Context Management in the ID Run Time System

Computation Structures Group Memo 319
September 21, 1990

Jonathan Young

This report describes research done at the Laboratory of Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-84-K-0099.

Context Management in the ID Run Time System

Jonathan Young

September 21, 1990

Abstract

A Monsoon multiprocessor includes any number of processing elements and I-structure boards, connected by a network. The Monsoon ID Run Time System (RTS) is that collection of low-level resource management routines which must exist in order for compiled ID programs to execute on a Monsoon multiprocessor. We describe implementations of the two most primitive RTS routines, `GET-CONTEXT` and `RETURN-CONTEXT`. A *context* (or *frame*) is a contiguous, non-interleaved block of storage, and one is necessary for every procedure call. We concentrate on the implementation of context management routines which are completely processor-local, which are highly optimized, and which have a high degree of thread independence. We first develop schemes which lock processor-global resources, including a simple free-list and various queue implementations. We then discuss more independent, two-level schemes, including one which provides for automatic deferral of `GET-CONTEXT` requests.

1 Introduction

A Monsoon multiprocessor includes any number of processing elements (PEs) and I-structure (IS) boards, connected by a network [5, 2]. The Monsoon ID Run Time System (RTS) [1] is that collection of low-level resource management routines which must exist in order for compiled ID programs [4] to execute on a Monsoon multiprocessor.

In this document, we describe several implementations of the two most primitive RTS routines, `GET-CONTEXT` and `RETURN-CONTEXT`. A *context* (or *frame*) is a contiguous, non-interleaved block of storage on a processing element. One context is necessary for every procedure call (as well as certain other operations). Since each context points to storage on one particular PE, we plan to achieve load balancing in the near term by distributing the contexts randomly to all PEs, so each PE manages a collection of contexts from all other PEs. Currently, there is no plan to achieve load balancing by migrating contexts across PEs; a context will always be managed by the same PE.

The Monsoon PE is a pipelined architecture. Currently, the machine can execute up to eight separate "sequential threads" in parallel. Although the ID language and compiler ensure that no two user threads can interfere with each other, the RTS routines must use locks and careful discipline to avoid interfering with any other thread.

RTS routines are called via asynchronous traps from user code. On the Monsoon PE, asynchronous traps are allocated a special *ephemeral context*. There is a different ephemeral context for each thread. Thus, trap code such as the RTS is constrained to be an uninterrupted thread of instructions unless it allocates another context.

Non-local memory references are *two-phase*, and may take an arbitrary amount of time to transmit and process. Thus, *low-level RTS routines may access only processor-local state.* This restriction is most exacting on the context-management routines; all other traps may simply allocate a context (e.g. make a procedure call) in order to access non-local state.

	SEND(*cont*, *value*)	
value ←	PLT(*ptr*)	
	PLP(*ptr*, *value*)	
value ←	STAKE(*loc*)	*loc* a known absolute location
	MOV(*loc*, *value*)	*loc* a known absolute location

Table 1: Pseudocode Primitives

In the following presentation, we concentrate on the implementation of context management routines which are completely processor-local, which are highly optimized, and which have a high degree of thread independence. We begin by introducing the imperative local memory operations and other primitives used in the pseudocode to follow. Our first context manager is a simple free-list. We discuss various queue implementations, and then more independent, two-level schemes, including one which provides for automatic deferral of `GET-CONTEXT` requests.

2 Machine Primitives

The sequential pseudocode in this document requires some notational explanation. Identifiers are either *lc* (lowercase) or *UC* (uppercase). Uppercase identifiers either point to a known absolute location in processor-local frame memory, or begin with a T, are thread-local, and are accessed by offsets from the base of the ephemeral frame. Lowercase identifiers denote values with no particular location; we assume that these values will be stored temporarily in processor registers and spilled to frame slots if necessary. Either type of value may be modified imperatively ("$a \leftarrow b$"). Multiple assignments ("$a, b \leftarrow c, d$") are performed in parallel.

Besides integers, we will manipulate two other types of primitive data: ***continuations*** and ***pointers*** to memory. A *continuation* contains both code and frame (stack) pointers, and is roughly equivalent to a return address on a von Neumann-style architecture. SEND(*cont*, *value*) returns *value* to the continuation *cont*.

Pointers allow more operations, including comparisons to other pointers, and addition or subtraction of an integer. Although most memory requests are two-phase and do not preserve the thread, two highly-optimized transactions have been designed to allow an indirect reference to a location on the same processor without losing the current thread. The PLT operation, or ***processor-local take***, takes 2 instructions and extracts a value from a location with presence bits ***pl-present***, while PLP, ***processor-local put***, takes three instructions and stores a value in an empty location. Any other use of these operations, e.g. non-local pointers, PLT on a location which is not ***pl-present***, or PLP on a location which is not ***empty***, is an error.[1]

Synchronization between threads on a processor is performed by means of the STAKE instruction, which spins on an absolute location until there is a value present, in which case it extracts the value and returns it. Because spin-locks are expensive, we try to avoid using them where possible. A spin-lock is released by the MOV instruction.

3 A simple free-list

[1]There is still some controversy over the completeness of this choice of primitives, particularly regarding the presence-bits transitions. Note that of all the schemes presented in this paper, only the free-list scheme depends on the presence bit transitions of PLP and PLT – locations must become empty after PLT.

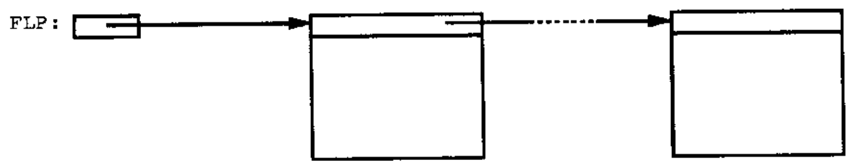

Figure 1: A free-list of local objects

```
GET-CONTEXT(cont) :
    oldp ← STAKE(FLP)   – Begin FLP critical section
    newp ← PLT(oldp)
    MOV(FLP, newp)   – End FLP critical section
    SEND(cont, oldp)


RETURN-CONTEXT(newp) :
    oldp ← STAKE(FLP)   – Begin FLP critical section
    PLP(newp, oldp)
    MOV(FLP, newp)   – End FLP critical section
```

Figure 2: Pop-Free-List and Push-Free-List

Perhaps the simplest mechanism for managing a collection of blocks is a *free-list*, a linked list of free objects [3]. In Figure 2, we implement GET-CONTEXT and RETURN-CONTEXT in pseudocode using a free-list. The *free-list pointer*, or *FLP*, points to the beginning of the list, and the first word in each block points to the next block (see Figure 1). All blocks in this scheme must be local to the processor which is managing them.

We use STAKE to synchronize on the location holding the free-list pointer, because this is a (processor) global resource, and other threads may attempt simultaneous access to the free-list. The critical section extends in both cases from the time when the STAKE instruction extracts the old pointer to the time when the MOV instruction returns the new one.

Implementation Status: Free-listing of contexts has been implemented on both the wire-wrap Monsoon prototype and the simulator for the new machine. Although the critical sections are very small, the spin-locks have been observed to be very inefficient – several simultaneous procedure calls in different threads will interfere. Furthermore, each object managed must be processor-local, and the behavior of GET-CONTEXT at the end of the list is undefined.

4 A double-buffered queue

The free-list implementation presented in the previous section suffers from one fatal flaw: because each context contains a pointer to the next, it must be local to the processor which manages it. This can be avoided by using processor-local indirection cells (Figure 3), or by using a circular queue with pointers to the beginning and the end of the active region [3]. Because of our multithreaded architecture, however, one lock must control access to both pointers in such a queue.

It is possible, however, to implement a *double-buffered* queue, in which access to the input and output buffers are controlled by different locks (Figure 4). Let one buffer, the *Input Block*, begin

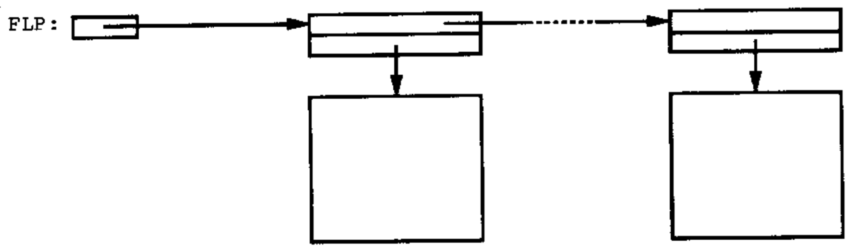

Figure 3: A free-list with local indirection cells

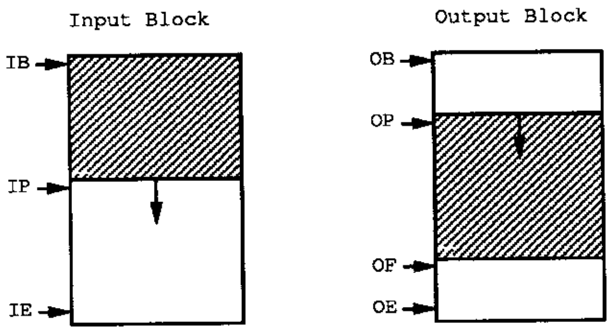

Figure 4: A double-buffered queue

empty and become gradually filled by enqueue requests, while the other buffer, the *Output Block* begins full, and gradually empties via dequeue requests. When the output block becomes empty, the buffers change roles: the input block becomes the output block, and the output block becomes the input block. The pseudocode for dequeue and enqueue appears in Figures 5 and 6.

Notational Convention: for consistency of presentation, all pointers will point to the *next available location*. For instance, the input block pointer, *IP*, points to the next empty slot (where the next object enqueued will be stored), while the output block pointer, *OP*, points to the next available object. Pointers are thus generally *incremented*, and memory grows *downward* in our diagrams.

In order to keep track of both buffers, several pointers are needed. *IB* (Input Begin) points to the beginning of the Input Block, *IE* (Input End) points to the end, and *IP* (Input Pointer) points to the next slot to fill. Similarly, *OB* (Output Begin) points to the beginning of the Output Block, *OE* points to the end, and *OP* points to the next object we will return. We also need a pointer, *OF* to (one past) the end of the valid objects in the output block.

Because of the possibility of multiple readers of the queue state, we must synchronize all readers of the state of each buffer. Since any operation on a buffer involves the pointer, this synchronization is efficiently achieved by storing *IP* in a locked cell at location *IPP* (and similarly for *OP* at location *OPP*), and using **STAKE** to ensure mutual exclusion.

Note that there are two possible errors raised by the enqueue and dequeue code. While we offer no constructive method for handling the empty queue condition at this time (see Section 6), we can avoid ever raising the queue full exception by the simple expedient of making both buffers at least as large as the number of objects managed. This also eliminates all need for pointers *IE* and *OE* to the end of each block.

Perhaps a more major disadvantage is that this mechanism will "thrash" if the average size of the queue is small, since the buffers are swapped (a relatively expensive operation) each time

```
GET-CONTEXT($cont$) :
    $op \leftarrow$ STAKE($OPP$)        – Begin OP critical section
    if $op < OF$ then
      $ctx \leftarrow$ PLT($op$)
      MOV($OPP, op + 1$)        – (Normal) End OP critical section
      SEND($cont, ctx$)
    else
      – Swap input and output buffers!
      $ip \leftarrow$ STAKE($IPP$)        – Begin IP critical section
      $IS, ip, IE, OS, op, OF, OE \leftarrow OS, OS, OE, IS, IS, ip, IE$
      MOV($IPP, ip$)        – End IP critical section
      if $op < OF$ then
        $ctx \leftarrow$ PLT($op$)
        MOV($OPP, op + 1$)        – (Alt) End OP critical section
        SEND($cont, ctx$)
      else
        PANIC("Dequeue – Queue empty")
      end if
    end if
```

Figure 5: Double-Buffered Dequeue

```
RETURN-CONTEXT($ctx$) :
    $ip \leftarrow$ STAKE($IPP$)        – Begin IP critical section
    if $ip \leq IE$ then
      PLP($ip, ctx$)
      MOV($IPP, ip + 1$)        – (Normal) End IP critical section
    else
      PANIC("Enqueue – Queue full")
    end if
```

Figure 6: Double-Buffered Enqueue

the output buffer is exhausted. Our primary objection, however, is to the existence of the two processor-global locks. The next two sections present a two-level scheme with infrequent demands on global resources, and an extension which can defer GET-CONTEXT requests.

5 A simple two-level scheme

Our dissatisfaction with the above resource management schemes stems from the fact that they all rely on processor-global spin-locks. Thus, different threads can interfere if demand for the resources is high.

Of course, we can trivially achieve our goal of context management without inter-thread interference by making eight different context pools, one for each thread. Unfortunately, this scheme can run out of contexts in one thread while there are contexts available in another. In fact, it seems clear that if we want our threads to share their resources, they must communicate at some point. Thus, we are willing to accept some degree of processor-global synchronization *provided it is not too often.*

Our first attempt at a non-interfering resource management resulted in a two-level scheme which pre-allocated a small cache of contexts to each thread, and pooled the rest. "Most of the time", the idea was, we would add and delete contexts from our local cache; only infrequently would we need to access the processor pool. Unfortunately, this resulted in a bad borderline effect: if our cache were almost empty, one allocate would empty it, so we would fill it from the processor-global collection. Then, of course, one deallocate would force us to empty the cache again. Although adding hysteresis to the system (e.g. by only copying half of the buffer on over- or underflow) might alleviate this problem, we felt that we would spend much time copying things, and thus judged this not to be a good option.

Instead, we noticed that we could copy "full" or "empty" blocks of contexts just by shuffling pointers. Thus, our next scheme resembles a double-buffered queue for each thread, but the buffers are spilled to processor-global pools instead of to the other local buffer. This is achieved by postulating two routines, *ptr* ← GET-BLOCK(*pool*) and RETURN-BLOCK(*pool*, *ptr*), which manage a processor-global collection (named *pool*) of buffers using one of the previous schemes.[2]

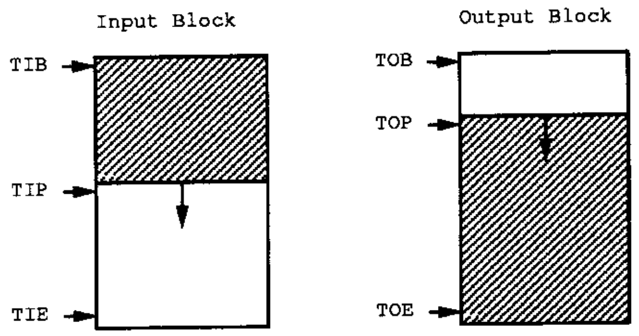

Figure 7: Thread-local double-buffered queue

Each thread manages a small cache of contexts using two buffers, in a manner similar to the double-buffered queue scheme. The Input Block gradually fills with pointers to contexts, and the Output Block is gradually emptied of contexts (see Figure 7). *TIB* points to the beginning of the

[2]As long as the number of such *pools* is small, this can be done within the addressing limits of the architecture. Of course, in an efficient implementation, these "procedures" will be inlined.

```
GET-CONTEXT($cont$) :
    $ctx \leftarrow$ PLT($TOP$); $TOP \leftarrow TOP + 1$
    SEND($cont, ctx$)
    if $TOP > TOE$ then
      – Restore invariant
      RETURN-BLOCK($the\text{-}empty\text{-}pool, TOB$)
      $b \leftarrow$ GET-BLOCK($the\text{-}full\text{-}pool$)
      $TOB, TOP, TOE \leftarrow b, b, b + len$
    end if
```

Figure 8: Two-Level Get-Context

```
RETURN-CONTEXT($ctx$) :
    PLP($TIP, ctx$); $TIP \leftarrow TIP + 1$
    if $TIP > TIE$ then
      – Restore invariant
      RETURN-BLOCK($the\text{-}full\text{-}pool, TIB$)
      $b \leftarrow$ GET-BLOCK($the\text{-}empty\text{-}pool$)
      $TIB, TIP, TIE \leftarrow b, b, b + len$
    end if
```

Figure 9: Two-Level Return-Context

Input Block, TIE points to the end, and *TIP* points to the next slot to fill. *TOB* points to the beginning of the Output Block, *TOE* points to the end, and *TOP* points to the next context we will return. For definiteness, we also enforce the invariant that $TIP \leq TIE$ and $TOP \leq TOE$, i.e. we can always get or return one more context. Note that no locks are required, because all of the queue state is *thread-local*, and that the pseudocode below assumes nothing about the block size, *len*.

Instead of *copying* contexts on over- or underflow, however, we access the appropriate global pools. In particular, on overflow we get a new empty block of contexts from the pool of empty buffers, and add our full block to the full pool. On underflow, we get a new full block of contexts from the full list, and add our empty block to the empty list. (Figures 8 and 9.)

Because we have a *cache* of contexts – and places to put returned contexts – we often avoid accessing the global queues. This avoidance can even be quantified: assuming the size of a block is 16, only 1/16 of the time does a GET-CONTEXT (or RETURN-CONTEXT) result in a global queue access. On the other hand, the most common case (15/16 of the time) requires only a few instructions.

Varying the block size is a trade-off between decreased global accesses and decreased hoarding of contexts in the local caches. In addition, we now have a significant storage overhead for the individual blocks. We need two blocks for each thread, and enough additional blocks to store all the contexts. Assuming that this processor manages 400 contexts, this amounts to at least 41 blocks, or 656 words. After the required 16 blocks, the marginal overhead is one word per context managed, which is quite reasonable.

Unfortunately, this scheme does not check for the possibility of failure within GET-BLOCK. While careful preallocation of blocks should ensure that there is always a block on the empty list whenever we need one in RETURN-CONTEXT (similar to the method discussed in the case of double-buffered enqueue, above), we cannot similarly guarantee the existence of a full block of contexts on the full list in GET-CONTEXT. In the next section, we consider the possibility of *deferring* GET-CONTEXT requests until more contexts arrive.[3]

Implementation Status: This scheme has been coded in MONASM and is currently being tested. For GET-BLOCK and RETURN-BLOCK, we use the double-buffered queue described in section 4, using the optimization described to avoid the need for end pointers.

6 Deferring Requests

The basic idea behind deferring a GET-CONTEXT request is simple: when we cannot immediately satisfy a request, we store the continuation somewhere in the hopes that a context will soon be deallocated. Unfortunately, this "steady-state" ideal is quickly contaminated by the realities of our situation. Where is this storage to come from? How is it to be allocated? Will it be thread-local or a global resource?

We have considered many schemes for implementing deferred context allocation, and are satisfied with none of them. Nevertheless, we present here one of the more promising schemes, along with indications of possible weaknesses and areas of flexibility. The fundamental choice – where to store deferred readers – is a simple extension of the local caches from the previous, two-level scheme: to each thread, we add a third block in which we cache deferred requests, and we add a global list of blocks of deferred requests.[4]

Each thread thus has additional pointers TDB, TDP, and TDE to the beginning, middle, and end of the deferred block. We add another bit, $TD?$, to each thread to indicate whether or not we are deferred. Unlike the previous scheme, however, the invariant that "there is always one more" *no longer holds*, so we must always check first before storing anything in a buffer. Preliminary investigations have indicated that this adds significantly to the critical path of GET-CONTEXT and RETURN-CONTEXT.

In each thread, we preserve the following invariant: if $TD?$ (we are deferred), then the output block is empty ($TOP = TOE$), and if we are not, then the deferred block is empty ($TDB = TDP$).[5] Furthermore, if the global list of deferred blocks is not empty, then the global list of full blocks *is* empty. Note, however, that a thread can be deferred without the global list of full blocks being empty, because the full block could be deallocated by one thread after another thread became deferred.

In addition, we explicitly allow *failure* as one of the possibilities from the procedure GET-BLOCK. In our code, GET-BLOCK is considered to return a boolean as well as a block pointer (see Figure 10).

In spite of the change in invariant, deferring GET-CONTEXT is very similar to the previous scheme: if there is one more context, return it (Figure 11). When there are no more cached contexts *and we are not already deferred*, we attempt to get more from the global collection of contexts via GET-BLOCK.

If GET-BLOCK fails, then we must defer this request. Since at this point the output block is

[3]I am indebted to Mike Beckerle for encouraging me to look for a robust storage management scheme which could also defer GET-CONTEXT requests.

[4]Note that local deferring may not be desirable. Since each thread defers and satisfies deferred requests independently ("most of the time"), if one thread deferred one request, and never handled another request, *the deferred request would never get satisfied.* We do not have a solution to this problem.

[5]Experienced hackers will see numerous ways to exploit this invariant, e.g. avoiding the need for both an output block and a deferred block or coding $TD?$ as a contorted logical case of the other pointers. We have resisted this temptation in the name of clearer exposition.

```
if $b \leftarrow GET-BLOCK(pool)$ then
  (Success, $b$ is a valid block)
else
  (Failure, $b$ is not a valid block)
end if
```

Figure 10: Interpretation of success/failure of GET-BLOCK

```
GET-CONTEXT(cont) :
    if $TOP < TOE$ then
      – Return available context
      $ctx \leftarrow$ PLT($TOP$); $TOP \leftarrow TOP + 1$
      SEND($cont, ctx$)
    else if $TD?$ then
      – Already deferred (Figure 12)
    else
      – Not deferred (yet)
      if $b \leftarrow$ GET-BLOCK(the-full-pool) then
        – We have a new (full) output block!
        RETURN-BLOCK(the-empty-pool, $TOB$)
        $TOB, TOP, TOE \leftarrow b, b, b + len$
        $ctx \leftarrow$ PLT($TOP$); $TOP \leftarrow TOP + 1$
        $SEND(cont, ctx)$
      else
        – No more full blocks – defer!
        $TD? \leftarrow True$
        PLP($TDP, cont$); $TDP \leftarrow TDP + 1$
      end if
    end if
```

Figure 11: Deferring Two-Level Get-Context

empty, we can safely change our state to *deferred*. We must also store the continuation in the defer block, which must have been empty (since we were just recently not deferred).

Of course, we could have chosen to test the state of *TD?* before testing for the availability of another context. By our previous design heuristic, this code attempts to optimize for the case where requests are not deferred at the expense of more overhead on deferred requests. Alternatively, we could try to get a full block before testing *TD?*. More experimentation will be needed to tune this code for the best performance.

Another optimization available would be to return one of our *incoming contexts* (if possible) either before attempting to get a full block or before deferring. We have not explored this option.

When we are deferred (Figure 12), we again try to obtain a full block of contexts from the global pool. If this is successful, we first satisfy all of the pending deferred requests. If there were not 16 pending deferred requests, we then also satisfy our own request, and become un-deferred (because we have emptied the deferred block and filled the input block). Otherwise, this request[6] becomes the only deferred request.[7]

When we are already deferred and there are no global full blocks of contexts, we must also defer this request. If there are already 16 deferred requests, this means swapping this full block of deferred requests out to the global list and finding another empty block in which to store this continuation. Note that this preserves the second half of the invariant stated above: if the global deferred block list is not empty, then the global full block list is empty.

Unfortunately, continuous GET-CONTEXT requests can still result in an error condition: there may not be another empty block available. This can be shown to happen under this scheme only when there are more than 16 deferred requests, although this is not likely to be of much consolation.

The code for RETURN-CONTEXT is similar to GET-CONTEXT. If there is room to store one more context, we store it. When we fill a block of contexts (and we are not deferred), we check for a full block of deferred requests before adding our block to the full pool (preserving our global invariant). When we are deferred, we satisfy our own requests.

This is another trade-off. If deferring is not common, then it may be acceptable to wait up to 16 RETURN-CONTEXT requests in this thread before one deferred GET-CONTEXT request is satisfied. If this is not acceptable, then every RETURN-CONTEXT request must first attempt to satisfy any deferred GET-CONTEXT requests, resulting in a performance penalty. (See also footnote 7.)

7 Conclusion

In this document, we presented two processor-global context management schemes (a free-list and a double-buffered queue). We then presented a two-level scheme in which each thread managed a local collection of contexts and only accessed the global collections of full and empty buffers of contexts when absolutely necessary, and extended it to allow GET-CONTEXT requests to be temporarily deferred. We expect to have performance data on these schemes soon.

[6]Obviously, any request could be chosen. Since the current one is clearly younger than any pending request, we have chosen to satisfy the older requests first.

[7]Note that there may be more than one full block available, although we only check for one. In the unlikely event that this happens, one request will be deferred until the next GET-CONTEXT request in this thread.

```
- Already deferred
if $b \leftarrow$ GET-BLOCK(the-full-pool) then
  - But lots of contexts now available!RETURN-BLOCK($the - empty - pool, TOB$)
  $TOB, TOP, TOE \leftarrow b, b, b + len$
  - Satisfy all prev deferred readers
  while $TDB < TDP$ do
    $cont' \leftarrow$ PLT($TDP$); $TDP \leftarrow TDP - 1$
    $ctx \leftarrow$ PLT($TOP$); $TOP \leftarrow TOP + 1$
    $SEND(cont', ctx)$
  end while
  - Try to satisfy this deferred reader, too
  if $TOP < TOE$ then
    - Not all gone!
    $ctx \leftarrow$ PLT($TOP$); $TOP \leftarrow TOP + 1$
    $SEND(cont, ctx)$
    $TD?$ $\leftarrow$ False
  else
    - Defer this reader - there were 16 before us
    PLP($TDP, cont$); $TDP \leftarrow TDP + 1$
  end if
else
  - Defer this reader
  if $TDP < TDE$ then
    - Room for one more
    PLP($TDP, cont$); $TDP \leftarrow TDP + 1$
  else
    - Get empty block for more defers?
    if $b \leftarrow$ GET-BLOCK(the-empty-pool) then
      - Yes: add this one to global pool
      RETURN-BLOCK(the-deferred-pool, $TDP$)
      $TDB, TDP, TDE \leftarrow b, b, b + len$
    else
      - No more empty blocks
      PANIC("Defer - Out of storage")
    end if
  end if
end if
```

Figure 12: Deferring Two-Level Get-Context (when deferred)

```
RETURN-CONTEXT($ctx$) :
  if $TIP \leq TIE$ then
    PLP($TIP, p$); $TIP \leftarrow TIP + 1$
  else if $TD?$ then
    – We have a full block, but this thread is deferred!
    – See Figure 14
  else
    – We're not deferred, but others might be
    if $tdb \leftarrow$ GET-BLOCK($the\text{-}deferred\text{-}pool$) then
      – Yes: satisfy others (Unroll this loop!)
      – For $tdp \leftarrow tdbtoTDB + len - 1$ and $TIP \leftarrow TIBtoTIB + len - 1$ do
      $i \leftarrow 0$
      $tdp \leftarrow tdb$
      $TIP \leftarrow TIB$
      while $i < len$ do
        $cont \leftarrow$ PLT($tdp$); $tdp \leftarrow tdp + 1$
        $ctx \leftarrow$ PLT($TIP$); $TIP \leftarrow TIP + 1$
        SEND($cont, ctx$)
      end while
      $TIP \leftarrow TIB$
      RETURN-BLOCK($the\text{-}empty\text{-}pool$, $tdb$)
    else
      – Nobody on global deferred list
      RETURN-BLOCK($the\text{-}full\text{-}pool$, $TIB$)
      if $b \leftarrow$ GET-BLOCK($the\text{-}empty\text{-}pool$) then
        $TIB, TIP, TIE \leftarrow b, b, b + len$
      else
        PANIC("Multiple Defers – Out of storage")
      end if
    end if
  end if
```

Figure 13: Deferring Two-Level Return-Context

```
– We have a full block, but this thread is deferred!
– swap I (full) and O (empty) blocks
$b \leftarrow TIB$
$TIB, TIP, TIE \leftarrow TOB, TOB, TOB + len$
$TOB, TOP, TOE \leftarrow b, b, b + len$
– Satisfy all prev deferred readers
while $TDB < TDP$ do
  $cont \leftarrow$ PLT($TDP$); $TDP \leftarrow TDP - 1$
  $ctx' \leftarrow$ PLT($TOP$); $TOP \leftarrow TOP + 1$
  $SEND(cont, ctx')$
end while
$TD? \leftarrow False$
  – Restore invariant: not deferred when $TDB = TDP$
```

Figure 14: Deferring Two-Level Return-Context (when deferred)

References

[1] S. A. Brobst, J. Hicks, G. M. Papadopoulos, and J. Young. Id Run-time System. Computation structures group memo, Massachusetts Institute of Technology Laboratory for Computer Science, July 1990.

[2] C. F. Joerg. Design and Implementation of a Packet Switched Routing Chip - S.M. Thesis. Technical Report 482, Massachusetts Institute of Technology Laboratory for Computer Science, May 1990.

[3] D.E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley Publishing Company, Reading, Mass., 1973.

[4] R. S. Nikhil. ID Reference Manual, Version 90.0. Computation Structures Group Memo 284, Massachusetts Institute of Technology Laboratory for Computer Science, March 1988, Revised August 1988, July 1990.

[5] G. M. Papadopoulos. Implementation of a General Purpose Dataflow Multiprocessor - Ph.D. Thesis. Technical Report 432, Massachusetts Institute of Technology Laboratory for Computer Science, August 1988.