Today

- Two interesting and sophisticated abstract domains

Context

- **A real-time system is one where the correctness of a result depends not only on the result itself but also on the time at which it is delivered**
  - **Examples?**

Real Time Problem #1

- Garbage collected languages like Java offer significant benefits
- We would like to use these languages in embedded and real-time systems
- It is possible to create garbage collectors whose worst-case collection latency is bounded
- But: Also need to bound the number of times the garbage collector is invoked
  - How to do this?

Allocation Rate

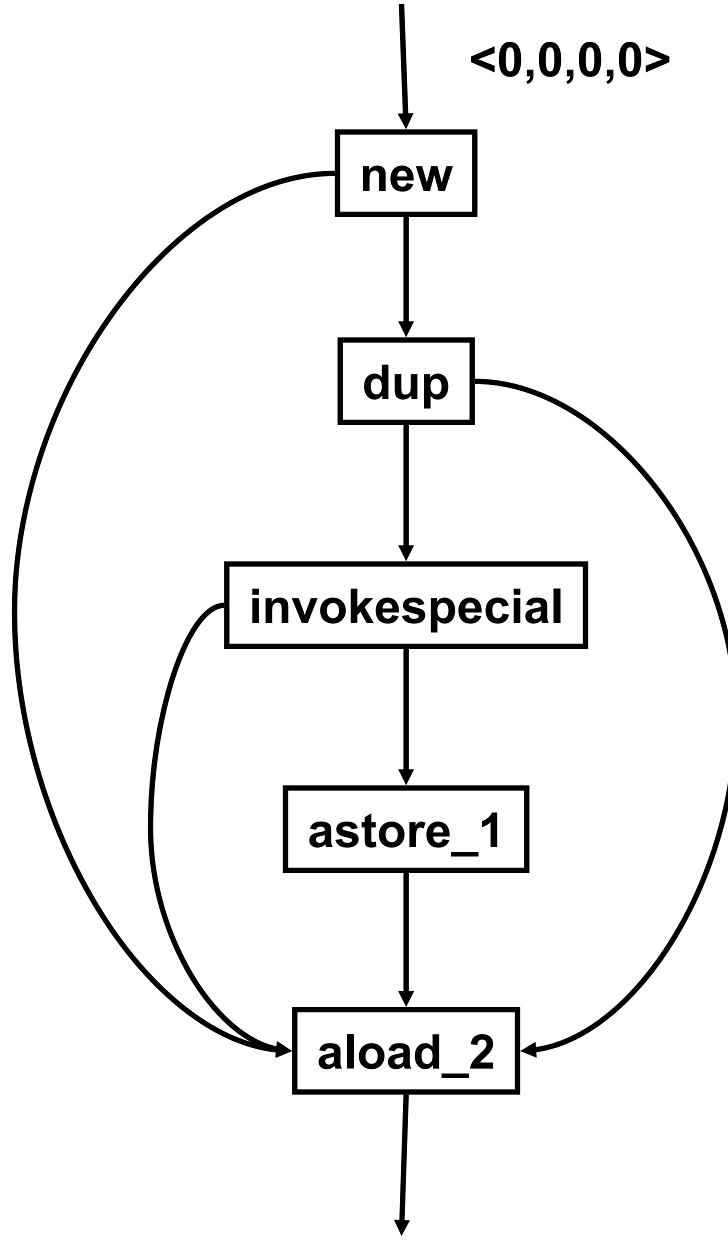
- Consider a sliding window of instructions
  - Here instructions are Java bytecodes
- Each instruction either allocates or not
  - For now we ignore the size of allocations
  - In other words, assume each allocation is of the maximum size
- Represent execution history as a bit vector
  - 0,1,0,0,1, …
  - Most significant bit is the most recent instruction

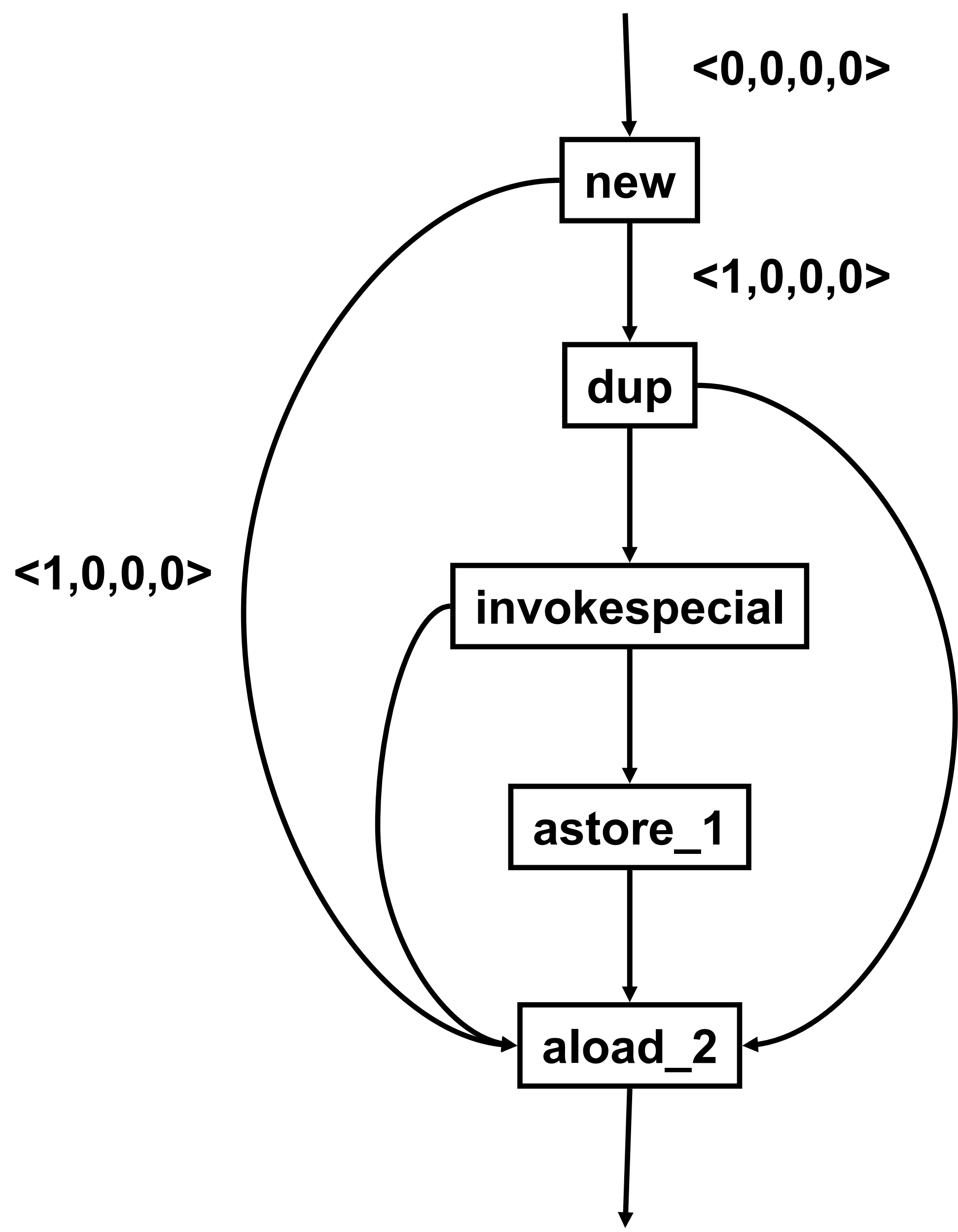
Allocation Rate via Dataflow

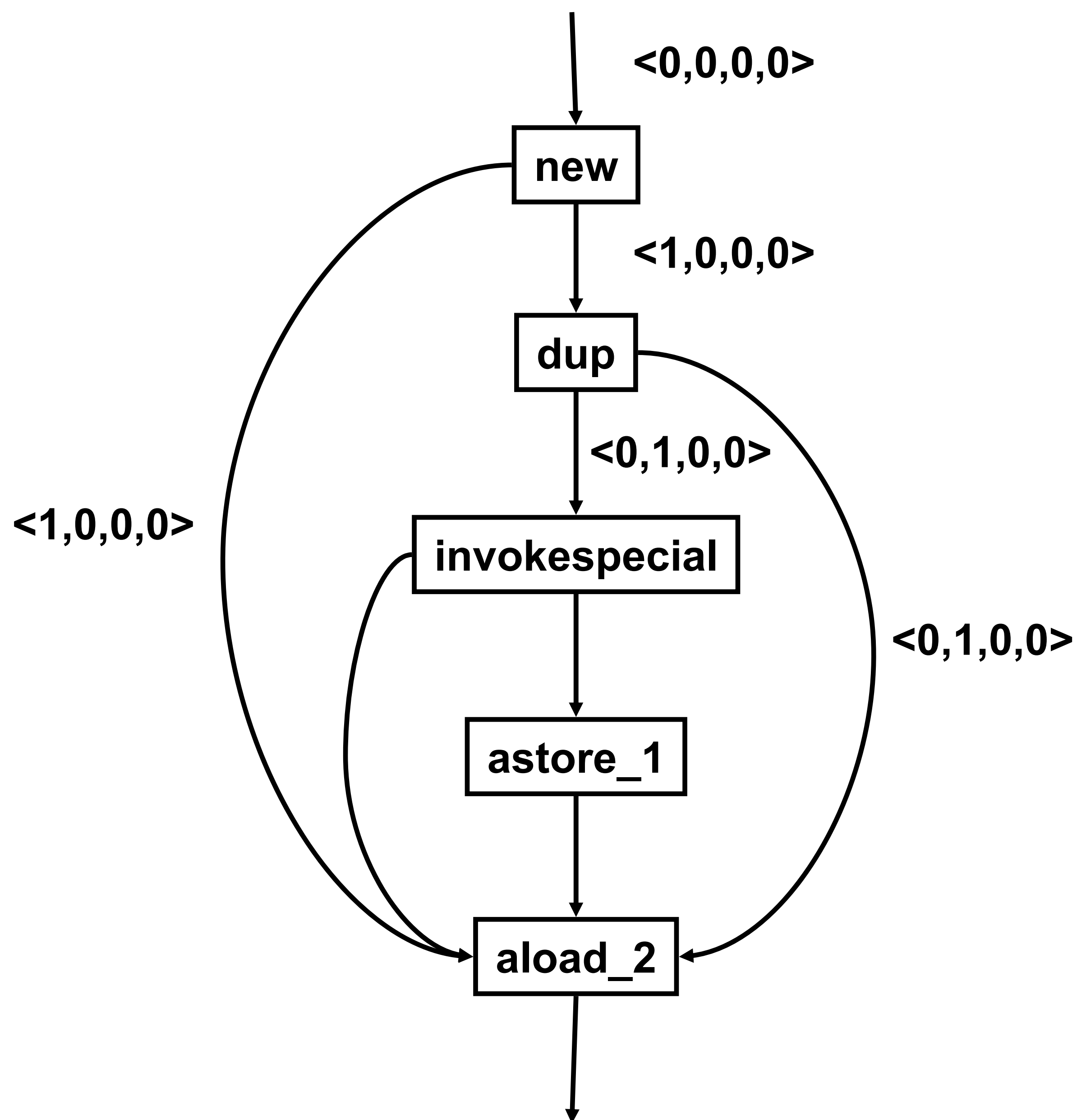
- $\bot$ = {0,0,0,0, … }
- $\top$ = {1,1,1,1, … }
- LUB operator is bitwise-or of two vectors
  - So for example {0,0,1,0} $\sqcup$ {0,1,0,0} = {0,1,1,0}
  - Can you argue that this is correct?

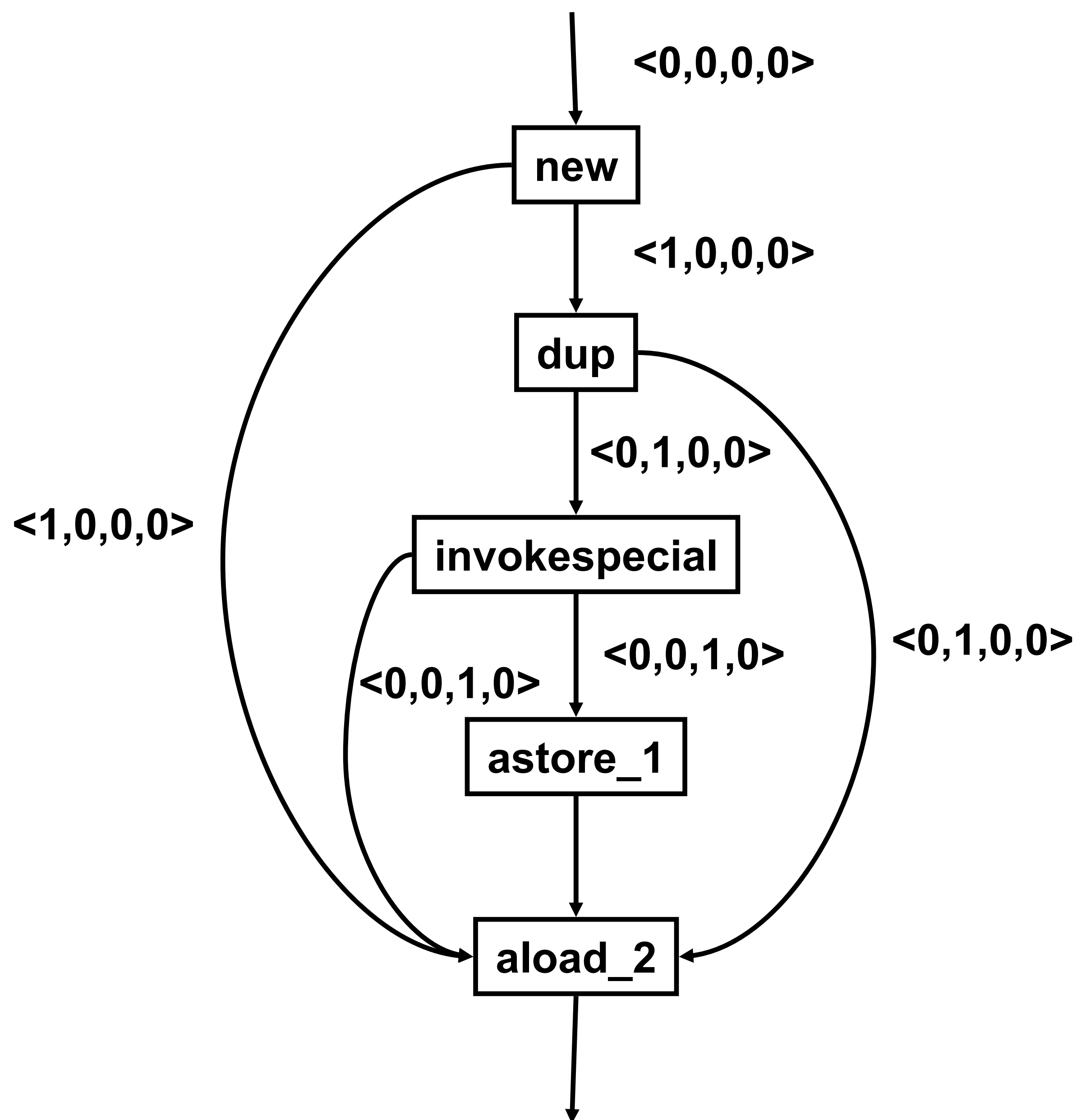
- Transfer function template:
  - Right-shift the abstract value by 1 bit position
    - Dropping the least recent bit
  - In the most significant bit position, put 1 if the current instruction allocates, 0 otherwise

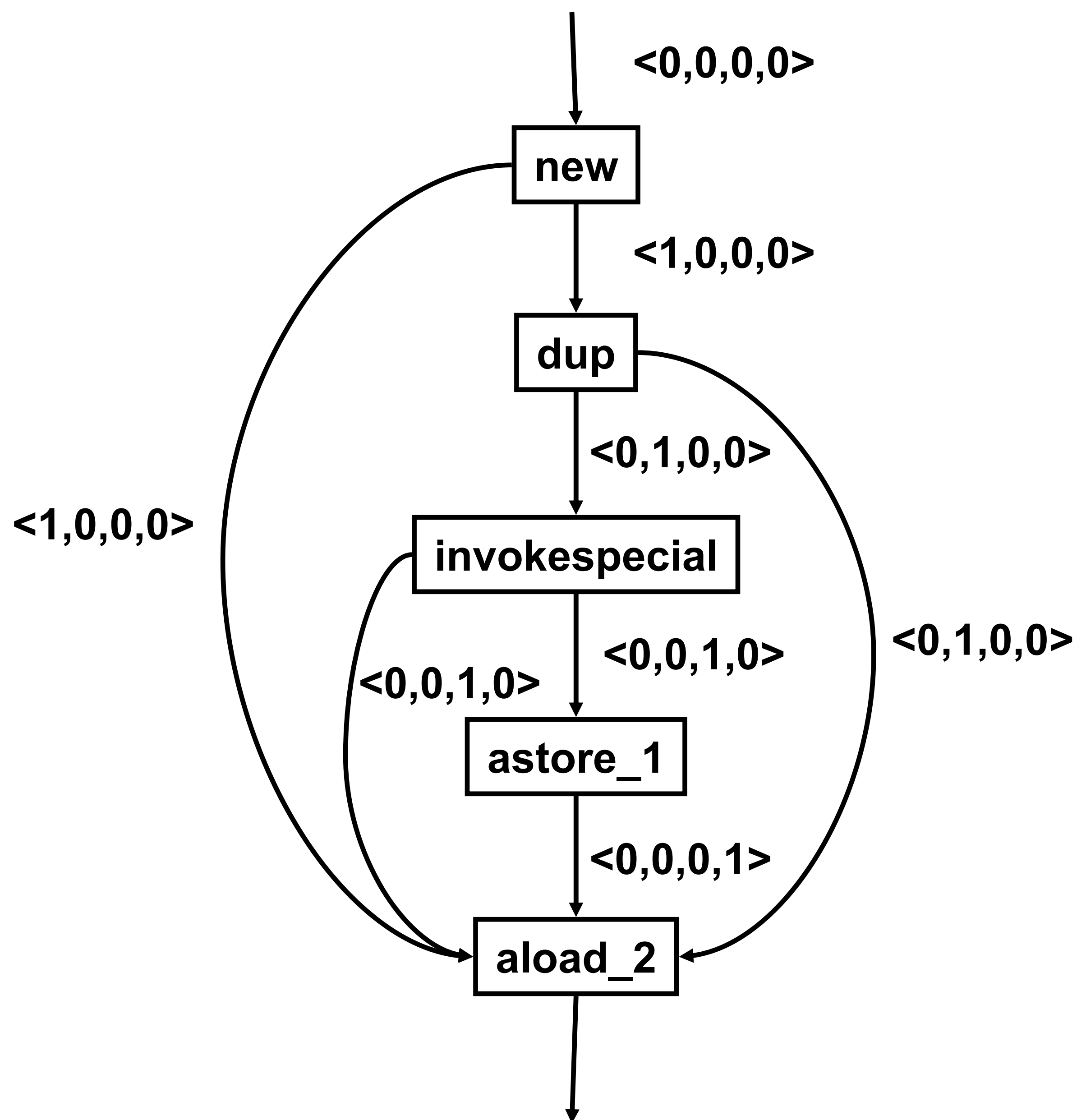
- **Result of applying simple analysis to real benchmarks: 15 out of 16 instructions allocate**
- **What is going on here?**

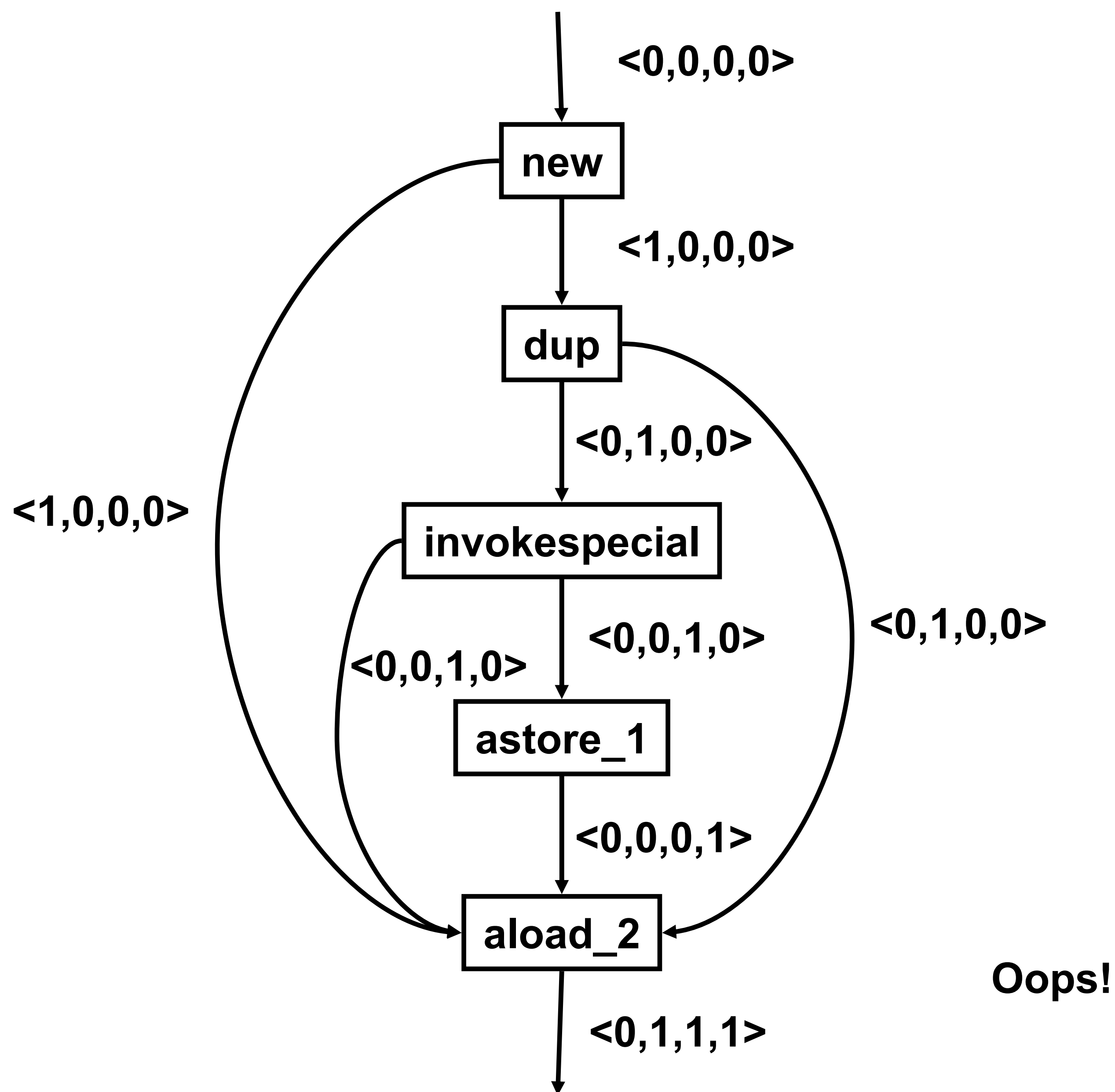

<0,0,0,0>
new
dup
invokespecial
astore_1
aload_2


<0,0,0,0>
new
<1,0,0,0>
dup
<1,0,0,0>
invokespecial
astore_1
aload_2


<0,0,0,0>
new
<1,0,0,0>
dup
<0,1,0,0>
<1,0,0,0>
invokespecial
<0,1,0,0>
astore_1
aload_2


<0,0,0,0>
new
<1,0,0,0>
dup
<0,1,0,0>
<1,0,0,0>
invokespecial
<0,1,0,0>
<0,0,1,0>
<0,0,1,0>
astore_1
aload_2


<0,0,0,0>
new
<1,0,0,0>
dup
<0,1,0,0>
<1,0,0,0>
invokespecial
<0,0,1,0>
<0,1,0,0>
<0,0,1,0>
astore_1
<0,0,0,1>
aload_2


<0,0,0,0>
new
<1,0,0,0>
dup
<0,1,0,0>
<1,0,0,0>
invokespecial
<0,1,0,0>
<0,0,1,0>
<0,0,1,0>
astore_1
<0,0,0,1>
aload_2
Oops!
<0,1,1,1>

A Better Join

- **Idea:**
  - **Let $<0,1,0,0> \sqcup <0,0,0,1> = <0,1,0,0>$**
- **Why is this ok?**
- **What is the general algorithm?**

A Better Join

- Scan the input vectors from most to least significant
- At each bit position, compute the output bit as the bitwise-or of the input bits
- However: After computing a 1 result, find the left-most 1 in each input vector and zero it
- So: <0,1,0,0,1> ⊔ <0,0,1,1,0> = <0,1,0,1,0>

- Insight: It is always OK to assume that an allocation happened more recently

Accounting for Allocation Size

- Most allocations are small, on the order of 12 bytes
- Most of the time, computing allocation sizes is easy
  - Just look at the types involved
- Dynamically sizes arrays are hard
  - Use interval domain analysis to compute safe upper bounds on array sizes
  - In real-time and embedded software, if array size is very difficult to analyze, the code is probably designed poorly

Refined Abstract Domain

- $\bot$ = {0,0,0,0, … }
- $\top$ = {M,M,M,M, … }
  - Where M is the maximum allocation size
- What's the LUB function for these?

LUB for Allocation Vectors

- Scan left to right
- At each position, the result is the maximum of the value in either input
  - But, subtract this value from the left-most non-zero value in each input vector
  - If the left-mode non-zero value is not big enough, zero it and keep scanning

- < 0, 0, 16, 4, 8, 4> ⊔
  < 0, 8, 16, 0, 4, 0>

- $\langle\ 0,\ \ 0,\ 16,\ \ 4,\ \ 8,\ \ 4\rangle \sqcup$

  $\langle\ 0,\ \ 8,\ 16,\ \ 0,\ \ 4,\ \ 0\rangle =$

  $\langle\ 0,\ \ldots$

- ```
  < 0,  8,  8,  4,  8,  4> ⊔
  < 0,  8, 16,  0,  4,  0> =
  < 0,  8 …
  ```
  ```

- < 0, 8, 16, 0, 4, 4> ⊔
  < 0, 8, 16, 0, 4, 0> =
  < 0, 8, 16, …

- $\langle 0, \ 8, \ 16, \ 0, \ 4, \ 4 \rangle \sqcup$

  $\langle 0, \ 8, \ 16, \ 0, \ 4, \ 0 \rangle =$

  $\langle 0, \ 8, \ 16, \ 0, \ 4, \ 4 \rangle$

Results

- With window size 512, worst-case allocation rate is no more than 2.5 times worse than observed allocation rate, for most benchmarks
- Is this an acceptable result? A good one? What are the limitations? What could be improved?
- The work is:
  - Static determination of allocation rates to support real-time garbage collection. Tobias Mann, Morgan Deters, Rob LeGrand, Ron Cytron. LCTES 2005: 193-202

Real-Time Problem #2

- We want to automatically and conservatively estimate WCET (worst case execution time) for embedded software
- Have to look at object code to do this effectively
- WCET estimation is…
  - Easy for simple code on simple hardware
  - Hard or impossible on complex HW
    - P4, Athlon, etc.
  - Hard or impossible for complex code
    - Linux kernel, Windows Media Player, etc.

Domain for WCET

- Abstract value is:
  - Memory / register values in the constant domain
    - Each location is either constant or top
  - Cycle count
- What's the LUB function?
- What is likely to happen when we apply this domain to real code?

- **LUB function:**
  - $WCET_{OUT} = \max(WCET_1, WCET_2)$
  - **Foreach storage location:**
    - $MEM_{OUT} = \top$ if $MEM_1 = \top$ or $MEM_2 = \top$
      $\top$ if $MEM_1 \neq MEM_2$
      MEM_1 otherwise

- **Improve using path sensitive analysis: No merging of abstract values**
  - **This is pretty heavyweight**

- **Global path sensitivity is infeasible, so we choose to collapse paths**
  - **At end of each function**
  - **At end of each loop iteration**

- What happens when we analyze this code with b unknown?

```
if (b < 100) fun1();
if (b > 200) fun2();
```

- How to fix this?

Supporting Real CPUs

- Add to the abstract state:
  - Model of the cache
  - Model of the processor pipeline
- Call these the “timing state”
- What is the LUB function for timing state?

Pessimistic LUB

- Cache lines containing different values are considered invalid
- Pipeline stages containing different values are considered unknown
- In practice this is too pessimistic
  - How can we do better?

Optimistic LUB

- **Idea: If we knew which partial path was going to end up being the worst-case path, we could just throw away information from the other path**
  - **Of course, in general we don't know which path is going to end up as being the longer one**
  - **However, intuitively it is likely that the current longer one is going to end up longer**

Optimistic LUB

- Let $WCET_L$ and $WCET_S$ be the current cycle counts on the longer and shorter paths being merged
- Estimate the worst-case penalty $WCET_P$ that the short path would incur on future execution time, that would not be incurred on the long path
- If $WCET_L \geq WCET_S + WCET_P$ return the timing state from the long path and discard the timing state from the short path
- Otherwise, use the pessimistic LUB
- Of course we need to figure out how to compute $WCET_P$

Example $WCET_P$

- Assume that all cache blocks present in and $WCET_S$ but not present in $WCET_L$ are used in the future
- $WCET_P$ = Number of such blocks multiplied by cache miss penalty

More Details

- **Lots left to work out:**
  - **Pessimistic merge for pipeline state**
  - **Pessimistic merge and $WCET_P$ computations for multi-way caches**

Results

- Estimated WCET is very close to (believed) actual WCET for many small benchmarks
  - Matrix multiply, sorting routines, crypto, etc.
- Estimated WCET is much larger than (believed) actual WCET for data compression
- Data-dependent code makes WCET very hard to compute
  - But most well-designed real-time software is not very data dependent!
- Work is:
  - An Integrated Path and Timing Analysis Method based on Cycle-Level Symbolic Execution. Thomas Lundqvist and Per Stenström. Real-Time Systems 17(2-3): 183-207 (1999)

Summary

- General idea that we saw two times:
  - Merge operation (LUB) is where pessimism comes in
  - A clever LUB is worth a lot and may determine whether an analysis is effective or not
- Additionally, in the second example we saw that merging can be avoided by adding path or context sensitivity
  - Unless used very carefully these make an analysis blow up on some codes