Type-Directed Compilation for Multicore Programming

Kohei Honda (Queen Mary, University of London)
Vasco T. Vasconcelos (University of Lisbon)
Nobuko Yoshida (Imperial College London)

PLACES'08, June 7, 2008, Oslo

Concurrency at the Core(s) of Computing

- From monolithic Von Neumann architectures to multicore CPUs

  "multiple concurrent modules in a single application"

- A multicore CPU can be:
  - SMP (cache coherent): Intel, Niagara, ..
  - NUMA (non-cache coherent): MPSoC, Cell, ..

- DMA (Direct Memory Access) used in NUMA.

  ⟹ Issue: fast but *unwieldy* and *dangerous*

Target Machine Model (1)

- We assume the following simple machine model:
  - Consisting of multiple isomorphic VNMs (same ISA), each with local memory
  - Data sharing among cores through asynchronous writes between local memories (DMA, only the push-version in this talk)
- Close to the LogGP model in parallel computing
- Local memory size, OCIN topology, etc.: deliberately unspecified

Target Machine Model (2)

Why choose this model?

- Simple, efficient, general
- Close to many MPSoCs (including Cell)
- Harder to program than SMP — but is it true?

NB: *Other CMP issues deliberately ignored.*

Using Types for Interactions

- Session Types for high-level abstraction of conversation structures
- *First* specifies scenarios of conversations; *then* program and validate conformance
- Ensures communication/synchronisation safety (and other properties such as liveness)
- Can be a basis of other correctness arguments
- Can be a basis of efficient executions

A Type-Directed Compilation Framework

L2 Domain Specific Language (a la StreamFlex)

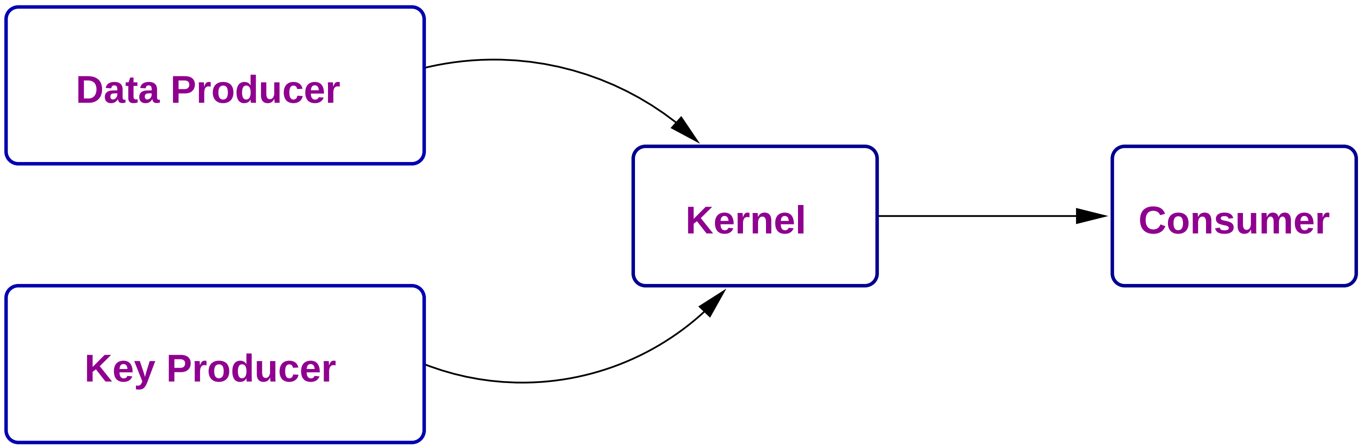
$\Downarrow$

L1 Applied π-Calculus with Session Types

$\Downarrow$

L0 Typed Assembly Language for CMP

L1: Streaming Example (1)



DataProducer

$$\text{def } \mathrm{P}(d, k, c) = d?();\ d!\langle data\rangle; \mathrm{P}\langle d, k, c\rangle \ \text{ in } \ a(d, k, c).\mathrm{P}\langle d, k, c\rangle$$

L1: Streaming Example (2)

Kernel

$$\text{def } \mathrm{K}(d, k, c)$$
$$= \; d!\langle\rangle;\; k!\langle\rangle;\; d?(x);\; k?(y);\; c?();\; c!\langle x \text{ xor } y\rangle;\; \mathrm{K}\langle d, k, c\rangle$$
$$\text{in } \; \overline{a}(d, k, c).\mathrm{K}\langle d, k, c\rangle$$

Types for Kernel

$$\mu\mathbf{t}.d!\,\langle\rangle; k!\,\langle\rangle; d?\,\langle\text{bool}\rangle; k?\,\langle\text{bool}\rangle; c?\,\langle\rangle; c!\,\langle\text{bool}\rangle; \mathbf{t}$$

Types for DataProducer

$$\mu\mathbf{t}.d?\,\langle\rangle; d!\,\langle\text{bool}\rangle; \mathbf{t}$$

L0: Streaming Example

```
main: {
  main: {
    r1 := getIdleCore
    r2 := getIdleCore
    r3 := getIdleCore
    r4 := getIdleCore
    fork dataProducer at r1
    fork keyProducer at r2
    fork kernel at r3
    fork consumer at r4
    yield
  }
}
```

```
dataProducer: {
  data: byte [128]
  ack: byte [0]
  main: {
    // produce data
    get ack
    put data in r3.data
    jump main
  }
}
```

L0: Streaming Example (1)

```
keyProducer: {
  key: byte [128]
  ack: byte [0]
  main: {
    // produce key
    get ack
    put key in r3.key
    jump main
  }
}
```

```
consumer: {
  buf: byte [128]
  ack: byte [0]
  main: {
    get buff
    // consume buf
    put ack in r3.ack
    jump main
  }
}
```

L0: Streaming Example (2)

```
kernel: {
  data: byte [128]
  key: byte [128]
  buf: byte [128]
  ackD: byte [0]
  ackK: byte [0]
  ackC: byte [0]
  main: {
    put ackD in r1.ack
    put ackK in r2.ack
    get data; get key
    r5 := 128; jump loop
  }
  loop: {
    when r5 < 0 jump done
    r6 := data[r4]; r7 := key[r4]
    buf[r4] := r7 xor r6;
    jump loop
  }
  done: {
    get ackS
    put sum in r1.arg
    jump main
  }
}
```

L0: Streaming Example (3)

A closer look at **get**:

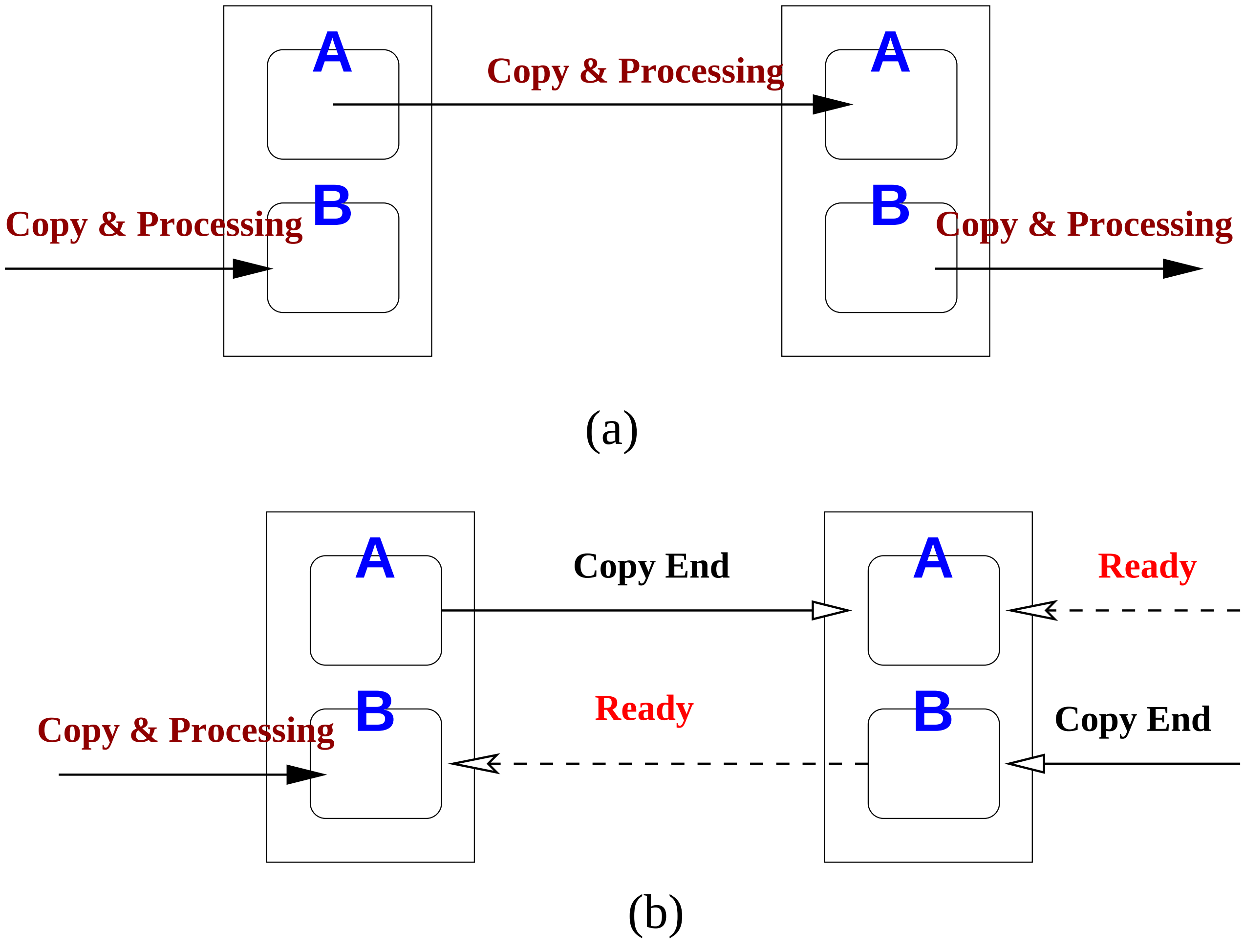
```
.....
get ack
.....
```

⟹

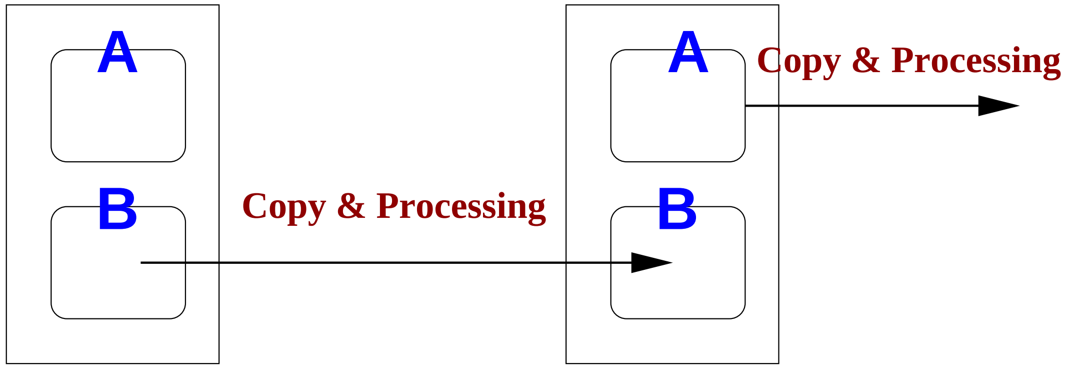
```
.....
__wait__ack: {
    when __received__ack jump done
    jump __wait__ack
}
done: {
    .....
}
```

where `__ack__received` is an implicit flag for `ack`, to be filled at the end of the corresponding **put**.

Double Buffering



Double Buffering



(c)

Types for Kernel

$$s \triangleleft \mathsf{ReadyA}; s \triangleleft \mathsf{ReadyB};$$

$$\mu \mathbf{t}.s?\,\langle T\rangle; k \triangleright \mathsf{ReadyA}; k!\,\langle T\rangle; s \triangleleft \mathsf{ReadyA}; s?\,\langle T\rangle;$$

$$k \triangleright \mathsf{ReadyB}; k!\,\langle T\rangle; \mathbf{t}$$

Related Works

- [Ennals et al 04; ...]: Thread-sensitive linear types for compiling packet processing to CMP.
- [Fahndrich et al 06; ...]: C# extended with a variant of session types for shared memory — for writing an OS.
- [Charles et al 05; ...]: Java extended with structured concurrent programming on PGAS — for HPC.
- [Spring et al 07]: JVM-based approach to fast stream programming.
- [Welch and Barnes 05]: A very fast CSP/π-based systems programming language.

Future Topics

- Can the framework really produce tight, fast code for a wide class of (stream) applications?
- Scalability to other application domains?
- Scalability to different HW configurations?
- Static (and dynamic) analyses for L1?
- Construction of performance models (cf. Log(G)P)
- Semantics, logic, ...