



Synthesis and Optimization of Complex Memory Operations

Tutorial @ FPL Conference 2017 – Ghent – Belgium

Christian Pilato

Università della Svizzera italiana (USI)
Facoltà di Informatica
christian.pilato@usi.ch

- ❑ Algorithms operate on data and data must be stored somewhere
- ❑ Memories are responsible for **70-90% of total cost**
 - ▶ In FPGA:
 - Small/frequently-accessed sets of data in distributed registers (low latency, high resource cost)
 - Medium sets of data in BRAMs (limited number of ports, limited number of resources)
 - Large sets of data in the off-chip memory (e.g., DRAM)

How to efficiently implement all memory operations?

❑ Software:

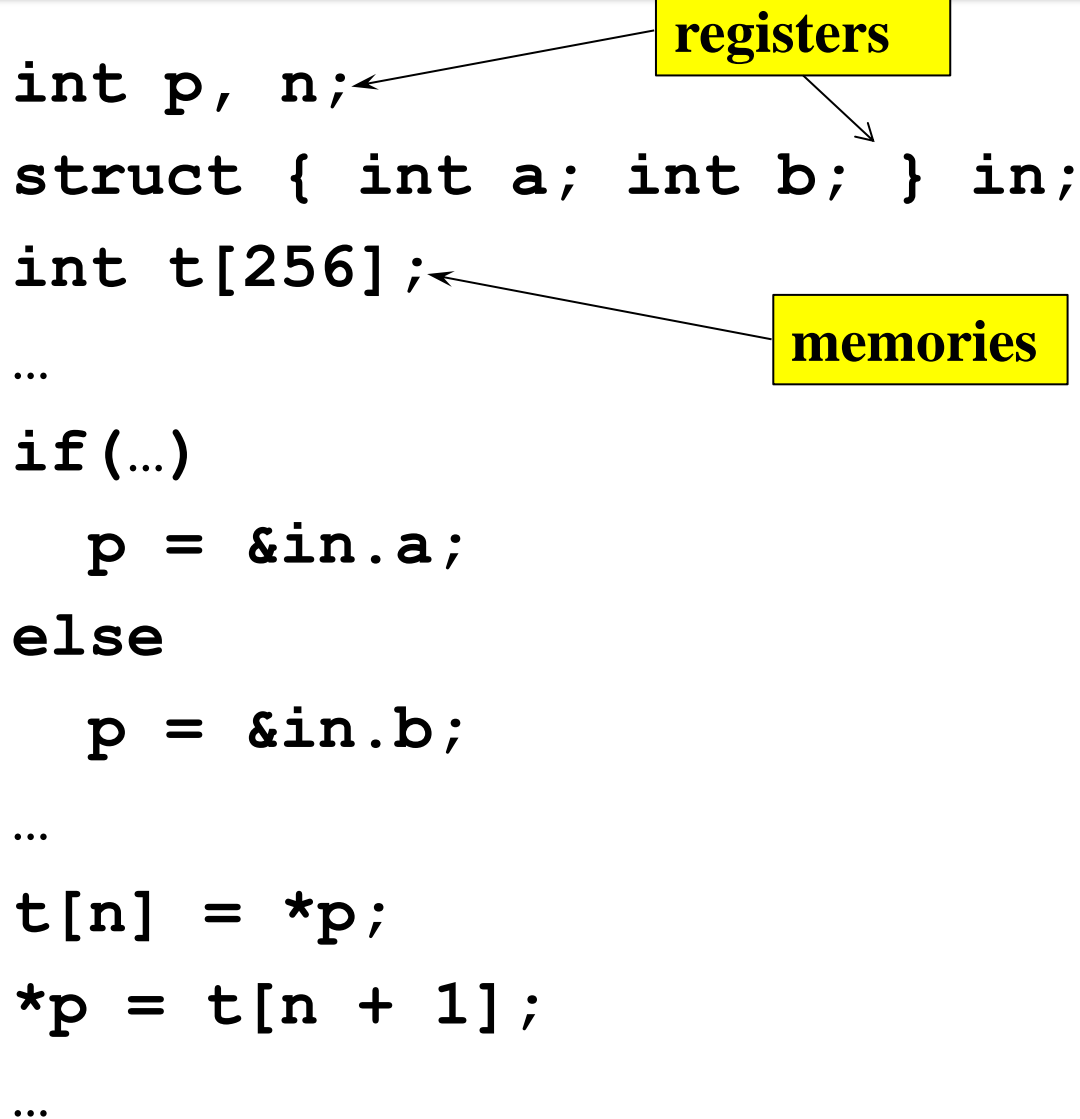
- ▶ *Stack* for storing data determined statically
- ▶ *Heap* for dynamically allocated data
- ▶ *Cache hierarchies* for hiding the latency
- ▶ *Sequential memory operations*

❑ Hardware

- ▶ *Heterogeneous and distributed memories*
- ▶ *Abundant hardware parallelism*
- ▶ *No “flexibility” during the execution*

Example

4



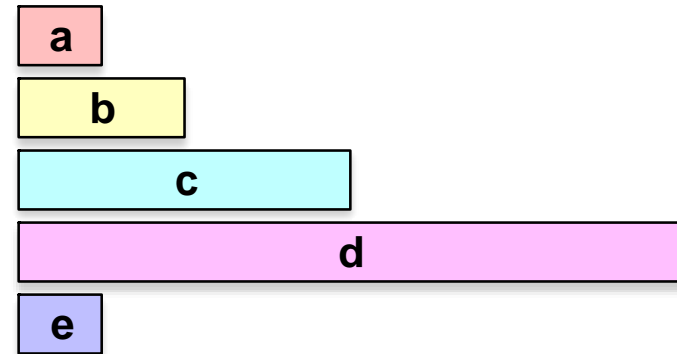
The diagram illustrates the mapping of code elements to hardware resources. A yellow box labeled 'registers' has two arrows pointing to the variables 'n' and 'in' in the code. Another yellow box labeled 'memories' has an arrow pointing to the array 't' in the code.

```
int p, n;
struct { int a; int b; } in;
int t[256];
...
if (...)
    p = &in.a;
else
    p = &in.b;
...
t[n] = *p;
*p = t[n + 1];
...
```

- ❑ **Arrays of primitive data types** easily implemented with HDL templates (supported by any synthesis tool)
- ❑ There are three types of memory operations to be supported:
 - ▶ Accesses to **complex data structures** (e.g., arrays of complex structures)
 - ▶ **Parameter passing** in function calls
 - ▶ Operation on **pointers** (dynamic resolution, pointer arithmetic, dynamic memory allocation)

It is relatively easy to implement memory operations, it is NOT easy to implement them efficiently

```
typedef struct
{
    char a;          // 1 byte
    short b;         // 2 bytes
    long c;          // 4 bytes
    long long d;     // 8 bytes
    char e;          // 1 byte
} mystruct; // tot: 16 bytes
```

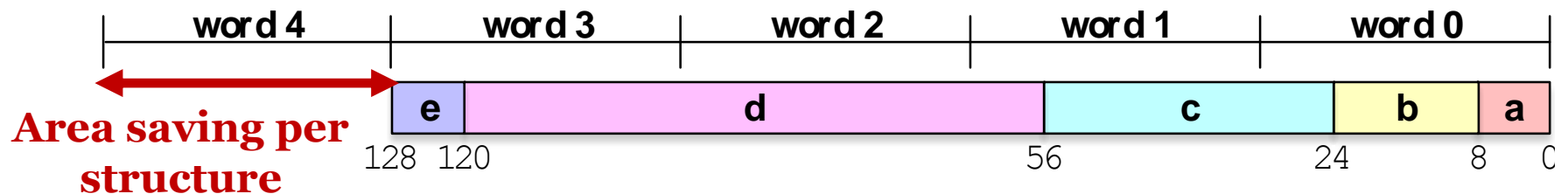
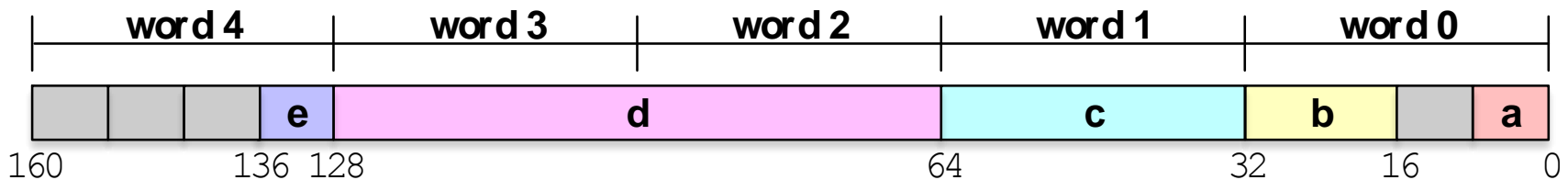


- ❑ How to *efficiently* store an array of **mystruct** structures?
- ❑ How to *efficiently* implement memory operations?

Reduce the memory footprint without compromising the execution latency

❑ Software annotations to create compact representations

```
typedef struct
{
    char a;          // 1 byte
    short b;         // 2 bytes
    long c;          // 4 bytes
    long long d;     // 8 bytes
    char e;          // 1 byte
} __attribute__((packed)) mystruct;
```



❑ Alternative solutions:

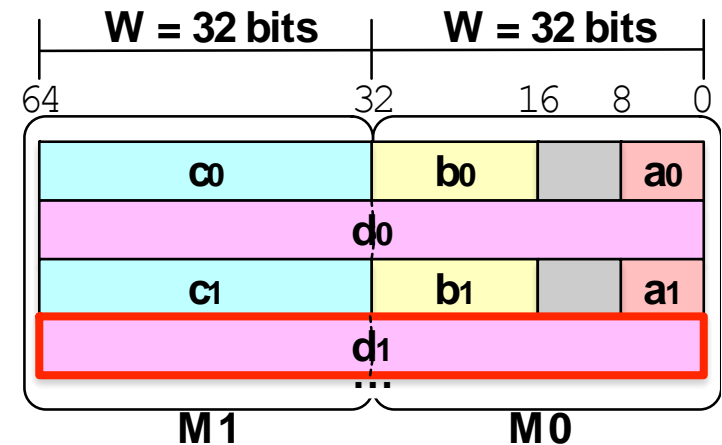
- ▶ Vivado HLS decomposes the *array of structures* into *multiple arrays of single fields*
- ▶ LegUp uses *subarrays* as large as the maximum field (support only for aligned accesses)

❑ We use a **single memory template** to implement both aligned and unaligned accesses

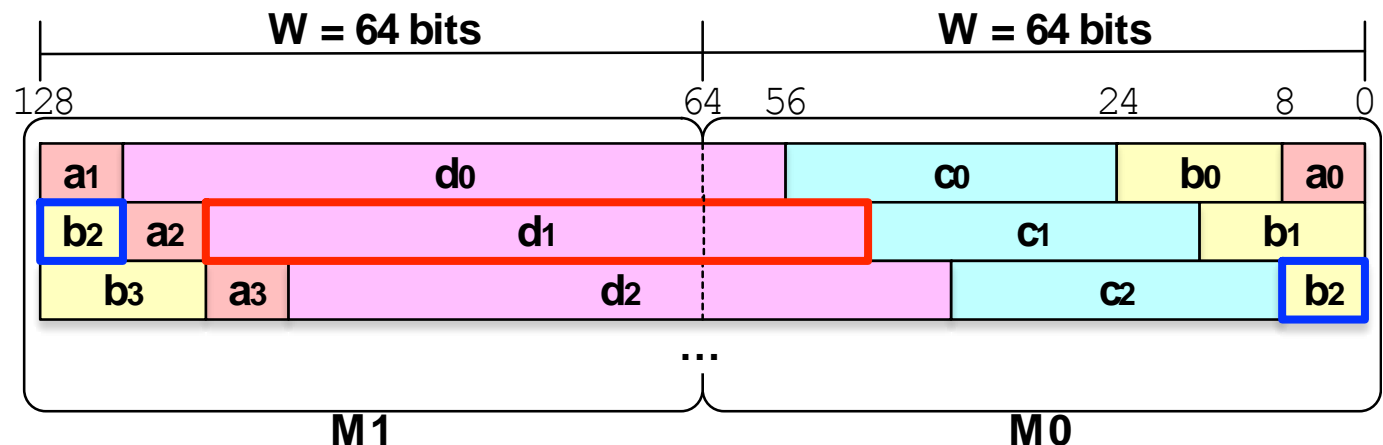
- ▶ *Portable and easy to maintain*
- ▶ *Simple logic* to convert datapath requests into actual memory operations

- Two arrays with bitwidth based of the type of accesses
 - At most two (parallel) memory ops for each field

Aligned structures
($W = SMAX/2$)

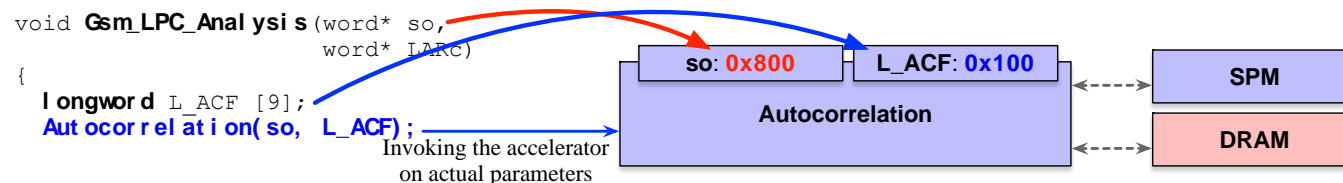


Unaligned structures
($W = SMAX$)



- POLITECNICO DI MILANO**

- ❑ Hardware modules are created hierarchically based on the *call graph*
 - ▶ *Software*: parameters on the stack
 - ▶ *Hardware*: actual values provided at the input ports



When to insert registers to avoid long critical paths?

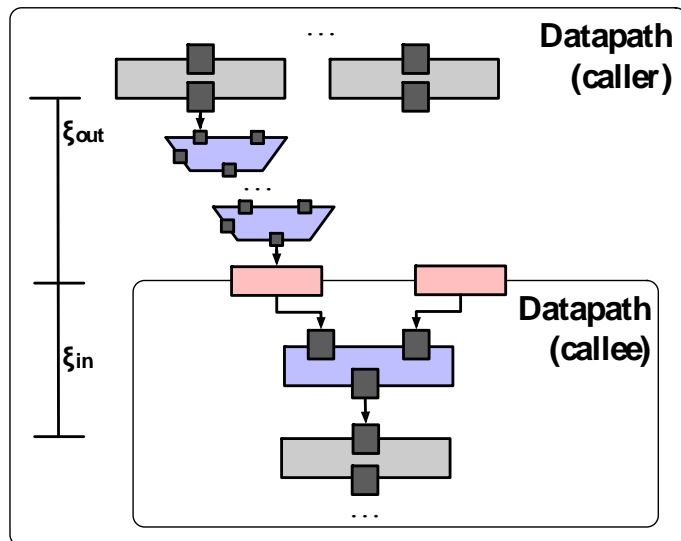
- ❑ We introduce input registers when

$$(\xi_{\text{out}} + \xi_{\text{in}}) > T * \beta$$

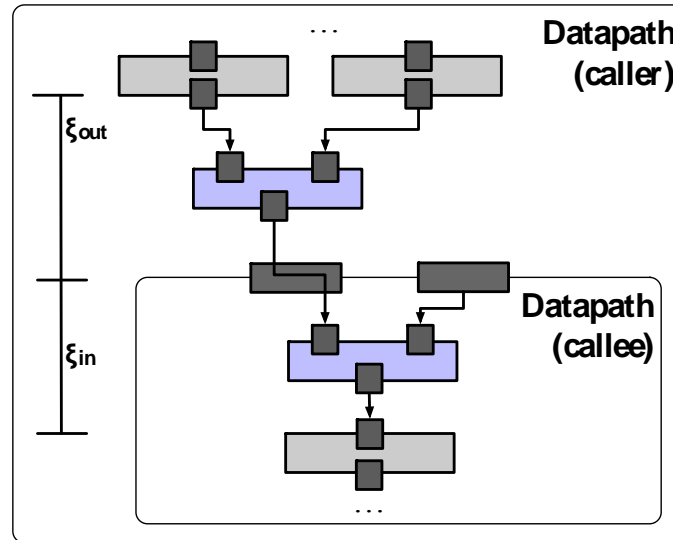
Some Examples of Input Registering

12

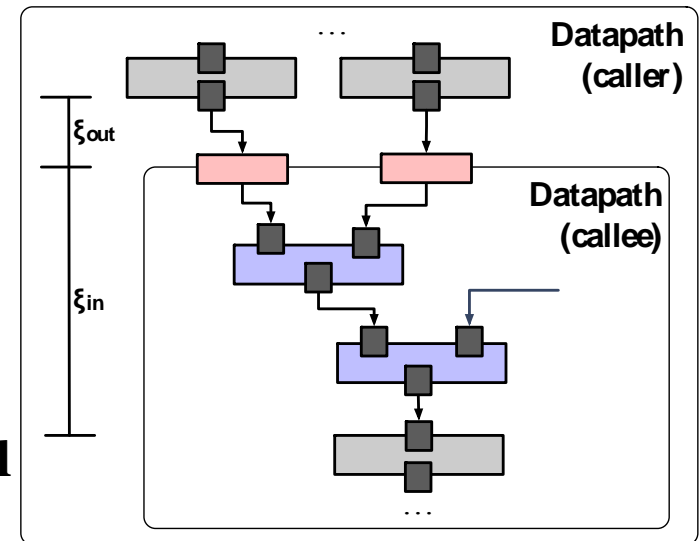
Long external logic (ξ_{out})



Direct connection



Long internal logic (ξ_{in})



- ❑ We require a **configurable memory space** to store the created data structures
 - ▶ *Memory management* is usually intrinsic within the algorithm

- ❑ We leverage **Memmgr**
 - <https://github.com/eliben/code-for-blog/tree/master/2008/memmgr>
 - ▶ library for dynamic memory allocation to a configurable space with a synthesizable **malloc** function
 - ▶ *pre-allocated memory space* with a default size of 384 KB
 - ▶ memory space either in a local memory or in DRAM
 - ▶ memory component to manage *pointer-based requests*

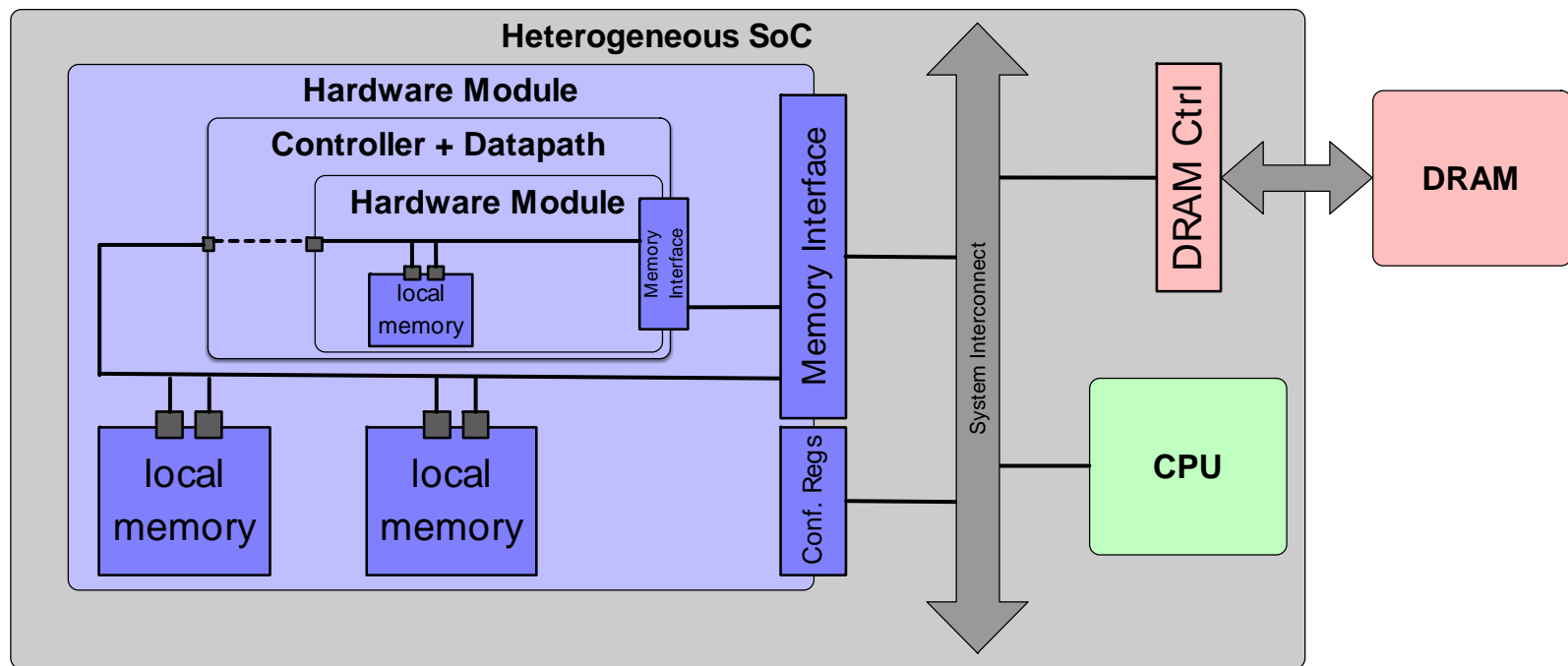
- ❑ Two conflicting goals:
 - ▶ Maintain **flexibility** as in software (operations on pointers, dynamic resolution, etc.)
 - ▶ Increase **efficiency** as in hardware (exploit hardware parallelism whenever possible)

Definition:

Points-to set: set of data structures that can be potentially accessed by the pointer used in a memory operation

- ❑ This information is obtained during compilation by means of (sophisticated) **alias analysis**

- ❑ **Direct connections** to datapath operators if the operation is completely defined
- ❑ **Internal memory bus** to connect all memory components potentially accessed by unresolved memory operations (*points-to set*)



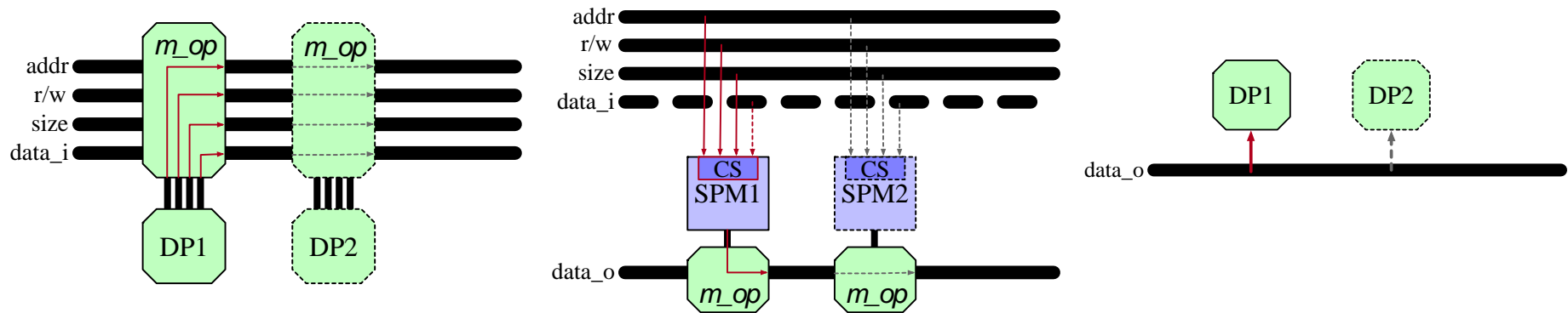
- ❑ **Memory analysis phase** to define an address for any
 - ▶ global scalar/aggregate variables,
 - ▶ local aggregate variables
 - ▶ local scalar variables used as argument of operator &

- ❑ Pointers are stored in standard registers
 - ▶ They can be considered as standard variables

- ❑ Load and store memory operations can be implemented as standard datapath operations
 - ▶ Connected to dedicated memory resources (memory controllers)

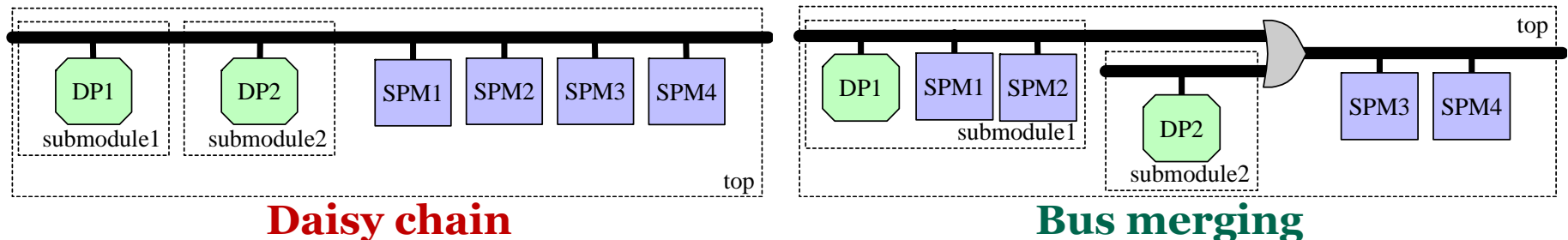
- | | |
|--|--------------------------------------|
| tag | offset |
| n. of allocated objs
←────────────────→ | maximum offset
←────────────────→ |
| chip select | offset in RAM |

□ Very powerful memory microarchitecture

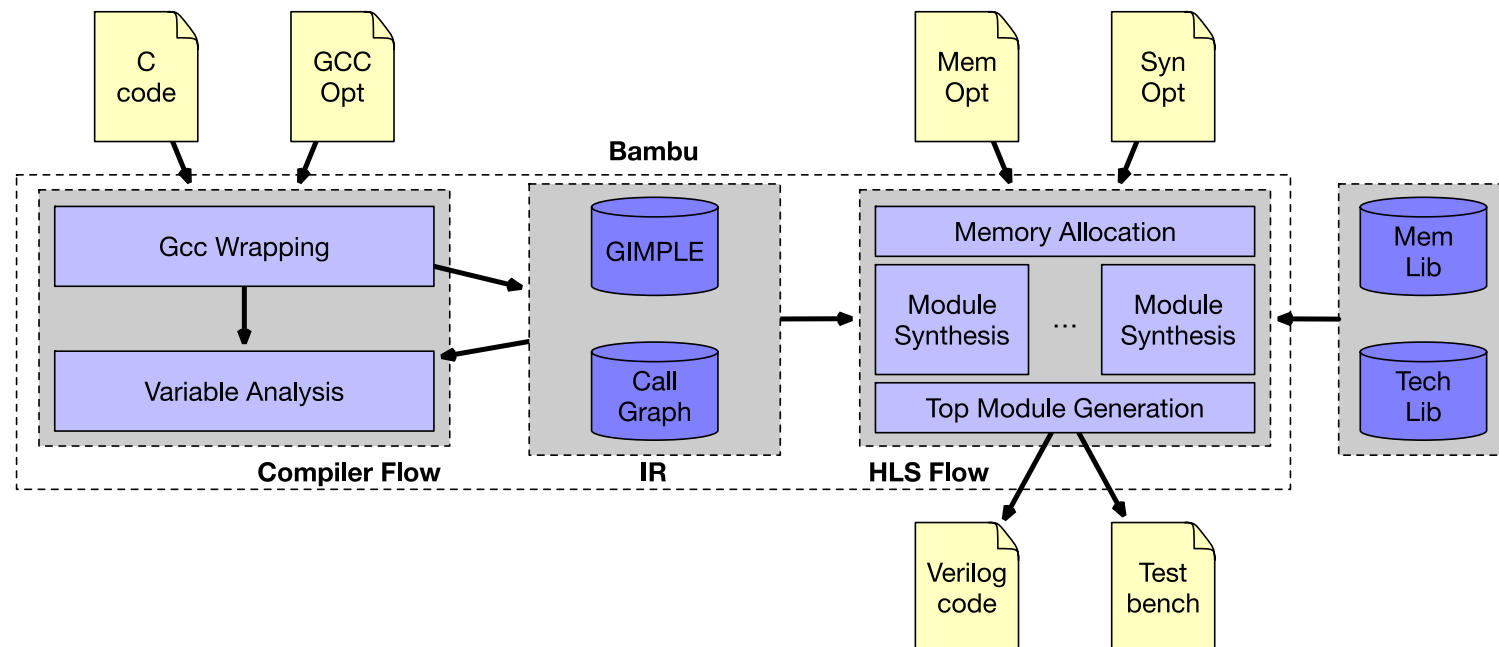


□ Possibility of creating long combinational paths with many memory components connected in chain

- Dominator-based analysis and bus merging (OR gates) for localizing the memories as much as possible



- ❑ **Automatic generation and optimization** fully implemented in Bambu
 - ▶ GCC for extracting **memory-related information**
 - ▶ Extensive set of **memory parameters** to explore many alternative configurations



❑ Reference/Naïve approach

- ▶ all data structures with standard memory components, instantiated in the top module and on the internal bus (**maximum flexibility**)

❑ Privatization

- ▶ flow-sensitive pointer analysis to reduce the points-to set
- ▶ resolved memory operations directly connected to the memories

❑ Creation and duplication of **read-only memories**

❑ Conversion into **distributed memories**

- ▶ configurable threshold
- ▶ higher threshold for read-only memories (simpler)

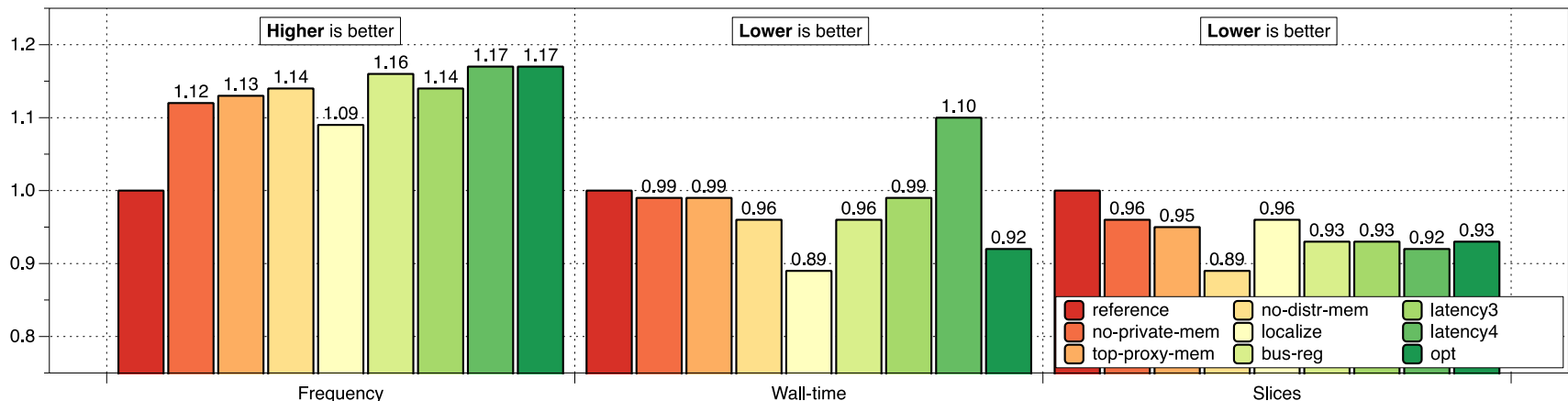
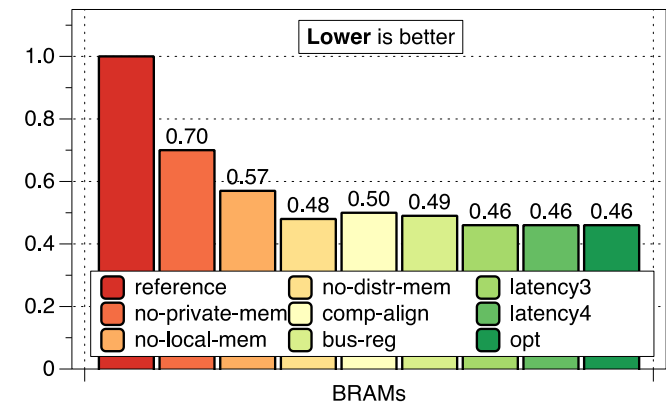
❑ **Memory registering**

- ▶ potential timing violations with complex memory operations and high target frequency
- ▶ registers between datapath resources and memory ports

❑ **Localization**

- ▶ dominator-based analysis to reduce long combinational paths

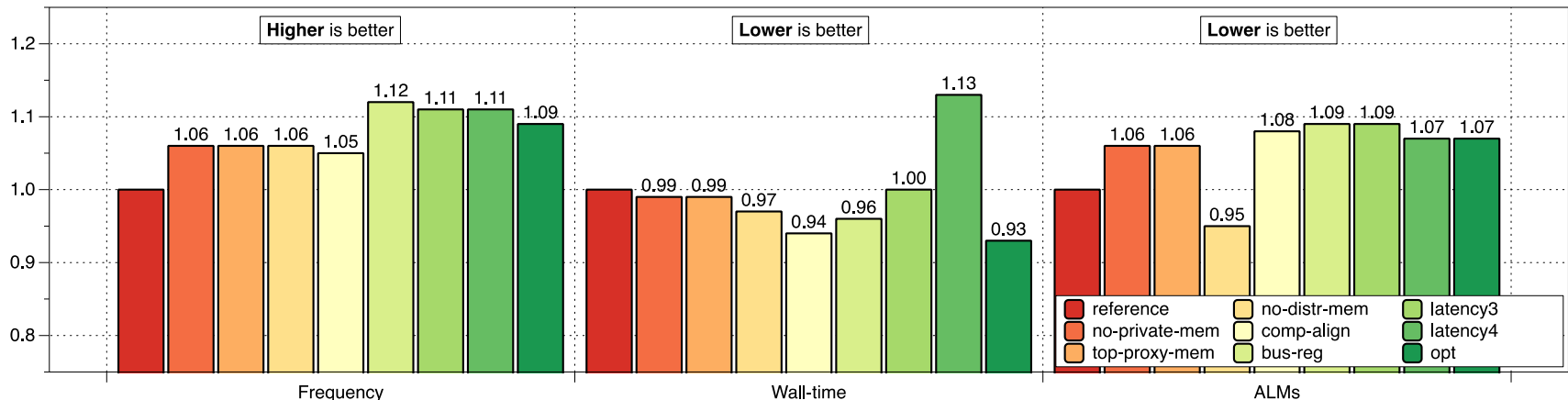
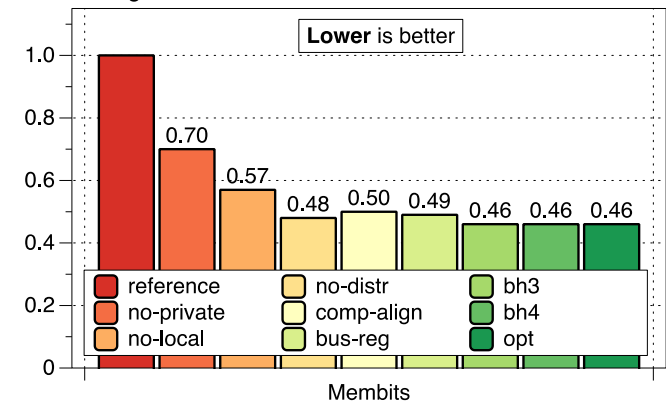
- ❑ Memory components can reach 312 MHz (XC7Zo20) and 608 MHz (XC7VX690T)
- ❑ Selected applications from a recent survey on HLS tools (Nane et al. TCAD 2016)
 - ▶ Frequency of 400MHz
 - ▶ Up to 54% of memory reduction



❑ Memory components can reach 600 MHz (Stratix-V)

❑ Selected applications from a recent survey on HLS tools (Nane et al. TCAD 2016)

- ▶ Frequency of 400MHz
- ▶ Up to 54% of memory reduction



Questions?