Compiler and Architecture Issues for Concurrent Multi-threaded Architectures

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 in a program
- 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

Strength & Limitations of Existing Approaches

Superscalar

- Strength
  - Accurate and effective run-time dependence check
  - 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.)

VLIW

- Strength
  - Simple HW support
- Limitations
  - Code not portable
  - Limited to instruction-level parallelism
  - Exploit only innermost loop

Strength & Limitations of Existing Approaches (Cont.)

Multiprocessor on a chip (MOAC)

- Strength
  - More scalable
  - More flexible
  - Exploit primarily loop-level parallelism
- Limitations
  - High communication overhead
    - Need large granularity
    - Doall loops only
  - Need a coherent cache
  - No 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 desiable)
    => not available in multiprocessors
- Parallelism exists in outer loops
  - requires thread-level support
    => not available in superscalar/VLIW
- Pointers/aliases complicate dependence analysis
  - requires run-time data dependence checking both within and between threads
    => not available in multiprocessors/superscalar/VLIW

Challenges in General-Purpose Application (Cont.)

- Small basic blocks in straight-line code
  - requires instruction-level speculative/predicated execution, branch prediction, profiling-based compilation, etc.
- Small loops
  - require fast communication/synchronization

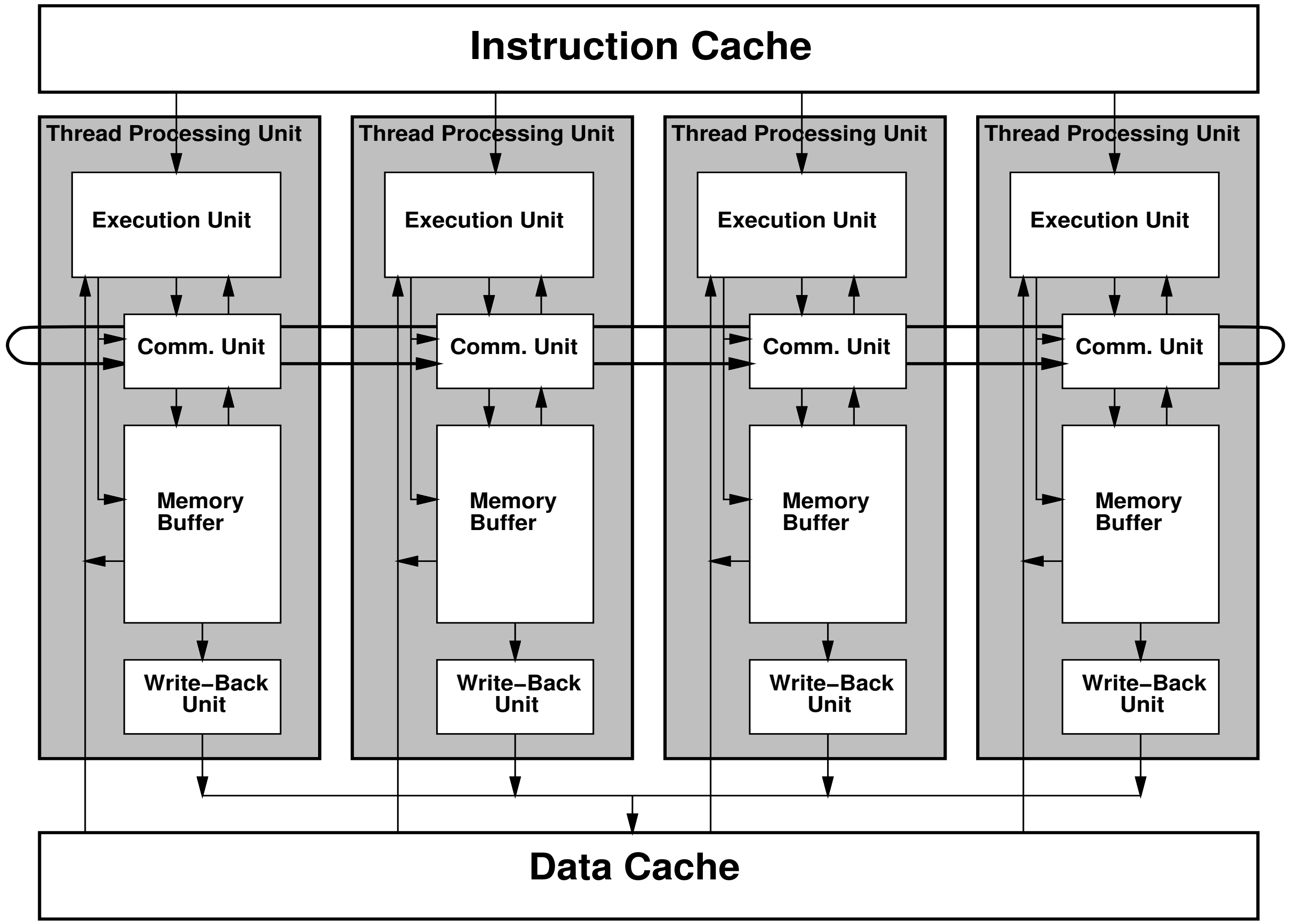
Concurrent Multi-threaded Architectures (CMA's)

- Support multiple threads (multiple PC's)
  - Handle loop-level parallelism more naturally
  - Not limited to innermost loops
- Support thread-level speculation
  - Handle Do-while loops
- Support run-time dependence checking (Superthreading) or data speculation (Multiscalar)
  - Solve pointer/alias problems
  - Allow fast communication/synchronization
- Support instruction-level parallelism in each thread
  - leverage existing superscalar techniques

Comparison of Concurrent Multi-threaded Architectures

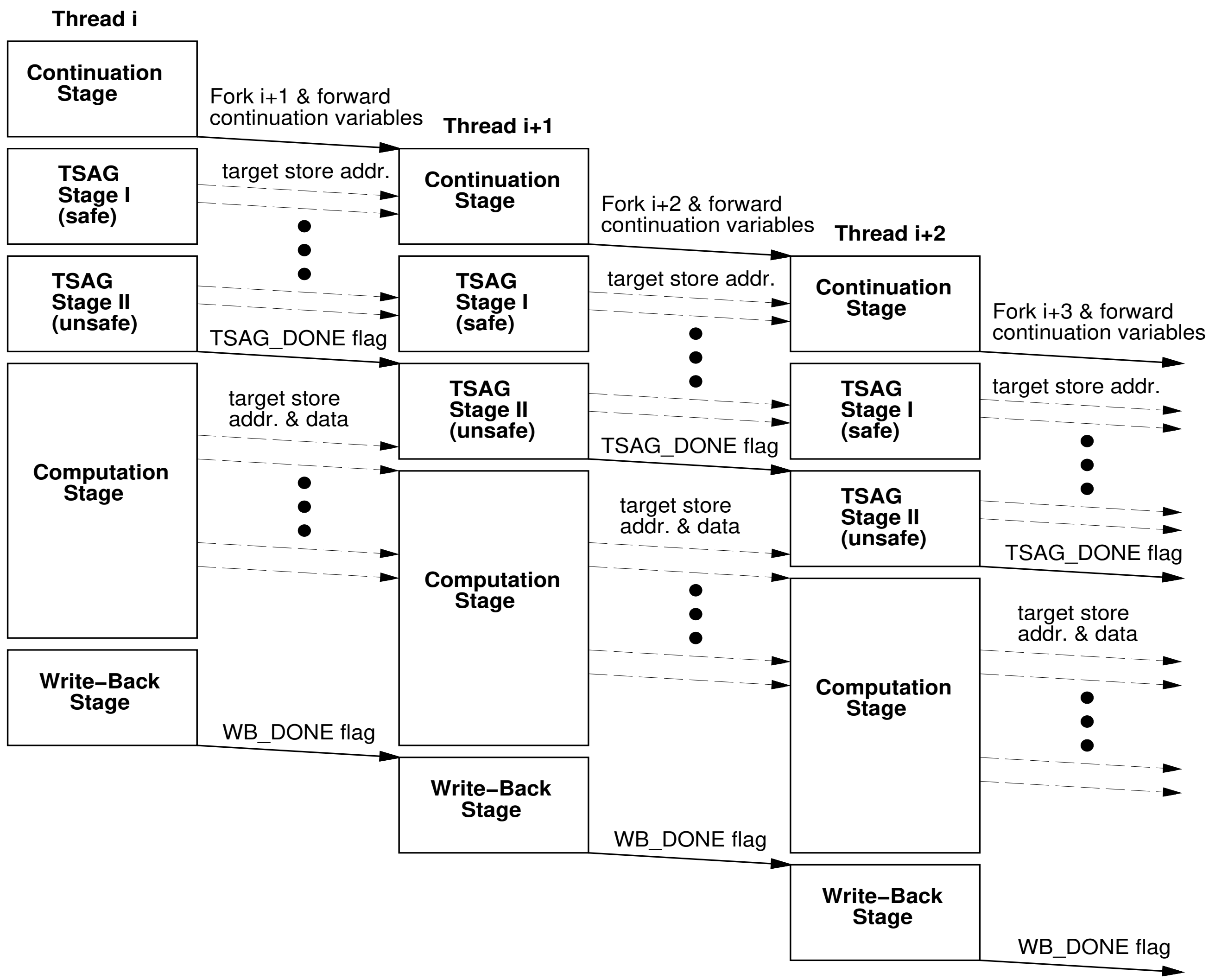
	XIMD	M-machine	Multiscalar	SPSM	Superthread
Control speculation	no	no	yes	yes	yes
Data speculation	no	no	yes	yes	no
Memory buffering	no	no	load/store	load/store	store
Run-time data dependence check	no	no	yes (squash)	yes (squash)	yes (enforce)
Data transfer between threads	register/ memory	register/ memory	register/ memory	no	memory
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



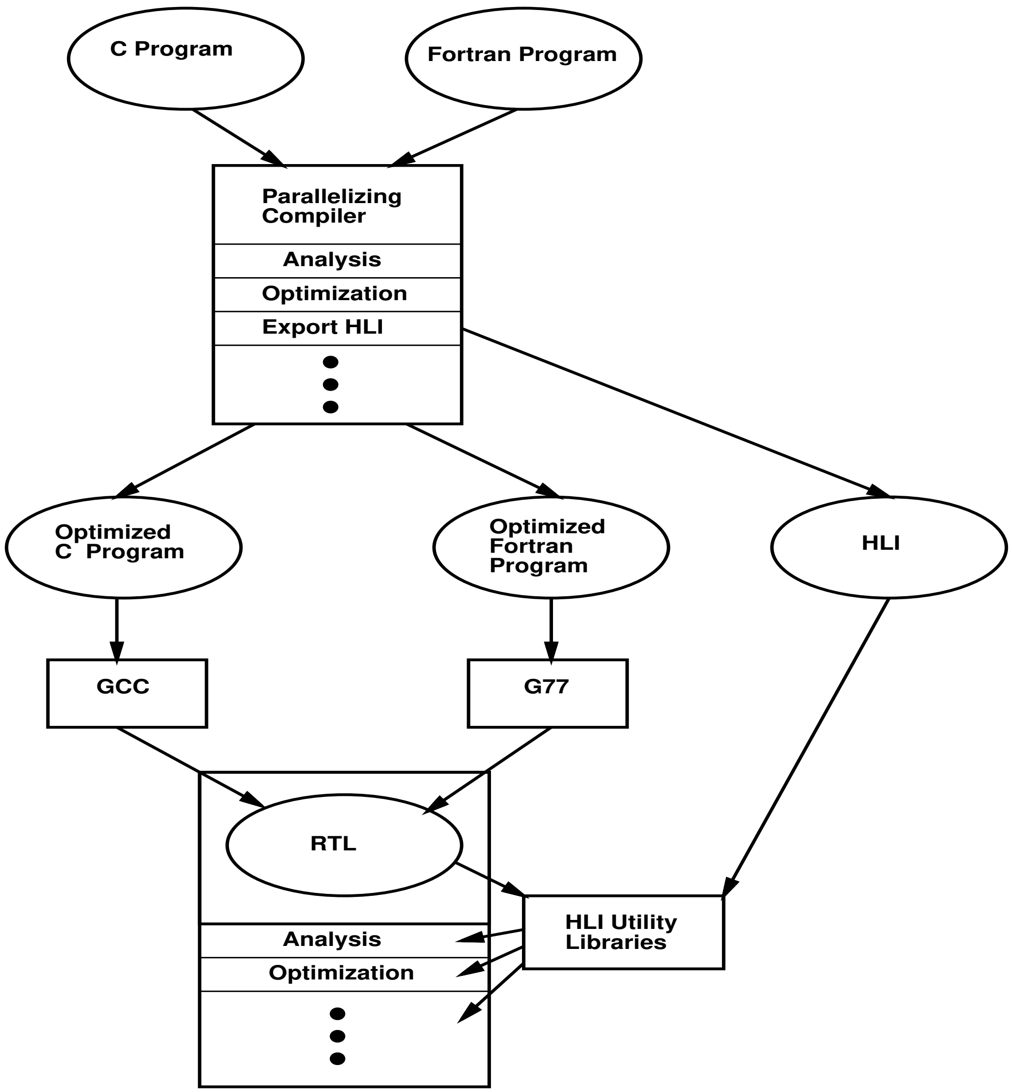
Compiler Challenges For CMA's

- 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 syn-chronization

Compiler Challenges For CMA (Cont.)

- Need to exploit both loop-level and instruction-level parallelism
  - Need integrated compilers
- 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 section for order-irrelevant operations
- Memory buffering in the main memory

Conversion of Data Speculation to Control Speculation

```
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

```
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

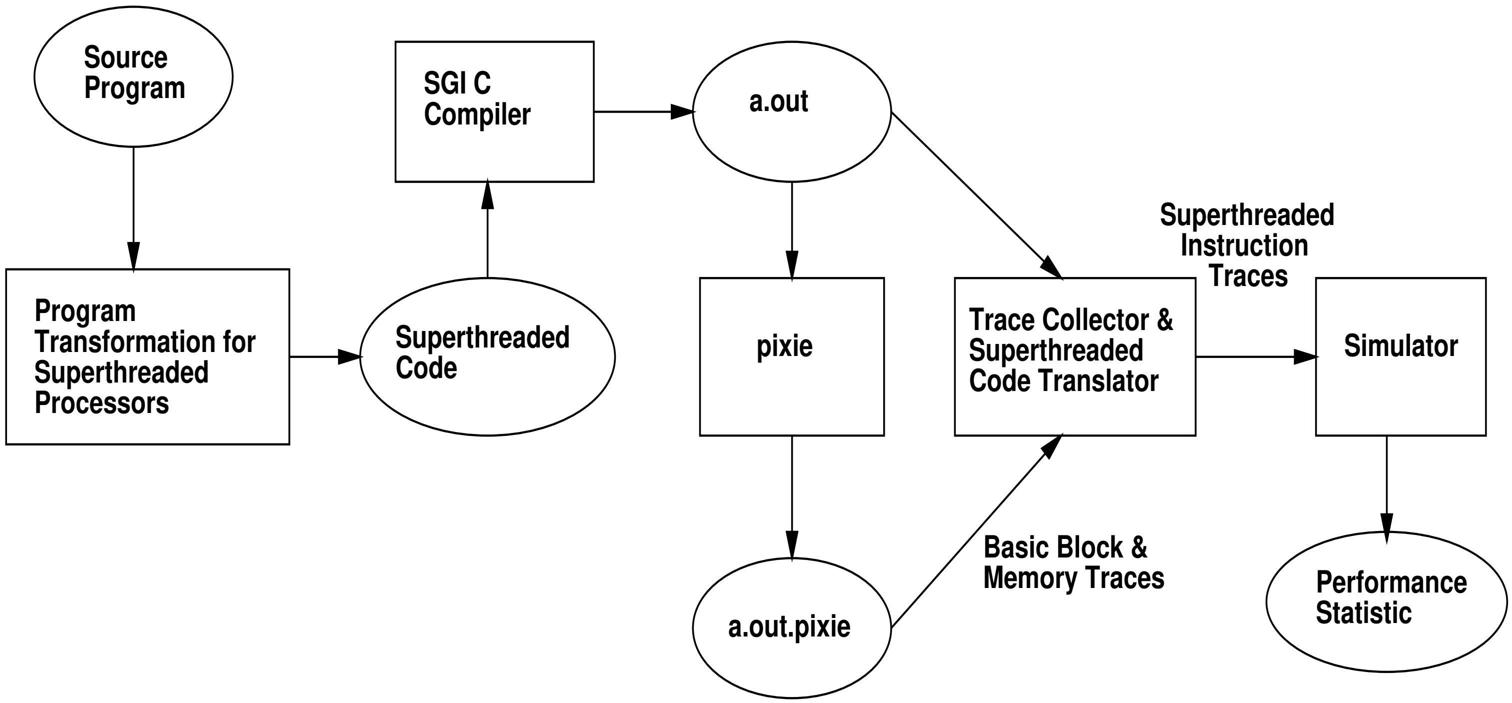
```
for (insn = f; insn; insn = NEXT_INSN (insn)) {
   . . .
   /* Scan the instructions and record each time it would
      save code to put a certain register in a certain class.  */
   . . .

   /* 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	m88ksim	gcc	compress	li	ijpeg	wc	cmp
Function inlining			X	X			
Induction	X				X	X	X
Loop unrolling						X	X
Stmt. reordering		X	X	X		X	
Data speculation		X	X	X			
Distributed heap		X		X			
Critical section		X				X	
Main Mem. buffer		X					

Simulation Environment



Simulation Configuration

- Superthreaded processor simulator
  - 4 or 8 thread processing units
  - Each thread processing unit
    - => Single-instruction issue per cycle
    - => In-order instruction execution
    - => Forward one target store entry per cycle
    - => One-cycle delay for loads and branches
- RISC processor simulator
  - Single-instruction issue per cycle
  - In-order instruction execution
  - One-cycle delay for loads and branches

Benchmarks

Program	m88ksim	gcc	compress	li	ijpeg	wc	cmp
Routine 1	killtime	cse_main	compress	mark	jpeg_fdct_islow	wc	block_compare_and_count
Routine 2	ckbrkpts	regclass	decompress	sweep	jpeg_idct_islow		
Routine 3	alignd			xlygetvalue	fullsize_smooth_downsample		
Routine 4	test_issue			xlxgetvalue	forward_DCT		
Routine 5					rgb_ycc_convert		
Routine 6					h2v2_smooth_downsample		
Routine 7					h2v2_merged_upsample		
% of prog simulated	43.6%	31.62%	91.60%	52.57%	54.44%	99.94%	99.85%

Simulation Results - IPC's

Useful Instruction count Per Cycle (IPC)

Program	m88ksim		gcc		compress		li		ijpeg		wc		cmp	
# of Thrds	4	8	4	8	4	8	4	8	4	8	4	8	4	8
Routine 1	3.03	3.23	1.16	1.23	1.08	1.08	0.95	0.95	3.42	5.92	2.61	3.78	3.20	5.90
Routine 2	1.72	1.72	1.85	2.95	1.21	1.21	1.82	1.81	2.69	3.44				
Routine 3	2.19	2.07					0.96	0.96	2.55	2.56				
Routine 4	1.13	1.13					0.90	0.90	3.24	4.87				
Routine 5									3.44	5.75				
Routine 6									3.52	6.33				
Routine 7									3.11	5.71				
Weighted Average	2.34	2.34	1.22	1.34	1.13	1.13	1.05	1.05	3.14	4.57	2.61	3.78	3.20	5.90

Simulation Results - Speedups

Speedups over a single-threaded RISC processor

Program	m88ksim		gcc		compress		li		ijpeg		wc		cmp	
# of Thrds	4	8	4	8	4	8	4	8	4	8	4	8	4	8
Routine 1	2.66	2.83	1.31	1.39	1.04	1.04	0.89	0.89	4.03	6.98	1.95	2.82	3.34	6.15
Routine 2	1.35	1.35	2.05	3.27	1.26	1.26	1.50	1.49	3.36	4.29				
Routine 3	2.06	1.95					0.83	0.89	1.60	1.60				
Routine 4	1.15	1.15					0.83	0.84	3.50	5.25				
Routine 5									2.74	4.57				
Routine 6									2.35	4.22				
Routine 7									2.86	5.07				
Weighted Average	2.09	2.09	1.37	1.50	1.13	1.13	0.96	0.96	2.96	4.30	1.95	2.82	3.34	6.15

Simulation Results - Memory Buffer Usage

Maximum number of memory buffer entries used in each routine

Program	m88ksim		gcc*		compress		li		ijpeg		wc		cmp	
# of Thrds	4	8	4	8	4	8	4	8	4	8	4	8	4	8
Routine 1	2	2	1	1	5	5	6	8	25	25	4	4	5	5
Routine 2	1	1	3	3	24	32	7	7	10	10				
Routine 3	11	19					1	1	9	9				
Routine 4	2	2					1	1	9	9				
Routine 5									5	5				
Routine 6									4	4				
Routine 7									16	16				

***For gcc, speculative and private data are buffered in the main memory.**

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 viable alternative to wide-issue superscalar with reasonable performance