CS184b: Computer Architecture (Abstractions and Optimizations)

Day 23: May 23, 2005
Dataflow

Today
• Dataflow Model
• Dataflow Basics
• Examples
• Basic Architecture Requirements
• Fine-Grained Threading
• TAM (Threaded Abstract Machine)
  – Threaded assembly language

Functional
• What is a functional language?
• What is a functional routine?
• Functional
  – Like a mathematical function
  – Given same inputs, always returns same outputs
  – No state
  – No side effects

Functional:
• $F(x) = x \times x$
• (define (f x) (* x x))
• int f(int x) { return(x * x); }

Non-Functional
Non-functional:
• (define counter 0)
  (define (next-number!)
    (set! counter (+ counter 1))
    counter)
• static int counter=0;
  int increment () { return(++counter); }

Dataflow
• Model of computation
• Contrast with Control flow
Dataflow / Control Flow

Dataflow
- Program is a graph of operators
- Operator consumes tokens and produces tokens
- All operators run concurrently

Control flow
- Program is a sequence of operations
- Operator reads inputs and writes outputs into common store
- One operator runs at a time
  - Defines successor

Models
- **Programming Model:** functional with I-structures
- **Compute Model:** dataflow
- **Execution Model:** TAM

Token
- Data value with presence indication

Operator
- Takes in one or more inputs
- Computes on the inputs
- Produces a result
- Logically self-timed
  - “Fires” only when input set present
  - Signals availability of output

Dataflow Graph
- Represents
  - computation sub-blocks
  - linkage
- Abstractly
  - controlled by data presence
Dataflow Graph Example

Straight-line Code

• Easily constructed into DAG
  – Same DAG saw before
  – No need to linearize

Dataflow Graph

• Real problem is a graph

Task Has Parallelism

DF Exposes Freedom

• Exploit dynamic ordering of data arrival
• Saw aggressive control flow implementations had to exploit
  – Scoreboarding
  – OO issue

Data Dependence

• Add Two Operators
  – Switch
  – Select
Switch

Select

Constructing If-Then-Else

Looping
• For (i=0; i<Limit; i++)

Dataflow Graph
• Computation itself may construct / unfold parallelism
  – Loops
  – Procedure calls
    • Semantics: create a new subgraph
      – Start as new thread
      – ...procedures unfold as tree / dag
      – Not as a linear stack
    – ...examples shortly...

Key Element of DF Control
• Synchronization on Data Presence
• Constructs:
  – Futures (language level)
  – I-structures (data structure)
  – Full-empty bits (implementation technique)
I-Structure

- Array/object with full-empty bits on each field
- Allocated empty
- Fill in value as compute
- Strict access on empty
  - Queue requester in structure
  - Send value to requester when written and becomes full

Future

- **Future** is a promise
- An indication that a value will be **computed**
  - And a handle for getting a handle on it
- Sometimes used as program construct

DF V-Mult product in C/Java

```c
int [] vmult (int [] a, int [] b)
{
    // consistency check on a.length, b.length
    int [] res = new int[a.length];
    for (int i=0; i<res.length; i++)
        future res[i]=a[i]*b[i];
    return (res);
}
```

I-Structure V-Mult Example
I-Structure V-Mult Example

I-Structure V-Mult Example

I-Structure V-Mult Example

I-Structure V-Mult Example

I-Structure V-Mult Example

I-Structure V-Mult Example
I-Structure V-Mult Example

Fib

\[
\text{(define (fib n)}
\begin{align*}
& \quad & (\text{if} \ (< \ n \ 2) \ 1 \\
& \quad & \quad \quad \quad (+ \ (\text{future} \ (\text{fib} \ (- \ n \ 1))) \\
& \quad & \quad \quad \quad (\text{future} \ (\text{fib} \ (- \ n \ 2))))
\end{align*}
\]

int fib(int n)
{
    if (n<2)
        return(1);
    else
        return ((future)fib(n-1) + (future)fib(n-2));
}
Futures

- Safe with functional routines
  - Create dataflow
  - In functional language, can wrap futures around everything
    - Don’t need explicit future construct
    - Safe to put it anywhere
      - Anywhere compiler deems worthwhile
  - Can introduce non-determinacy with side-effecting routines
    - Not clear when operation completes

Future/Side-Effect hazard

```plaintext
(define (decrement! a b) (set! a (- a b)) a)
(print (* (future (decrement! c d))
  (future (decrement! c d))
  (future (decrement! d e))))
```

```c
int decrement (int &a, int &b)
{
  *a=*a-*b; return(*a);
}
printf("%d %d",
  (future)decrement(&c,&d),
  (future)decrement(&d,&e));
```

Architecture Mechanisms?

- Thread spawn
  - Preferably lightweight
- Full/empty bits
- Pure functional dataflow
  - May exploit common namespace
  - Not need memory coherence in pure functional \( \rightarrow \) values never change

Fine-Grained Threading
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? \rightarrow handful of instructions)
• Inlet – lightweight thread for handling inputs
Analogies

- Activation Frame ~ Stack Frame
  - Heap allocated
- Procedure Call ~ Frame Allocation
  - Multiple allocation creates parallelism
  - Recall Fib example
- 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
- Activation Frame (like stack frame)
  - Variables
  - Synchronization
  - Thread stack (continuation vectors)
- Heap Storage
  - I-structures

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
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

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/C5M 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

**Multiprocessor Parallelism**

- Comes from frame allocations
- Runtime policy decides 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 runnable 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

**Thread Stats**

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

**Instruction Mix**

![Table 9: Dynamic scheduling characteristics under TAM for two programs on a 44 processor CM-5](Culler et al. JPDC, July 1993)
Correlation

Suggests:
need ~20+ instr/thread to amortize out control

<table>
<thead>
<tr>
<th>QP</th>
<th>Overall</th>
<th>Buffer</th>
<th>Should</th>
<th>Speed</th>
<th>Snoop</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6</td>
<td>3.2</td>
<td>3.1</td>
<td>5.3</td>
<td>6.5</td>
<td>17.6</td>
</tr>
</tbody>
</table>

Threads per Queue:
- RCV time when Scheduled: 1.1, 1.2, 1.5, 1.6, 1.6, 1.8, 1.9, 1.9, 2.2, 2.7, 2.8, 2.7, 2.7, 3.6

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

Speedup Example

[Speedup Graph]

[Culler et. Al. JPDC, July 1993]

Big Ideas

- Model
- Expose Parallelism
  - Can have model that admits parallelism
  - Can have dynamic (hardware) representation with parallelism exposed
- Tolerate latency with parallelism
- Primitives
  - Thread spawn
  - Synchronization: full/empty

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