Compiler-based Instrumentation

Dr. Brian P. Railing

Overview of Talk

- **General Program Instrumentation**
- Path Profiling
- Memory Tracing
- Integrated Instrumentation
- Results

What is program instrumentation?

- **We want to embed a second program**
  - Code that is doing something that is not the original program
- **All compilers should still generate valid code**
  - Sometimes the instrumentation gets to terminate the program

Common Examples

- Execution - Path profiling
- Memory – Address tracing
- Validation – malloc / free, bounds checking

Overview of Talk

- General Program Instrumentation
- **Path Profiling**
- Memory Tracing
- Integrated Instrumentation
- Results

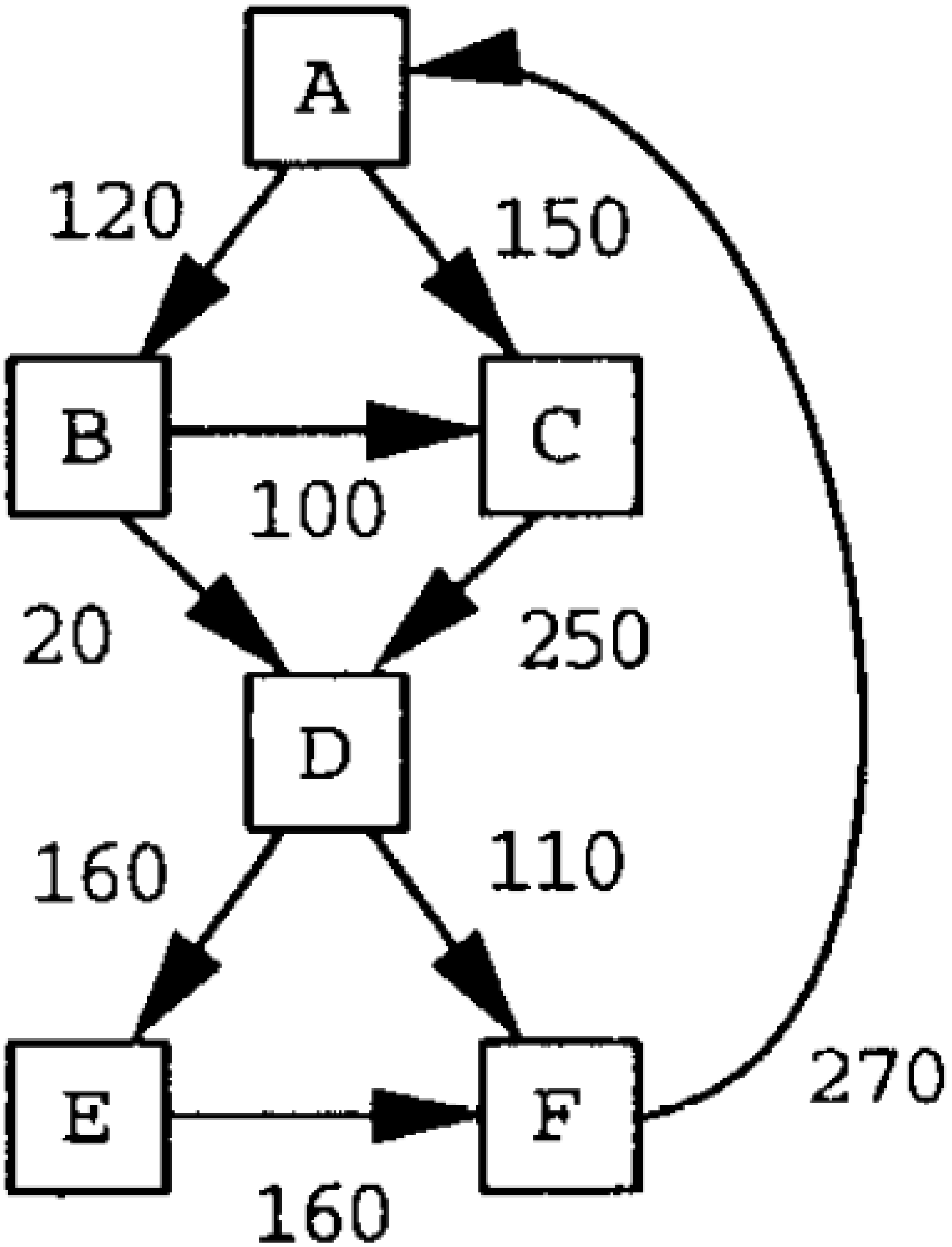
What is path profiling?

- Record the execution of the program
- Find hot paths
- Measure test coverage

Path Profiling

- **Starting with a control flow graph**
  - What is the hot path(s)?
- **How can we record this information?**
  - Block counts?
  - Edge frequencies?
  - Paths?

Path	Prof1	Prof2
ACDF	90	110
ACDEF	60	40
ABCDF	0	0
ABCDEF	100	100
ABDF	20	0
ABDEF	0	20



Figures from Ball & Larus, "Efficient Path Profiling"

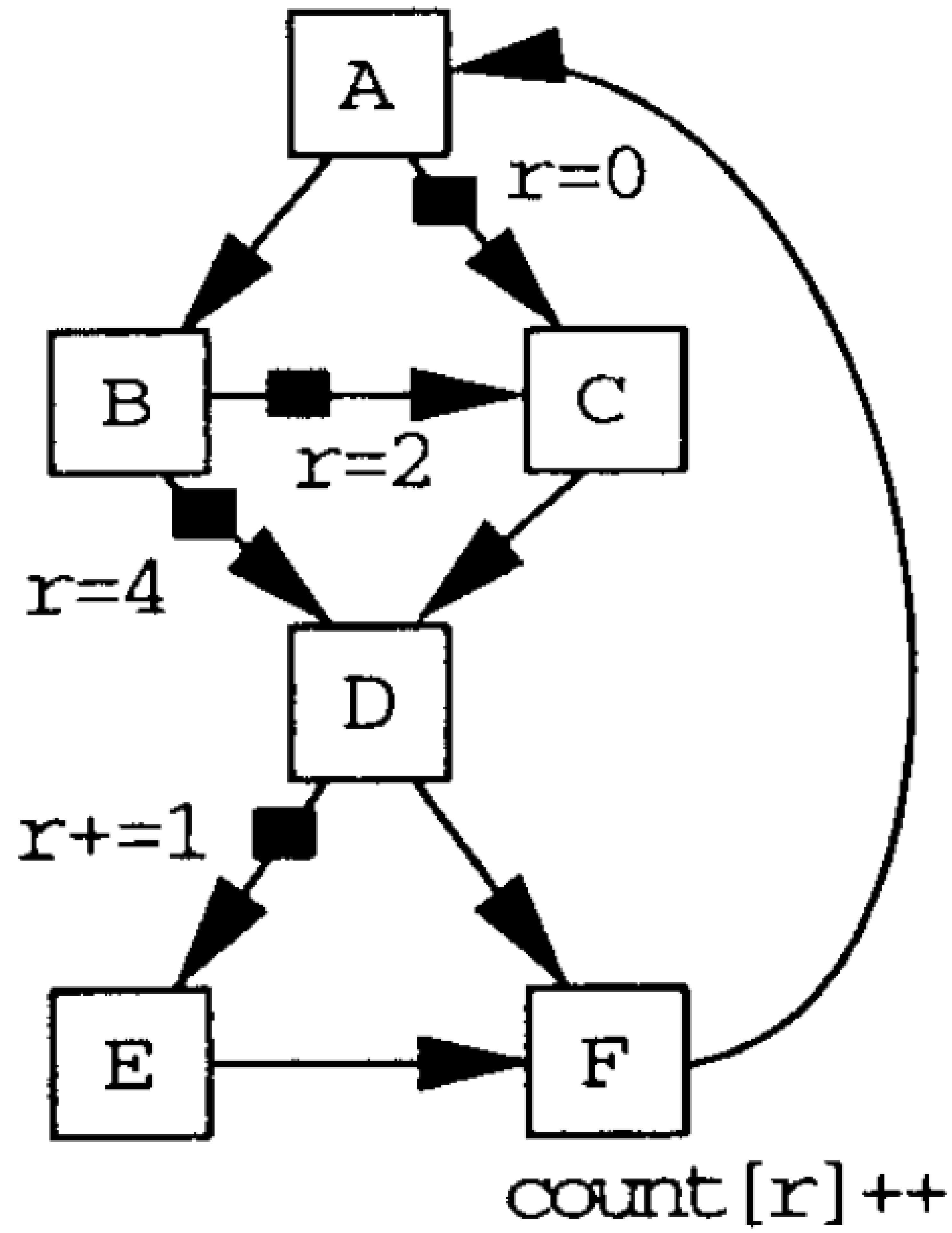
Block identifiers

- **Map every block "name" to an integer**

```
Func(Module M)
  foreach (F in M)
    // Canonicalize LLVM Basic Blocks
      foreach (B in F)
        bbid++
```

Path Profiling

- Given the CFG
  - What are the possible paths?
  - What edges distinguish paths?

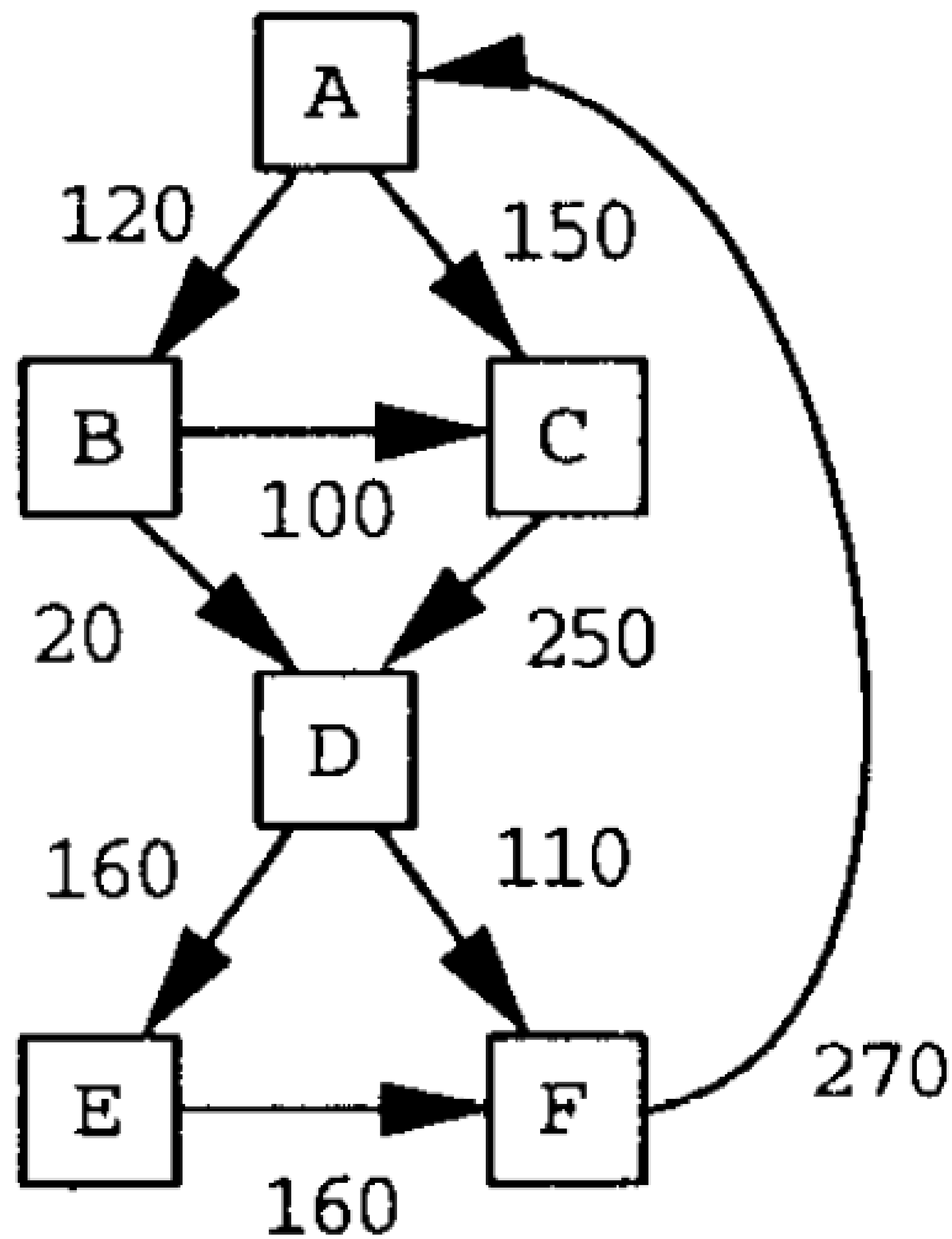


Figures from Ball & Larus, "Efficient Path Profiling"

Path Profiling (Probably)

- **Suppose we recorded one path in CFG**
  - How likely does it represent a hot path?

Path	Prof1	Prof2
ACDF	90	110
ACDEF	60	40
ABCDF	0	0
ABCDEF	100	100
ABDF	20	0
ABDEF	0	20



Figures from Ball & Larus, "Efficient Path Profiling"

Overview of Talk

- General Program Instrumentation
- Path Profiling
- **Memory Tracing**
- Integrated Instrumentation
- Results

Memory Tracing

- **Record every address accessed**
  - Asked to implement the instrumentation in cachelab
- **Record the details of those accesses**
  - Type / size
  - Load / store
  - Value (?)

Memory access in LLVM

```
%indvars.iv = phi i64 [ %16, %.lr.ph.us ], [ %indvars.iv.next, %17 ]
%i.01.us = phi i32 [ %6, %.lr.ph.us ], [ %32, %17 ]
%18 = getelementptr inbounds float, float* %9, i64 %indvars.iv
%19 = load float, float* %18, align 4, !tbaa !5
%20 = getelementptr inbounds float, float* %10, i64 %indvars.iv
%21 = load float, float* %20, align 4, !tbaa !5
%22 = getelementptr inbounds float, float* %11, i64 %indvars.iv
%23 = load float, float* %22, align 4, !tbaa !5
%24 = getelementptr inbounds float, float* %12, i64 %indvars.iv
%25 = load float, float* %24, align 4, !tbaa !5
%26 = getelementptr inbounds float, float* %13, i64 %indvars.iv
%27 = load float, float* %26, align 4, !tbaa !5
%28 = getelementptr inbounds i32, i32* %14, i64 %indvars.iv
%29 = load i32, i32* %28, align 4, !tbaa !1
%30 = tail call float @_Z19BlkSchlsEqEuroNoDivfffffif(float %19,
  float %21, float %23, float %25, float %27, i32 %29, float undef)
%31 = getelementptr inbounds float, float* %15, i64 %indvars.iv
store float %30, float* %31, align 4, !tbaa !5
%32 = add nsw i32 %i.01.us, 1
%33 = icmp slt i32 %32, %7
%indvars.iv.next = add nsw i64 %indvars.iv, 1
br i1 %33, label %17, label %._crit_edge.us
```

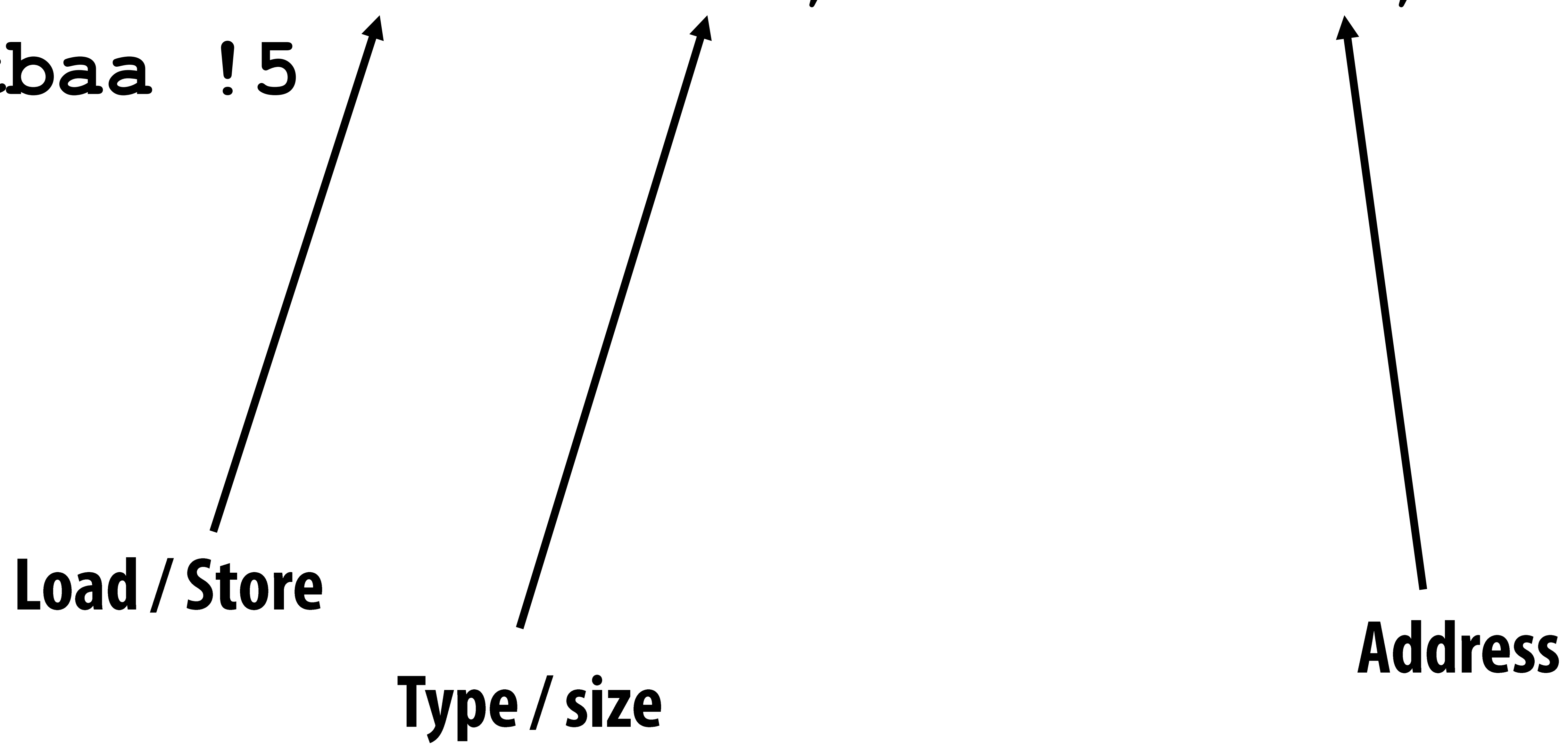
Memory access in LLVM

```
%indvars.iv = phi i64 [ %16, %.lr.ph.us ], [ %indvars.iv.next, %17 ]
%i.01.us = phi i32 [ %6, %.lr.ph.us ], [ %32, %17 ]
%18 = getelementptr inbounds float, float* %9, i64 %indvars.iv
%19 = load float, float* %18, align 4, !tbaa !5
```

Memory access in LLVM

- `%18 = getelementptr inbounds float, float* %9, i64 %indvars.iv`

- `%19 = load float, float* %18, align 4, !tbaa !5`



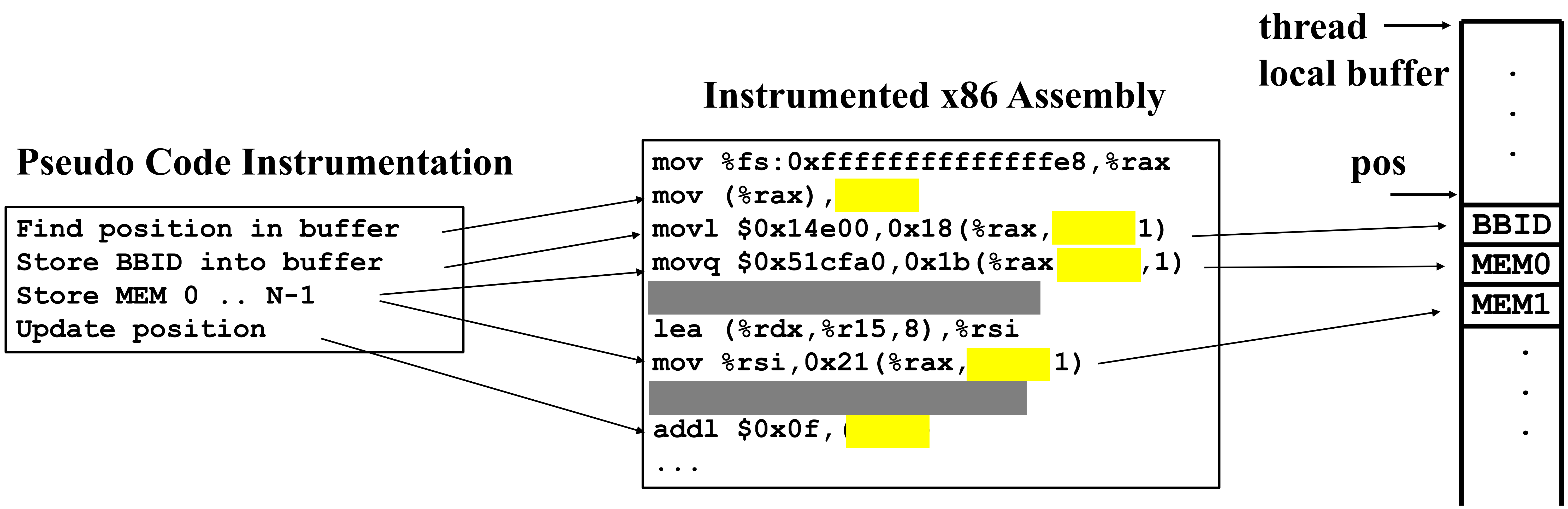
Overview of Talk

- General Program Instrumentation
- Path Profiling
- Memory Tracing
- **Integrated Instrumentation**
- Results

LLVM Instrumentation Design (Contech)

- **Compile the source language into LLVM IR**
- **Instrument each basic block**
  - Record its execution (i.e. control flow)
  - Record memory operations
  - Record other operations

Instrumentation Design



Basic Block Instrumentation

- **Prologue:**

```
Buffer = __ctGetBuffer()
Buffer Position = __ctGetBufferPos()
fence singlethread acquire
*Buffer Position = __ctStoreBasicBlock(BBID, Buffer Position,Buffer)
```

- **Body:**

```
__ctStoreMemOp(Addr, Number, *Buffer Position)
```

- **Epilogue:**

```
New Pos = __ctStoreBasicBlockComplete(Number of MemOps,
                                       Buffer Position, Buffer)
fence singlethread release
__ctCheckBufferSize(New Pos)
```

Aggressive Inling

- **Buffer = __ctGetBuffer()**
  `mov %fs:0xffffffffffffffe8,%rax`
- **Buffer Position = __ctGetBufferPos()**
  `mov (%rax),%ecx`
- **fence singlethread acquire**
  `// Compiler directive`
- ***Buffer Position = __ctStoreBasicBlock(BBID, Buffer Position, Buffer)**
  `movl $0x14e00,0x18(%rax,%rcx,1)`
- **__ctStoreMemOp(Addr, Number, *Buffer Position)**
  `movq $0x51cfa0,0x1b(%rax,%rcx,1)`

Research thesis

- **Memory traffic from instrumentation dominates overheads**
- **Each instrumented thread generates 100MB/s – 1GB/s**
- **Basic blocks are 90+% of trace**
  - And each basic block event is mostly memory operations

Control Flow Improvements

- **A basic block event generates significant memory traffic**
  - 3 loads
  - 2 stores
  - 0-N Memory operations (currently 0.9 / block)

- **Analyzing the CFG, memory traffic can be reduced**

Basic Block Instrumentation - Chained

- **Prologue:**

```
Buffer = __ctGetBuffer()
Buffer Position = __ctGetBufferPos()
fence singlethread acquire
*Buffer Position = __ctStoreBasicBlock(BBID, Buffer Position, Buffer)
```

- **Body:**

```
__ctStoreMemOp(Addr, Number, *Buffer Position)
```

- **Epilogue:**

```
New Pos = __ctStoreBasicBlockComplete(Number of MemOps,
                                      Buffer Position, Buffer)
fence singlethread release
__ctCheckBufferSize(New Pos)
```

Basic Block Instrumentation - Chained

- If predecessor is unconditional, then path is known

- **Prologue:**

```
Buffer = __ctGetBuffer()
Buffer Position = __ctGetBufferPos()
fence singlethread acquire
*Buffer Position = __ctStoreBasicBlock(BBID, Buffer Position, Buffer)
```

- **Body:**

```
__ctStoreMemOp(Addr, Number, *Buffer Position)
```

- **Epilogue:**

```
New Pos = __ctStoreBasicBlockComplete(Number of MemOps,
                                      Buffer Position, Buffer)
fence singlethread release
__ctCheckBufferSize(New Pos)
```

Basic Block Instrumentation - Chained

- **If predecessor is conditional, then path is knowable**

```
%9 = icmp eq i32 %argc, 4
%dir9 = zext i1 %9 to i8, !ContechInst !1
```

- **Prologue:**

```
Buffer = __ctGetBuffer()
Buffer Position = __ctGetBufferPos()
fence singlethread acquire
*Buffer Position = __ctStoreBasicBlock(BBID, Buffer Position, Buffer)
```

- **Body:**

```
__ctStoreMemOp(Addr, Number, *Buffer Position)
```

- **Epilogue:**

```
__ctStorePathInfo(Predecessor Direction)
New Pos = __ctStoreBasicBlockComplete(Number of MemOps,
                                      Buffer Position, Buffer)
fence singlethread release
__ctCheckBufferSize(New Pos)
```

Path "Profiling"

- **20 years of research on efficiently recording paths**
  - No one has integrated memory addresses and synchronization
  - And every reviewer wants to know why I am not using path profiling techniques

- **Prototypes:**
  - Elide basic block IDs at convergence points
  - Chain buffer loads / stores
  - Store conditional direction information on branches
  - TODO: Assign path IDs

Recording a Memory Operation

- **What can a memory operation trace record**
  - Load/Store
  - Address
  - Size / Type
  - Value

Prior Static Analysis in Contech

- **Basic blocks are consistent in memory operations**
  - If we record basic blocks, then load/store and size/type is unchanged on each execution
  - Record the load/store and size/type once in basic block info table

- **64-bit Addresses are only 6 bytes**

```
00 01 02 03 04 05 00 01 02 03 04 05 00 01 02 03 04 05 06 07
```

Current Static Analysis

- **Not all addresses are required.**
  - Addresses are constant
  - Or are constant offsets from other addresses.

Detecting Similar Addresses

- **For each basic block**
  - For each memory operation
    - Check if any prior operation in this basic block has a similar address calculation

- **Similar Address Calculations**
  - Is it this a getelementptr instruction?
  - Does each component match?
  - If not, is the component a constant value?
    - Accumulate constant differences

- **Store memory operation indices and constant differences into basic block info table**

Address Offsets

- **Given a base address, each value passed to GetElementPtr applies some offset**
  - Struct Type – value selects a field in the struct
  - Otherwise – value is an index into an array of type

```
%90 = getelementptr inbounds %struct.OptionData_,
          %struct.OptionData_* %89, i64 1, i64 %iv, i32 8
```

Current Static Analysis

- **Not all addresses are required, exceptions:**
  - Addresses that are constant
  - Or are constant offsets from other addresses.
  - Or are linear functions of loop induction variables.

Global Variables

- **Static analysis checks each address calculation**
  - If the base address is a global and uses constant offsets, then add a static instrumentation call to record the runtime address once
  - Elide the instruction to record the runtime address in the basic block.

- **LLVM:**

```
NULL != dyn_cast<GlobalValue>(addr) &&
gv->isThreadLocal()
```

Loop-based Calculations

- **Identify memory accesses in loops where:**
  - Each calculation component is constant
  - Or is loop invariant
  - Or is the loop induction variable
- **Iteration distance must be constant**

LLVM Scalar Evolution

```
for (i = m; i < n; i += step)
{
  ...
  x = q[i+1];
  p[i] = x;
  ...
}
```

```
while (x != last)
{
  *x = ...;
  x++;
}
```

Loop Example

- What does this code look like in LLVM IR?
- What is the induction step?

```
.lr.ph:
  %4 = phi i8 [ %10, %8 ], [ %2, %.lr.ph.preheader ]
  %.01 = phi i8* [ %9, %8 ], [ %s, %.lr.ph.preheader ]

  %10 = load i8, i8* %9, align 1, !tbaa !1
  %11 = icmp eq i8 %10, 0
  br i1 %11, label %.loopexit.loopexit, label %.lr.ph
```

```
while (x != last)
{
   *x = ...;
   x++;
}
```

Loop Example

- C code for earlier LLVM IR example

```
for (i=start; i<end; i++) {
   price = BlkSchlsEqEuroNoDiv( sptprice[i],
                strike[i], rate[i], volatility[i],
                otime[i], otype[i], 0);
```

- 6 Load instructions do not need to be directly instrumented

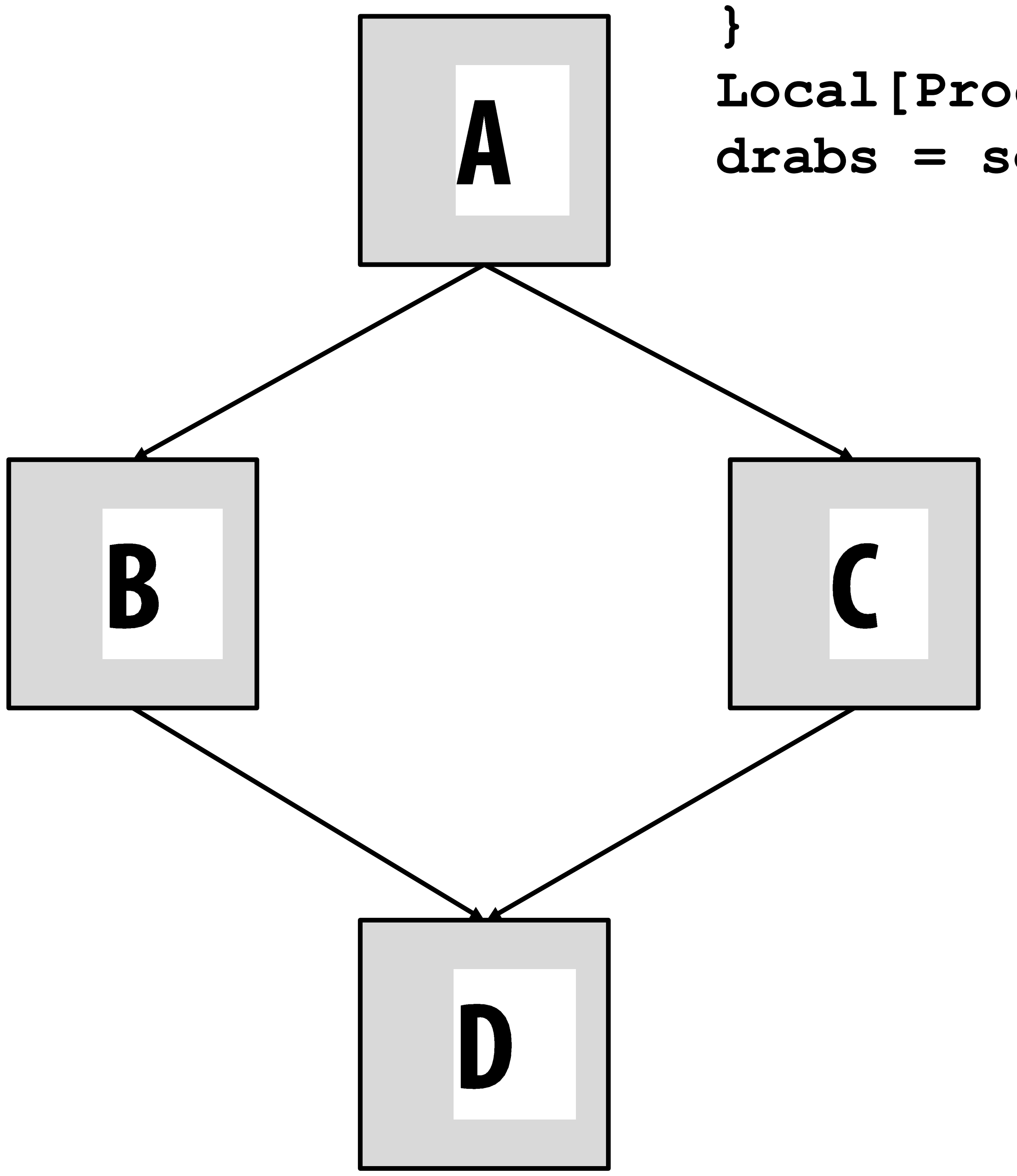
Current / Future Memory Work

- **Loop entry / exit headers cost space**
  - Overhead on low iteration loops
- **Extend loop work**
  - Mod operations (usually AND)
  - Variable iteration distance
- **Identify other elidable calculations**
  - `q[0] = q[*x]` // if sizeof(*x) < 6, then store value instead of address)

Similar Address Problem (barnes)

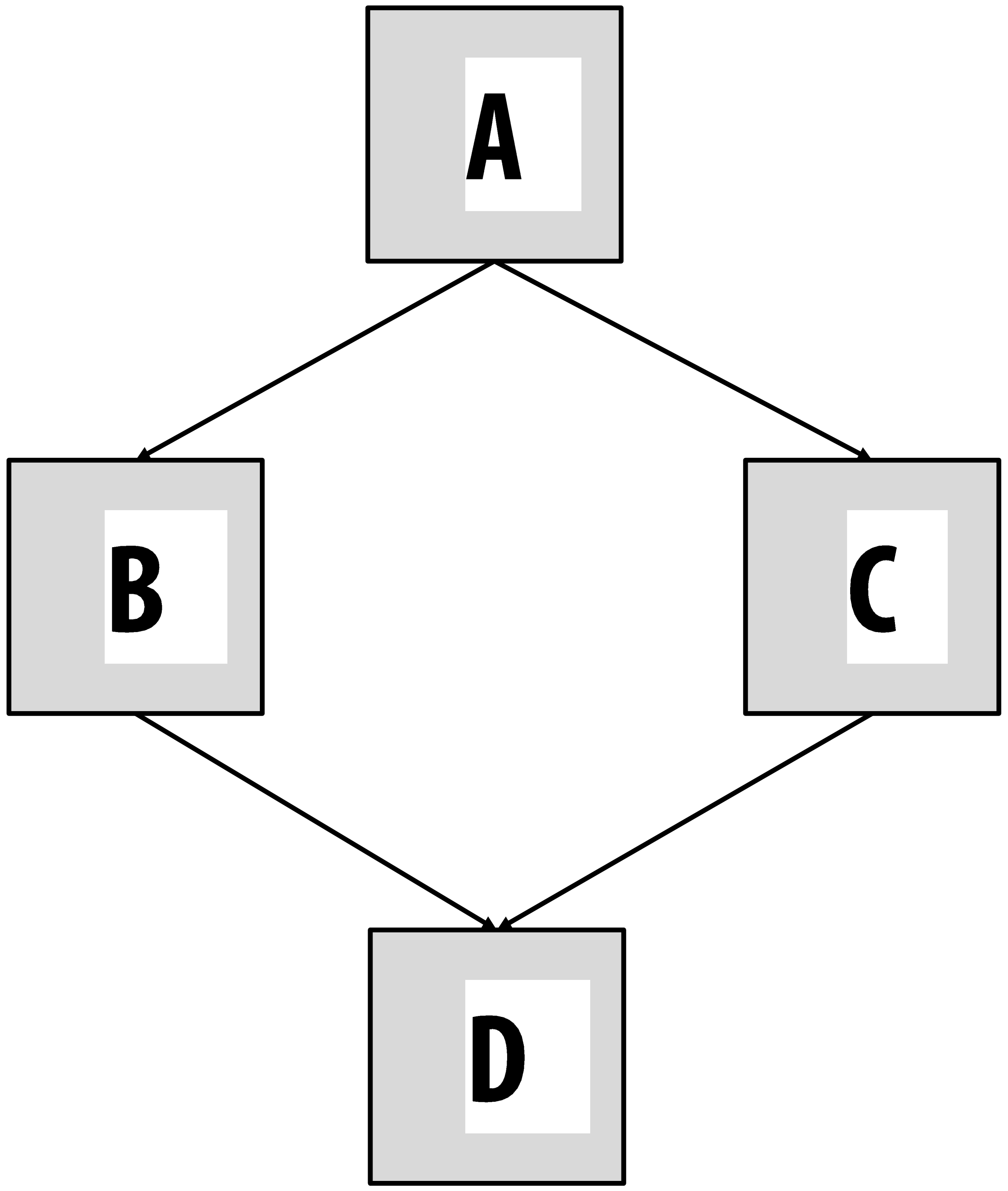
- Conditional code in one path
- Load/Store in tail block

```
if (p != Local[ProcessId].pmem) {
    SUBV(Local[ProcessId].dr,
         Pos(p),
         Local[ProcessId].pos0);
    DOTVP(Local[ProcessId].drsq,
          Local[ProcessId].dr,
          Local[ProcessId].dr);
}
Local[ProcessId].drsq += epssq;
drabs = sqrt((double) Local[ProcessId].drsq);
```



Tail Duplication

- Duplicate the tail block to enlarge the scope for finding similar addresses
- Merge it with each of the predecessor blocks



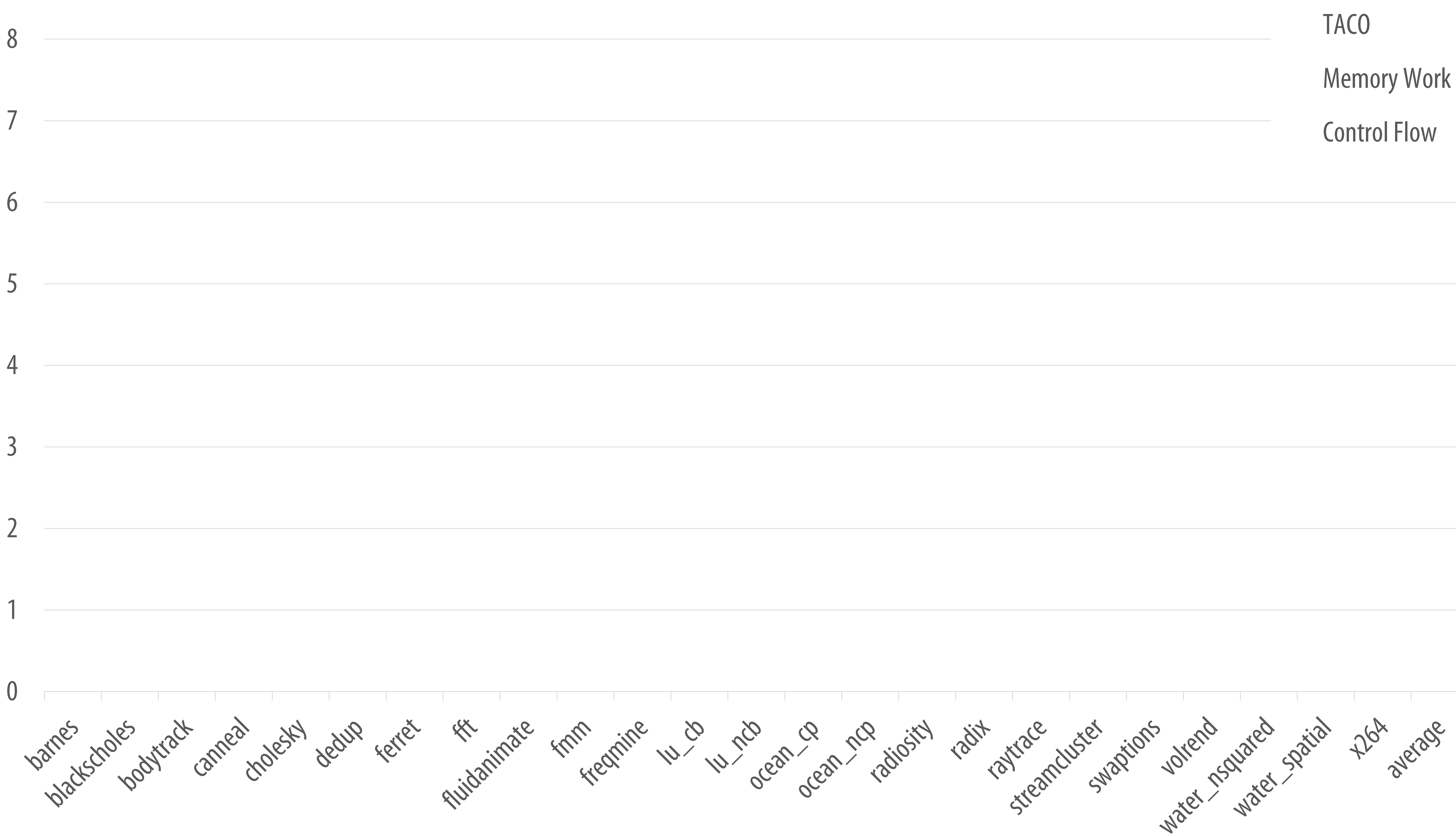
Tail Duplication Algorithm

- **Determine if the tail block is valid for duplication**
  - Not the return block
  - No address taken
  - Etc.
- **Determine that each predecessor is valid**
  - Unconditional branch to tail block
- **Duplicate and Merge**
  - Duplicate the tail block
  - Create / update PHI nodes as appropriate

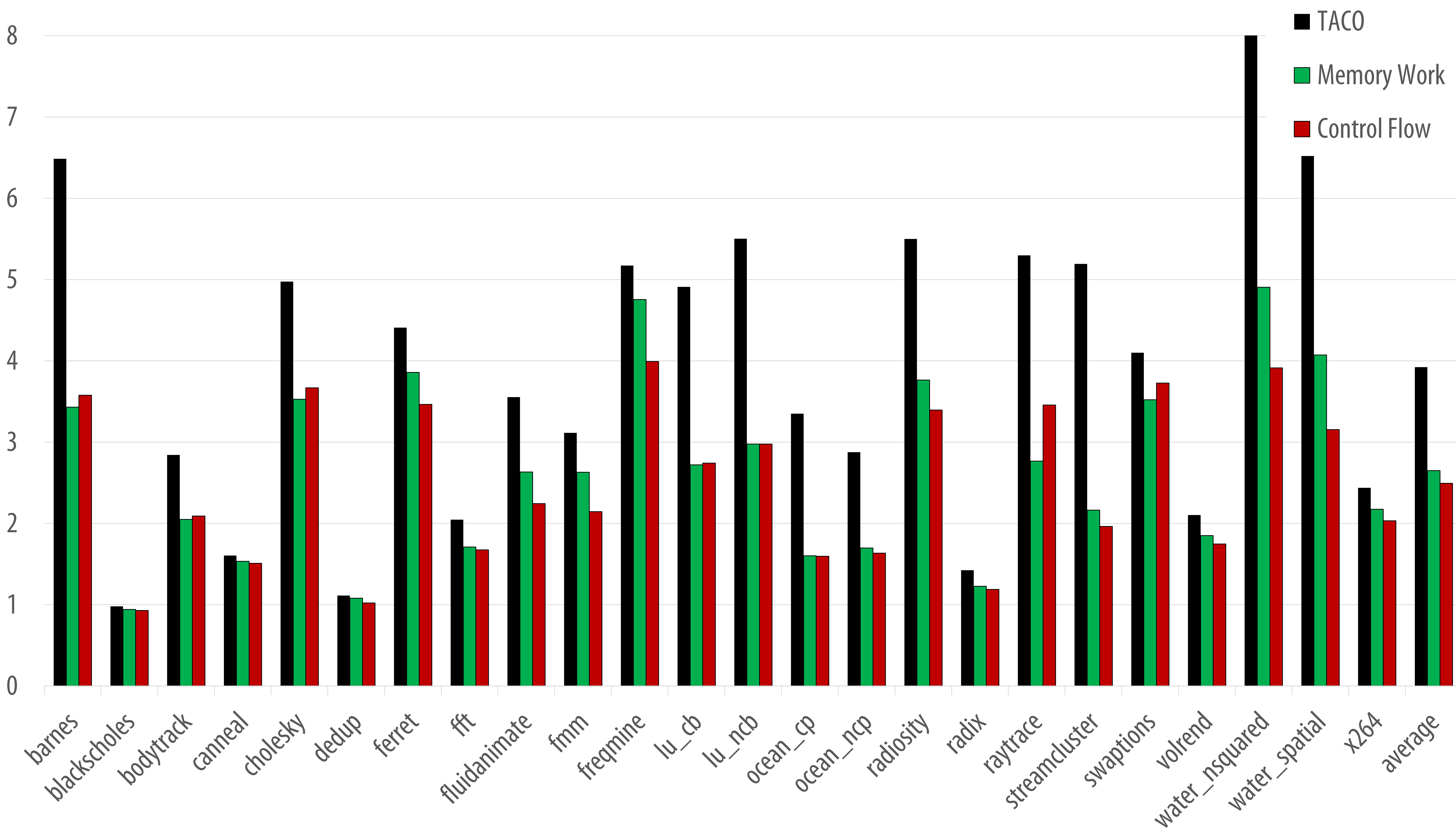
Overview of Talk

- General Program Instrumentation
- Path Profiling
- Memory Tracing
- Integrated Instrumentation
- **Results**

Instrumentation Performance Comparison



Instrumentation Performance Comparison



Related Tools

	Slowdown	Control Flow	Memory Accesses	Parallel Actions
Pin BBCount	2x-4x	b		
Harmony	1.2x	b		
CAB Path Profiling	1.4x-2.2x	x		
Pin Memory Trace	2x-8x		x	
PEBIL	7.7x		x	
MACPO	1.5x-6x		x	
ShadowReplica	2.7x	x	x	
PiPA	5x	x	x	
Cilkview	2x-10x	b		x
ParaMeter	3x-200x	b		x
Peregrine	2x-35x		x	x
Pin Task Graph	16x	x	x	x
ParaOPs	n/a	x	x	x
Contech	1x-5x	x	x	x

Conclusion

- **Prior work reduced instrumentation instructions required**
- **Prior work minimized instrumented thread interactions**
  - Tickets to order locks and barrier operations
  - Maximize usage of buffers
- **Instrumentation performance is often memory bandwidth constrained**
  - Minimize the size of records
  - Find redundant data and elide
- **LTO is very valuable**

Future Work

- **Global Variables**
  - Address is known at link time, how to record this
- **Memory Operations in a Loop**
  - Base pointer + offset function to reconstruct addresses
- **Release set of collected task graphs**

Code Available

- http://bprail.github.io/contech/

Hardware Configuration

- **Intel Xeon E3-1240v5 (Skylake)**
  - 3.50 GHz Quad-core, 2-way Hyperthreading
- **32 GB Main Memory**
- **256 GB NVMe M.2 PCIe SSD**
  - minimal speedup versus tmpfs or local storage

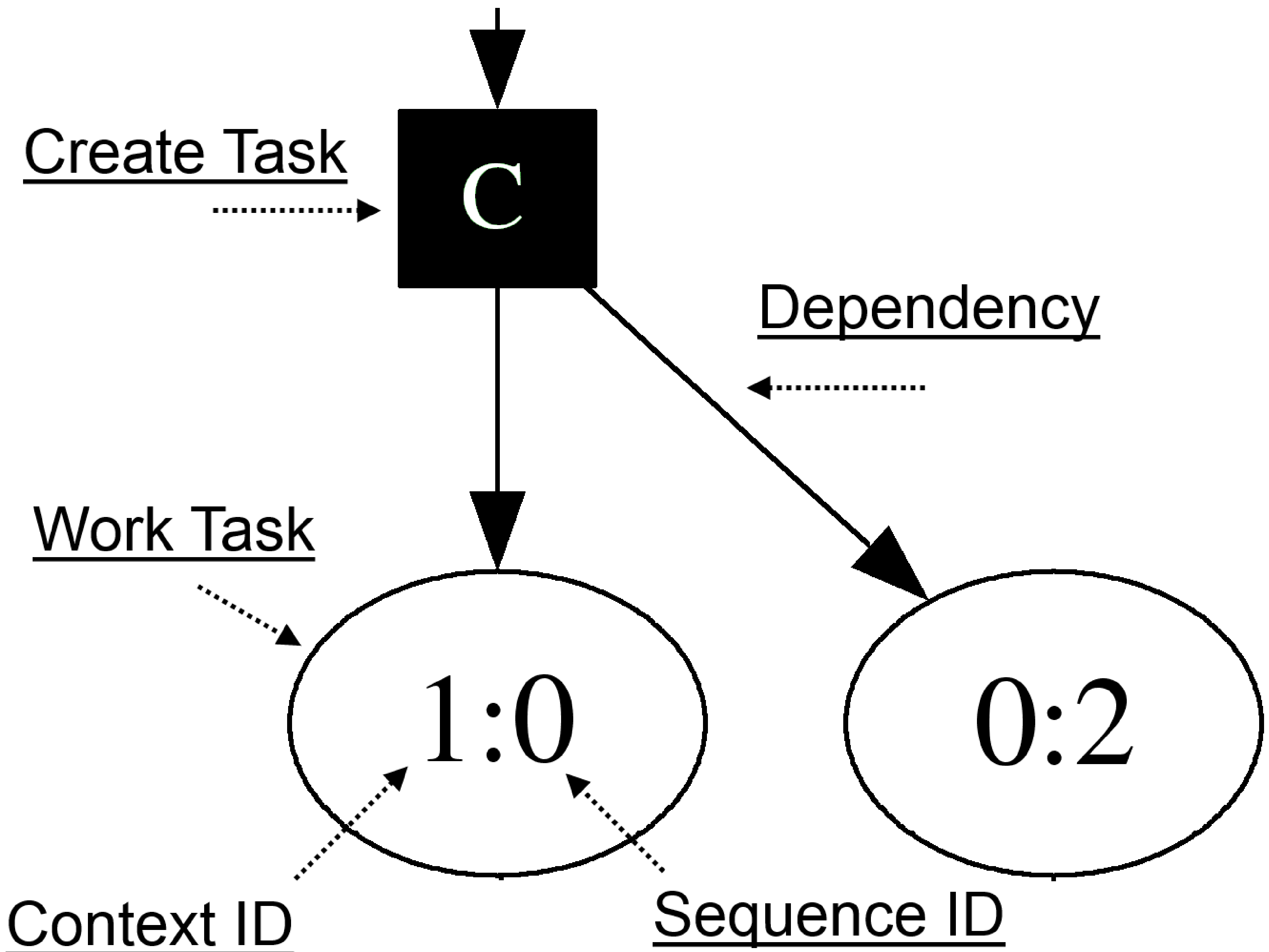
Objectives of Parallel Program Representation

- **A common representation needs**
  - What was executed
  - What was accessed
  - In what order did threads execute
- **Generate the representation with no user intervention**
  - Without constraint of language, library, or structure
- **Without recording architecture / runtime effects**
  - Context switches
  - Consistency model
  - Cache Effects
  - …

Contech's Task Graph Representation

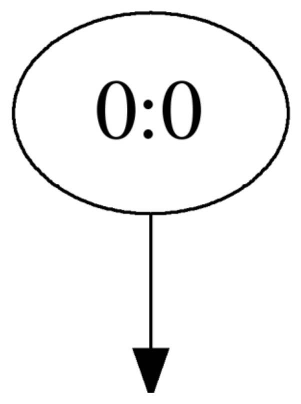
- **Task Graphs are directed, acyclic graphs containing**
  - Nodes partitioned based on type
  - Edges as scheduling dependencies
  - Nodes contain lists of actions and data
  - Other graph annotations such as start / end time

Task Graph Legend



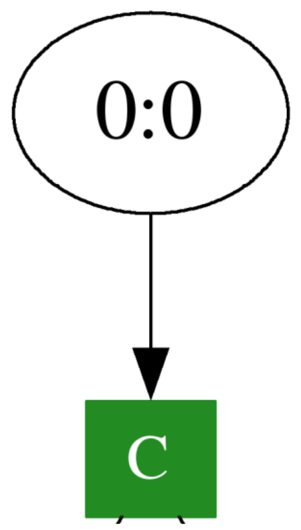
Task Graph Example

```
int fib(int n) {
  if (n < 2)
    return n;
  int a = cilk_spawn fib(n-1);
  int b = fib(n-2);
  cilk_sync;
  return a + b;
}
```



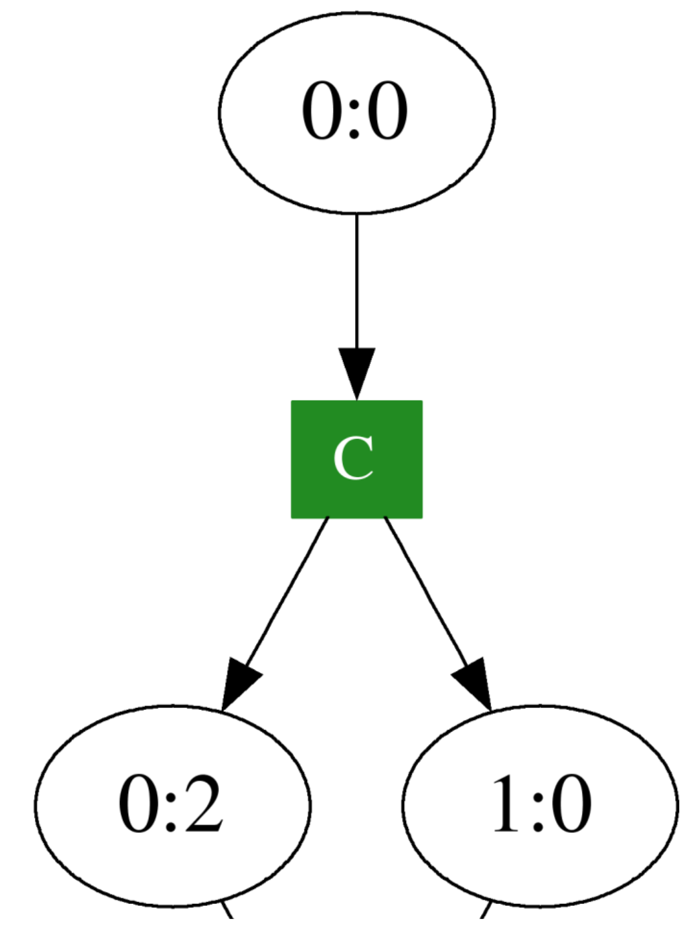
Task Graph Example

```
int fib(int n) {
  if (n < 2)
    return n;
  int a =             fib(n-1);
  int b = fib(n-2);
  cilk_sync;
  return a + b;
}

fib(2);
```



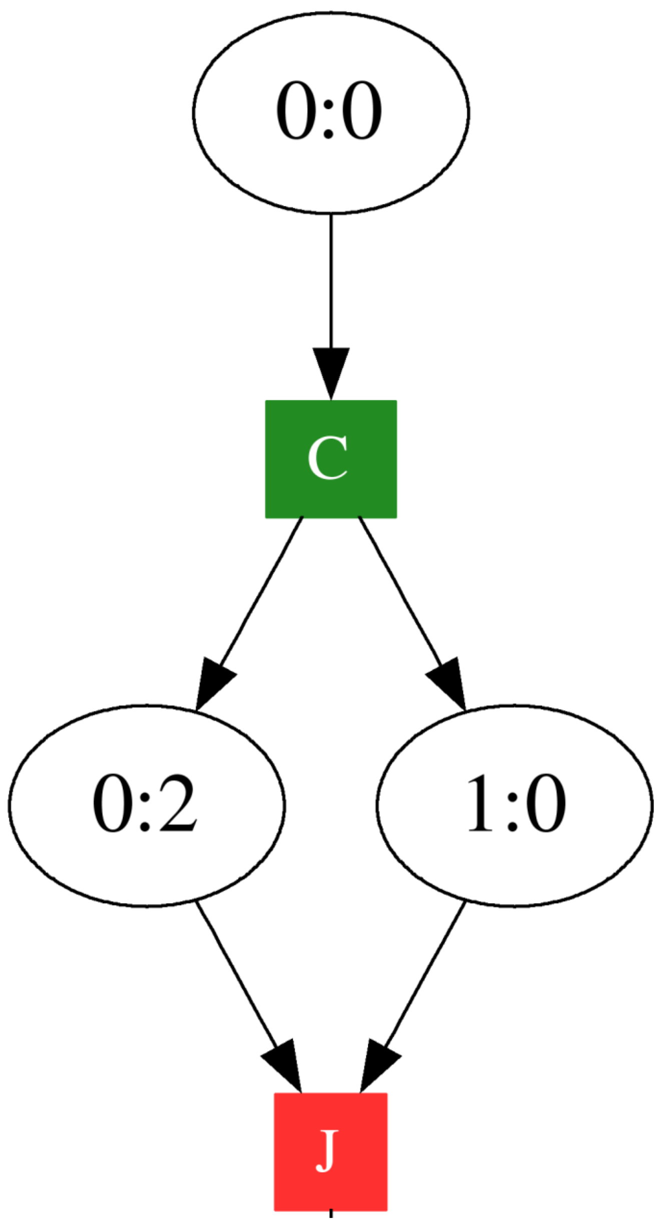
Task Graph Example

```
int fib(int n) {
  if (n < 2)
    return n;
  int a = cilk_spawn
  int b =
  cilk_sync;
  return a + b;
}

fib(2);
```



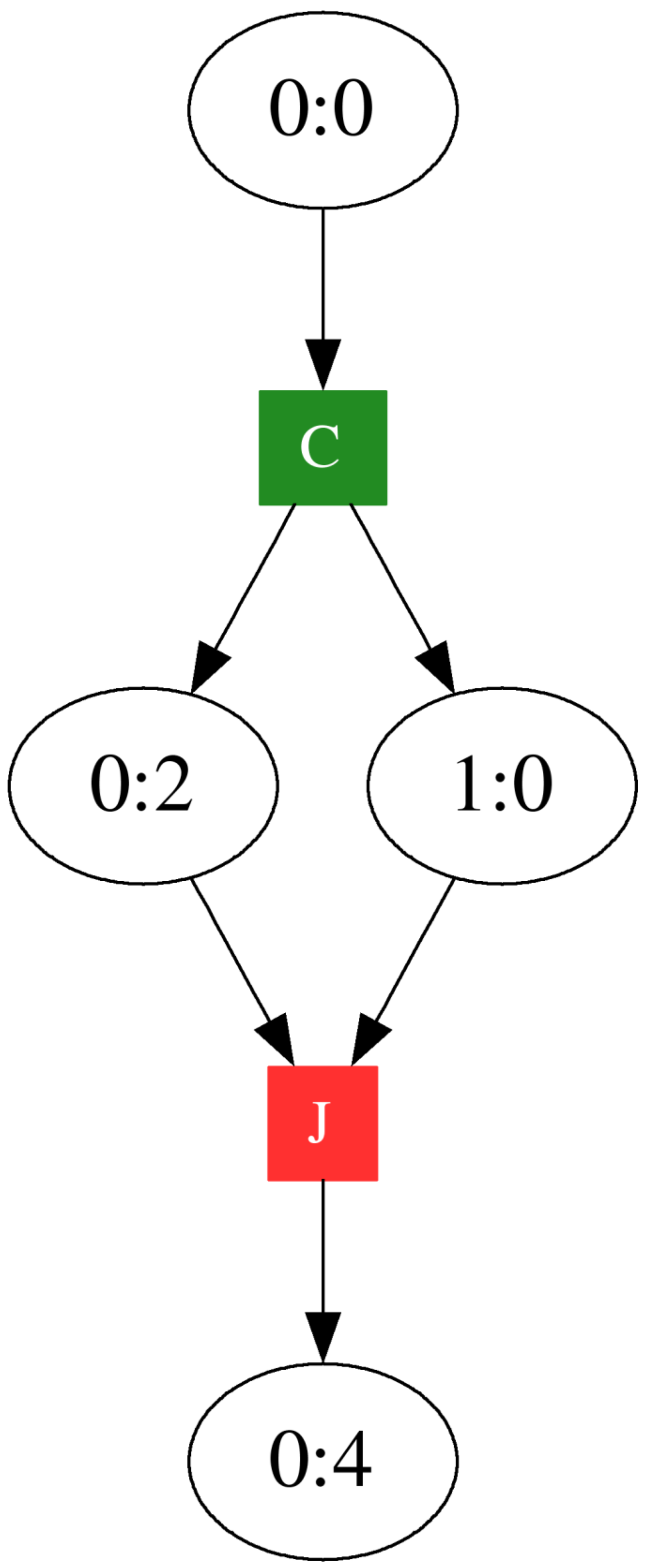
Task Graph Example

```
int fib(int n) {
  if (n < 2)
    return n;
  int a = cilk_spawn fib(n-1);
  int b = fib(n-2);

  return a + b;
}

fib(2);
```



Task Graph Example

```
int fib(int n) {
  if (n < 2)
    return n;
  int a = cilk_spawn fib(n-1);
  int b = fib(n-2);
  cilk_sync;

}

fib(2);
```



Parallel Program Diversity

- Language Diversity
  - **C, C++, Fortran**, Java, Go, Rust, X10, . . .
- Runtime Diversity
  - **Pthreads, OpenMP, MPI,** *Cilk*, Galois, Legion, CnC, . . .
- Pattern Diversity
  - **Regular, pipelines,** *graphs*, Map-reduce, Gather-scatter, . . .
- Architecture Diversity
  - **32- / 64-bit x86, ARM**, MIPS, Power, . . .