A Compiler Framework to Support Speculative Multi-Core Processors for General-Purpose Applications

Pen-Chung Yew
游本中

Department of Computer Science and Engineering
University of Minnesota

Department of Computer Science and Engineering

http://www.cs.umn.edu/Agassiz

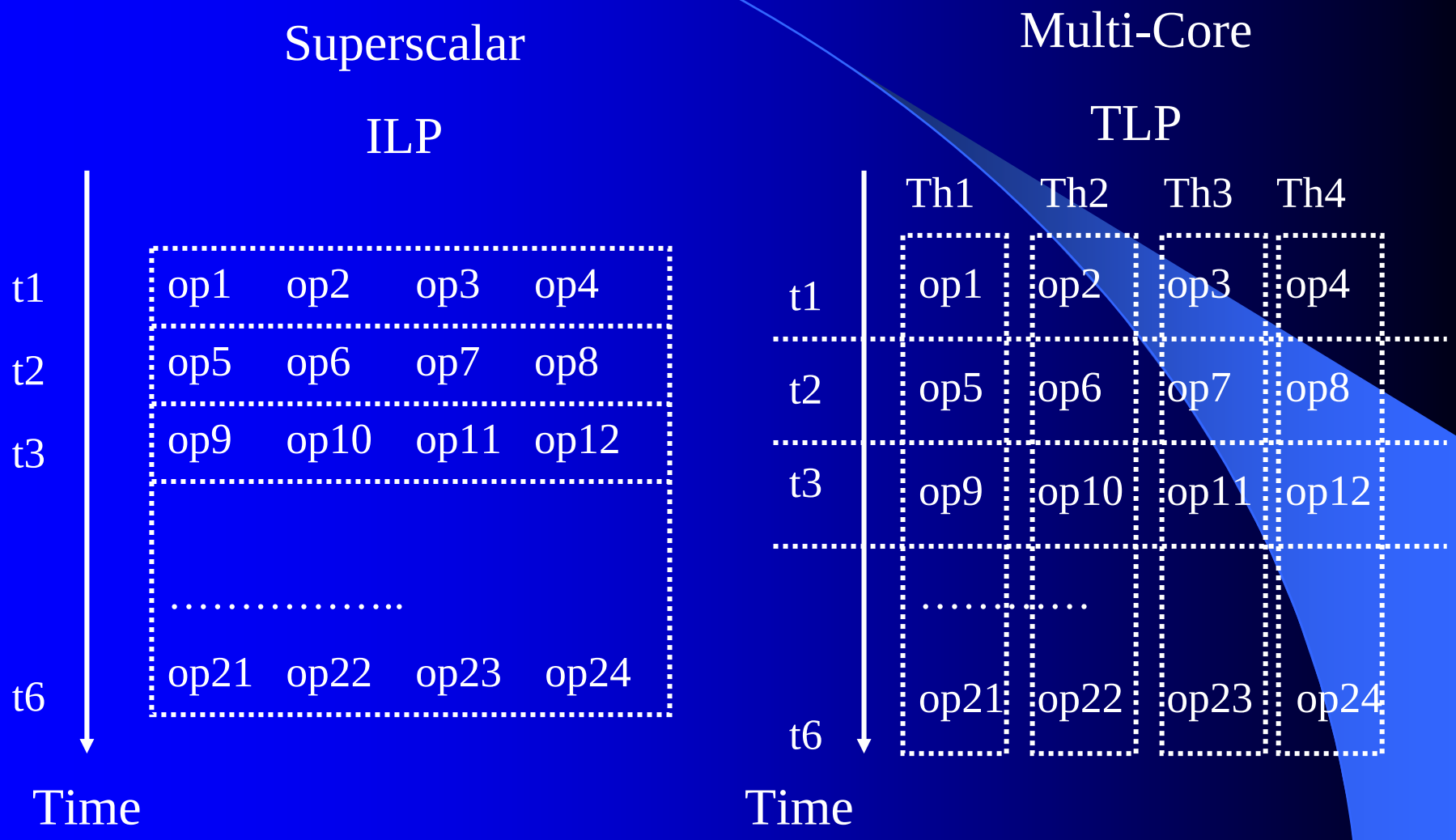
Outline

- Speculative multi-core processors and general-purpose applications
- A compiler framework to support speculative execution and optimizations
- Speculative optimizations for single-core processors
- Speculative optimizations for multi-core processors
- Conclusions

Multi-Core Processors on Technology Road Map

- Multi-core processors on Intel's roadmap
- SMPs have been around for a long time. What is new for multi-core?
- Why general-purpose applications now?
- Use *thread-level parallelism* (TLP) to improve *instruction-level parallelism* (ILP)

Use TLP to Support ILP



TLP Challenges in General-Purpose Applications

- Mostly Do-while loops
  - Need thread-level *control speculation*
- Parallelism exists mostly in outer loops
  - Not good for VLIW (i.e. software pipelining) or vector processing => need thread-level support
- Pointers complicate alias and data dependence analysis
  - Need runtime support for disambiguation and *data speculation*
- Many *small* loops and *doacross* loops
  - Need fast and low overhead communication
- Small basic blocks – need to exploit both ILP and TLP

Need new approaches to apply parallel processing for such applications!!

Speculation: Breaking Program Dependency

- *Speculation* is an effective approach to break *dependences*
  - Optimize program execution by ignoring *infrequent* data dependence edges, or taking *predicted* paths
  - *Check* possible violation (mis-speculation) at runtime
  - *Recover* if violation occurs

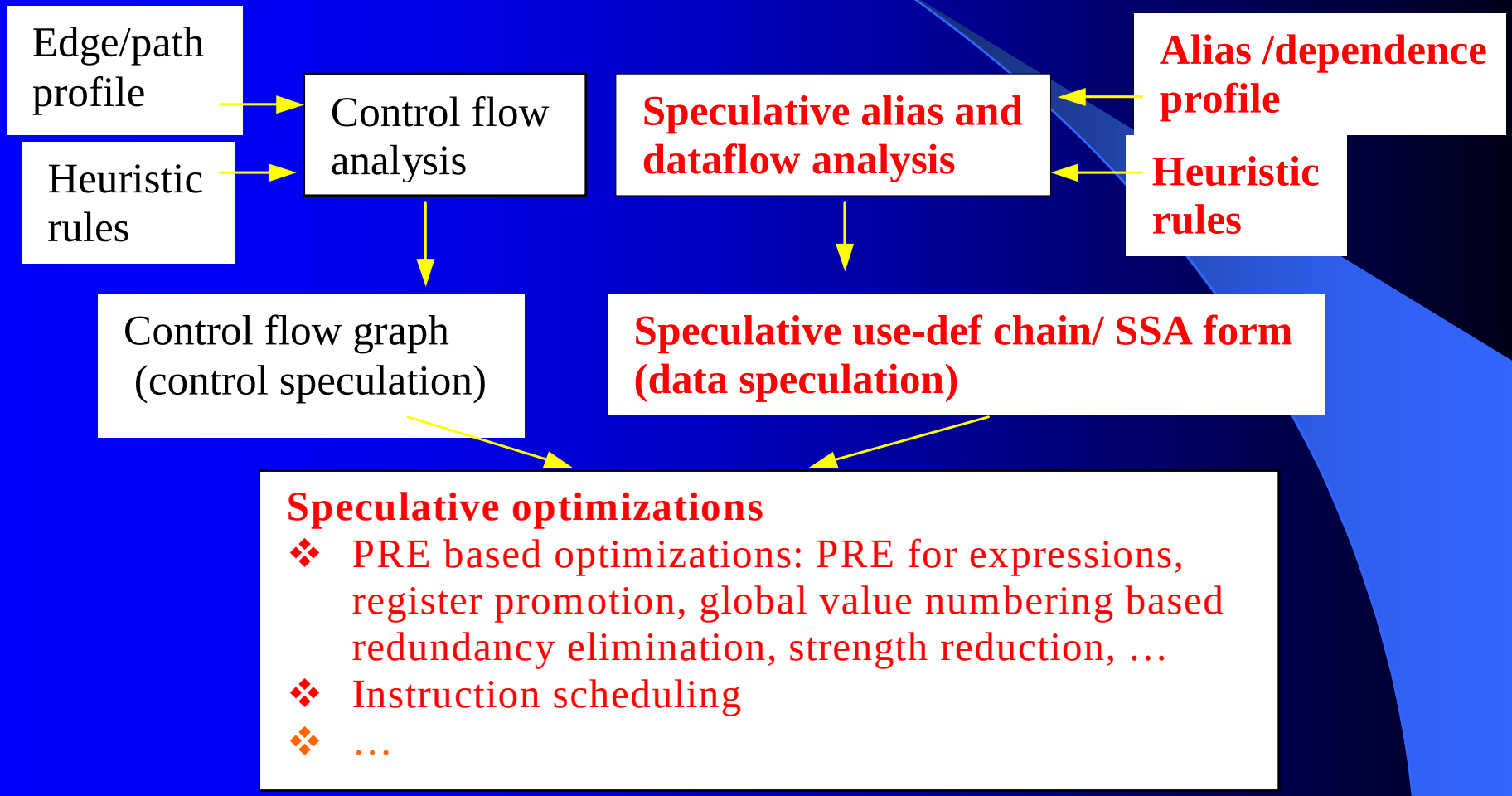
Speculation on Intel IA64

- Both *control* and *data* speculation are supported on Intel IA64
  - Special instructions and hardware are provided
  - ld.s, ld.a, ld.sa and ld.c, chk.a, chk.s
- Memory *load* operation is targeted for speculation
  - Memory *delay* is usually the bottleneck of performance
  - Memory *load* is usually the start of speculative operations

Outline

- Speculative multi-core processors and general-purpose applications
- **A compiler framework to support speculative execution and optimizations**
- Speculative optimizations for single thread
- Speculative optimizations for multi-threaded processors
- Conclusions

A Compiler Framework: Intel Open Research Compiler (ORC)



Main Compiler Components in the Framework

- Efficient alias and data dependence profiling tools, or heuristic algorithms
- Annotating profiled information in compiler
- Speculative optimizations
- Recovery code generation

Alias and Data Dependence Profiling

- Instrumentation-based alias profiling
- Instrumentation-based data dependence profiling
- Techniques to reduce profiling overhead

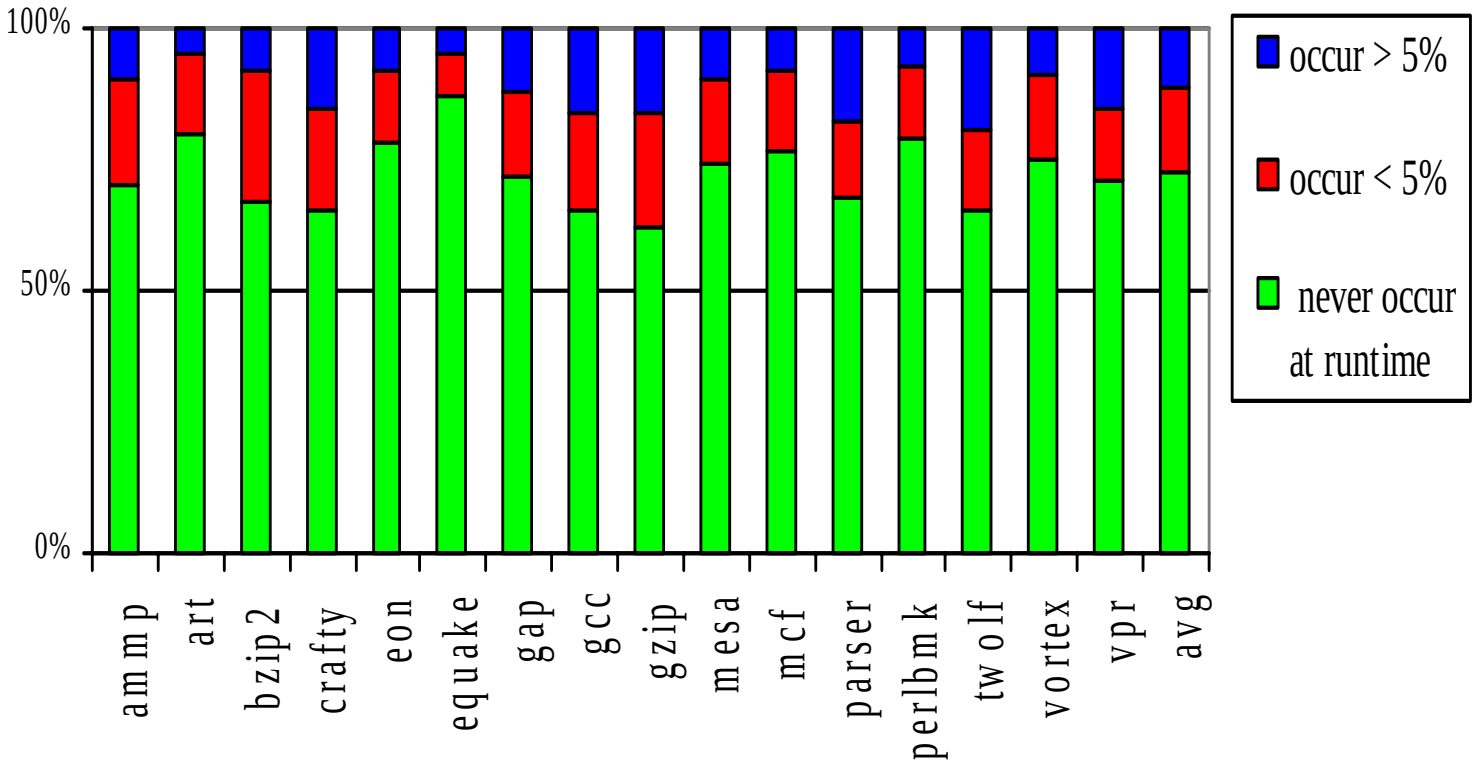
• T.Chen et al, *Data Dependence Profiling for Speculative Optimizations*, Proc. Of Int'l Conf on Compiler Construction (CC), March 2004

• T.Chen et al, *An Empirical Study on the Granularity of Pointer Analysis in C programs*, Proc. 15th Workshop on Languages and Compilers for Parallel Computing (LCPC15), August 2002

Crucial Considerations

- Program coverage: 10/90 rule
  - uncovered regions => use compiler analysis results or heuristic rules
- Input sensitivity
- Profiling overhead (space and time)
- Using alias and data dependence profiles is inherently *speculative* => need hardware support for correct execution

Alias Profiling vs. Static Analysis

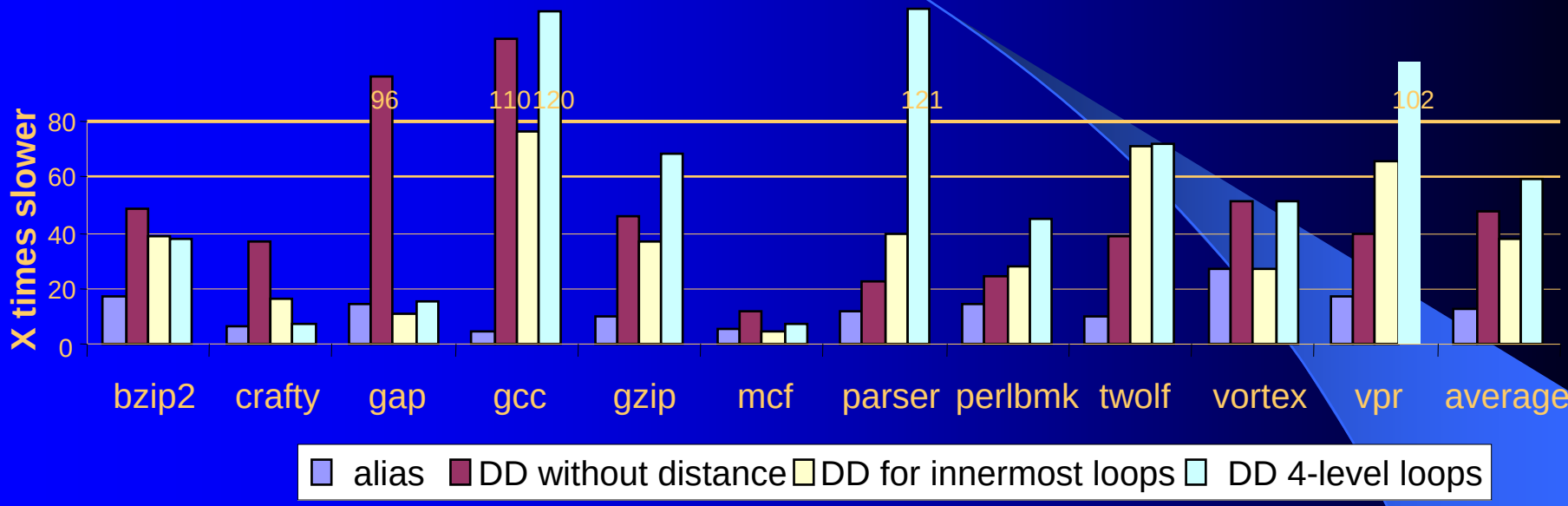


Most possible aliases reported by compiler do not occur at runtime

Data Dependence Profiling

- Data dependence edges among memory references and function calls
- Detailed information
  - *type*: flow, anti, output, or input
  - *probability*: frequency of occurrence
- When loops are targeted
  - *dependence distance*: limited

Overhead of DD Profiling



- Compiler: ORC version 2.0
- Machine: Itanium2, 900 MHz and 2G memory
- Benchmarks: SPEC CPU2000 Int
- Instrumentation optimization has been done

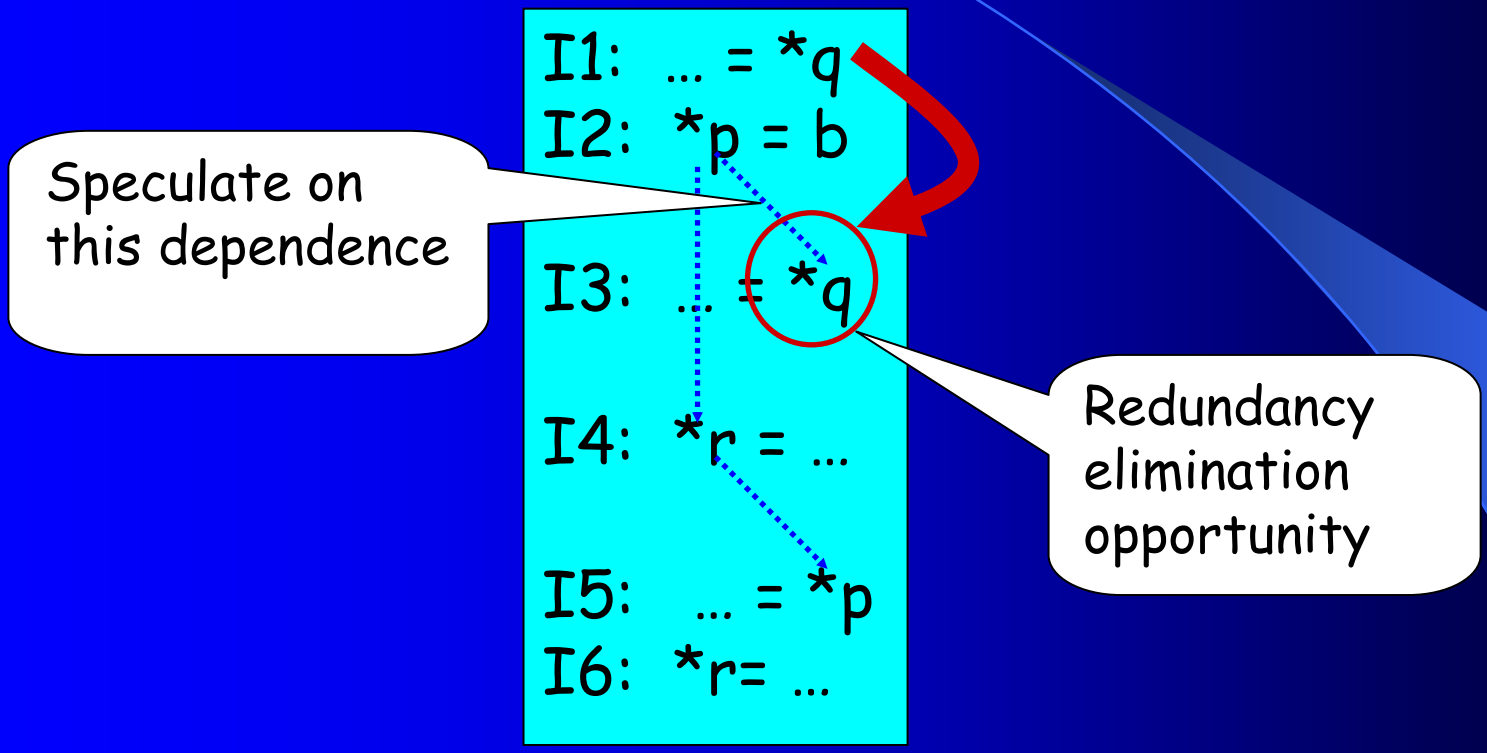
Techniques to Reduce Profiling Overhead

- Reduce the space and time requirements by *hash table*
  - Larger granularity of address
  - Smaller iteration counter
- Sampling
  - *Sample* the snap shots of procedures or loops instead of individual references
  - Use *instrumentation-based sampling* framework
    - Switch at procedures or loops

Outline

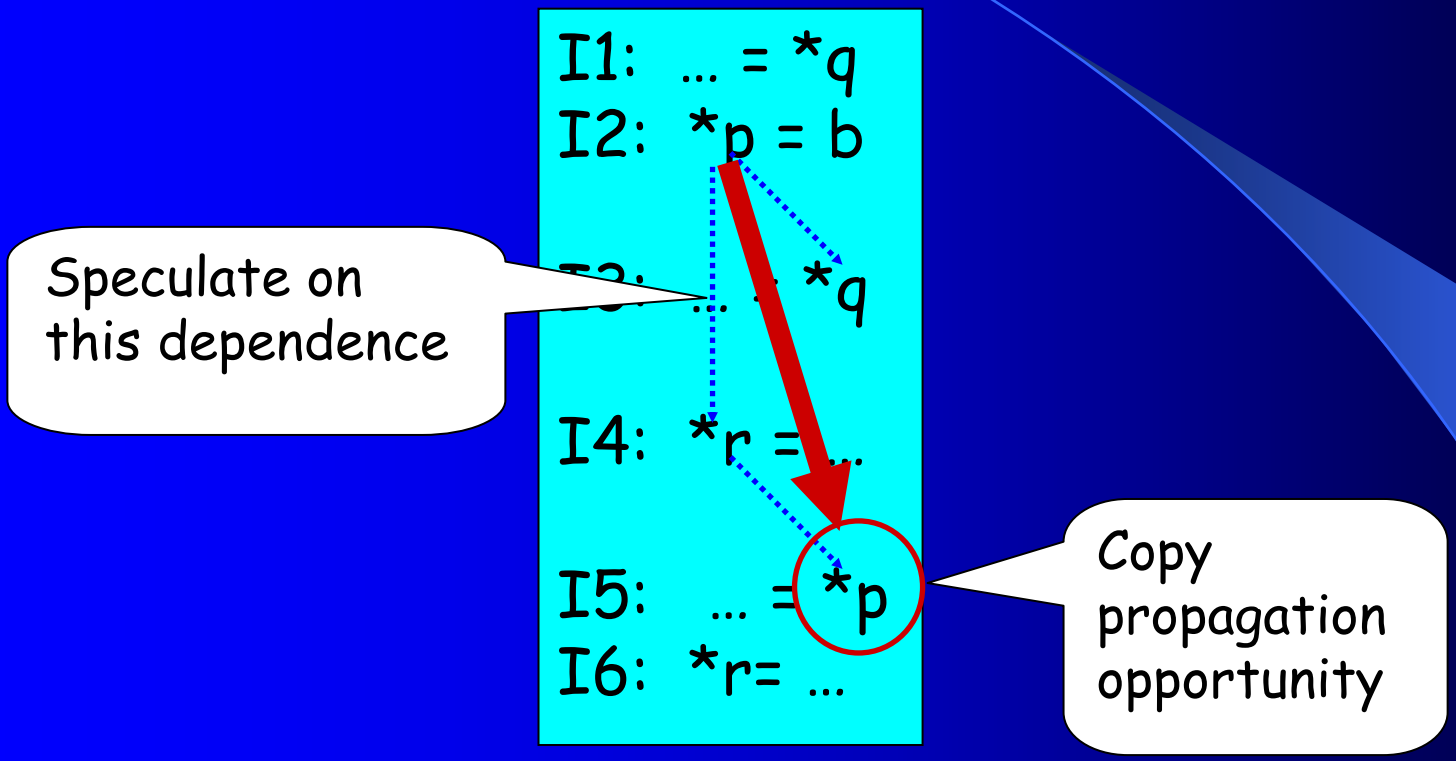
- Speculative multi-core processors and general-purpose applications
- A compiler framework to support speculative execution and optimizations
- **Speculative optimizations for single thread**
- Speculative optimizations for multi-threaded processors
- Conclusions

Speculating on Data Dependence



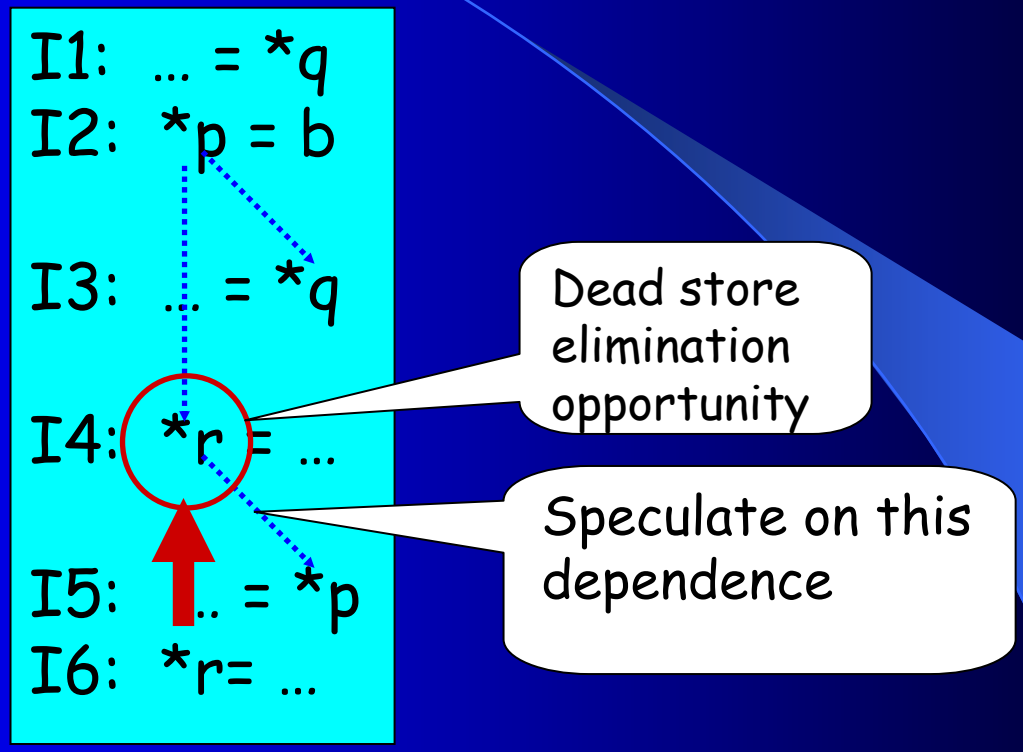
More speculative optimizations

Speculate on Data Dependences



More speculative optimizations

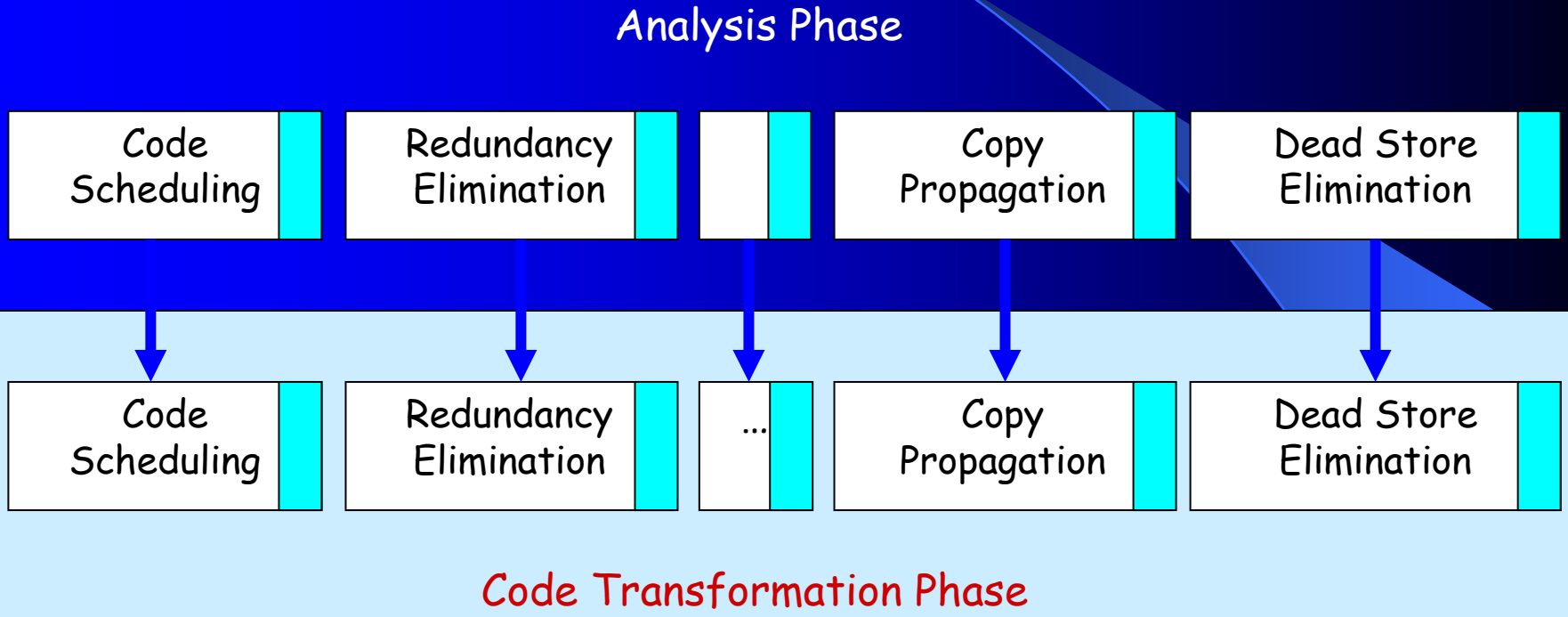
Speculate on Data Dependences



More speculative optimizations

Integrate Data Speculation in Traditional Optimizations

Optimization extension to handle data dependence speculation



Traditional optimizations need to be patched to handle data speculation

Example of Speculative SSA Form

$a_1 = \ldots$
$*p_1 = 4$
$a_2 \leftarrow \chi(a_1)$
$b_2 \leftarrow \chi(b_1)$
$v_2 \leftarrow \chi(v_1)$
$\ldots = a_2$
$a_3 = 4$
$\mu(a_3), \mu(b_2), \mu(v_2$
$\ldots = *p_1$

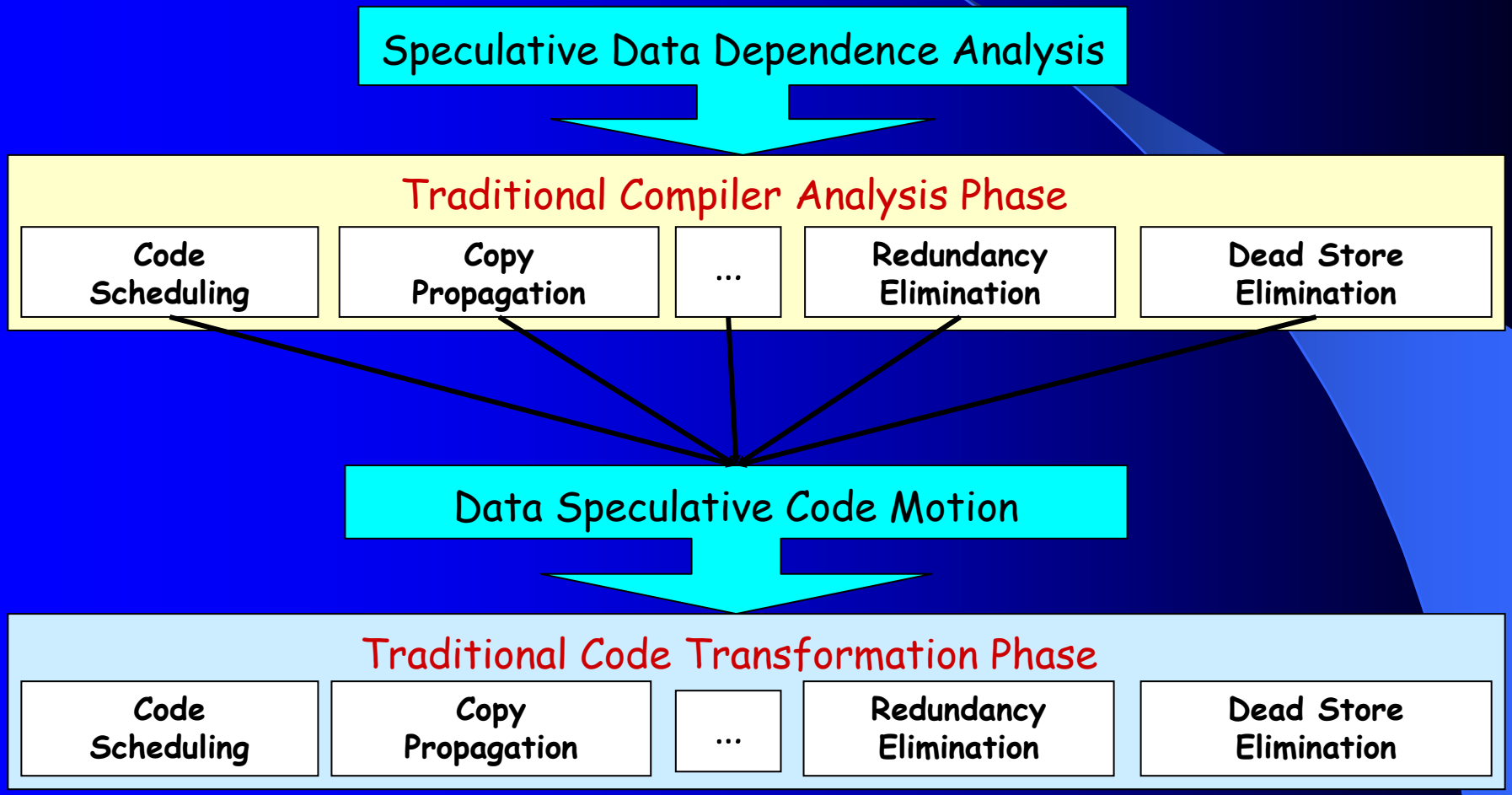
(a) traditional SSA graph

$a_1 = \ldots$
$*p_1 = 4$
$a_2 \leftarrow \chi(a_1)$
$b_2 \leftarrow \chi_s(b_1)$
$v_2 \leftarrow \chi(v_1)$
$\ldots = a_2$
$a_3 = 4$
$\mu(a_3), \mu_s(b_2), \mu(v_2)$
$\ldots = *p_1$

(b) speculative SSA graph

The points-to set of pointer *p* obtained by alias profiling is {b}

Improved Speculative Optimizations Framework



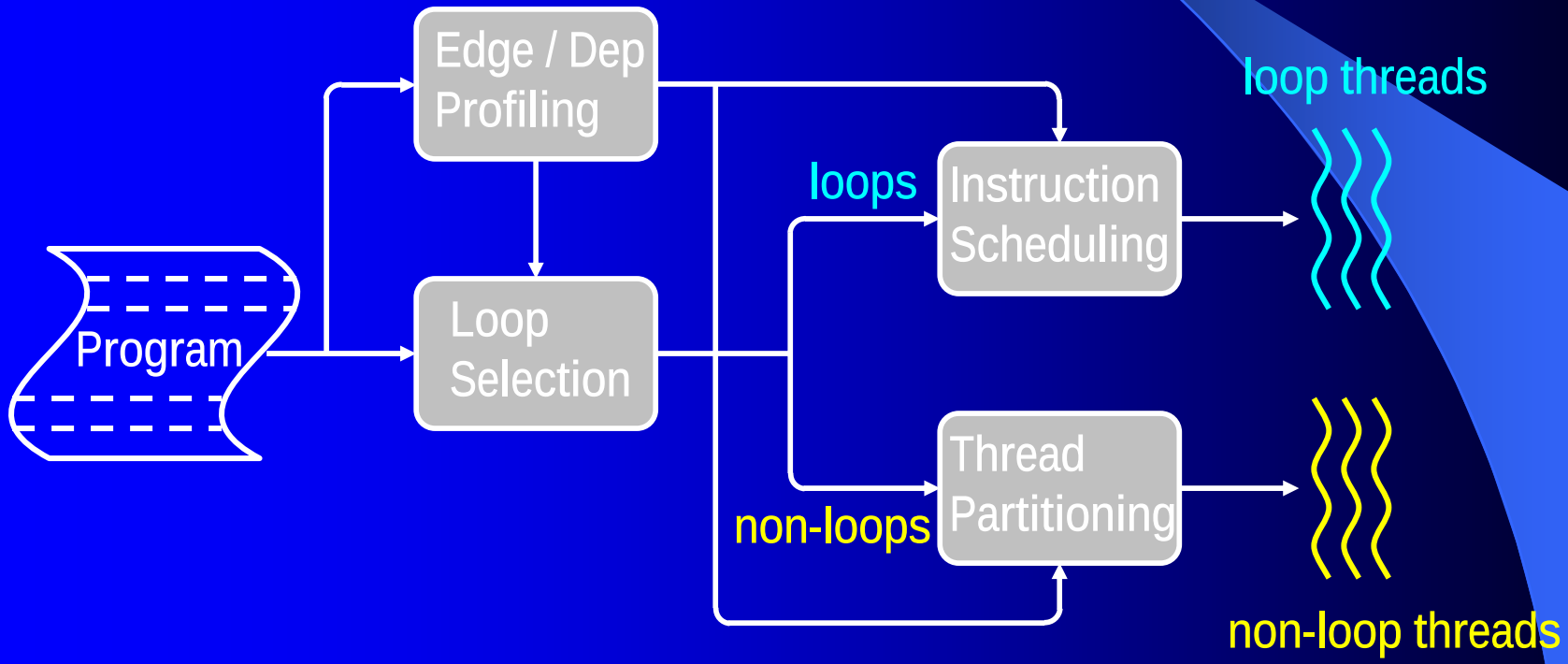
Outline

- Speculative multi-core processors and general-purpose applications
- A compiler framework to support speculative execution and optimizations
- Speculative optimizations for single thread
- **Speculative optimizations for multi-threaded processors**
- Conclusions

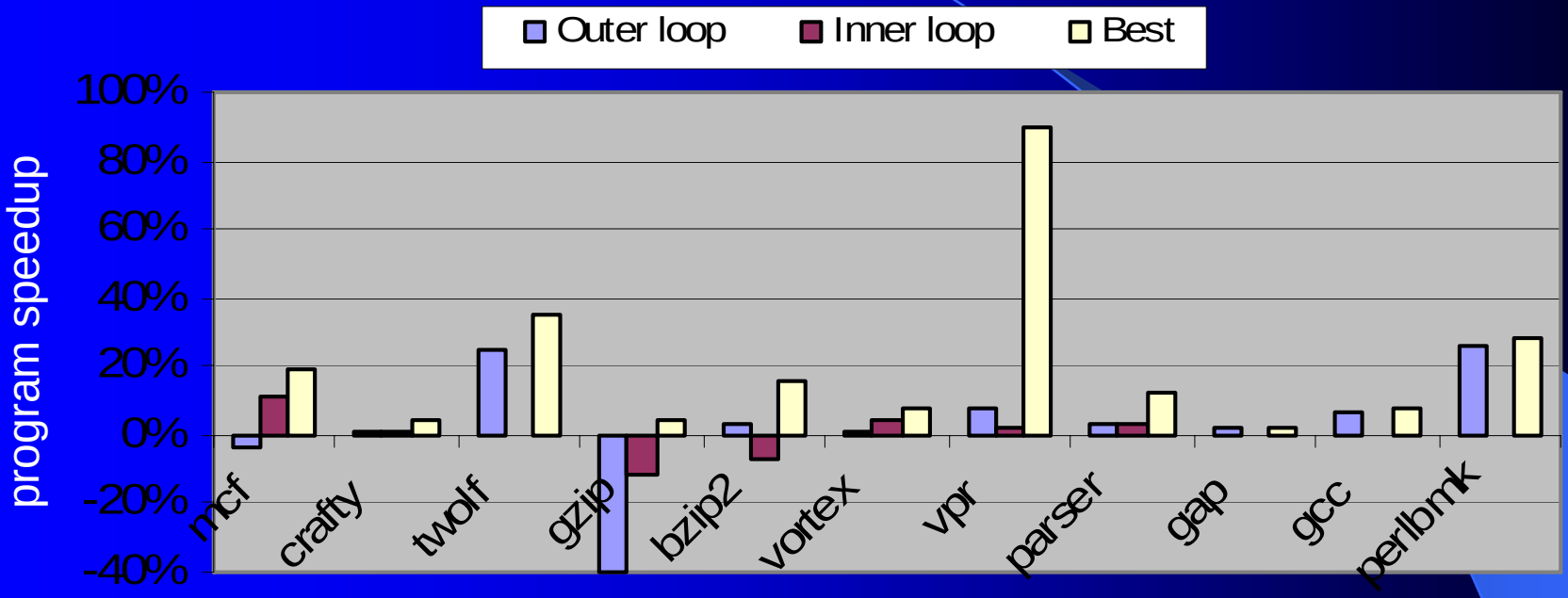
Compiler Optimizations for Speculative Threads

- Without compiler optimization, there is limited TLP even under *perfect* hardware support. [Oplinger PACT 99]
- Compiler have to decide
  - Which loops/regions to be transformed into thread
  - Use synchronization or speculation
  - How to schedule the code to improve overlaps
  - What transformations to be used
  - When/How to generate recovery code

Compiler Framework

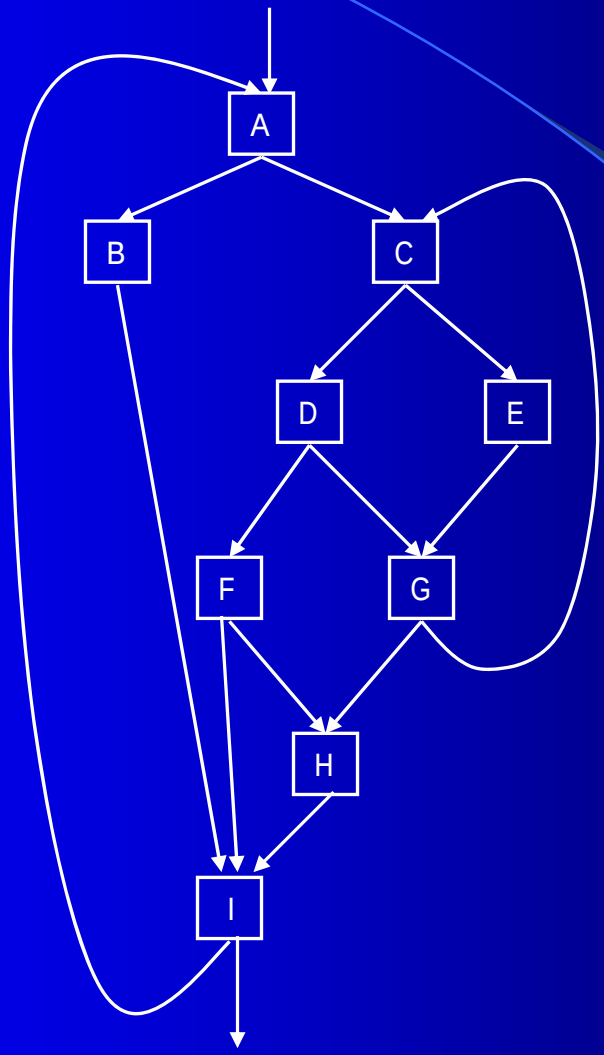


Loop Selection



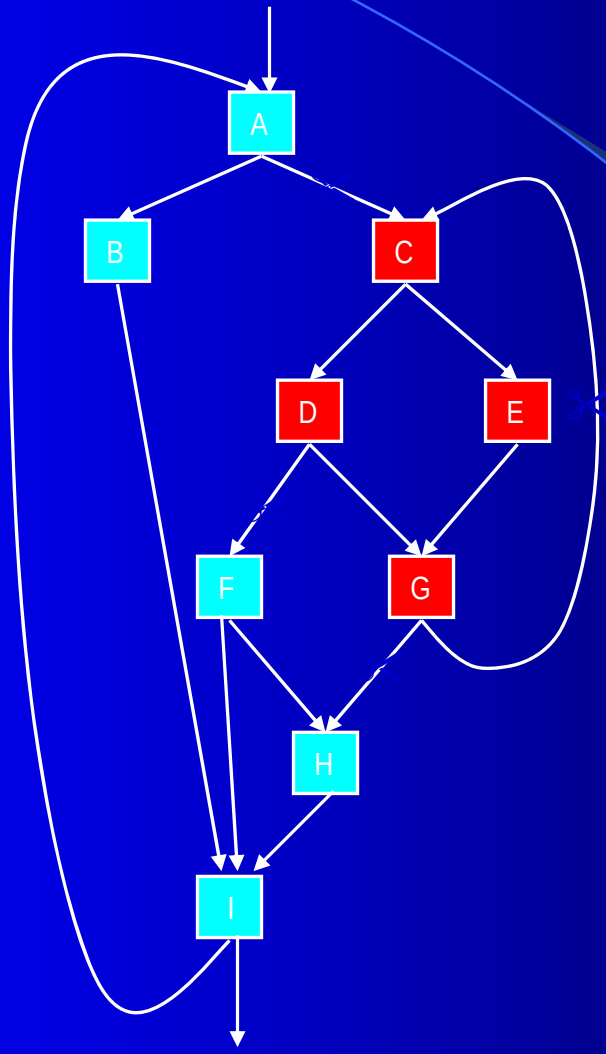
Carefully selected loops can improve performance significantly!

Example



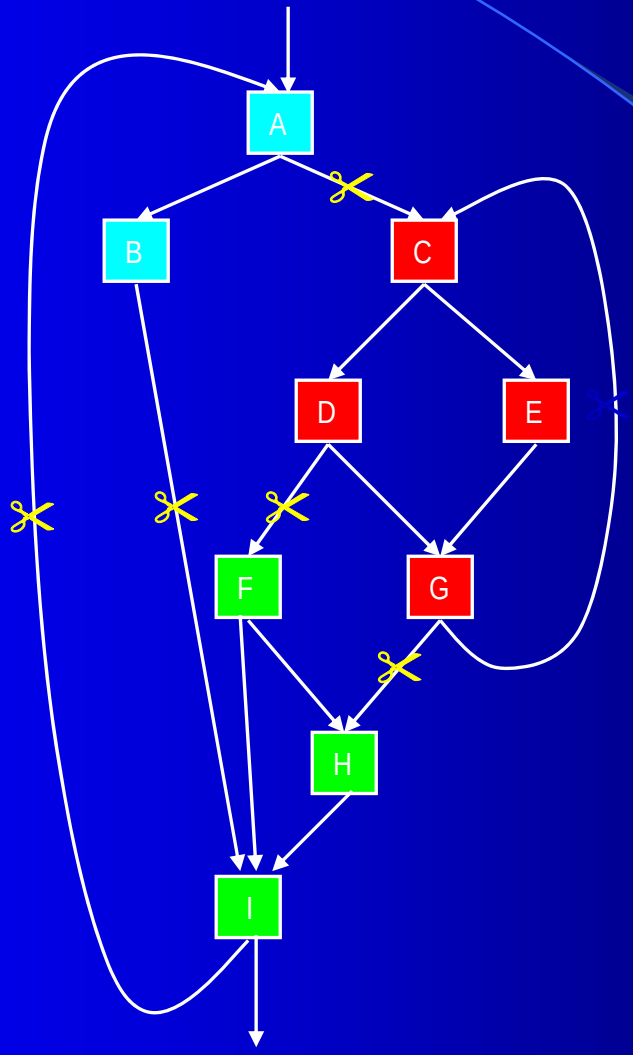
A two-level nested loop

Example: Loop Selection



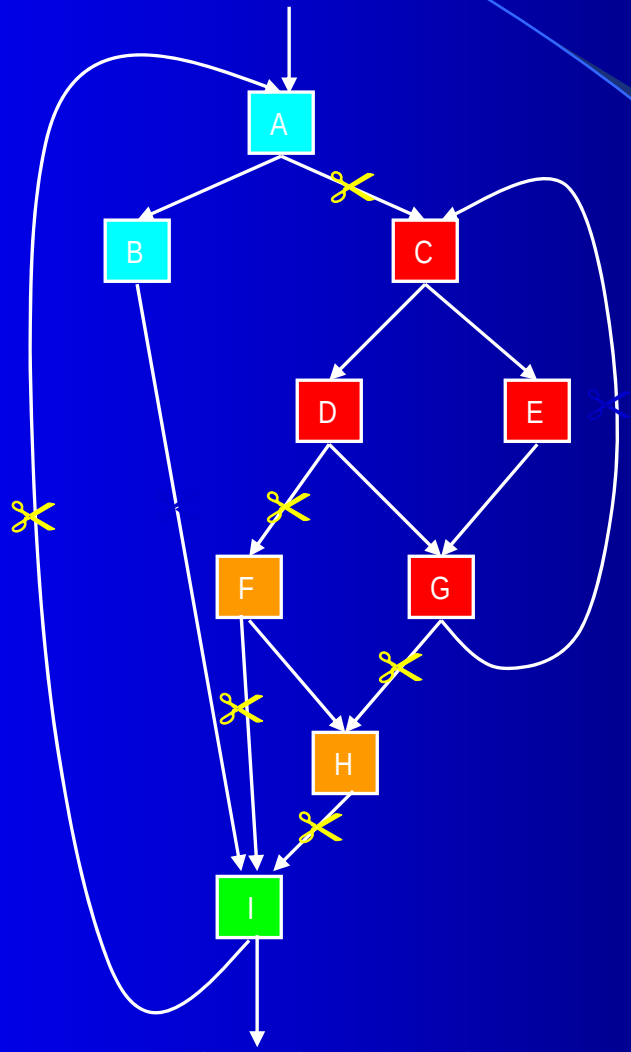
Generate loop threads for the selected inner loops

Example



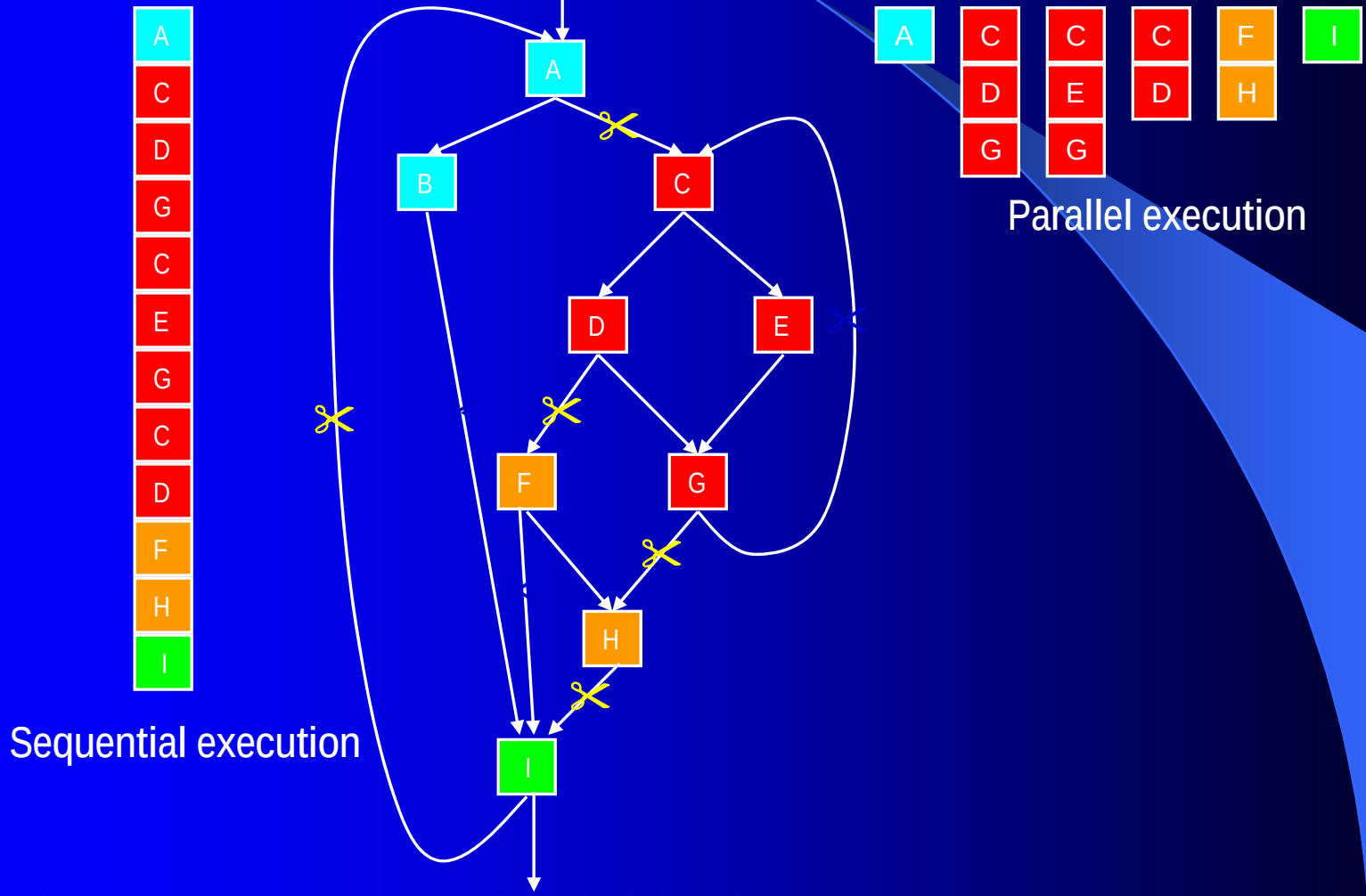
Partition the remaining part

Example



Further partitioning

Example


A
C
D
G
C
E
G
C
D
F
H
I
Sequential execution
A
B
C
D
E
F
G
H
I
A
C
D
G
C
E
G
C
D
F
H
I
Parallel execution

Thread Partition Algorithm

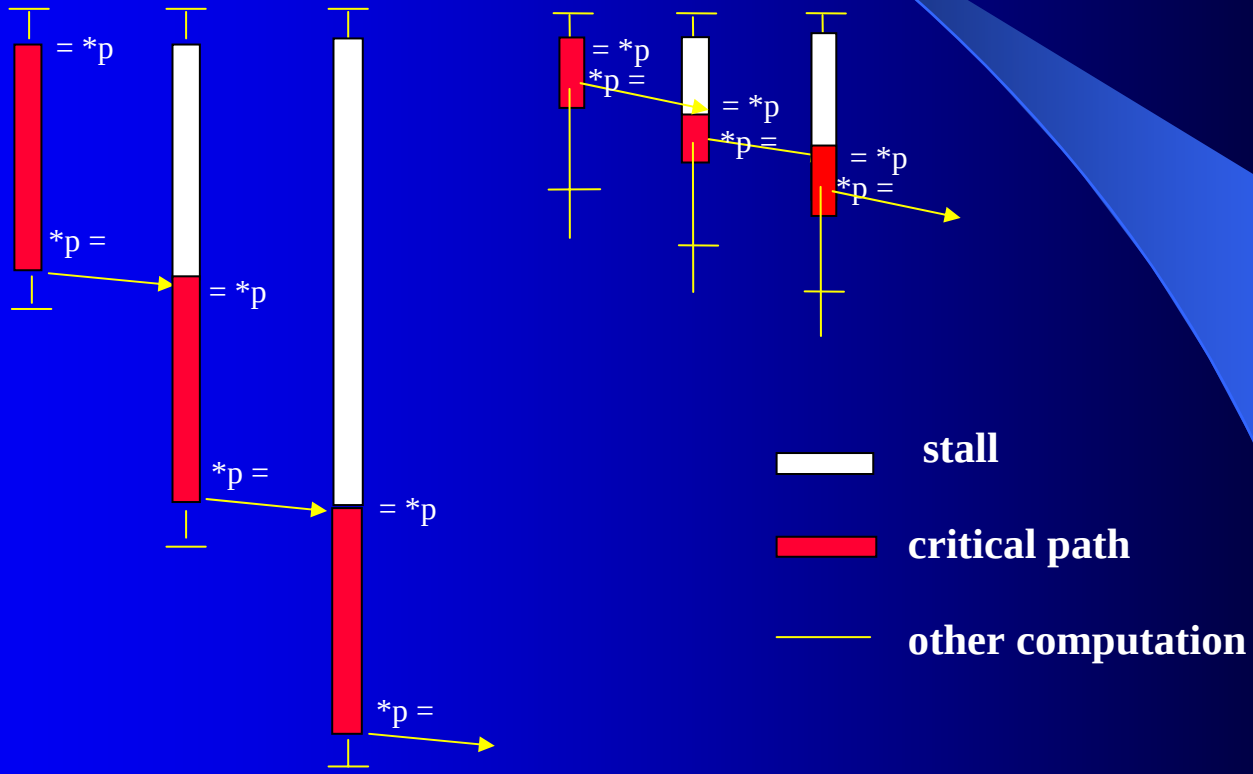
```
Thread_Partition(Procedure P)  {
   Form_Loop_Tree(P);
   for each unselected loop L in the loop
tree {
      Initialize_Thread_Queue(L);
      while (!Thread_Queue_Empty())  {
         T = Thread_Queue_Pop();
         if (Bi_Partition(T, T1, T2))  {
            Thread_Queue_Push(T1);
            Thread_Queue_Push(T2);
         }
      }
   } // bottom-up traversal
}
```

```
Bi_Partition(T, T1, T2)  {
   P = Form_Path(T);
   for each BB b on the path P  {
      benefit = Perf_Estimation(P, b);
      if (benefit > max_benefit)  {
         max_benefit = benefit;
         T1 = all BBs reachable from b;
         T2 = all remaining BBs in T;
      }
   }
}
```

Speculative Code Motion

before code motion

after code motion



Recovery Code Generation

- Representation of recovery code is crucial
- It should not affect the later compiler optimizations
- Recovery code could be represented as a *heavily-biased* If-Then-Else structure
- Generation of recovery code is rather straightforward with such a representation

Conclusions

- Microprocessors have caught up with supercomputers in ’90 and have gone *multi-core*
- It is non-trivial to apply current supercomputing technologies to *general-purpose* applications
- New architectural support such as *thread-level speculative execution*, and new compiler techniques such as *speculative optimizations* using *alias and data dependence profiling*, even *dynamic optimization* at runtime, are crucial – as always
- A new exciting era for parallel processing has arrived – regardless of we are prepared or not!

References

(1) J.Lin et al, *A Compiler Framework for Speculative Analysis and Optimizations*, Proc. Of ACM/SIGPLAN Conf. On Programming Language Design and Implementation (PLDI), June 2003, also in ACM Trans. On Architecture and Code Optimization (TACO), Vol. 1, No. 3, Sept. 2004, pp. 247-271

(2) J. Lin et al, *Recovery Code Generation for General Speculative Optimizations*, to appear in ACM Trans. On Architecture and Code Optimization (TACO) 2006.

(3) X.Dai et al, A General Compiler Framework for Speculative Optimizations Using Data Speculative Code Motion, Proc. Of the 3rd Annual IEEE/ACM Int'l Symp. On Code Generation and Optimization (CGO)

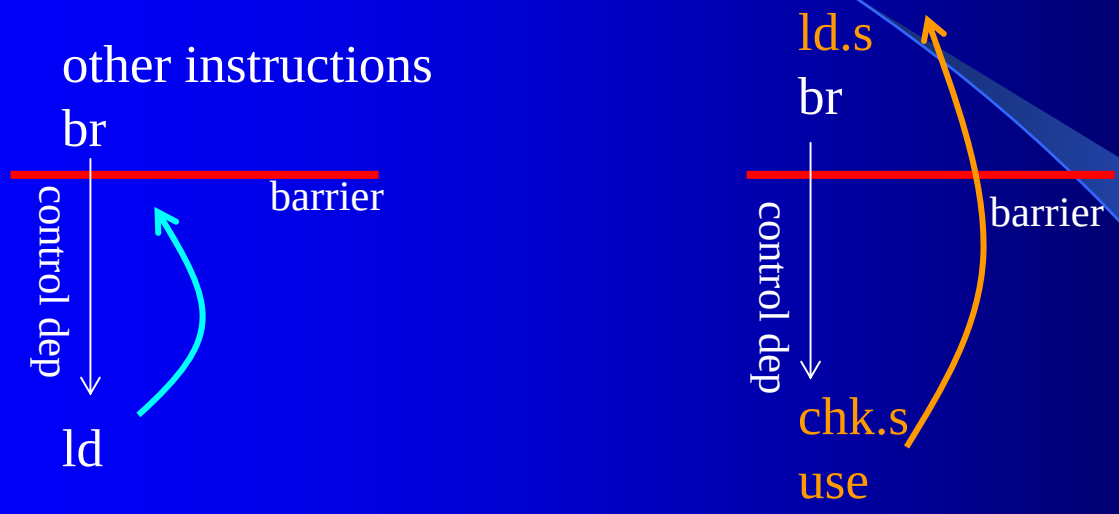
(4) T.Chen et al, *Data Dependence Profiling for Speculative Optimizations*, Proc. Of Int'l Conf on Compiler Construction (CC), March 2004

(5) T.Chen et al, *An Empirical Study on the Granularity of Pointer Analysis in C programs*, Proc. 15th Workshop on Languages and Compilers for Parallel Computing (LCPC), August 2002

(6) J.Y.Tsai et al, *The Superthreaded Processor Architecture*, IEEE Trans on Computers, special issue on Multithreaded Architecture, Vol. 48, No. 9, Sept 1999

Supplement Slides

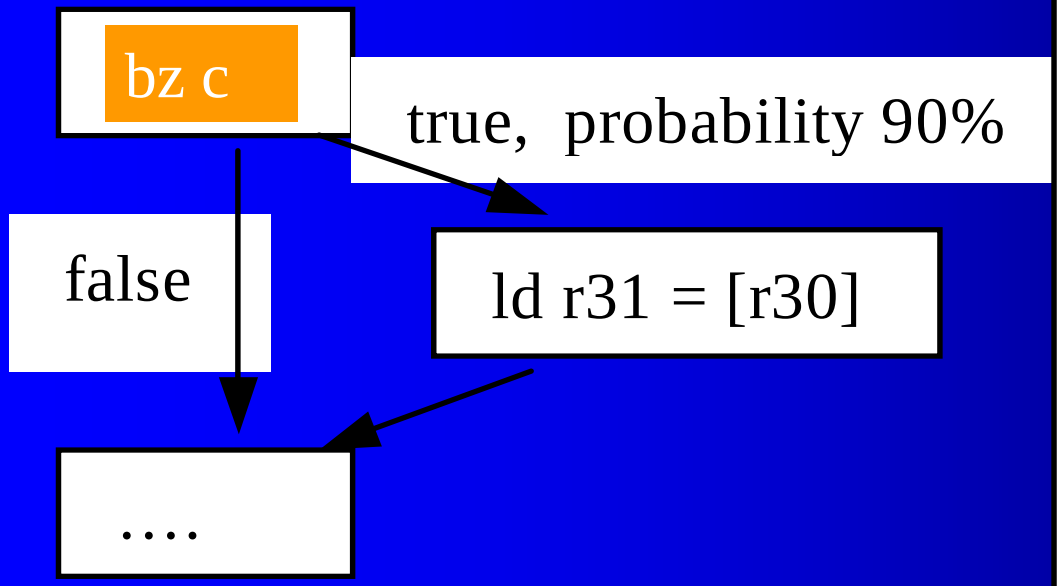
Control Speculation



traditional optimizations control speculation

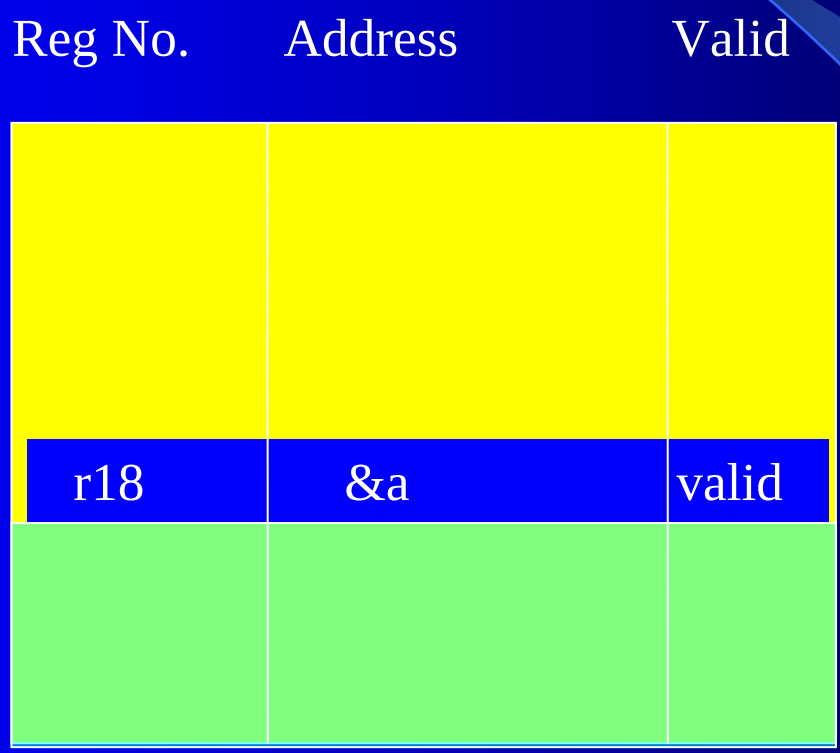
- ld.s: move the load operation across the barrier imposed by control dependency (*branch* instruction)
- Check the speculation with chk.s

Control Speculation – Instruction Scheduling




```
ld.s r31 = [r30]
if (c){
        chk.s r31, recovery
        next:
        ….
}
recovery:
        ld r31 = [r30]
        br next
```

Data Speculation: Advance Load Address Table (ALAT)



Data Speculation: Example

… = a *q = .. … = a Original program	*ld* r18=[a] store [*q] = *ld* r18=[a] Traditional compiler code

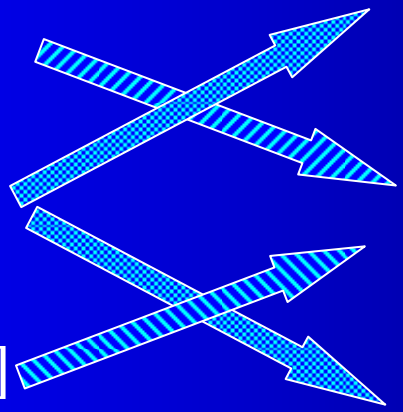
- *q and a are possible aliases
- a is reloaded after the store of *q

Data Speculation: Example

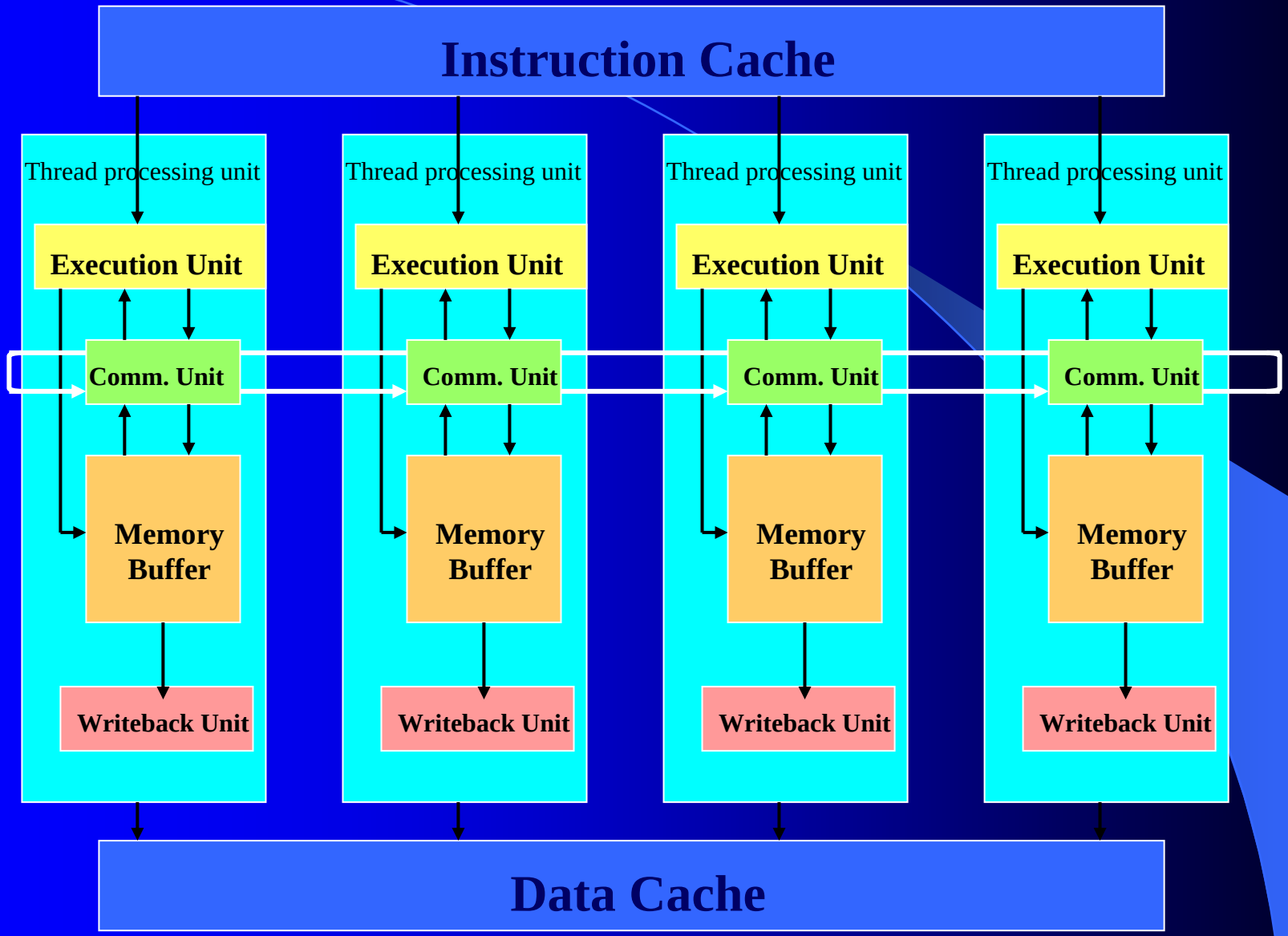
ld.a r18 = [a]

st [*q] =

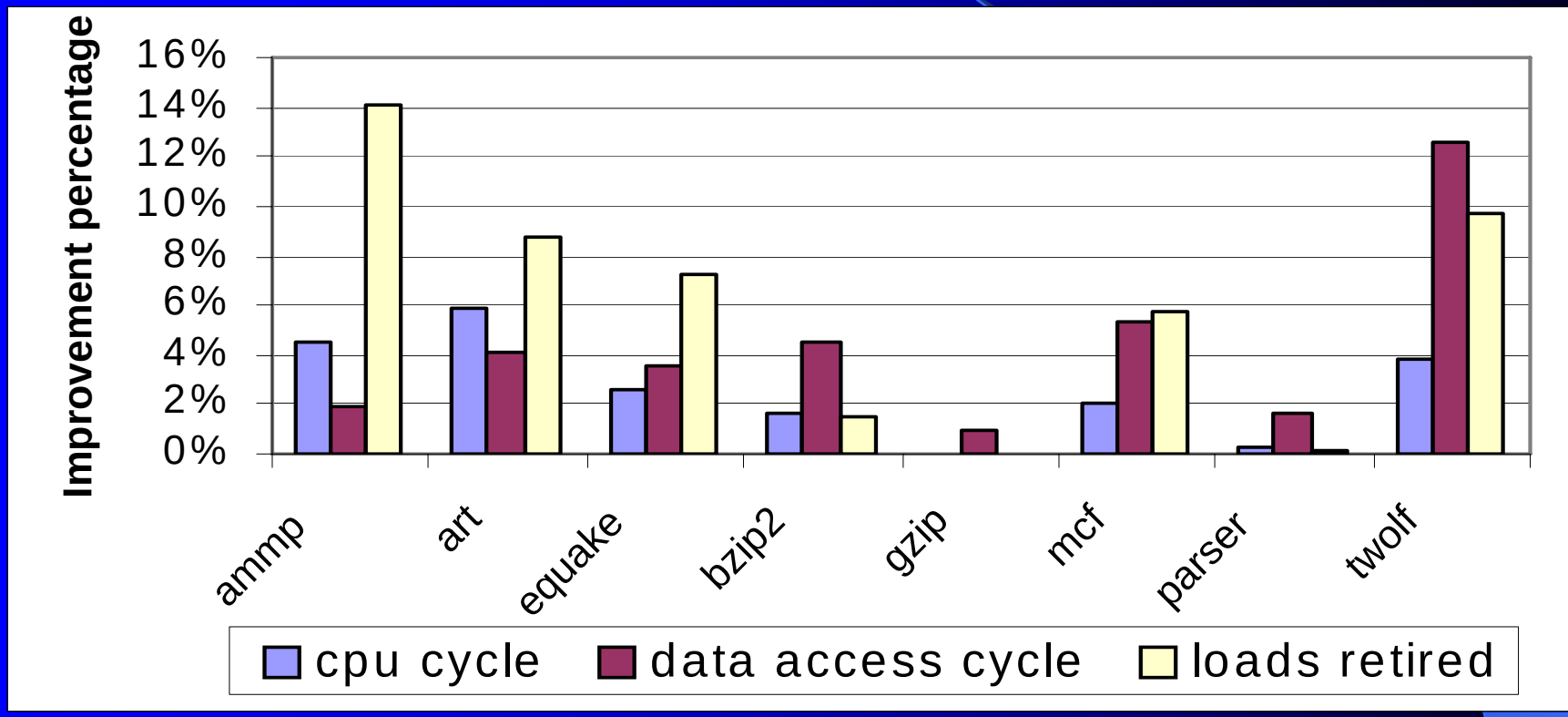
ld.c r18 = [a]



reg	addr	valid
r18	&a	valid


Instruction Cache
Thread processing unit
Execution Unit
Comm. Unit
Memory Buffer
Writeback Unit
Thread processing unit
Execution Unit
Comm. Unit
Memory Buffer
Writeback Unit
Thread processing unit
Execution Unit
Comm. Unit
Memory Buffer
Writeback Unit
Thread processing unit
Execution Unit
Comm. Unit
Memory Buffer
Writeback Unit
Data Cache

Performance Improvement of Speculative Register Promotion



Based on alias profile and compared with –O3 with type-based alias analysis on Intel ORC compiler

TLP for Each Benchmark