CS184b: Computer Architecture (Abstractions and Optimizations)

Day 17: May 14, 2003
Threaded Abstract Machine

Today

- Fine-Grained Threading
- Split-Phase
- TAM

Fine-Grained Threading

- Familiar with multiple threads of control
  - Multiple PCs
- Difference in power / weight
  - Costly to switch / associated state
  - What can do in each thread
- Power
  - Exposing parallelism
  - Hiding latency

Fine-grained Threading

- Computational model with explicit parallelism, synchronization

Split-Phase Operations

- Separate request and response side of operation
  - **Idea:** tolerate long latency operations
- Contrast with waiting on response

Canonical Example: Memory Fetch

- Conventional
  - Perform read
  - Stall waiting on reply
  - Hold processor resource waiting
- Optimizations
  - Prefetch memory
  - Then access later
- **Goal:** separate request and response

Split-Phase Memory

- Send memory fetch request
  - Have reply to **different** thread
- Next thread enabled on reply
- Go off and run rest of this thread (other threads) between request and reply

Prefetch vs. Split-Phase

- Prefetch in sequential ISA
  - Must guess delay
  - Can request before need
  - …but have to pick how many instructions to place between request and response
- With split phase
  - Not scheduled until return

Split-Phase Communication

- Also for non-rendezvous communication
  - Buffering
- Overlaps computation with communication
- Hide latency with parallelism

Threaded Abstract Machine

TAM

- Parallel Assembly Language
  - What primitives does a parallel processing node need?
- Fine-Grained Threading
- Hybrid Dataflow
- Scheduling Hierarchy

Pure Dataflow

- Every operation is dataflow enabled
- Good
  - Exposes maximum parallelism
  - Tolerant to arbitrary delays
- Bad
  - Synchronization on event costly
    - More costly than straightline code
    - Space and time
  - Exposes non-useful parallelism

Hybrid Dataflow

- Use straightline/control flow
  - When successor known
  - When more efficient
- Basic blocks (fine-grained threads)
  - Think of as coarser-grained DF objects
  - Collect up inputs
  - Run basic block like conv. RISC basic-block (known non-blocking within block)

TAM Fine-Grained Threading

- **Activation Frame** – block of memory associated with a procedure or loop body
- **Thread** – piece of straightline code that does not **block** or branch
  - single entry, single exit
  - No long/variable latency operations
  - (nanoThread? → handful of instructions)
- **Inlet** – lightweight thread for handling inputs

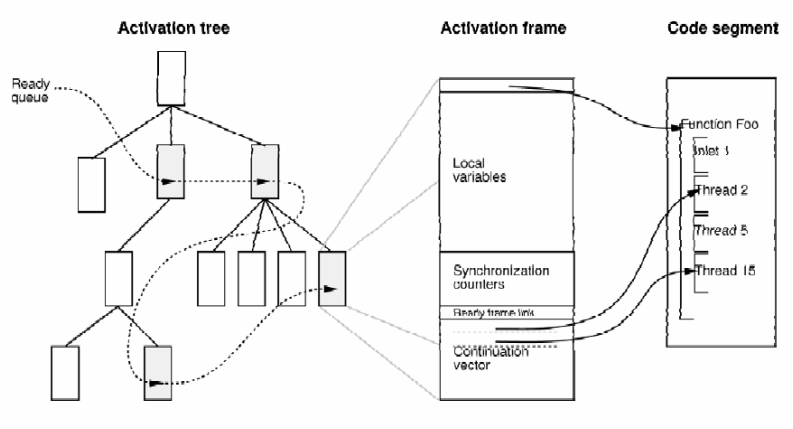
Analogies

- Activation Frame ~ Stack Frame
  - Heap allocated
- Procedure Call ~ Frame Allocation
  - Multiple allocation creates parallelism
- Thread ~ basic block
- Start/fork ~ branch
  - Multiple spawn creates local parallelism
- Switch ~ conditional branch

TL0 Model

- Threads grouped into activation frame
  - Like basic blocks into a procedure
- Activition Frame (like stack frame)
  - Variables
  - Synchronization
  - Thread stack (continuation vectors)
- Heap Storage
  - I-structures

Activation Frame



Recall Active Message Philosophy

- Get data into computation
  - No more copying / allocation
- Run to completion
  - Never block
- …reflected in TAM model
  - Definition of thread as non-blocking
  - Split phase operation
  - Inlets to integrate response into computation

Dataflow Inlet Synch

- Consider 3 input node (*e.g.* add3)
  - "inlet handler" for each incoming data
  - set presence bit on arrival
  - compute node (add3) when all present

Active Message DF Inlet Synch

- inlet message
  - node
  - inlet_handler
  - frame base
  - data_addr
  - flag_addr
  - data_pos
  - data
- Inlet
  - move data to addr
  - set appropriate flag
  - if all flags set
    - enable DF node computation

Example of Inlet Code

Add3.in:

```
*data_addr=data
*flag_addr && !(1<<data_pos)
 if *(flag_addr)==0  // was initialized 0x07
    perform_add3
else
   next←lcv.pop()
   goto next
```


Non-Idempotent Inlet Code

Add3.in:

```
*data_addr=data
*flag_addr-- // decrement synch. cntr
 if *(flag_addr)==0  // was initialized 0x03
    perform_add3
else
   next←lcv.pop()
   goto next
// this version what TAM assuming
```

TL0 Ops

- Start with RISC-like ALU Ops
- Add
  - FORK
  - SWITCH
  - STOP
  - POST
  - FALLOC
  - FFREE
  - SWAP

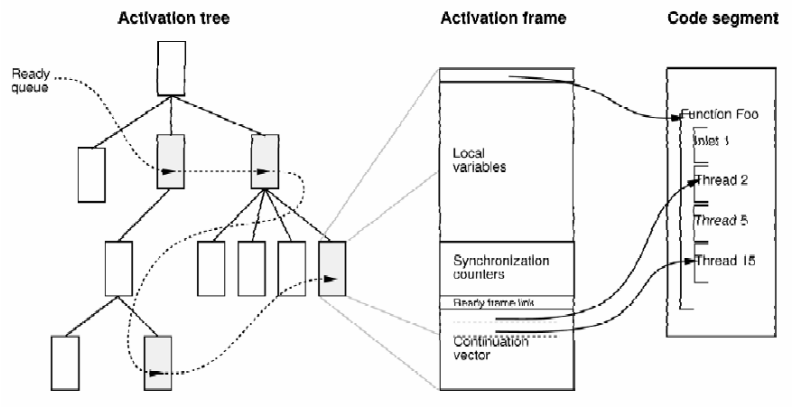
Scheduling Hierarchy

- Intra-frame
  - Related threads in same frame
  - Frame runs on single processor
  - Schedule together, exploit locality
    - contiguous alloc of frame memory→cache
    - registers
- Inter-frame
  - Only swap when exhaust work in current frame

Intra-Frame Scheduling

- Simple (local) stack of pending threads
  - LCV = Local Continuation Vector
- **FORK** places new PC on LCV stack
- **STOP** pops next PC off LCV stack
- Stack initialized with code to exit activation frame (**SWAP**)
  - Including schedule next frame
  - Save live registers

Activation Frame



POST

- **POST** – synchronize a thread
  - Decrement synchronization counter
  - Run if reaches zero

TL0/CM5 Intra-frame

- Fork on thread
  - Fall through 0 inst
  - Unsynch branch 3 inst
  - Successful synch 4 inst
  - Unsuccessful synch 8 inst
- Push thread onto LCV 3-6 inst
  - Local Continuation Vector

Fib Example

```
CBLOCK FIB.pc
FRAME_BODY RCV=3 % frame layout, RCV size is 3 threads
    islot1.i islot1.i islot2.i % argument and two results
    pfslot1.pf pfslot2.pf % frame pointers of recursive calls
    sslot0.s % synch variable for thread 6
    pfslot0.pf jslot0.j % return frame pointer and inlet
REGISTER % registers used
    breg0.b ireg0.i % boolean and integer temps
INLET 0 % recv parent frame ptr, return inlet, argument
    RECEIVE pfslot0.pf jslot0.j islot0.i
    FINIT % initialize frame (RCV,my_fp,...)
    SET_ENTER 7.t % set enter-activation thread
    SET_LEAVE 8.t % set leave-activation thread
    POST 0.t "default"
    STOP
```


Fib Cont.

```
INLET 1 % receive frame pointer of first recursive call
    RECEIVE pfslot1.pf
    POST 3.t "default"
    STOP
INLET 2 % receive result of first call
    RECEIVE islot1.i
    POST 5.t "default"
    STOP
INLET 3 % receive frame pointer of second recursive call
    RECEIVE pfslot2.pf
    POST 4.t "default"
    STOP
INLET 4 % receive result of second call
    RECEIVE islot2.i
    POST 5.t "default"
    STOP
```

Fib Threads

```
THREAD 0 % compare argument against 2
    LT breg0.b = islot0.i 2.i
    SWITCH breg0.b 1.t 2.t
    STOP
THREAD 1 % argument is <2
    MOVE ireg0.i = 1.i % result for base case
    FORK 6.t
    STOP
THREAD 2 % arg >=2, allocate frames for recursive calls
    MOVE sslot0.s = 2.s % initialize synchronization counter
    FALLOC 1.j = FIB.pc "remote" % spawn off on other processor
    FALLOC 3.j = FIB.pc "local" % keep something to do locally
    STOP
THREAD 3 % got FP of first call, send its arg
    SUB ireg0.i = islot0.i 1.i % argument for first call
    SEND pfslot1.pf[0.i] <- fp.pf 2.j ireg0.i % send it
    STOP
```


Fib Threads Cont.

```
THREAD 4 % got FP of second call, send its arg
    SUB ireg0.i = islot0.i 2.i % argument for second call
    SEND pfslot2.pf[0.i] <- fp.pf 4.j ireg0.i % send it
    STOP
THREAD 5 SYNC sslot0.s % got results from both calls (synchronize!)
    ADD ireg0.i = islot1.i islot2.i % add results
    FORK 6.t
    STOP
THREAD 6 % done!
    SEND pfslot0.pf[jslot0.j] <- ireg0.i % send result to parent
    FFREE fp.pf "default" % deallocate own frame
    SWAP "default" % swap to next activation
    STOP
THREAD 7 % enter-activation thread
    STOP % no registers to restore...
THREAD 8 % leave-activation thread
    SWAP "default" % swap to next activation
    STOP % no registers to save...
```

Fib Example

- Work through
  (fib 3)

Multiprocessor Parallelism

- Comes from frame allocations
- Runtime policy where allocate frames
  - Maybe use work stealing?
    - Idle processor goes to nearby queue looking for frames to grab and run
    - Will require some modification of TAM model to work with

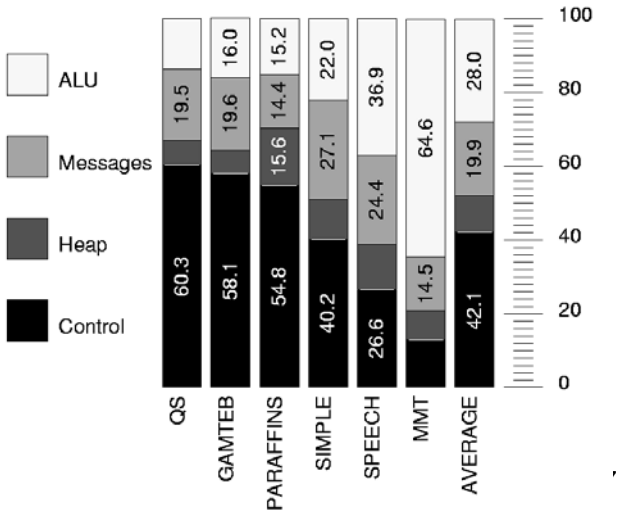
Frame Scheduling

- Inlets to non-active frames initiate pending thread stack (RCV)
  - RCV = Remote Continuation Vector
- First inlet may place frame on processor's runable frame queue
- **SWAP** instruction picks next frame branches to its enter thread

CM5 Frame Scheduling Costs

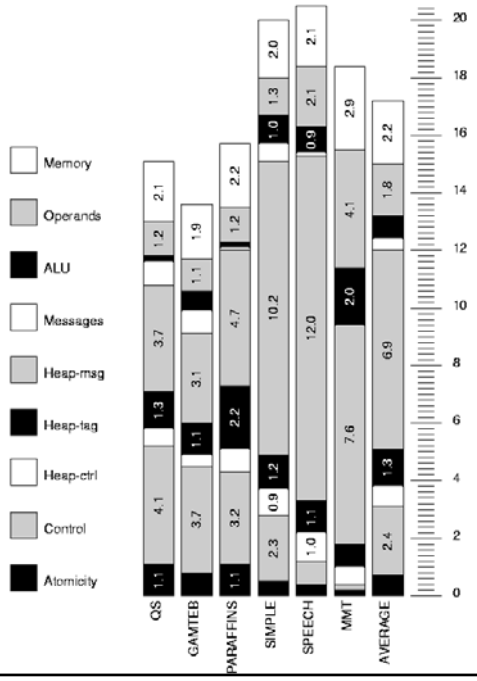
- Inlet Posts on non-running thread
  - 10-15 instructions
- Swap to next frame
  - 14 instructions
- Average thread control cost 7 cycles
  - Constitutes 15-30% TL0 instr

Instruction Mix



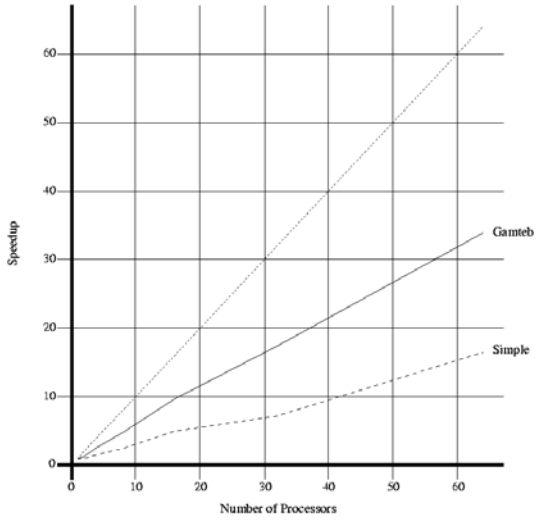
[Culler et. Al. JPDC, July 1993]

Cycle Breakdown



[Culler et. Al. JPDC, July 1993]

Speedup Example



[Culler et. Al.
JPDC, July 1993]

Thread Stats

- Thread lengths 3—17
- Threads run per “quantum” 7—530

	QS	Gamteb	Paraffins	Simple	Speech	MMT
Ave TL0 Insts. per Thread	2.6	3.2	3.1	5.3	6.3	17.6
Threads per Quanta	11.5	13.5	215.5	7.5	16.7	530.0
RCV Size when Scheduled	1.1	1.6	1.3	1.4	1.0	1.6
Threads forked during Quantum	8.8	10.2	168.4	4.1	11.7	406.6
Threads posted during Quantum	1.5	1.6	45.7	1.9	4.0	121.9
Quanta per Invocation	4.1	3.4	2.7	4.8	21.7	3.4

Table 9: Dynamic scheduling characteristics under TAM for two programs on a 64 processor CM-5

[Culler et. Al. JPDC, July 1993]

Admin/Read

- Fri. – no lecture (project meet)
- Mon. GARP
  - Two papers
    - *Computer* – more polished – read all
    - FCCM – more architecture details – skim redundant stuff and peruse details
- Wed. SCORE
  - Assume most of you already have tutorial

Big Ideas

- Balance
  - Cost of synchronization
  - Benefit of parallelism
- Hide latency with parallelism
- Decompose into primitives
  - Request vs. response … schedule separately
- Avoid constants
  - Tolerate variable delays
  - Don't hold on to resource across unknown delay op
- Exploit structure/locality
  - Communication
  - Scheduling