Integrating Compilation Technology and Processor Architecture for Cost-Effective Concurrent Multithreading

Pen-Chung Yew

Department of Computer Science
University of Minnesota

http://www.cs.umn.edu/Research/Agassiz
yew@cs.umn.edu

Technology Trends

- Available transistors on a chip will continue to increase
  - Good for exploiting more parallelism and more functionalities
- Clock rate is being limited more by the interconnect delay than by gate delay
  - Single large window with wide-issue is not scalable
  - Use deeper pipeline, multiprocessor (on a chip) or concurrent multi-threaded architecture
- Compiler technology will become even more important to simplify hardware complexity and to improve efficiency

Strength & Limitations of Existing Approaches

Superscalar

- Strength
  - Accurate and effective run-time dependence check
  - Out-of-order execution
  - HW supported branch prediction, speculative and predicated execution at instruction level
- Limitations
  - A single instruction window is not easy to scale
  - Limited to instruction-level parallelism
  - Exploit only innermost loops

Strength & Limitations of Existing Approaches (Cont.)

Multiprocessor on a chip (MOAC)

- Strength
  - More scalable (shorter wires)
  - More flexible (various interconnect schemes)
  - Exploit primarily loop-level parallelism
- Limitations
  - Higher communication overhead
    - Need larger granularity
    - Doall loops only
  - Need a coherent cache
  - No thread-level speculation
  - Primarily for scientific codes, not general-purpose

Concurrent Multi-Threaded Architectures

- A hybrid of superscalar & multiprocessor-on-a-chip
  - More scalable than superscalar
  - More general-purpose than multiprocessor-on-a-chip
- We focus on speeding up single program execution
- Not for throughput (multiprogramming)
- Not for memory latency hiding (single thread active at a time)

Challenges in General-Purpose Applications

- Loops with early exits (Do-while loops)
  - requires speculation (thread-level most desirable)
    => not available in multiprocessors
- Parallelism exists in some outer loops
  - requires thread-level support
    => not available in superscalar
- Pointers/aliases complicate dependence analysis
  - requires run-time data dependence checking both within and between threads, or data speculation
    => not available in multiprocessors/superscalar

Challenges in General-Purpose Application (Cont.)

- Small basic blocks in straight-line code
  - requires instruction-level speculative/predicated execution, branch prediction, profiling-based compilation, out-of-order execution
    => very-well developed in superscalar
- Small loops or parallel sections
  - require fast communication/synchronization
  - communication directly between register files, or special buffers without going through memory => not available in multiprocessors

Concurrent Multi-threaded Architectures (CMA's)

- Support multiple threads (multiple PC's)
  - Handle loop-level parallelism (or parallel sections) more naturally
  - Not limited to innermost loops
- Support thread-level speculation (on data and/or control dependences)
  - Handle Do-while loops, Do-across loops
- Support run-time dependence checking within and between threads (Superthreading) or data speculation (Multiscalar)
  - Solve pointer/alias problems
  - Allow fast communication/synchronization

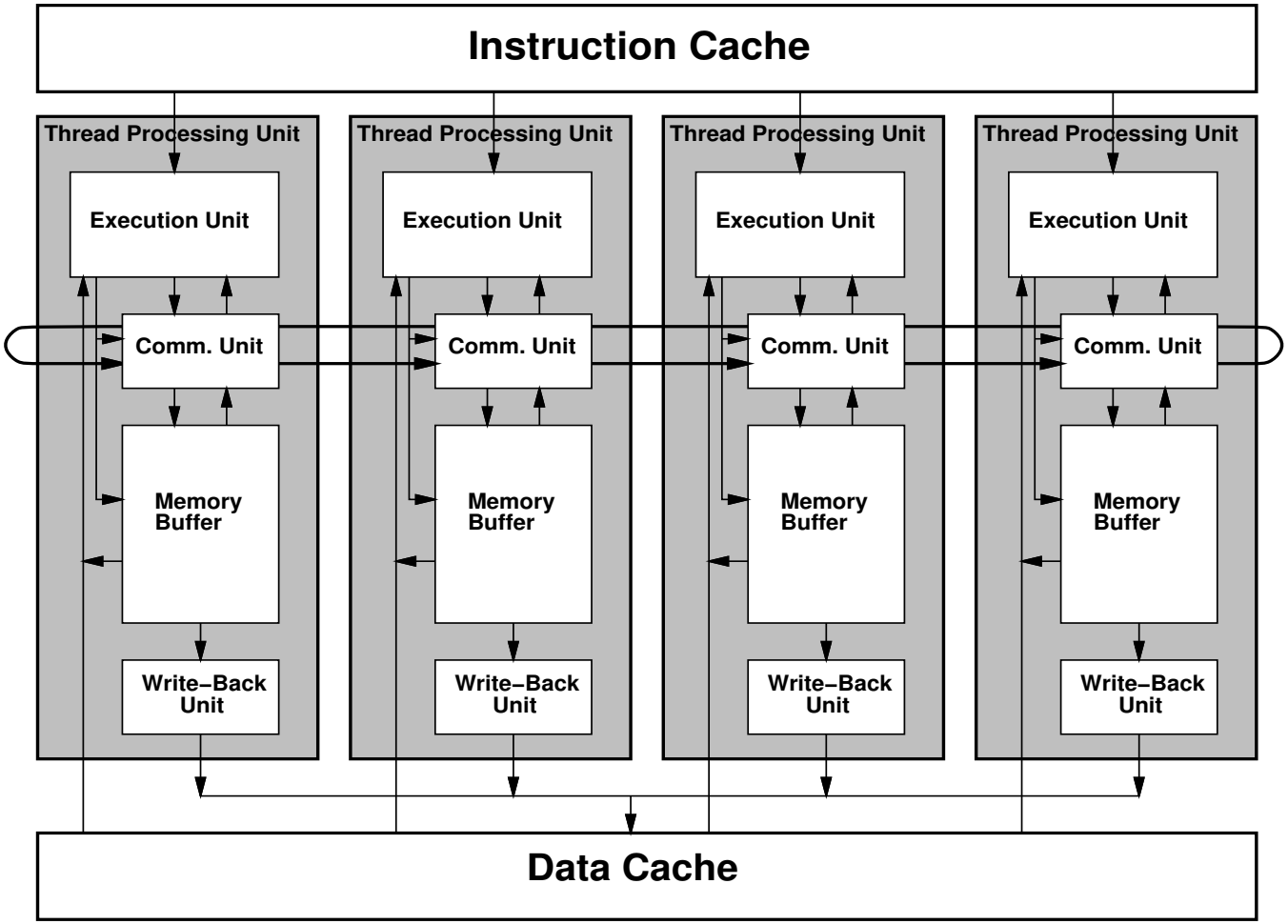
Concurrent Multithreaded Architectures (Cont.)

- Allow instruction-level parallelism in each thread to leverage existing superscalar technology
- Support fast communication/synchronization between threads
  - more tightly coupled than multiprocessors

Comparison of Concurrent Multi-threaded Architectures

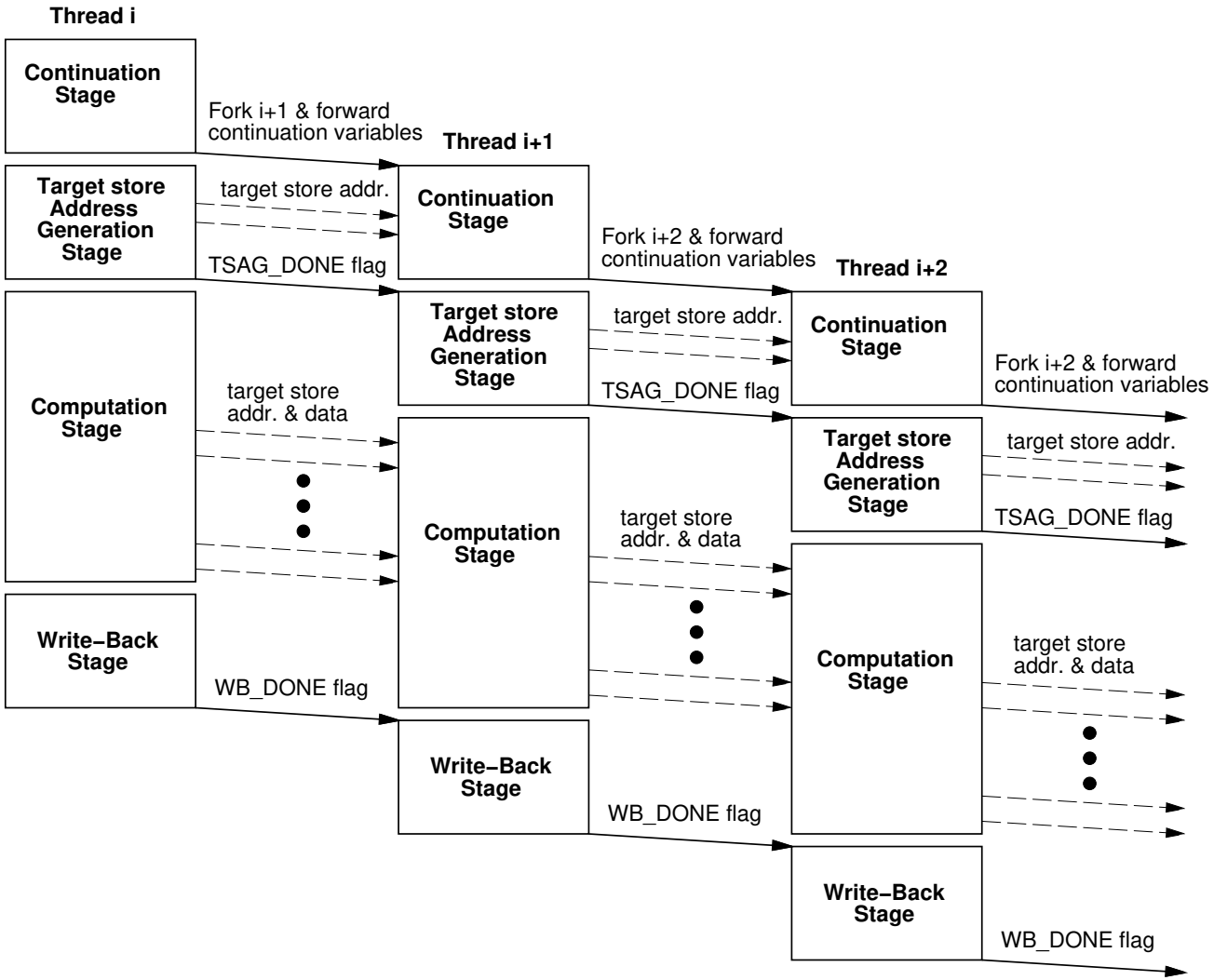
Feature	XIMD	M-machine	Multiscalar	SPSM	Superthread
Control speculation	no	no	yes	yes	yes
Data speculation	no	no	yes	yes	no
Memory buffering	no	no	load/store	load/store	store
Run-time data dependence check	no	no	yes (squash)	yes (squash)	yes (enforce)
Data transferred between threads	register/ memory	register/ memory	register/ memory	no	memory
Data transfer mechanism	global register	register/ data cache	register/ ARB	data cache	memory buffer
Data synchroniza-tion	barrier	full/empty bit	register mask	no	target store

Superthreaded Architecture



J.-Y. Tsai and P.-C. Yew, "The Superthreaded Architecture: Thread Pipelining with Run-time Data Dependence Checking and Control Speculation," *PACT'96, pp 25-46*

Thread Pipelining Execution Model



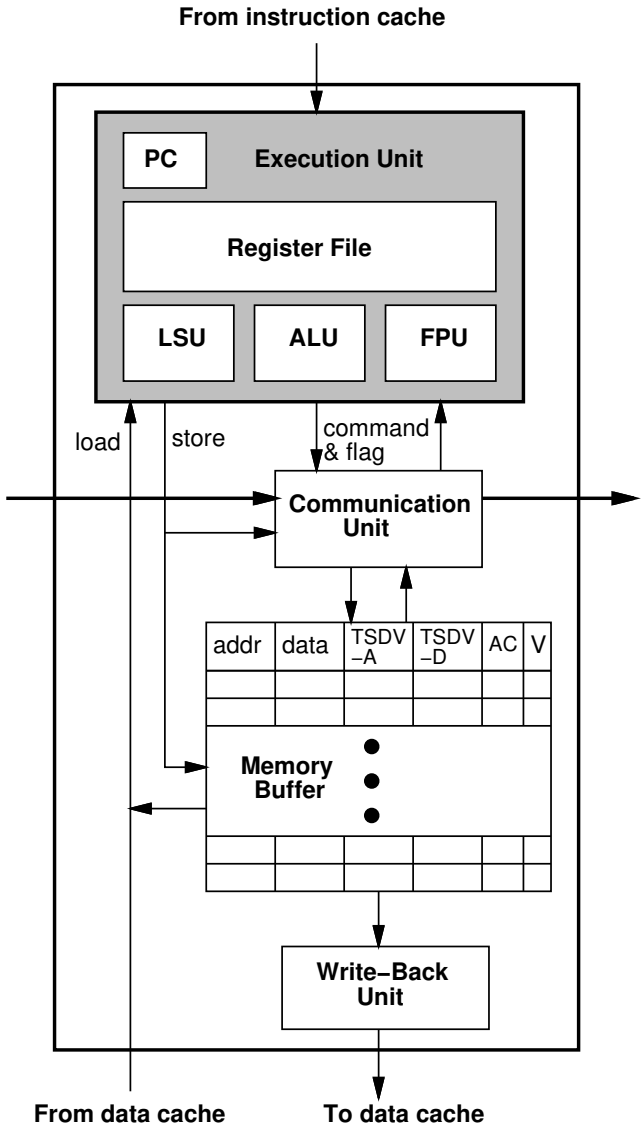
Example Code (from 124.m88ksim in SPEC95)

```
while (funct_units[i].class != ILLEGAL_CLASS)
  {
    if (f->class == funct_units[i].class)
      {
        if (minclk > funct_units[i].busy)
          {
            minclk = funct_units[i].busy;
            j = i;
            if (minclk == 0) break;
          }
      }
    i++;
  }
```

```
/* Continuation Stage */
L1:  i_1 = i;
     store_ts(&i, i_1+1);
     fork L1;
/* TSAG Stage */
     allocate_ts(&minclk);
     wait_tsag_done;
     release_tsag_done;
/* Computation Stage */
     if (funct_units[i_1].class == ILLEGAL_CLASS)  {
       abort_future;
       i = i_1;
       goto L2;
     }
     if (f->class == funct_units[i_1].class) {
       if (minclk > funct_units[i_1].busy) {
         store_ts(&minclk, funct_units[i_1].busy);
         j = i_1;
         if (minclk == 0) {
           abort_future;
           i = i_1;
           goto L2;
         }
       } else
         release_ts(&minclk);
     } else
       release_ts(&minclk);
     stop;
/* Write-back Stage -> performed automatically */
L2:
```

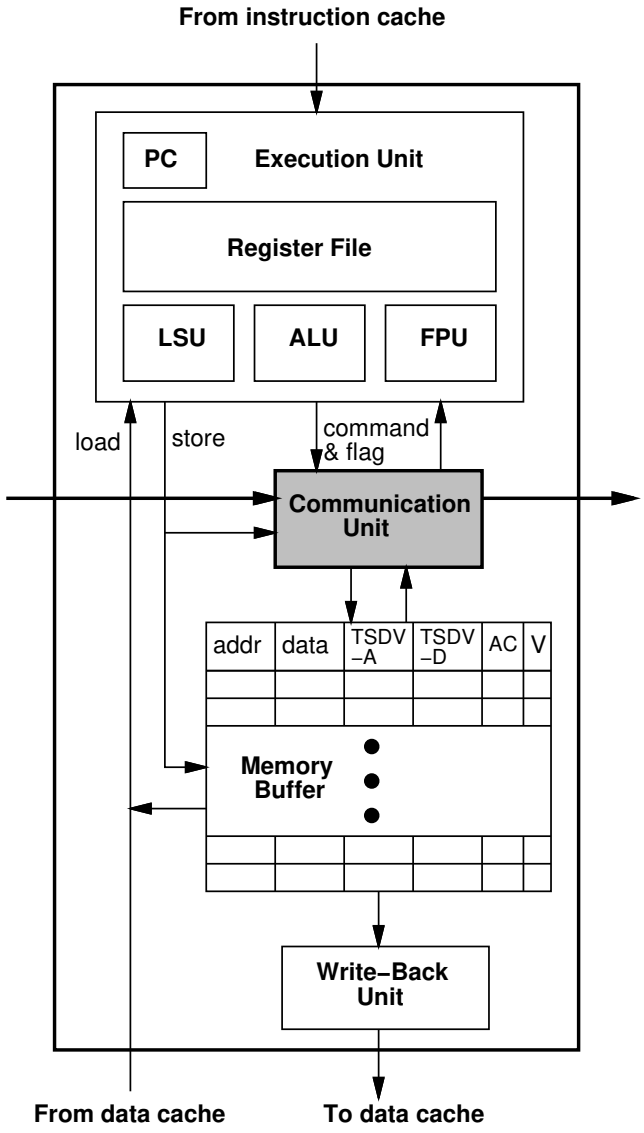
Execution Unit

- A superscalar processor
- Forward load/store, synchronization, and thread management instructions to the communication unit or the memory buffer



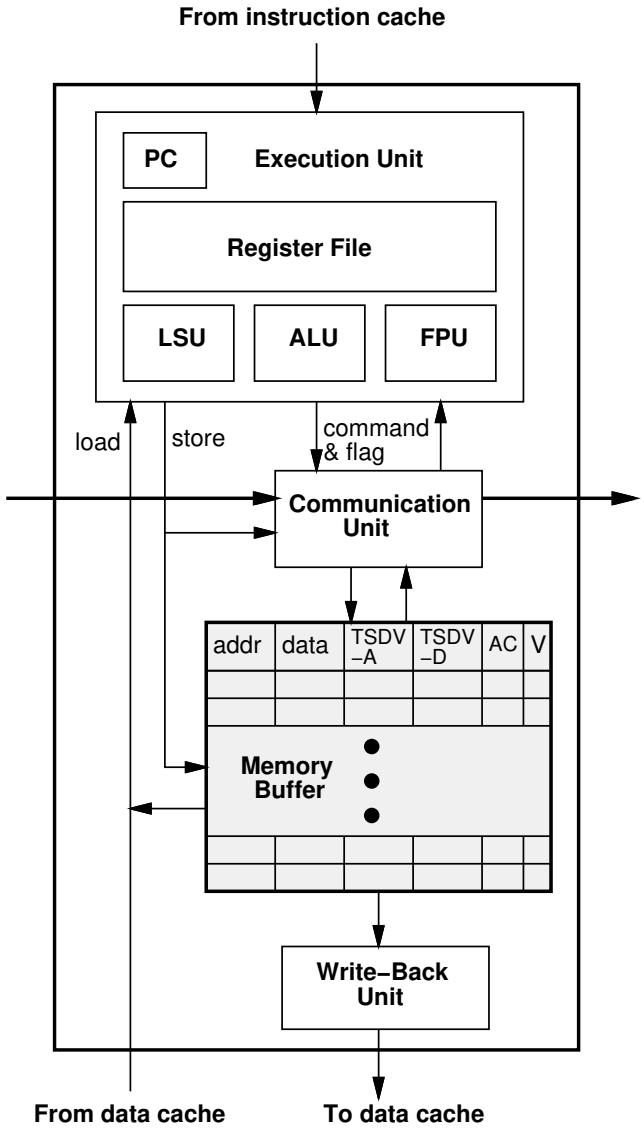
Communication Unit

- Receive commands and data from the execution unit and the up-stream communication unit
- Forward commands and data to the execution unit, the memory buffer, and the down-stream communication unit



Memory Buffer

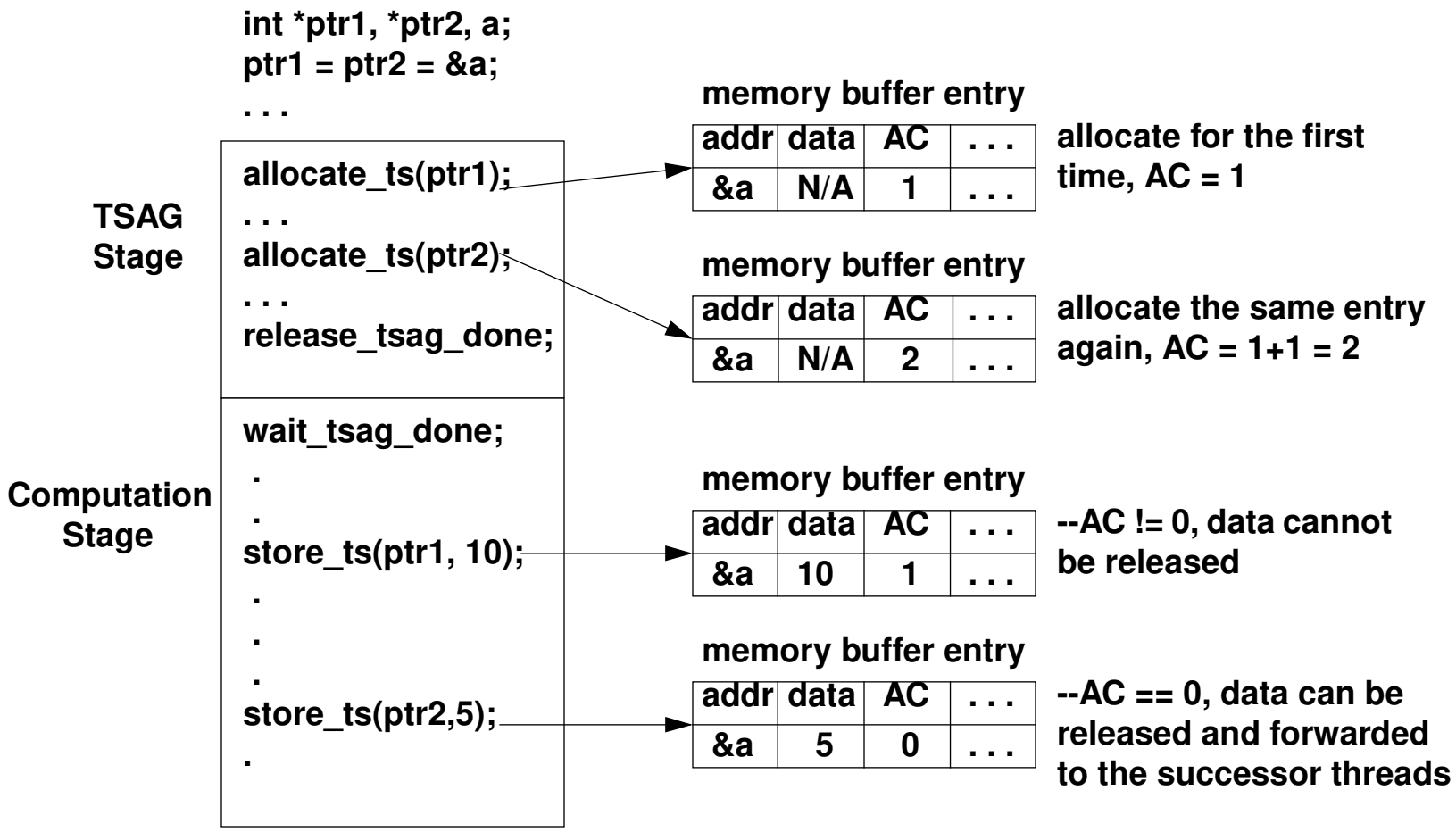
- Support run-time data dependence checking
  - Store target store entries from predecessor threads and the current thread
  - Perform data dependence checking for load operations
- Support speculative execution
  - Buffer all store data written by the current thread



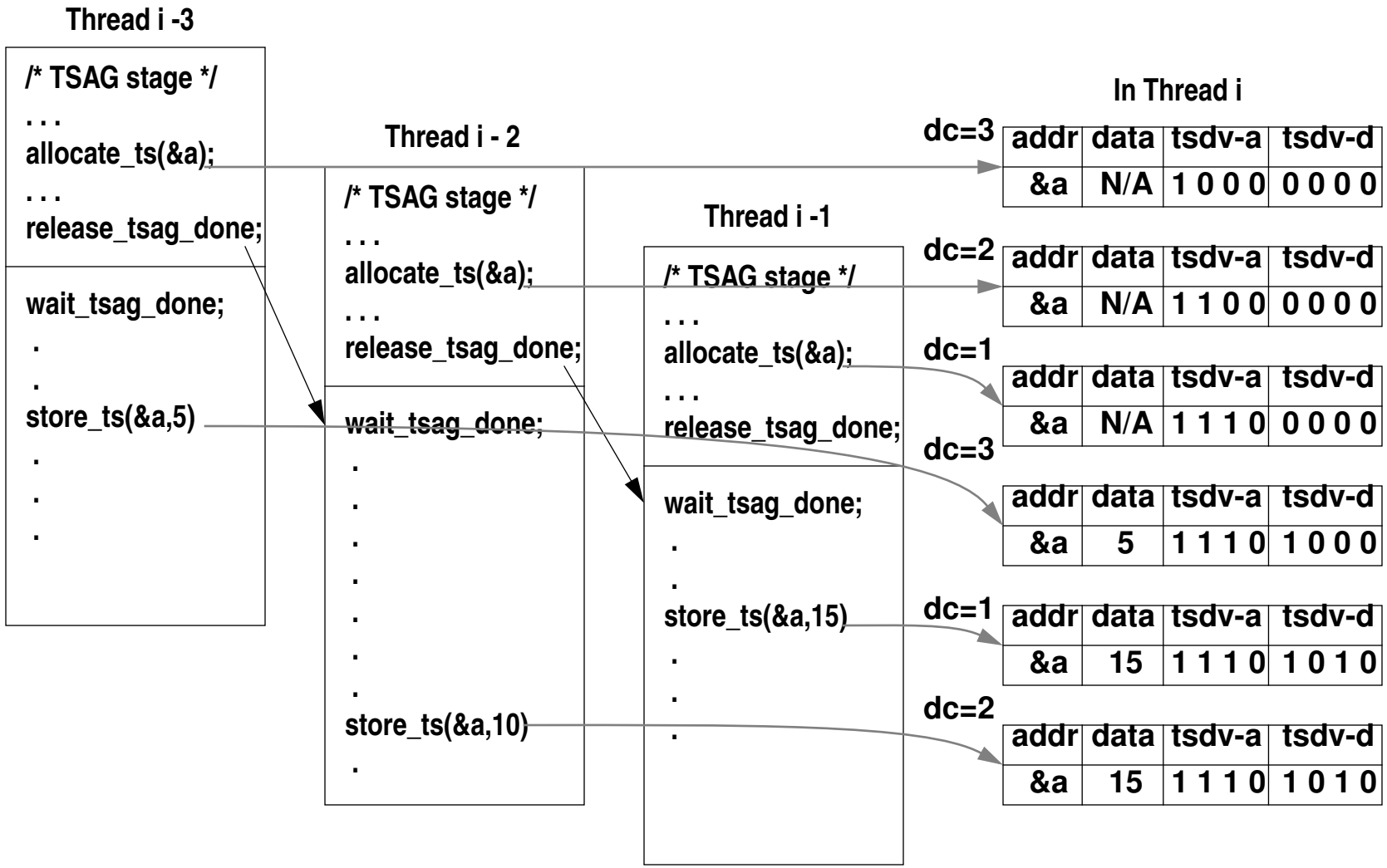
Memory Buffer Entry

- Data/Address field
- Valid bit
  - indicate whether this entry is valid or not
- Alias Count (AC)
  - Number of aliases to the same target store address in the current thread
- Target Store Distance Vectors (TSDV-A/TSDV-D)
  - Bit *n* is set if the target store address/data is received from the *n*th predecessor thread
  - For enforcing output dependences between threads

Usage of Alias Count (AC)



Usage of Target Store Distance Vectors



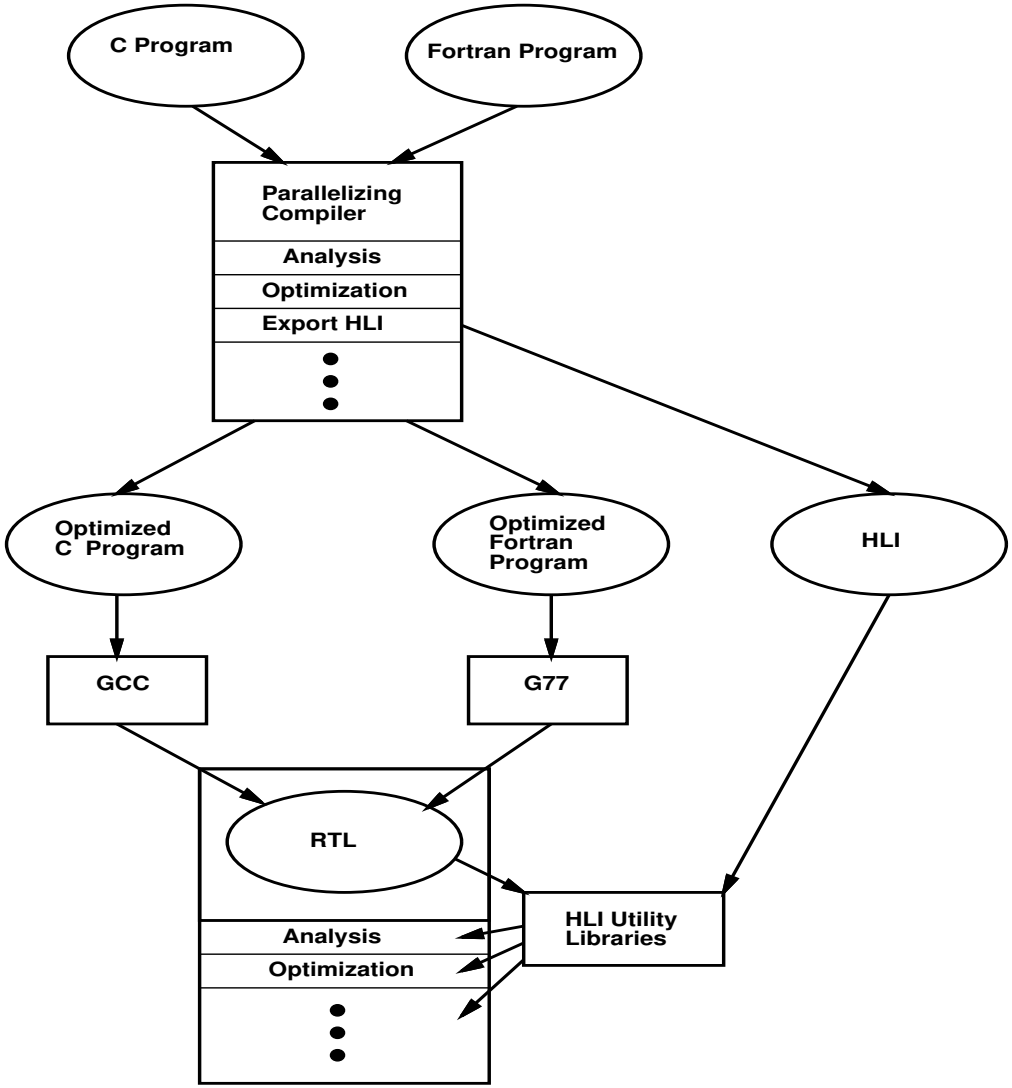
Compiler Challenges For CMA's

- Exploiting loop-level parallelism
  - Similar to multiprocessors
  - Leverage existing parallelizing compiler techniques
  - Data dependence analysis
    => exact/maybe/no dependence
  - Alias analysis
    => exact/maybe/no dependence
  - Loop level optimization
    => loop interchange
    => Do-across/Do-while loop optimization
      - increase overlap between iterations
      - rely on HW to perform run-time dd check and synchronization

Compiler Challenges For CMA (Cont.)

- Need to exploit both loop-level and instruction-level parallelism
  - Need an integrated compiler
- Thread partitioning
  - Load balancing
  - Avoid memory buffer/ARB overflow

Agassiz Integrated Compiler



High-Level Information (HLI)

- Program structures
  - loops, if-then-else blocks, etc.
- Alias information
  - equivalent (exact/maybe) memory accesses
- Loop-carried data dependences
- Intraprocedural data flow information
  - sub-region (loop) REF/MOD information
- Interprocedural data flow information
  - function call REF/MOD information

CMA Specific Compilation Techniques

- Conversion of conditional data speculation to control speculation
- Distributed heap memory management
- Using critical sections for order-irrelevant operations
- Memory buffering in the main memory

Conversion of Data Speculation to Control Speculation

From procedure *compress* in 129.compress:

```
   while ((c = getbyte()) != EOF ) {
      in_count++;
      fcode = (long) (((long) c << maxbits) + ent);
      i = ((c << hshift) ^ ent);

      if ( htabof (i) == fcode ) {
         ent = codetabof (i);
         continue;
      } else if ((long)htabof (i) < 0)
         goto nomatch;
probe:
      . . .
      if ( htabof (i) == fcode ) {
         ent = codetabof (i);
         continue;
      }
      if ( (long)htabof (i) > 0 )
         goto probe;
nomatch:
      /* computation on which the successor threads
         are dependent  */
       . . .
   }
```

```
/* Continuation Stage */
loop_continue:
      FORK(loop_continue);
/* TSAG Stage - generate only ts-addresses in speculated paths*/
      ALLOCATE_TS(&ent);
      ALLOCATE_TS(&in_count);
      RELEASE_TSAG_DONE;
      WAIT_TSAG_DONE;
/* Computation Stage */
      c = getbyte();
      if ((c = getbyte()) == EOF ) {
         ABORT_FUTURE; /* control speculation incorrect */
         goto loop_exit;
      }
      STORE_TS(&in_count, in_count+1);
      fcode = (long) (((long) c << maxbits) + ent);
      i = ((c << hshift) ^ ent);
      if (htab[i] == fcode) {
         STORE_TS(&ent,codetab[i]);
         STOP;          /* thread retires as expected */
      } else if ((long)htabof (i) < 0)
         goto nomatch;
probe:
     . . .
nomatch:
      ABORT_FUTURE;  /* data speculation incorrect */
      . . .
      goto loop_continue;
loop_exit:
      . . .
```

Distributed Heap Memory Management

From procedure *sweep* in 130.li:

```
for (n = seg->sg_size; n--; p++)
  if (!(p->n_flags & MARK)) {
      switch (ntype(p)) {
      case STR:
         if (p->n_strtype == DYNAMIC && p->n_str != NULL){
            total -= strlen(p->n_str)+1;
            free(p->n_str);
         }
         break;
       case FPTR:
          if (p->n_fp)
             fclose(p->n_fp);
          break;
       case VECT:
          if (p->n_vsize) {
             total -= p->n_vsize * sizeof(NODE **);
             free(p->n_vdata);
          }
          break;
       }
       p->n_type = FREE;
       p->n_flags = 0;
       rplaca(p,NIL);
       rplacd(p,fnodes);
       fnodes = p;
       nfree++;
   }
   else
       p->n_flags &= ~(MARK | LEFT);
}
```

```
/* Continuation stage  */
loop_continue:
    if (!(n+1) goto loop_exit;
    STORE_TS(&p,p+1);
    STORE_TS(&n,n-1);
    FORK(loop_continue);
/* TSAG stage */
     . . .
/* Computation stage */
   tpu_id = get_thrd_unit_id();
   if (!(p->n_flags & MARK)) {
      switch (ntype(p)) {
      case STR:
        if (p->n_strtype == DYNAMIC && p->n_str != NULL) {
           STORE_TS(&total, total - strlen(p->n_str) - 1);
           st_free(tpu_id, p->n_str);
         }
        break;
      case FPTR:
           . . .
      case VECT:
        if (p->n_vsize) {
          STORE_TS(&total, total - p->n_vsize * sizeof(NODE **));
          st_free(tpu_id, p->n_vdata);
         }
       break;
      }
       . . .
   } else
       . . .
   STOP;
```

Using Critical Section for Order-irrelevant Operations

From procedure *regclass* in 126.gcc:

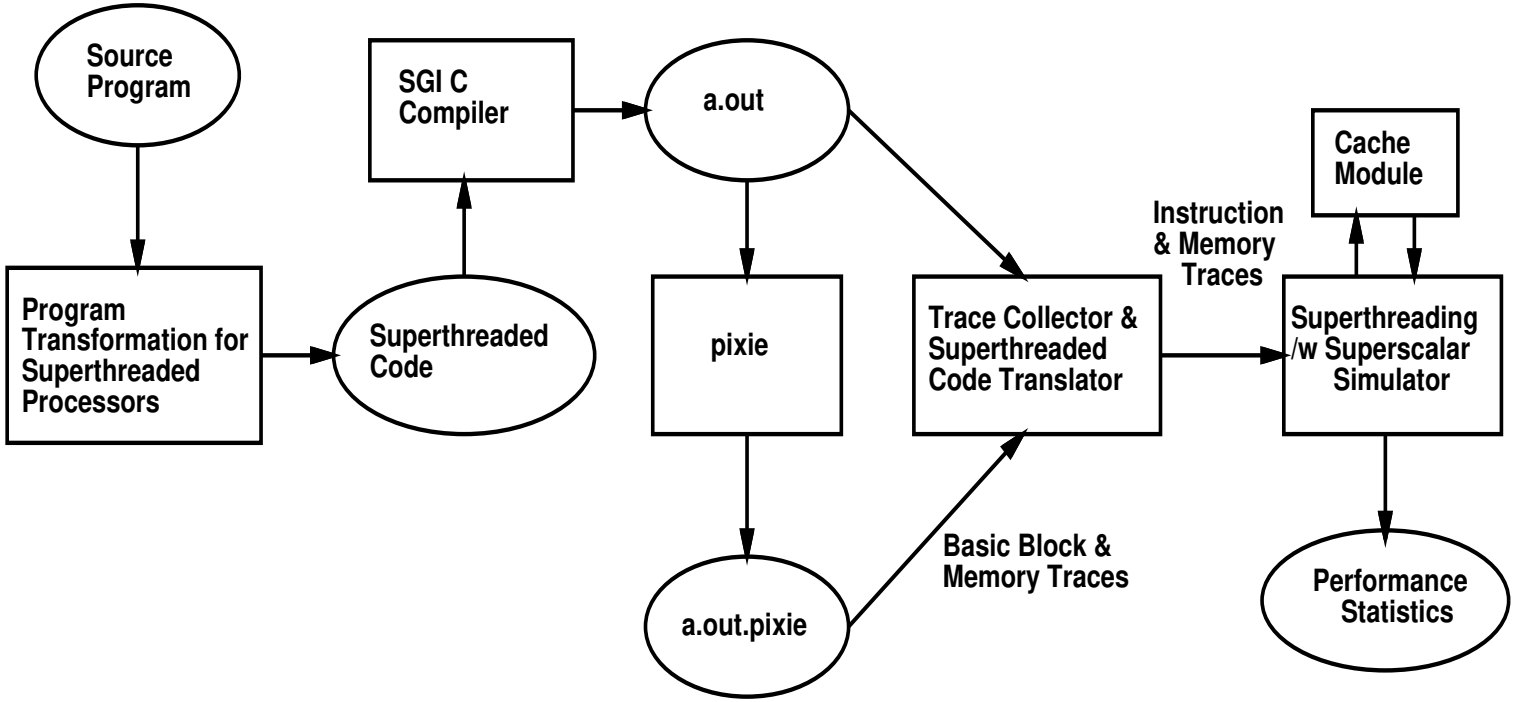
```
for (insn = f; insn; insn = NEXT_INSN (insn)) {
    . . .
    /* Scan the instructions and record each time it would
       save code to put a certain register in a certain class.  */
    . . .

    /* add the cost for each operand to the total costs for
       its register.  */

    for (i = 0; i < noperands; i++) {
        if (GET_CODE (recog_operand[i]) == REG
            && REGNO (recog_operand[i]) >=
            FIRST_PSEUDO_REGISTER) {
            int regno = REGNO (recog_operand[i]);
            struct costs *p = &costs[regno], *q = &op_costs[i];

            p->mem_cost += q->mem_cost * loop_cost;
            for (j = 0; j < N_REG_CLASSES; j++)
                p->cost[j] += q->cost[j] * loop_cost;
        }
    }
}
```

```
insn = f
/* Continuation Stage */
loop_continue:
    if (insn == 0)  goto loop_exit;
    insn_1 = insn;
    ST_STORE_TS(&insn, NEXT_INSN (insn))
    FORK(loop_continue);

/* TSAG Stage */
    . . .

/* Computation Stage */
    . . .
    for (i = 0; i < noperands; i++) {
        if (GET_CODE (recog_operand[i]) == REG
            && REGNO (recog_operand[i]) >=
            FIRST_PSEUDO_REGISTER)  {
            int regno = REGNO (recog_operand[i]);
            struct costs *p = &costs[regno], *q = &op_costs[i];

            ACQUIRE_LOCK(regno);
            p->mem_cost += q->mem_cost * loop_cost;
            for (j = 0; j < N_REG_CLASSES; j++)
                p->cost[j] += q->cost[j] * loop_cost;
            RELEASE_LOCK(regno);
        }
    }
    STOP;
loop_exit:
```

Compiler Techniques Used in Hand Compiled Codes

Transformations	wc	cmp	alvinn	ear	m88ksim	gcc	compress	ijpeg
Function inlining							X	
Induction	X	X	X	X	X			X
Loop unrolling	X	X	X	X	X			
Loop interchange			X					
Stmt. reordering	X				X	X	X	
Data speculation						X	X	
Distributed heap						X		
Critical section	X					X		
Main Mem. buffer						X		

Simulation Environment



Simulation Configuration

- For superscalar
  - Branch target buffer: 1024 entries, 4-way associative
  - Reorder buffer size: issue_rate x 8
  - Functional units: 2 IPU, 1 FPU, 1 LDU per two issues (using the same execution latency as MIPS R10000)
- For superthreading
  - Memory buffer size: 96 entries, full associative, 2-cycle pipeline
- Cache module
  - 64KB, 4-way associative, 16-byte block, 2 read ports, 1 write port, 2-cycle latency, 12/24-cycle miss penalty

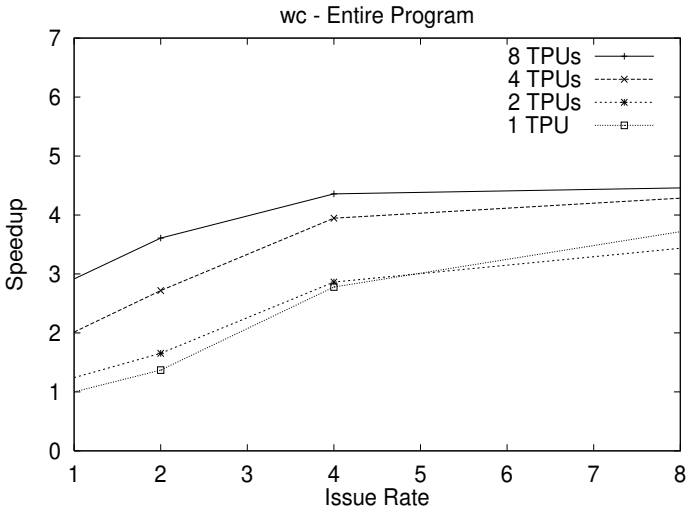
Simulation Configuration (cont.)

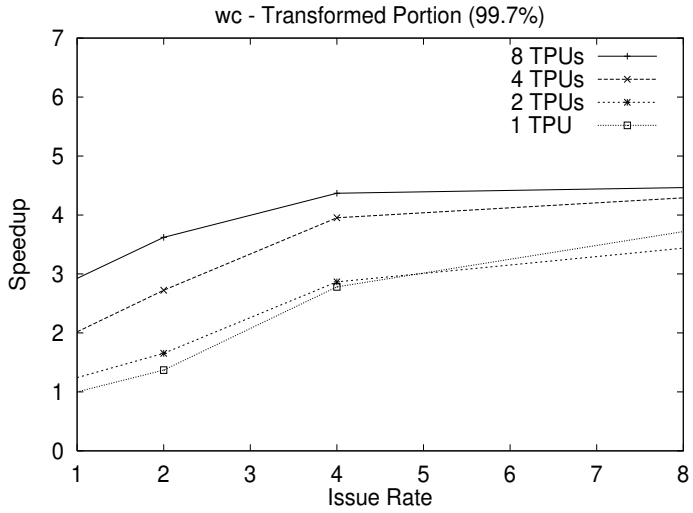
#TPU x Issue Rate	1 x 1	1 x 2	1 x 4	1 x 8	2 x 1	2 x 2	2 x 4	2 x 8	4 x 1	4 x 2	4 x 4	4 x 8	8 x 1	8 x 2	8 x 4	8 x 8	1 x 16	16 x 1
Reorder buffer size	8	16	32	64	8	16	32	64	8	16	32	64	8	16	32	64	128	8
Pending conditional branches	1	1	2	4	1	1	2	4	1	1	2	4	1	1	2	4	8	1
# IPUs (per TPU) #FPUs (per TPU) #LDUs (per TPU)	1 1 1	2 1 1	4 2 2	8 4 4	1 1 1	2 1 1	4 2 2	8 4 4	1 1 1	2 1 1	4 2 2	8 4 4	1 1 1	2 1 1	4 2 2	8 4 4	16 8 8	1 1 1
Communication bandwidth	-	-	-	-	1	1	2	2	1	2	2	3	2	2	3	4	-	2
Memory write-back bandwidth	1	1	1	2	1	1	2	4	1	2	4	8	2	4	8	16	4	4
# of Interleaved cache banks	1	1	1	2	1	1	2	4	1	2	4	8	2	4	8	16	4	4

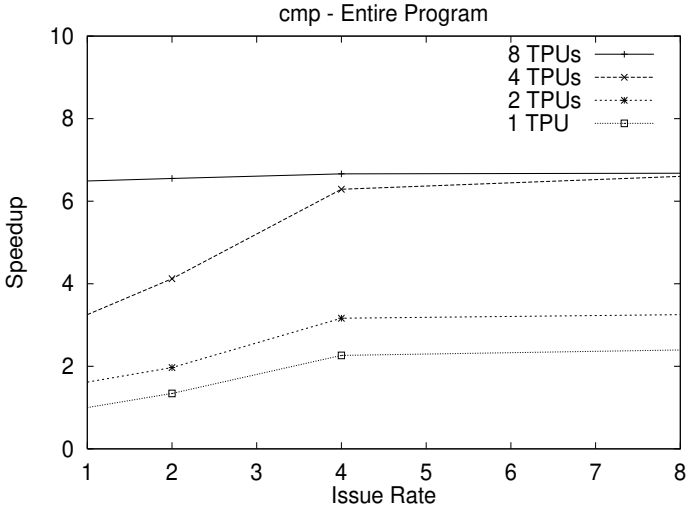
Benchmarks

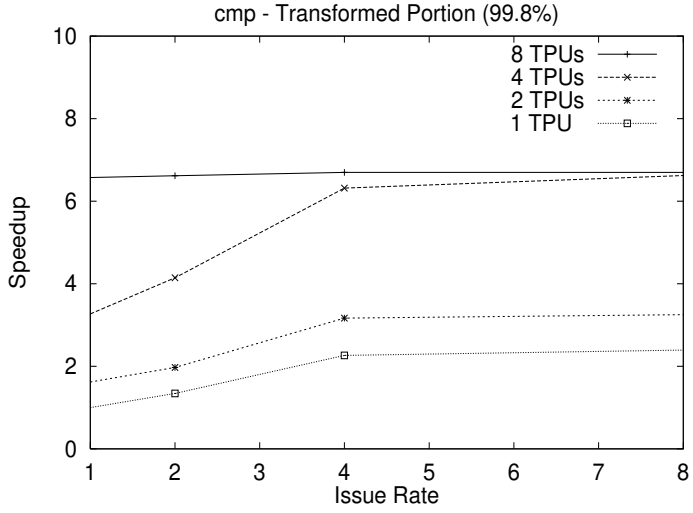
Program	wc	cmp	alvinn	ear	m88ksim	gcc	compress	ijpeg
Routine 1	wc	block_compare_and_count	input_hidden	EAR STEP	killtime	cse_main	compress	jpeg_fdct_islow
Routine 2			hidden_output	sos	ckbrkpts	regclass	decompress	jpeg_idct_islow
Routine 3			output_hidden	agc	alignd			fullsize_smooth_downsample
Routine 4			hidden_input	hwr	test_issue			forward_DCT
Routine 5			update_stats	differ-ence	loadmem			rgb_ycc_convert
Routine 6			update_weights	fos	rdwr			h2v2_smooth_downsample
Routine 7								h2v2_merged_upsample
dynamic %	99.7	99.8	88.6	98.6	75.9	29.8	90.2	69.9

Simulation Results - wc and cmp

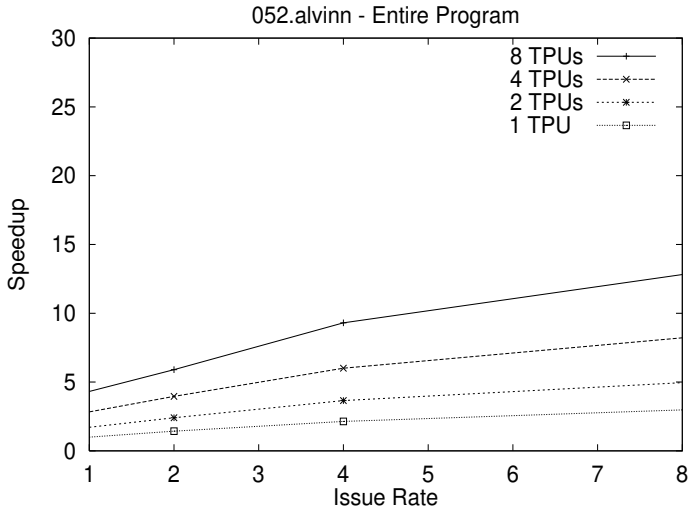


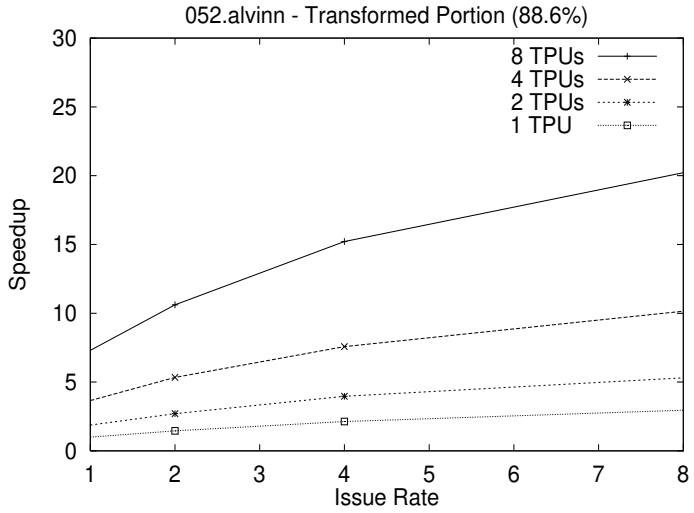


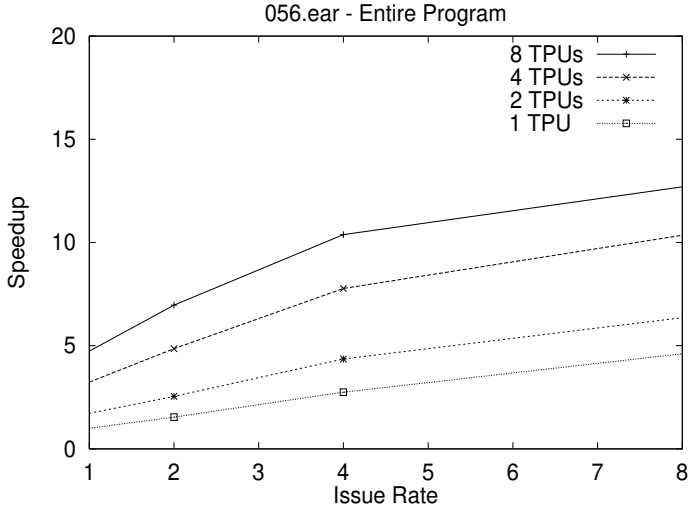


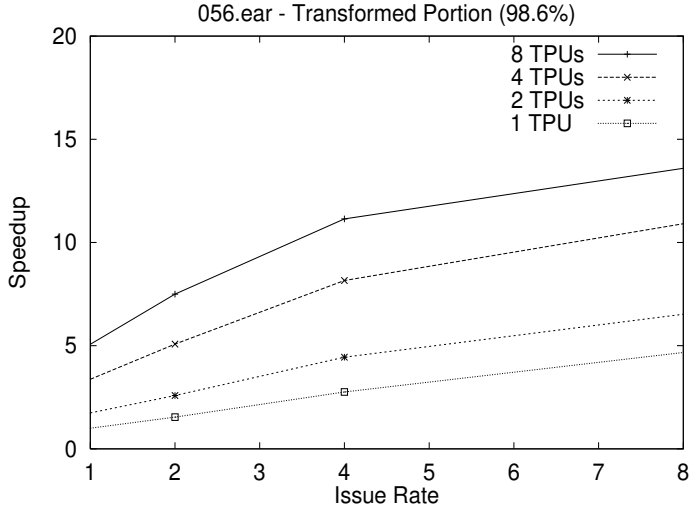


Simulation Results - 052.alvinn and 056.ear

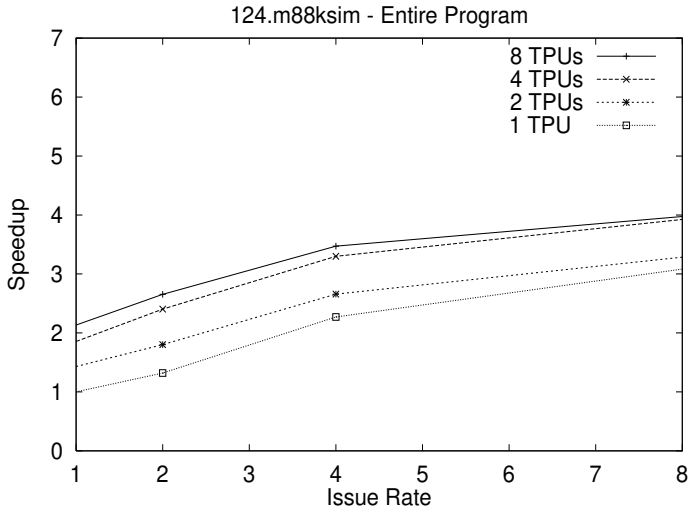


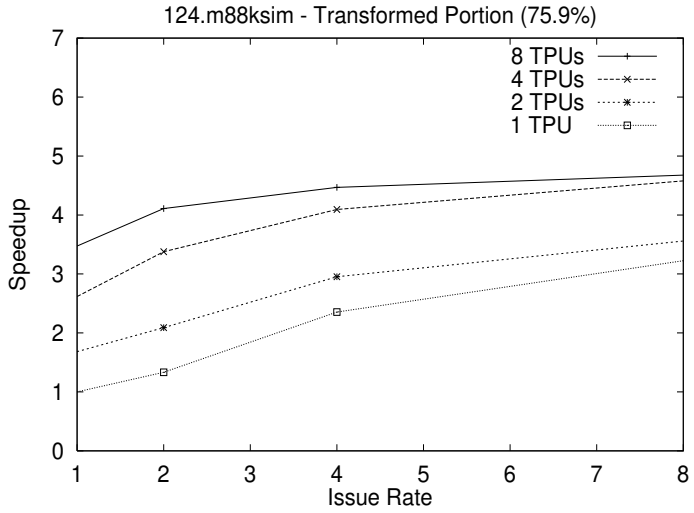


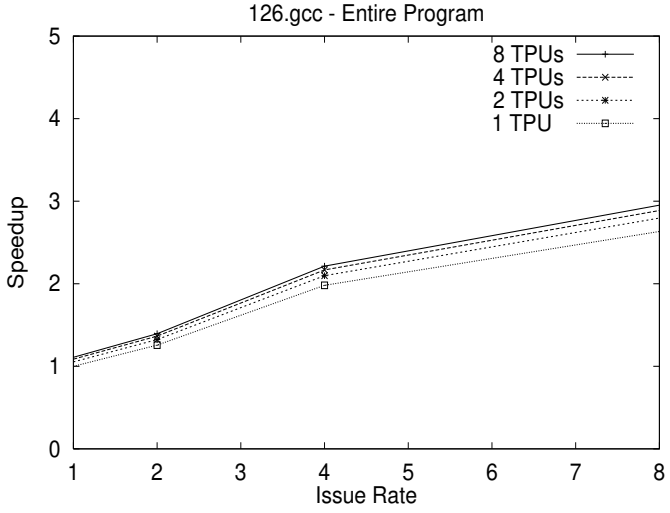


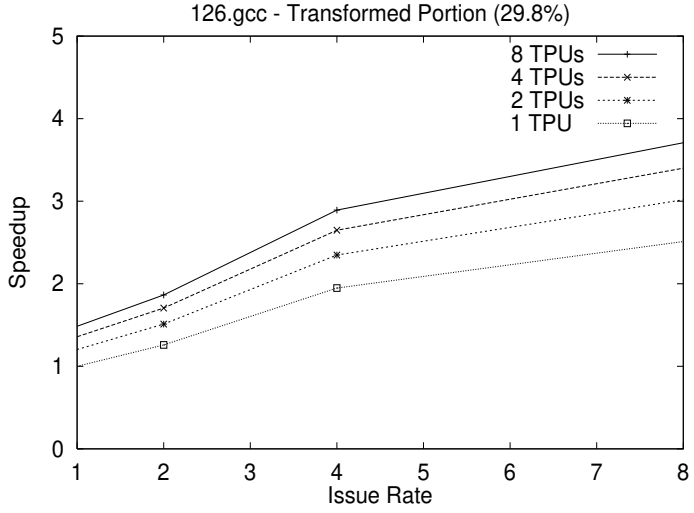


Simulation Results - 124.m88ksim and 126.gcc

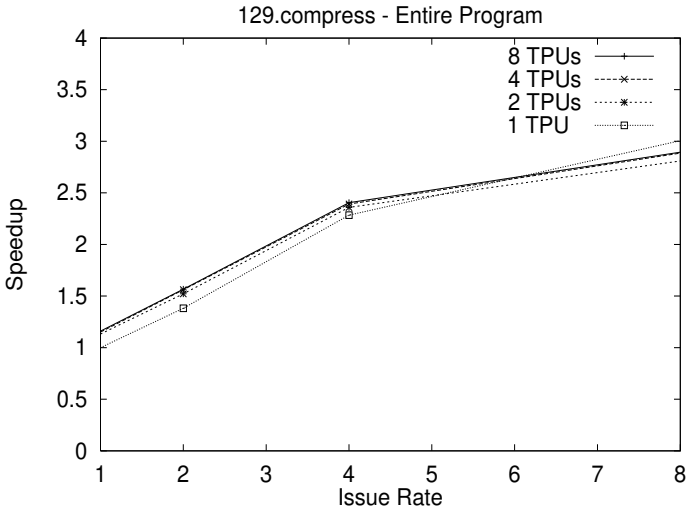


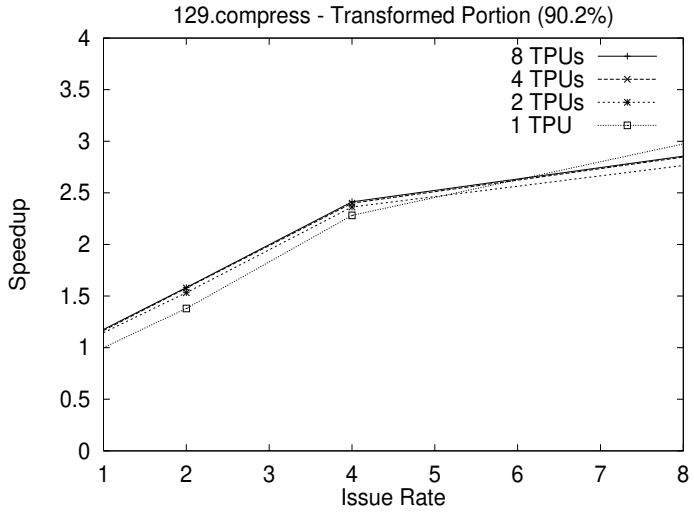


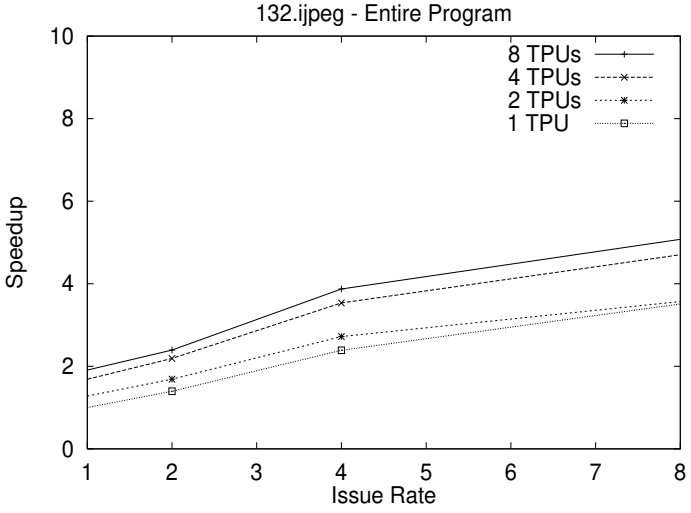


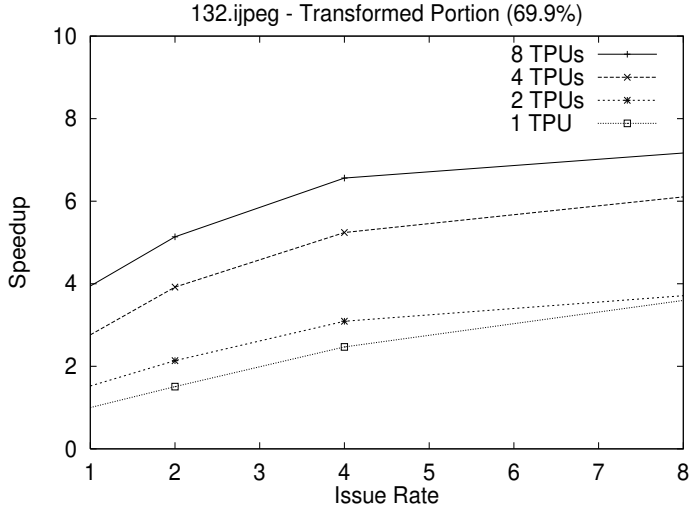


Simulation Results - 129.compress and 132.ijpeg

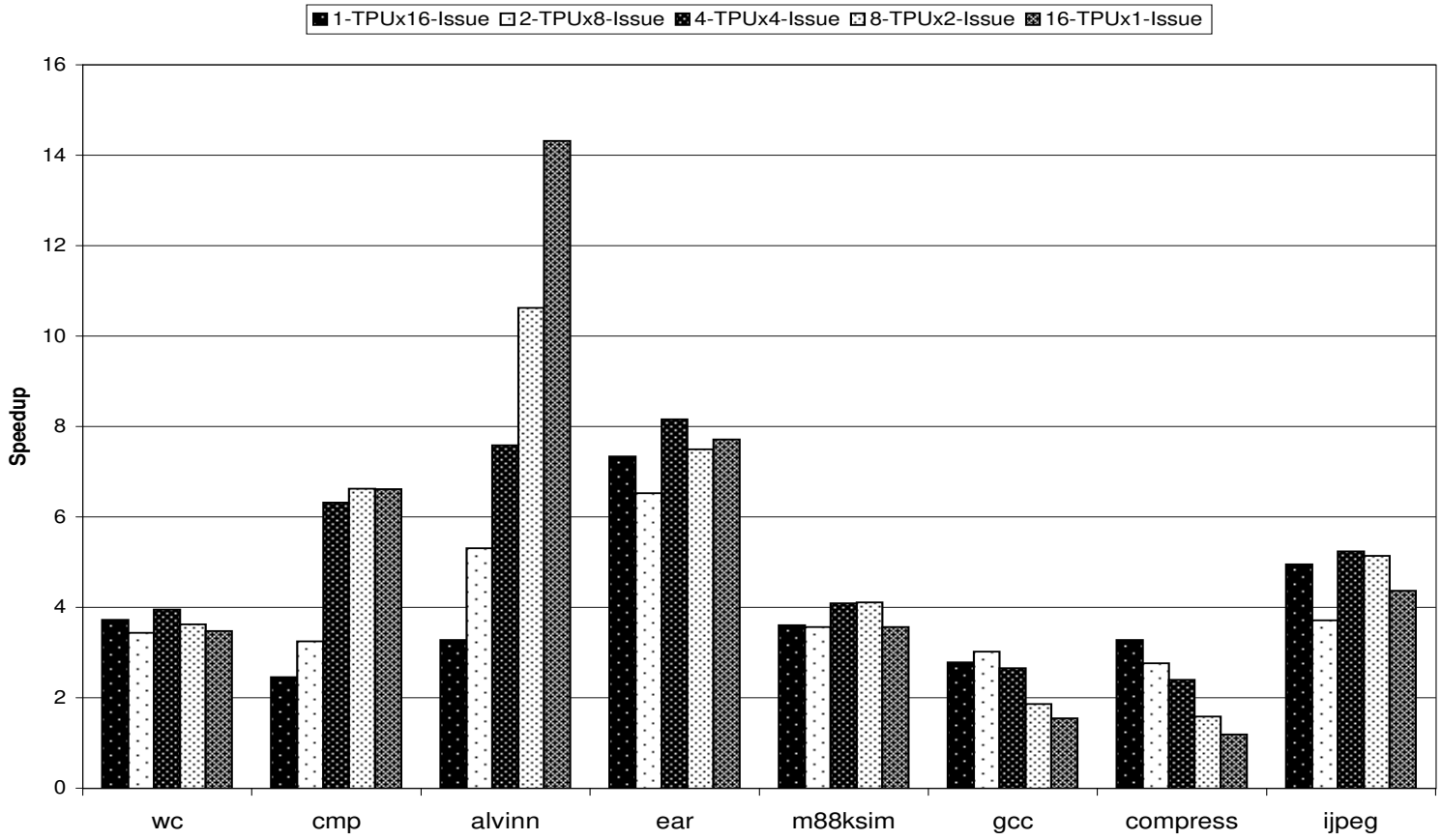








Performance Comparison (Transformed Portion)



Memory Buffer Usage

Programs	# of TPUs		
	2	4	8
wc	4	4	4
cmp	4	4	4
052.alvinn	72	73	73
056.ear	25	25	25
124.m88ksim	26	26	26
126.gcc*	3	3	3
129.compress	20	24	32
132.ijpeg	26	26	26

* using local memory buffer in main memory

Communication Bandwidth Requirement

Speedups (entire program) of a 4-TPU, 4-issue superthreaded processor with different communication bandwidth

Programs	Communication Bandwidth			
	1	2	3	4
wc	3.56	3.95	4.00	4.00
cmp	5.66	6.29	6.41	6.34
052.alvinn	5.90	6.01	6.04	6.06
056.ear	7.43	7.77	7.83	7.85
124.m88ksim	3.07	3.30	3.34	3.34
126.gcc*	2.17	2.17	2.17	2.17
129.compress	2.32	2.39	2.39	2.39
132.ijpeg	3.39	3.54	3.56	3.57

Data Cache bandwidth Requirement

4-TPU, 4-issue superthreaded processor

read ports	write ports	banks	wc			cmp			052.alvinn			056.ear		
			contention %		speed up	contention %		speed up	contention %		speed up	contention %		speed up
			r	w		r	w		r	w		r	w	
1	1	2	3.20	34.56	3.95	20.40	15.99	6.19	51.81	14.73	5.60	64.41	27.36	4.28
1	1	4	2.38	16.33	3.95	19.43	16.12	6.29	32.57	16.48	5.91	47.76	34.88	5.70
1	1	8	2.27	16.32	3.95	18.54	16.12	6.29	22.10	8.73	5.96	36.02	30.20	6.55
2	1	2	0.04	34.57	3.95	0.00	16.00	6.19	11.16	13.90	6.00	25.18	26.11	6.87
2	1	4	0.04	16.32	3.95	0.00	16.13	6.29	6.11	16.46	6.01	10.20	32.59	7.77
2	1	8	0.04	16.31	3.95	0.00	16.12	6.29	3.52	8.99	6.01	4.65	28.66	7.89
3	2	2	0.00	14.53	3.95	0.00	0.00	6.29	3.45	6.35	6.01	6.25	16.45	7.84
3	2	4	0.00	3.82	3.95	0.00	0.00	6.29	1.43	1.71	6.02	1.16	5.59	7.96
3	2	4	0.00	3.82	3.95	0.00	0.00	6.29	0.69	0.81	6.02	0.43	3.94	7.96
Cache hit ratio %			99.95	99.98		99.72	99.98		92.25	99.88		99.99	99.98	

Data Cache bandwidth Requirement (Cont.)

4-TPU, 4-issue superthreaded processor

read ports	write ports	banks	124.m88ksim			126.gcc			129.compress			132.ijpeg		
			contention %		speed up	contention %		speed up	contention %		speed up	contention %		speed up
			r	w		r	w		r	w		r	w	
1	1	2	34.52	17.07	3.21	18.38	22.07	2.13	27.37	11.03	2.24	53.41	20.40	2.97
1	1	4	20.87	24.38	3.27	13.56	31.01	2.14	22.70	12.10	2.25	37.72	29.24	3.28
1	1	8	14.92	17.42	3.29	11.62	25.66	2.15	20.90	7.19	2.26	29.50	23.30	3.38
2	1	2	2.03	16.43	3.30	0.90	23.19	2.16	0.65	11.48	2.39	15.93	20.25	3.45
2	1	4	0.96	24.93	3.30	0.37	32.34	2.17	0.20	13.26	2.39	6.36	28.64	3.54
2	1	8	0.60	17.63	3.30	0.18	27.29	2.17	0.11	8.86	2.39	3.36	23.29	3.55
3	2	2	0.10	7.88	3.29	0.13	9.69	2.17	0.02	2.58	2.39	2.76	10.34	3.55
3	2	4	0.05	2.10	3.30	0.04	7.21	2.17	0.00	2.06	2.39	0.98	5.42	3.56
3	2	4	0.04	0.87	3.30	0.02	5.05	2.17	0.00	0.00	2.39	0.30	4.03	3.56
Cache hit ratio %			99.61	98.24		99.41	99.37		99.11	98.52		99.81	99.50	

Conclusions

- CMA approach is a hybrid of superscalar & multiprocessor-on-a-chip with thread-level & instruction-level speculation
  - Can leverage existing superscalar & multiprocessor technology (both SW & HW)
- CMA allows both instruction-level & thread-level parallelism to be exploited with run-time dd checking
- CMA seems to be a viable alternative to wide-issue superscalar with reasonable performance
  - fill the gap between an wide-issue superscalar and a multiprocessor-on-a-chip