Active Messages on the CM-5: A mechanism for Integrated Communication and Computation

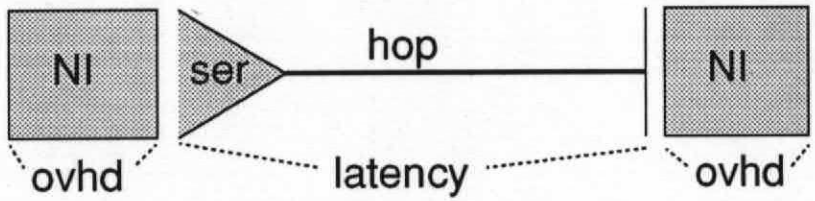
Thorsten von Eicken
David Culler

Computer Science Division
University of California at Berkeley

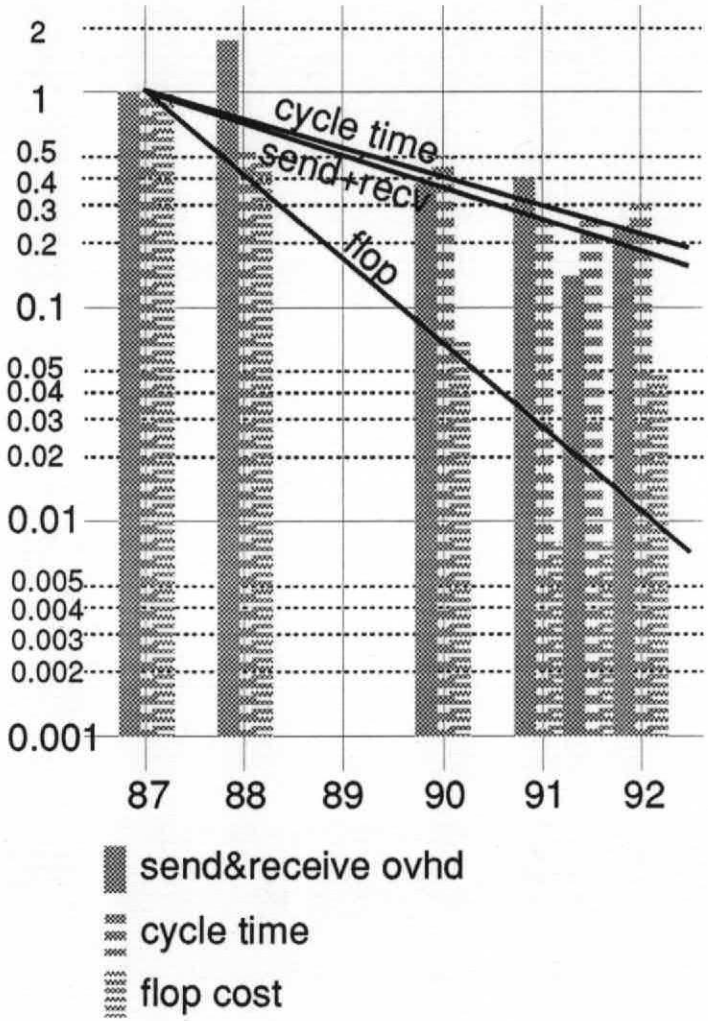
July
~~Aug~~ 30 @ TMC

Message-passing performance

- 1000 processor machines built since 1988



machine	year	send+recv overhead	flop cost	flops / snd+rcv	cycle time
nCUBE/10	87	400 µs	6.5 µs	62	100ns
IPSC/2	88	700 µs	3.1 µs	226	62ns
nCUBE/2	90	150 µs	0.45µs	333	50ns
IPSC/860	91	160 µs	0.05µs	3200	25ns
Delta	91	55 µs	0.05µs	1100	25ns
CM-5	92	95 µs	0.31µs	306	30ns



Questions:

- Would you want to build on top of send&receive?
- Is this the best the hardware can do?
- Does the hardware really "do" send&receive?
- Can one accelerate send&receive with better hardware?

Large-Scale Multiprocessor Building Block *"Sparc/MP"*

(low-cost commodity part for building multiprocessors)

Requirements:

1: general-purpose
(all programming models)

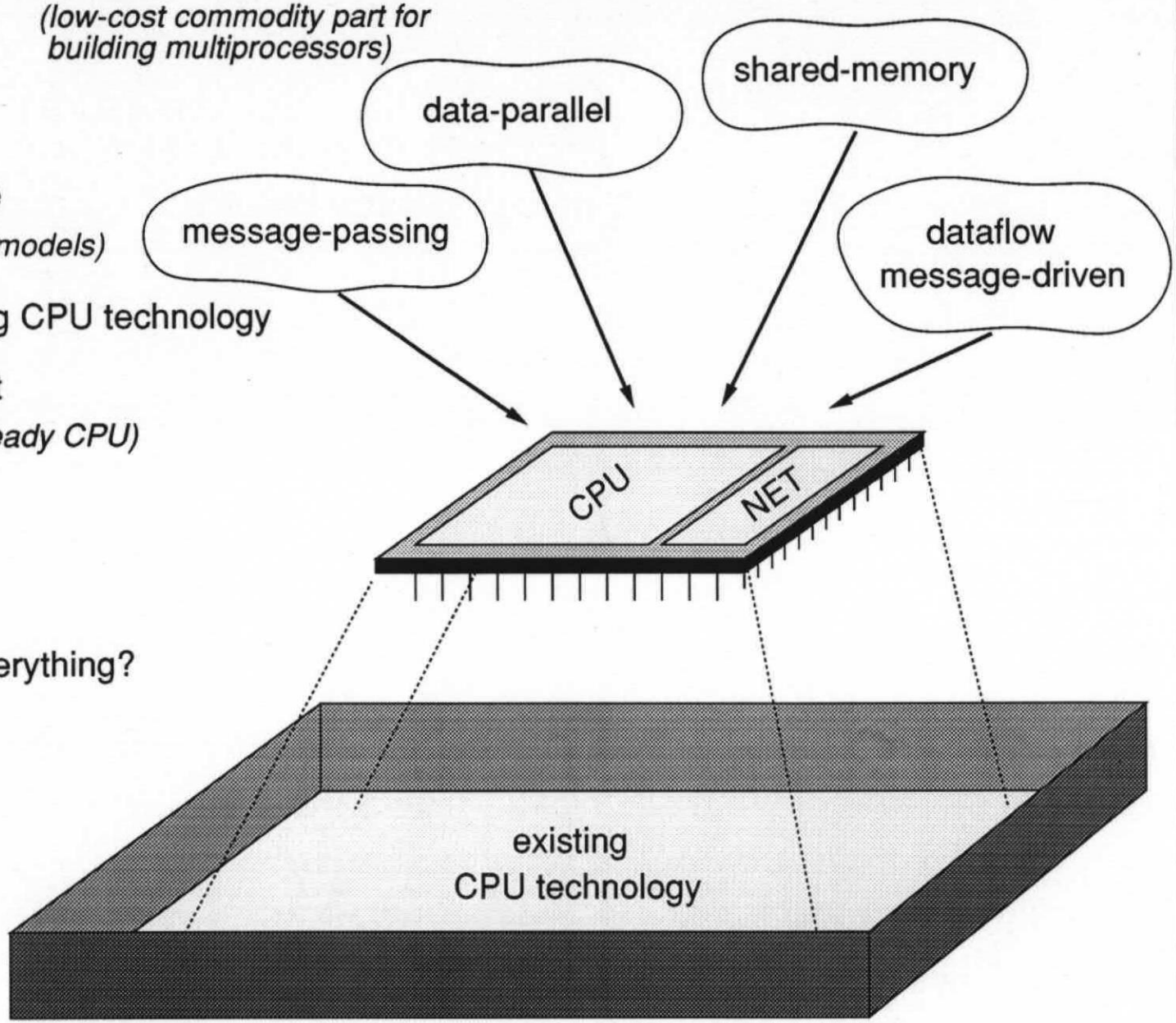
2: leverage existing CPU technology

3: incremental cost
(multiprocessor-ready CPU)

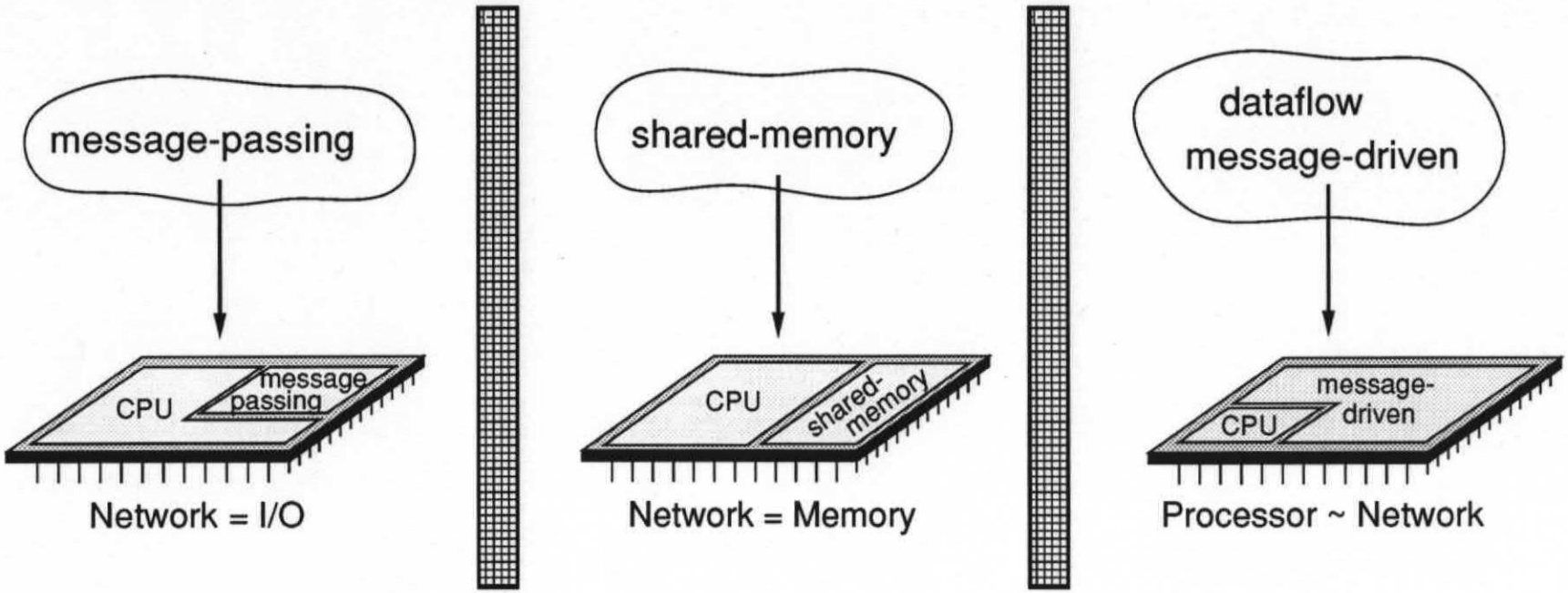
Questions:

- Is this possible?

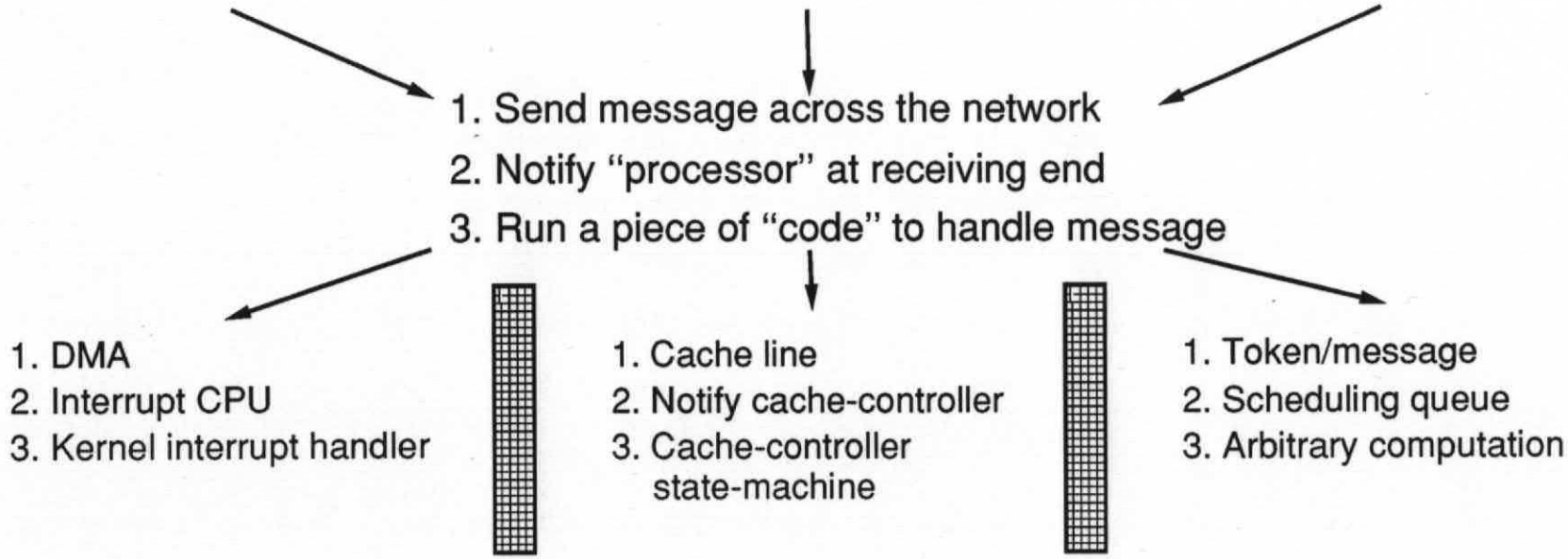
- Is it the sum of everything?



State of the art



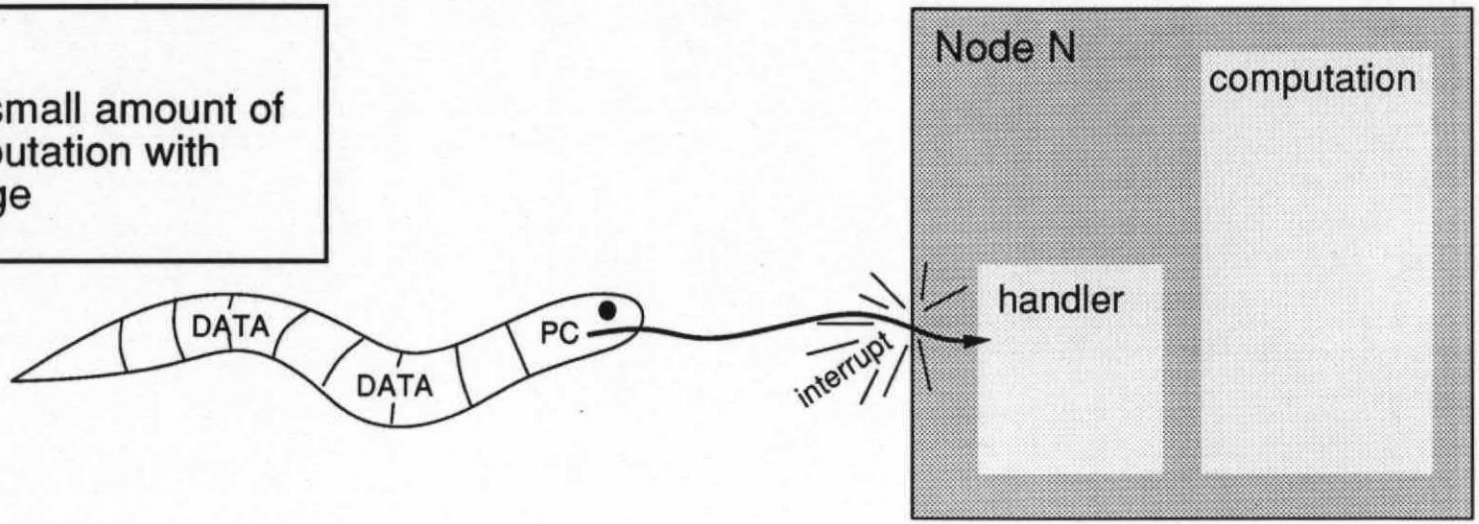
multiprocessor hardware converges



Active Messages

Idea:

associate a small amount of remote computation with each message



- The first word of each message points to the handler for the message
- On message arrival: execute handler, possibly interrupting computation
- Handler: piece of user-level code

  Get message out of network and → into computation
  and ↘ reply

Notes:

Handlers execute atomically

- → *handler is NOT arbitrary computation*
- → *cannot block/suspend*
- → *keeps scheduling simple*

Assumes same program loaded on every node

No buffering

- → *done by handler if necessary*

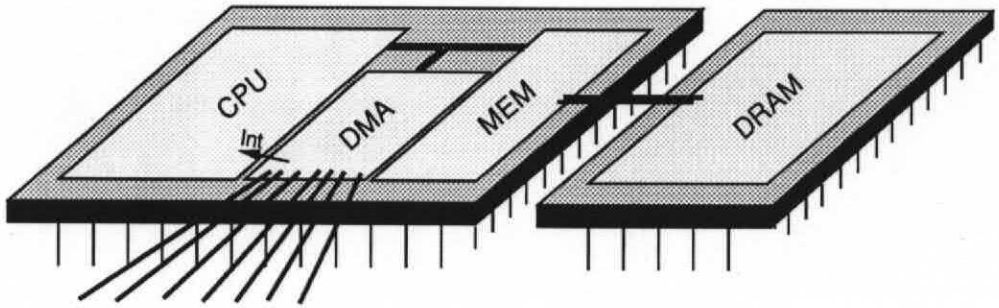
Active Messages implementation on the nCUBE/2

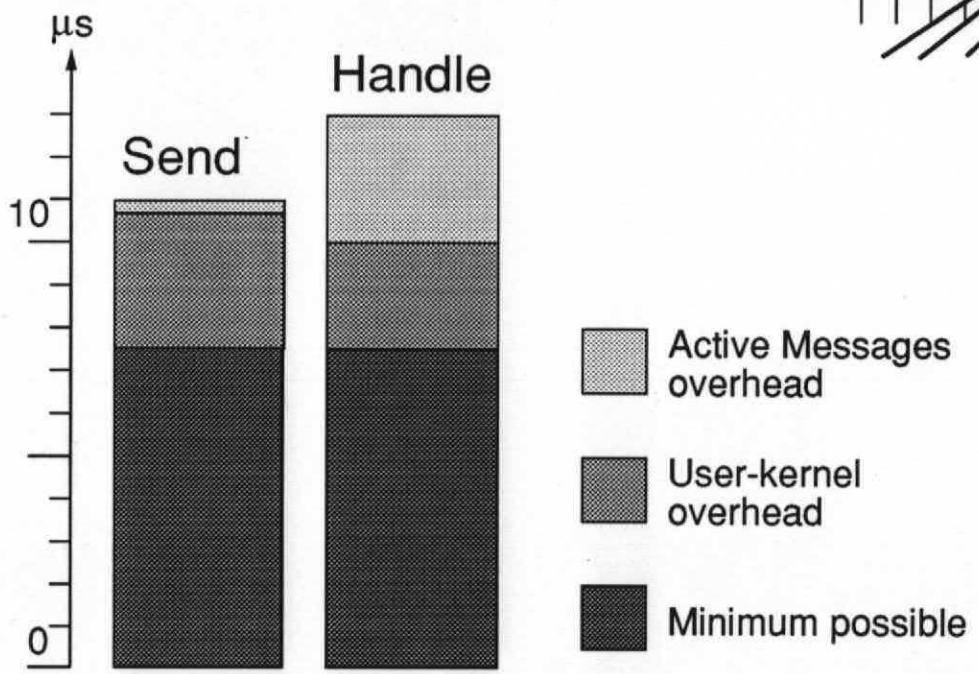
Active Messages overhead:

send: 11μs@20Mhz
21 instructions

handle: 13μs@20Mhz
34 instructions

→ 6x improvement over send&receive





Critical pieces:

Minimal buffering for DMA

Optimized kernel/user interface

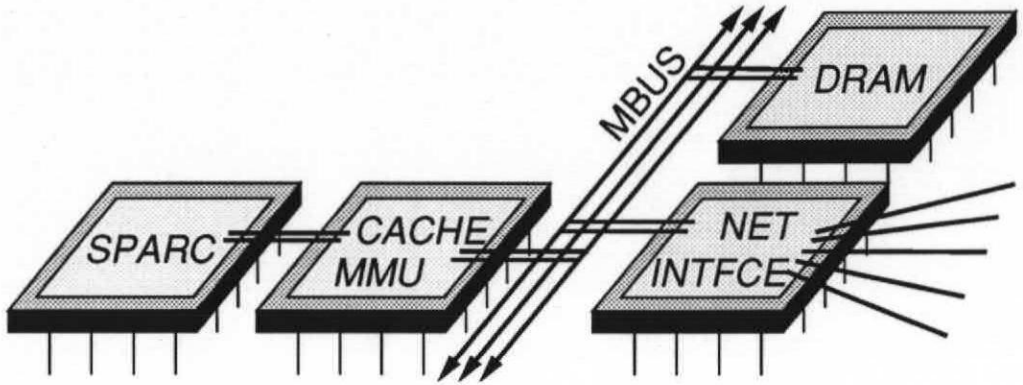
Fast user-level message handler

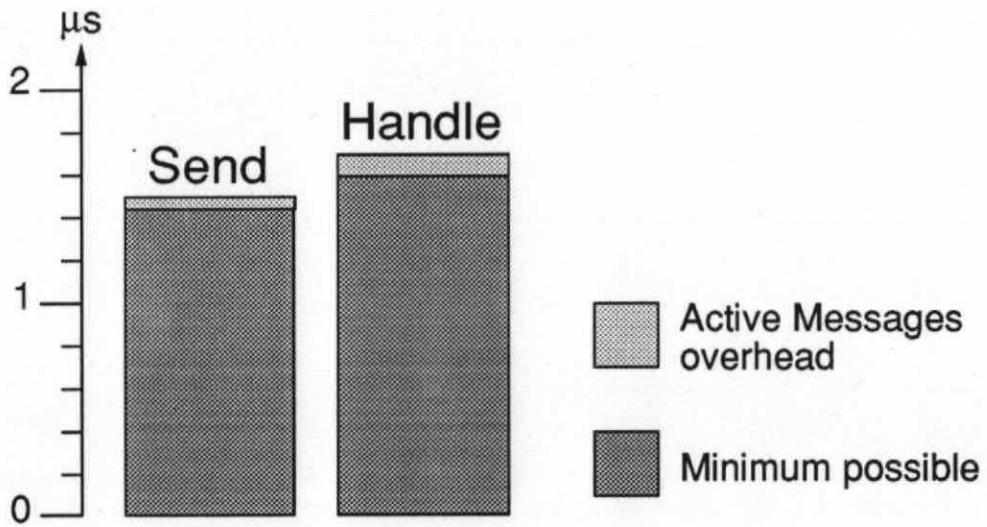
Active Messages implementation on the CM-5

Active Messages overhead:

send: 1.5 µs@33Mhz
20 instructions

handle: 1.7 µs@33Mhz
19 instructions
\+ µs trap





Critical pieces:

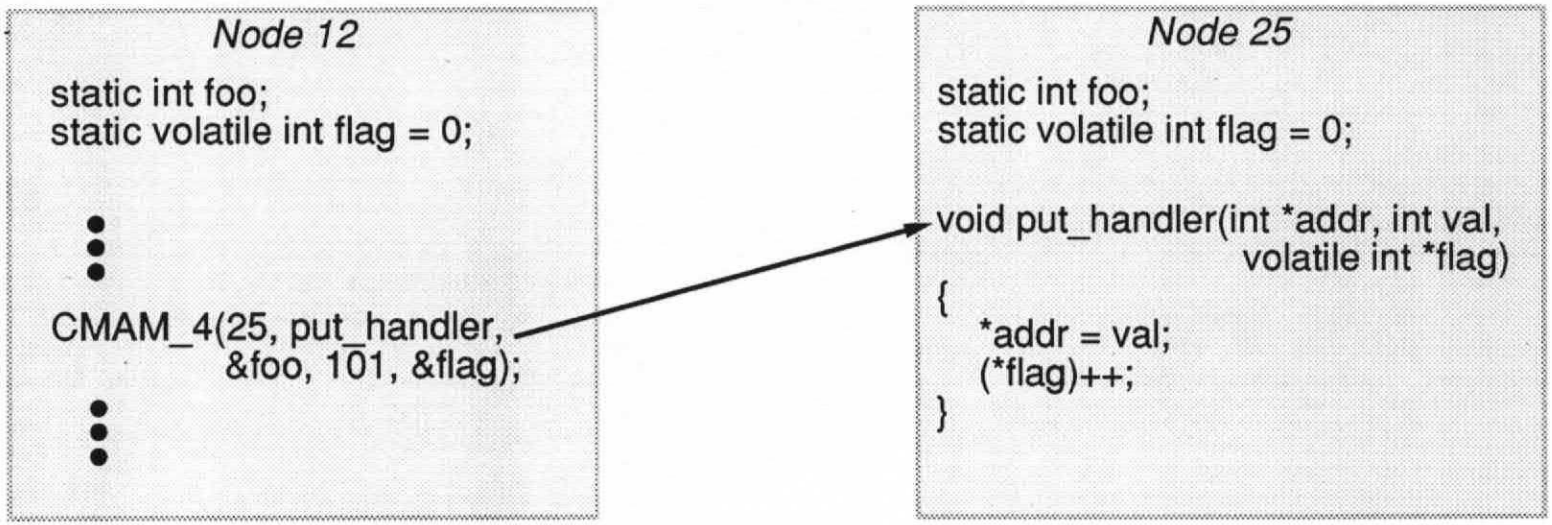
Loads/stores to network interface

Loads/stores to memory

Interrupt overhead

User-level interrupt handler

CMAM - CM-5 Active Messages (User's view)



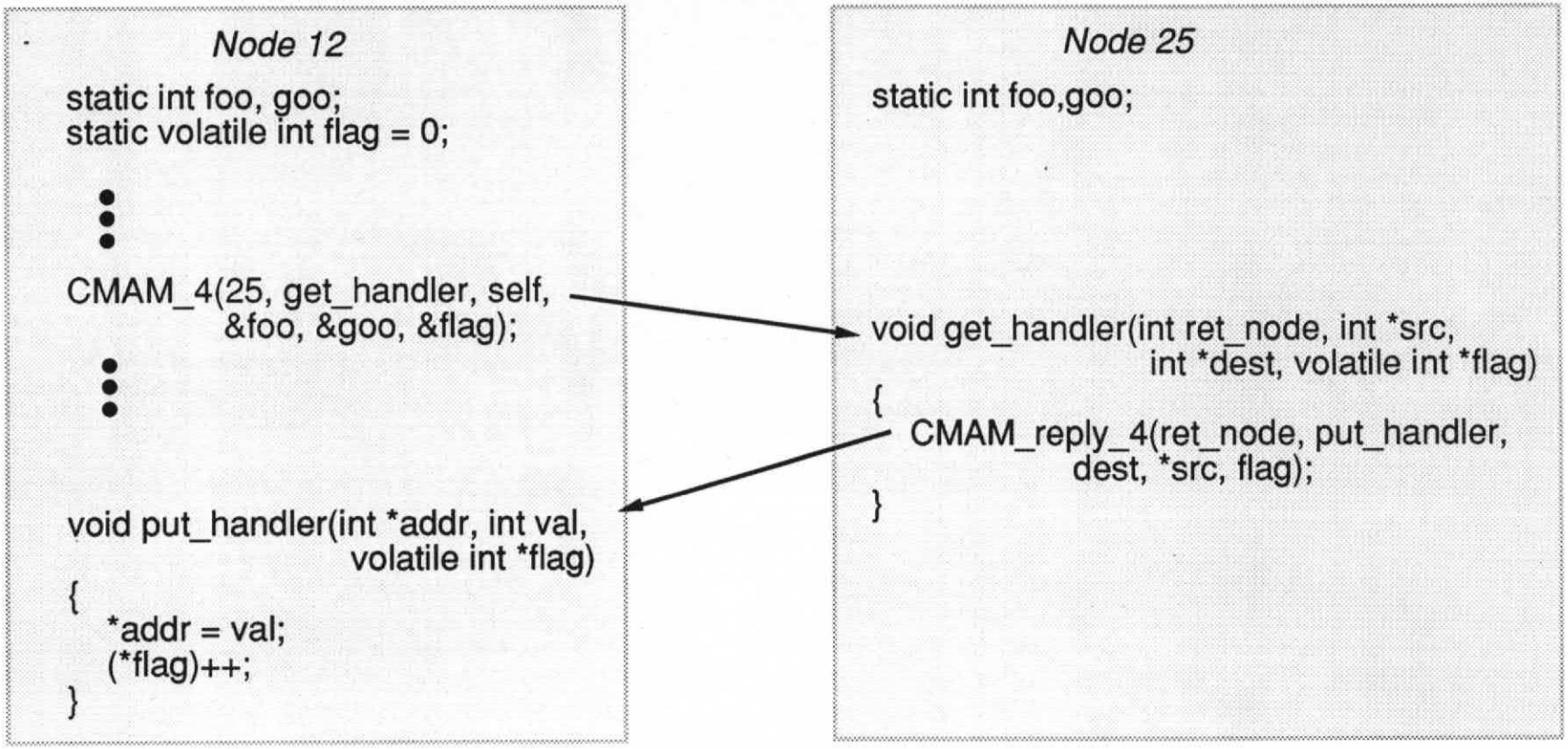
SPMD: same program (addresses) on all nodes (except host...)

Polling: every send polls the network (i.e. handles incoming messages)
explicit polling required in compute-only loops

Packet: CMAM_4( ) sends one CM-5 packet (1 word fun addr + 4 words args)
CMAM( ) sends multiple packets, 3 words of args per packet

Replies: ...

CMAM - CM-5 Active Messages (User's view)



Replies: requests use left data router — replies use right data router

requests poll left+right — replies poll right only

→ no deadlock & no buffering when network backed-up

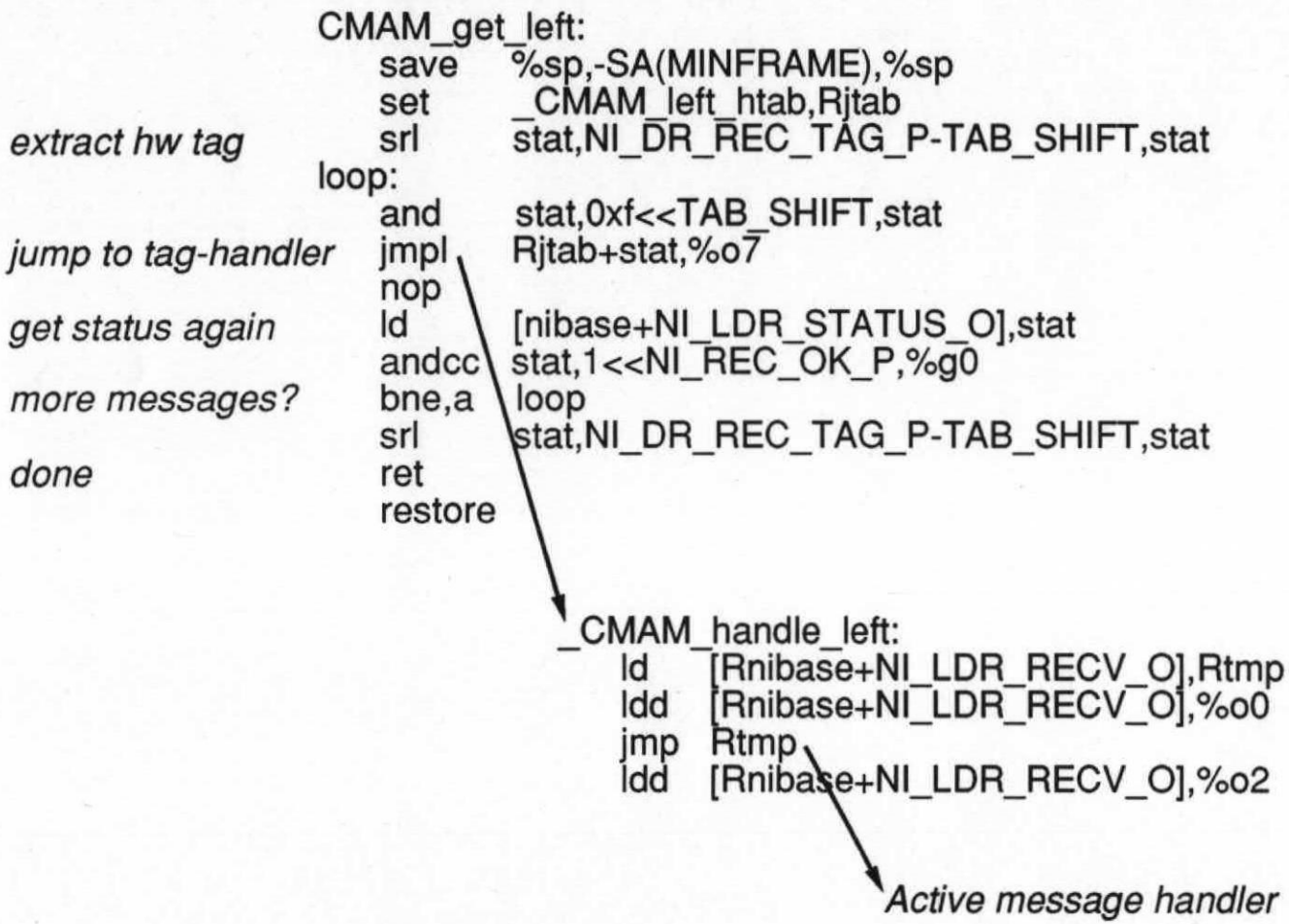
Max performance: one-way communication may use both data routers

(CMAM_both_4 being added ...)

CMAM - CM-5 Active Messages (Implementation)

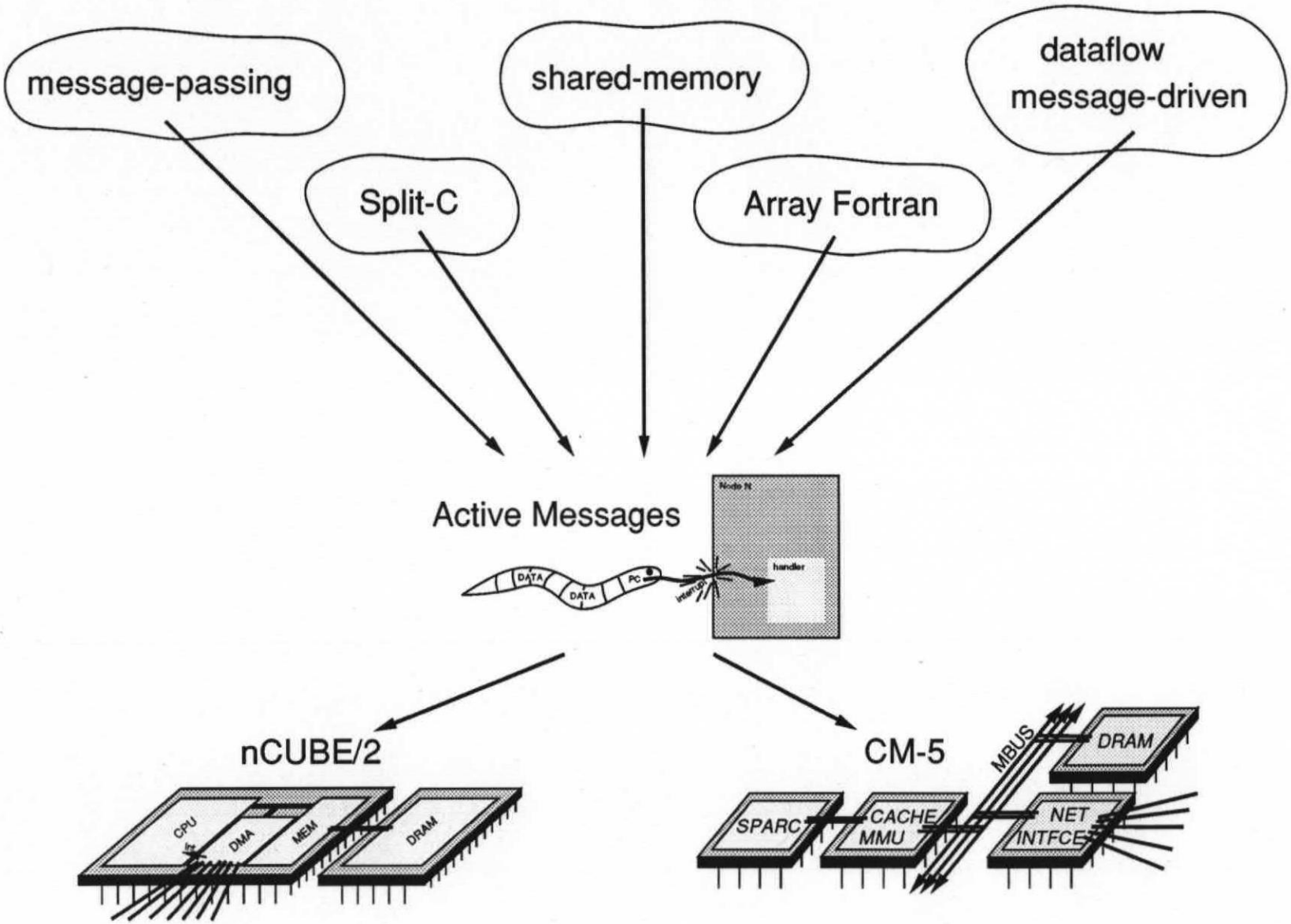
```
                    _CMAM_4:
                        save    %sp,-SA(MINFRAME),%sp
set NI addresses        sethi   %hi(_CMAM_NI_first),Rni1st
                        ld      [Rni1st+%lo(_CMAM_NI_first)],Rni1st
                  8     set     NI_BASE,Rnibase                  ! get base address of NI chip
chip bug                sethi   %hi(~0x8007ffff),Rtmp            ! mask destinations
                        andn    Rnode,Rtmp,Rnode
                        /* Loop: send, check right, check left */
                    Lout1l:
get right status        ld      [Rnibase+NI_RDR_STATUS_O],Rstat2 ! load right status register
store into left         std     Rnode,[Rni1st]                   ! push message
                        std     Rout1,[Rnibase+NI_LDR_SEND_O]
                        std     Rout3,[Rnibase+NI_LDR_SEND_O]
check right receive     andcc   Rstat2,1<<NI_REC_OK_P,%g0        ! message received on right?
                        bne     Lout5l                             ! yes
load left status  35    ld      [Rnibase+NI_LDR_STATUS_O],Rstat
check left sent         andcc   Rstat,1<<NI_SEND_OK_P,%g0        ! message sent?
                        be      Lout4l                             ! nope
check left receive      andcc   Rstat,1<<NI_REC_OK_P,%g0         ! message received on left?
                        be      Lout2l                             ! nope
receive on left         sethi   %hi(_CMAM_left_htab),Rjtab
                        or      Rjtab,%lo(_CMAM_left_htab),Rjtab
                        SERVICE_LEFT(0,Rnibase,Rstat)
                    Lout2l:
done                3   ret
                        restore

check left, resend  Lout4l:

receive right,      Lout5l:
  check left
```

CMAM - CM-5 Active Messages (Implementation)



- Thin "veneer" over hardware
- Possible improvement: "send_and_receive" loop, inline code generation
  interrupts

Active Messages: a communication substrate



Shared-memory issues

- Global addresses

  64 bits: <node, local addr>

- Asynchronous remote service

- Memory consistency

  Acknowledge writes

  Sequential consistency $\simeq$ one outstanding read/write

  g_read: wait for previous read/write to complete
  send read request
  wait for reply (+ value)

  g_write: wait for outstanding read/write to complete
  send write request (+value)

  Weak consistency $\simeq$ more than one outstanding read/write

  g_weak_write: send write request (+value)

- Caveats

  No local/remote check

  No data replication

Shared-memory implementation

```
volatile int sm_count = 0;
volatile int read_buf;
```

Global Read

```
inline void sync()
{ while(sm_count) CMAM_poll(); }

int g_read_i(int node, int *addr)
{ sync();
  sm_count++;
  CMAM_4(node, read_i_handler, self, addr);
  sync();
  return read_buf;
}

void read_i_handler(int ret_node, int *addr)
{ CMAM_reply_4(ret_node,
               read_i_reply_handler, *addr);
}

void read_i_reply_handler(int data)
{ read_buf = data; sm_count--;
}
```

Global Write

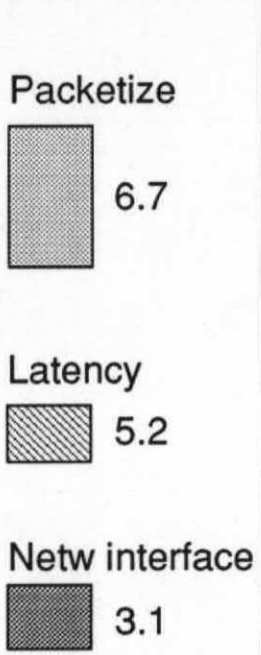
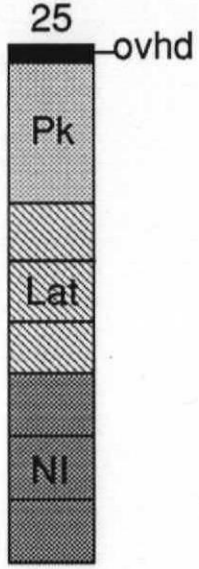
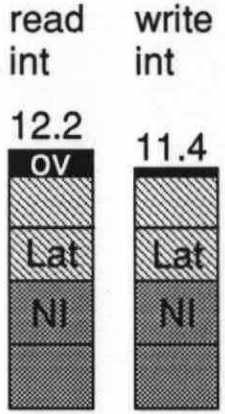
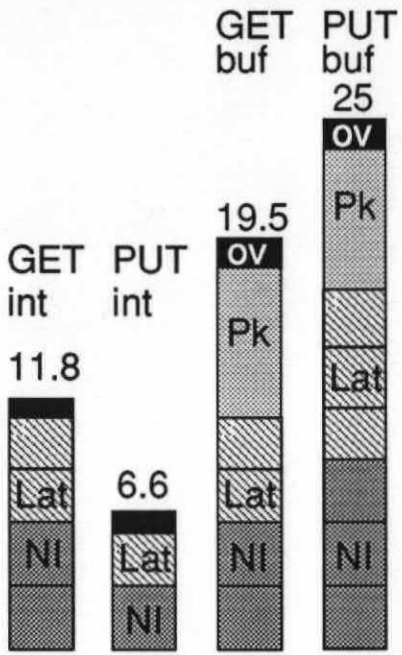
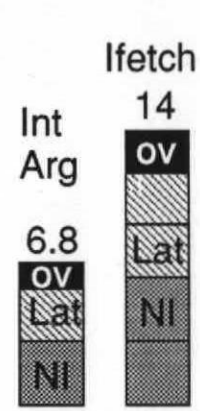
```
void g_write_i(int node, int *addr, int data)
{ sync();
  sm_count++;
  CMAM_4(node, write_i_handler,
         self, addr, data);
}

void write_i_handler(int ret_node, int *addr, int data)
{ *addr = data;
  CMAM_reply_4(ret_node, write_ack_handler);
}

void write_ack_handler() { sm_count--; }
```

Shared-memory implementation

"Weak consistency" models

```
void g_weak_write_i(int node, int *addr, int data)
{ sm_count++;
  CMAM_4(node, write_i_handler, self, addr, data);
}


int g_prefetch_i(int node, int *addr)
{ sync();
  sm_count++;
  CMAM_4(node, read_i_handler, self, addr);
}

int g_accept_i(int node, int *addr)
{ sync();
  return read_buf;
}
```

Programming models on top of Active Messages (CM-5)

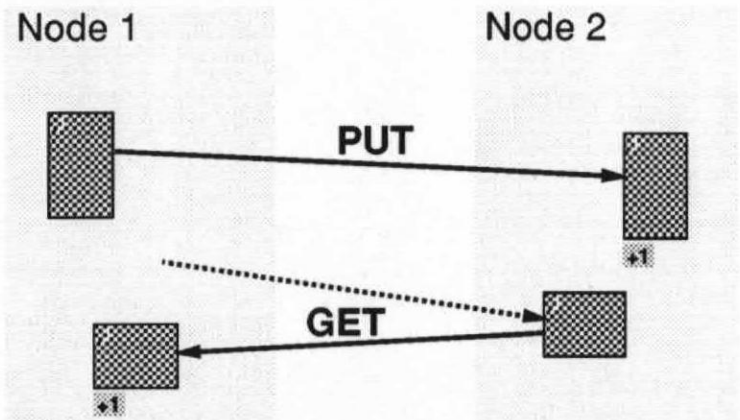
Programming model	Active Message	Send&receive (synchronous)	Shared-memory	Split-C split-phase remote access	Id-90 (TAM)
CM-5 time(μs)					
Comparison		CMMD: 95us	DASH read=3us write=3us		Monsoon Arg:1.8us IFetch:2.8us
Notes		cost: 3-way handshake asynch: malloc=15us	caveats: no local/rem check no data replication	partly compiler generated	compiler generates handlers I-structures

Split-C

Blocking read/write:

```
int *global gptr;
*gptr = ...;
... = *gptr;

struct foo *global gptr;
*gptr = ...;
... = *gptr;
```

Non-blocking put/get:



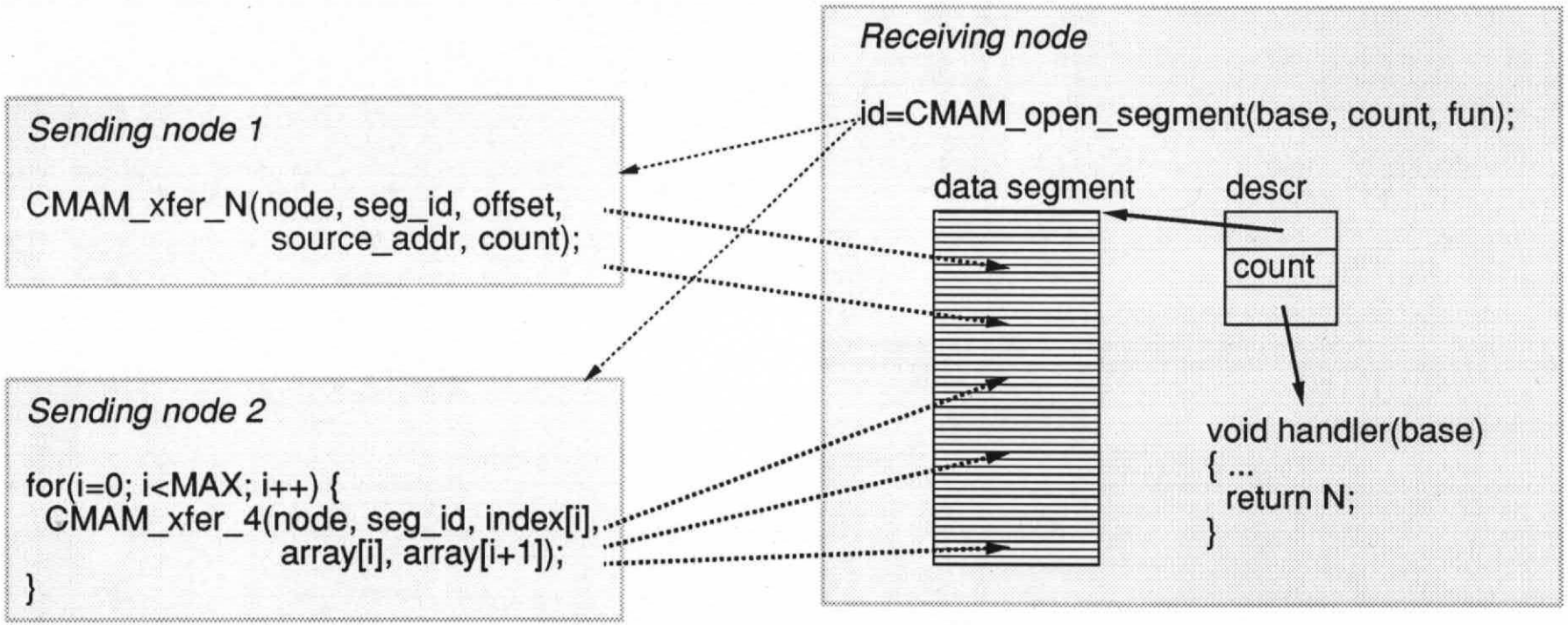
Performance:

CM-5:	*GET*	*11us*	*<= 16 bytes*
		19us	*+ 0.1us/bytes*
	PUT	*7us*	*<= 16 bytes*
		25us	*+ 0.1us/bytes*
nCUBE:	*PUT*	*26us*	*+ 0.5us/bytes*
	GET	*52us*	*+ 0.5us/bytes*

CMAM extension: segments for block transfer

Motivation — data transfer at "max" bandwidth

- → 4 data words "payload" per 5-word packet
- → 1 word for "transfer id" and "sequence number"



Get block using CMAM segments

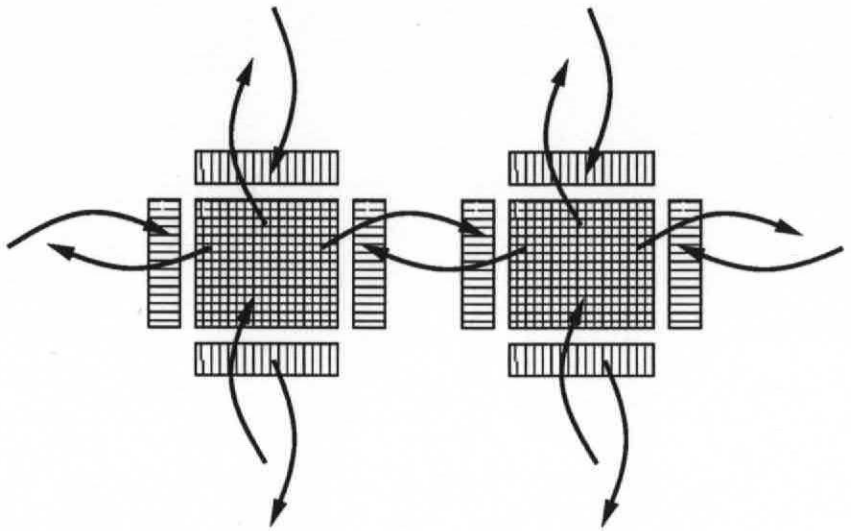
```
void get_block(int node, void *src, void *dst, int bytes, volatile int *flag)
{
  int seg_id = CMAM_open_segment(dst, bytes, end_get, flag);
  CMAM_4(node, get_block_handler, self, seg_id, src, bytes);
}

void get_block_handler(int ret_node, int seg_id, void *src, void *bytes)
{
  CMAM_xfer_N(node, seg_id, 0, src, bytes);
}

void get_end(void *base, int *flag)
{
  (*flag)++;
  return 0;
}


volatile int flag = 0;
struct node { ... } *src, dst;
...
get_block(25, src, &dest_array, sizeof(struct node), &flag);
...
CMAM_wait(&flag, 1);
...
```

Grid communication using CMAM segments

Each node opens 1 segment and passes the seg_ID to all 4 neighbors.

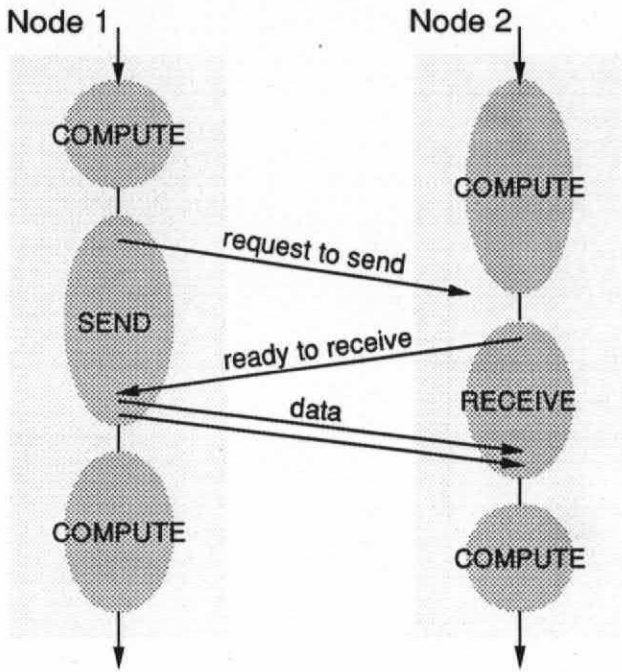
All 4 neighbors transfer into the same segment, but at different offsets (and stride).

When the transfer terminates, computation may start. The segment can be re-opened for the next iteration.



Send & receive issues

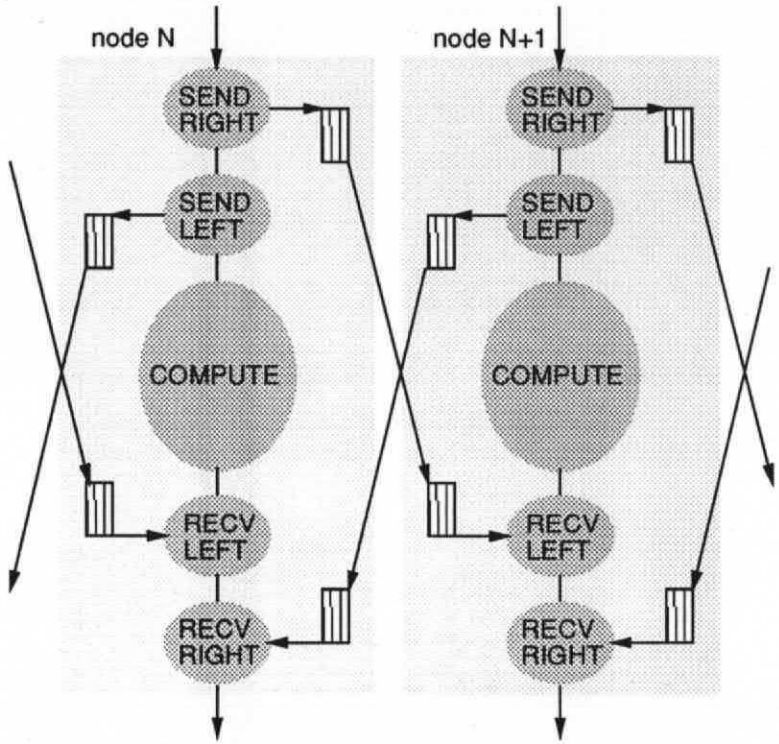
Blocking send&receive



→ 3-way latency

Time

Non-blocking send&receive



→ Buffering required

Active Messages — Summary

Simple idea:

Associate a little remote computation with every message

→ universal communication mechanism

Key contribution:

Restrict remote computation

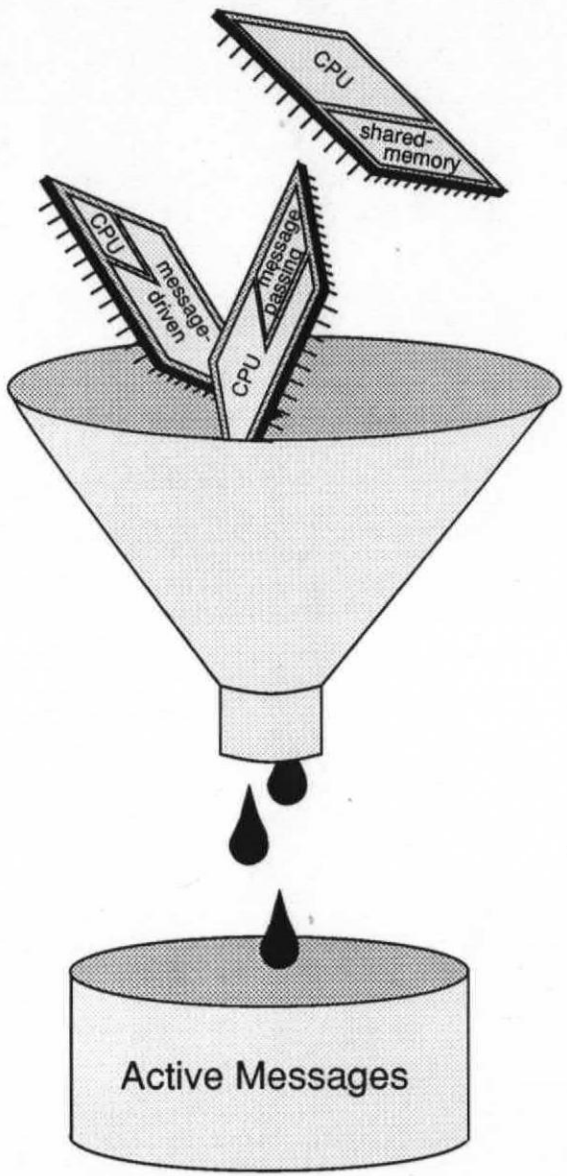
→ simplicity

Expose what the hardware does
(in a meaningful way)
(the hardware doesn't do send&receive)

Multiprocessor building block is
NOT necessarily sum of existing hardware.

Potential pitfalls:

Buffering, scheduling, deadlock



Active Message Handlers

Restricting message handlers is crucial for performance

Handlers

Provide remote service

Allowed:

- Access & change memory
- Perform simple arithmetic (e.g. address calculations)
- Send reply

NOT allowed:

- Chaining remote services (e.g. forward requests)
- Systolic computation
- Suspend (e.g. out of resources)
- Arbitrary computation

Integrate message into computation

Allowed:

- Store message data into memory
- Perform simple arithmetic (e.g. address calculations)
- Change variables in program (e.g. signal completion of comm.)

NOT allowed:

- Arbitrary computation
- Suspend

Network substrate vs. communication library

- Active Message vs. RPC

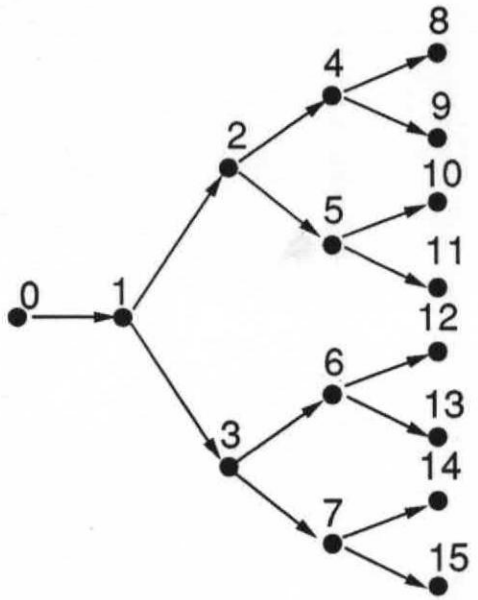
  RPC requires task queue

  → more sophisticated scheduling than Active Messages provide
  → use Active Messages to manage task queue

- Forwarding of messages

  Example: broadcast tree

```
void bcast_handler(int *addr, int data)
{ *addr = data;
  if(self*2 < partition_size) {
    CMAM_4(self*2,   bcast_handler, addr, data);
    CMAM_4(self*2+1, bcast_handler, addr, data);
  }
}
```



  Imagine (friday 13th):

  - Node 0 sends broadcasts at max rate
  - Node 4 doesn't service network (e.g. OS call)
  - Node 2 can't send to 4 (network backed-up)
  - Node 2 must service network (to avoid deadlock)
  - Node 2 will have to buffer all broadcasts (on the stack)
  - How much buffering is there?

  → Use task queue (+ buffering?)

  Limit buffering thru flow control or higher-level knowledge (acyclic pattern)

Try it out!

Slides

file: `~tve/talk/slides.tar`

print: `lpr ~tve/talk/[0-9]*.ps`

info: `~tve/talk/README`

CMAM:

doc - Int'l Symposium on Computer Architecture paper `~tve/cmam/doc/isca.ps`

doc - Getting Started `~tve/cmam/doc/intro.ps`

doc - Man pages `~tve/cmam/man3/*`

source `~tve/cmam/src`

examples `~tve/cmam/*`