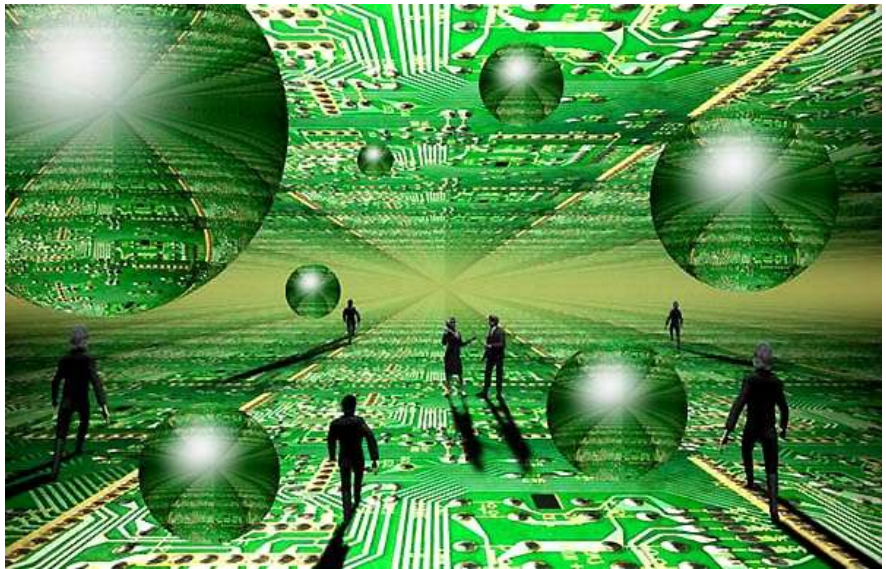
NYU
New York University

CSCI-GA.3033-015

Virtual Machines: Concepts & Applications

Lecture 8: Co-Designed VMs

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

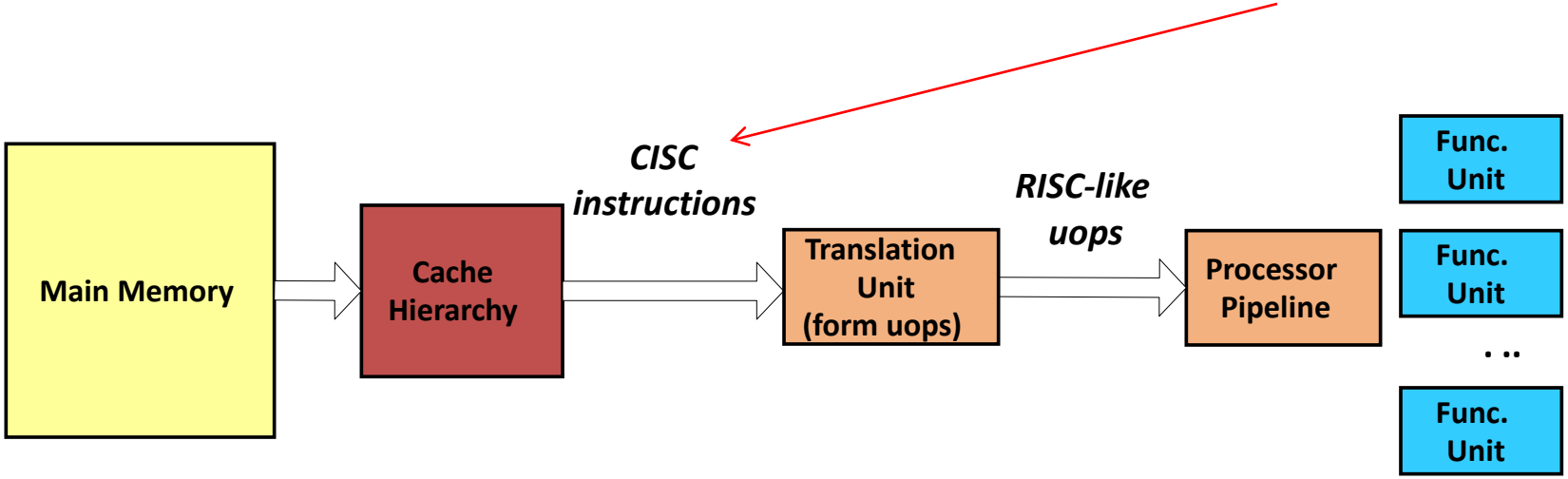


The ISA

- The main interface between hardware and software
- Hardware and software evolved over the years.
- ISA cannot evolve as such because of backward compatibility
- Consequently, ISAs currently in use reflect a perspective and a division of labor between hardware and software that is decades old.

CISC-to-RISC

underlying processor hardware has grown to be quite different from the image presented by the commonly used ISAs



Conventional superscalar implementation

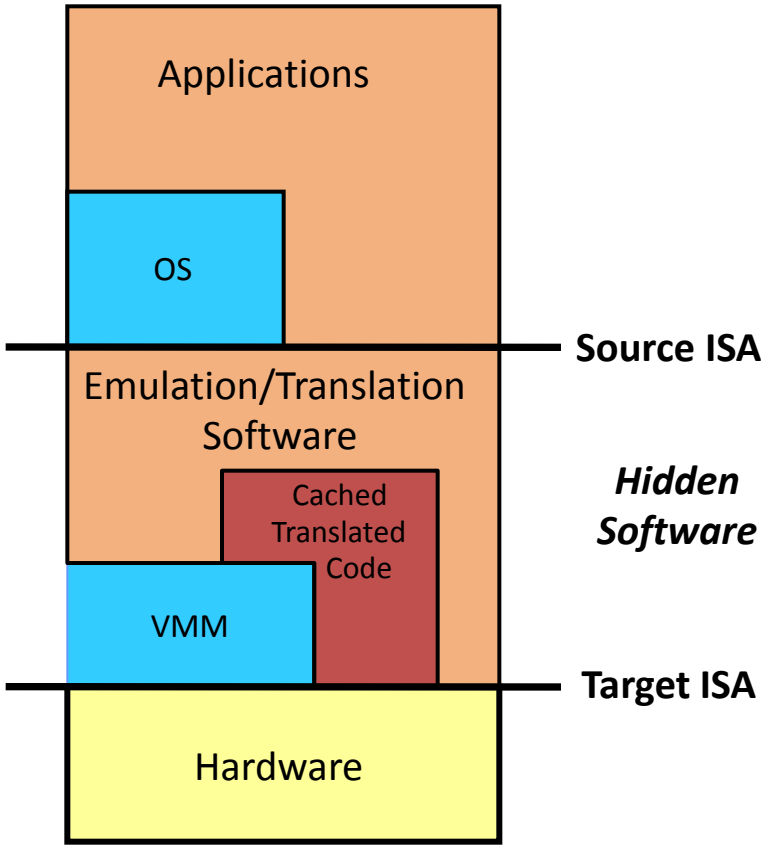
How can VM Help?

- VM technologies permit new ISAs by enabling a different approach to general-purpose processor design.
- Host architecture (the target) is designed concurrently with the VM software that runs on it.
- The interface between hardware and conventional software is shifted upward.
- A special kind of system VM?

System VM but …

- Are not intended to virtualize hardware resources other than the processor.
- Are not intended to support multiple VM environments.
- Goals include:
  - performance
  - power efficiency
  - design simplicity.

Co-Designed VMs



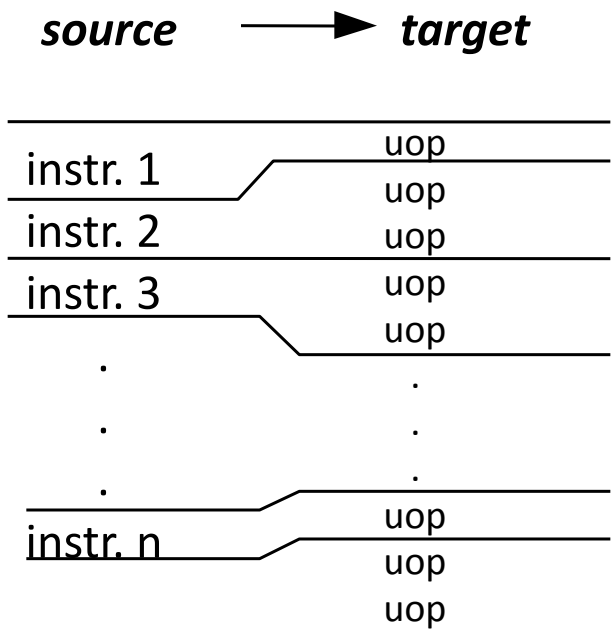
Similar to process VM in:

- emulation done in stages
- code cache for dynamic translation

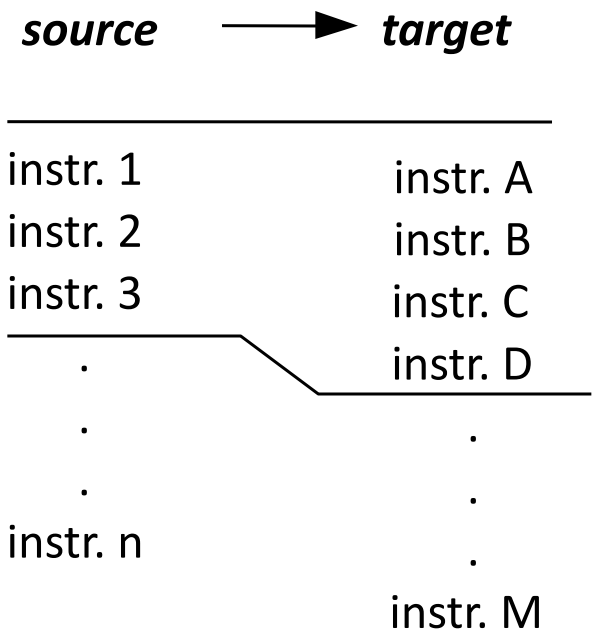
Different from process VM in:

- Intrinsic compatibility at ISA level
- The main goals are performance, power efficiency, design simplicity, or combination of that.

Hardware Vs Software Translation



context-free translation to uops



context-sensitive translation to target ISA.

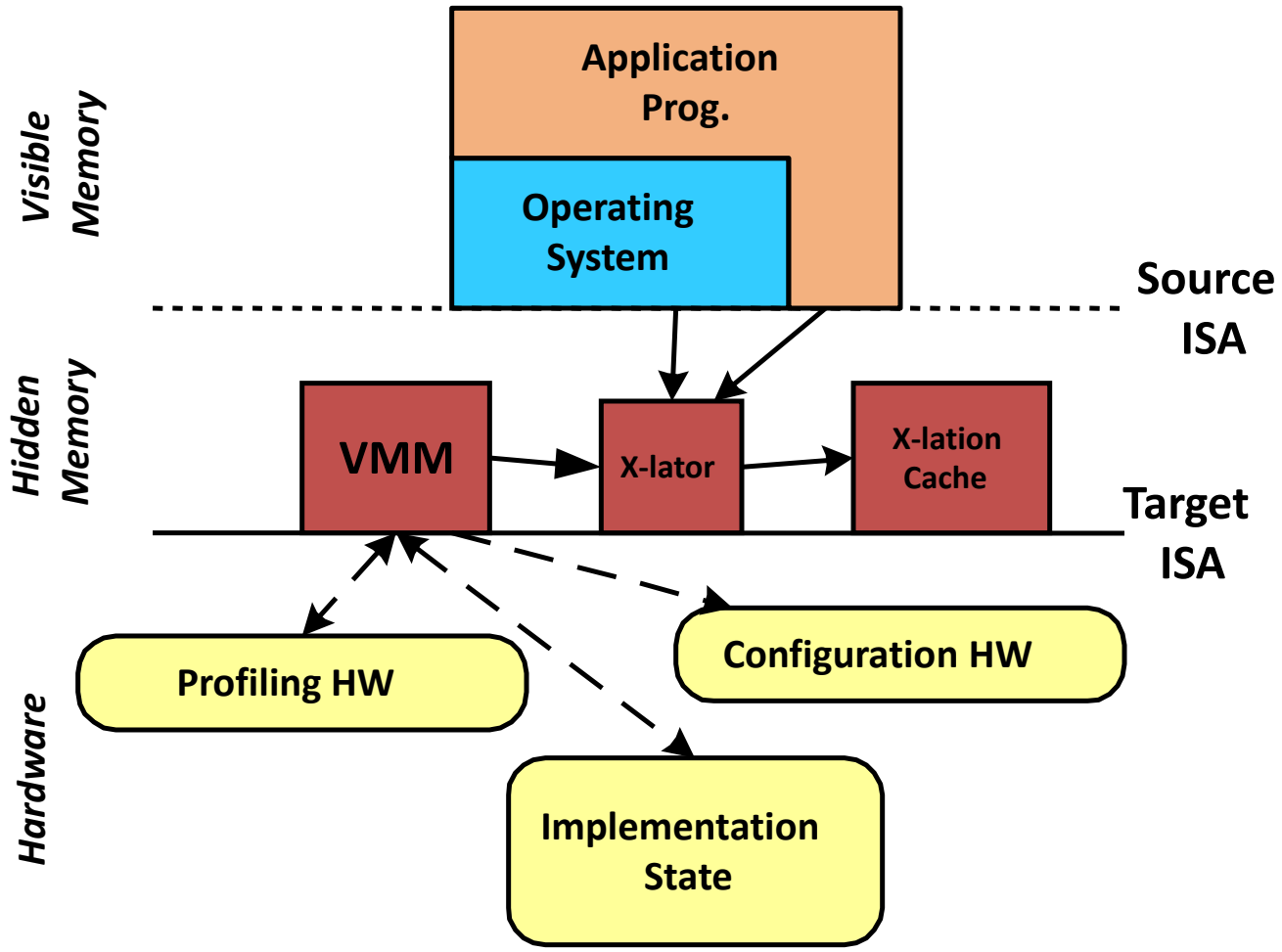
Hardware Vs Software Translation

- Hardware solution leads to higher cost of design verification
- Higher power consumption due to the added complexity
- Not flexible

What Do We Gain?

- Can implement new high performance ISA
  - ISA can be technology- and application-dependent
    - VLIW
    - (re-invented) Superscalar
    - Distributed Microarchitectures
- Enhancing existing ISA
  - Add functional instructions (and still benefit existing binaries)
  - Delete instructions (and still run existing binaries)
- Managing adaptive hardware
  - Add special "sensor/actuator" instructions
    - Read dynamic performance data from profile registers
    - Write configuration control registers

Co-Designed VMs: Big Picture

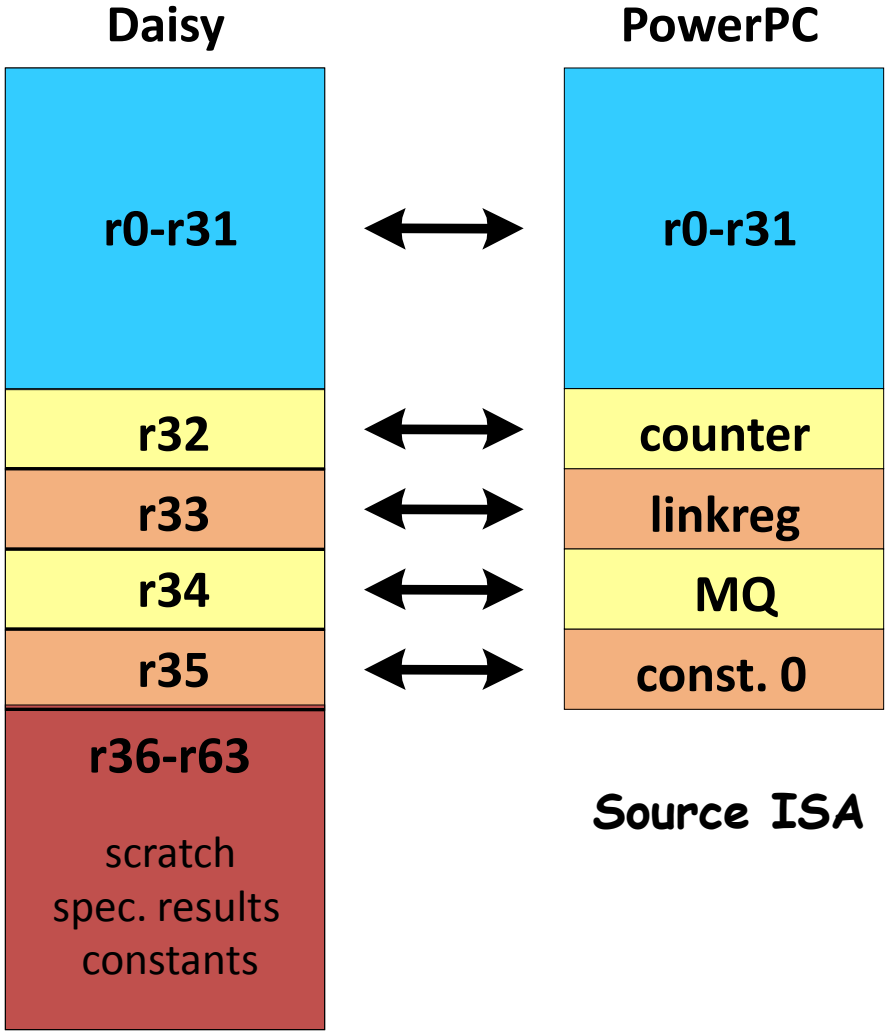


Register Mapping

- Situation here is easy
- target ISA is designed specifically for the source ISA
- host register file(s):
  - can be made large enough to accommodate the guest's requirements
  - extra scratch registers left over to enhance performance and/or simplify the translation process

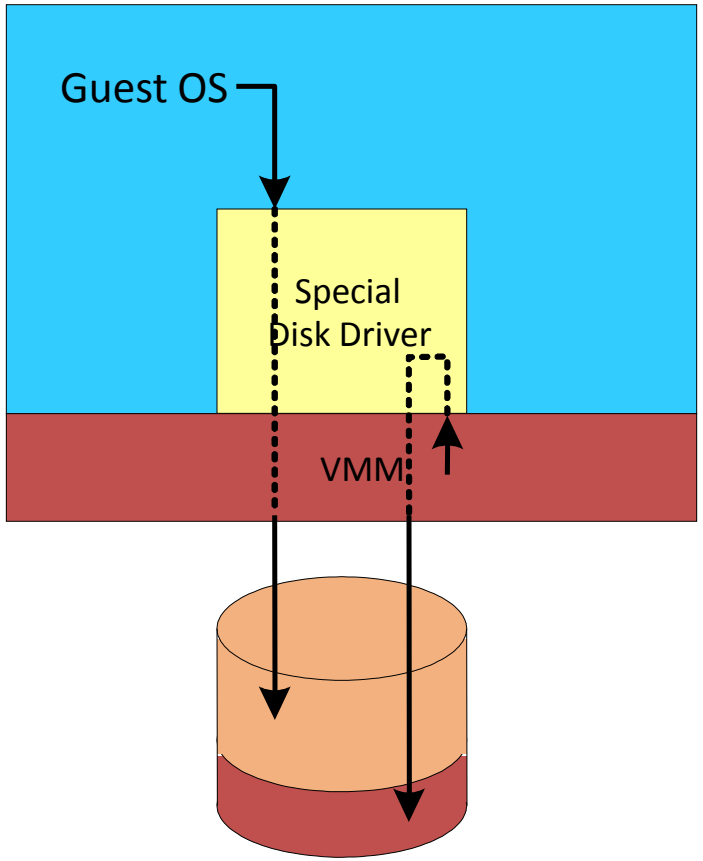
Register Map: Example

- 0-31 hold PPC regs.
- 32-34 hold other PPC state values
- 35 holds 0
- 36-63 hold scratch values, constants, speculative values

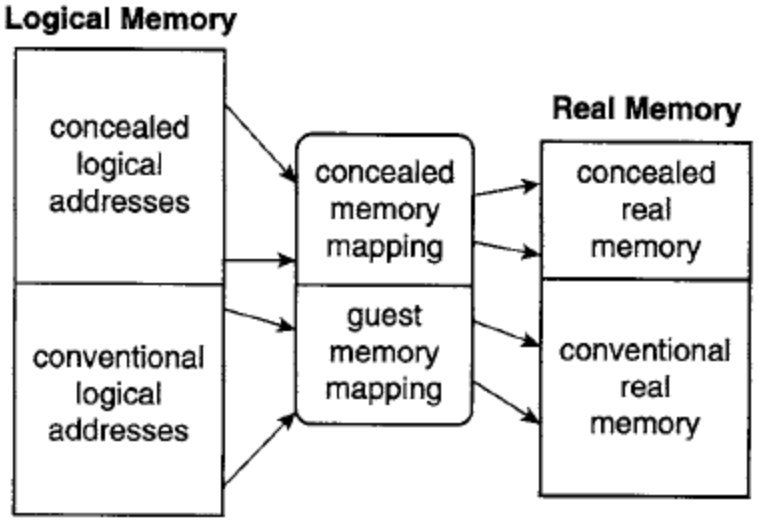


Concealed Secondary Memory

- Allows persistent caching of translations
  - Faster startup
  - Larger code cache
- Concealing main memory from OS is relatively easy
  - Just lie to it at boot time
- Concealing secondary memory can be done
  - Install special disk driver that conceals part of disk



Memory Architecture: Option 1

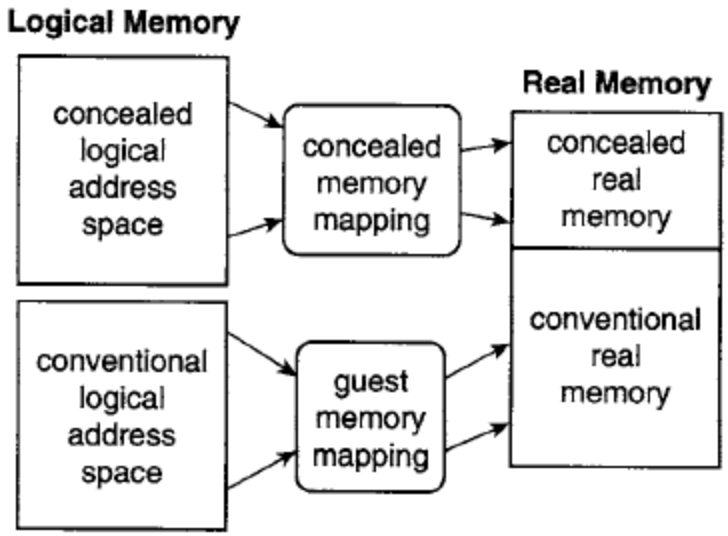


- Concealed logical memory shares an address space with the guest.
- The host address space must be enlarged, with the guest space fitting inside.

+ straightforward

- may make the host ISA awkward or expensive to implement

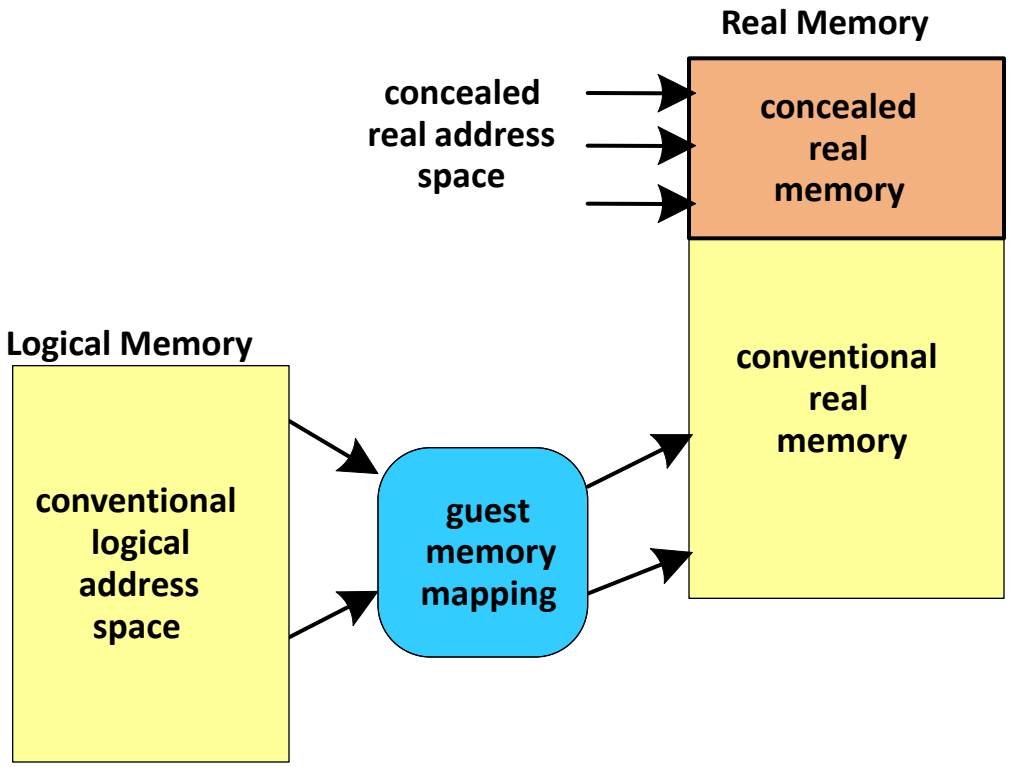
Memory Architecture: Option 2



- This approach implies that load and store instructions must select one of the two mapping tables

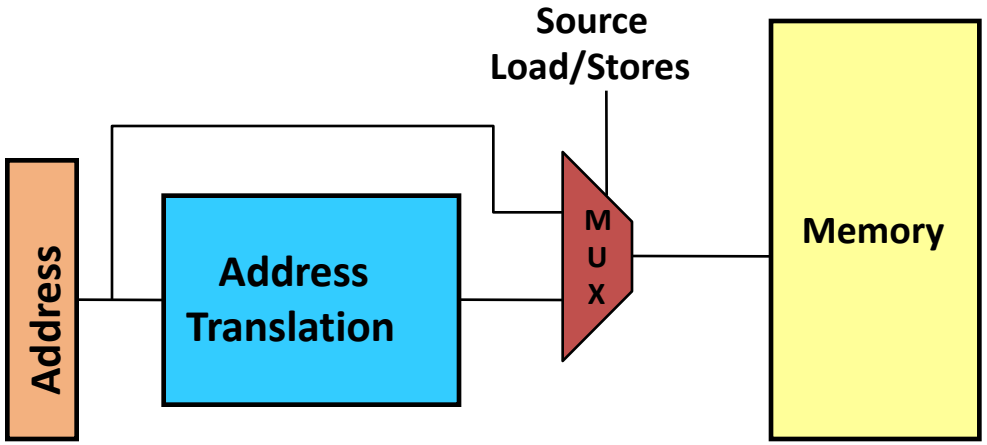
Memory Architecture: Option 3

- Let guest OS manage conventional memory
- Place VMM in separate real space



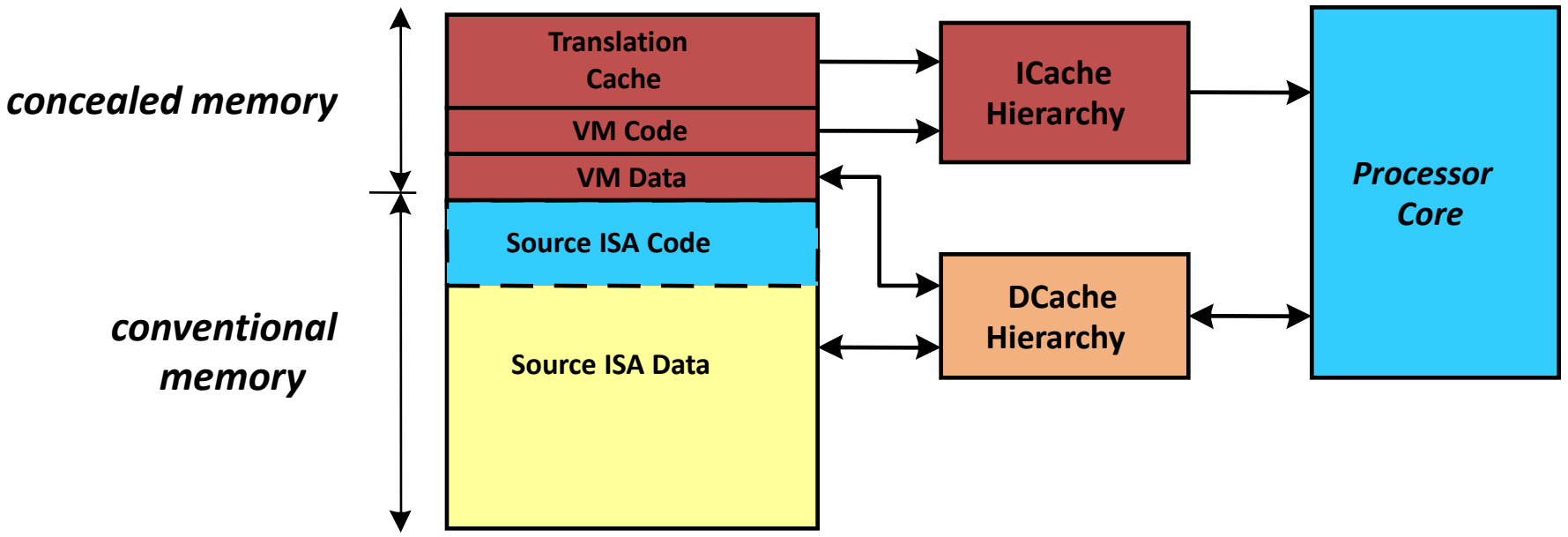
Memory Architecture: Option 3 (cont'd)

- Real Addresses
  - VMM Code
  - VMM Data
  - Translated Code (code cache)
- Virtual Addresses
  - Source ISA loads and stores
    - to both data and code areas
- Select via opcode or mode bit



Concealed Memory: The Key to Memory Mapping

- VM software resides in memory concealed from all conventional software
- The VMM takes control immediately after system reset
- It essentially has control of the system from the beginning, including the boot process, so it can make sure the conventional software never sees the concealed memory.



Code Caching … Again

- Crucial for performance
- Summary of access:
  - hashing the source PC (SPC) value to an entry in a map table
  - reading a corresponding SPC value (or tag) from the map table, and performing a comparison
  - If hit, the target PC (TPC) at the map table entry can be used for accessing the code cache
  - If miss, additional probes of the table may be required in order to handle potential hash collisions

Code Caching … Again

- Map table lookup for direct jumps and branches is eliminated through chaining
- How about indirect jumps?
  - Depend on register value
  - Register values change during execution
  - Register holds SPC not TPC
  - Do we need to access map table for every indirect jump?

Code Caching … Indirect Jumps

- Doesn't software prediction solve the problem?

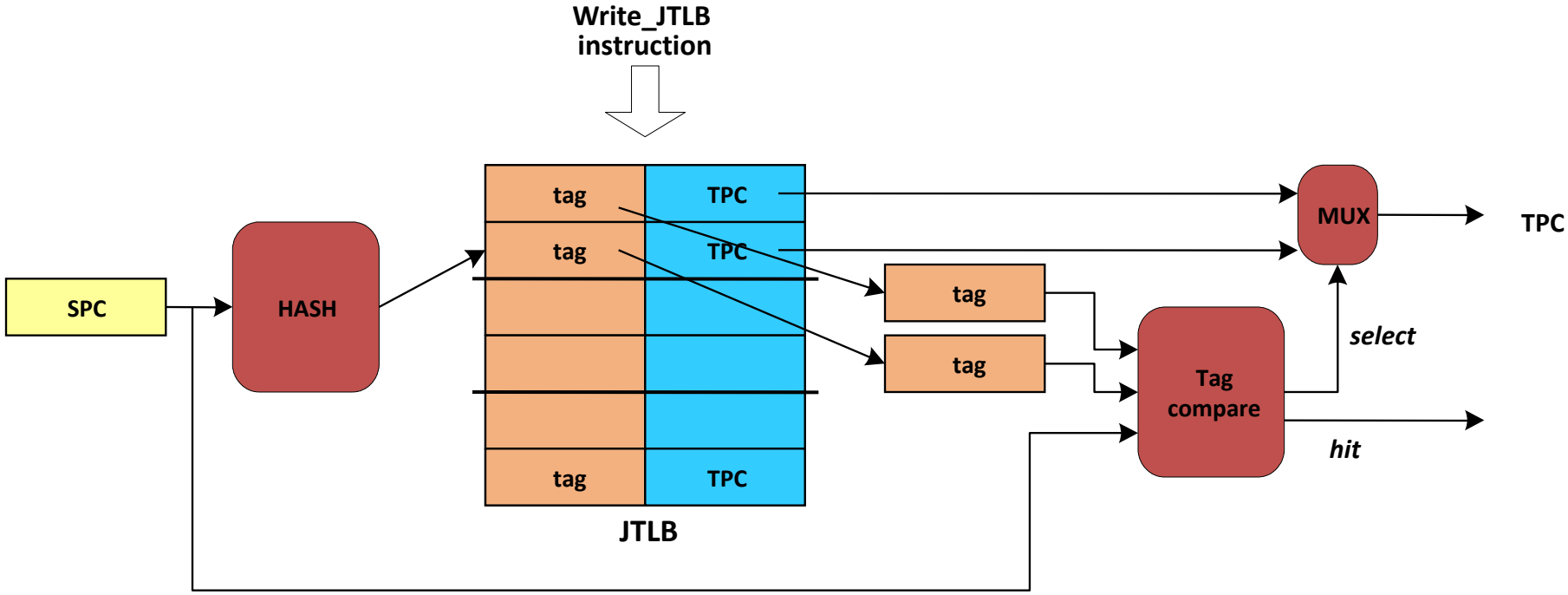
```
if ((Rx) == #addr_1) goto #target_1
else if ((Rx) == #addr_2) goto #target_2
else map_lookup (Rx)          ; do it the slow way
```

  - If it misses then more overhead is added beside map table access.
  - Some jumps are very hard to predict (e.g. return from procedures).
- The indirect-jump problem is probably the greatest source of performance loss in a software-only code cache system.

How do codesigned VMs solve this?

Support for Code Caching

- Add Jump TLB
  - A hardware cache of dispatch table entries
  - Similar to software-managed TLB in virtual memory
  - Entries are written into the JTLB by the VMM



Example of a two-way set associative JTLB

Incorporating JTLB in codesigned ISA

- Method 1:
  - JTLB_Lookup instruction that accesses the JTLB
  - SPC address held in a register
  - reads out a TPC and places it in a second register.
  - A third register (or condition code) indicates a hit or miss in the JTLB.

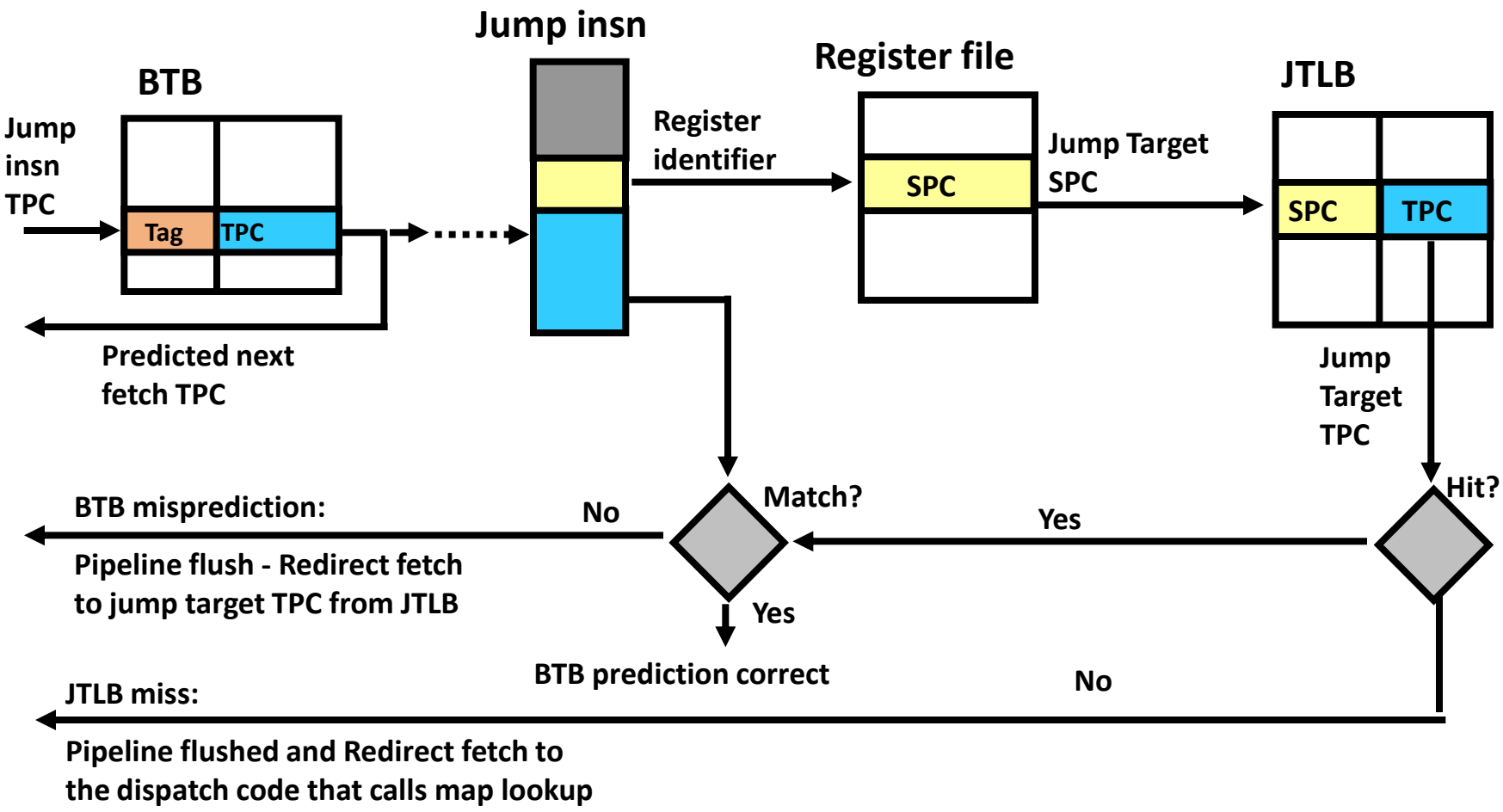
```
JTLB_Lookup Ri, Rj, Rk      ; TPC to Ri, hit/miss to Rj
Jump Ri, Rj==0              ; conditional indirect jump
Jump map_lookup             ; do it the slow way
```

Incorporating JTLB in codesigned ISA

- Method 2:
  - combine the lookup with the conditional jump
  - Lookup_Jump Rk performs the jump to the TPC if there is a hit, otherwise it falls through.
- when a jump is encountered, instruction fetching must stall until the JTLB is accessed and the jump is executed → bottleneck!

How about predicting the TPC value immediately after a Lookup_Jump instruction is fetched?

Jump TLB


Jump insn
BTB
Jump
insn
TPC
Tag
TPC
Predicted next
fetch TPC
Register file
Register
identifier
SPC
Jump Target
SPC
JTLB
SPC
TPC
Jump
Target
TPC
Hit?
Match?
BTB misprediction:
No
Pipeline flush - Redirect fetch
to jump target TPC from JTLB
Yes
Yes
BTB prediction correct
JTLB miss:
No
Pipeline flushed and Redirect fetch to
the dispatch code that calls map lookup

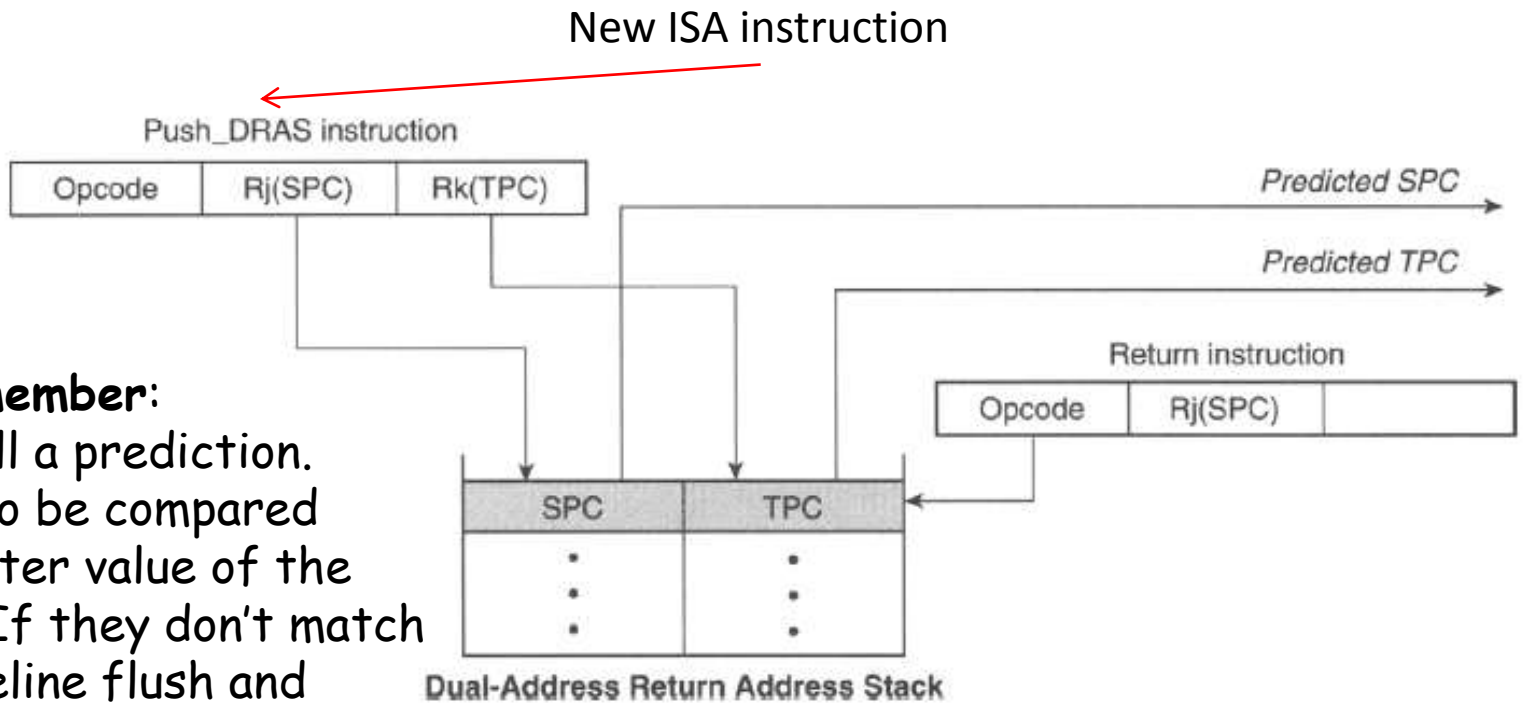
Does It Work?

- As long as BTB predictions are correct a high percentage of the time.
- It is usually so, except for procedure return jumps.
- Most modern processors employ a hardware return address stack (RAS) mechanism that can predict a return instruction's target address very accurately → It basically mimics the software procedure stack by pushing the fall-through PC onto a hardware prediction stack.
- Can we apply this in codesigned VM? Not right away, because:
  - the saved return destination address would be an SPC and we need TPC
  - If the procedure jump is at the end of a translated superblock then the address of the instruction following the jump is not the correct return address (the correct address is at the beginning of a different superblock).

What can we do?

Dual-address RAS

- Solution: save both SPC and TPC
- Sometimes we may need to put invalid TPC if it is not known at the time
- Later when the return target superblock is constructed, the invalid address is replaced with a valid TPC.



Remember:
This is still a prediction. It needs to be compared to the register value of the indirect jump. If they don't match then a pipeline flush and redirection to the map table.

Precise Exceptions

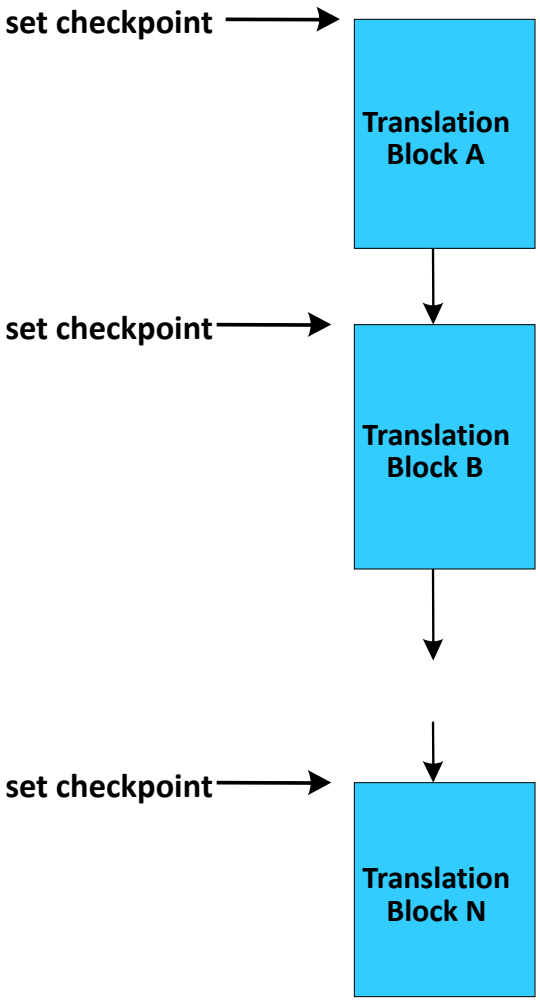
- Traps must be precise wrt original binary
  - All conventional software is unaware of underlying VM
  - Code may undergo heavy duty re-organization
    - E.g. CISC $\rightarrow$ VLIW
- Checkpoint and rollback
  - Have VMM periodically checkpoint state
  - Consistent with a point in original binary
  - On fault, rollback and interpret original binary

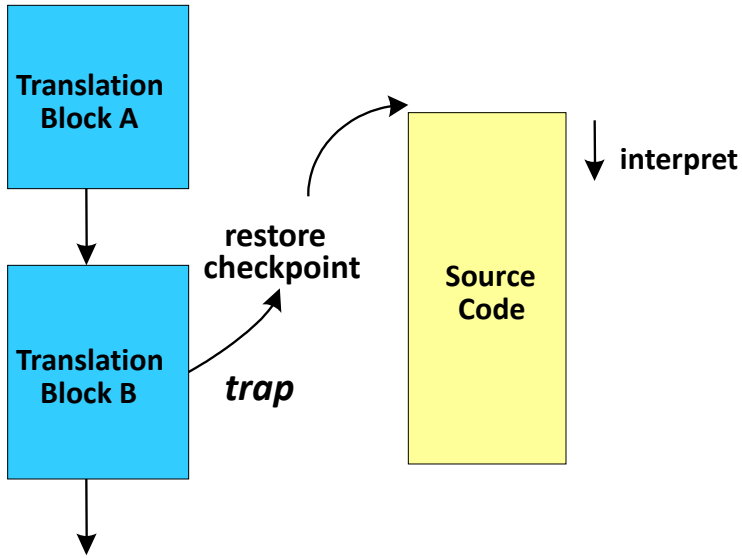
Do We Need Hardware Support?

- If code motion is involved:
  - register live ranges are extended to ensure that checkpoint values can be restored when there is a trap. Then interpretation beginning at a checkpoint prior to a trapping instruction,
  - In codesigned VMs this approach is facilitated because the host ISA can be designed to have enough registers so that live ranges can be extended without excessive register pressure.
- However, this software-based approach does limit the types of code motion that can be performed.
  - For example, many instructions cannot be moved below store instructions.
- By adding hardware support for checkpoints, this restriction on code motion can be removed

Checkpoint and Rollback

Checkpointing can be done in hardware





Page Faults

- Were not an issue in process VMs (why?)
- A codesigned VM is a system VM, not a process VM → The guest OS must observe exactly the same page faults as it would if it were running on a native platform.
- We have to deal with two types of page faults:
  - From conventional memory:
    - For data (no problem)
    - When interpreting instructions
  - From concealed memory
    - When executing translated instructions (more difficult!)

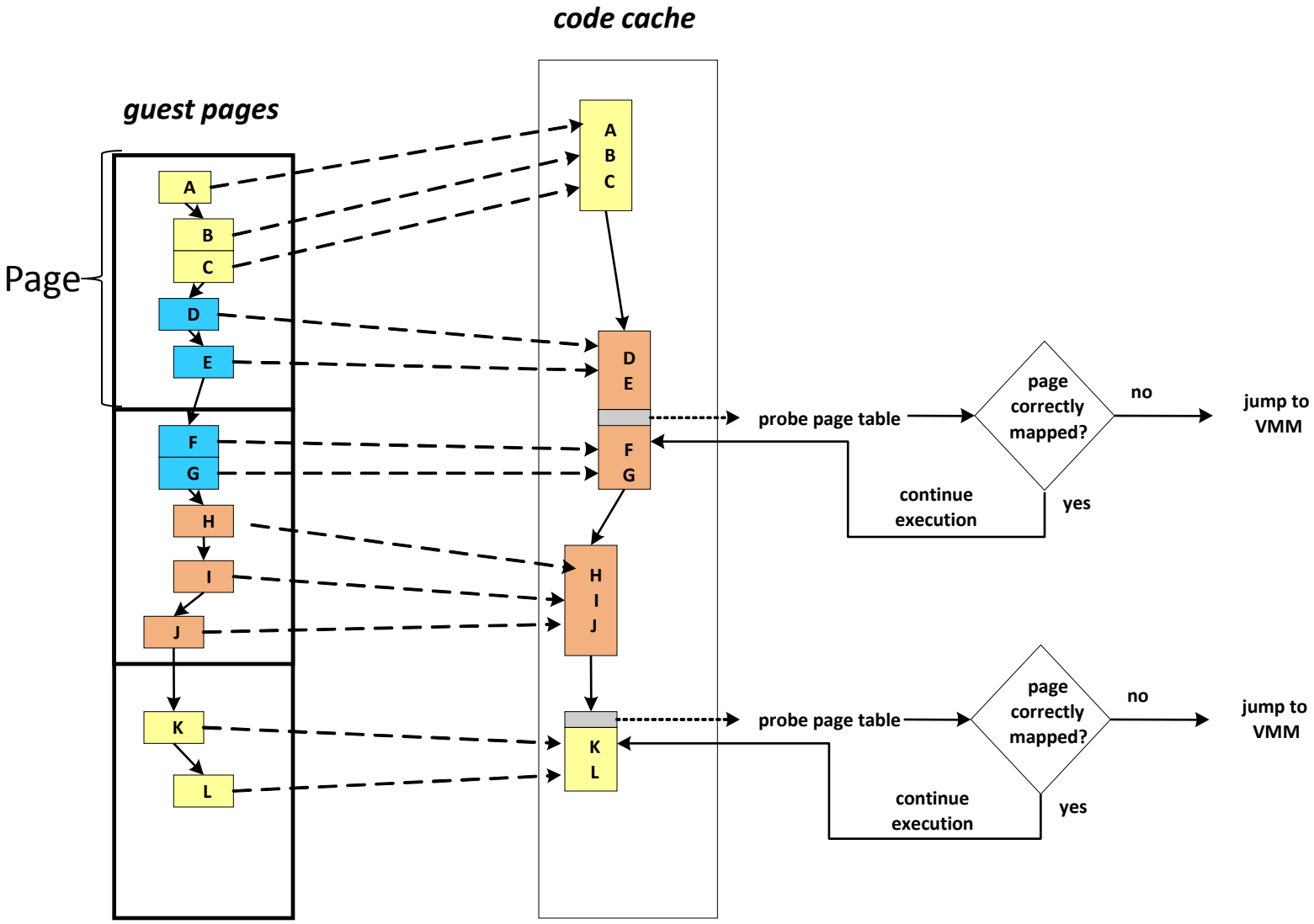
Active Page Fault Detection

- Monitor Guest Page Table updates
  - Make page table read only
  - *Also monitor* page table pointer (just in case page table changes)
- VMM needs to keep track of which pages correspond to which translated blocks.
- When a page holding translated code is removed, flush corresponding code cache entries
  - Requires side tables that map pages to code cache entries
  - As well as corresponding DRAS and JTLB entries
- After flushing code must be re-interpreted
  - And page fault will be detected at that time

Lazy Page Fault Detection

- when a source code page is replaced by the guest OS, the code cache is not immediately flushed of corresponding translations. It waits until there is an attempt to actually use the translated code.
- Every time the translated code crosses a source page boundary, the page table is probed to see if the mapping is the same as on the original translation
- Probe page table when control crosses page boundary
  - Use special **`Verify_Translation`** instruction (inserted by translator)
    - Arguments: source virtual address
    - real address at time code was translated
  - This instruction probes page table
  - Jump to VMM if there has been a change (or page fault)

Page Crossings

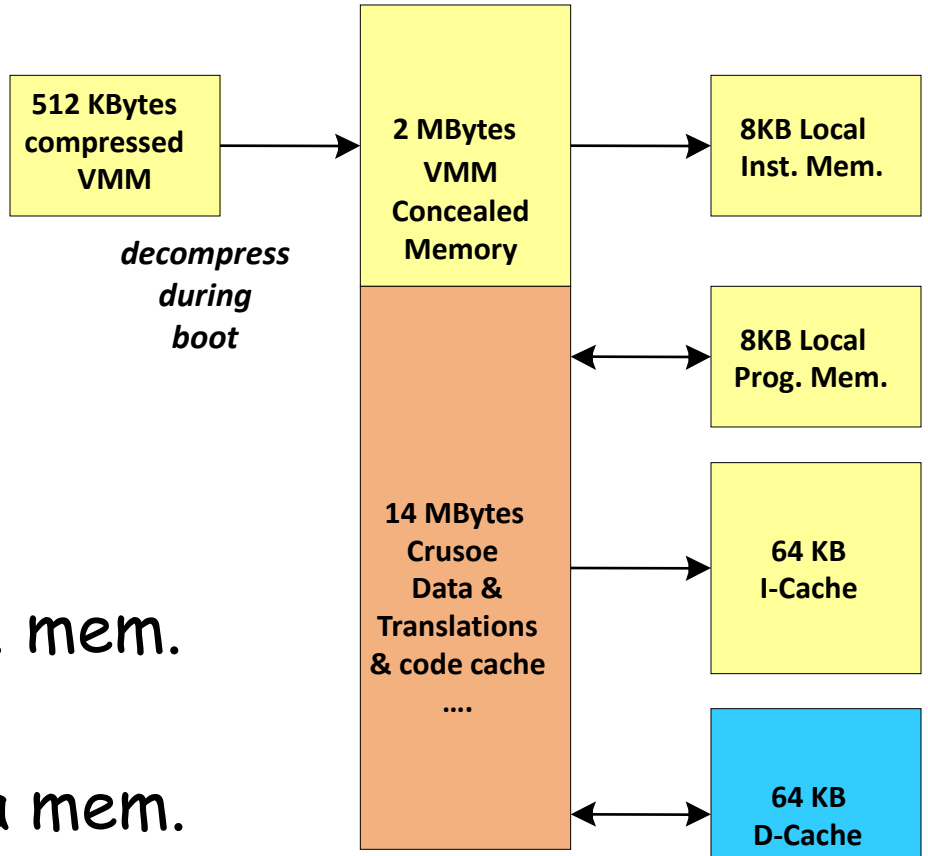


Input/Output

- VMM itself uses no I/O
- Run guest I/O drivers as-is
  - Let I/O drivers directly control I/O signals
- Problems w/ Memory-Mapped I/O
  - Use access-protect in TLB to detect accesses to volatile pages
  - De-optimize code that accesses volatile pages
  - Enhance ISA w/ load/store opcodes that over-ride access-protect

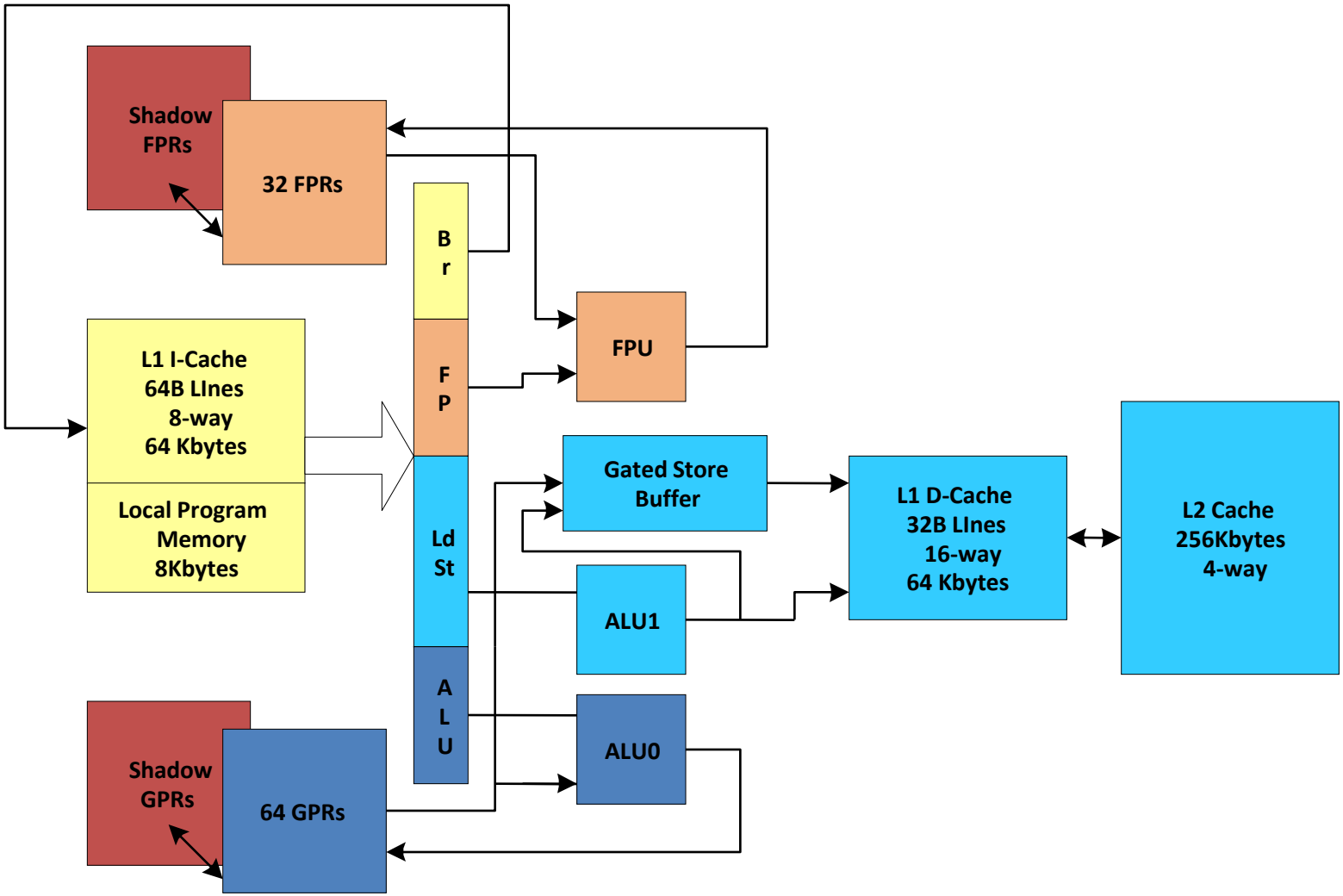
Case Study: Transmeta Crusoe

- Main goals:
  - power efficiency
  - design simplicity
- IA-32 to VLIW (4-way)
  - Specialized Fields
    - (ALU, LD/ST, FP, Br)
- 16M Translation Cache
- 8K bytes VMM local inst. mem.
  - Reduces I cache pollution
- 8K bytes VMM local data mem.
  - Reduces D cache pollution



Memory Hierarchy

Transmeta Crusoe Block Diagram



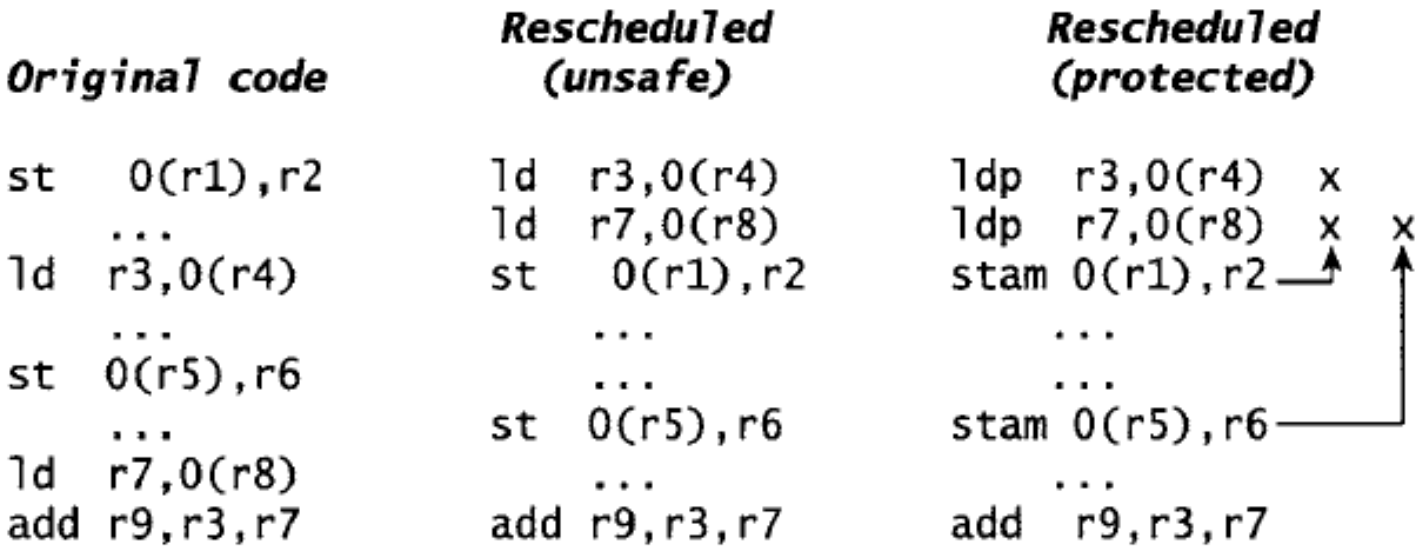
Translation

- Staged optimization
  - Interpretation (with profiling)
  - Simple translation
  - Highly optimized translation
- Algorithm translates "multiple basic blocks"
- Assumes in-order VLIW processor → code reordering is of crucial importance
  - load operations have high latency
  - Reordering them way earlier than the instructions that use them is beneficial

A codesigned approach for load reordering

- To allow re-scheduling of memory ops
- *load-and-protect*
  - Special load opcode
  - records load address and loaded data size in table
- *store-under-alias-mask*
  - Special store opcode
  - Checks specified (via mask) loads in table
  - if conflict, triggers re-do of loads

Alias Hardware Example

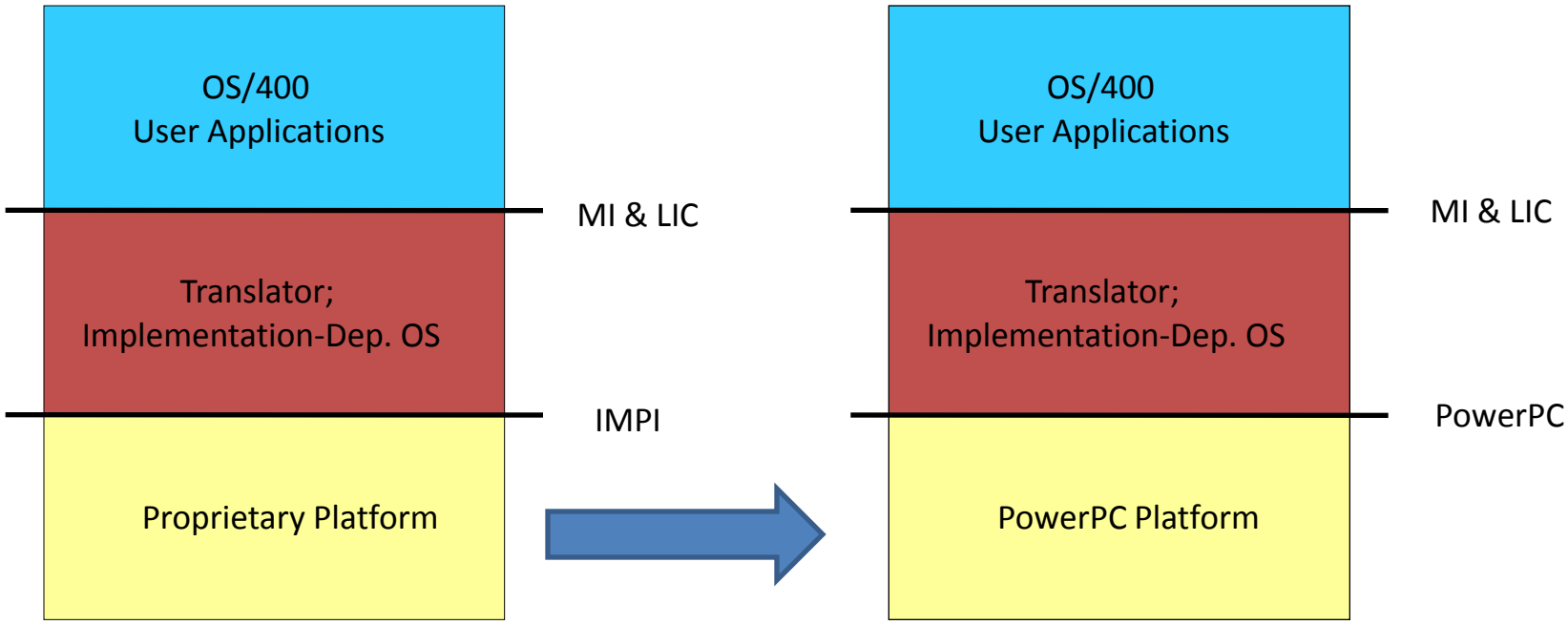


Case Study: IBM AS/400

- A very early (and successful) co-designed VM
- Goals
  - Hardware independence
    - Demonstrated by move to PowerPC
  - Support robust/well integrated software
    - Re-define conventional software boundaries
    - Architect object-orientation

IBM AS/400 Platforms

- System /38 – Proprietary implementation ISA
- AS/400 – First, extend proprietary ISA
  Then transition to PowerPC ISA

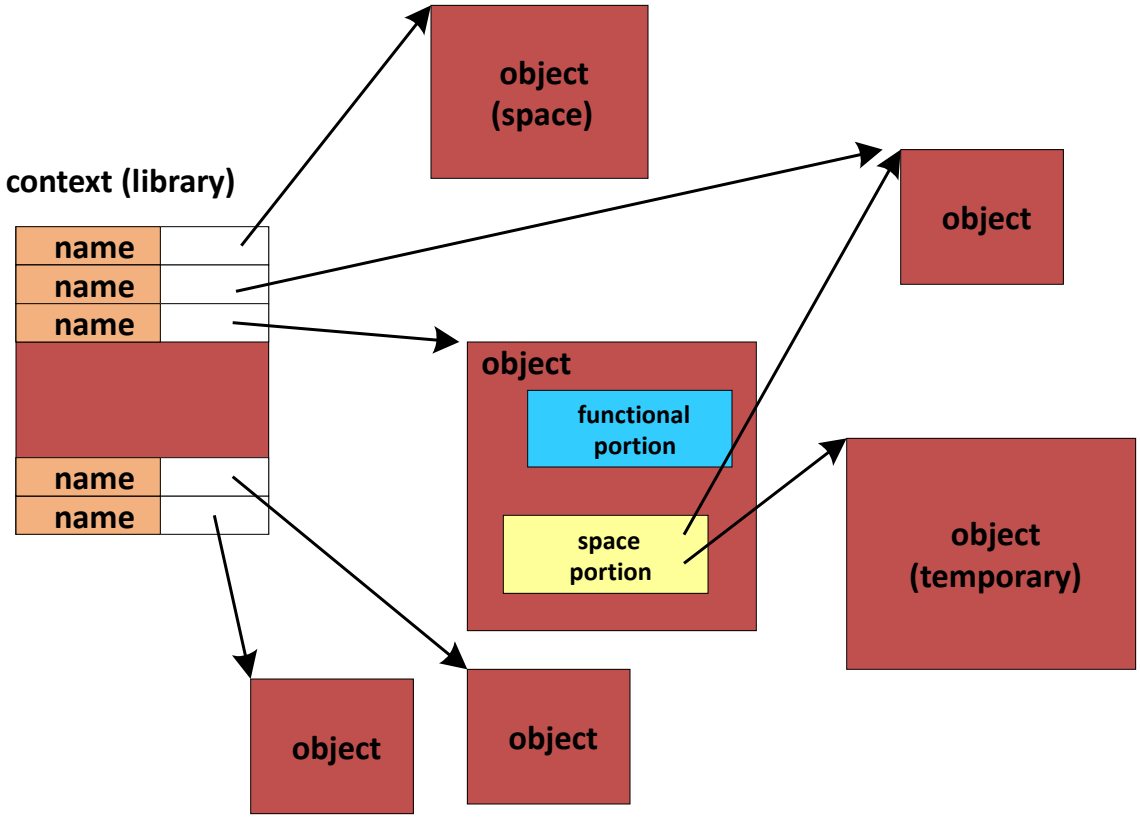


MI: Machine Interface

LIC: Licensed Internal Code

Memory Architecture

- Architected Objects
  - Space objects: data only
  - Other objects – also have "functional part"
  - Can be accessed through pointers only
  - Pointers cannot be modified by ordinary instructions



Object Oriented ISA

- All objects accessed by protected, implementation dependent pointers
- Objects can persist over system lifetime
  - Unlike process VMs like Java
    - Objects go away when program terminates
  - Objects can persist literally for years
    - There are no conventional files; only objects
    - Large, flat memory space (128 bit pointers)
- Context object points to accessible objects
- No garbage collection
  - Temporary objects go away at re-boot

V-ISA Example

- Addn & Branch

  sum = addend1 + addend2

  if sum == 0 goto destination1

  else if sum > 0 goto destination2

- 19 bytes total

  But this is only a *representation* of what has to be done

  Translation reduces it to PowerPC equivalent instructions

Conclusions

- Co-designed VMs enable hardware change that was prohibited by backward compatibility
- Main goals: power efficiency, performance, and/or design simplicity.