Private IP addresses reside in a range that may be reused in other private networks; they should not appear on the public Internet [9]. Each TCP packet or UDP datagram sent from a private IP source to a public IP destination has a private IP source address and associated 16-bit TCP or UDP port number. NAPT's job is to map each private IP address + port source to a public address + port source when sending a public-destined packet. NAPT is initialized with a limited set of valid IP addresses for its public Internet connection, each of which can support 2^{16} TCP/UDP ports. Private hosts can send public-destined packets with overlapping source port ranges, so that NAPT cannot use a private network port number directly. Instead, whenever a new private IP address + port combination appears in a public-destined packet, NAPT must allocate a port number from a pool associated with one of NAPT's public IP addresses. Thereafter, NAPT maps private address + port to public address + allocated_port for private-to-public packet source address fields, and it maps public address + allocated_port to private address + port for public-to-private packet destination address fields. NAPT allocates a port from a public pool only upon the appearance of a new private address + port combination, and it returns a port to its pool if this private-to-public mapping expires after not being used for some predetermined time. Mapping private address + port to public address + allocated_port and the inverse mapping require a mapping mechanism such as content addressable memory (CAM) or a hash table [5]. This discussion does not examine this mapping, focusing instead on allocation and recovery of public TCP/UDP port numbers from a pool.

Ephemeral port numbers [10] are so named because they are not associated with some well-known service accessed via TCP or UDP; instead, they are allocated dynamically to supply port identifiers for client processes on an IP network. Since NAPT has a limited set of ports available for its limited set of public IP address (as few as one public IP address), it must reuse ports that have expired. FIFO allocation and recovery of ephemeral port numbers is ideal because FIFO reallocation ensures that a port recovered after expiration will not be reallocated until after all other ports have been allocated. For example, if a port pool FIFO is initialized with the ephemeral values 1024, 1025, ..., 4095, where 1024 is at the head of the FIFO, initial allocation occurs in that order. Suppose that after 1024, 1025, and 1026 have been allocated for NAPT mappings, 1024 and 1025 remain in use while 1026's mapping expires. NAPT then returns 1026 to the tail of the FIFO, delaying its reuse after expiration the maximum possible time. If any delayed packets on either network require public port 1026 for their mapping, NAPT will have deleted their mapping and will correctly discard them. Allowing immediate reuse of a public port such as 1026 from the pool could lead to aliasing problems, since delayed packets associated with the expired mapping and new packets associated with a new mapping would both map public port 1026 using the new mapping. FIFO reallocation minimizes the likelihood of this aliasing.

1.3 Connection database management

Table 2 gives another example of resource allocation, that of rows in a TCP connection status table. Each row corresponds to a C struct, and the entire table corresponds to an array of these structs. A row is allocated for a new connection, and the row is returned to a pool when its connection closes or expires. Many connection-oriented network processor applications use a similar data structure.

Table 2: TCP connection status table with Index as a managed resource

Index	Source IP	Source port	Destination IP	Destination Port	Connection Status	Other State
0	192.19.194.178	1024	64.236.16.52	80	SYN1,SYN2	in use, etc.
1	192.11.226.2	32769	64.236.16.52	80	SYN1	in use, etc.
2	192.20.12.56	4000	192.11.226.2	21	SYN1,SYN2	in use, etc.
3						free

This example is an application-specific case of memory management discussed for Table 1. The managed resource pool is a list of numbers in a FIFO, initialized to 0, 1, ..., N-1, where there are N rows in the status table. A new connection allocates a row in the table in which to store its status by dequeueing its index from the head of its FIFO pool. A terminated connection recovers a row in the table by enqueueing its index at the tail of its FIFO. As is the case with NAPT port management, FIFO management of connection row indices ensures maximum aging before reuse of expired connection status. For example, if an outdated packet with source IP + port 192.19.194.178:1024 arrives after connection expiration has recovered its row, it will find the row marked as "free" rather than "in use," and the network processor will discard the packet. FIFO reuse again provides time for discard of outdated packets.

Any pool of resources whose members can be represented by integers or similar bit patterns can be managed by FIFO allocation and recovery. Allocation of hardware resources such as sensors or communication lines are additional examples. For network processor applications, FIFO management works well with expiration-based recovery of resources as discussed. Many other applications of resource pools that do not entail expiration could use a LIFO or FIFO for O(1) allocation and recovery with equal facility.

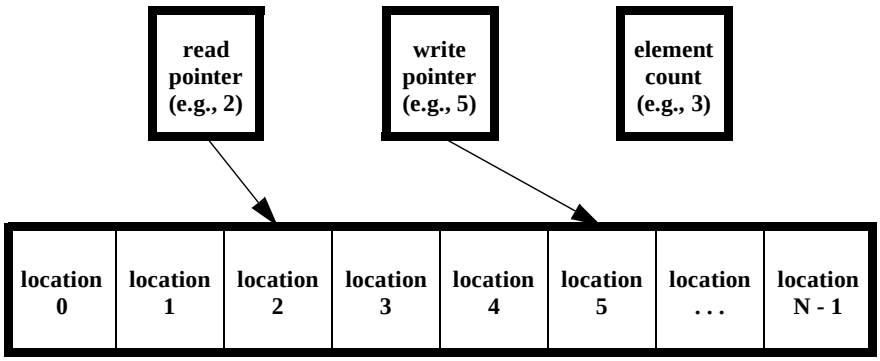

Figure 1: Circular buffer implementation of a FIFO in software or hardware

2. Software FIFO resource management

While FIFO resource pools provide O(1) allocation and recovery time complexity and maximum delay before reuse of expired resources, software FIFO pools present some problems for real-time systems, especially those running on multithreaded processors. This section summarizes the construction and problems of software FIFOs.

Figure 1 shows the most common implementation of a software [3] or hardware [11] FIFO, the *circular buffer*. The *read pointer* indicates the head of the queue, and the *write pointer* indicates the tail of the queue. Reading and writing occur at the locations referenced by the respective pointer, followed by a post-increment of the pointer, modulo the FIFO size N. The *element count* distinguishes between an empty and a full FIFO, since the read and write pointers hold identical values in both those cases. Hardware FIFOs often replace the element

count with a 1-bit empty/full flag; some provide a half-full flag as well.

Listing 1 gives the O(1) algorithm for FIFO resource allocation from the pool of Figure 1.

```
if element_count equals 0 {
      assert an empty_error_flag
} else {
      set resource = contents at read_pointer
      increment read_pointer
      if read_pointer >= N {
            set read_pointer = 0 }
      decrement element_count
      return resource to application
}
```

Listing 1: FIFO resource allocation

Listing 2 gives the O(1) algorithm for FIFO resource recovery to the pool of Figure 1.

```
if element_count equals N {
      assert a full_error_flag
} else {
      set contents at write_pointer to recovered
            resource
      increment write_pointer
      if write_pointer >= N {
            set write_pointer = 0 }
      increment element_count
}
```

Listing 2: FIFO resource recovery

Despite the fixed-time complexity of these algorithms, there are some problems associated with software queues. Depending on the target processor architecture, each algorithm expands to at least eight target instructions, and typically to more. Moreover, storing the read and write pointers and element counter of Figure 1 in RAM entails more memory latency than access to an individual queue element housing a resource identifier. Listing 1 requires a fetch and a store of both element count and read pointer in addition to a fetch of a queue element, totalling five processor interactions with memory. Listing 2 requires five similar interactions with memory for recovering a resource, using the write pointer instead of the read pointer, with storing a resource element taking the place of fetching a resource element.

In addition to latency issues, the algorithms of Listings 1 and 2 must execute atomically in a multithreaded processor such as the APP550 network processor, presenting more performance problems. When multiple software threads or hardware contexts need to use a resource pool concurrently, they must serialize access to that resource to avoid the *critical section problem* (i.e., simultaneous access to shared data structures by multiple threads) [12]. For example, two concurrent contexts allocating a resource could both find an element count of 2 in Listing 1 but decrement it to a value of either 1 or 0, depending on their compiled instructions and relative timing.

Unlike software multithreading, where an operating system can de-schedule one thread that requests to enter a critical section already in use by another thread, a multithreaded hardware processor that supports genuine concurrency must use hardware to keep a context out of a critical section being used. If there is no hardware support for a mutex (mutual exclusion lock) such as a hardware semaphore, the hardware context must check a semaphore bit stored in memory using a test-and-set instruction within a spin loop (a.k.a. spin lock) that is also executed by other contexts contending for that semaphore. Only the context that acquires the semaphore is free to manipulate its associated resource FIFO; other contexts must repeatedly test the semaphore until the owning context releases it.

An early experimental implementation of NAPT on the APP550 spent the majority of its context execution time spinning on resource semaphore bits in memory, despite the fact that the critical sections of code were very short and fast. The reason is that only one context can acquire the resource pool semaphore at a time, and therefore every other context in the processor repeatedly writes and reads information to the memory bus via the test-and-set instruction, contending for the memory bus. The context holding the semaphore must contend for the memory bus when releasing the semaphore. The situation is analogous to funneling multiple lanes of automobile traffic into a single toll booth. Memory latency skyrockets, and for an embedded processor, so does unproductive power consumption.

The solution for the prototype APP550 NAPT code was to move as many application data structures as possible out of spin loop-protected memory and into hardware-protected mechanisms such as the APP550 policing engine memory, which allows only single-threaded access via the policing engine which is invoked by classifier contexts. When more than one context invokes the policing engine, subsequent contexts stall until the first completes. Although contexts stall, the spin loops and concomitant pollution of the memory bus are eliminated, and performance improves dramatically. The only remaining data structure requiring a spin

loop-protected queue is the TCP/UDP port pool for NAPT already discussed. This pool continues to suffer from spin loop overhead, but since only a minority of TCP packets actually initiate a new connection and access the port pool, the overhead is tolerable.

There are two mechanisms going into the upcoming APP650 network processor from Agere as a result of these NAPT experiments. First, there will be four distinct 1-bit hardware semaphores with instructions for obtaining and releasing a semaphore. A context requesting a free semaphore acquires that semaphore by inverting its value. Contexts requesting a busy semaphore stall, entering a hardware FIFO of context IDs until the owning context releases the semaphore. Releasing the semaphores dequeues the context at the head of the FIFO if there is one, reactivating it; otherwise, the semaphore bit inverts to its free value. These hardware semaphores avoid the memory bus pollution and power consumption of spin loops while also avoiding taxing the policing engine, which is normally used for more complex network processing tasks.

The other enhancement to the APP650 over the APP550 is the provision of a set of registers shared by all contexts. In the APP550 it is necessary to store the read and write pointers and element counter of Figure 1 in RAM because contexts have no other means of sharing data. In the APP650 it will be possible to house these data in cross-context registers. Access to a resource FIFO will be arbitrated by a 1-bit hardware semaphore (as discussed in the last paragraph) protecting a circular queue. With the read and write pointers and element counter of Figure 1 resident in cross-context registers, the number of memory accesses for Listings 1 and 2 drop from five to one. The APP650 eliminates the APP550's two main sources of overhead for software resource FIFOs, spin loop bus contention and memory latency.

The addition of hardware semaphores and cross-context registers in the APP650 is not a matter of correcting deficiencies; it is a matter of extending the range of applications to include more types of *stateful processing*. Earlier generations of multithreaded network processors including the APP550 focused on minimizing context interaction in support of stateless processing, i.e., classifying, modifying, and scheduling a packet based primarily on its contents, without real-time comparison of its fields to those of preceding or concurrent packets. Excessive multithreaded synchronization intrinsically incurs delay. The APP550 does include some mechanisms for stateful classification of a packet. They include memory-resident arithmetic/ logic and sequence number test-and-set instructions that execute atomically when invoked by concurrent contexts. The APP550 even includes a special-purpose variant of the hardware semaphore called an *ordered function queue* for serializing access to a variable shared among contexts. Each of four ordered function queues enforces wire ordering of context access to a shared memory variable guarded by that queue. Arrival of a packet dispatches a context to process it; if a context attempts to use a variable guarded by an ordered function queue, it will not get access to the variable until earlier contexts have done so. In fact, the circuit implementation of APP650 semaphores uses much of the synchronization hardware of APP550 ordered function queues, extending it to protect multiple memory locations and cross-context registers, requested explicitly by program instructions rather than implicitly by single-variable memory references, as is the case for ordered function queues.

3. Hardware FIFO resource management

After eliminating spin loop bus contention and memory latency for multithreaded processor access to FIFO resource pools via hardware semaphores and cross-context registers, respectively, we are left with the instruction latency of running between eight and tens of instructions to perform the work of Listings 1 and 2. This remaining latency leads us to look at a hardware implementation of FIFO resource pools.

Figure 2 shows two conventional applications of FIFOs [11]. The FIFO on the left buffers bursty input data arriving from a communications network or an input device such as a video camera. Although the average arrival rate of input data must be low enough to allow processing by the CPU, data may arrive in bursts whose rate exceeds that of the CPU for some period of time. In these cases the input FIFO smooths the bursts by buffering them. The hardware FIFO has sufficient speed to accept the bursts, while the CPU reads the input FIFO (possibly via DMA) at an acceptable rate. Similarly, the output FIFO at the right of Figure 2 smooths bursts of data delivered by the CPU (possibly via DMA) to an output network or device such as a printer by buffering the data. An additional use of a FIFO is to connect processing stages in a processor such as a pipelined network processor [5], again to avoid having bursts in the data source making immediate processing demands on the data sink, while maintaining an average data flow rate acceptable to both processing stages.

Figure 3 shows a novel configuration of a

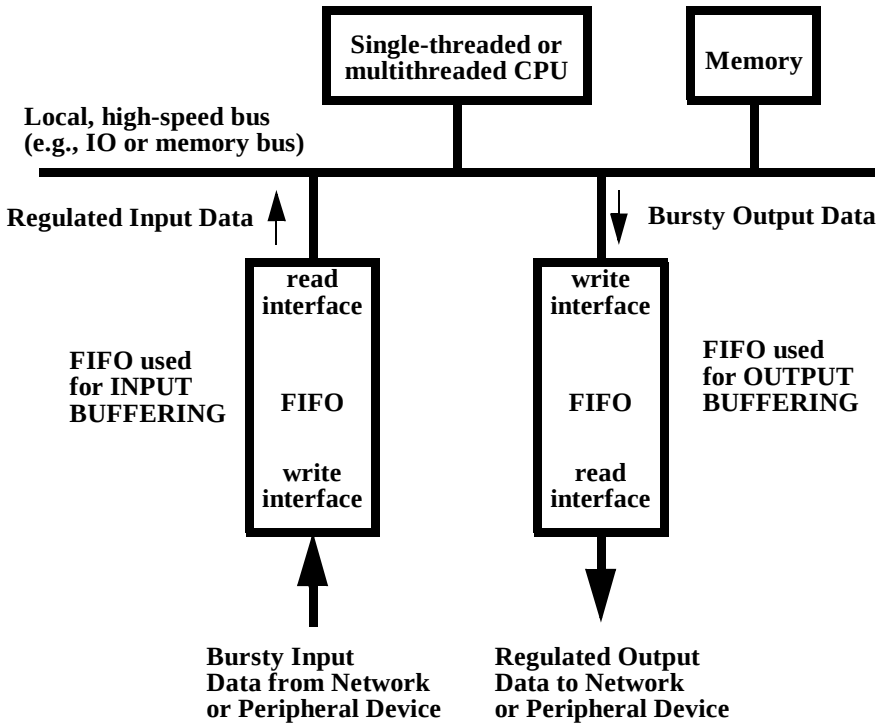

Figure 2: Conventional use of hardware FIFOs to smooth bursts in input or output data

hardware FIFO used as a resource pool. The CPU is both the source and the sink for the FIFO, again possibly via DMA. This hardware configuration supports management of a pool of resource identifiers as already discussed, and it has a number of advantages over software FIFO resource pools.

• Allocation or recovery of a resource identifier from/to the FIFO pool entails one CPU instruction, with latency dominated by the time to read or write one FIFO word. Inclusion of the FIFO within the CPU would reduce latency by avoiding contention and delays on the system bus.

• The read and write pointers and element counter of Figure 1 have become registers within the FIFO, freeing CPU registers that would otherwise be used for a software FIFO if available.

• Bus arbitration logic handles contention among multiple hardware contexts for resource pool access. There is no need for spin loops or hardware semaphores. In cases where hardware semaphores are available, the configuration of Figure 3 frees these semaphores for other application uses.

• The system could also be a multiprocessor system with multiple processors accessing the resource pool via the system bus.

The configuration of Figure 3 comes at the cost of dedicated hardware that is only useful for O(1), hardware-speed application of FIFOs within a processor system. Whereas hardware semaphores and cross-context registers provide building blocks for an assortment of cross-context data structures, the FIFO of Figure 3 is strictly a queue intended to house a resource allocation pool. Hardware LIFOs and double-ended queues could also be used in embedded system configurations similar to Figure 3, but both the ready availability of stock FIFO hardware designs and the preferred applicability of FIFO pools to applications that recover resources based on expiration periods recommend using hardware FIFOs.

4. Conclusions

There are several O(1) hardware and software data access structures, including indexed addressing (e.g., indexing into memory devices or software arrays), hash tables (O(1) for an ideal hash function, with content addressable memory as a hardware counterpart), FIFOs, LIFOs and double-ended queues. Application of a FIFO as a hardware resource pool is appropriate because of its O(1) allocation and recovery time complexity, and for expiration-based recovery because of its maximum delay before reallocating a recovered resource identifier. Software FIFOs are limited by multiple instruction latency, presenting substantial critical

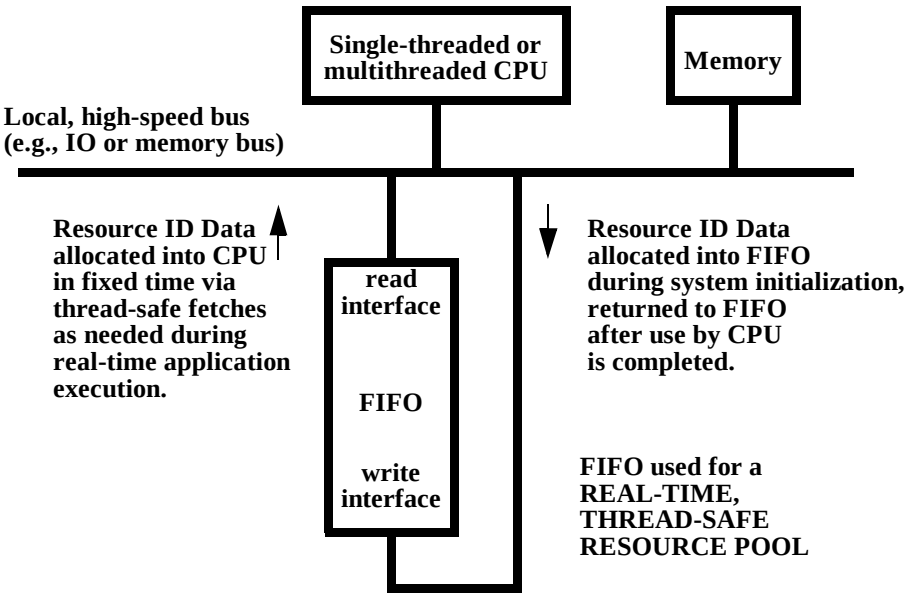

Figure 3: Novel use of hardware FIFO to allow thread-safe, fixed time allocation of Resource IDs

section delays for multithreaded processors. Such processors must use spin locks that clutter the system bus with contention for access and unnecessary power consumption, or blocking hardware semaphores to avoid spin locks, to avoid cross-context interference during software FIFO access. However, hardware semaphores and cross-context registers added to a multithreaded processor do provide support for an assortment of software-controlled inter-context resource and communication structures that include FIFO resource pools.

Hardware FIFOs are typically used to buffer bursts in input or output data arrival rate as shown in Figure 2 [11], or to decouple timing between adjacent stages in pipelined processing. Using a FIFO to store resource identifiers within a processing system as shown in Figure 3 is a novel application that avoids the multiple instruction latency and synchronization issues of a software FIFO. Allocation and recovery occur within a single instruction, and available bus arbitration hardware avoids contention during concurrent access.

5. References

[1] R. Jones, *Garbage Collection, Algorithms for Automatic Dynamic Memory Management*. Chichester, England: John Wiley & Sons, 1999.

[2] D. Knuth, *The Art of Computer Programming, Volume 3, Sorting and Searching*. Reading, MA: Addison-Wesley, 1973.

[3] Aho, Hopcroft and Ullman, *Data Structures and Algorithms*, Section 2.4, "Queues." Reading, MA: Addison-Wesley, 1983.

[4] Agere Systems, Advanced PayloadPlus® Network Processor (APP550) Product Brief, September, 2002, http://www.agere.com/enterprise_metro_access/docs/PB02024-2.pdf.

[5] D. E. Comer, *Network Systems Design using Network Processors*, Agere version. Upper Saddle River, NJ: Prentice Hall, 2004.

[6] "Traditional IP Network Address Translator (Traditional NAT)." Internet Engineering Task Force and the Internet Engineering Steering Group, Request for Comments 3022, January, 2001, see http://www.rfc-editor.org/.

[7] "Architectural Implications of NAT." Internet Engineering Task Force and the Internet Engineering Steering Group, Request for Comments 2993, November, 2000, see http://www.rfc-editor.org/.

[8] "IP Network Address Translator (NAT) Terminology and Considerations." Internet Engineering Task Force and the Internet Engineering Steering Group, Request for Comments 2663, August, 1999, see http://www.rfc-editor.org/.

[9] "Address Allocation for Private Internets." Internet Engineering Task Force and the Internet Engineering Steering Group, Request for Comments: 1918, February, 1996, see http://www.rfc-editor.org

[10] W. R. Stevens, *TCP/IP Illustrated, Volume 1 — The Protocols*. Boston, MA: Addison Wesley, 1994.

[11] "FIFO Architecture, Functions, and Applications," Texas Instruments, November, 1999, see http://focus.ti.com/lit/an/scaa042a/scaa042a.pdf.

[12] B. Carlson, "Packets challenge next-gen nets." EETimes, August, 2002, http://www.eetimes.com/story/OEG20020802S0033.