BedRock Systems Inc

Modularizing CPU Semantics for Virtualization

Paolo Giarrusso, Abhishek Anand, **Gregory Malecha**, Frantisek Farka, Hoang-Hai Dang

BedRock Systems Inc.

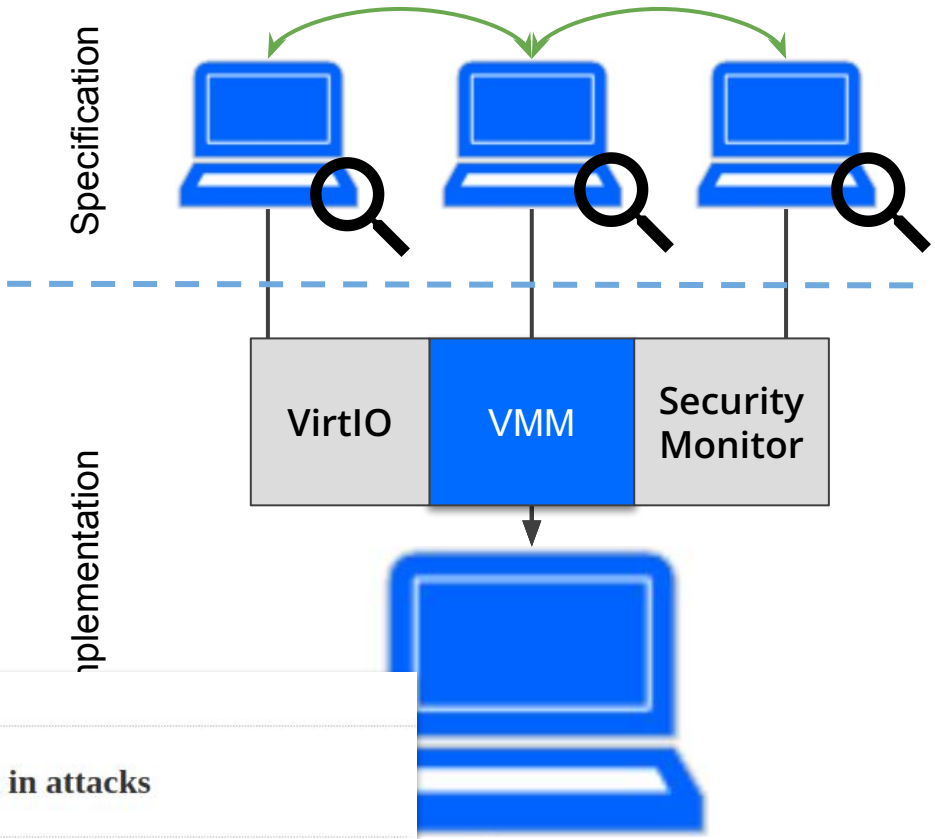
Virtualization

Virtualization provides:

- Consolidation
- Runtime services
- Monitoring & remediation
- Management services

Highly privileged code software running arbitrary/malicious code.

- Vulnerabilities are common & severe.



Home > News > Security > VMware confirms critical vCenter flaw now exploited in attacks

VMware confirms critical vCenter flaw now exploited in attacks

By Sergiu Gatlan — January 19, 2024 — 08:22 AM — 0

Virtualization

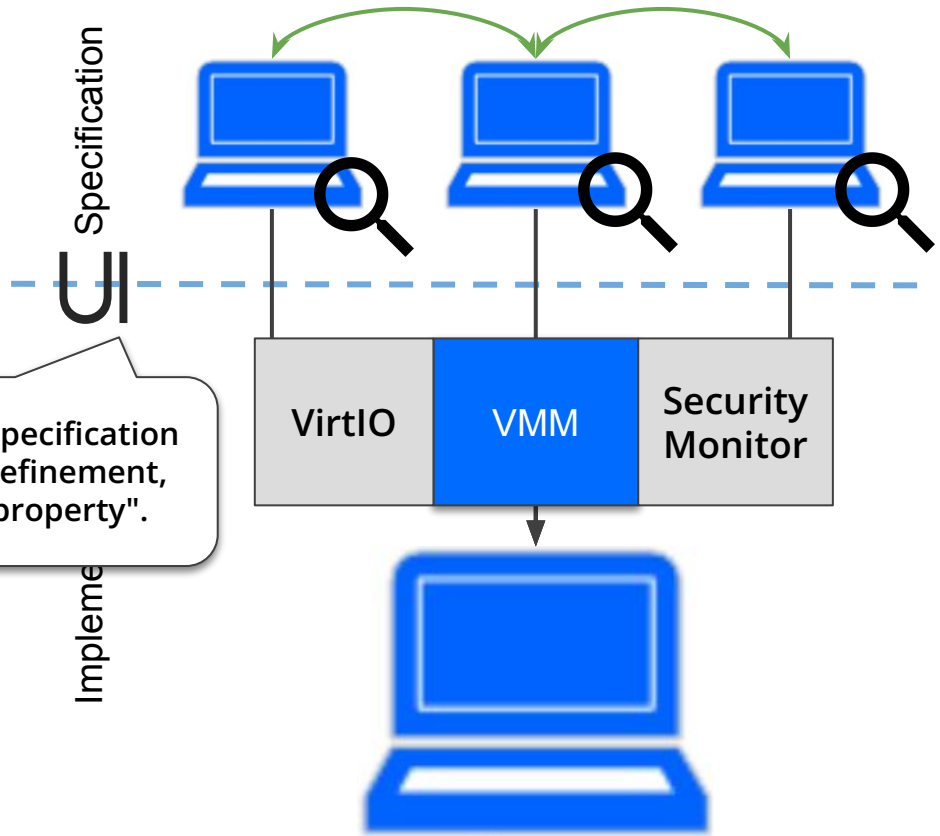
Virtualization provides:

- Consolidation
- Runtime services
- Monitoring & remediation
- Management services

Highly privileged code softwar
running arbitrary/malicious c

- Vulnerabilities are common & severe.

Goal Prove a VMM correct.



Virtualization

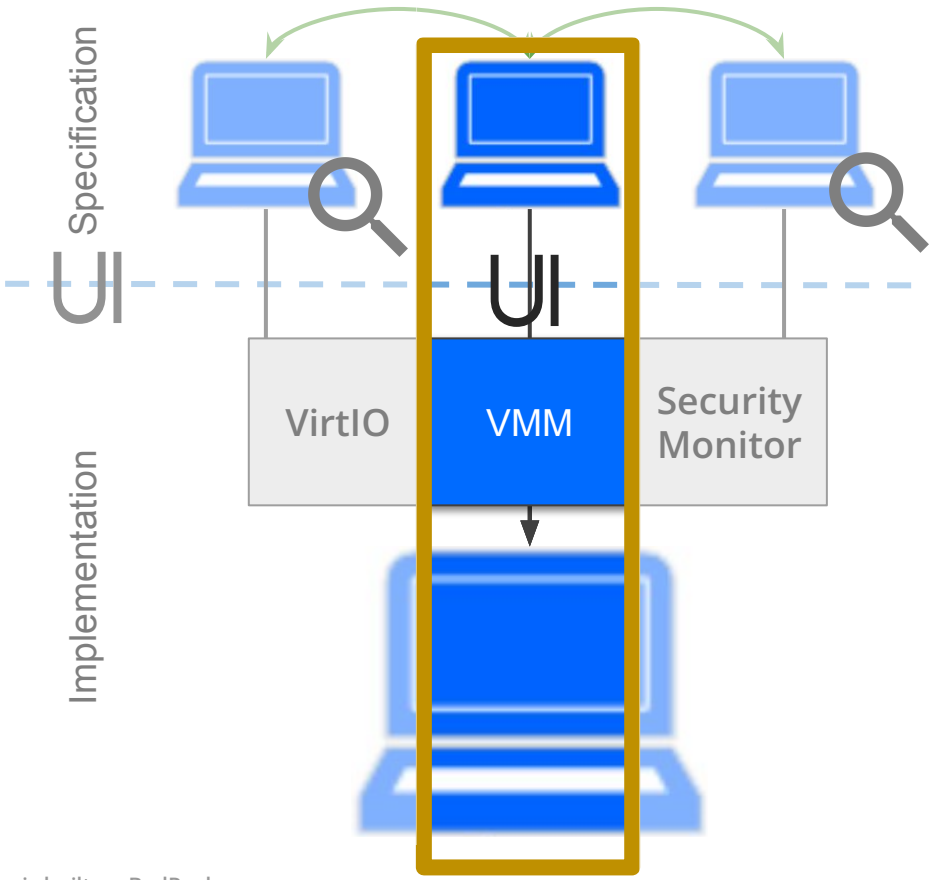
Virtualization provides:

- Consolidation
- Runtime services
- Monitoring & remediation
- Management services

Highly privileged code software running arbitrary/malicious code.

- Vulnerabilities are common & severe.

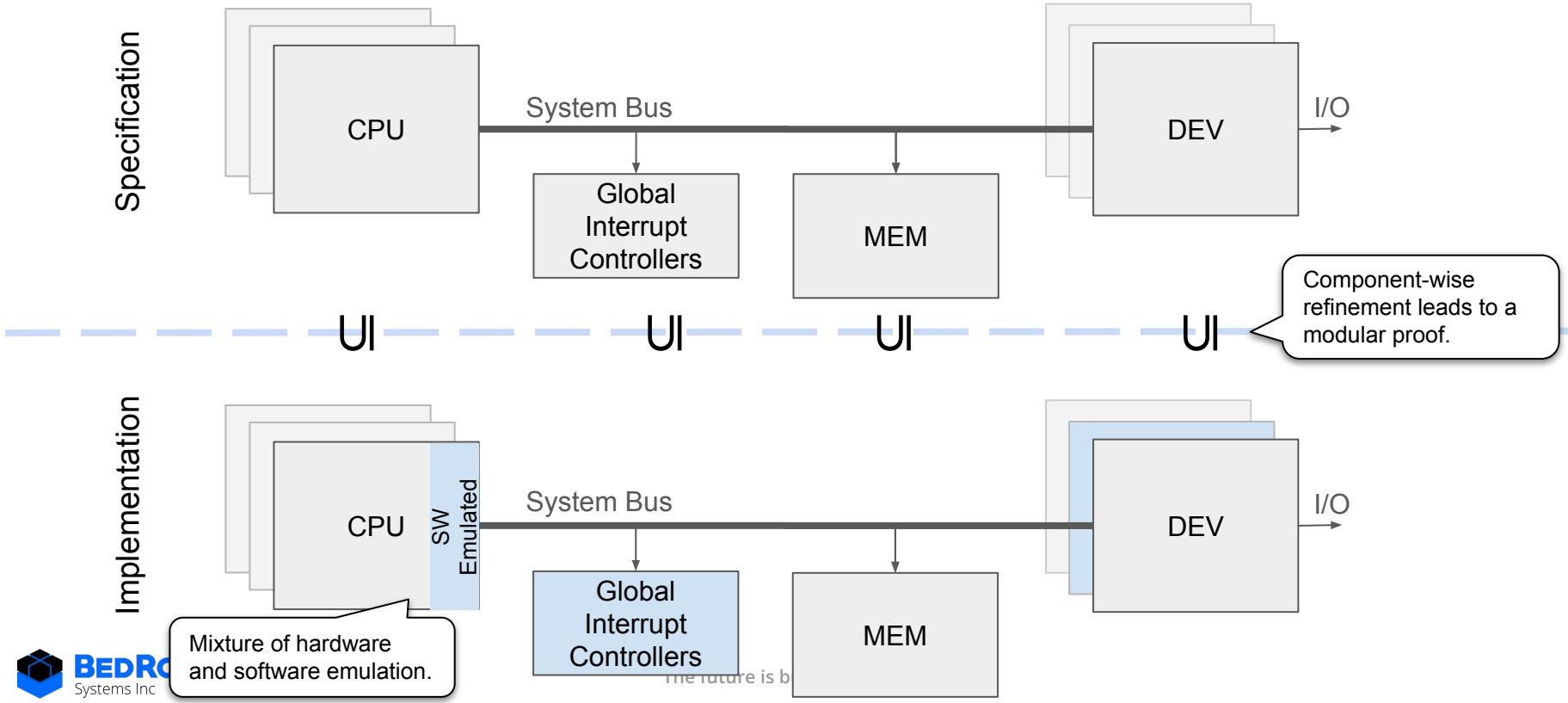
Goal Prove a VMM correct.



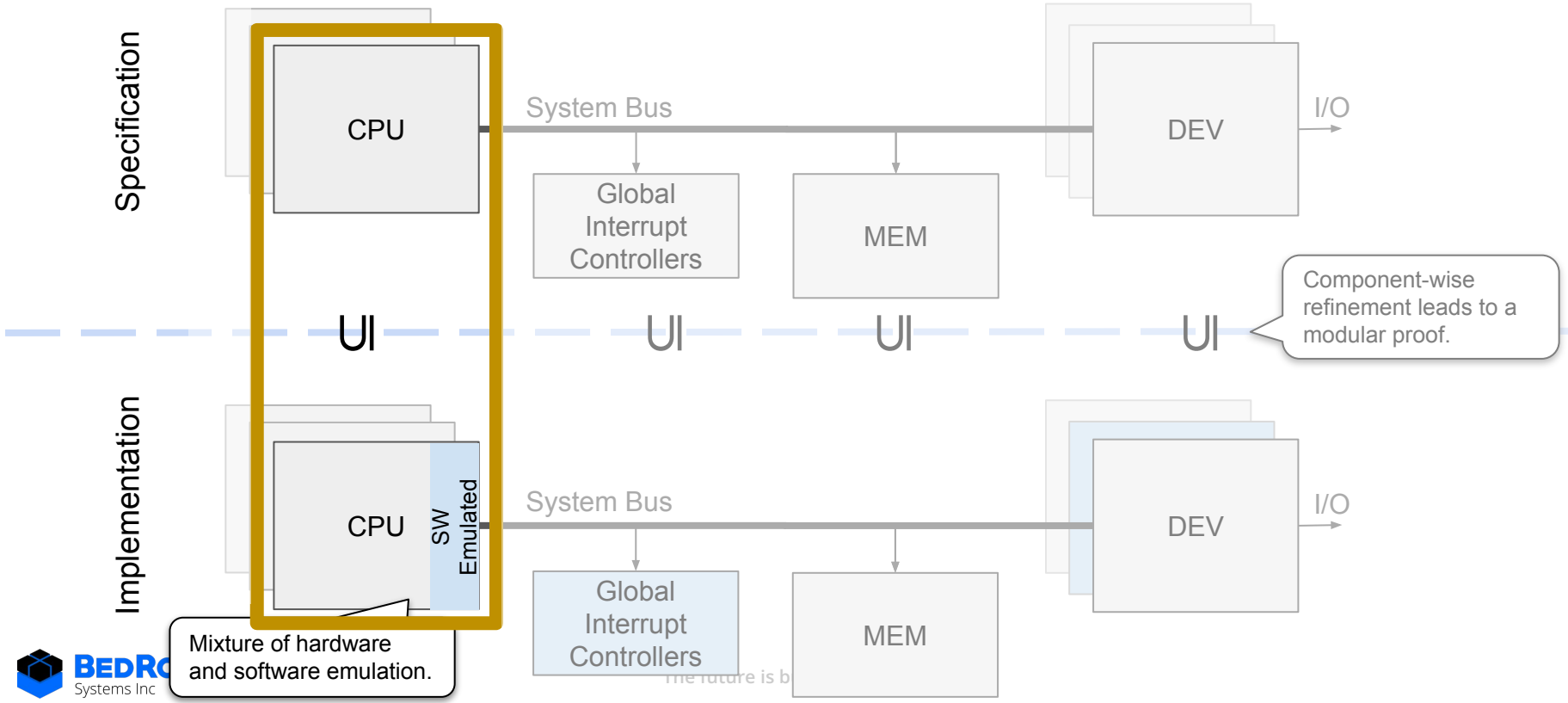
Outline

1. **Hardware**
2. The CPU Refinement
3. Decomposing CPU Semantics
4. Using the Decomposition

A Modular VMM Proof



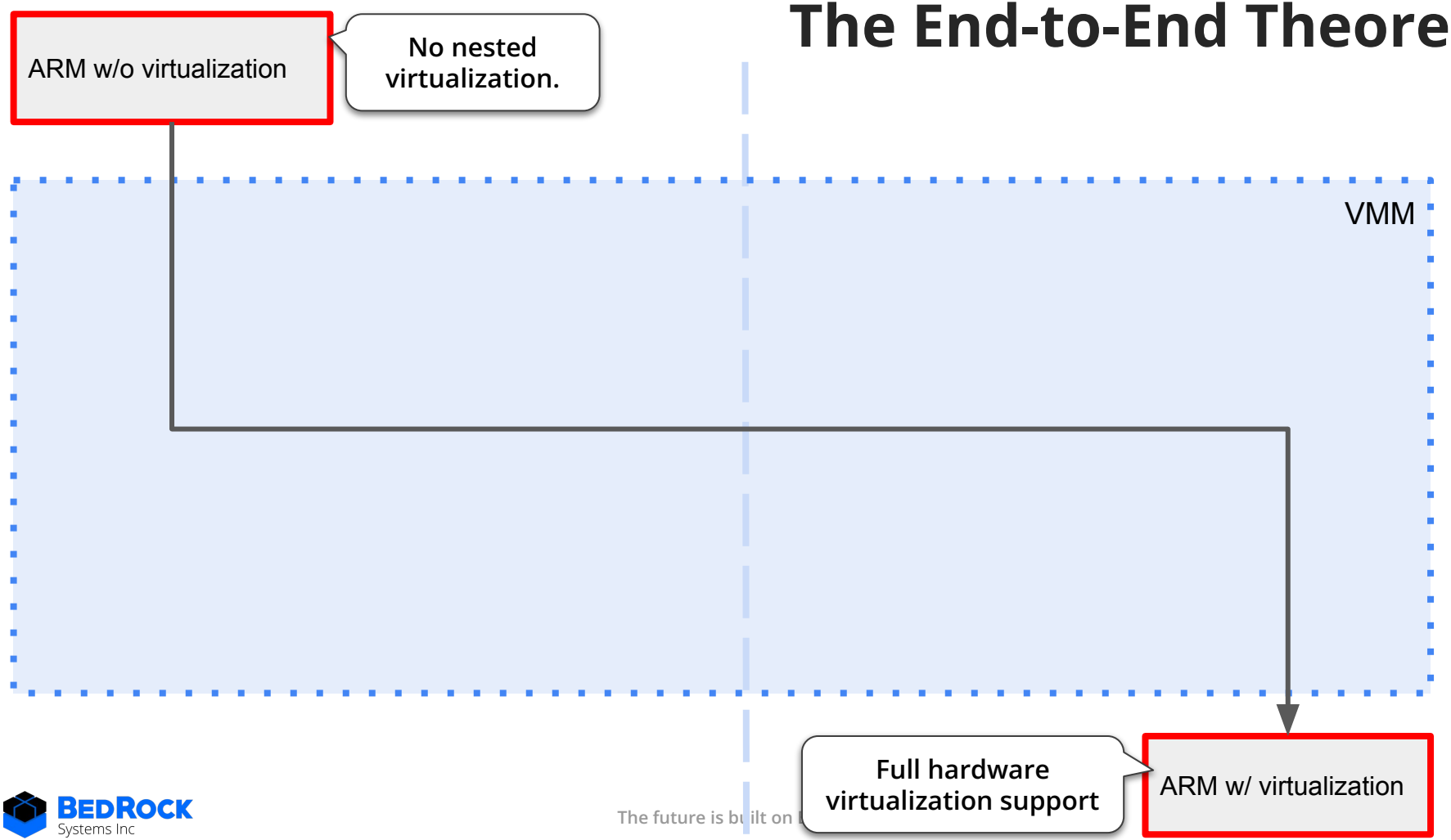
A Modular VMM Proof



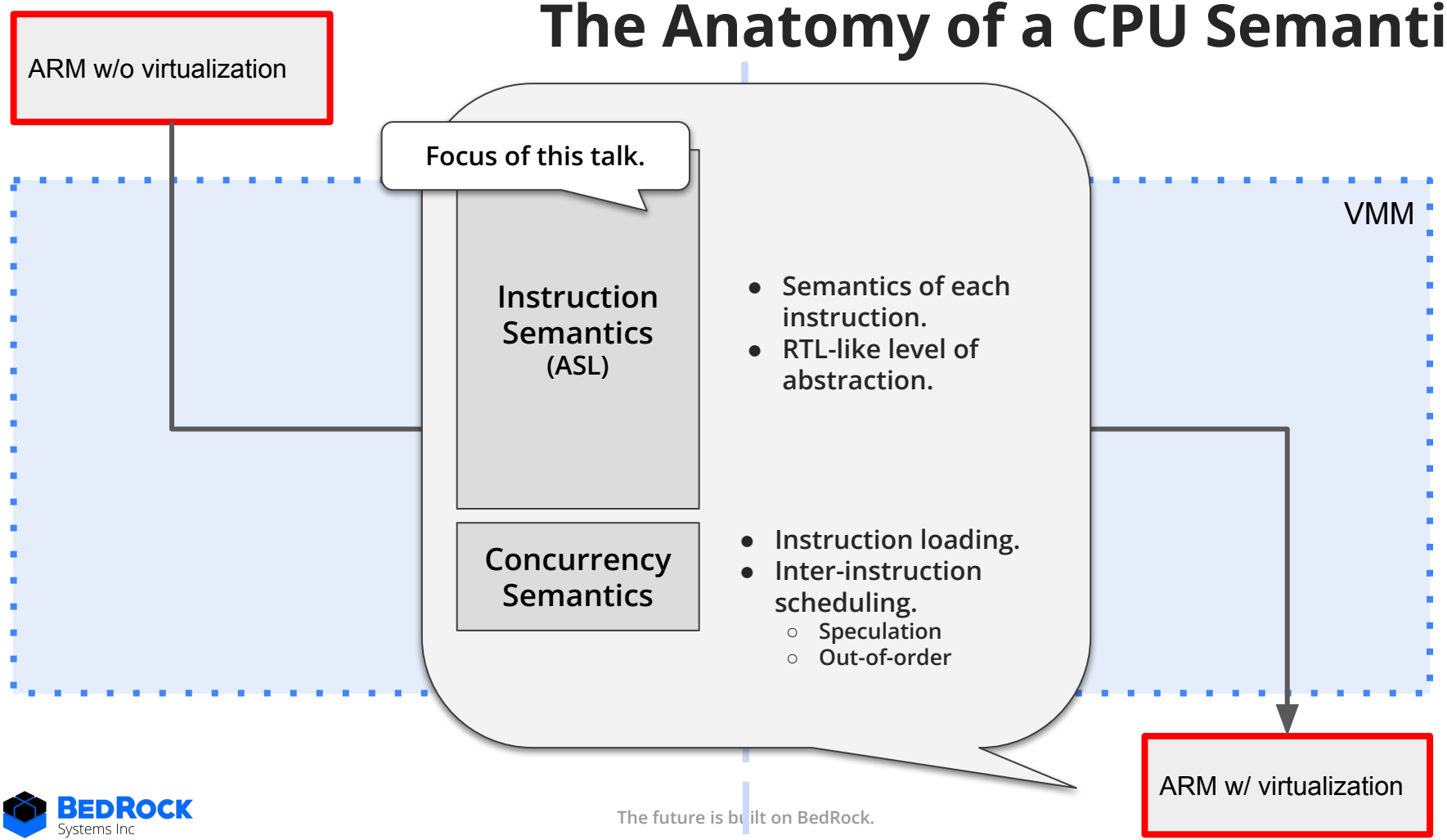
Outline

1. Hardware
2. **The CPU Refinement**
3. Decomposing CPU Semantics
4. Using the Decomposition

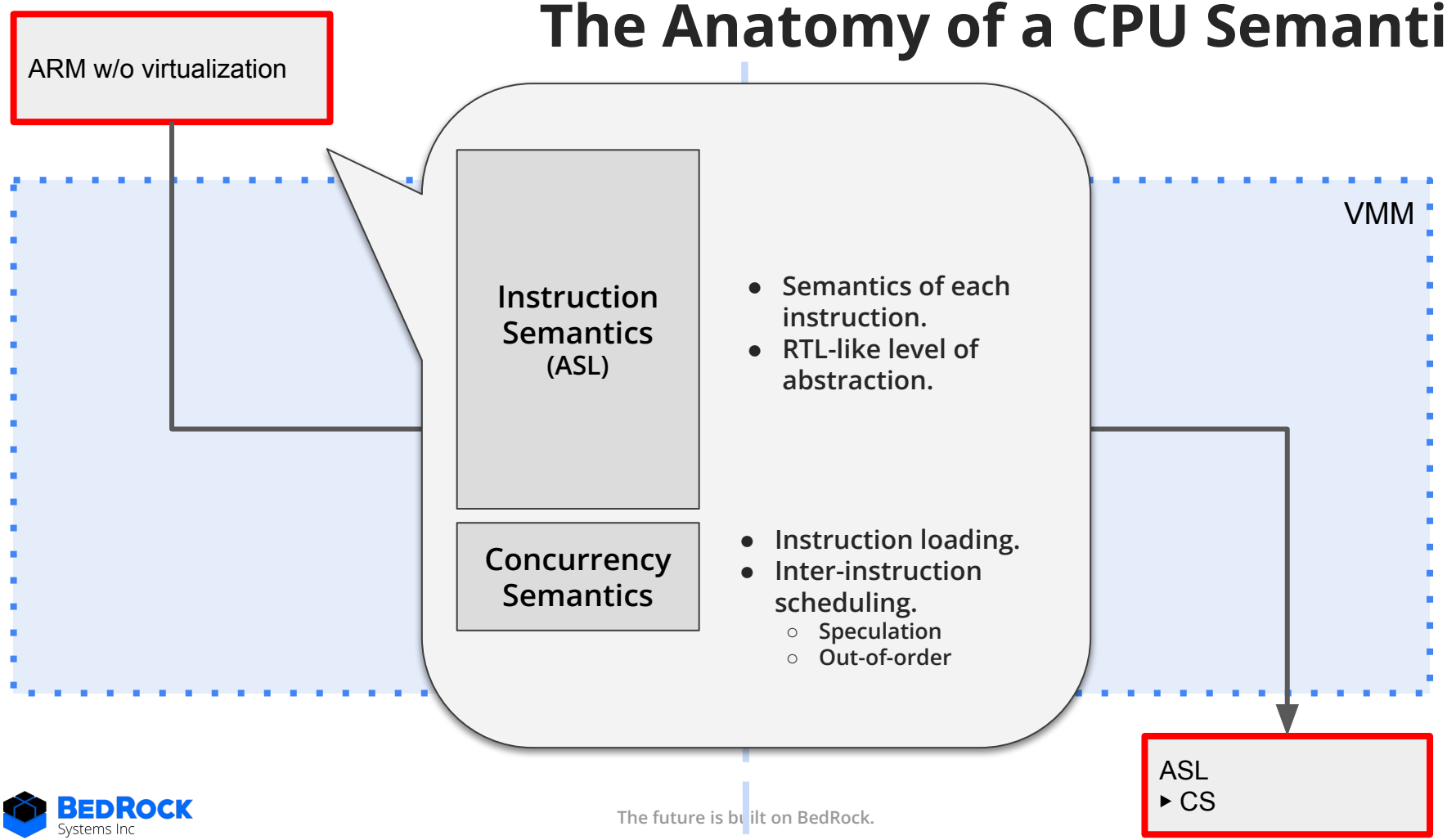
The End-to-End Theorem



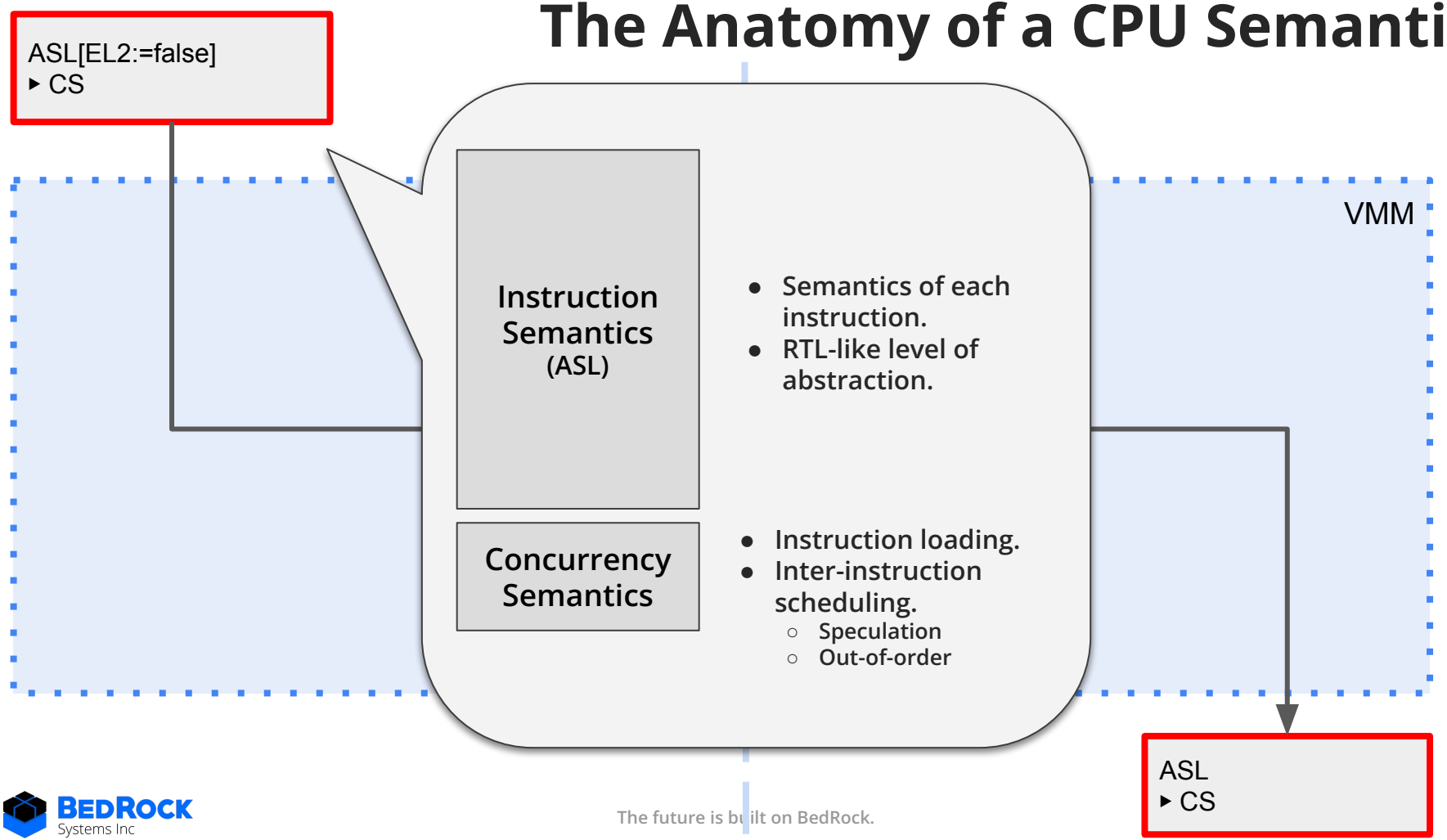
The Anatomy of a CPU Semantics



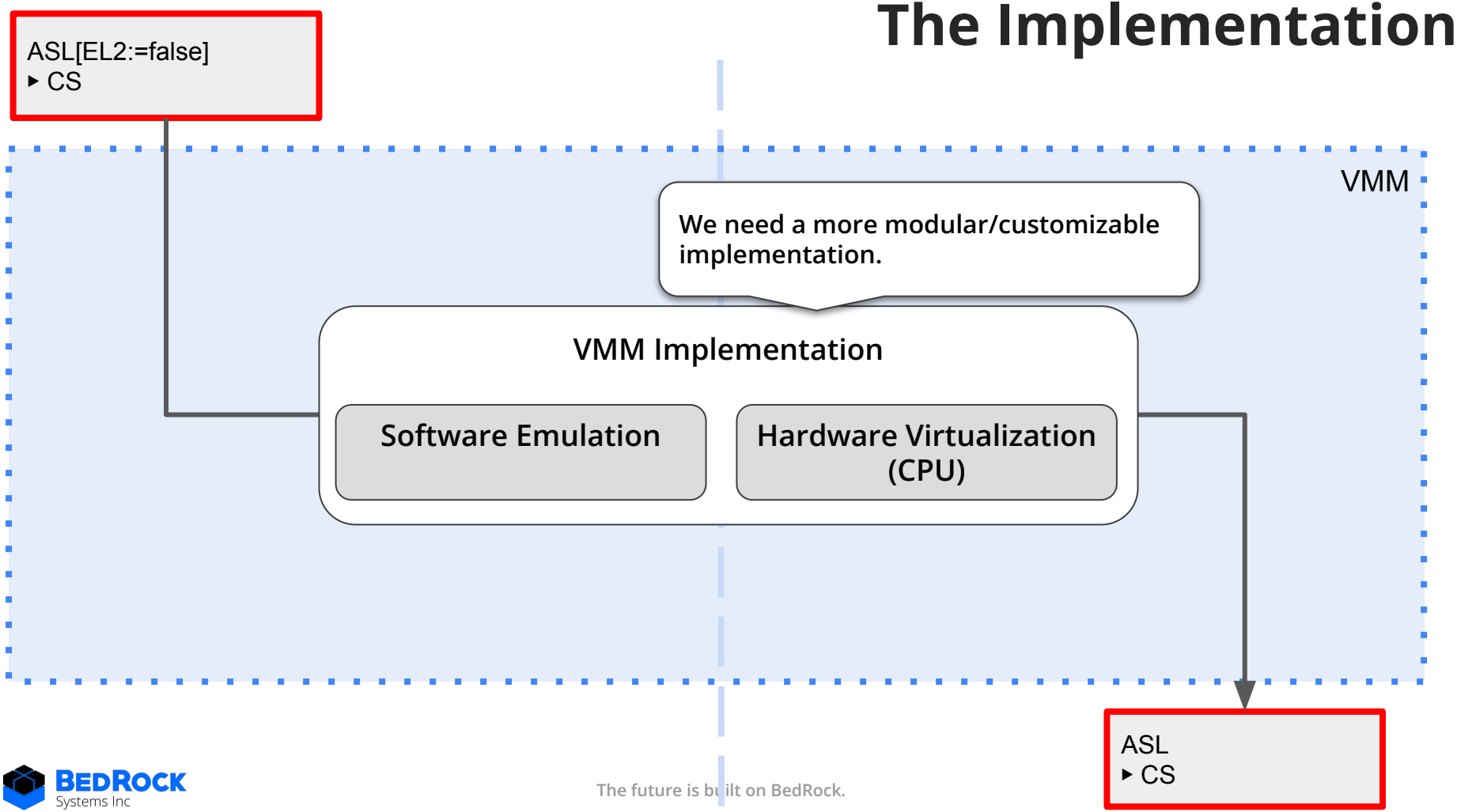
The Anatomy of a CPU Semantics



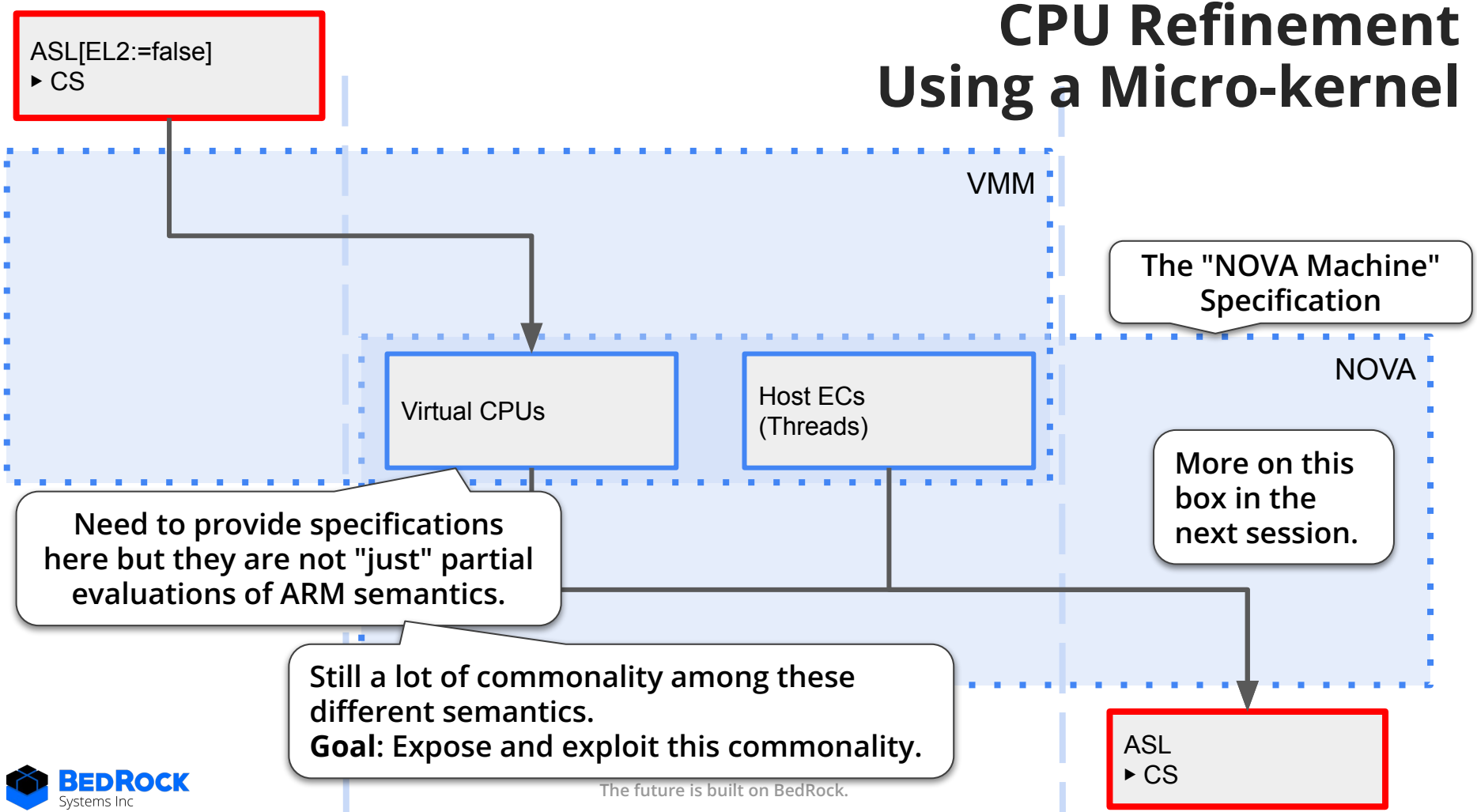
The Anatomy of a CPU Semantics



The Implementation

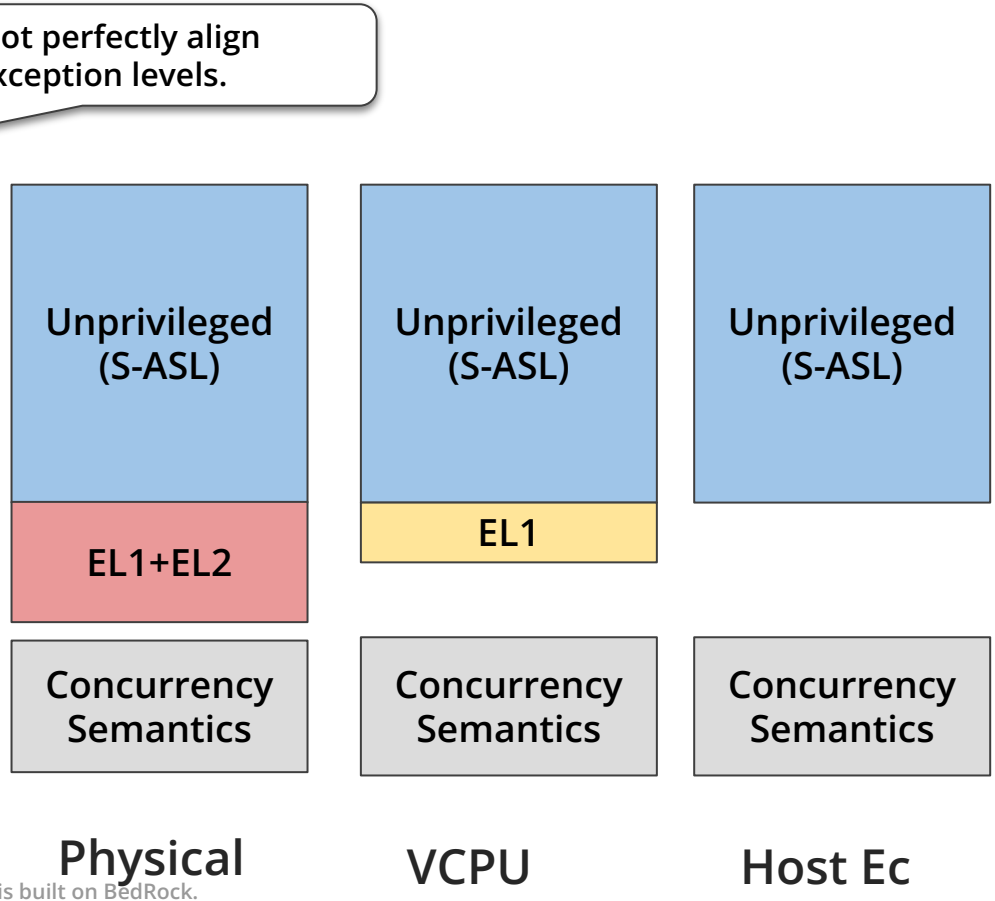


CPU Refinement Using a Micro-kernel

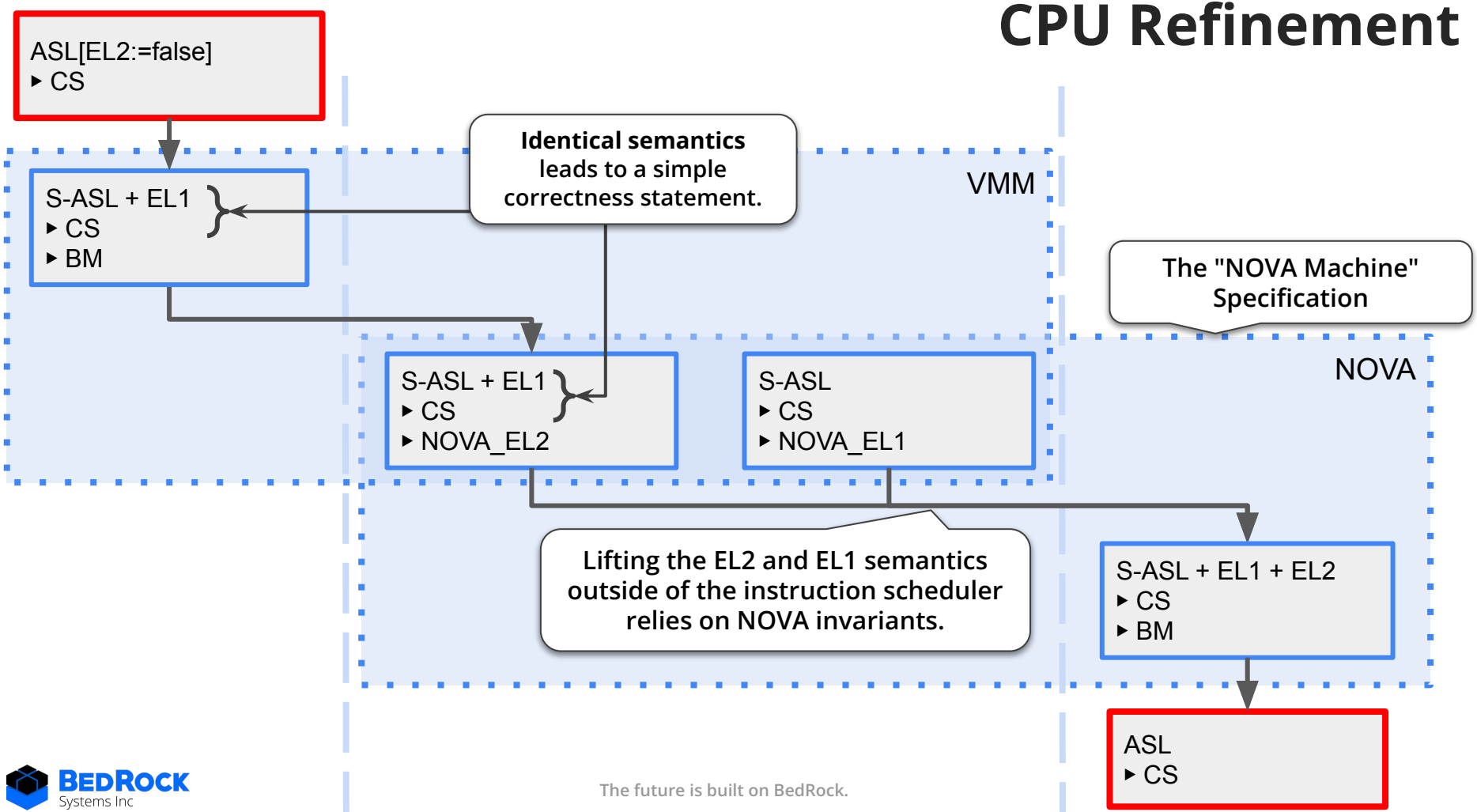


Refactoring CPU Semantics

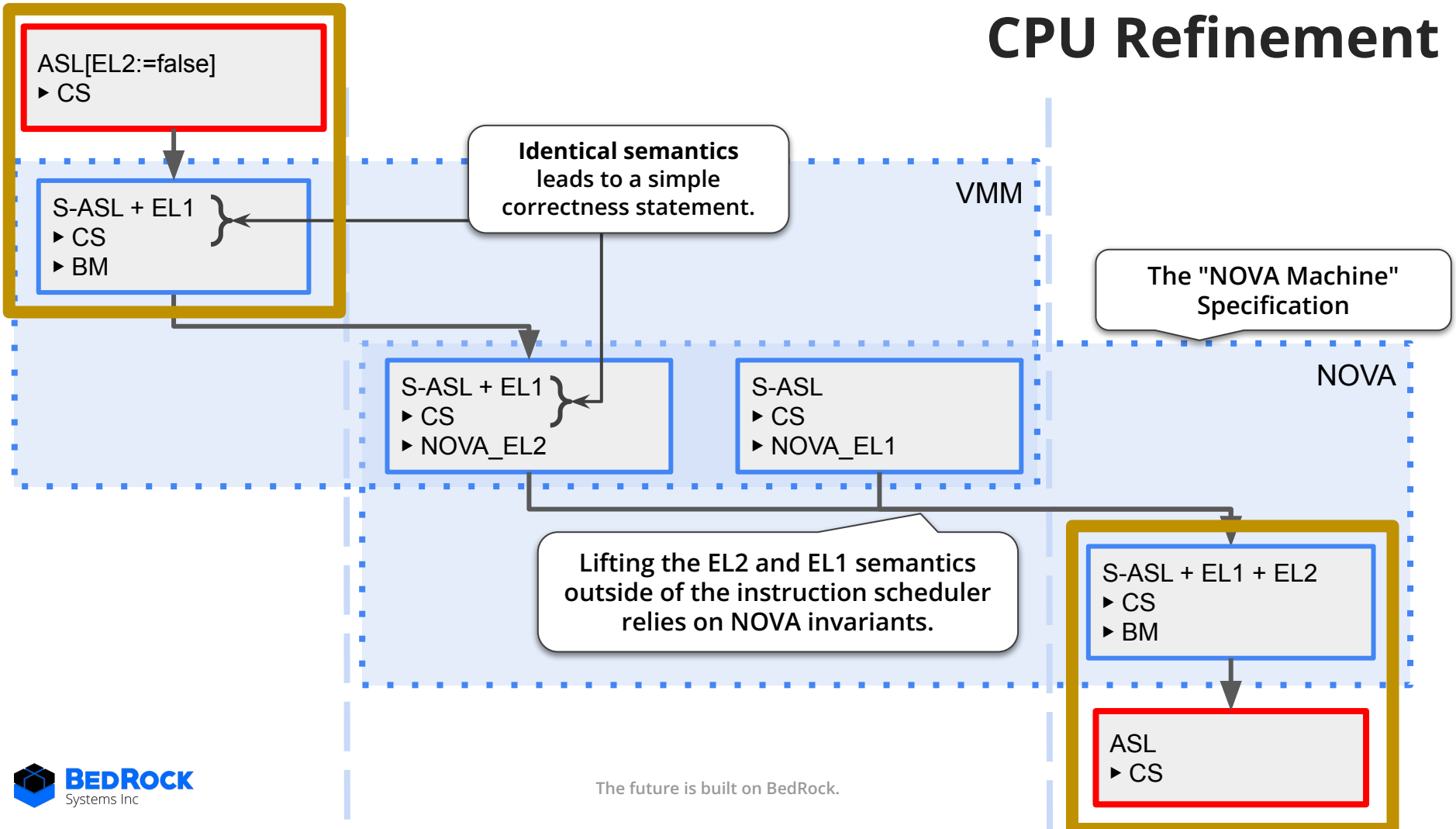
- Manually decompose the model along privilege boundaries.
  - Enables code reuse across instances.
- Exposes the abstractions in the semantics.
- Encapsulation allows precise reasoning about a single component while ignoring other components.
- Virtualization "works" parametrically in the unprivileged logic.



CPU Refinement



CPU Refinement



Outline

1. Hardware
2. The CPU Refinement
3. **Decomposing CPU Semantics**
4. Using the Decomposition

Simplified Example
MRS <Xt>,TTBR0_EL1

- Identify accesses to privileged state.
- Refactor privileged accesses to invoke security handlers.

```
// ASL (from ARM manual)
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TRVM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled()
-    && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')
     && HFGRTR_EL2.TTBR0_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
-  elsif EL2Enabled()
-     && HCR_EL2.<NV2,NV1,NV> == '111' then
-    X[t, 64] = NVMem[0x200];
  else
    X[t, 64] = TTBR0_EL1;
-elsif PSTATE.EL == EL2 then
- if HCR_EL2.E2H == '1' then
-   X[t, 64] = TTBR0_EL2;
-  else
-    X[t, 64] = TTBR0_EL1;
-elsif PSTATE.EL == EL3 then
-  X[t, 64] = TTBR0_EL1;
```

Simplified Example
MRS <Xt>,TTBR0_EL1

- Identify accesses to privileged state.
  - **TTBR0_EL1**
- Refactor privileged accesses to invoke security handlers.

```
// ASL (from ARM manual)
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TRVM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled()
-    && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')
     && HFGRTR_EL2.TTBR0_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
-  elsif EL2Enabled()
-     && HCR_EL2.<NV2,NV1,NV> == '111' then
-    X[t, 64] = NVMem[0x200];
  else
    X[t, 64] = TTBR0_EL1;
-elsif PSTATE.EL == EL2 then
- if HCR_EL2.E2H == '1' then
-   X[t, 64] = TTBR0_EL2;
-  else
-    X[t, 64] = TTBR0_EL1;
-elsif PSTATE.EL == EL3 then
-  X[t, 64] = TTBR0_EL1;
```

Access control checks use EL2 state. To enforce these, we must pull TTBR0_EL1 into the privileged state.

Simplified Example
MRS <Xt>,TTBR0_EL1

- Identify accesses to privileged state.
  - **TTBR0_EL1**
- Refactor privileged accesses to invoke security handlers.

```
// ASL (from ARM manual)
if PSTATE.EL == EL0 then
  UNDEFINED;
elsif PSTATE.EL == EL1 then
  if EL2Enabled() && HCR_EL2.TRVM == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
  elsif EL2Enabled()
-    && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')
     && HFGRTR_EL2.TTBR0_EL1 == '1' then
    AArch64.SystemAccessTrap(EL2, 0x18);
-  elsif EL2Enabled()
-     && HCR_EL2.<NV2,NV1,NV> == '111' then
-    X[t, 64] = NVMem[0x200];
  else
    X[t, 64] = TTBR0_EL1;
-elsif PSTATE.EL == EL2 then
- if HCR_EL2.E2H == '1' then
-   X[t, 64] = TTBR0_EL2;
-  else
-    X[t, 64] = TTBR0_EL1;
-elsif PSTATE.EL == EL3 then
-  X[t, 64] = TTBR0_EL1;
```

```
// Unprivileged
if PSTATE.EL == EL0 then
  UNDEFINED;
else // PSTATE.EL == EL1
  try:
    X[t, 64] = el2.read_msr(TTBR0_EL1, true)
  except:
    AArch64.SystemAccessTrap(EL2, 0x18)
```

```
// EL1 + EL2 handler
el2.read_msr(TTBR0_EL1, direct : bool)
  if direct:
    if HCR_EL2.TRVM == '1' then
      throw
    elsif HFGRTR_EL2.TTBR0_EL1 == '1' then
      throw
    else
      return TTBR0_EL1
  else
    return TTBR_EL1
```

Example
LDUR Xt,<Xn>

- Identify accesses to privileged state.
  - **Memory**
  - **TTBR0_EL1** (not shown)
- Refactor privileged accesses to invoke security handlers.

```
// ASL: LDUR Xt, <Xn>
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
integer n = UInt(Xn);
integer t = UInt(Xt);
integer regsize;
regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
boolean tag_checked = n != 31;
bits(64) address;
bits(datasize) data;
if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n, 64];

address = address + offset;
data = Mem[address, datasize DIV 8, NORMAL];
X[t, regsize] = ZeroExtend(data, regsize);
```

Memory access:
- splits accesses,
- address translation,
- accesses memory

Example
LDUR Xt,<Xn>

- Identify accesses to privileged state.
  - **Memory**
  - **TTBR0_EL1** (not shown)
- Refactor privileged accesses to invoke security handlers.

```
// ASL: LDUR Xt, <Xn>
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
integer n = UInt(Xn);
integer t = UInt(Xt);
integer regsize;
regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
boolean tag_checked = n != 31;
bits(64) address;
bits(datasize) data;
if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n, 64];

address = address + offset;
data = Mem[address, datasize DIV 8, NORMAL];
X[t, regsize] = ZeroExtend(data, regsize);
```

Memory access:
- splits accesses,
- address translation,
- accesses memory

Example
LDUR Xt,<Xn>

- Identify accesses to privileged state.
  - **Memory**
  - **TTBR0_EL1** (not shown)
- Refactor privileged accesses to invoke security handlers.

```
// ASL: LDUR Xt, <Xn>
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
integer n = UInt(Xn);
integer t = UInt(Xt);
integer regsize;
regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
boolean tag_checked = n != 31;
bits(64) address;
bits(datasize) data;
if HaveMTE2Ext() then
  SetTagCheckedInstruction(tag_checked);
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n, 64];

address = address + offset;
data = Mem[address, datasize DIV 8, NORMAL];
X[t, regsize] = ZeroExtend(data, regsize);
```

Memory access:
- splits accesses,
- address translation,
- accesses memory

Privileged interface assumes unprivileged logic decomposes unaligned accesses.

```
try:
  // splitting for unaligned accesses
  [handle, perms] = el1.addr.translate(addr, datasize);
  data = el1.mem.read(handle, addr, datasize, NORMAL);
except:
  AArch64.SystemAccessTrap(EL
```

Handles abstract physical addresses. *S-ASL never sees physical addresses.*

BEDROCK Systems Inc

The S-ASL Memory Interface

Separate address translation and access is necessary for multi-access instructions, e.g. STP.

`[handle, perms] = el1.addr.translate(va)`

- Perform address translation by consulting the page table.
- Returns a handle justifying the translation that can be checked for freshness.

`[val] = el1.mem.read(handle, va, size, acc)`

- Atomically validates the translation and performs permission checks.
- Performs a read on the system bus.

`[] = el1.mem.write(handle, va, size, val, acc)`

- Atomically validates the translation and performs permission checks.
- Performs a write on the system bus.

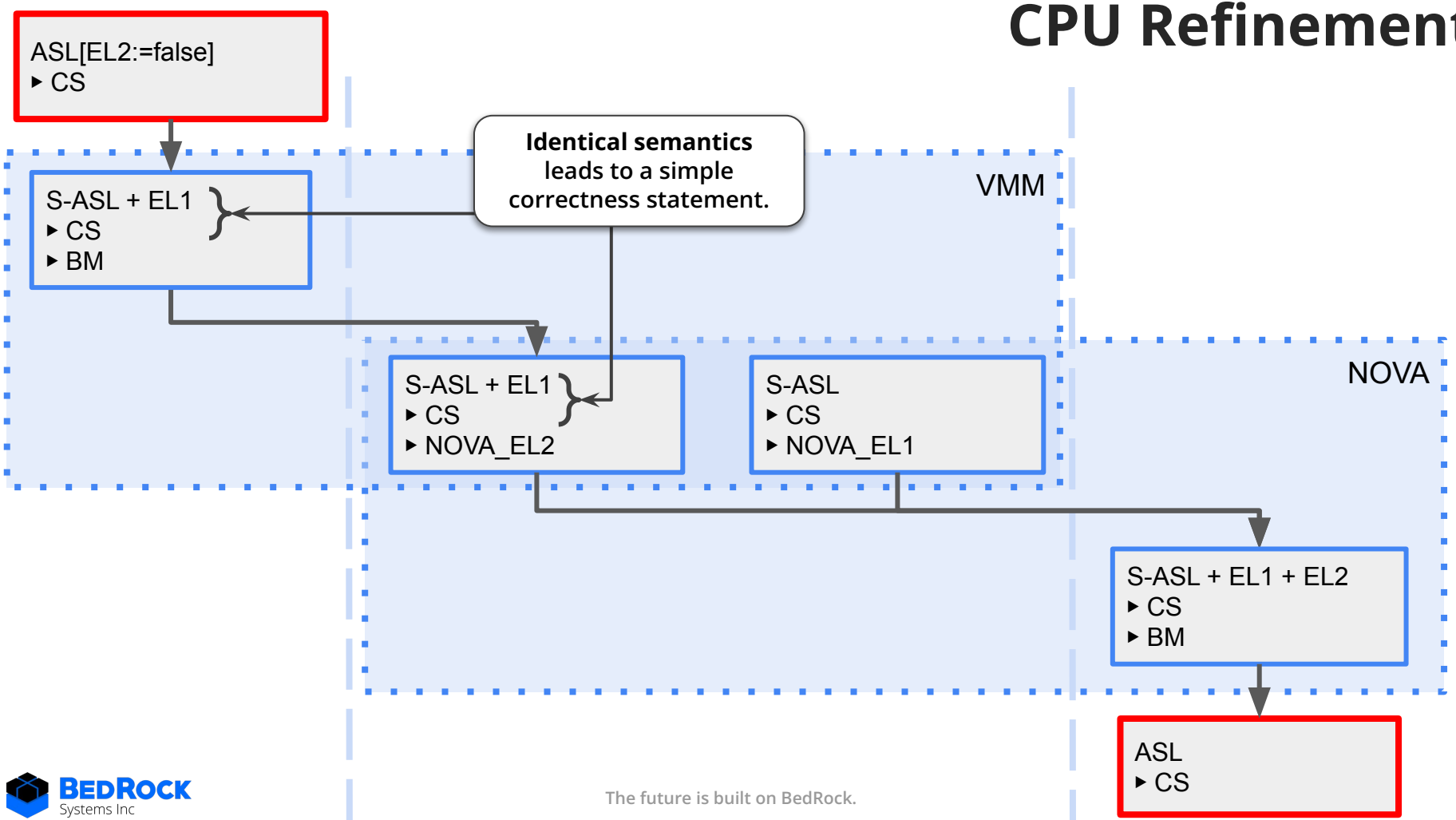
Permission checking occurs atomically with the access. Enables a much simpler model of accessibility.

Atomicity guaranteed by MMU semantics.

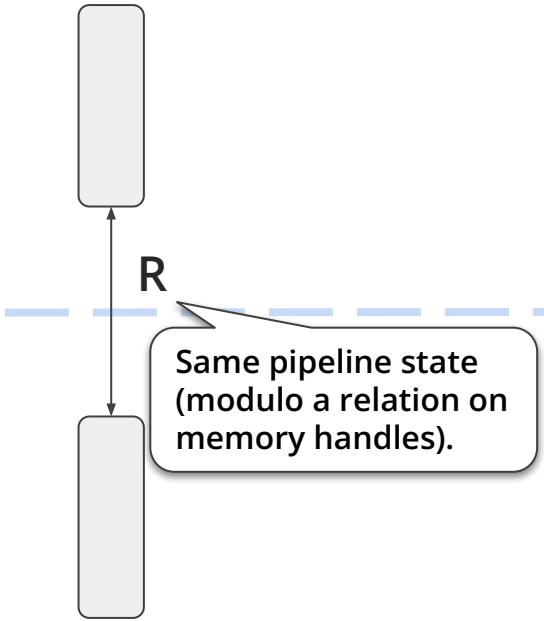
Outline

1. Hardware
2. The CPU Refinement
3. Decomposing CPU Semantics
4. **Using the Decomposition**

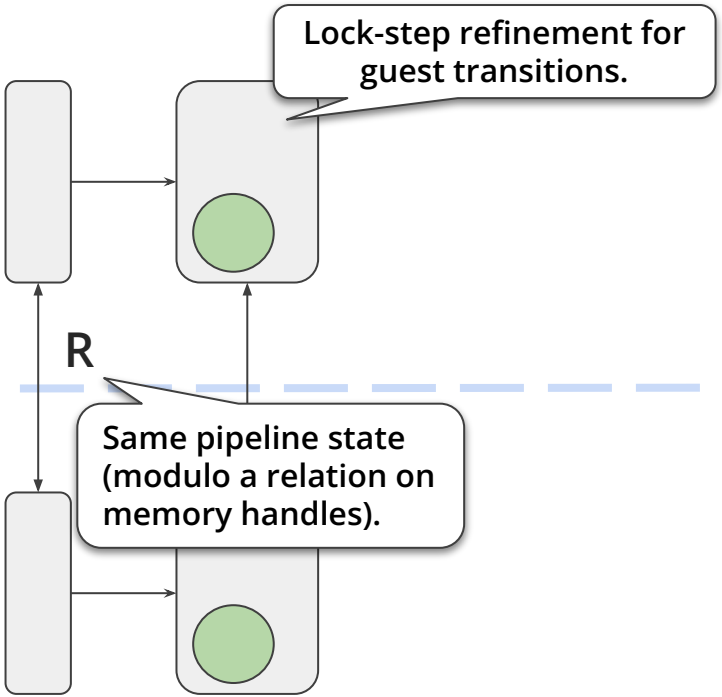
CPU Refinement



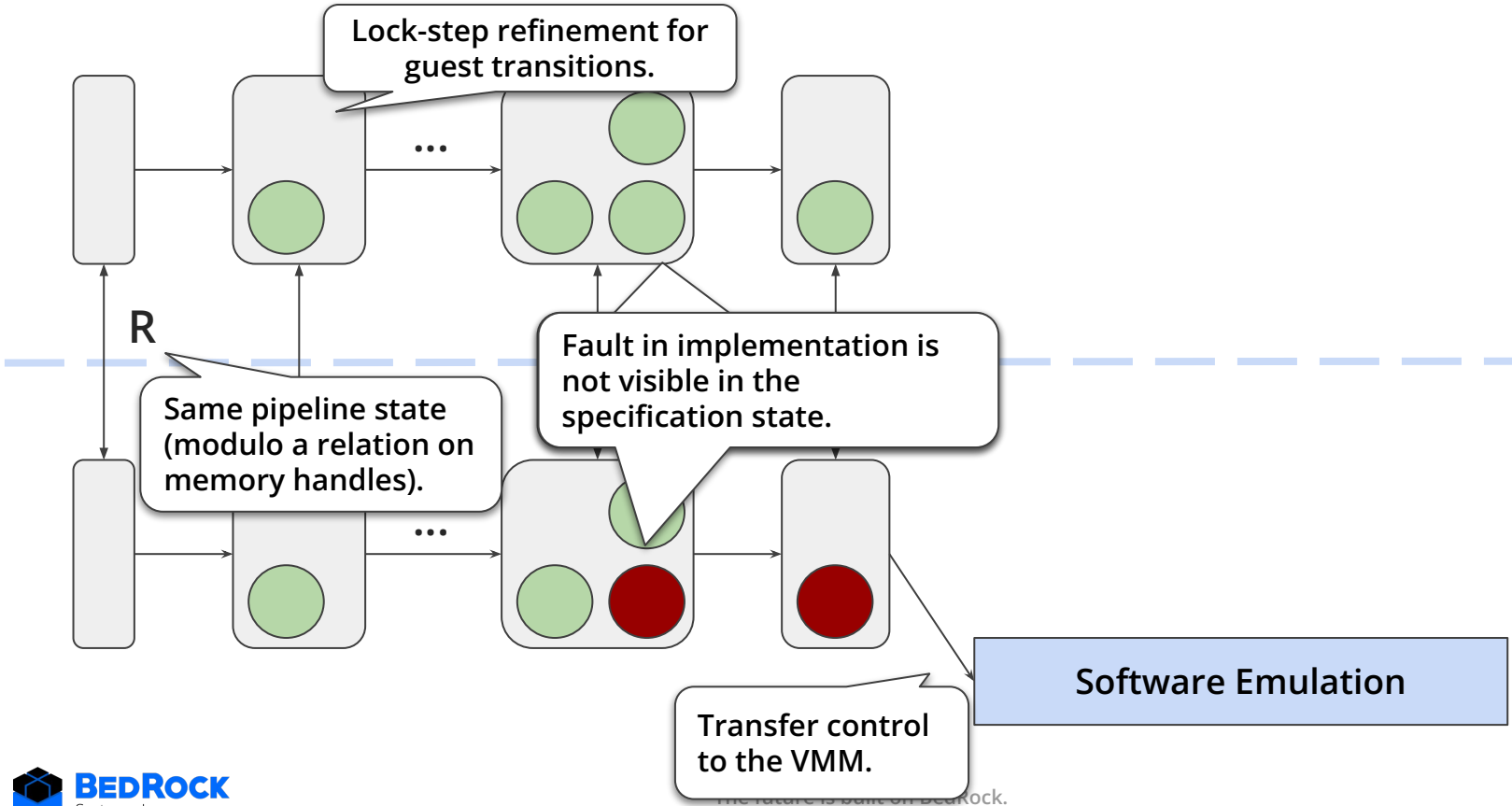
Hardware Virtualization Works



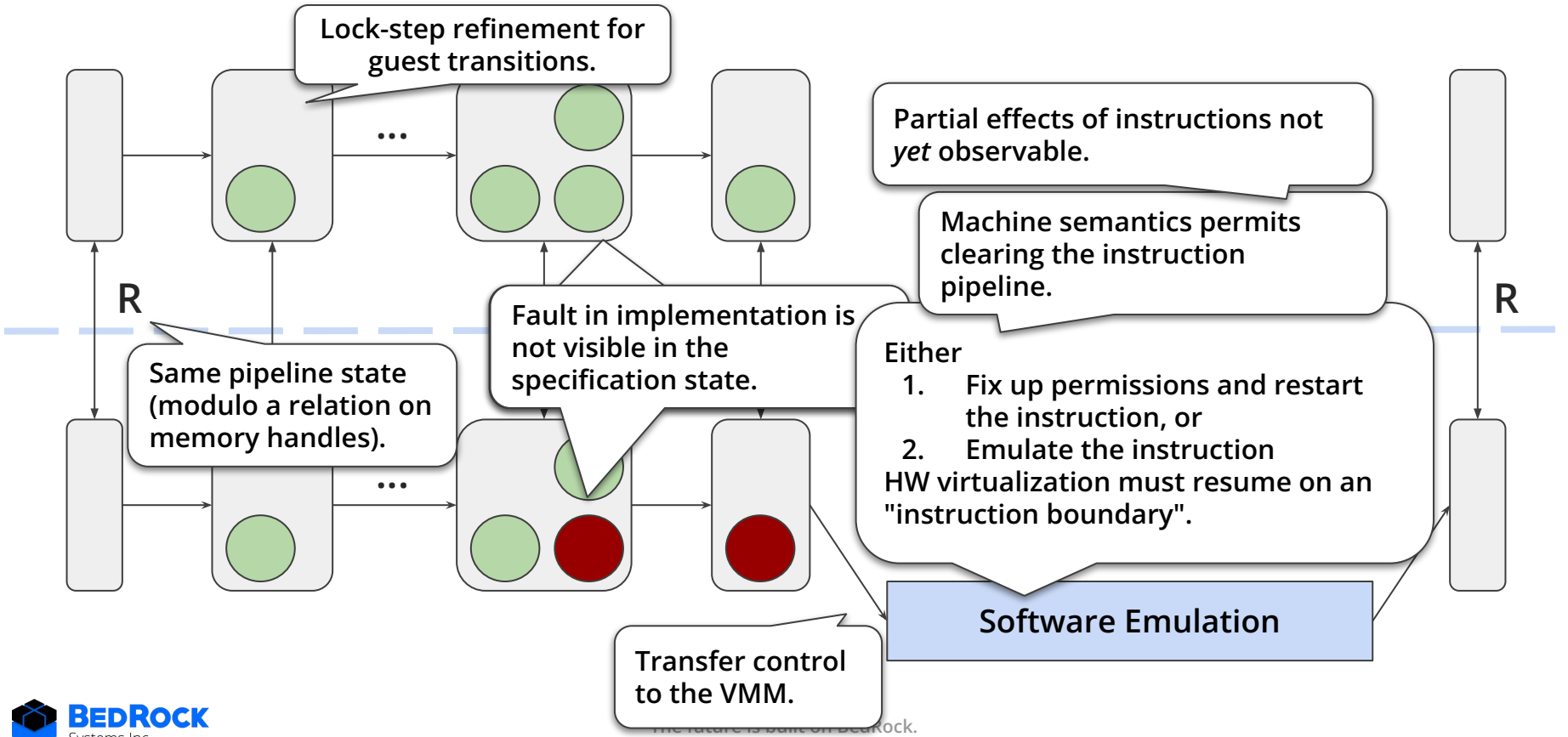
Hardware Virtualization Works



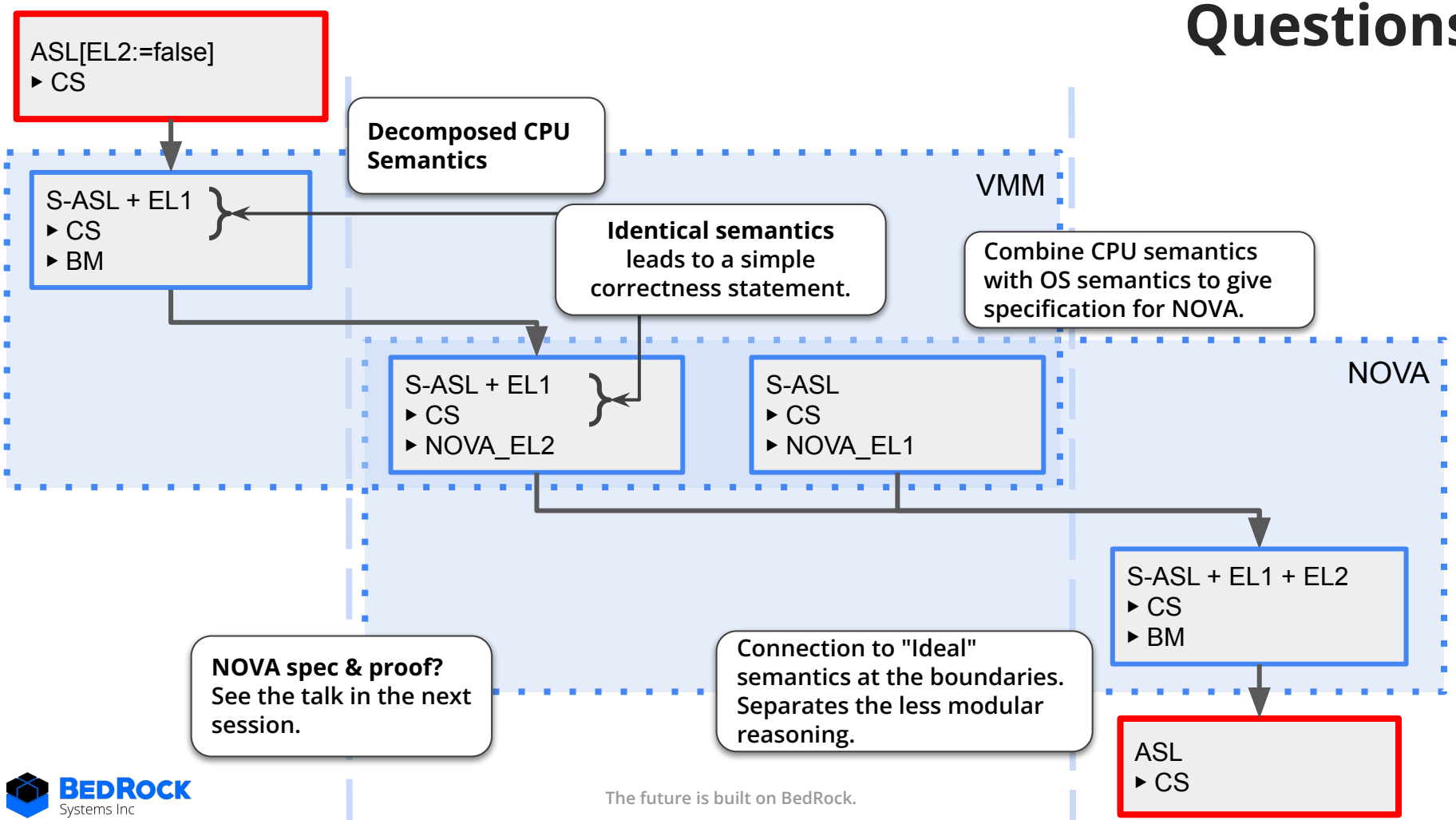
Hardware Virtualization Works



Hardware Virtualization Works



Questions


ASL[EL2:=false]
▸ CS
Decomposed CPU Semantics
VMM
S-ASL + EL1
▸ CS
▸ BM
Identical semantics leads to a simple correctness statement.
Combine CPU semantics with OS semantics to give specification for NOVA.
NOVA
S-ASL + EL1
▸ CS
▸ NOVA_EL2
S-ASL
▸ CS
▸ NOVA_EL1
S-ASL + EL1 + EL2
▸ CS
▸ BM
NOVA spec & proof?
See the talk in the next session.
Connection to "Ideal" semantics at the boundaries. Separates the less modular reasoning.
ASL
▸ CS