An Abstract Description of a Monsoon-like ETS interpreter

Computation Structures Group Memo 308
June 5, 1990

Rishiyur S. Nikhil
Arvind

This report describes research done at the Laboratory for 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 the Office of Naval Research contract N00014-89-J-1988.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139
An Abstract Description of a Monsoon-like ETS interpreter
Rishiyur S. Nikhil and Arvind
MIT Laboratory for Computer Science

1 Introduction

This document is an abstract description of a Monsoon-like ETS interpreter. It is an ETS interpreter because there is a notion of a frame store in which waiting-matching can occur and in which loop constants may be stored and accessed. It is Monsoon-like because it also has a notion of “threads” and “temporary registers”. Threads are unbroken sequences of instructions, and values can be communicated down a thread via the temporary registers. For more information on Monsoon, please see [1, 2, 3]. However, it is more abstract than Monsoon itself because:

- It leaves out much detail (such as pipelining and data path widths).
- Instructions retain some orthogonality that is sacrificed in Monsoon.
- The queueing and scheduling is more flexible.
- etc.

The interpreter is described as an Id program (listed in Appendix A), making heavy use of Id’s algebraic type notation. The program uses side-effects to update the large data structures representing frame and heap memories, and thus requires some sequencing (not shown) to ensure correct operation.

The reader should first understand the functionality of the processor and heap memory modules (l-structure memory) from Appendix A, without paying any attention to the system for queueing tokens. After that, the reader may turn to Appendices B and C, which show modifications to the code, in particular the queueing system, in order to gather statistics such as parallelism profiles and instruction counts.

This document is intended to be “living” document, i.e., it will be continually modified as we discover new requirements for functionality, both in the architecture itself and in the instrumentation (statistics gathering).

2 Top-level view of an ETS system

The basic operation of the ETS system is to process tokens. An ETS system consists of an array of nodes that can send tokens to each other via an interconnection network, as shown in Figure 1. A node is identified by a node number, which is a natural number (0, 1, ..., n – 1). We will always use the lower-case identifier n for node numbers.

Each node is either a processor or a heap memory node. A processor node contains a code memory and a frame memory. It accepts tokens that are meant for processors. Processors
can send tokens to other processors and to heap nodes. A heap node contains a heap memory. A heap memory node only accepts non-processor tokens. Heap nodes can send tokens only to processors; they cannot send tokens to other heap nodes.

All communication between nodes are called remote communications. Activities entirely within a node are termed local.

A key idea: all remote communications are done with split-phase transactions. For example, if a processor needs to read a location in a heap node, it sends a token to the heap node with the read request, along with a return continuation. Later, the heap node sends a token back to the processor node carrying the requested value and the continuation. When the processor receives this token, it can execute this continuation, which processes the value appropriately. Inter-processor communication also follows this plan.

The two phases are often referred to as the request and the response, respectively. The benefits of this scheme are:

- Split-phase transactions are non-blocking, i.e., during a split-phase transaction, the processor is free to perform other activities.

- The interconnection network can be pipelined, i.e., a processor can issue multiple remote requests before receiving any responses.

- Responses may arrive out of order — the accompanying continuations serve to disambiguate them.

In general, the entire operation of an ETS system (indeed, any dataflow system) can be regarded as event-driven — the arrival of tokens triggers node activity.

2.1 Simulating the system

To simulate an ETS system, we do not need an array of processor and heap memory nodes. As we shall see in Section 8, the effect of multiple nodes can be obtained by suitable design of a queuing system for tokens in transit. Thus, our ETS interpreter is a function that simply takes:

- a single processor node containing all frames;
• a single heap memory node containing all of heap memory, and
• a primordial token

and runs the system.

We assume that the various system memories have been initialized (e.g., with code). Then, the behavior of the system is the following:

Procedure  ets_interpreter:

Create a new token queue with the primordial token

While there is a token in the token queue
  Dequeue a token from the token queue;
  Process it in a processor or heap node, as appropriate;
  This produces a (possibly empty) list of tokens for other nodes.
  These are merged into the token queue.

The "merging" of the output tokens into the token queue is very simple, for now—the output tokens are simply appended onto the token queue. However, when we later modify the code for gathering statistics, this merging will become non-trivial, because both the output tokens and the token queue will be sorted according to timing information.

3 Tokens, continuations and values

Each token consists of a continuation and a value. There are three kinds of continuations, indicating whether the token is destined for a processor's pipeline (Pcont), a processor's memory pipeline (PMcont) or a heap memory pipeline (Hcont), respectively.

A Pcont continuation contains a node number, an instruction pointer, a frame pointer and a port. The first three components are just numbers, while the port designates either a left or a right input of an instruction.

PMcont and Hcont continuations contain a node number, a memory opcode, and a memory address. Memory opcodes are explained in a later section; a memory address is just a number.

A value is either a number, a boolean, a heap address or a continuation. Void is used where the value does not matter, e.g., for signals and triggers. DList's are used for deferred continuation lists in heap (I-structure) memory.

4 Processor nodes

4.1 Threads and registers

A processor, when given a token, executes the resulting thread of tokens, as specified in the function p_exec. Processing a token can produce a direct token and a queue token. A direct
token is fed back immediately and processed, possibly producing more direct and queue tokens. In this way, a node continues to process direct tokens as long as they are produced. Such a sequence of tokens is called a thread. During the execution of a thread, the queue tokens are collected in the list out_toks. Finally, when no direct token is produced (i.e., the end of the thread), this list is returned.

Both the direct token and the queue token are optional, i.e., each time a token is processed, it may produce both a direct token and a queue token, just a direct token, just a queue token, or neither. The absence of a direct token, of course, signals the end of a thread. The simulator models optional tokens using the type (yesno token) which either takes the value “No” (when there is no token) or “Yes t” (when there is a token t). The optional direct and queue tokens are usually labelled t1 and t2, respectively, in the figures and in the code.

During the execution of a thread, instructions have access to a set of registers. This state is volatile, i.e., it does not survive across threads—it may only be used during a single thread. We model this in our simulator by creating a new register set at the beginning of a thread and discarding it at the end of the thread.

4.2 Pipelines in a processor node

A processor has two pipelines, as shown in Figure 2. One pipeline is the main processor pipeline, which accepts tokens with Pcont continuations (modeled by the function P.p.pipe). The other pipeline is used for split-phase access to frame and instruction memory, and accepts tokens with PMcont continuations (modeled by the function P.m.pipe).

![Figure 2: Processor node pipelines](image)

The main pipeline has read-access to code memory, and read/write access to frames and registers. The memory pipeline has read/write access to code and frames.
Note: We use the term "pipeline" loosely—while Monsoon processors are actually pipelined, that behavior is not visible to the compiler, nor is it visible in our ETS specification in this document.

Note: The actual Monsoon hardware has a single pipeline that serves both purposes. However, since the "memory pipeline" is only used for the second phase of split-phase transactions, it is not really visible to the programmer or compiler. Thus, we find it more convenient to think of it as a separate memory pipeline in the processor.

4.3 Overview of a processor's main pipeline

A processor's main pipeline is shown in Figure 3. The function \texttt{p-p-pipe} mirrors exactly the structure of the figure, and describes the processing of a single token. Based on the instruction pointer \texttt{ip} on the incoming token, we fetch an instruction from code memory (this is a simple array selection). The instruction consists of three opcodes, one for each of the remaining stages—frame operation, register operation, and alu-form-token operation.

![Diagram](image)

Figure 3: An ETS processor's main pipeline

In the frame-operation stage, we use the frame opcode, and the frame pointer and port and value from the incoming token to access frame memory. This may involve a read, a write or a read-modify-write. The result is either a pair of values \texttt{a} and \texttt{b}, or a bubble in the pipeline (described by the type \texttt{ab}).
A bubble indicates that no operation should be performed in the register and alu-form-token stages, and that no output tokens are produced.

Otherwise, in the register operation stage, the two values a and b may be written to registers. The two values are also either passed through or replaced by values read from the registers. Finally, the output values may be saved in two special registers for access by exception handlers. The exact behavior is specified by the register opcode.

In the alu-form-token stage, the two values a and b are combined with information from the token and the $\texttt{aluop}$ code to produce a pair—an optional direct token $t_1$ and an optional token to be queued $t_2$.

5 Instruction set and operation of a processor's main pipeline

As mentioned earlier, an instruction consists of three opcodes, one for each of the subsequent pipeline stages—frame operation, register operation and alu-form-token operation.

5.1 Frame operation codes

The frame opcode specifies what operation, if any, is to be performed against frame memory. Frame memory is an array of locations indexed by a numeric frame address. Each frame location contains presence bits and a value. The presence bits indicate whether the location is empty or full (actually, only one bit is necessary for this, but we may expand this in the future).

The frame operation produces either a bubble in the pipeline or a pair of values A and B.

Bypass specifies that no frame operation is to be done. Otherwise, we have a frame opcode, an effective address mode and a frame offset $r$. The effective address mode mode $\texttt{ea}$ specifies how to interpret $r$: as an absolute address ($0 + r$) or relative to the current frame ($f_p + r$).

The $\texttt{fr.join}$ frame operation is the normal case for binary dataflow instructions (i.e., wait-match, synchronizing using presence bits). $\texttt{fr.fetch}$ is used for reading values out of the frame, ignoring the presence bits (this is used for loop constants). $\texttt{fr.xch}$ is used to exchange the token value with the value in the frame.

Note that many of the frame operations both read and write a frame location. These must be performed atomically.

Finally, the values a and b are swapped, if necessary, based on the port from the incoming token, to ensure that a is the left value and b is the right value.

5.2 Register stage operation codes

This stage can be ignored on a first reading, i.e., it can be treated as the identity function, just passing A and B through unchanged to $A'$ and $B'$.
Register memory is implemented as an array of locations, indexed by a register number. Each location can contain a value.

The register opcode specifies what operation is to be done against the register set (temporaries). Referring to Figure 3, the A and B values can optionally be written into registers. The output A' and B' values can either be taken from A and B, respectively, or from registers. Finally, A' and B' may be optionally saved in registers (in case there is an exception in the alu-form-token stage and the exception handler needs to access A' and B'). If A' and B' are saved, they are always saved in registers XA and XB, respectively.

If A and B are a bubble, they are passed through unchanged, and the registers are unchanged. Otherwise, six things happen:

- A' is taken either from A or from a register.
- B' is taken either from B or from a register.
- A is optionally stored in a designated register $r$.
- B is optionally stored in a designated register $r$.
- A' is optionally saved in register $X_1$.
- B' is optionally saved in register $X_2$.

A' and B' are returned, along with the final state of the registers is returned.

5.3 ALU and form-token operation codes

The ALU and form-token opcodes specify how to combine $A'$, $B'$ and information from the incoming continuation, in order to produce the optional direct and the optional queue tokens.

If $A'$ and $B'$ indicate a pipeline bubble, no tokens are produced. Otherwise, we dispatch on the major category of the opcode. Each category specifies destinations (successor instructions) in various ways. Some of these destinations are implicitly direct.

A destination specifies an offset relative to the current instruction pointer $ip$, and a port.

5.3.1 Arithmetic-Logic operations

Arith ops include all the arithmetic, logic and comparison operations, specified by $aop$, for combining the values on its A and B inputs. There may be one or two destinations— the same value (the output of arithmetic operation) is sent to both destinations. The token sent to the first destination is always direct.

ALU operations may also produce exceptions (e.g., divide by zero, overflow, etc.), in which case a service call is invoked (these are discussed in Section 5.3.6).

5.3.2 Switch operator

Switch is used for conditionals. It contains two destinations. The value on its A input is sent to one of the two destinations, based on the boolean value on its B input. Thus, it produces one token, which is always direct.
5.3.3 Remote memory operators

Fetch'es and Store's are used for all the different kinds of remote fetches and stores (specified by sop) from I-structure memory, other frames, code memory, etc. The destination of a Fetch is the destination expecting the result of the fetch. The token sent to memory is never direct. In both Fetch's and Store's, there is an optional signal destination; the signal token is always direct.

For fetches, the memory token contains, in its memory continuation (MCnt or PMCnt):
- the node number from the memory address in the a input,
- the memory opcode, and
- the memory address \( \lambda \) from the a input adjusted by the offset given on the b input.

In the value part of the memory token, we place a processor continuation based on the given destination.

For stores, the memory token contains, in its memory continuation (MCnt or PMCnt):
- the node number from the memory address in the a input.
- the memory opcode, and
- the memory address \( \lambda \) from the a input.

In the value part of the memory token, we place the value from the b input.

5.3.4 Data-driven inter-frame communication

CnCnt is used to send a value to a remote frame and to simultaneously initiate a thread there. They are used for procedure calls and returns. We receive a processor continuation in the a input and a value in the b input. These are glued together into a token, adjusting the instruction pointer using adj. This token is never direct.

The CnCnt operator can also include an optional signal destination; this token is always direct.

5.3.5 Local frame fetches and stores

LFFetch'es and LFStore's are used for fetches and stores to local frame memory (on the same processor). These are always done with direct tokens. For LFStore, an optional queued signal is also produced.

Local frame fetches and stores are also performed in a split-phase manner. In the first phase, described by do_LFFetch and do_LFStore, we produce a direct token containing a memory continuation specifying the desired operation. Since they are direct, they will be processed immediately in the memory pipeline of the processor. In the case of the fetches, these will produce a response as a direct token. Thus, the entire operation is done using direct tokens, and the thread is unbroken.

The second phase of these operations will be described later, in Section 6, which describes the processor's memory pipeline.
5.3.6 Service calls

The `svc` is used for service calls. It specifies a trap number and a single destination where the result of the service call is to be sent. The trap number is used as an index into a trap vector.

Service calls are performed in two phases. The function `do_svc` depicts the first phase, which creates a direct memory token that will loop back into the processor's memory pipeline. The memory continuation on the token contains the current node number, the `Mop_trap_enter` memory opcode, and a frame address where the entry continuation of the trap handler may be found. This address is computed by adding the supervisor call number `j` to the fixed starting address of the trap vector. The value on the memory token is the continuation that expects the result of the service call.

It is assumed that prior to issuing this token, the `a` and `b` values were saved in registers `xa` and `xb`.

The second phase of the instruction is described in Section 6. Both phases of the trap sequence are performed using direct tokens, so that the thread is unbroken, so all registers are preserved. In particular, the exception handler has access to the `a` and `b` values in the `xa` and `xb` registers.

6 A processor's memory pipeline

The processor's memory pipeline is used to perform ordinary reads and writes into code and frame memory. The specification is quite straightforward. The I-fetch-1 and I-store-1 cases are used for synchronized reads and writes, where it is guaranteed that there will be no more than one deferred reader. If the I-fetch-1 arrives when the frame location is empty, the continuation is stored there and marked full. If an I-store-1 arrives at a full location, it is assumed that a continuation resides there. The value is stored and sent to the continuation.

Note that for local frame fetches and for trap entry, the result is a direct token, so that the thread remains unbroken.

7 Heap Memory Nodes

A heap memory node is an array of locations, indexed by a numeric address. Each location has presence bits and a value.

A heap memory node takes a token with a memory continuation, and produces 0 or more tokens in a list. Notice that the memory operations come in matched pairs.

Ordinary fetches and stores ignore the presence bits.

I-fetches and I-stores (I-structure operations) synchronize using the presence bits. Note that for I-fetches that arrive at a location before it is full, the continuations are retained in a list which is stored as a `DList` in the location.
The I-fetch-1 and I-store-1 operators are special cases of I-fetch and I-store where it is guaranteed that there will only be one deferred read. Thus, a deferred list is not necessary—the deferred continuation is stored directly in the slot.

The take and put operators are used to provide one-at-a-time (serialized) access to a heap location. They synchronize using the presence bits, with deferred continuations retained in a Dlist deferred list.

8 Gathering statistics

The specification in Appendix A is useful in understanding the functionality of the ETS processor and heap (I-structure) memories. However, the simulator would be more useful if it also collected some statistics during a run, such as parallelism profiles, memory-fetch profiles, dynamic instruction counts, thread lengths, frequency of processor pipeline bubbles, etc.

Some statistics are time-based, i.e., they count the number of events of some kind that occur at each point in time. These statistics are called profiles; an example is the so-called parallelism profile, which counts the number of instructions executed at each time step. Other statistics, such as dynamic instruction counts, are independent of time.

In outline, we need to modify the simulator as follows. First, we need to maintain a notion of time at the top level of the simulator, i.e., a "clock" that is incremented each time around some top-level loop. In each time step, we need to process some number of tokens. When a token is processed at a particular time, we need to know the latency for processing it, and the latency with which the tokens that it produces become available for subsequent processing. Second, we need to maintain the token queue in an order sorted according to time so that, to steal a phrase from Paul Masson, "we shall process no token before its time". Finally, we need to propagate this time information into all the components of the simulator so that, whenever we recognize an "interesting" event e, we can record it using the call:

```plaintext
record_event e t
```

where t is the current time. The "event recorder"\(^1\) can choose how to record the event—as a time plot, as a histogram, as a simple counter, as a data structure or as a real-time display, or whatever.

Please see Appendices B and C for modifications to the code of Appendix A to accomplish gathering of statistics.

\(^1\)Manufactured by Toktronix.
Acknowledgements

This work is based on numerous discussions with Jonathan Young, Jamey Hicks and Greg Papadopoulos. Jonathan Young took a first cut at a specification of an ETS machine in this style [4]. Our thanks to Steve Glim for detailed comments on a draft of the paper.

References


A The Id Code for the ETS Interpreter

The major purpose of the code in this Appendix is to specify the functionality of the processor and heap memories—the queueing system should not be taken seriously.
type node = Processor code frames | Heap_node heapmem;

typeof ets_interpreter = node -> node -> token -> void;
def ets_interpreter (Processor code frames) (Heap_node heapmem) tok =
  (tokq = init_queue tok
   in
     while not (empty? tokq) do
       (tok,tokq') = dequeue tokq;
       out_toks = if (processor_token? tok) then
         _P_exec code frames tok
         % SIDE EFFECT code,frames
       else
         M_exec heapmem tok;
         % SIDE EFFECT heapmem
       next tokq = merge_q tokq' out_toks
     end
   end;

% WARNING: Because of the side-effects in the above loop, it should be
% executed sequentially.

% -----------------------------------------------
% A SIMPLE QUEUING SYSTEM
% This queueing system is not to be taken seriously, either for its
% functionality or for its efficiency.

typesyn token_queue = list token;
def init_queue tok = tok:nil;
def empty? nil = true;
def empty? (tok:toks) = false;
def dequeue (tok:toks) = (tok,toks);
def merge_q tokq out_toks = tokq ++ (filter_toks out_toks);
def filter_toks nil = nil
  | filter_toks (No |out_toks) = (filter_toks out_toks)
  | filter_toks (Yes tok|out_toks) = tok; (filter_toks out_toks);

% -----------------------------------------------
% UTILITY FUNCTION TO FIND TYPE OF DESTINATION NODE OF A TOKEN

def processor_token? (Proc n ip fp p,v) = true
  | processor_token? (Proc n mop addr,v) = true
  | processor_token? (Proc n mop addr,v) = false;

% VALUES

type value = Void
  | Num N
  | Bool B
  | Address nodenum addr
  | Contval cont
  | List (list cont) % used for deferred lists in heap mem

% The following type is used for various functions that return an
% optional result. Such functions return
% No if there is no result, and
% (Yes x) if x is the result.

type yesno +0 = No | Yes +0;
def yes? No = false
  | yes? (Yes x) = true;

% PROCESSOR NODE

% THREAD EXECUTION

typeof P_exec = code ->
    frames ->
    token -> (list (yesno token));

def P_exec code frames tok =
    | out_toks = Nil;
    | regs = make_new_regs ;
    | in_thread = True;
    | t1 = Yes tok
    | In
    | (while (yes? t1) do
        | Yes tok = t1 ;
        | (cont,val) = tok ;
        | [(next t1, t2), next regs] =
            (case cont of
                | Ppoint n ip fp port = P_p_pipe code frames regs tok
                | #Ppoint n mop addr = (F_p_pipe code frames tok, regs) ) ;
        | next out_toks = out_toks ++ [t2:Nil] ;
    |
    | Finally out_toks ;

% PROCESSOR MAIN PIPELINE OPERATION

typeof P_p_pipe = code ->
    frames ->
    regs ->
    token -> (yesno token, yesno token), regs ;

def P_p_pipe code frames regs tok =
    | (cont,val) = tok ;
    | (cont, val) = tok ;
    | (fr_opcode, reg_opcode, alu_ft_opcode) = instr ;
    | ab = fr_stage frames fr_opcode fp port val ;
    | (regs', ab') = reg_stage regs reg_opcode ab ;
    | (t1, t2) = alu_ft_stage alu_ft_opcode ab' n fp
    | In
    | (t1, t2, regs') ;

% CODE MEMORY

typesyn code = (array instruction) ;

% INTERMEDIATE VALUES AFTER FRAME OPERATION and REGISTER STAGES

type ab = Bubble | AB value value ;
% PROCESSOR'S INSTRUCTION SET

typesyn instruction = (fr_opcode, reg_opcode, alu_ft_opcode);

% FRAME_OPCODES

type fr_opcode = Bypass | Fr_access fr_action ea_mode r ;
type fr_action = Fr_join | Fr_fetch | Fr_xch ;
type ea_mode = Absolute | Frame_rel ;
typesyn r = N ;

% REGISTER_OPCODES

typesyn reg_opcode = (tA, tB, A', B', saveA, saveB) ;
type tA = NoStore | StoreInt t ;
typesyn tB = ta ;
type A' = PassThrough | FromReg t ;
typesyn B' = A' ;
type saveA = NoSave | Save ;
typesyn saveB = saveA ;
typesyn t = N ;

% ALU and FORM-TOKEN_OPCODES

type alu_ft_opcode = Arith aop one_or_two_dests
  | Switch truedest falsedest
  | Fetch mop dest signal
  | Store mop signal
  | ChCont adj signal
  | LFetch dest signal
  | LStore dest signal
  | SVC N dest ;

% ARITHMETIC_OPCODES

type aop = Aop_Plus | Aop_Minus | Aop_Negate | Aop_Mult | Aop_Divide
  | Aop_eq | Aop_le | Aop_ge | Aop_eq | Aop_ge | Aop_div
  | Aop_and | Aop_or | Aop_not
  | Aop_left | Aop_right ;

% MEMORY_OPCODES

type mop = Mop_H_Fetch | Mop_H_I_Fetch | Mop_H_Fetch_l | Mop_H_I_Fetch_l
  | Mop_H_Take | Mop_H_Store
  | Mop_F_fetch | Mop_F_I_fetch
  | Mop_F_I_fetch_l | Mop_F_I_fetch_l
  | Mop_F_Take | Mop_F_Store
  | Mop_F_Fetch | Mop_F_Store
  | Mop_F_Fetch_l | Mop_F_Fetch_l
  | Mop_IF_fetch | Mop_IF_store
  | Mop_code_fetch | Mop_code_store
  | Mop_H_Store | Mop_H_I_Store
  | Mop_H_Fut | Mop_F_Fut

% SVC and trap entry

% DESTINATIONS

type one_or_two_dests = Oneest dest | Twoest dest dest ;
typesyn truedest = dest ;
typesyn falsedest = dest ;
type signal = NoSignal | Signal dest ;
typesyn adj = N ;
typesyn dest = (delta_FP, port) ;
typesyn delta_FP = N ;
% FRAME OPERATION STAGE

typesyn frames = array f_location ;
typesyn f_location = (frame_pbs, value) ;
type frame_pbs = F_Empty | F_Full ;
% WARNING: each call to FR_STAGE may both read and write frame memory.
% The read-and-write must be performed atomically.
% (Once way to achieve this is to run the program sequentially).

typedef fr_stage = frames ->
    fr_opcode ->
    port ->
    value ->
    ab ;

def fr_stage frames fr_opcode fp port a =
    (case fr_opcode of
    Bypass
      | (F_access fr_action a c) =
        (case a of
        Absolute x
t        Frame_Rel fp + x ) ;
    (pbs,b) = frames[addr] ;
    abvals = (case fr_opcode of
    Fr_Join = (case pbs of
    F_Full = (frames[addr] = (F_Empty,Void) In
    AB a b )
    F_Empty = (frames[addr] = (F_Full, a ) In
    Bubble ))
    Fr_fetch = AB a b
    | Fr_xch = (frames[addr] = (F_Full, a ) In
    AB Void b ) )
    In
    swapAB port abvals ) ;

def swapAB q = port -> ab -> ab ;

def swapAB Left (AB a b) = AB a b
| swapAB Right (AB a b) = AB b a ;

% REGISTER MEMORY

typesyn regs = (array value) ;
MAXREGNUM = 5 ;
XA = 4 ;
XA = 5 ;
def make_new_regs = (array 0,MAXREGNUM) |
    [1] = void || 1 <= j < MAXREGNUM |
    [1] = value |

def register_upd regs j v = (array 0,MAXREGNUM) |
    [1] = v |

% REGISTER OPERATION STAGE

typedef reg_stage = regs -> reg_opcode -> ab -> (regs, ab) ;
def reg_stage regs reg_opcode Bubble = (regs, Bubble)
| reg_stage regs reg_opcode (AB a b) =
    (ta_spec, tb_spec, a'spec, b'spec, saveA, saveB) = reg_opcode ;
    a' = (case a'spec of
    PassThrough = a.
    FromReg t = regs[t] ) ;
| b' = (case b'spec of
    PassThrough = b
    FromReg t = regs[t] ) ;
| regS1 = (case ta_spec of
    NoStore = regs
    StoreIn t = register_upd regs t a ) ;
| regS2 = (case tb_spec of
    NoStore = regS1
    StoreIn t = register_upd regS1 t b ) ;
| regS3 = (case saveA of
    NoSave = regS2
    Save = register_upd regS2 XA a' ) ;
| regS4 = (case saveB of
    NoSave = regS3
    Save = register_upd regS3 XA b' ) ;
    In
    (regS4, AB a' b' ) ;
ets.id Tue Jun 5 11:44:12 1990

---

ALU and FORM-TOKEN OPERATION

typeof alu_ft_stage = alu_ft_opcode ->
    ab ->
    nodenum ->
    ip ->
    fp -> (yesno token, yesno token);

def alu_ft_stage alu_ft_opcode Bubble n ip fp = (No, No)
    alu_ft_stage alu_ft_opcode AB a b n ip fp =
        (case alu_ft_opcode of
            Arith op d s = do_arith aop a b n ip fp ds
            Switch dT DF = do_switch a b n ip fp dT DF
            Fetch mop d sig = do_fetch mop a b n ip fp d sig
            Store mop sig = do_store mop a b n ip fp sig
            ChCont adj sig = do_change a b n ip fp adj sig
            LFetch d = do_lfetch a b n ip fp d
            LFStore sig = do_lfstore a b n ip fp sig
            SVC j d = do_svc j a b n ip fp d
        )
    )

ARITHMETIC-LOGIC

We assume a primitive ALU function that either returns a value or an
exception indicator.

typeof alu = aop -> value -> value -> val_or_exc ;
type val_or_exc = Normal value | Exception N ;

def do_arith aop a b n ip fp ds =
    alu_result = alu aop a b
    in
    { case (alu aop a b)
        Normal c = (case ds of
            Oneest d = (Yes (form_p_token n ip fp d c), No)
            Twodests d1 d2 = (Yes (form_p_token n ip fp d1 c),
                Yes (form_p_token n ip fp d2 c))
        )
        | Exception: j = do_svc j a b n ip fp (0, 0)
    )

SWITCHES

def do_switch a b n ip fp dT DF =
    { d = (case b of
        Bool True = dT
        Bool False = DF)

    t1 = Yes (form_p_token n ip fp d a)
    in
    (t1, No) ;

REMOTE FETCHES AND STORES

def do_fetch mop a b n ip fp (dip, p) sig =
    Address n' X = a;
    Num adj = b;

t1 = form_sig n ip fp sig ;
cont = {case mop of
    Mop_FFetch = PMcont
    Mop_F_I_fetch = PMcont
    Mop_F_Take_l = PMcont
    ... = PMcont ;

    t2 = Yes (form_n_token (cont n' mop (X + adj))
        (Contval (Cont n (ip=ip dip fp p)) ;
    in
    (t1, t2) ;

def do_store mop a b n ip fp sig =
    | Address n' X = a ;
    t1 = form_sig n ip fp sig ;
cont = {case mop of
    Mop_F_store = PMcont
    Mop_F_I_store = PMcont
    Mop_F_Take_l = PMcont
    ... = PMcont ;

    t2 = Yes (form_n_token (cont n' mop X)
        b) ;
    in
    (t1, t2) ;

DATA-DRIVEN INTER-FRAME COMMUNICATION

def do_change a b n ip fp adj sig =
    { Contval (Cont n' ip' fp' port') = a ;
    t1 = form_sig n ip fp sig ;
cont = {case mop of
    Mop_F_store = PMcont
    Mop_F_I_store = PMcont
    Mop_F_Take_l = PMcont
    ... = PMcont ;

    t2 = form_p_token n' ip' fp' (adj, port') b
    in
    (t1, t2) ;

LOCAL FRAME FETCHES AND STORES

def do_lfetch a b n ip fp (dip, p) =
    { Num addr = a ;
    Num adj = b ;
    t1 = Yes (form_n_token (PMcont n Mop_F_fetch (addr+adj))
        (Contval (Cont n (ip=ip dip fp p)) ;
    in
    (t1, No) ;

def do_lfstore a b n ip fp sig =
    { Num addr = a ;
    t1 = Yes (form_n_token (PMcont n Mop_F_store addr)
        b) ;
cont = {case mop of
    Mop_F_I_store = PMcont
    Mop_F_Take_l = PMcont
    ... = PMcont ;

    t2 = form_sig n ip fp sig
    in
    (t1, t2) ;

---
A PROCESSOR'S MEMORY PIPELINE

WARNING: each call to P_M_PIPE may both read and write frame memory.
- The read-and-write must be performed atomically.
- (Once way to achieve this is to run the program sequentially).

typeof P_m_pipe = code -> frames ->
token -> (yesno token, yesno token)

def P_m_pipe code frames (PMcount n mop addr, v) =
| case mop of
  | Mop_code_fetch = { t2 = (v, Num (code[addr]))
  | In
    | (No, Yes t2)
  | Mop_code_store = { Num j = v ;
    | code[addr] := j
    | In
      | (No, No)
  | Mop_F_fetch = { (pbs,b) = frames[addr] ;
    | t2 = (v, b)
    | In
      | (No, Yes t2)
  | Mop_F_store = { frames[addr] := (F_Full, v)
    | In
      | (No, No)
  | Mop_F_I_fetch_1 = { (pbs,w) = frames[addr]
    | In
      | case pbs of
        | F_Empty = { frames[addr] := (F_Full, v)
          | In
            | (No, No)
        | F_Full = (No, Yes (v,w)) }
  | Mop_F_I_store_1 = { (pbs,cont) = frames[addr] ;
    | frames[addr] := (F_Full, v)
    | In
      | case pbs of
        | F_Empty = (No, No)
        | F_Full = (No, Yes (cont, v)) }
  | Mop_F_Take_1 = { (pbs,w) = frames[addr]
    | In
      | case pbs of
        | F_Empty = { frames[addr] := (F_Full, v)
          | In
            | (No, No)
        | F_Full = { frames[addr] := (F_Empty, Void)
          | In
            | (No, Yes (v,w)) }
  | Mop_F_Put_1 = { (pbs,cont) = frames[addr]
    | In
      | case pbs of
        | F_Empty = { frames[addr] := (F_Full, v)
          | In
            | (No, No) }
HEAP MEMORY NODES

types

heapmem = array h_location;

h_location = (heap_pbs, value);

type heap_pbs = H_empty | H_deferred | H_full;

WARNING: each call to M_EXEC may both read and write heap memory.

The read-and-write must be performed atomically.

(Once way to achieve this is to run the program sequentially).

typeof M_exec = heapmem -> token -> (list (yesno token))

def M_exec (heapmem (cont n nop addr, v) =
    \case nop of
        | Hop H_fetch = heap_fetch n heapmem addr v
        | Hop H_store = heap_store n heapmem addr v
        | Hop H_I_fetch = heap_I_fetch n heapmem addr v
        | Hop H_I_store = heap_I_store n heapmem addr v
        | Hop H_I_fetch_1 = heap_I_fetch_1 n heapmem addr v
        | Hop H_I_store_1 = heap_I_store_1 n heapmem addr v
        | Hop H.Take = heap.take n heapmem addr v
        | Hop H.Put = heap.put n heapmem addr v
    \);

ORDINARY FETCHES AND STORES

def heap_fetch n heapmem addr cont =
    \case heapmem[addr] of
        | H_Empty, = heapmem[addr] := (H_Full, v)
        \in
            Nil
        \;

def heap_store n heapmem addr v =
    \case heapmem[addr] of
        | H_Empty, = heapmem[addr] := (H_Full, v)
        \in
            Nil
        \;

SYNCHRONIZED FETCHES AND STORES (I-STRUCTURE OPERATIONS)

def heap_I_fetch n heapmem addr cont =
    \case heapmem[addr] of
        | H_Empty, = heapmem[addr] := (H_Deferred, DList (cont.nil))
        \in
            Nil
        | (H_Deferred, DList cons) =
            \case heapmem[addr] of
                | H_Empty, = heapmem[addr] := (H_Deferred, DList (cont:cons))
                \in
                    Nil
                | (H_Full, v) = (Yes (cont,v))::Nil
        \;

def heap_I_store n heapmem addr v =
    \case heapmem[addr] of
        | H_Empty, = heapmem[addr] := (H_Full, v)
        \in
            Nil
        | (H_Deferred, DList cons) =
            \case heapmem[addr] of
                | H_Empty, = heapmem[addr] := (H_Full, v)
                \in
                    Nil
                | (H_Deferred, DList cons) =
                    heapmem[addr] := (H_Full, v)
                    def send_v cont = Yes (cont,v)
def heap_fetch(addr):  
    if heapmem[addr] == H_Empty:  
        heapmem[addr] = H_Full  
        return True  
    else:  
        raise Error("Multiple I-structure writes")

def heap_store(addr, v):  
    if heapmem[addr] == H_Empty:  
        heapmem[addr] = H_Full, v  
        return True  
    else:  
        raise Error("Multiple I-structure writes")

def heap_take():  
    if heapmem[addr] == H_Empty:  
        heapmem[addr] = DList(None)  
        return True  
    else:  
        raise Error("Multiple I-structure writes")

def heap_put(v):  
    if heapmem[addr] == H_Empty:  
        heapmem[addr] = H_Full, v  
        return True  
    else:  
        raise Error("Multiple I-structure writes")
B Statistics gathering: ideal mode

This appendix shows the modifications to the simulator in order to collect statistics for so-called ideal mode execution, i.e.,:

- No limit on the number of processor and memory tokens processed per time step— all available tokens are processed.
- No communication latency, i.e., a token produced by a node in one time step may be processed in the very next time step.

We maintain a global notion of time in the top-level loop in the function ets-interpreter. Each iteration of the loop constitutes one time step. The token queue in ets-interpreter represents tokens in transit from one operation to another. We maintain this queue in a sorted structure where the sorting order corresponds to the notion of time.

The token queue is modeled as a list of timestep queues. If \( t \) is the current time, the first element of the list contains tokens that can be processed no earlier than time \( t \); the second element of the list contains tokens that can be processed no earlier than \( t + 1 \), and so on.

Each timestep queue consists of a pair of token lists— processor tokens and memory tokens for that timestep.

The dequeue operation returns \( \text{ptoks} \)— all the processor tokens for the current timestep, \( \text{mtoks} \)— all the memory tokens for the current timestep, and \( \text{qs} \), a queue containing all the remaining tokens (for subsequent timesteps).

All the tokens in \( \text{ptoks} \) and \( \text{mtoks} \) are processed in the current timestep. The processor and memory pipeline functions are suitably modified so that each non-direct token that is produced is accompanied by a latency (delta time), representing the transit time for that token to its destination. These latencies are then used in merging the tokens back into the token queue, i.e., each token is inserted into the time step queue representing the earliest time that it can be processed.

The current time value is passed down into the processor and memory pipes. At various interesting points, "events" are recorded using calls of the form:

\[
\text{record_event \ kind_of_event \ current_time}
\]
stats-ideal.id   Tue Jun 5 11:45:18 1990

% ETS SYSTEM
% INSTRUMENTED FOR COMPUTING PARALLELISM PROFILES
% ONLY THE CHANGED PIECES OF CODE ARE SHOWN.
typesyn time = N;
typesyn latency = N;
typeof ets_interpreter = node -> node -> token -> void;
def ets_interpreter (Processor code frames) {Heap_node heapmem} tok =
{ q = init_queues tok;
  time = 0;
  if while not (empty? q)
do
  {ptoks,mtoks,qsl1) = dequeue q time
    q2 = {for tok < ptoks do
        out_toks = P_exec code frames tok time
        next qsl = merge_q out_toks qsl
        finally qsl1;
    }
    q3 = {for tok < mtoks do
        out_toks = M_exec heapmem tok time
        next q2 = merge_q out_toks q2
        finally q22;
    }
    next q = q3;
  next time = time + 1;
} ;

% QUEUING SYSTEM FOR IDEALIZED PARALLELISM PROFILES
% This queueing system is functionally correct. However, it is by no means
% efficient. It is written the way it is more for clarity than for speed.
typesyn queues = list timestep_queues;
typesyn timestep_queues = (list token, list token);

% Representation QUEUES: (Pq1,Mq1);(Pq2,Mq2); ...
% Top-level list is sorted by time, starting from current time.
% Each element is a pair: a list of processor tokens, a list of memory tokens.
def init_queues tok = (tok:N11,N11):N11 ;
def empty? N11 = true
  l.. empty _ = false;
typeof dequeue = queues -> time -> (list token, list token, queues) ;
def dequeue ((Pq,Mq):q) time = (Pq,Mq,q) ;
typeof merge_q = (list (yesno (token, latency))) -> queues -> queues;
def merge_q out_toks q =
  { t = 0;
in
  {for out_tok < out_toks do
    next q = insert out_tok t q;
    next t = t + 1
    finally q;
  } ;

  def insert No
t q = q ;
de insert (yes (tok,L)) t q = insertl tok (L t) 0 q ;

def insertl tok L L1 N11 = if (L < L') then
  {insert2 tok (N11,N11):N11
  else
    (N11,N11):insertl tok L (L'+1) N11
  } ;
def insertl tok L L' ((Pq,Mq):q) = if (L < L') then
  {insert2 tok (Pq,Mq):q
  else
    (Pq,Mq):insertl tok L (L'+1) q ;
  ;
def insert2 tok (Pq,Mq) = if processor_token? tok then
  {tok:Pk, Mq
  else ( Pk,tok:Mq) ;
GLOBAL THREAD EXECUTION

typedef f_exec = code ->
frames ->
token -> (list (yes:no (token, latency))) ;

def f_exec code frames tok time =
(out_toks = Nil ;
regs = make_new_regs ;
in_thread = true ;
thread_length = 0 ;
t_i = Yes tok
In
|while (yes? t_i) do
|CALL (record_event Ev_p_tot time) ;
|Yes tok = t_i ;
|(cont, val) = tok ;
|{(next t_i, t_i2, next regs) =
|{case cont of
|Pcount n ip fp port = P_p_pipe code frames regs tok time
|NPcount n mop addr = (P_m_pipe code frames tok time, regs})}
|t_i = t_i2 ;
|next out_toks = out_toks ++ (t_i2:Nil) ;
|next time = time + 1 ;
|next thread_length = thread_length + 1 ;
|Finally { CALL (record_event (Ev_thread_finished thread_length) time)
|In
|out_toks }) ;
|}
def P_p_pipe code frames regs tok time =
  (time, val) = tok;
  (cont, val) = cont;
  instr = code[ip];
  (fr_opcode, reg_opcode, alu_ft_opcode) = instr;
  ab = fr_stage frames fr_opcode fp port val time;
  (regs', ab') = reg_stage regs reg_opcode ab time;
  (t1, t2) = alu_ft_stage alu_ft_opcode ab' n ip fp time
  In ((t1, t2), regs') ;
\* FRAME OPERATION STAGE
\* The only change: TIME is extra parameter, and is used to record events
\* such as BUBBLE event.

typeof fr_stage = frames ->
  fr_opcode ->
  fp ->
  port ->
  value ->
  time -> ab ;

def fr_stage frames fr_opcode fp port a time =
  \{ case fr_opcode of
    Bypass
      \{ call \{record_event \{Ev_bypass time\}\} \In AB \ a Void\}\}
  \| (Fr_access fr_action ea r) =
    \{ addr = \{case ea of
      Absolute = r
      | Frame_Rel = fp + r\}\};
  \{ pos, b \} = frames[addr] ;
  abvals = \{ case fr_opcode of
    Fr_join = \{ case pbe of
      F_Full = \{ frames[addr] := \{F_Empty, Void\}\}
      \In AB a b \}\}
    | F_Empty = \{ call \{record_event \{Ev_bubble time\}\} ;
      frames[addr] := \{F_Full, a\}\}
    \In Bubble \}\}
  \| Fr_fetch = AB a b
  \| Fr_xch = \{ frames[addr] := \{F_Full, a\}
    \In AB Void b \}\}
\In swapAB port abvals \};

\* REGISTER MEMORY
\* ... no changes ...

\* REGISTER OPERATION STAGE
\* Only change: time is an extra parameter.

typeof reg_stage = reg -> reg_opcode -> ab -> time -> \{reg, ab\};

def reg_stage reg reg_opcode Bubble time = \{reg, Bubble\}
  \| reg_stage reg reg_opcode (AB a b) time =
    \{ (tspec, tbspec, a'spec, b'spec) = reg_opcode ;
      a' = \{ case a'spec of
        \| PassThrough = a
        \| FromReg t = \{ reg[t] \}\};
      b' = \{ case b'spec of
        \| PassThrough = b
        \| FromReg t = \{ reg[t] \}\};
      reg1 = \{ case tspec of
        \| NotSave = \{ reg \}
        \| StoreIn t = \{ register_upd reg \ t a \}\};
      reg2 = \{ case tbspec of
        \| NotSave = \{ reg1 \}
        \| StoreIn t = \{ register_upd reg2 \ t b \}\};
      reg3 = \{ case saveA of
        \| NoSave = \{ reg2 \}
        \| Save = \{ register_upd reg3 \ xa \ a' \}\};
      reg4 = \{ case saveB of
        \| NoSave = \{ reg3 \}
        \| Save = \{ register_upd reg4 \ xb \ b' \}\};
      \In \{ reg4, AB a' b' \}\};

% ALU and FORM-TOKEN OPERATION
% Only change: TIME is extra parameter, is used to record events, and
% is passed into various DO foo modules.

typeof alu_ft_stage = alu_ft_opcode ->
    ab ->
    n ->
    ip ->
    fp ->
    time -> (yeno token, yeno (token, latency));

% alu_ft_opcode Bubbles n ip fp time = (No,No)
% alu_ft_opcode (AB a b) n ip fp time =
%   [ call record_event Py_alu_ft time ]
%   In
%   { case alu_ft_opcode of
%       Arith aop ds = do_arith aop b n ip fp ds time
%       Switch dt df = do_switch a b n ip fp df dt time
%       Fetch mop d sig = do_fetch mop a b n ip fp d sig time
%       Store mop sig = do_store mop a b n ip fp d sig time
%       ChCont adj sig = do_chcont a b n ip fp adj sig time
%       IFFetch d = do_iffetch a b n ip fp d time
%       IFSig d = do_ifstore a b n ip fp d sig time
%       SVC j d = do_svc j a b n ip fp d time
%     });

% ARITHMETIC-LOGIC
% Only change: TIME is extra parameter, is used to record events.

def do_arith aop a b n ip fp ds time =
    { call record_event Py_arith time };

% alu_result = alu aop a b
% In
%   { case (alu aop a b) of
%       Normal c = [ case ds of
%         One itself d = (Yes (form_p_token n ip fp d c),)
%         Two itself d = (Yes (form_p_token n ip fp d1 c),)
%         Yes (form_p_token n ip fp d2 c, l1) ]
%     };

% Exception) = do_svc j a b n ip fp (0,0)
% );

% SWITCHES
% Only change: TIME is extra parameter, is used to record events.

def do_switch a b n ip fp dt df time =
    { call record_event Py_switch time };

    d = (case b of
        Bool True = dt
        Bool False = df);

    t1 = Yes (form_p_token n ip fp d a)

    In
    { t1, No };

% REMOTE FETCHES AND STORES
% Only change: TIME is extra parameter, is used to record events.
stats-ideal.id  Tue Jun  5 11:45:18 1990

% A PROCESSOR'S MEMORY PIPELINE
% Only change: TIME is extra parameter, is used to record events.

typecase P_m_pipe = code ->
  frames ->
  token ->
  time -> [yesno token, yesno (token, latency)] ;

def P_m_pipe code frames (PMcont n mop addr, v) time =
  case mop of
    Mop_code_fetch = 
      (t2 = (v, Num [code[addr]])
       In (No, Yes t2) )
    Mop_code_store =
      [Num j = v ;
       code[addr] := j]
       In (No, No) )
    Mop_F_fetch =
      ( (pbs, b) = frames[addr] ;
       t2 = (v, b)
       In (No, Yes t2) )
    Mop_F_store =
      ( frames[addr] := (F_Full, v)
       In (No, No) )
    Mop_F_fetch_1 =
      ( (pbs, w) = frames[addr]
       In [case pbs of
            F_Empty = [frames[addr] := (F_Full, v)
            In (No, No) ]
            F_Full = (No, Yes (v, w) ) ]
    Mop_F_store_1 =
      ( (pbs, cont) = frames[addr] ;
       frames[addr] := (F_Full, v)
       In [case pbs of
            F_Empty = (No, No)
            F_Full = (No, Yes (cont, v) ) ]
    Mop_F_Take_1 =
      ( (pbs, w) = frames[addr]
       In [case pbs of
            F_Empty = [ frames[addr] := (F_Full, v)
            In (No, No) ]
            F_Full = [ frames[addr] := (F_Empty, Void)
            In (No, Yes (v, w) ) ]
    Mop_F_Put_1 =
      ( (pbs, cont) = frames[addr]
       In [case pbs of
            F_Empty = [ frames[addr] := (F_Full, v)
            In (No, No) ]
            F_Full = [ frames[addr] := (F_Empty, Void)
            In (No, Yes (v, w) ) ]

% SERVICE CALLS (Software interrupts/traps)
% Only change: TIME is extra parameter, is used to record events.
def do_svc j a b n ip fp (dip, p) time =
  call [record_event (Ev_svc j) time]
  t1 = Yes (form_m_token (PMcont n Mop_LF_store addr)
    b) ;
  t2 = (form_sig n ip fp sig, 1)
  In (t1, t2) ;

% -----------------------------------
stats-ideal.id  Tue Jun  5 11:45:18 1990

Mop_LF_fetch = (pbs, a) = frames[addr] ;
    tl = (v, a)
    In
    (Yes tl, No)

Mop_LF_store = (frames[addr] := (F_Full, v)
    In
    (No, No)

Mop_trap_enter = (pbs, trapcont) = frames[addr] ;
    tl = (trapcont, v)
    In
    (Yes tl, No)

% HEAP MEMORY NODES
% Only change: TIME is extra parameter, is used to record events,
% and is passed down into various HEAP_foo operations.

typedef M_exec = heapmem ->
    token -> (list (yesno (token, latency))) ;

def M_exec heapmem (Mcount n mop addr, v) time =
    { CALL (record_event ev_m_loc time)
      In
      [case map of
        Mop_K_fetch = heap_fetch n heapmem addr v time
        Mop_K_store = heap_store n heapmem addr v time
        Mop_K_I_fetch = heap_fetch n heapmem addr v time
        Mop_K_I_store = heap_store n heapmem addr v time
        Mop_K_Ifetch_l = heap_fetch n heapmem addr v time
        Mop_K_Istore_l = heap_store n heapmem addr v time
        Mop_K_Take = heap_take n heapmem addr v time
        Mop_K_PUT = heap_put n heapmem addr v time
      ]
    } ;

% ORDINARY FETCHES AND STORES
% Only change: TIME is extra parameter, is used to record events.

def heap_fetch n heapmem addr cont time =
    { (pbs, w) = heapmem[addr]
      In
      (Yes {((cont, w), l)} :Nil ;)  
    } ;

def heap_store n heapmem addr v time =
    { heapmem[addr] := (F_Full, v)
      In
      No:Nil ;
    } ;

% SYNCHRONIZED FETCHES AND STORES (I-STRUCTURE OPERATIONS)
% Only change: TIME is extra parameter, is used to record events.

def heap_I_fetch n heapmem addr cont time =
    { case heapmem[addr] of
      (H_Empty, _) = { heapmem[addr] := (H_Defers, DList (cont:nil))
        In
        No:Nil ;
      | (H_Defers, DList conts) =
        { heapmem[addr] := (H_Defers, DList (cont:conts))
          In
          No:Nil ;
      | (H_Full, v) = (Yes {((cont, w), l)} :Nil ;) ;
    } ;

def heap_I_store n heapmem addr v time =
    { case heapmem[addr] of
      (H_Empty, _) = { heapmem[addr] := (H_Full, v)
        In
        No:Nil ;
      | (H_Defers, DList conts) =
        { heapmem[addr] := (H_Full, v) ;
          def mk_toks L nil = Nil
          | mk_toks L (cons) = { if (L > l) then
        } ;
    } ;
def heap_fetch n heapmem addr cont time -
| case heapmem(addr) of
| (H_Empty, _) = (heapmem(addr) := (H_Deferred, cont)
| In
| No Nil
| (H_Deferred, cont) = Error "Multiple reads"
| (H_Full, v) = (Yes [(cont, v), Nil]) Nil
|

def heap_store n heapmem addr v time -
| case heapmem(addr) of
| (H_Empty, _) = (heapmem(addr) := (H_Full, v)
| In
| No Nil
| (H_Deferred, cont) = (heapmem(addr) := (H_Full, v)
| In
| (Yes [(cont, v), Nil]) Nil
| (H_Full, w) = Error "Multiple I-structure writes"
|

def heap_take n heapmem addr cont time -
| case heapmem(addr) of
| (H_Empty, _) = (heapmem(addr) := (H_Deferred, DList (cont:nil))
| In
| No Nil
| (H_Deferred, DList cont nil) = (heapmem(addr) := (H_Deferred, DList (cont:cont))
| In
| No Nil
| (H_Full, v) = (heapmem(addr) := (H_Empty, Void)
| In
| (Yes [(cont, v), Nil]) Nil
|

def heap_put n heapmem addr v time -
| case heapmem(addr) of
| (H_Empty, _) = (heapmem(addr) := (H_Full, v)
| In
| No Nil
| (H_Deferred, DList (cont:nil)) = (heapmem(addr) := (H_Empty, Void)
| In
| (Yes [(cont, v), Nil]) Nil
| (H_Deferred, DList (cont:cont2:cont2)) = (heapmem(addr) := (H_Deferred, DList (cont2:cont2:cont2))
| In
| (Yes [(cont, v), Nil]) Nil
|
C Statistics gathering: finite-processor mode

In finite-processor mode, there are at most $NP$ processor "slots" and $NM$ memory "slots" in each time step, for some global constants $NP$ and $NM$, representing the number of processors and number of heap modules, respectively. In other words, no more than $NP$ processor tokens and no more than $NM$ memory tokens may be processed in each time step.

Thus, the only change from the ideal mode program is in the procedure dequeue. We compute $jp$ and $jm$, the number of remaining processor and memory slots in the current time step (since some of the slots may have already been used by threads that began at an earlier time step and ran through the current time). We dequeue $jp$ and $jm$ processor and memory tokens, pushing back remaining tokens from the current time slot into the next time slot. Only these tokens $jp$ and $jm$ tokens are executed in the current time slot.
The only change is in parts of the queueing system.

def dequeue Nil  time = (Nil,Nil,Nil)
| dequeue ((Pq,Mq):qs) time = 
  { np = p_slots_available time ;
    (Pq1,Pq2) = (take np Pq, drop np Pq) ;
    nm = m_slots_available time ;
    (Mq1,Mq2) = (take nm Mq, drop nm Mq) ;
    qs2 = (case qs of
      Nil = (Pq2,Mq2) :Nil
      _ = (Pq2++Pq',Mq2++Mq') :qs' ) ;
  In
  (Pq1,Mq1,qs2) ;
% STATS RECORDING

NP = ... number of processors ...
NM = ... number of heap memory modules ...

def p_slots_available time = NP - (select p_profile time) ;
def m_slots_available time = NM - (select m_profile time) ;