FDA125 Advanced Parallel Programming

Run-Time Parallelization and Thread-Level Speculation

Christoph Kessler, IDA

Christoph Kessler, IDA,
Linköpings universitet, 2007.

Goal of run-time parallelization

- Typical target: **irregular loops**

  for (i=0; i<n; i++)
  a[i] = *f* (a[*g*(i)], a[*h*(i)], ...);

  - Array index expressions *g, h...* depend on run-time data
  - Iterations cannot be statically proved independent (and not either dependent with distance +1)
- **Principle:**
  At runtime, inspect *g, h* ... to find out the real dependences and compute a schedule for partially parallel execution
  - Can also be combined with speculative parallelization

Overview

- **Run-time parallelization of irregular loops**
  - DOACROSS parallelization
  - Inspector-Executor Technique (shared memory)
  - Inspector-Executor Technique (message passing)
  - Privatizing DOALL Test *
- **Speculative run-time parallelization of irregular loops ***
  - LRPD Test *
- **General Thread-Level Speculation**
  - Hardware support *

* = not yet covered in this lecture. See the references.

FDA125 Advanced Parallel Programming

Runtime parallelization

DOACROSS Parallelization

DOACROSS Parallelization

- Useful if dependence distances are unknown, but often > 1
- Allow independent subsequent loop iterations to overlap
- Bilateral synchronization
- Simple example for shared memory:

```
for ( i=0; i<n; i++)
    a[i] = f ( a[ g(i) ], ... );

sh flag done[n];  // flag array, initialized to 0 (not done);
forall ( i=0; i<n; i++) {   // spawn n threads, one per iteration
   if (g(i)<i)  wait until done[ g(i) ] );
   a[i] = f ( a[ g(i) ], ... );
   set( done[i] );
}
```

A kind of run-time software pipelining

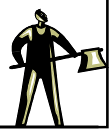
FDA125 Advanced Parallel Programming

Runtime parallelization

Inspector-Executor Technique

Inspector-Executor Technique (1)

- Compiler generates 2 pieces of customized code for such loops:
- **Inspector**
  - calculates values of index expression by simulating whole loop execution
    - typically, based on sequential version of the source loop (some computations could be left out)
  - computes implicitly the real iteration dependence graph
  - computes a parallel schedule as (greedy) wavefront traversal of the iteration dependence graph in topological order
    - all iterations in same wavefront are independent
    - schedule depth = #wavefronts = critical path length
- **Executor**
  - follows this schedule to execute the loop

Inspector-Executor Technique (2)

- **Source loop:**

```
for ( i=0; i<n; i++)
     a[i]  =  f ( a[ g(i) ], a[ h(i) ], ... );
```

- **Inspector:**

```
int wf[n];  // wavefront indices
int depth = 0;
for (i=0; i<n; i++)
   wf[i] = 0;   // init.
for (i=0; i<n; i++) {
   wf[i] = max ( wf[ g(i) ], wf[ h(i) ], ... ) + 1;
   depth = max ( depth, wf[i] );
}
```

- Inspector considers only flow dependences (RAW), anti- and output dependences to be preserved by executor

Inspector-Executor Technique (3)

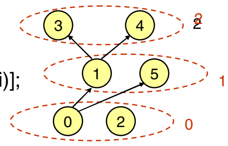
- **Example**:

```
for (i=0; i<n; i++)
     a[i] = ... a[ g(i) ] ...;
```

i	0	1	2	3	4	5
g(i)	2	0	2	1	1	0
wf[i]	0	1	0	2	2	1
g(i)<i ?	no	yes	no	yes	yes	yes

- **Executor**:

```
float aold[n];  // buffer array
aold[1:n] = a[1:n];
for (w=0; w<depth; w++)
   forall (i, 0, n, #)  if (wf[i] == w)  {
      a1 = (g(i) < i)? a[g(i)] : aold[g(i)];
      ...  // similarly, a2 for h etc.
      a[i] =  f ( a1, a2, ... );
   }
```



iteration (flow) dependence graph

Inspector-Executor Technique (4)

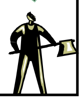
Problem: Inspector remains sequential – no speedup

Solution approaches:

- Re-use schedule over subsequent iterations of an outer loop if access pattern does not change
  - amortizes inspector overhead across repeated executions
- Parallelize the inspector using doacross parallelization [Saltz,Mirchandaney'91]
- Parallelize the inspector using sectioning [Leung/Zahorjan'91]
  - compute processor-local wavefronts in parallel, concatenate
  - trade-off schedule quality (depth) vs. inspector speed
  - Parallelize the inspector using bootstrapping [Leung/Z.'91]
  - Start with suboptimal schedule by sectioning, use this to execute the inspector → refined schedule

Inspector-Executor Technique (5) - DMS

- Global address space (GAS) languages for DMS (HPF, UPC, NestStep, Co-Array Fortran, ...)
  - Compiler must insert necessary **Send** / **Recv** operations to move data from owning to reading processor
  - Necessary even for (irregular) *parallel* loops (iterations are statically asserted to be independent, e.g. by user directive)
- Can use inspector-executor method for run-time scheduling of communication in irregular loops
- **Inspector**:
  - determines communication map + reverse map (schedule): Who has to send which owned elements to whom
  - allocate buffer for received elements; adapt access functions
- **Executor**:
  - communicates according to schedule
  - executes loop

Inspector-Executor Technique (5) - DMS

- **Example**:

```
forall (i, 0, 12, #)
     y[i]  =  y[i] + a [ ip[i] ] * x[i]
```

  - y[1:n], a[1:n], ip[1:n], x[1:n] aligned and block-distributed across 3 processors P0, P1, P2
  - Compiler applies owner-computes rule

i	0	1	2	3	4	5	6	7	8	9	10	11
owner of y[i]	P0	P0	P0	P0	P1	P1	P1	P1	P2	P2	P2	P2
ip[i]	1	5	6	10	0	3	4	8	10	5	4	9
owner of a [ip[i]]	P0	P1	P1	P2	P0	P0	P1	P2	P2	P1	P1	P2

Inspector-Executor Technique (6) - DMS

- **Inspector step 1:**
  construct communication map
  (here, in parallel)

dest	source	data	local buffer area (private)
P0	P1	a[5], a[6]	lb[0:1]
	P2	a[10]	lb[2]
P1	P0	a[0], a[3]	lb[0:1]
	P2	a[8]	lb[2]
P2	P1	a[4], a[5]	lb[0:1]

Inspector-Executor Technique (7) - DMS

- **Inspector step 2:**
  construct reverse communication map
  (communication schedule)

source	dest	data	remote buffer area
P0	P1	a[0], a[3]	lb[0:1]
P1	P0	a[5], a[6]	lb[0:1]
P1	P2	a[4], a[5]	lb[0:1]
P2	P0	a[10]	lb[2]
P2	P1	a[9]	lb[2]

Inspector-Executor Technique (8) - DMS

Inspector, step 3:

Construct modified access functions
(represented as local table of pointers)

i	0	1	2	3	4	5	6	7	8	9	10	11
owner of y[i]	P0	P0	P0	P0	P1	P1	P1	P1	P2	P2	P2	P2
ip[i]	1	5	6	10	0	3	4	8	10	5	4	9
owner of a [ip[i]]	P0	P1	P1	P2	P0	P0	P1	P2	P2	P1	P1	P2
accesstable [i] = where to find a[ip[i]] in local memory	a +1	lb +0	lb +1	lb +2	lb +0	lb +1	a +0	lb +2	a +2	lb +0	lb +1	a +1

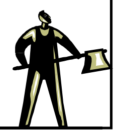
Remark: Communication maps and address tables can be reused
if ip[:] does not change between subsequent executions of the source loop.

Inspector-Executor Technique (9) – DMS

- **Executor**:

```
// send data according to reverse communication map:
for each Pj in dest
    send requested a[:] elements to Pj
// receive data according to communication map:
for each Pi in source
    recv a[:] elements, write to respective lb entries
// Remark:  the above part can be skipped in subsequent
//      executions of the executor if ip[] and a[] do not change.

// execute loop with modified access function:
forall (i, 0, 12, #)
   y[i] = y[i] + *( accesstable[i] ) * x[i];
```



Some references on run-time parallelization

- R. Cytron: Doacross: Beyond vectorization for multiprocessors. Proc. ICPP-1986
- D. Chen, J. Torrellas, P. Yew: An Efficient Algorithm for the Run-time Parallelization of DOACROSS Loops, Proc. IEEE Supercomputing Conference, Nov. 2004, IEEE CS Press, pp. 518-527
- R. Mirchandaney, J. Saltz, R. M. Smith, D. M. Nicol, K. Crowley: Principles of run-time support for parallel processors, Proc. ACM ICS-88 Int. Conf. on Supercomputing, July 1988, pp. 140-152.
- J. Saltz and K. Crowley and R. Mirchandaney and H. Berryman: Runtime Scheduling and Execution of Loops on Message Passing Machines, Journal on Parallel and Distr. Computing 8 (1990): 303-312.
- J. Saltz, R. Mirchandaney: The preprocessed doacross loop. Proc. ICPP-1991 Int. Conf. on Parallel Processing.
- S. Leung, J. Zahorjan: Improving the performance of run-time parallelization. Proc. ACM PPoPP-1993, pp. 83-91.
- Lawrence Rauchwerger, David Padua: The Privatizing DOALL Test: A Run-Time Technique for DOALL Loop Identification and Array Privatization. Proc. ACM ICS-94 Int. Conf. on Supercomputing, July 1994, pp. 33-45.
- Lawrence Rauchwerger, David Padua: The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. Proc. ACM SIGPLAN PLDI-95, 1995, pp. 218-232.

Thread-Level Speculation

Christoph Kessler, IDA,
Linköpings universitet, 2007.

Speculative execution

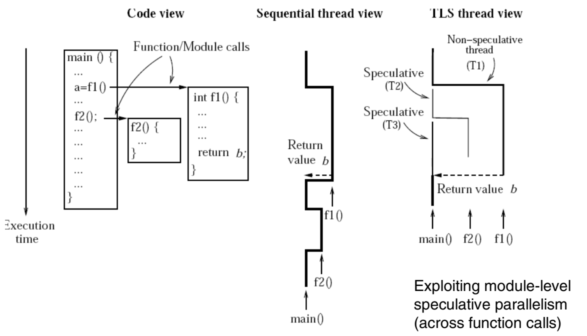
- For automatic parallelization of sequential code where dependences are hard to analyze statically
- Works on a **task graph**
  - constructed implicitly and dynamically
- **Speculate on:**
  - control flow, data independence, synchronization, values

    We focus on thread-level speculation (TLS) for CMP/MT processors. Speculative ILP is not considered here.
- **Task:**
  - **statically:** Connected, single-entry subgraph of the control-flow graph
    - Basic blocks, loop bodies, loops, or entire functions
  - **dynamically:** Contiguous fragment of dynamic instruction stream within static task region, entered at static task entry

Speculative execution of tasks

- **Speculation on inter-task control flow**
  - After having assigned a task, predict its successor task and start it speculatively
- **Speculation on data independence**
  - For inter-task memory data (flow) dependences
    - conservatively: await write (memory synchronization, message)
    - speculatively: hope for independence and continue (execute the load)
- **Roll-back** of speculative results on mis-speculation (expensive)
  - When starting speculation, state must be buffered
  - Squash an offending task and all its successors, restart
- **Commit speculative results** when speculation resolved to correct
  - Task is retired

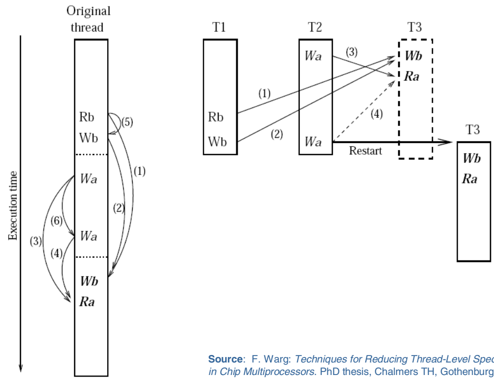
TLS Example



Exploiting module-level speculative parallelism (across function calls)

Source: F. Warg: *Techniques for Reducing Thread-Level Speculation Overhead in Chip Multiprocessors.* PhD thesis, Chalmers TH, Gothenburg, June 2006.

Data dependence problem in TLS



Source: F. Warg: *Techniques for Reducing Thread-Level Speculation Overhead in Chip Multiprocessors.* PhD thesis, Chalmers TH, Gothenburg, June 2006.

Selecting Tasks for Speculation

- **Small tasks:**
  - too much overhead (task startup, task retirement)
  - low parallelism degree
- **Large tasks:**
  - higher misspeculation probability
  - higher rollback cost
  - many speculations ongoing in parallel may saturate the resources
- **Load balancing issues**
  - avoid large variation in task sizes
- Traversal of the program's control flow graph (CFG)
  - Heuristics for task size, control and data dep. speculation

TLS Implementations

- **Software-only speculation**
  - for loops [Rauchwerger, Padua '94, '95]
  - ...
- **Hardware-based speculation**
  - Typically, integrated in cache coherence protocols
  - Used with multithreaded processors / chip multiprocessors for automatic parallelization of sequential legacy code
  - If source code available, compiler may help e.g. with identifying suitable threads

Some references on speculative execution / parallelization

- T. Vijaykumar, G. Sohi: Task Selection for a Multiscalar Processor. Proc. MICRO-31, Dec. 1998.
- J. Martinez, J. Torrellas: Speculative Locks for Concurrent Execution of Critical Sections in Shared-Memory Multiprocessors. Proc. WMPI at ISCA, 2001.
- F. Warg and P. Stenström: Limits on speculative module-level parallelism in imperative and object-oriented programs on CMP platforms. Pr. IEEE PACT 2001.
- P. Marcuello and A. Gonzalez: Thread-spawning schemes for speculative multithreading. Proc. HPCA-8, 2002.
- J. Steffan et al.: Improving value communication for thread-level speculation. HPCA-8, 2002.
- M. Cintra, J. Torrellas: Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors. HPCA-8, 2002.
- Fredrik Warg and Per Stenström: Improving speculative thread-level parallelism through module run-length prediction. Proc. IPDPS 2003.
- F. Warg: *Techniques for Reducing Thread-Level Speculation Overhead in Chip Multiprocessors.* PhD thesis, Chalmers TH, Gothenburg, June 2006.
- T. Ohsawa et al.: Pinot: Speculative multi-threading processor architecture exploiting parallelism over a wide range of granularities. Proc. MICRO-38, 2005.