HeteroRefactor: Refactoring for Heterogeneous Computing with FPGA

Jason Lau*, Aishwarya Sivaraman*, Qian Zhang*,
Muhammad Ali Gulzar, Jason Cong, Miryung Kim
University of California, Los Angeles
**Equal co-first authors in alphabetical order*

VLSI architecture, synthesis & technology

Software Evolution & Analysis Laboratory

The 42nd International Conference on Software Engineering
ICSE 2020

HeteroRefactor: Refactoring for Heterogeneous Computing with FPGA






Jason Lau

Aishwarya Sivaraman

Qian Zhang

Muhammad Ali Gulzar

Jason Cong

Miryung Kim

vast laboratory VLSI architecture, synthesis & technology

SEAL Software Evolution & Analysis Laboratory

The 42nd International Conference on Software Engineering

ICSE 2020

FPGA*-based Acceleration

Fast

Efficient

* FPGA: Field Programmable Gate Array

FPGA*-based Acceleration

Fast

Efficient

Effort

* Field Programmable Gate Array

Evolution of Programming Model

- typeless.
- registers.
- instructions.
- goto-style control.

```
module vecdot(a, b, c, clk, rst);
    input [67:0] a, b;
    output [16:0] c;
    reg [5:0] s; reg [16:0] prod [0:7]; ...
always @(posedge clk or posedge rst)
if (!rst) begin
    if (s == 6'b00001)
        prod[0] = a[..] * b[..]; prod[1] =...
        s = 6'b00010;
    else if (s == 6'b00010)
        reg1 = prod[0] + prod[1] + prod[2];
        s = 6'b00100; // goto L00100;
    else if (s == 6'b00100)
        reg1 = reg1 + prod[3] + prod[4];
        s = 6'b01000;
    else ... ;
...
endmodule
```

Verilog HDL*

* HDL: Hardware Description Language

Evolution of Programming Model

- typed.
- auto schedule.
- auto resource.
- auto optimization.

```
fpga_float<8,15> vecdot(
         fpga_float<8,15> a[],
         fpga_float<8,15> b[],
         fpga_int<31> n) {
     for (fpga_int<31> i = 0; i < n; i++)
          sum += a[i] * b[i];
     return sum;
}
```

Merlin HLS*, etc.

* HLS: High-Level Synthesis

Something is missing...

bit-width.

```
fpga_float<8,15> vecdot(
        fpga_float<8,15> a[],
        fpga_float<8,15> b[],
        fpga_int<31> n) {
    for (fpga_int<31> i = 0; i < n; i++)
bitwidth = 31 sum += a[i] * b[i];
    return sum;
}
```

waste scarce memory!

FPGA memory: < 100 MB

Merlin HLS*, etc.

* HLS: High-Level Synthesis

Something is missing...

bit-width.

floating-point precision.

exponent 8 bits
fraction 15 bits

```
fpga_float<8,15> vecdot(
          fpga_float<8,15> a[],
          fpga_float<8,15> b[],
          fpga_int<31> n) {
     for (fpga_int<31> i = 0; i < n; i++)
          sum += a[i] * b[i];
     return sum;
}
```

precision?
memory?

Merlin HLS*, etc.

* HLS: High-Level Synthesis

Something is missing...

4 errors in 14 lines of code

- bit-width.
- floating-point precision.
- recursive data structure.
- nested pointers

```
struct Node {
    Node *left, *right;
    int val; };

void init(Node **root) {
    *root = (Node *)malloc(sizeof(Node)); }

void insert(Node **root, int *arr);
void delete_tree(Node *root) {
// … free(root); }
void traverse(Node *curr) {
    if (curr == NULL) return;
    int ret = visit(curr->val);

    traverse(curr->left);
    traverse(curr->right);
}
```

Merlin HLS*, etc.

* HLS: High-Level Synthesis

Something is missing...

4 errors in 14 lines of code

- bit-width.
- floating-point precision.
- recursive data structure.
- nested pointers
- dynamic mem mgmt

```
struct Node {
      Node *left, *right;
      int val; };

void init(Node **root) {
      *root = (Node *)malloc(sizeof(Node)); }

void insert(Node **root, int *arr);
void delete_tree(Node *root) {
// … free(root); }
void traverse(Node *curr) {
      if (curr == NULL) return;
      int ret = visit(curr->val);

      traverse(curr->left);
      traverse(curr->right);
}
```

preallocated size?

Merlin HLS*, etc.

* HLS: High-Level Synthesis

Something is missing...

4 errors in 14 lines of code

- bit-width.
- floating-point precision.
- recursive data structure.

nested pointers

dynamic mem mgmt

pointer operations

```
struct Node {
     Node *left, *right;
     int val; };

void init(Node **root) {
     *root = (Node *)malloc(sizeof(Node)); }

void insert(Node **root, int *arr);
void delete_tree(Node *root) {
// … free(root); }
void traverse(Node *curr) {
     if (curr == NULL) return;
     int ret = visit(curr->val);

     traverse(curr->left);
     traverse(curr->right);
}
```

preallocated size?

Merlin HLS*, etc.

* HLS: High-Level Synthesis

Something is missing...

4 errors in 14 lines of code

- bit-width.
- floating-point precision.
- recursive data structure.

nested pointers

dynamic mem mgmt

pointer operations

recursion functions

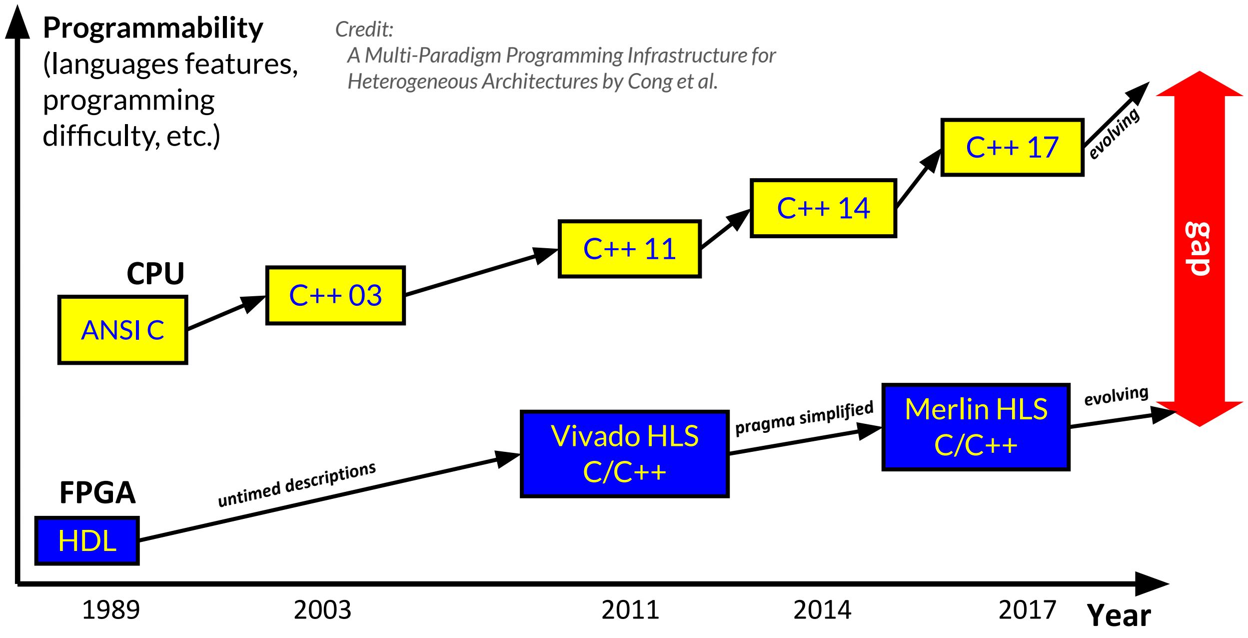
```
struct Node {
    Node *left, *right;
    int val; };

void init(Node **root) {
    *root = (Node *)malloc(sizeof(Node)); }

void insert(Node **root, int *arr);
void delete_tree(Node *root) {
// … free(root); }
void traverse(Node *curr) {
    if (curr == NULL) return;
    int ret = visit(curr->val);

    traverse(curr->left);
    traverse(curr->right);
}
```

preallocated size?

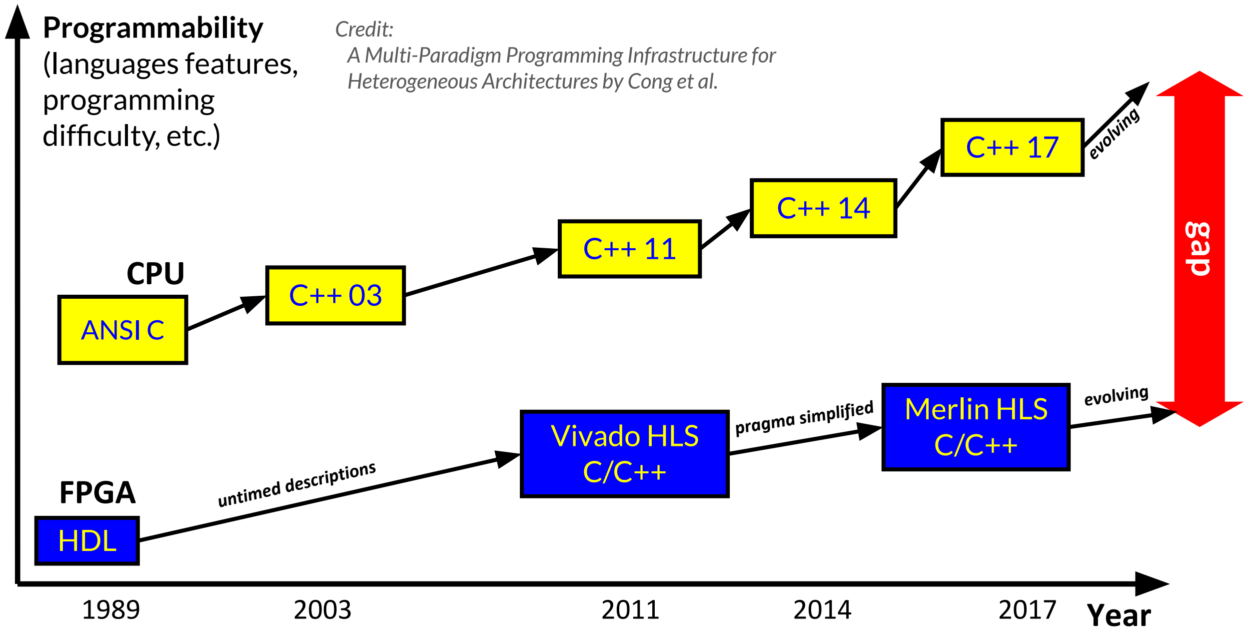
Merlin HLS*, etc.

* HLS: High-Level Synthesis

Evolution of Programming Model



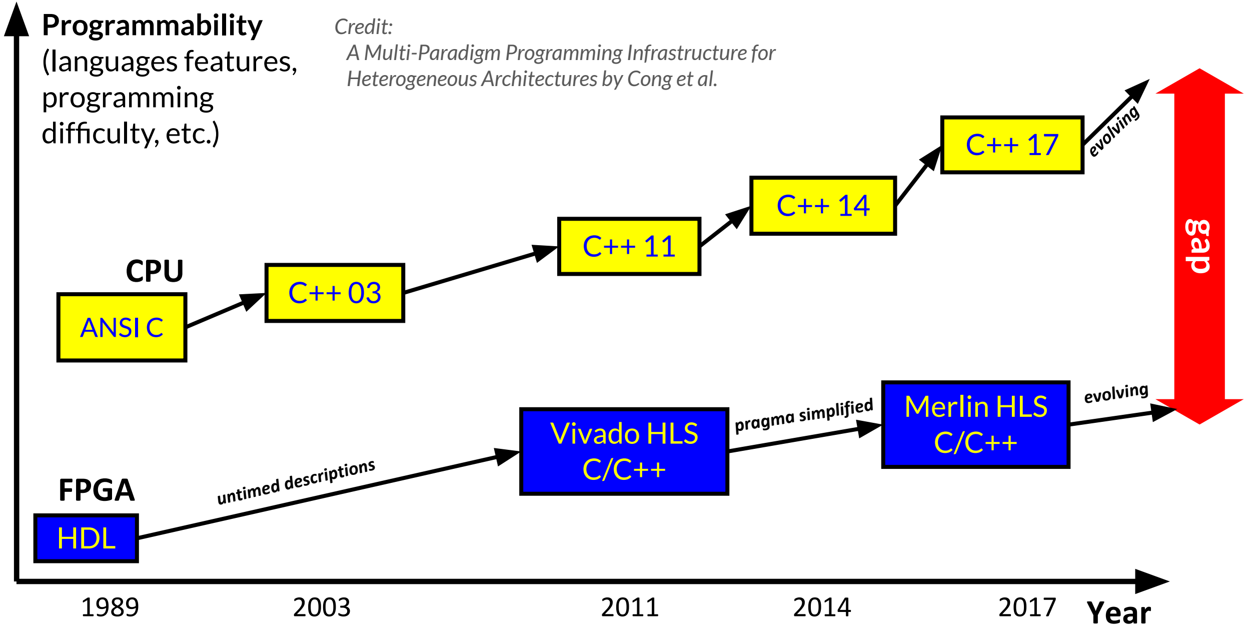
Evolution of Programming Model



significant human effort.

& error-prone.

Evolution of Programming Model



significant human effort.

& error-prone.

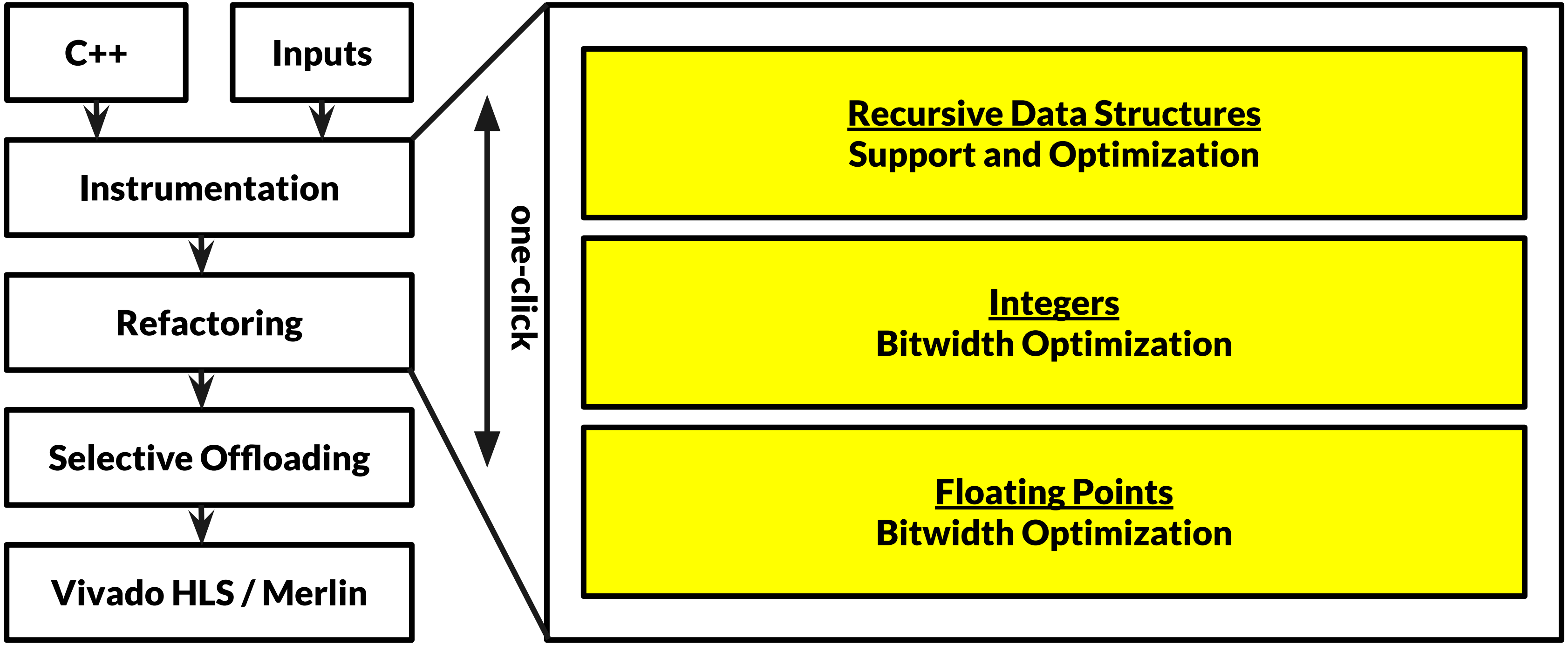
waste scarce memory

I want it to run!

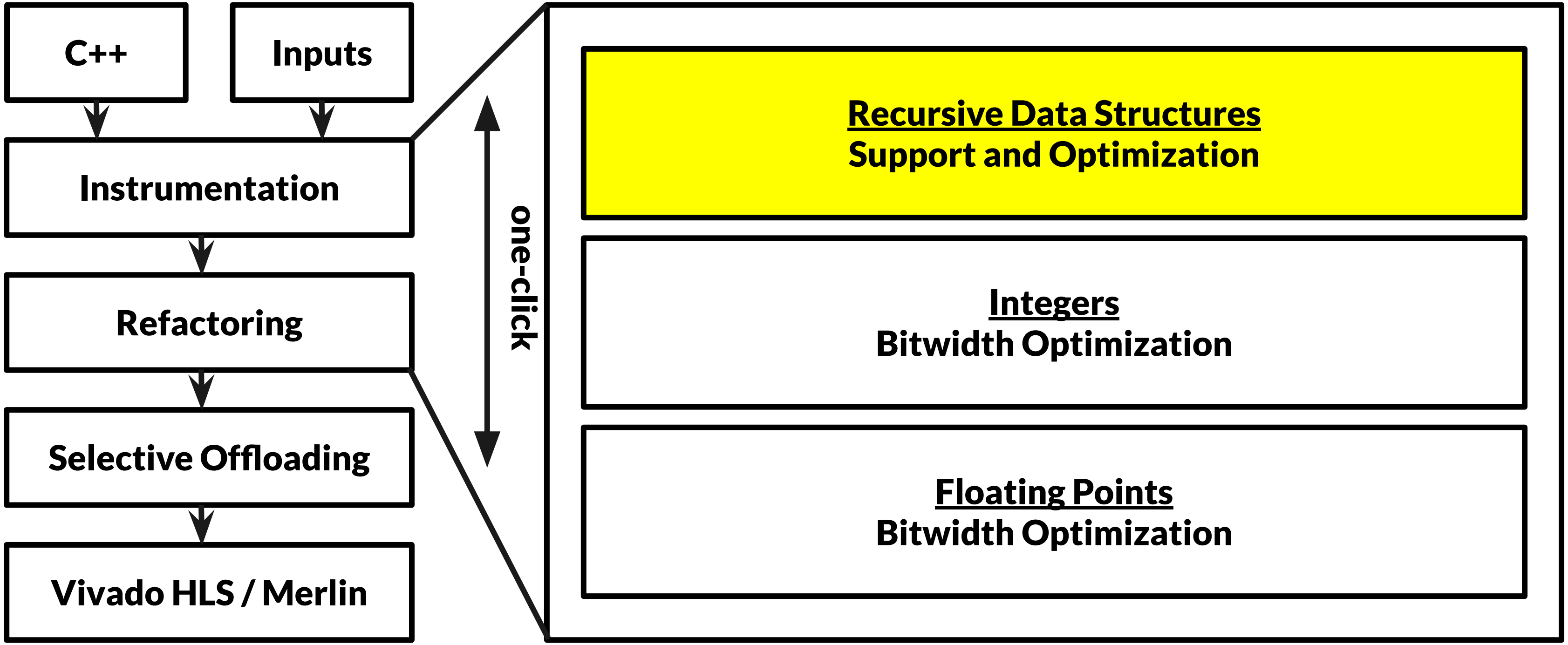
I want it to run efficiently!

Automation!

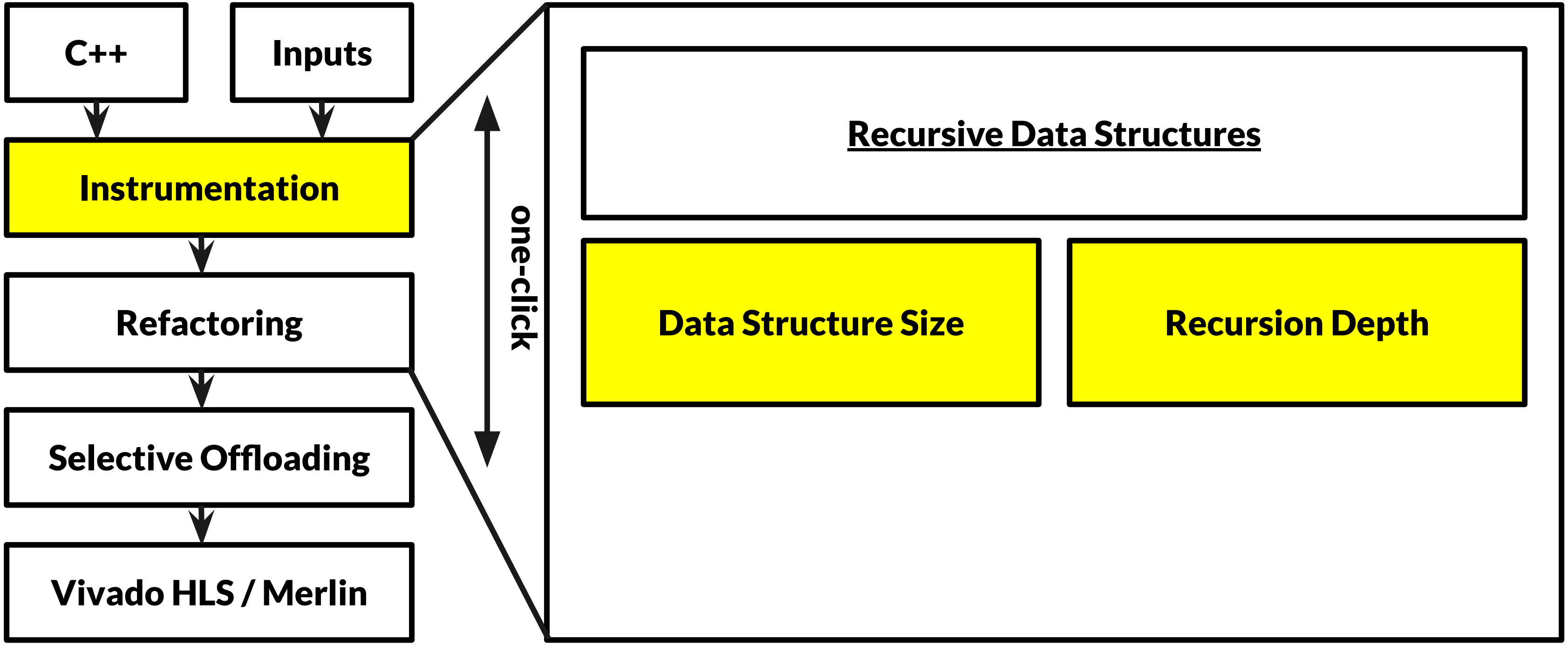
HeteroRefactor



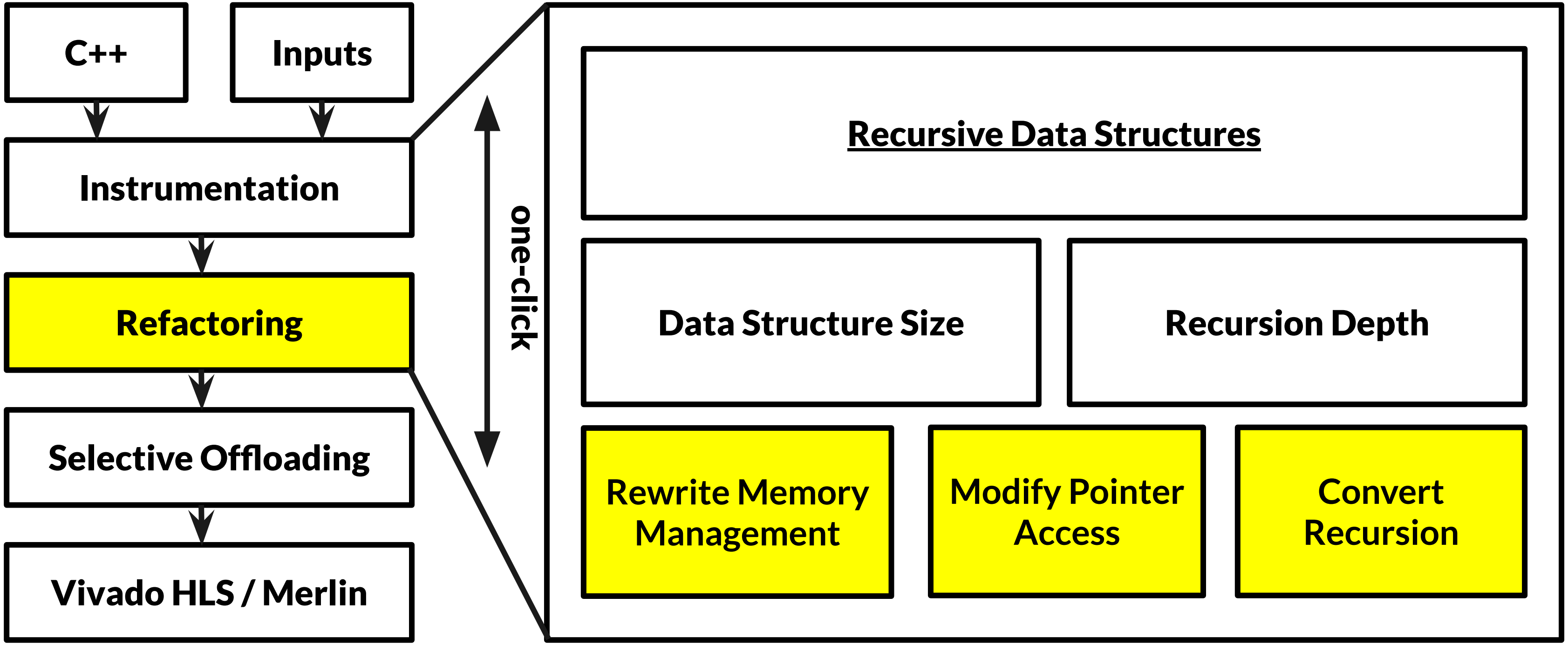
Part 1. Dynamic Data Structures



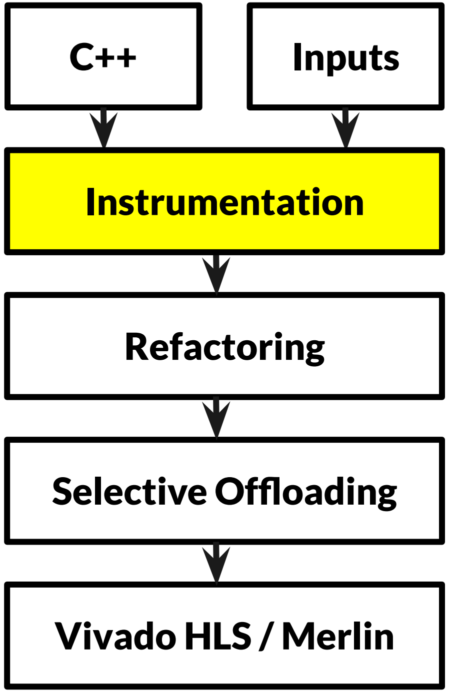
Dynamic Data Structures: Instrumentation



Dynamic Data Structures: Refactoring

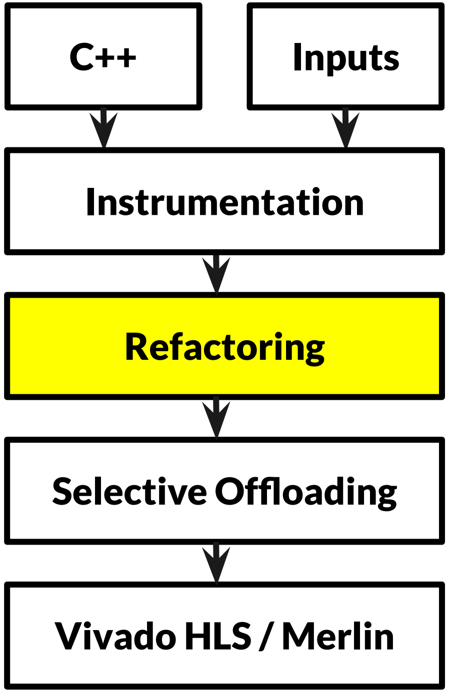


Example Program




```
void init(Node **root) {
     *root = (Node *)malloc(sizeof(Node)); }

void delete_tree(Node *root) { // …
     free(root); }

void traverse(Node *curr) { // entry
     if (curr == NULL)
          return;
     int ret = visit(curr->val);
     traverse(curr->left);
     traverse(curr->right); // return
}

// top-level function
float kernel(float input[], int n) {
     float value = computation(float(..), ..);
}
```

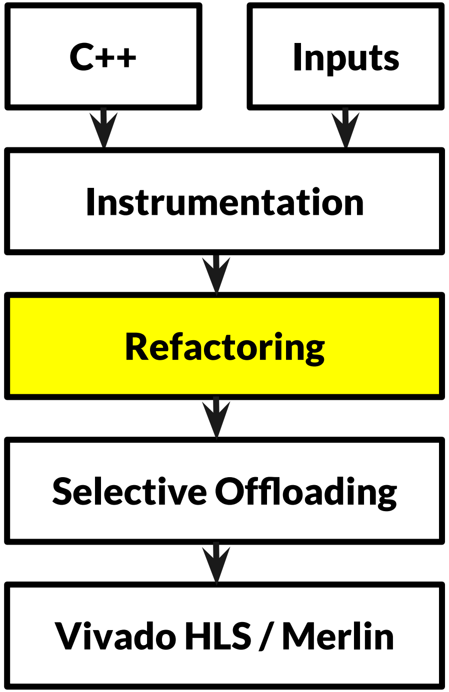
Refactoring Rule 1: Rewrite Mem. Mgmt.




```
void init(Node **root) {
     *root = (Node *)malloc(sizeof(Node)); }

void delete_tree(Node *root) { // …
     free(root); }
```

```
void init(Node_ptr *root) {
     *root = (Node_ptr)Node_malloc(sizeof(Node)); }

void delete_tree(Node_ptr root) { // …
     Node_free(root); }
```

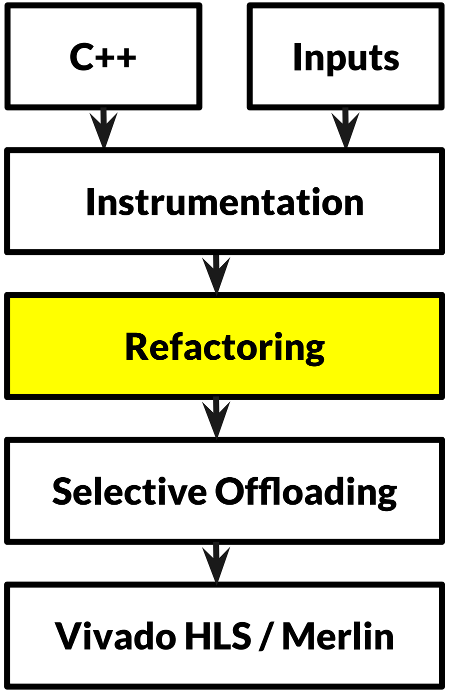
Refactoring Rule 1: Rewrite Mem. Mgmt.




```
void init(Node **root) {
     *root = (Node *)malloc(sizeof(Node)); }

void delete_tree(Node *root) { // …
     free(root); }
```

```
void init(Node_ptr *root) {
     *root = (Node_ptr)Node_malloc(sizeof(Node)); }

void delete_tree(Node_ptr root) { // …
     Node_free(root); }
```

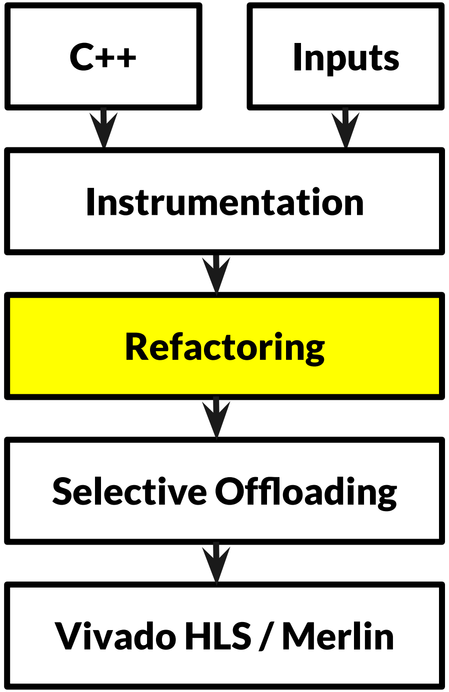
Refactoring Rule 2: Rewrite Pointer Access




```
void traverse(Node_ptr curr) {
     if (curr == NULL) return;
     int ret = visit(curr->val);
     traverse(curr->left);
     traverse(curr->right); }
```

```
Node Node_arr[NODE_ARR_SIZE];
void traverse(Node_ptr curr) {
     if (curr == NULL) return;
     int ret = visit(Node_arr[curr].val);
     traverse(Node_arr[curr].left);
     traverse(Node_arr[curr].right); }
```

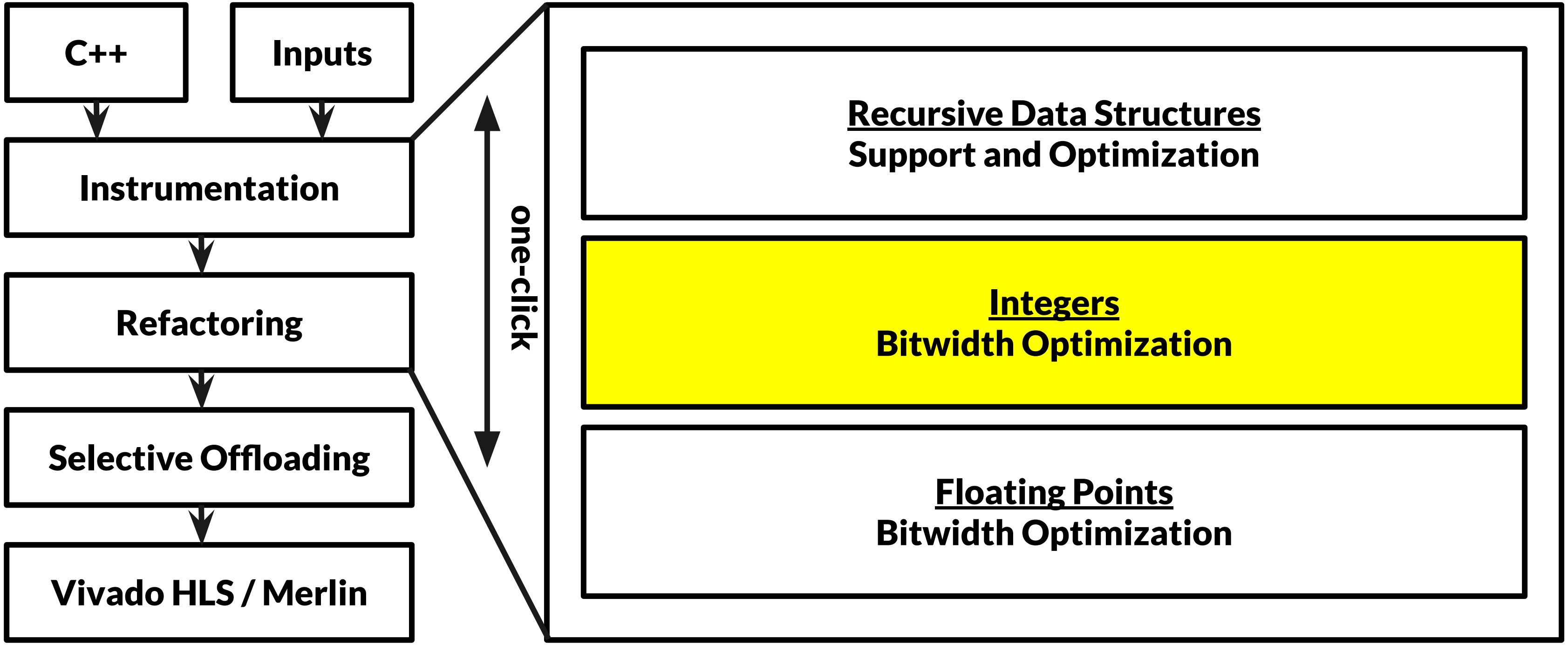
Refactoring Rule 3: Convert Recursion



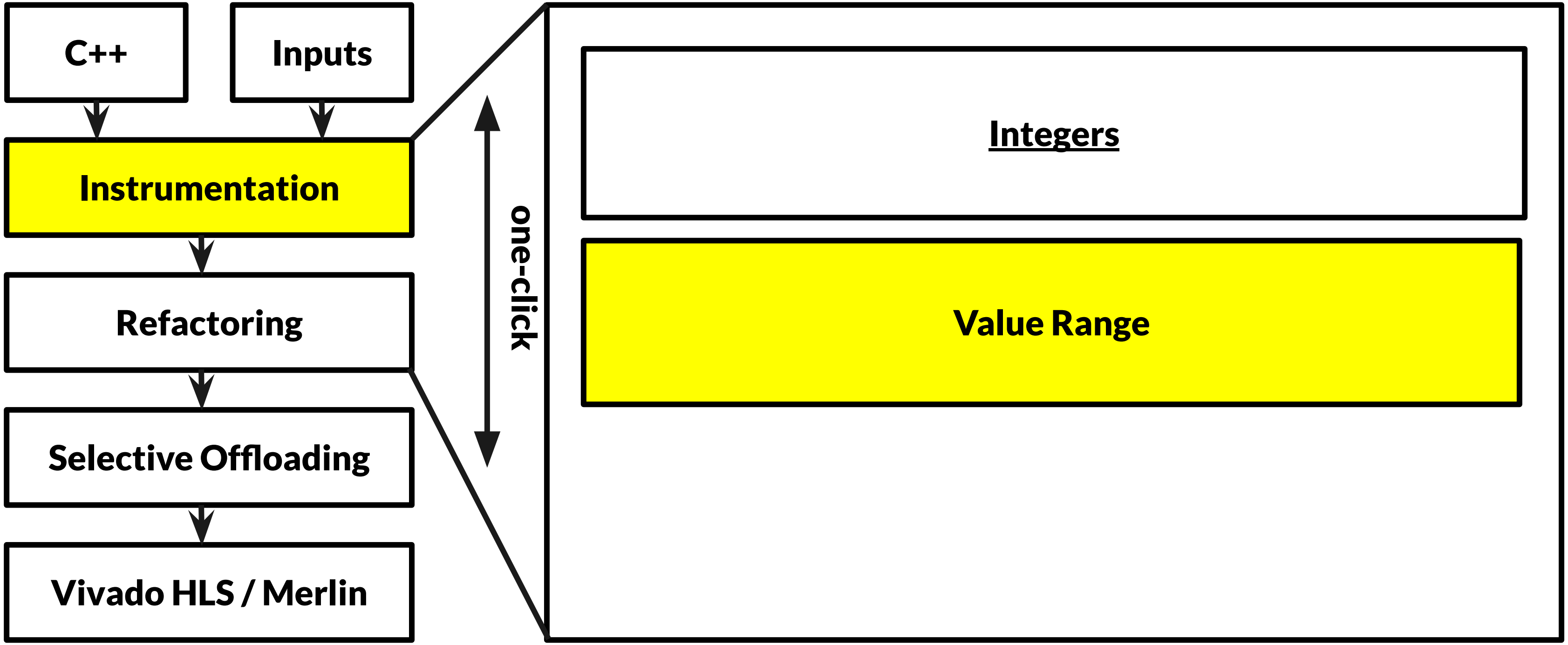

```
void traverse(Node_ptr curr) {
          traverse(Node_arr[curr].left);
          traverse(Node_arr[curr].right); }
```

```
void traverse_converted(Node_ptr curr) {
     stack<context> s(STACK_SIZE);
     while (!s.empty()) {
          context c = s.pop();
          goto c.location;
L0:
          // traverse(Node_arr[curr].left);
          c.location = L1;
          s.push(c);
          s.push({curr: Node_arr[curr].left});
          continue;
L1:
          // ...
     }
}
```

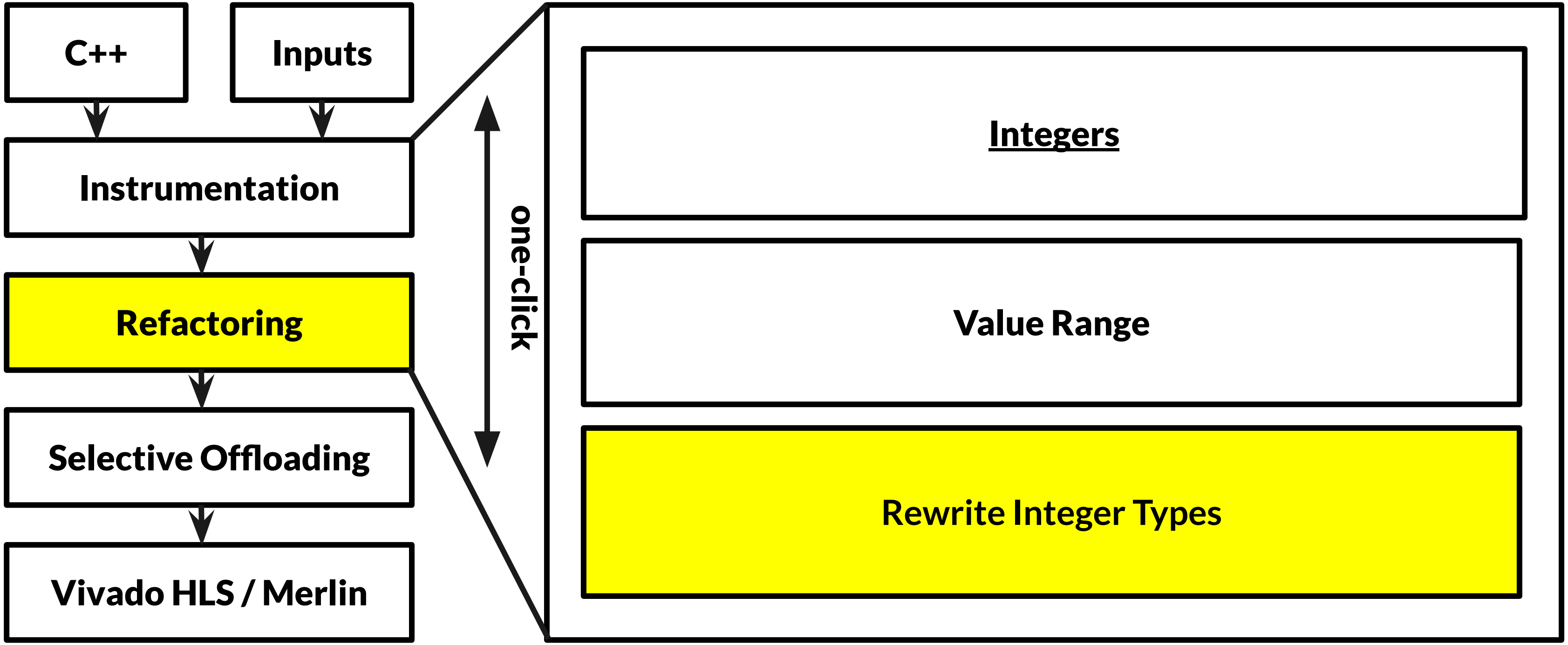
Part 2. Integers



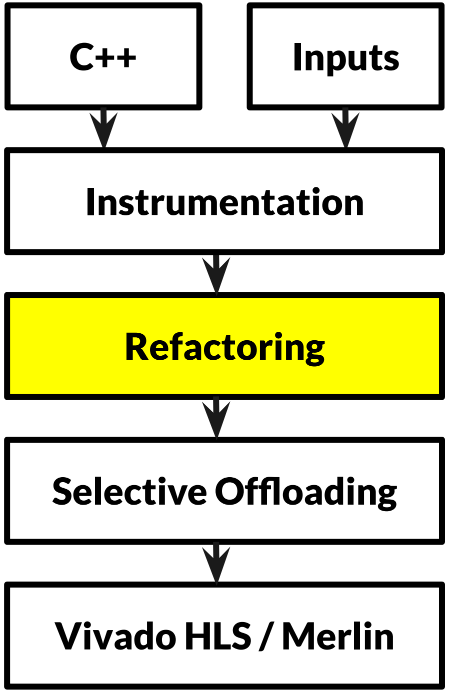
Integers: Kvasir-based Instrumentation



Integers: Refactoring

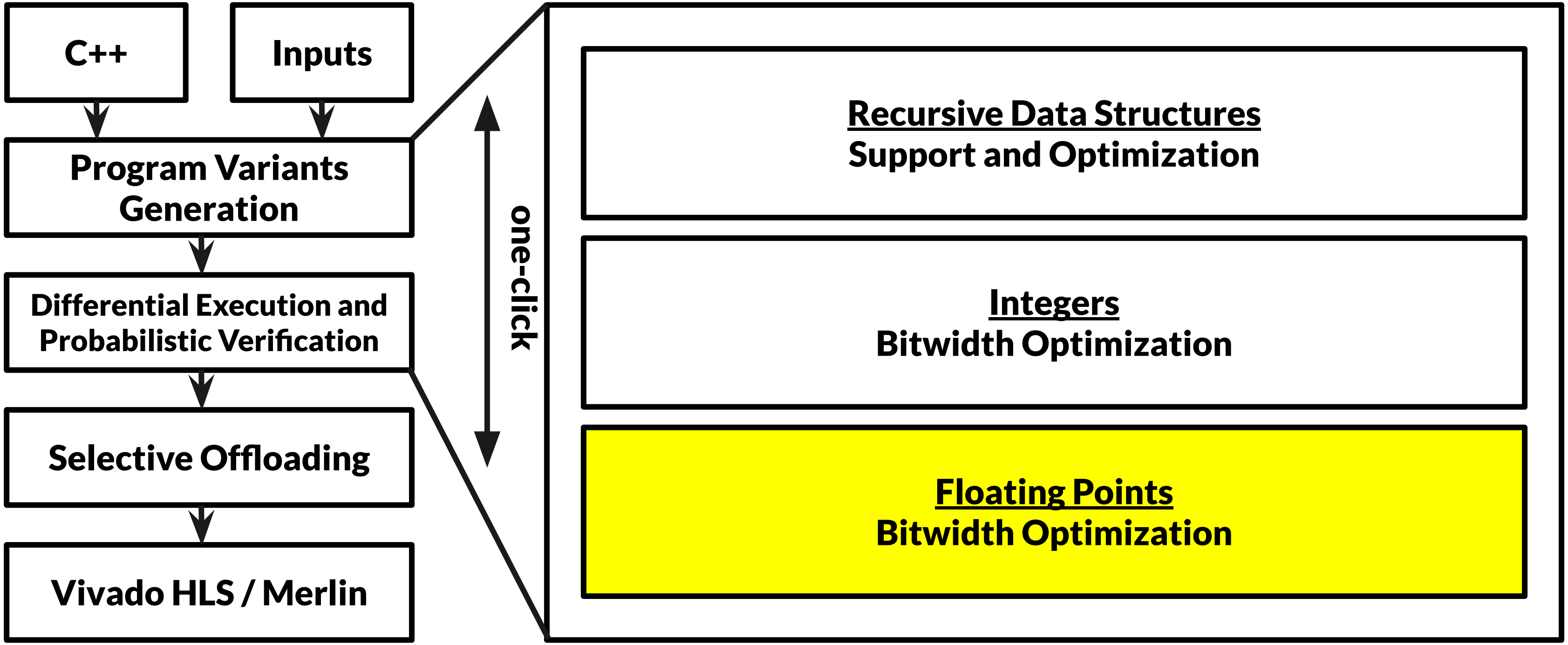


Refactoring Rule: Modify Integer Type

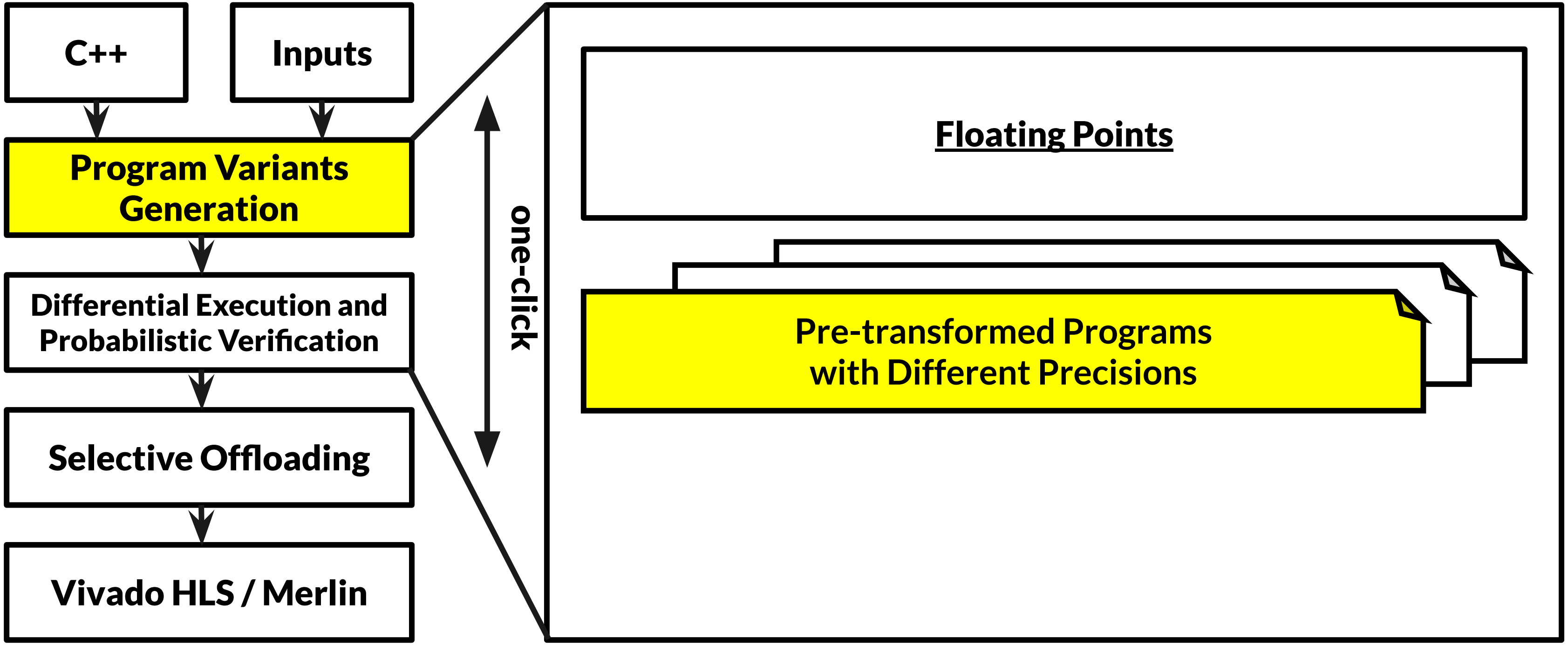



```
Node Node_arr[NODE_ARR_SIZE];
void init(Node_ptr *root) {
     *root = (Node_ptr)Node_malloc(sizeof(Node)); }

void delete_tree(Node_ptr root) { // …
     Node_free(root); }

void traverse(Node_ptr curr) {
     if (curr == NULL) return;
//   @invarants(ret[21,255])
//   int ret = visit(Node_arr[curr].val);
     fpga_uint<8> ret = visit(Node_arr[curr].val);
     traverse(Node_arr[curr].left);
     traverse(Node_arr[curr].right); }

float kernel(float input[], int n) {
     float value = computation(float(..), ..);
}
```

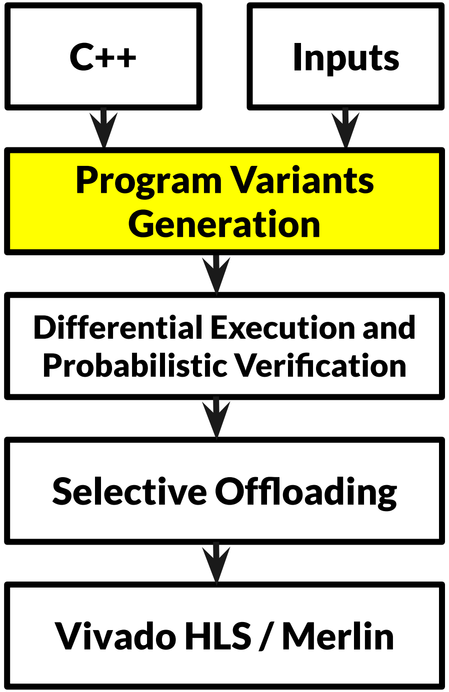
Part 3. Floating Points



Floating Points: Program Variants Generation



Floating Points: Program Variants Generation



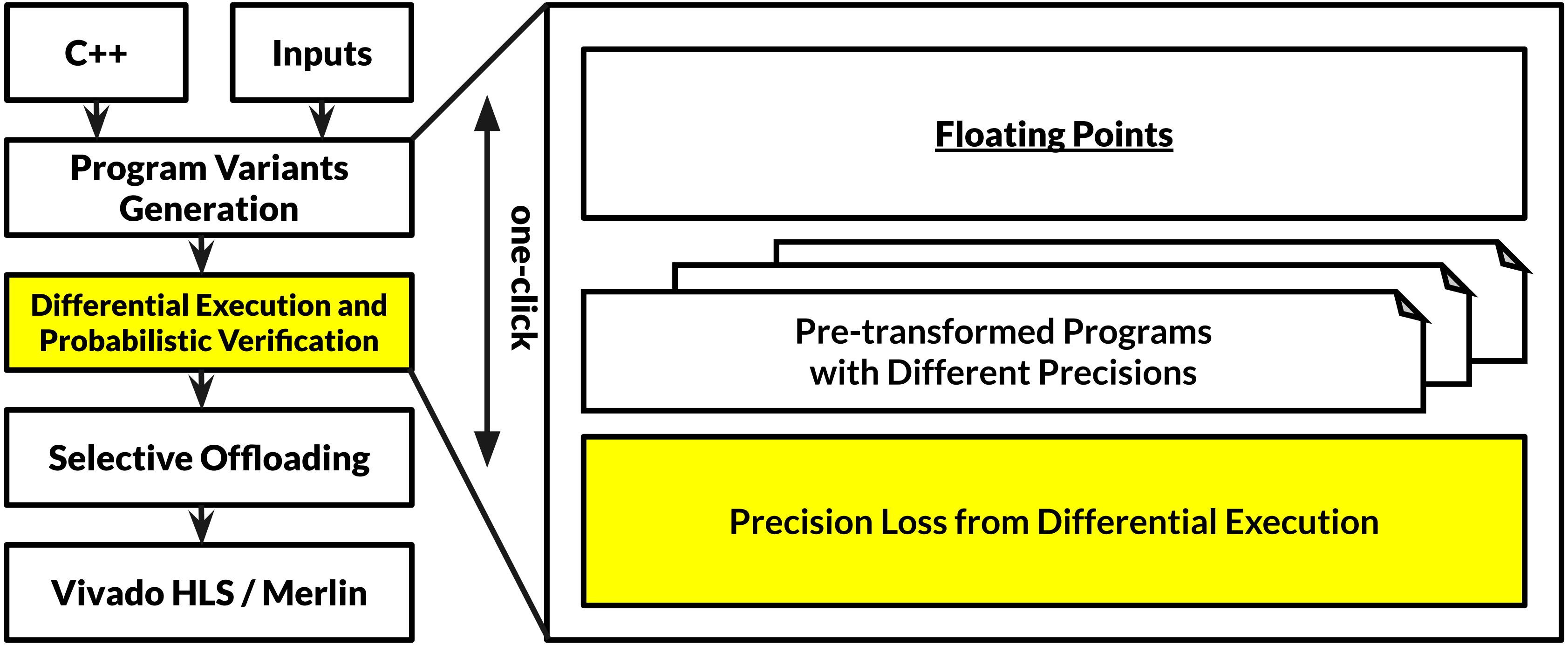

```
float kernel(float input[], int n) {
     float value = computation(float(..), ..);
}
```

```
float low_bit(float input[], int n) {
     fpga_float<8,16> value =
          computation(fpga_float<8,16>(..), ..);
}

float high_bit(float input[], int n) {
     fpga_float<8,23> value =
          computation(fpga_float<8,23>(..), ..);
}
```

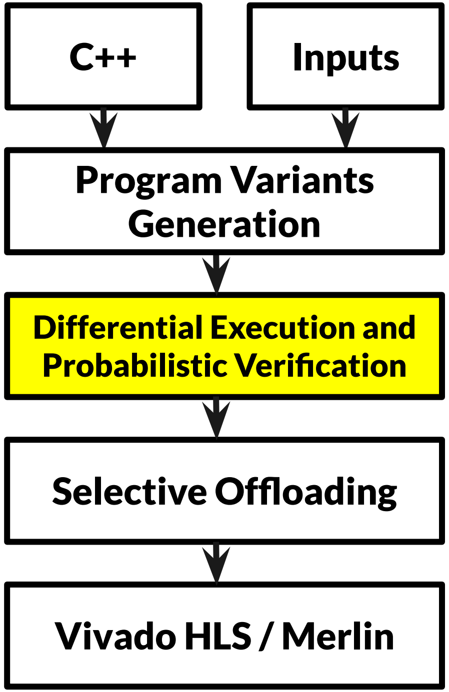
`fpga_float<Exponent, Fraction>` to customize FP precision

* note: `fpga_float<8,23>` is 32 bit float type, `fpga_float<5,16>`uses 22 bits in total

Floating Points: Differential Execution



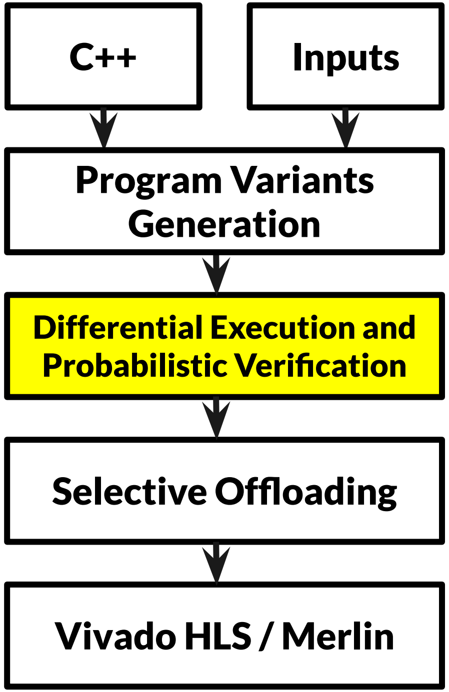
Floating Points: Differential Execution




```
float kernel(float input[], int n) {
     float value = computation(float(..), ..);
}
```

```
float low_bit(float input[], int n) {
     fpga_float<8,16> value =
          computation(fpga_float<8,16>(..), ..);
}
float high_bit(float input[], int n) {
     fpga_float<8,23> value =
          computation(fpga_float<8,23>(..), ..);
}

void verification() {
     float diff = high_bit(..) - bit_ver(..);
     if (diff > epsilon) // failed sample
}
```

Floating Points: Probabilistic Verification



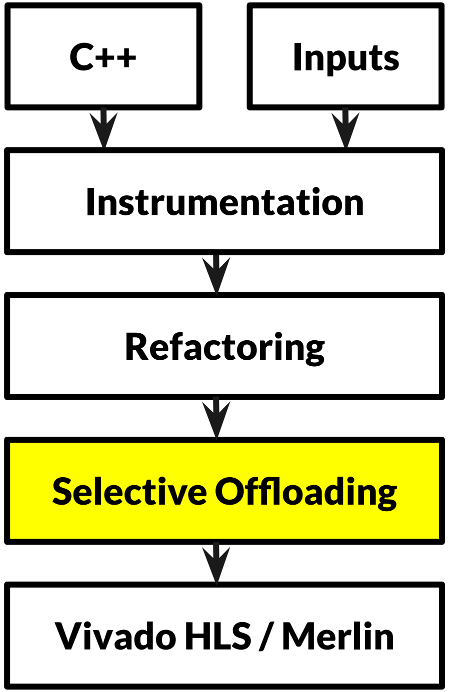

```
void verification() {
     float diff = high_ver(..) - low_ver(..);
     if (diff > epsilon) // failed sample
}
```

Use **Hoeffding's inequality** [1] to calculate the number of samples to meet the required confidence level: alpha.

$$n \geq ln(2/\alpha)/(2\epsilon^2)$$

[1] Hoeffding, Wassily (1963). "Probability inequalities for sums of bounded random variables"

Guard Checking



- **Input check on host and intermediate check on device**
- **Send a signal to the host to indicate fallback when:**
  - Recursive programs: stack overflow, memory failure
  - Integers: overflow
- **The host restart computation on CPU**

Evaluation: Coding Effort

ID / Program	Orig. LOC	Manual LOC	Δ LOC	Auto. LOC	Orig. Chars	Manual Chars	Δ Chars
R1/A.-C.	190	291	33%	557	5673	8776	35%
R2/DFS	86	198	57%	464	2236	5699	61%
R3/L. List	131	235	44%	329	3061	6686	54%
R4/M. Sort	128	342	63%	390	3267	9124	64%
R5/Strassen's	342	735	53%	1006	10026	40971	76%
Geomean			49%				56%

49%

reduction in LOC

56%

in chars

Evaluation: Resource Reduction

Recursive Data Structures*

83% reduction in **BRAM**

42% increase in **Fmax**

** assuming a typical size of 2k, + a conservative size of 16k*

Integer

22% reduction in **FF**

21% reduction in **LUT**

41% reduction in **BRAM**

52% increase in **DSP**

Floating-point

61% reduction in **FF**

39% reduction in **LUT**

50% increase in **DSP**

Acknowledgement

- **Guy Van den Broeck, Brett Chalabian, Todd Millstein, Peng Wei, Cody Hao Yu, Janice Wheeler**
- **Intel CAPA grant**
- **CRISP**, one of six centers in JUMP, a SRC program
- **NSF grants:** CCF-1764077, CCF-1527923, CCF-1723773
- **ONR grant:** N00014-18-1-2037
- **Samsung grant**
- **Center for Domain-Specific Computing (CDSC)**
  - **Xilinx** and **VMWare**.

HeteroRefactor: Refactoring for Heterogeneous Computing with FPGA

Jason Lau*, Aishwarya Sivaraman*, Qian Zhang*,
Muhammad Ali Gulzar, Jason Cong, Miryung Kim
University of California, Los Angeles
**Equal co-first authors in alphabetical order*

VLSI architecture, synthesis & technology

Software Evolution & Analysis Laboratory

The 42nd International Conference on Software Engineering ICSE 2020

- **We adapt and expand automated refactoring to heterogeneous computing with FPGA.**
- **HeteroRefactor provides a novel, end-to-end solution that combines:**
  - **dynamic invariant analysis for identifying common-case.**
  - **kernel refactoring to enhance HLS synthesizability and to reduce memory usage.**
  - **selective offloading with guard checking to guarantee correctness.**
- **The proposed combination is unique to the best of our knowledge.**