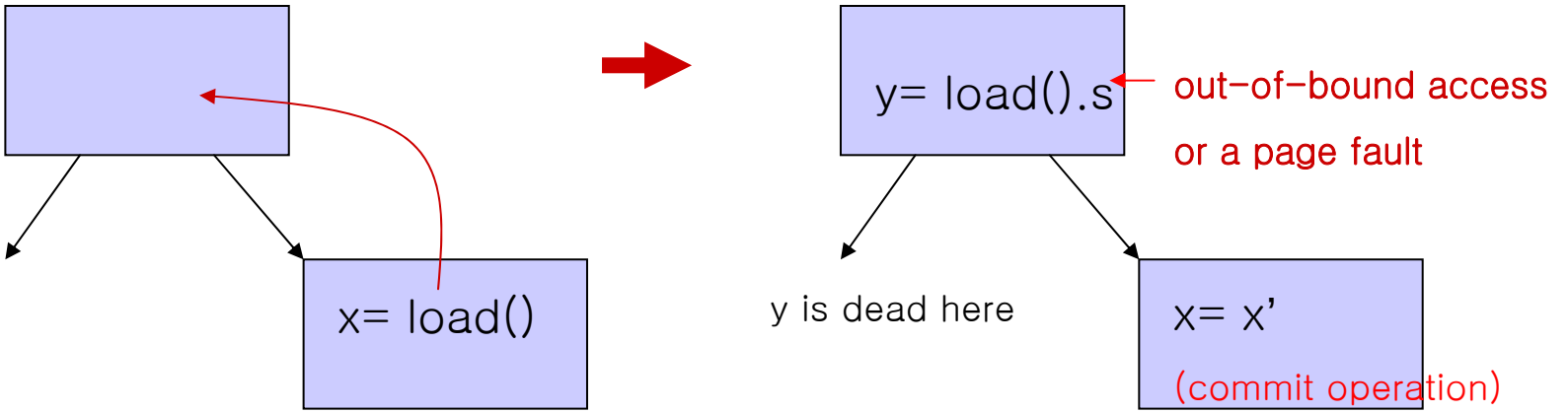
Handling of Compatibility Issues

Speculative Exception Problem

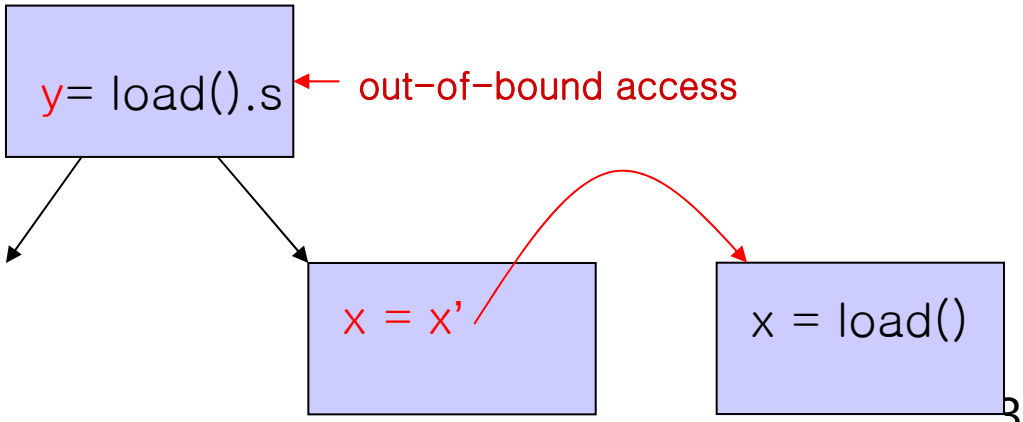
- If a speculative instruction causes an exception,
  - It is a legitimate exception if its original path is taken
  - It is a bogus exception if its original path is not taken
  - So, we should wait until the execution path is known



- Static compilers had the same problem
  - It is often handled by a dirty bit and recovery code (e.g., in IPF)

Static Compiler's Solution: Recovery Code & Dirty Bit

- If a speculative instruction raises an exception,
  - Instead of raising a trap, H/W will set a dirty bit (33th) for y
    - The dirty bit can be propagated via other dependent speculative ops
  - If the original path is not taken, it will be fine
  - If the original path is taken such that y is used by a non-speculative instruction (or commit), jump to recovery code
    - A non-speculative version of the load is executed again, causing a real exception this time



Dynamic Optimizer's Solution: Commit/Roll-back in Crusoe

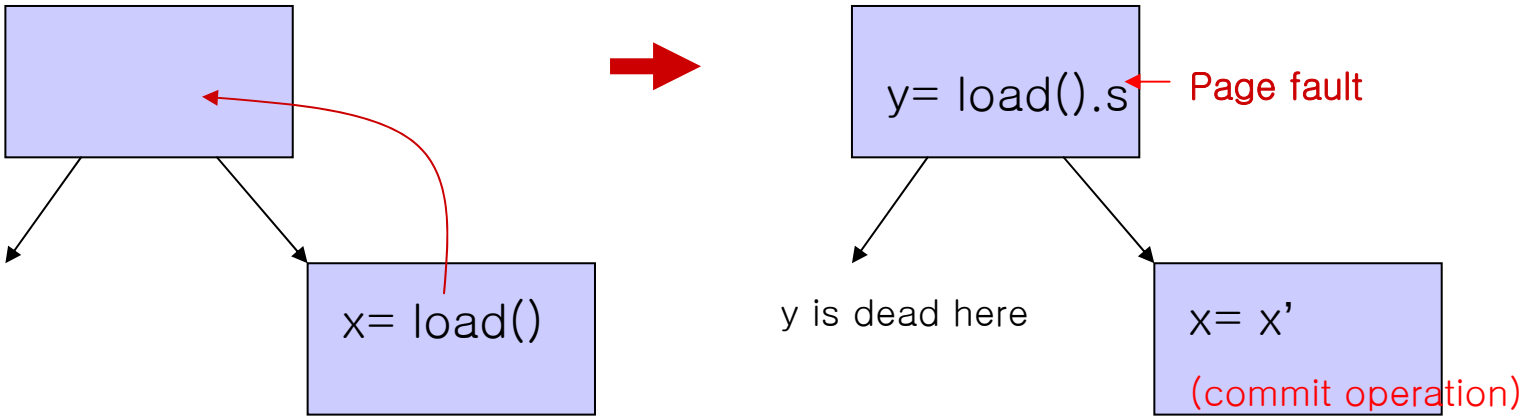
- Each x86 register has two copies in Crusoe
  - A working copy and a shadow copy
  - Translated code updates only working copies
- At the end of execution for a translated block T,
  - A special "commit" is performed to commit work done in T
    - It will copy all working registers to their shadow registers
- If any exception occurs in a translation T,
  - A special "roll-back" is performed to undo work done in T
- Commit/Roll-back also work for memory stores
  - Store data are held in store buffer and released on commit
- After roll-back, we interprets x86 instructions
  - If the exception is genuine, it will occur again

Precise Exception with Commit/Roll-back

- Commit/roll-back can support precise exception while allowing aggressive code scheduling
  - Interpretation of x86 allows all preceding instructions to be executed while none of following instructions are executed
  - We can know the address of excepting instruction
  - Works even with aggressive speculative code motions
- For frequently recurring speculative faults, we can retranslate the region more conservatively
  - After cutting the translation into smaller regions

Dynamic Optimizer's Solution: Commit Instruction in DAISY

- There are two types of registers in DAISY
  - Base-architected registers (PPC) and non-architected ones
- When we make a code motion, we rename it with a non-architected register with a commit op left there



- In this way, assignments to architected registers (and stores) are committed in the original program order

The Case of DAISY (2)

- If there is an exception during execution of VLIW, it is delayed until a commit operation (using a dirty bit)
  - At the commit operation, the architected state will be the same as when the original PPC instruction excepts
    - Due to its in-order commit property
  - DAISY can find the address of the PPC instruction
    - By consulting some tables and searching VLIWs
    - The address of the original PPC instructions were saved in VLIWs
  - Interpretation start from the PPC instruction
    - which may except genuinely, reported to the kernel of the Base OS
  - No recovery code needs to be generated

Comparison of DAISY and Crusoe

- DAISY can have a larger region and a simpler H/W
  - Crusoe's region can be limited by the size of store buffer
  - DAISY does not require shadow registers

- Crusoe allows much more aggressive optimizations
  - No need for preserving precise exception in a region
  - No need for commit operation or dirty bit