UMOC: Unified Modular Ordering Constraints to Unify Cycle- and Register-Transfer-Level Modeling

Shunning Jiang, Yanghui Ou, Peitian Pan, Christopher Batten
Computer Systems Laboratory
Cornell University
Hardware Design Trend

• Hardware Specialization!
• Heterogeneous System-on-Chips (SoC)
Hardware Specialization!

- Hardware Design Trend
- Heterogeneous System on Chips (SoC)

A12 Bionic – Apple
Hardware Specialization

- Heterogeneous System on Chips (SoC)

A12 Bionic – Apple

Exynos 990 – Samsung
Hardware Specialization

- Heterogeneous System-on-Chips (SoC)

Hardware Design Trends

- A12 Bionic – Apple
- Exynos 990 – Samsung
- Jacinto – Texas Instrument
• Hardware Specialization

• Heterogeneous System on Chips (SoC)

A12 Bionic – Apple

Exynos 990 – Samsung

Jacinto – Texas Instrument

Figure 3.1.2: Xbox Series X SoC architecture block diagram.
Hardware Specialization!

- Heterogeneous System on Chips (SoC)

A12 Bionic – Apple

Exynos 990 – Samsung

Jacinto – Texas Instrument

Figure 3.1.2: Xbox Series X SoC architecture block diagram.
Hardware Specialization

- **Heterogeneous System on Chips (SoC)**

**Hardware Design Trend**

- **A12 Bionic – Apple**
- **Exynos 990 – Samsung**
- **Jacinto – Texas Instrument**

Figure 3.1.2: Xbox Series X SoC architecture block diagram.
Hardware Specialization!

- Hardware Design Trend
- Heterogeneous System on Chips (SoC)

A12 Bionic – Apple

Exynos 990 – Samsung

Jacinto – Texas Instrument

Figure 3.1.2: Xbox Series X SoC architecture block diagram.
Cycle-Level Simulators/Models for Design Space Exploration
Cycle-Level Simulators/Models for Design Space Exploration

Choice#1: $80,000
Choice#2: $90,000

Choice#1: $......
Choice#2: $......

Choice#1: $......
Choice#2: $......

Choice#1: $......
Choice#2: $......
Cycle-level Simulators/Models for Design Space Exploration

Cycle-level (CL) modeling:
- Approximate timing behaviors
- Analytical area, energy, timing models
CL models provide valuable insights to help make first-order design decisions (e.g., cycle-level “N-cycle hit-latency cache”)
Composing Cycle-Level/RTL Models for Design Space Exploration
CL/RTL Composition:
- Use some CL models for faster overall simulation
- Gradually replacing CL models with RTL models
Challenge #1: Trade-off between model fidelity and scheduling modularity in cycle-level modeling
Challenge #1: Trade-off between model fidelity and scheduling modularity in cycle-level modeling

```c
void Proc::tick()
{
    writeback();
    mem();
    execute();
    decode();
    fetch();
}

void Accel::tick()
{
    work();
    interface();
}
```
Challenge #1: Trade-off between model fidelity and scheduling modularity in cycle-level modeling

```cpp
void Proc::tick()
{
    writeback();
    mem();
    execute();
    decode();
    fetch();
}

void Accel::tick()
{
    work();
    interface();
}

void Tile::tick()
{
    // modular
    accel.tick();
    proc.tick();
}
```

modular
but inaccurate
Challenge #1: Trade-off between model fidelity and scheduling modularity in cycle-level modeling

```cpp
void Proc::tick()
{
    writeback();
    mem();
    execute();
    decode();
    fetch();
}

void Accel::tick()
{
    work();
    interface();
}
```

```cpp
void Tile::tick()
{
    // modular
    accel.tick();
    proc.tick();
}

void Tile::tick()
{
    // flattened
    proc.writeback();
    accel.work();
    proc.memory();
    accel.interface();
    proc.execute();
    proc.decode();
    proc.fetch();
}
```

**modular but inaccurate**

**accurate but flat**
Challenge #1: Trade-off between model fidelity and scheduling modularity in cycle-level modeling

```c
void Proc::tick()
{
    writeback();
    mem();
    execute();
    decode();
    fetch();
}

void Accel::tick()
{
    work();
    interface();
}
```

```c
void Top::flat_tick()
{
    // hundreds of lines
    mem.array.advance();
    mem.ctrl.work();
    ...
    tile[0].l2.access();
    ...
    tile[5].accel.work();
    tile[5].proc.memory();
    ...
    tile[7].proc.decode();
    ...
    tile[1].proc.fetch();
    ...
}
```

```c
void Tile::tick()
{
    // modulo
    accel.tile();
    proc.tile();
    ...
    proc.writeback();
    proc.accel();
    proc.memory();
    proc.execute();
    proc.decode();
    proc.fetch();
}
```
Challenge #2: Seamless General-Purpose CL/RTL Composition Methodologies

• No seamless CL/RTL compositions
Challenge #2: Seamless General-Purpose CL/RTL Composition Methodologies

- No seamless CL/RTL compositions
- PyMTL: manually CL ordering mixed with event-driven RTL

```python
@src.tick_cl
def block():
    # TODO: we might want to see if this ticking order makes sense
    if s.10_enabled:
        s.icache_mem_req_adapter.xtick()
        s.icache_mem_resp_adapter.xtick()
    if s.dmem_funnels is not None:
        for req_adapter, resp_adapter in zip(s.dcache_mem_req_adapters,
                                              s.dcache_mem_resp_adapters):
            req_adapter.xtick()
            resp_adapter.xtick()
    if s.tmu:
        s.xcelreq_adapter.xtick()
        s.xcelresp_adapter.xtick()
        s.tmuxtick()
```
Challenge #2: Seamless General-Purpose CL/RTL Composition Methodologies

- No seamless CL/RTL compositions
- PyMTL: manually CL ordering mixed with event-driven RTL
- SystemC: RTL/CL communication need to go through a clock edge

```python
@s.tick_cl
def block():
    # TODO: we might want to see if this ticking order makes sense

    if s.l0_enabled:
        s.icache_mem_req_adapter.xtick()
        s.icache_mem_resp_adapter.xtick()

    if s.dmem_funnels is not None:
        for req_adapter, resp_adapter in zip(s.dcache_mem_req_adapters,
                                              s.dcache_mem_resp_adapters):
            req_adapter.xtick()
            resp_adapter.xtick()

    if s.tmu:
        s.xcelreq_adapter.xtick()
        s.xcelresp_adapter.xtick()
        s.tmu.xtick()
```
Challenge #2: Seamless General-Purpose CL/RTL Composition Methodologies

- No seamless CL/RTL compositions
- PyMTL: manually CL ordering mixed with event-driven RTL
- SystemC: RTL/CL communication need to go through a clock edge
- ... other ad-hoc approaches

```python
@s.tick_cl
def block():
    # TODO: we might want to see if this ticking order makes sense
    if s.l0_enabled:
        s.icache_mem_req_adapter.xtick()
        s.icache_mem_resp_adapter.xtick()

    if s.dmem_funnels is not None:
        for req_adapter, resp_adapter in zip( s.dcache_mem_req_adapters, s.dcache_mem_resp_adapters ):
            req_adapter.xtick()
            resp_adapter.xtick()

    if s.tmu:
        s.xcelreq_adapter.xtick()
        s.xcelresp_adapter.xtick()
        s.tmu.xtick()
```
Challenge #2: Seamless General-Purpose CL/RTL Composition Methodologies

• No seamless CL/RTL compositions
• PyMTL: manually CL ordering mixed with event-driven RTL
• SystemC: RTL/CL communication need to go through a clock edge
• ... other ad-hoc approaches

```python
@tick_cl
def block():
    # TODO: we might want to see if this ticking order makes sense
    if s.l0_enabled:
        s.icache_mem_req_adapter.xtick()
        s.icache_mem_resp_adapter.xtick()

    if s.dmem_funnels is not None:
        for req_adapter, resp_adapter in zip(s.dcache_mem_req_adapters,
                                              s.dcache_mem_resp_adapters):
            req_adapter.xtick()
            resp_adapter.xtick()

    if s.tmu:
        s.xcelreq_adapter.xtick()
        s.xcelresp_adapter.xtick()
        s.tmu.xtick()
```
Unified Modular Ordering Constraints (UMOC)

Unified abstraction for signal-based RTL modeling and method-based CL modeling
Unified Modular Ordering Constraints (UMOC)

Unified abstraction for signal-based RTL modeling and method-based CL modeling

\[ \begin{align*}
x \text{ is a combinational wire} \& \quad A \text{ writes signal } x \\
& \quad B \text{ reads signal } x \quad \implies \quad A \text{ precedes } B \quad (A < B)
\end{align*} \]
Unified Modular Ordering Constraints (UMOC)

Unified abstraction for signal-based RTL modeling and method-based CL modeling

\[
x \text{ is a combinational wire} \\
A \text{ writes signal } x \\
B \text{ reads signal } x
\]

\[\implies A \text{ precedes } B \quad (A < B)\]
Unified Modular Ordering Constraints (UMOC)

Unified abstraction for signal-based RTL modeling and method-based CL modeling

\[
\begin{align*}
\text{x is a combinational wire} & \quad A \text{ writes signal } x \\
\text{B reads signal } x & \quad \Rightarrow \quad A \text{ precedes } B \\
& \quad (A < B)
\end{align*}
\]

```cpp
void Proc::tick()
{
    writeback();
    mem();
    execute();
    decode();
    fetch();
}

void Proc::execute()
{
    auto i = DX_q.dequeue();
    switch (i.type) {
        case ...

    SWITCH

    XM_q.enqueue(...);
}
```
Unified Modular Ordering Constraints (UMOC)

Unified abstraction for signal-based RTL modeling and method-based CL modeling

\[ x \text{ is a combinational wire} \]
\[ A \text{ writes signal } x \]
\[ B \text{ reads signal } x \]
\[ \implies A \text{ precedes } B \]
\[ (A < B) \]

```cpp
void Proc::decode()
{
    auto i = FD_q.dequeue();
    ...
    if (i.is accel inst)
        Accel_q.enqueue(...);
    DX_q.enqueue(...);
}

void Proc::tick()
{
    writeback();
    mem();
    execute();
    decode();
    fetch();
}

void Proc::execute()
{
    auto i = DX_q.dequeue();
    switch (i.type) {
        ...
        XM_q.enqueue(...);
    }
```

\[ \text{subcomponent subcomponent_instance_name (}
    .clk ( clk_sub ), // input
    .rst_n ( rst_n ), // input
    .data_rx ( data_rx ), // input [9:0]
    .data_tx ( data_tx ) // output [9:0]
\)

\[ \text{q.dequeue precedes q.enqueue} \]
\[ A \text{ calls q.dequeue} \]
\[ B \text{ calls q.enqueue} \]
\[ \implies A \text{ precedes } B \]
\[ (A < B) \]
We insert a queue with deq<enq to accelerator
CL Model Fidelity & Schedule Modularity

- We insert a queue with deq<enq to accelerator
  - The interface process invokes deq
CL Model Fidelity & Schedule Modularity

- We insert a queue with deq<enq to accelerator
  - The interface process invokes deq
  - Expose its enq method to the parent tile
We insert a queue with deq<enq to accelerator
- The interface process invokes deq
- Expose its enq method to the parent tile
- Pass the enq method to processor
CL Model Fidelity & Schedule Modularity

• We insert a queue with `deq<enq` to accelerator
  • The interface process invokes `deq`
  • Expose its `enq` method to the parent tile
  • Pass the `enq` method to processor
  • The decode process invokes `enq`
We insert a queue with deq<enq to accelerator
- The interface process invokes deq
- Expose its enq method to the parent tile
- Pass the enq method to processor
- The decode process invokes enq
- Global scheduler: interface before decode
Seamless CL/RTL Composition

- **x** is a combinational wire
  - A writes signal x
  - B reads signal x

  \[ \Rightarrow A \text{ precedes } B \quad (A < B) \]

- **q.dequeue** precedes **q.enqueue**
  - A calls q.dequeue
  - B calls q.enqueue

  \[ \Rightarrow A \text{ precedes } B \quad (A < B) \]

- \( x \) is signal
  - q.dequeue < q.enqueue

  \[
  \begin{align*}
  A: & \quad x = y + 1 \\
  B: & \quad q.dequeue(x \times 2) \\
  C: & \quad z = q.dequeue() 
  \end{align*}
\]

- \( \rightarrow \): explicit constraint
- \( \rightarrow \): implicit constraint
Seamless CL/RTL Composition

• Creating the Unified Directed Graph (UDG)
  • Edges include implicit and explicit ordering constraints
Seamless CL/RTL Composition

- Creating the Unified Directed Graph (UDG)
  - Edges include implicit and explicit ordering constraints
  - Loops between RTL processes are allowed
Seamless CL/RTL Composition

• Creating the Unified Directed Graph (UDG)
  • Edges include implicit and explicit ordering constraints
  • Loops between RTL processes are allowed
  • CL processes are not allowed to appear in any loop

\[
\begin{align*}
\text{x is a combinational wire} & \quad (A < B) \\
A \text{ writes signal } x & \\
B \text{ reads signal } x & \\
q.\text{dequeue \ precedes } q.\text{enqueue} & \\
A \text{ calls } q.\text{dequeue} & \\
B \text{ calls } q.\text{enqueue} & \\
\implies & \\
A \text{ precedes } B & (A < B) \\
\end{align*}
\]

\[
\begin{align*}
x \text{ is signal } & \quad q.\text{dequeue} < q.\text{enqueue} \\
A: \quad x = y + 1 & \\
B: \quad \text{enqueue}(x + 2) & \\
C: \quad z = q.\text{dequeue}() & \\
\text{a,b,x,y,z are signals} & \\
A: \quad x = a + 1 & z = y + 1 \\
B: \quad y = x + 1 & \\
C: \quad b = y \times 2 & \\
\text{q1.dequeue < q1.enqueue} & \\
A: \quad x = a + 1 & \text{q1.enqueue}(a) \\
B: \quad x = \text{q1.dequeue}() & b = y + x \\
C: \quad z = y \times 2 & \\
\end{align*}
\]
Scheduling the Unified Directed Graph

• Some properties of the UDG:
  • CL processes execute exactly once per cycle
  • RTL processes need to execute until value stabilize
Scheduling the Unified Directed Graph

• Some properties of the UDG:
  • CL processes execute exactly once per cycle
  • RTL processes need to execute until value stabilize

  ▪ If the UDG has no cycle
    • Topological sort that statically schedules all processes in the DAG
Scheduling the Unified Directed Graph

• Some properties of the UDG:
  • CL processes execute exactly once per cycle
  • RTL processes need to execute until value stabilize

  ▪ If the UDG has no cycle
    • Topological sort that statically schedules all processes in the DAG
  ▪ If the UDG has cycle
    • Strongly connected components (SCC) algorithm
      » Shrink cycles into a single node
    • Execute the DAG of SCCs
      » Topological sort of the DAG
      » Iteratively Execute the SCC
UMOC Implemented in PyMTL3

• PyMTL3 is a state-of-the-art Python-based hardware generation and simulation framework

• PyMTL3 is very extensible thanks to modular framework architecture
  • Frontend: Embedded domain specific language (EDSL) modeling primitives
  • IR: Native in-memory intermediate representation (NIMIR)
  • Backend: Passes that systematically manipulate NIMIR

• UMOC implemented in PyMTL3:
  • EDSL modeling primitives
  • NIMIR data structures
  • Graph generation and scheduling passes
UMOC Implementation of PyMTL3 EDSL Primitives

- UMOC PyMTL3 EDSL primitives:
  - Inherit from Component
  - InPort, OutPort, Wire, CalleePort, CallerPort
  - @update_ff, @update, @update_once, @method_port
  - add_constraints

```python
class RegIncrRTL( Component ):
    def construct( s ):
        s.in_ = InPort (32)
        s.out = OutPort(32)
        s.reg = Wire(32)

    @update_ff
    def seq_reg():
        s.reg <<= s.in_

    @update
    def comb_out():
        s.out @= s.reg + 1

class RegIncrCL( Component ):
    def construct( s ):
        # Model sequential behavior!
        s.add_constraints(
            M(s.read) < M(s.write),
        )

    @method_port
    def read( s ):
        return s.v + 1

    @method_port
    def write( s, v ):
        s.v = v

class RegIncrCLRTL( Component ):
    def construct( s ):
        s.write = CalleePort()
        s.out = OutPort(32)
        s.r1 = RegIncrCL()  
        s.r2 = RegIncrRTL()

    @update_once
    def send_to_r2():
        s.r2.in_ @= s.r1.read()
```

UMOC Implementation of PyMTL3 NIMIR & Passes
UMOC Implementation of PyMTL3 NIMIR & Passes

• Supporting UMOC in PyMTL3 NIMIR elaboration
UMOC Implementation of PyMTL3 NIMIR & Passes

• Supporting UMOC in PyMTL3 NIMIR elaboration
  • Collecting all the update blocks and ordering constraints
UMOC Implementation of PyMTL3 NIMIR & Passes

• Supporting UMOC in PyMTL3 NIMIR elaboration
  • Collecting all the update blocks and ordering constraints
  • Exposing all metadatas with APIs
UMOC Implementation of PyMTL3 NIMIR & Passes

• Supporting UMOC in PyMTL3 NIMIR elaboration
  • Collecting all the update blocks and ordering constraints
  • Exposing all metadatas with APIs

• UMOC passes
UMOC Implementation of PyMTL3 NIMIR & Passes

• Supporting UMOC in PyMTL3 NIMIR elaboration
  • Collecting all the update blocks and ordering constraints
  • Exposing all metadatas with APIs

• UMOC passes
  • GenUDGPass to generate the unified directed graph
    • Update blocks as vertices, explicit/implicit constraints as edges
  • UMOCSchedulingPass to schedule the UDG
UMOC Implementation of PyMTL3 NIMIR & Passes

• Supporting UMOC in PyMTL3 NIMIR elaboration
  • Collecting all the update blocks and ordering constraints
  • Exposing all metadatas with APIs

• UMOC passes
  • GenUDGPass to generate the unified directed graph
    • Update blocks as vertices, explicit/implicit constraints as edges
  • UMOCSchedulingPass to schedule the UDG

• The user only need to set local explicit ordering constraints. No global scheduling required.
UMOC Case Study in PyMTL3
UMOC Case Study in PyMTL3

• 5-stage RTL Processor, 3-stage CL processor
UMOC Case Study in PyMTL3

• 5-stage RTL Processor, 3-stage CL processor
• RTL/CL checksum accelerators
UMOC Case Study in PyMTL3

• 5-stage RTL Processor, 3-stage CL processor
• RTL/CL checksum accelerators
• Manual == Execute everything in accelerator before processor (or vice versa)

```cpp
void Tile::tick()
{
    // ac
    // pr
}
```
UMOC Case Study in PyMTL3

• 5-stage RTL Processor, 3-stage CL processor
• RTL/CL checksum accelerators
• Manual == Execute everything in accelerator before processor (or vice versa)

```cpp
void Tile::tick()
{
    // modular
    accel.tick();
    proc.tick();
}
```
UMOC Case Study in PyMTL3

- 5-stage RTL Processor, 3-stage CL processor
- RTL/CL checksum accelerators
- Manual == Execute everything in accelerator before processor (or vice versa)

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Composition</th>
<th>#Cycles</th>
<th>Deviation</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event-driven</td>
<td>RTL Proc + RTL Accel</td>
<td>565</td>
<td>-</td>
<td>baseline</td>
</tr>
<tr>
<td>UMOC</td>
<td>RTL Proc + RTL Accel</td>
<td>565</td>
<td>0%</td>
<td>same as baseline</td>
</tr>
<tr>
<td>UMOC</td>
<td>CL Proc + CL Accel</td>
<td>541</td>
<td>4%</td>
<td>due to 3-stage</td>
</tr>
<tr>
<td>Manual Proc&lt;Accel</td>
<td>CL Proc + CL Accel</td>
<td>416</td>
<td>26%</td>
<td>modular sub-tick</td>
</tr>
<tr>
<td>Manual Accel&lt;Proc</td>
<td>CL Proc + CL Accel</td>
<td>416</td>
<td>26%</td>
<td>modular sub-tick</td>
</tr>
<tr>
<td>UMOC</td>
<td>CL Proc + RTL Accel</td>
<td>541</td>
<td>4%</td>
<td>same as CL+CL</td>
</tr>
<tr>
<td>UMOC</td>
<td>RTL Proc + CL Accel</td>
<td>565</td>
<td>0%</td>
<td>same as RTL+RTL</td>
</tr>
</tbody>
</table>
CL/RTL Compositions Helps Chip Tape-outs

- Main-memory only needs CL
- CL shared MDU/FPU for DSE
- CL cache for DSE
- CL on-chip networks for DSE
- Processor IP already developed
Takeaways & Conclusion

• UMOC’s explicit ordering constraints achieves model fidelity and scheduling modularity at once.
• UMOC’s implicit & explicit constraints achieves seamless CL/RTL composition.
• UMOC has been implemented in PyMTL3. Many IPs have been built using UMOC scheme.