From OO to FPGA

Fitting Round Objects into Square Hardware?

Based on Stephan Kou and Jens Palsberg paper but first...

What is FPGA?

(z slajdów Marcina Peczarskiego oraz Szymona Acedańskiego)

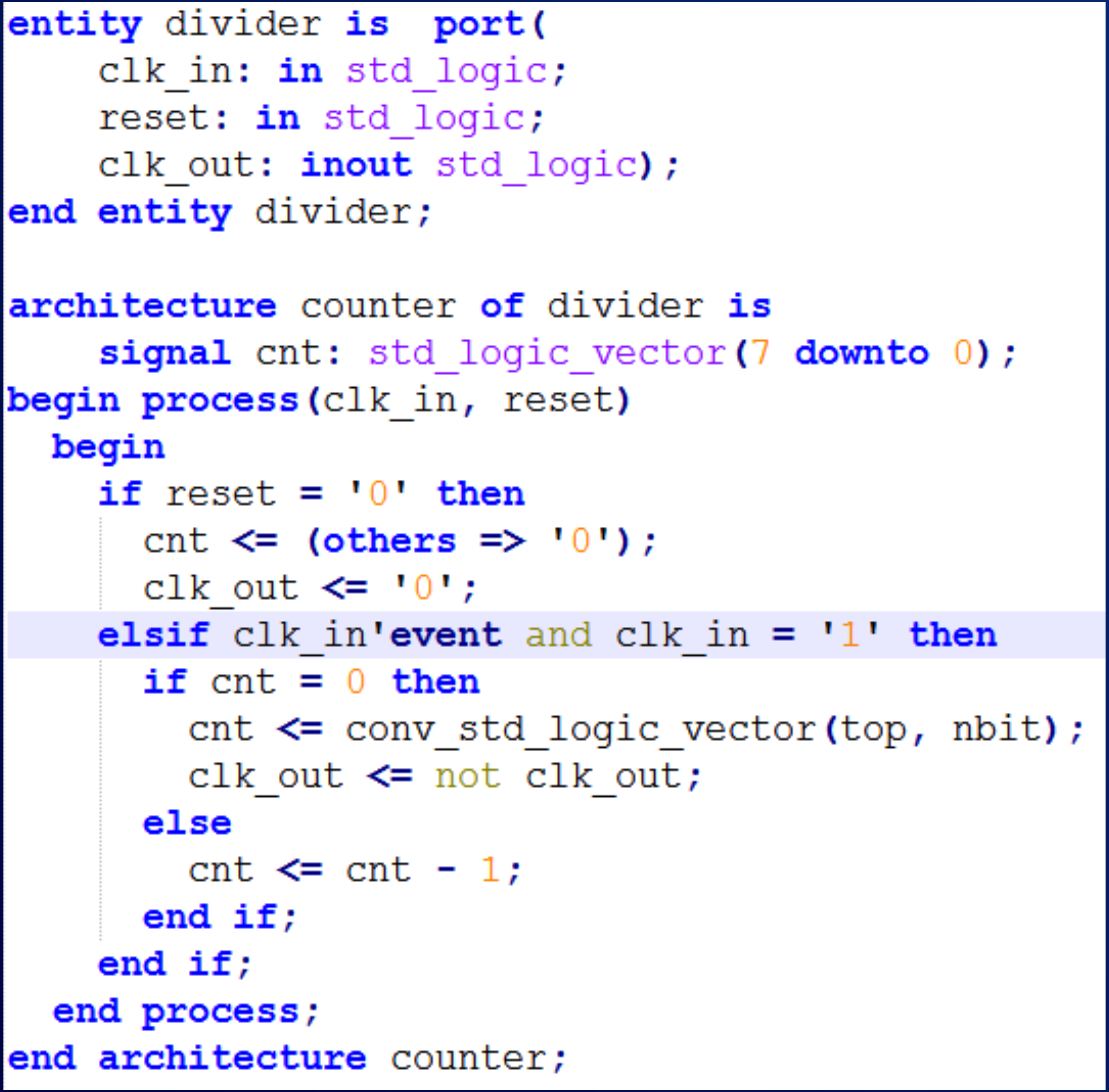
Co zrobić, gdy potrzebujemy układu cyfrowego?

- Znaleźć gotowy układ realizujący potrzebną nam funkcję.
    - Dla wielu typowych zastosowań produkuje się gotowe układy scalone.
    - A co, gdy potrzebujemy nietypowego układu?
- Zbudować potrzebny układ z układów scalonych małej i średniej skali integracji, np. z układów serii 74HC.
    - Opłacalne tylko, gdy projektowany układ jest niewielki i da się złożyć z co najwyżej kliku układów scalonych.
    - Układ składający się z kilkudziesięciu lub kilkuset układów scalonych jest za duży, za drogi w produkcji i pobiera za dużo prądu.
- Zbudować układ mikroprocesorowy i zaprogramować w nim potrzebną funkcjonalność.
    - Mikrokontrolery mają wiele przydatnych peryferii.
    - Obecnie jest to bardzo powszechna praktyka.

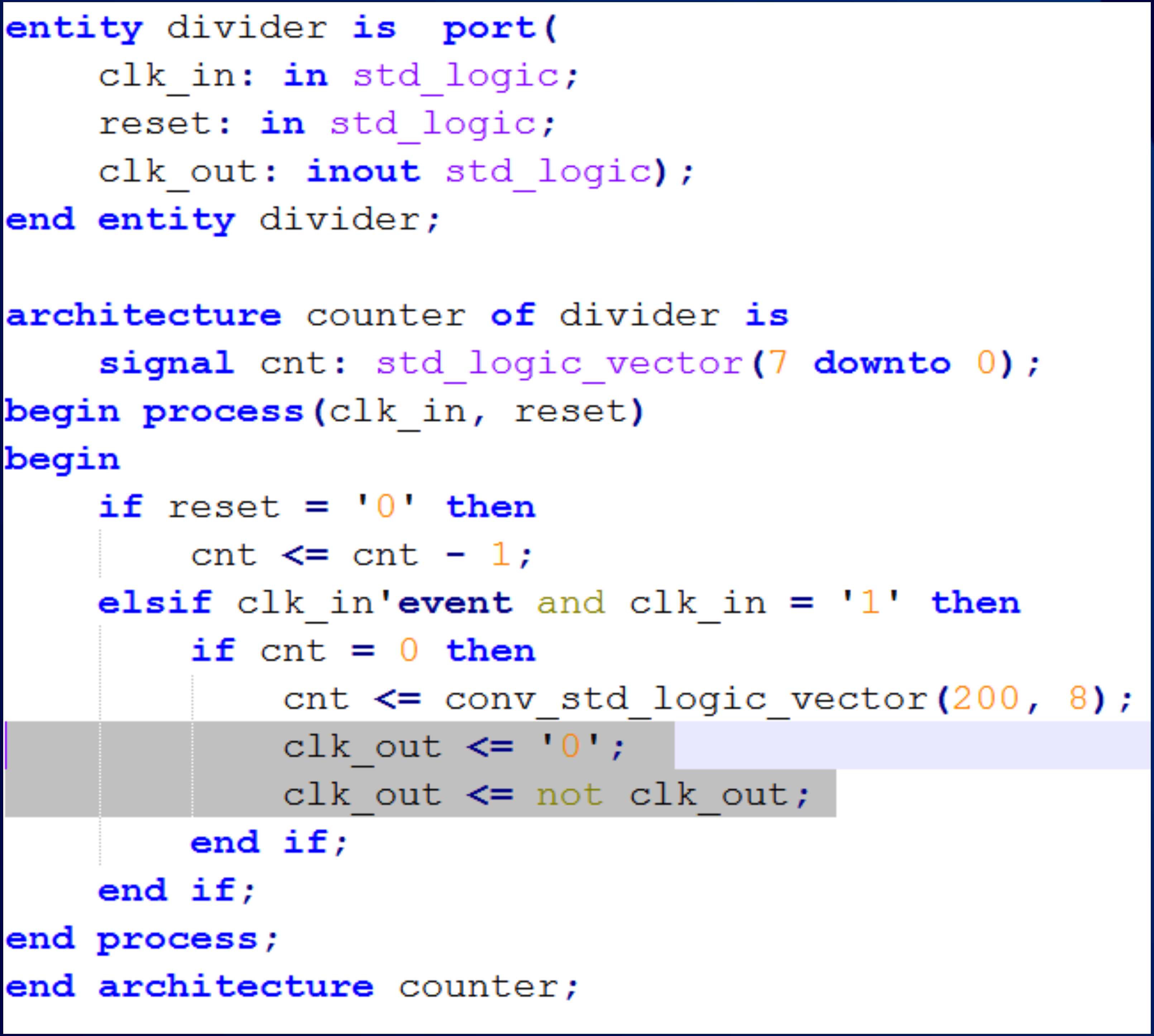
Co zrobić, gdy potrzebujemy układu cyfrowego?

- Zaprojektować własny specjalizowany (ang. *full custom*) układ scalony od podstaw:
  - trzeba poznać bardzo skomplikowane reguły projektowe;
  - proces projektowania jest długi i kosztowny;
  - opłaca się tylko przy bardzo dużych seriach produkcyjnych.
- Użyć ASIC (ang. *application specific integrated circuit*):
  - bramki logiczne są już wykonane w krzemie;
  - wymaga tylko zaprojektowania połączeń między bramkami;
  - opłaca się tylko przy dużych seriach produkcyjnych.
- Użyć FPGA:
  - proces projektowania jest istotnie krótszy i tańszy niż dla układów specjalizowanych lub ASIC;
  - opłaca się przy małych i dużych seriach produkcyjnych;
  - można tanio i szybko skonstruować i przetestować prototyp przed podjęciem decyzji o projektowaniu specjalizowanego układu scalonego;
  - układ wolniejszy od układu specjalizowanego.

So what's the problem with FPGA?



```vhdl
entity divider is  port(
    clk_in: in std_logic;
    reset: in std_logic;
    clk_out: inout std_logic);
end entity divider;

architecture counter of divider is
    signal cnt: std_logic_vector(7 downto 0);
begin process(clk_in, reset)
  begin
    if reset = '0' then
      cnt <= (others => '0');
      clk_out <= '0';
    elsif clk_in'event and clk_in = '1' then
      if cnt = 0 then
        cnt <= conv_std_logic_vector(top, nbit);
        clk_out <= not clk_out;
      else
        cnt <= cnt - 1;
      end if;
    end if;
  end process;
end architecture counter;
```



```
entity divider is  port(
    clk_in: in std_logic;
    reset: in std_logic;
    clk_out: inout std_logic);
end entity divider;

architecture counter of divider is
    signal cnt: std_logic_vector(7 downto 0);
begin process(clk_in, reset)
begin
    if reset = '0' then
        cnt <= cnt - 1;
    elsif clk_in'event and clk_in = '1' then
        if cnt = 0 then
            cnt <= conv_std_logic_vector(200, 8);
            clk_out <= '0';
            clk_out <= not clk_out;
        end if;
    end if;
end process;
end architecture counter;
```

Maybe we could use OO language?

but…

JAVA/C++/ect
bad idea

- Dynamic memory allocation
  - memory blocks can not be requested on runtime
- Pointers
  - memory is not continuous space
  - problem with re-assigning pointers
- Function pointers
  - on FPGA function is a black boxed entity with physical connection between caller and callee (think of virtual methods and virtual tables)
- Recursion
  - for n-deep recursion we would have to synthesis n instances of the function
  - how would we know how big n can be?

but there is Virgil
"light-weight OO programming language"

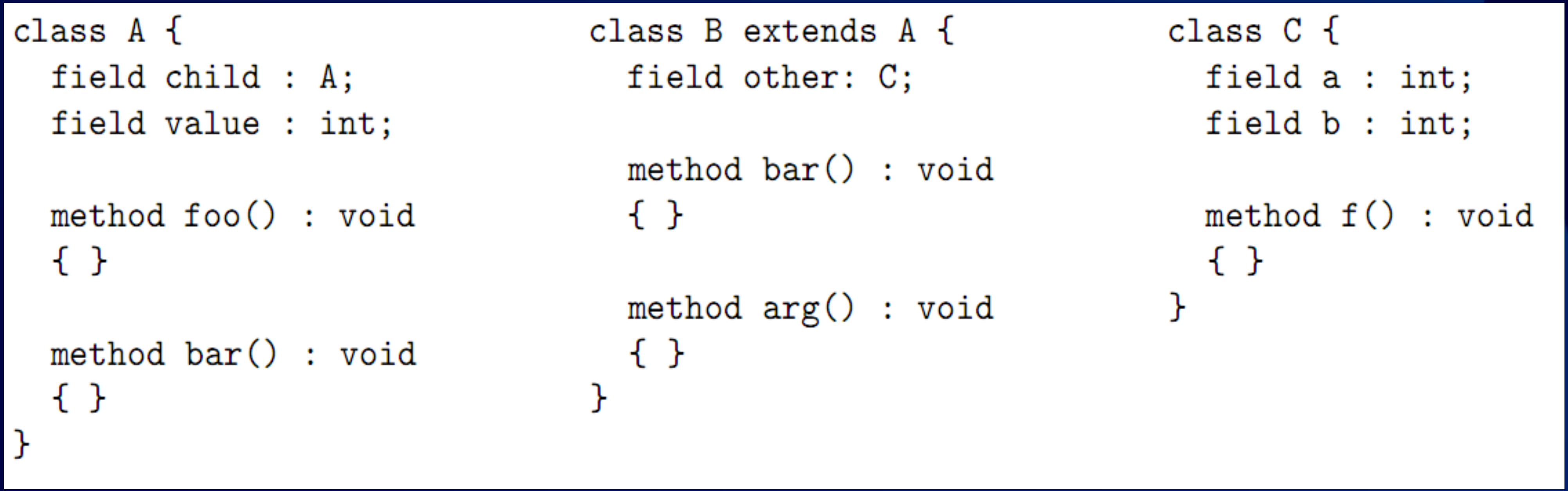
- Java/C style syntax
- No dynamic memory allocation
  - memory can be allocated ONLY in constructors
  - two phases: initialization and execution
- Every method is virtual
- Simple objects hierarchy
  - only single inheritance (only parent <-> child relation)
- Recursion and delegates (function pointers)
  - both are present in virgil, but they won't support them :(

What are they doing?

AutoPilot FPGA

- translating Virgil to C-subset
  - without recursion
  - no C-pointers
  - no function pointers
- run AutoPilot to convert C code to VHDL
- let the standard (Xilinx) toolchain takeover from here

A set of classes written in Virgil



```
class A {
  field child : A;
  field value : int;

  method foo() : void
  { }

  method bar() : void
  { }
}
```

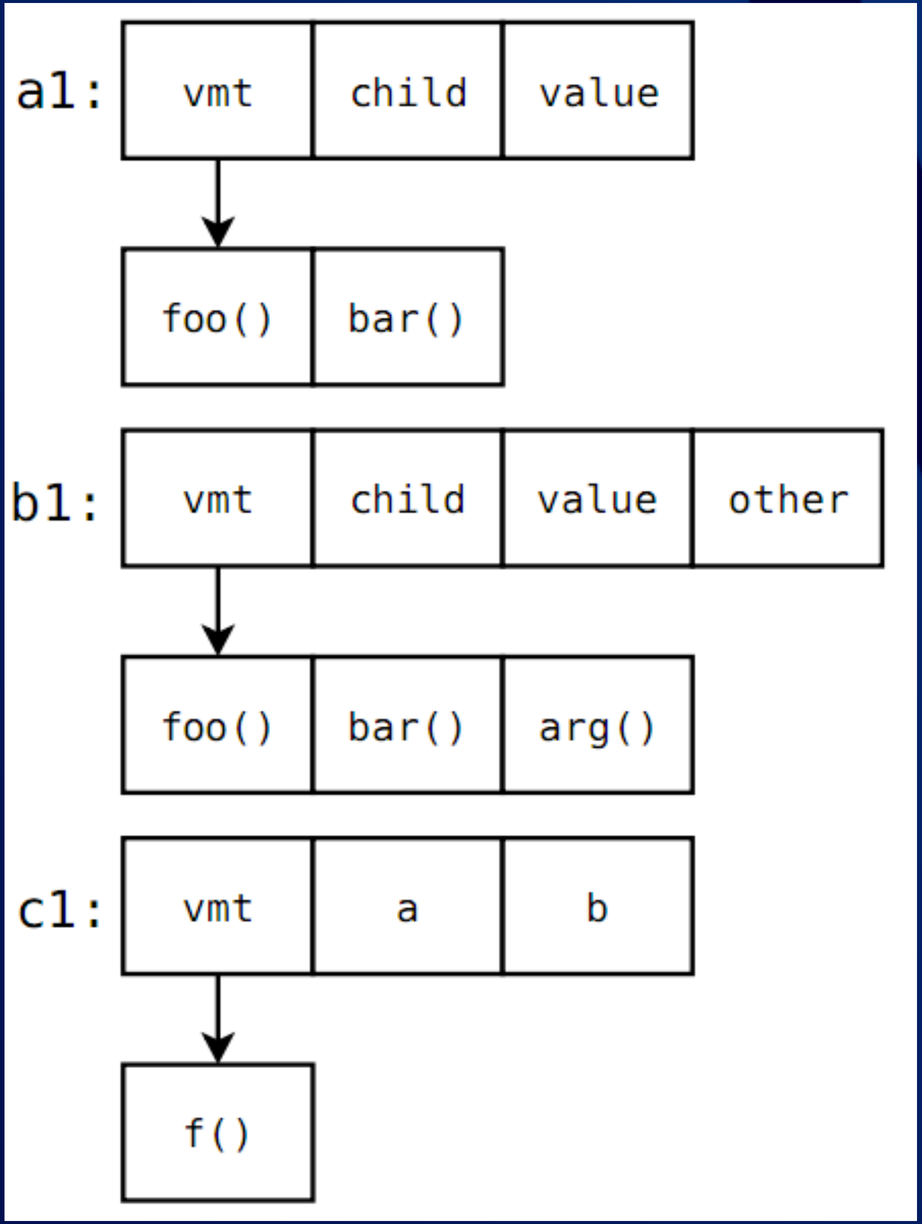
```
class B extends A {
  field other: C;

  method bar() : void
  { }

  method arg() : void
  { }
}
```

```
class C {
  field a : int;
  field b : int;

  method f() : void
  { }
}
```

Memory layout of A, B and C
Horizontal model (standard one)

- Polymorphism by pointers casting

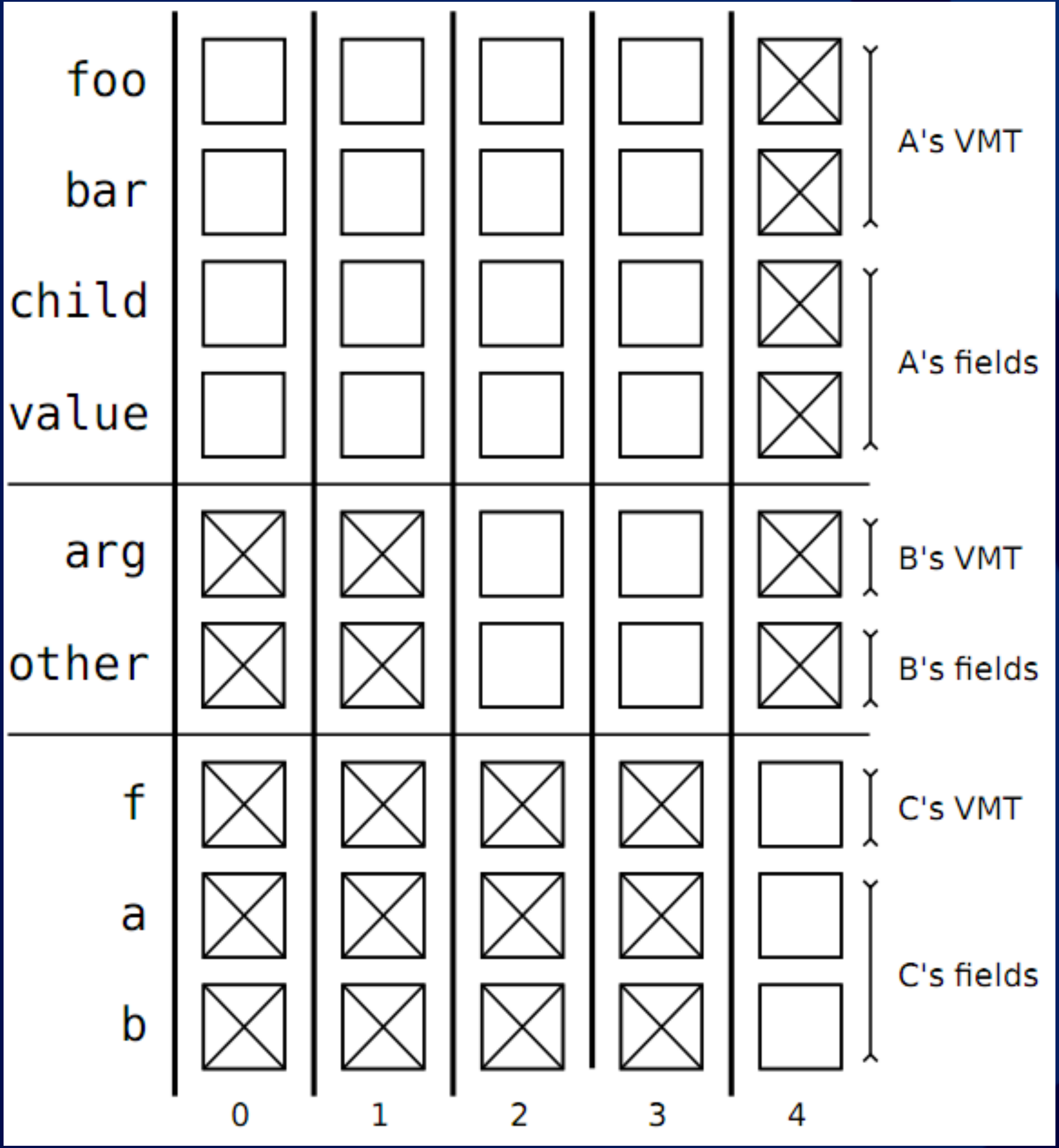
- VMT



Memory layout of A, B and C

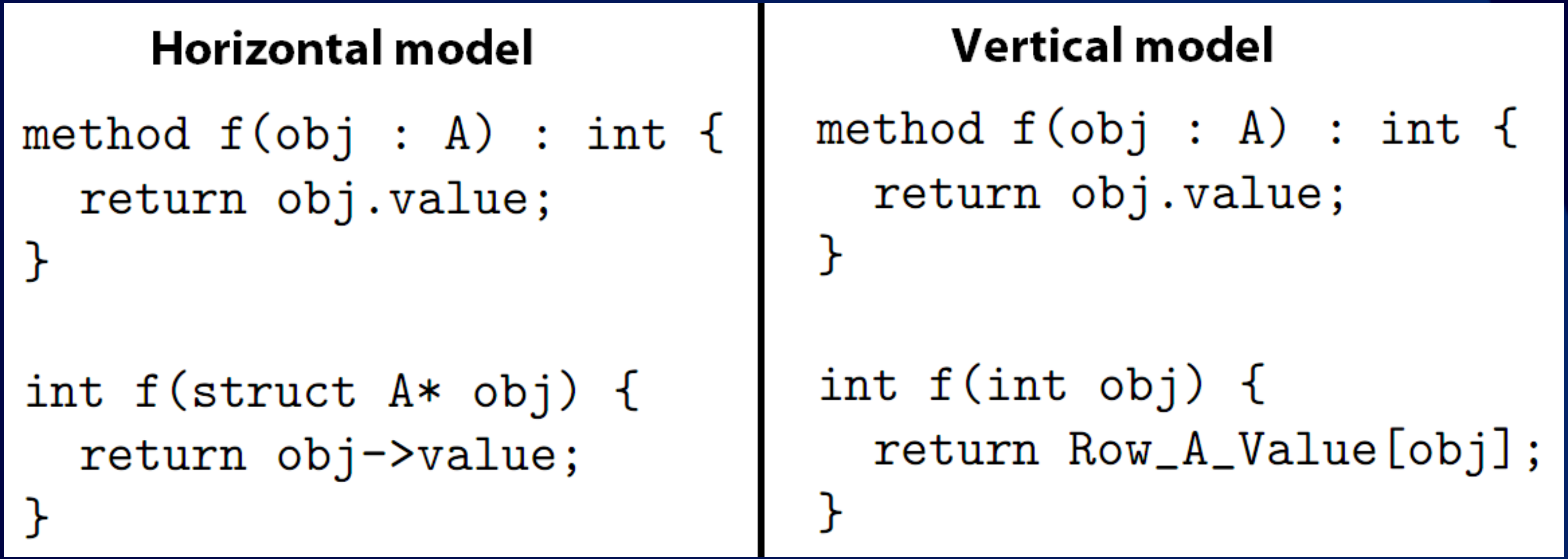
Vertical uncompressed model

- One table (row) per class field/method
- Column represents an instance of the class
- Indexes as pointers
- Memory waste
- VMT will be dealt later



Memory layout of A, B and C

Vertical vs Horizontal model fields accessing



Horizontal model	Vertical model

```
method f(obj : A) : int {
  return obj.value;
}

int f(struct A* obj) {
  return obj->value;
}
```
	```
	method f(obj : A) : int {
	  return obj.value;
	}

	int f(int obj) {
	  return Row_A_Value[obj];
	}
	```

Horizontal model

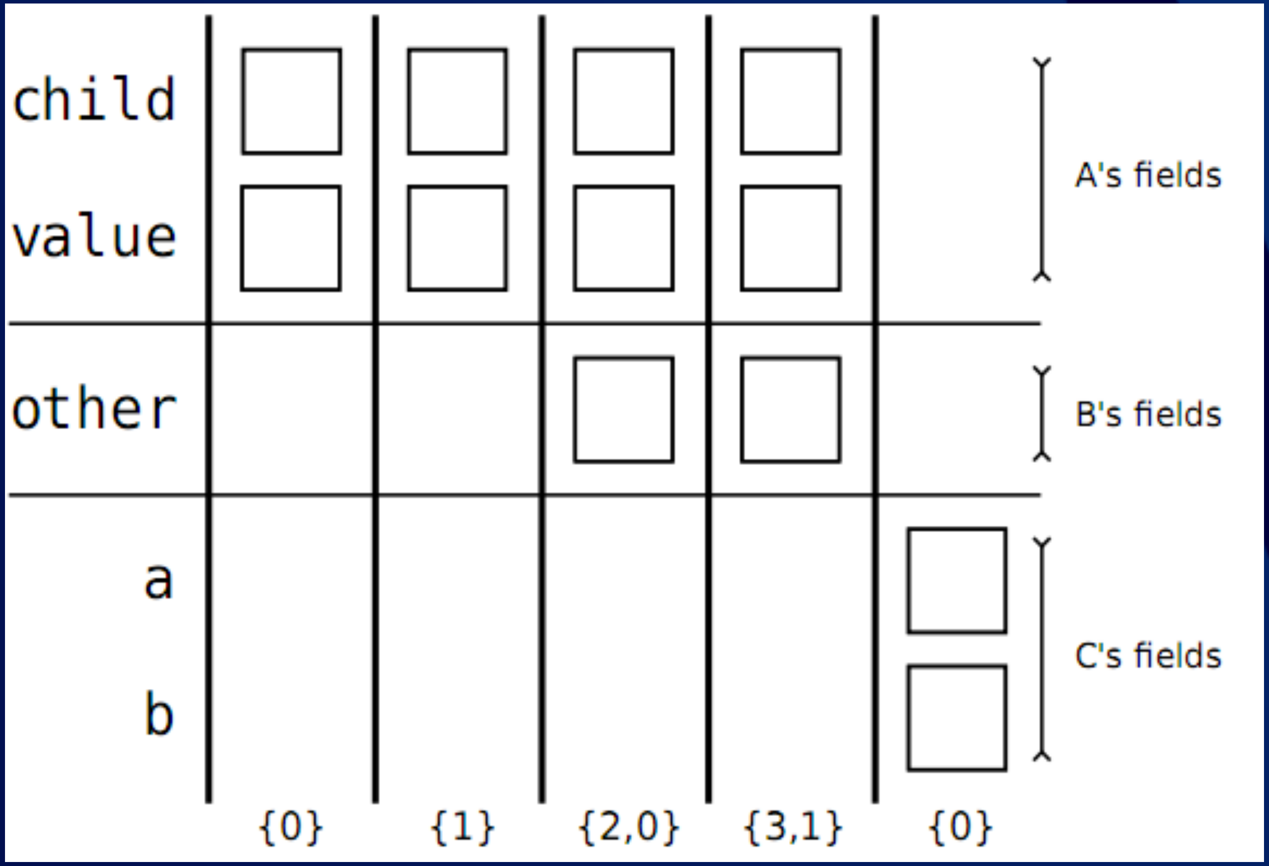
```
method f(obj : A) : int {
  return obj.value;
}

int f(struct A* obj) {
  return obj->value;
}
```

Vertical model

```
method f(obj : A) : int {
  return obj.value;
}

int f(int obj) {
  return Row_A_Value[obj];
}
```

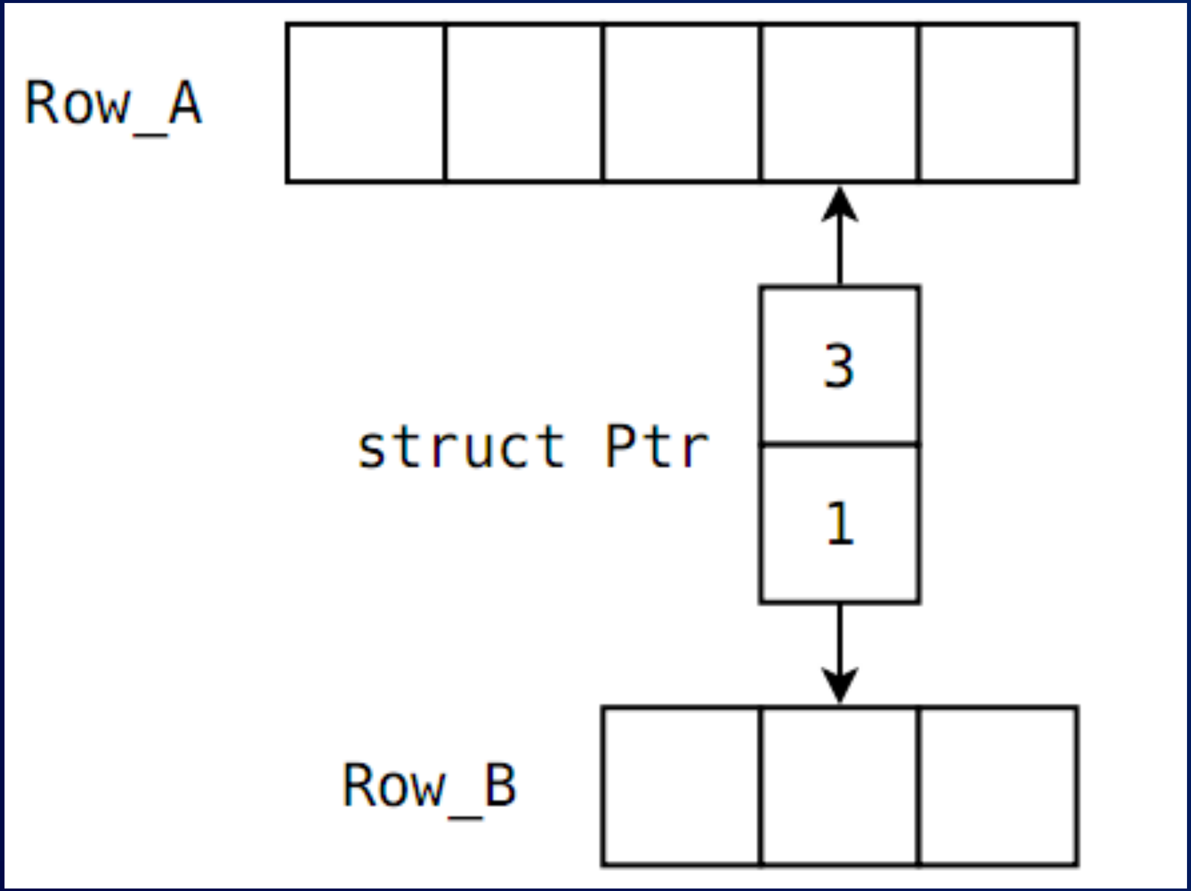
Memory layout of A, B and C
Hybrid compressed model

- No memory wastes
- One row per class field
- Pointers as series of integer offsets
- In our case, a pair of integers



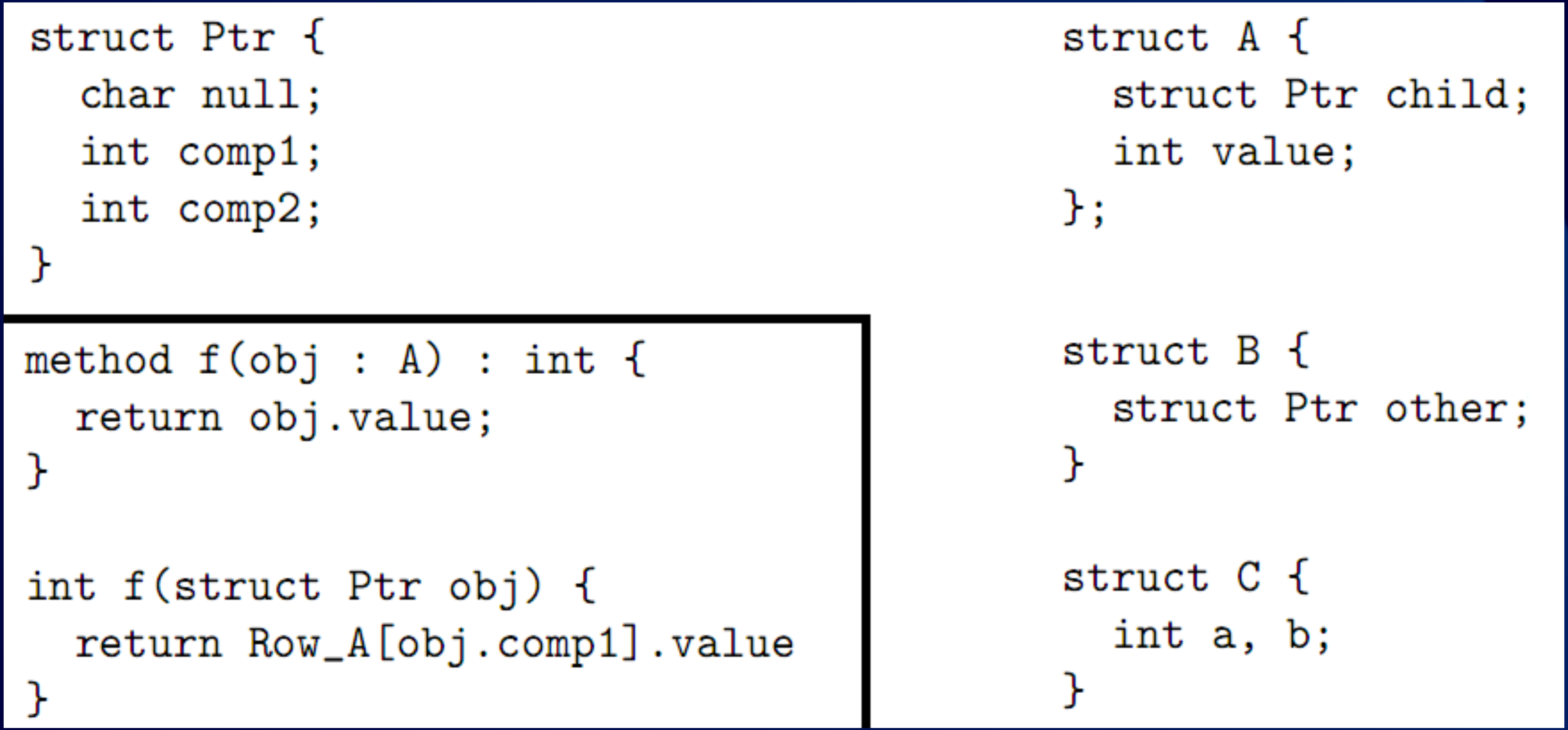
- There is no problem with longer pointers. Bigger pointer means larger data bus, but execution time stays the same

Memory layout of A, B and C
Hybrid compressed model



Memory layout of A, B and C

Hybrid compressed model fields accessing



```
struct Ptr {
  char null;
  int comp1;
  int comp2;
}
```

```
method f(obj : A) : int {
  return obj.value;
}

int f(struct Ptr obj) {
  return Row_A[obj.comp1].value
}
```

```
struct A {
  struct Ptr child;
  int value;
};

struct B {
  struct Ptr other;
}

struct C {
  int a, b;
}
```

TYPEID
Implementing Virgil's 'instanceof' operator

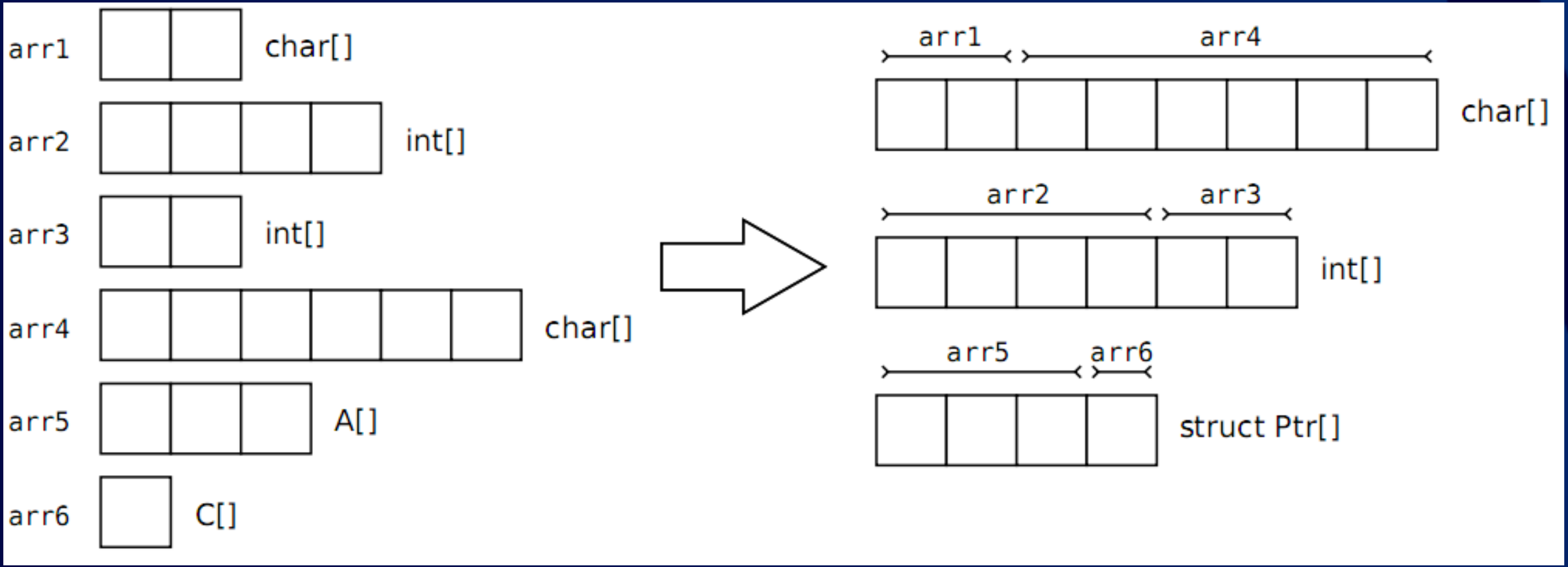
- Statically assigned unique integer to each class by visiting class hierarchy in PRE-ORDER
- Invariants
    - subclasses have greater TYPEID
    - superclasses have smaller TYPEID
- Simple type checking

- **a instanceof Y <=> Y.TYPEID <= a.TYPEID <= max ({X.TYPEID : X subclass of Y})**

Arrays
same problems as with objects

- In Virgil arrays are passed around as references to actual arrays
- Normally, array referencing would be implement as pointers to some global variables
    - We cannot use pointers
- So lets group all same typed arrays into one huge global array
    - We have full information about types during compilation
    - We can implement references to arrays as simple integer indexes (offsets in global arrays)

Arrays



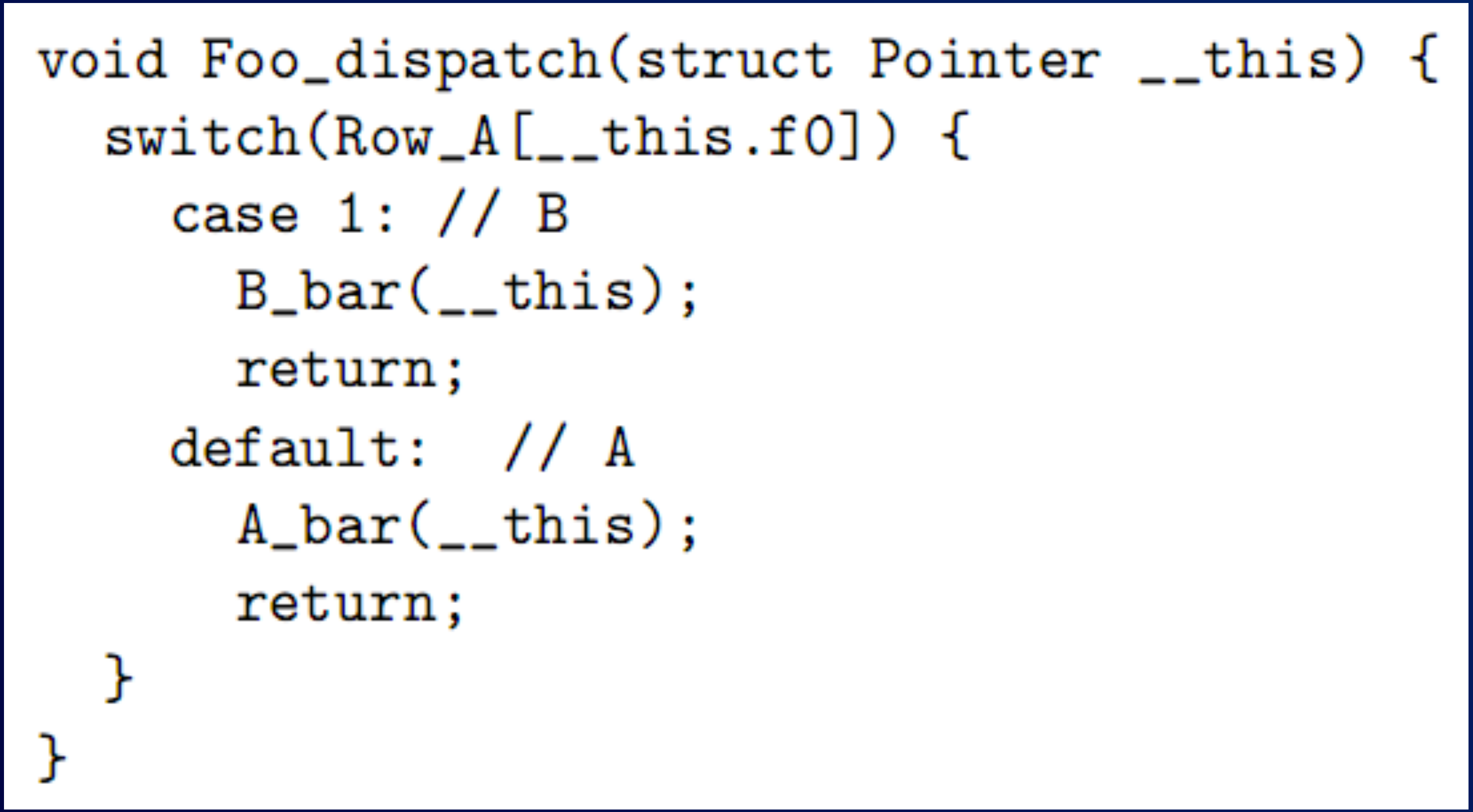
```
method f(arr : int[], x : int) : int {
    return arr[x];
}

int f(struct Array arr, int x) {
    return int_array[arr.start + x];
}
```

```
struct Array {
   char null;
   int start;
   int length;
}
```

Virtual methods

no function pointers -> no virtual method tables

- Instead of VMT we use method dispatcher
- Simple switch-case construction on TYPEID



```
void Foo_dispatch(struct Pointer __this) {
  switch(Row_A[__this.f0]) {
    case 1: // B
      B_bar(__this);
      return;
    default:  // A
      A_bar(__this);
      return;
  }
}
```

Delegates and recursion

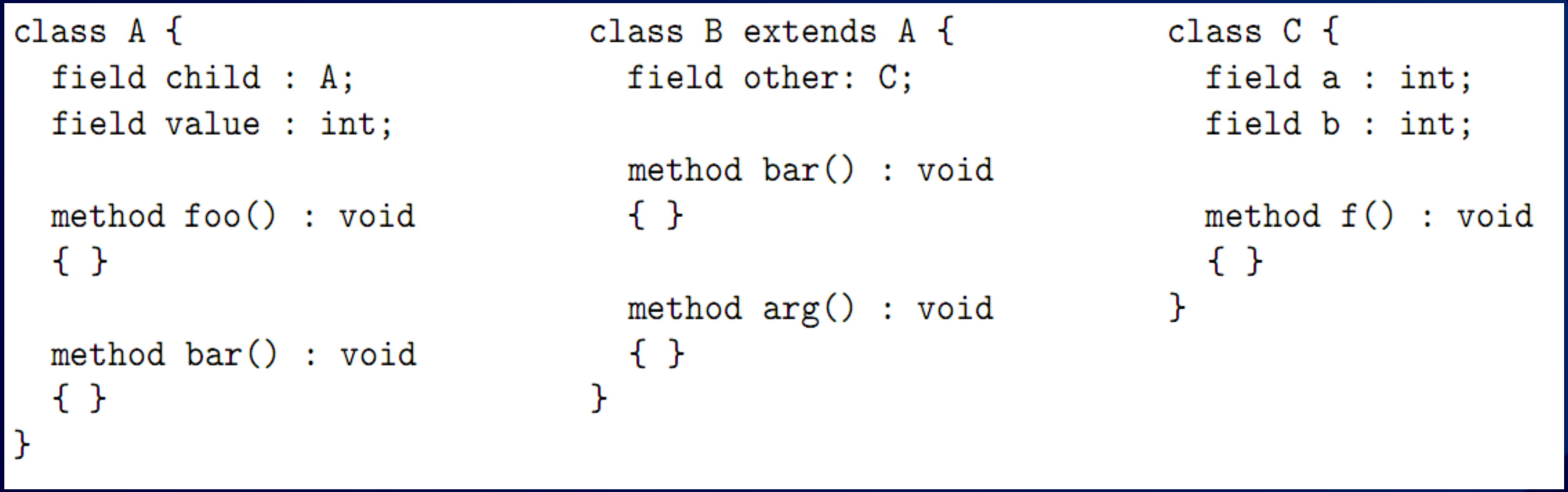
- Not supported but...
- Delegates could be implemented as giant switch-case on all possible functions
- There are many papers about eliminating recursions, but this was beyond the scope of the paper
- We can live without them

Optimizations
virtual methods

- Dispatchers are the biggest overhead in such OO design
- Eliminating dispatchers
    - In obvious places (A::foo, B::arg or C::f)
    - By static analysis

    { A a; a.bar(); }

    we know that we should invoke A::bar not B::bar



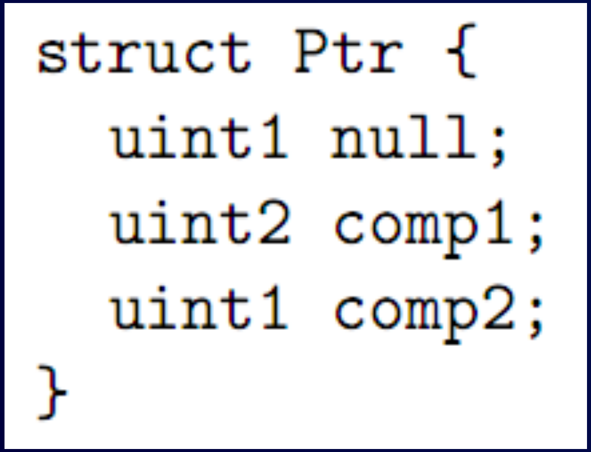
```
class A {
  field child : A;
  field value : int;

  method foo() : void
  { }

  method bar() : void
  { }
}
```

```
class B extends A {
  field other: C;

  method bar() : void
  { }

  method arg() : void
  { }
}
```

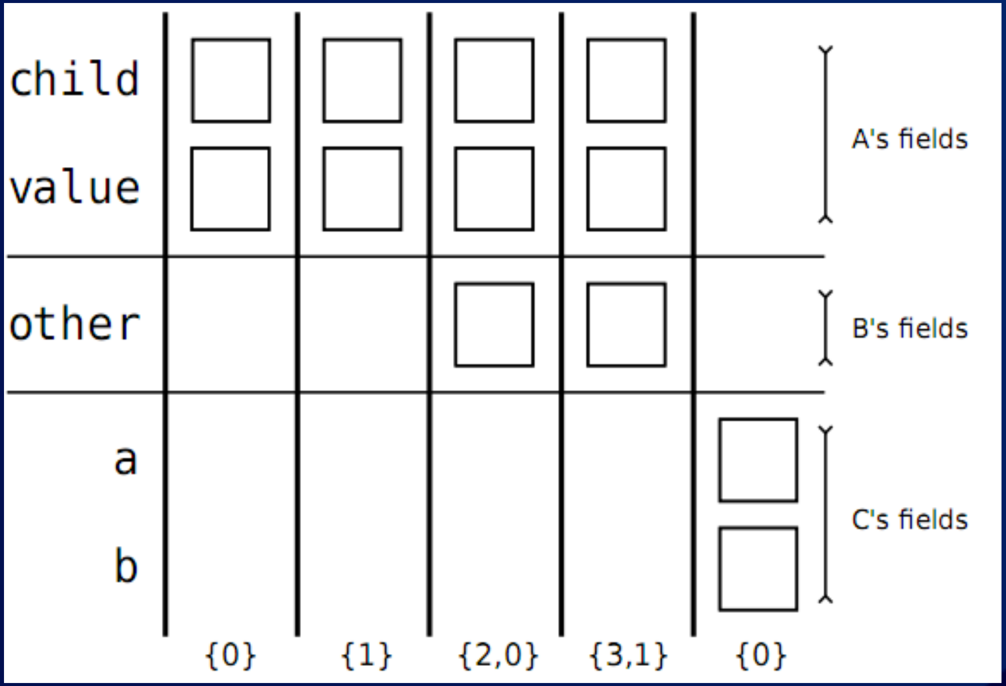
```
class C {
  field a : int;
  field b : int;

  method f() : void
  { }
}
```

Optimizations
bitwidth

- From FPGA point of view we would like to work on smallest possible types
  - smaller bit-length is always better (not like on CPUs)
- In our case pointer size can be reduced
  - widest row for first component is A - 4 columns wide
  - second component for B is 2 columns wide



```
struct Ptr {
  uint1 null;
  uint2 comp1;
  uint1 comp2;
}
```



What have they accomplished?

Benchmarks
experimental results

- We evaluate our results using following benchmarks from well known benchmark suites
- AES - popular modern encryption cipher
- Blowfish - popular modern encryption cipher
- SHA - an implementation of SHA-1 hash function
- Richard's benchmark - simulation of a task-dispatcher component of an operating system
- Benchmarks were translated from existing C-implementations to Virgil
- Platform:
- CPU (xeon) - Quad Core e5430 (2.66GHz, 6MB cache, 32GB ram)
- CPU (atom) - Single Core (1.6GHz, 512KB cache, 1GB ram)
- FPGA (confirmed simulation) - 100MHZ

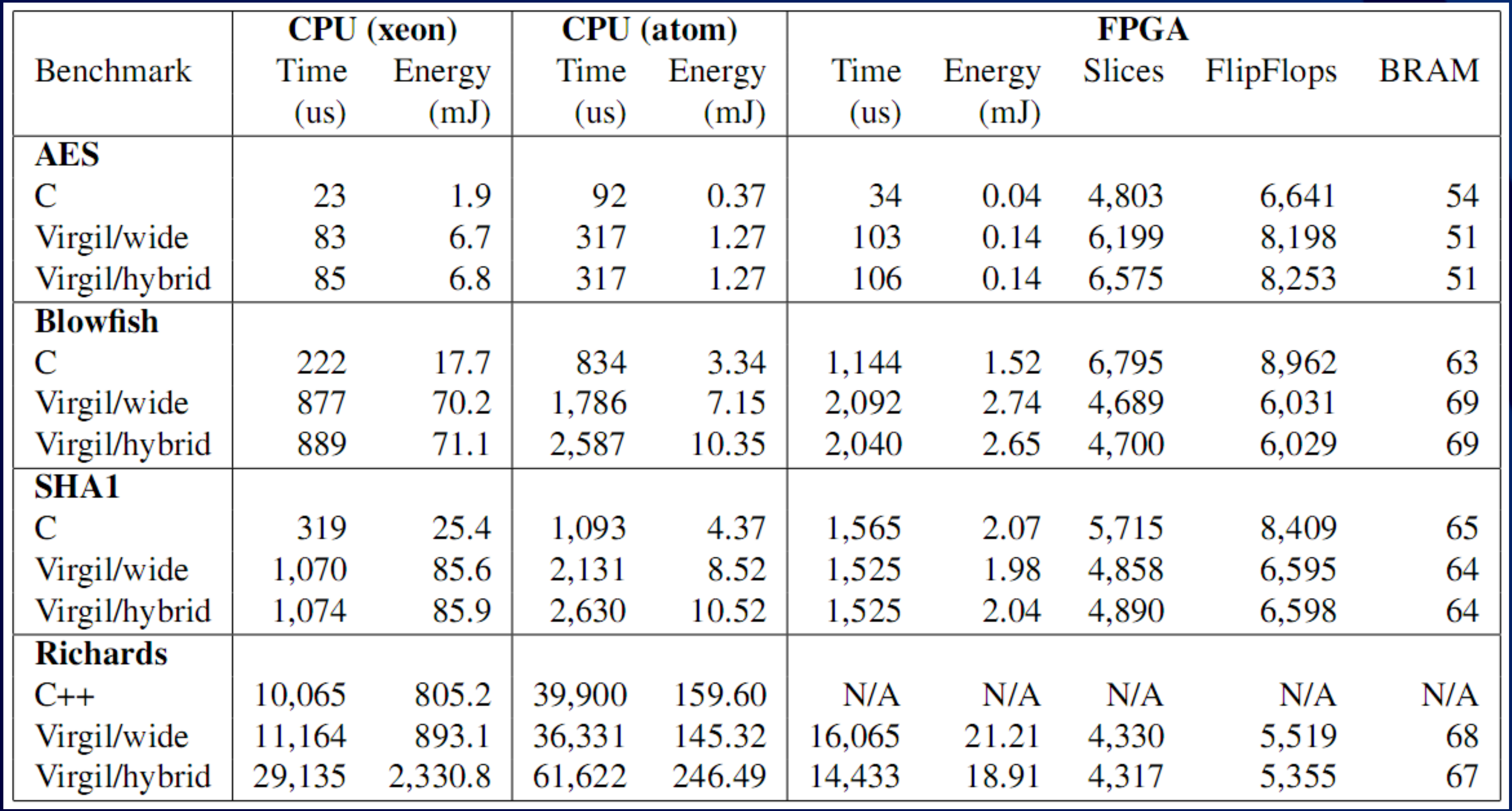
Benchmarks

experimental results

- Each benchmark was executed in following way
  1. orginal C/C++ code compiled with GCC on Ubuntu Linux
     - executed on CPU (xeon)
     - executed on CPU (atom)
  2. orginal C/C++ code compiled using AutoPilot
     - executed/simulated on FPGA
  3. Our virgil code compiled with our compiler to C (both wide and hybrid versions) then compiled with GCC
     - executed on CPU (xeon)
     - executed on CPU (atom)
  4. Our virgil code compiled with our compiler to C (both wide and hybrid versions) then compiled with AutoPilot to FPGA
     - executed/simulated on FPGA
- Our primary interest is comparison of 1. vs 4.

Benchmarks

experimental results



Benchmark	CPU (xeon) Time (us)	CPU (xeon) Energy (mJ)	CPU (atom) Time (us)	CPU (atom) Energy (mJ)	FPGA Time (us)	FPGA Energy (mJ)	FPGA Slices	FPGA FlipFlops	FPGA BRAM
AES									
C	23	1.9	92	0.37	34	0.04	4,803	6,641	54
Virgil/wide	83	6.7	317	1.27	103	0.14	6,199	8,198	51
Virgil/hybrid	85	6.8	317	1.27	106	0.14	6,575	8,253	51
Blowfish									
C	222	17.7	834	3.34	1,144	1.52	6,795	8,962	63
Virgil/wide	877	70.2	1,786	7.15	2,092	2.74	4,689	6,031	69
Virgil/hybrid	889	71.1	2,587	10.35	2,040	2.65	4,700	6,029	69
SHA1									
C	319	25.4	1,093	4.37	1,565	2.07	5,715	8,409	65
Virgil/wide	1,070	85.6	2,131	8.52	1,525	1.98	4,858	6,595	64
Virgil/hybrid	1,074	85.9	2,630	10.52	1,525	2.04	4,890	6,598	64
Richards									
C++	10,065	805.2	39,900	159.60	N/A	N/A	N/A	N/A	N/A
Virgil/wide	11,164	893.1	36,331	145.32	16,065	21.21	4,330	5,519	68
Virgil/hybrid	29,135	2,330.8	61,622	246.49	14,433	18.91	4,317	5,355	67

What's next?

- We clearly see that results are very promising
  - Similiar performances to latest Intel's high-end CPU with small and relatively cheap units using 40 times less energy
- FPGAs are getting more and more attention
- There is undergoing work with for example
  - JVM processors (JOP)
  - Neural networks (think of this massive parallelism)