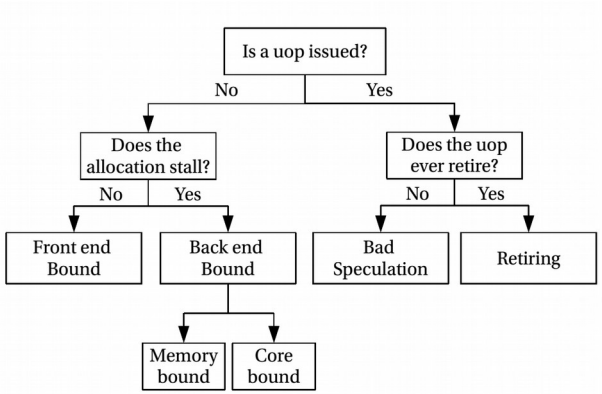
Advanced Optimization Techniques

ICTP Trieste 2015

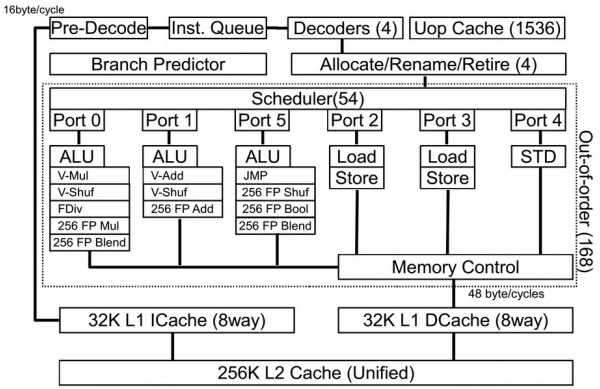
Dr. Christopher Dahnken
Intel GmbH

Outline

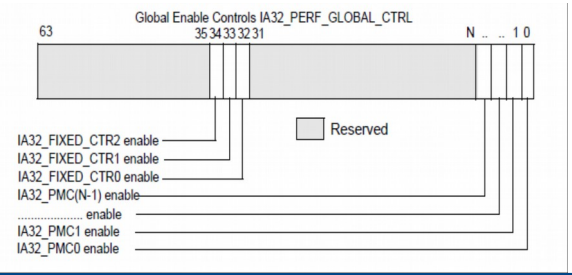
Method



CPU



Measurement



Code

```
!$OMP SECTION
 !   tsend=dclock()
 if(iblock.lt.(nblocks)) then
    nxti=m_of_i(iblock+1)
    nxtj=n_of_i(iblock+1)
    nxtk=k_of_i(iblock+1)

    nxt_buffsize_m=buffersize(ms,bm,nxti)
    nxt_index_m=(nxti-1)*bm+1

    nxt_buffsize_n=buffersize(ns,bn,nxtj)
    nxt_index_n=(nxtj-1)*bn+1

    nxt_buffsize_k=buffersize(ks,bk,nxtk)
    nxt_index_k=(nxtk-1)*bk+1
```

Legal Disclaimer & Optimization Notice

- INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.
- Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.
- Copyright © , Intel Corporation. All rights reserved. Intel, the Intel logo, Xeon, Xeon Phi, Core, VTune, and Cilk are trademarks of Intel Corporation in the U.S. and other countries.

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

A Hierarchical Method for Performance Optimization

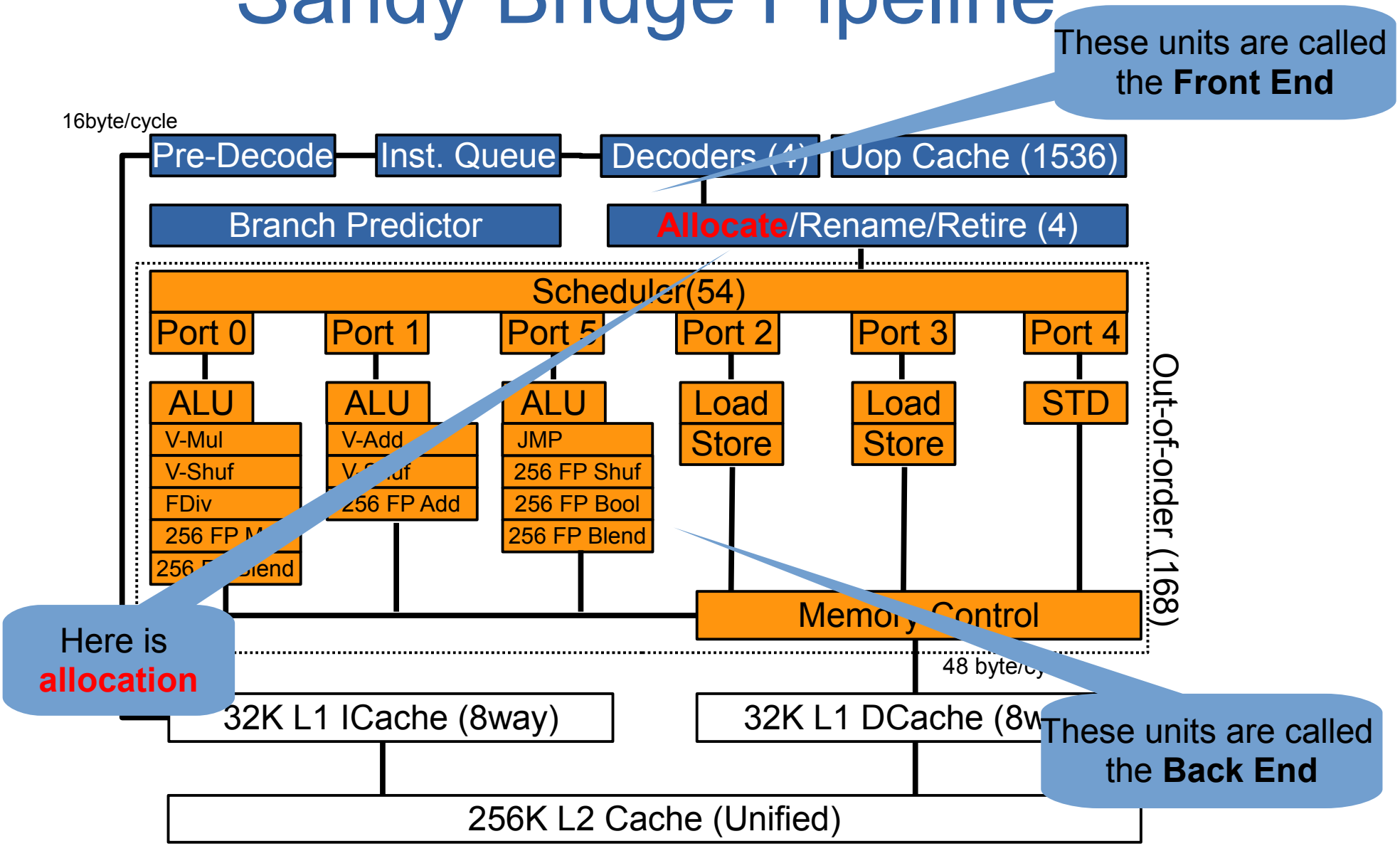
A hierarchical method for performance optimization

- Which code section shows bad performance?
  - Choose only code that executes a considerable fraction of time
- What micro-architectural feature causes this?
  - Front-end, back-end, retirement
  - From there, refine
- Remove the cause/rewrite the code
- Start again until you reach the expected result

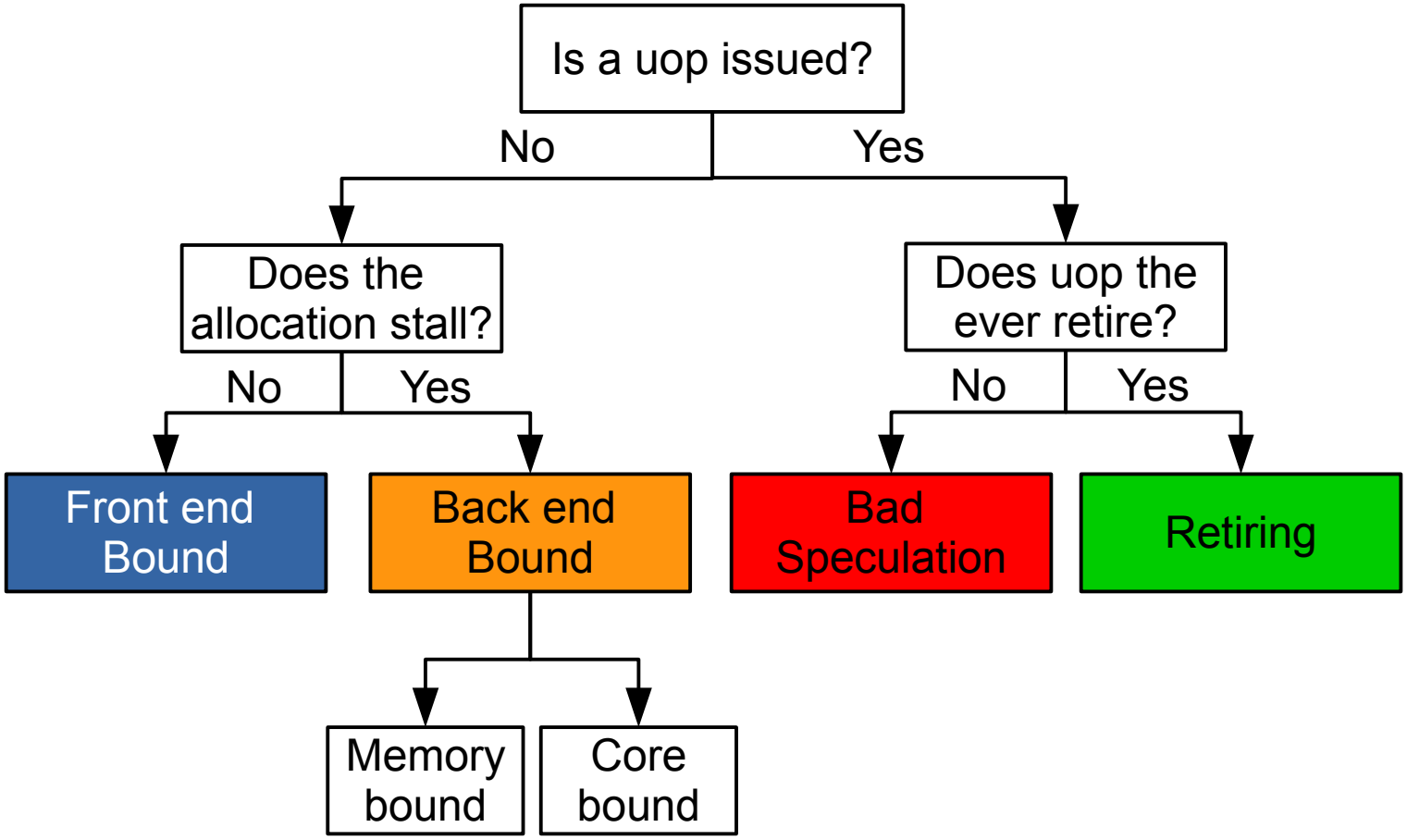
Good measures of performance

- What is a good measure to see the efficiency of the executed code?
- Generally Instructions/clock cycle (IPC) is considered a reasonable magnitude
- SNB/IVB can retire 4 instructions simultaneously: IPC=4 is the optimum
- IPC doesn't say anything about the quality of the instructions
- Generally, GFLOPs or similar are better if you can know them for your algorithm.

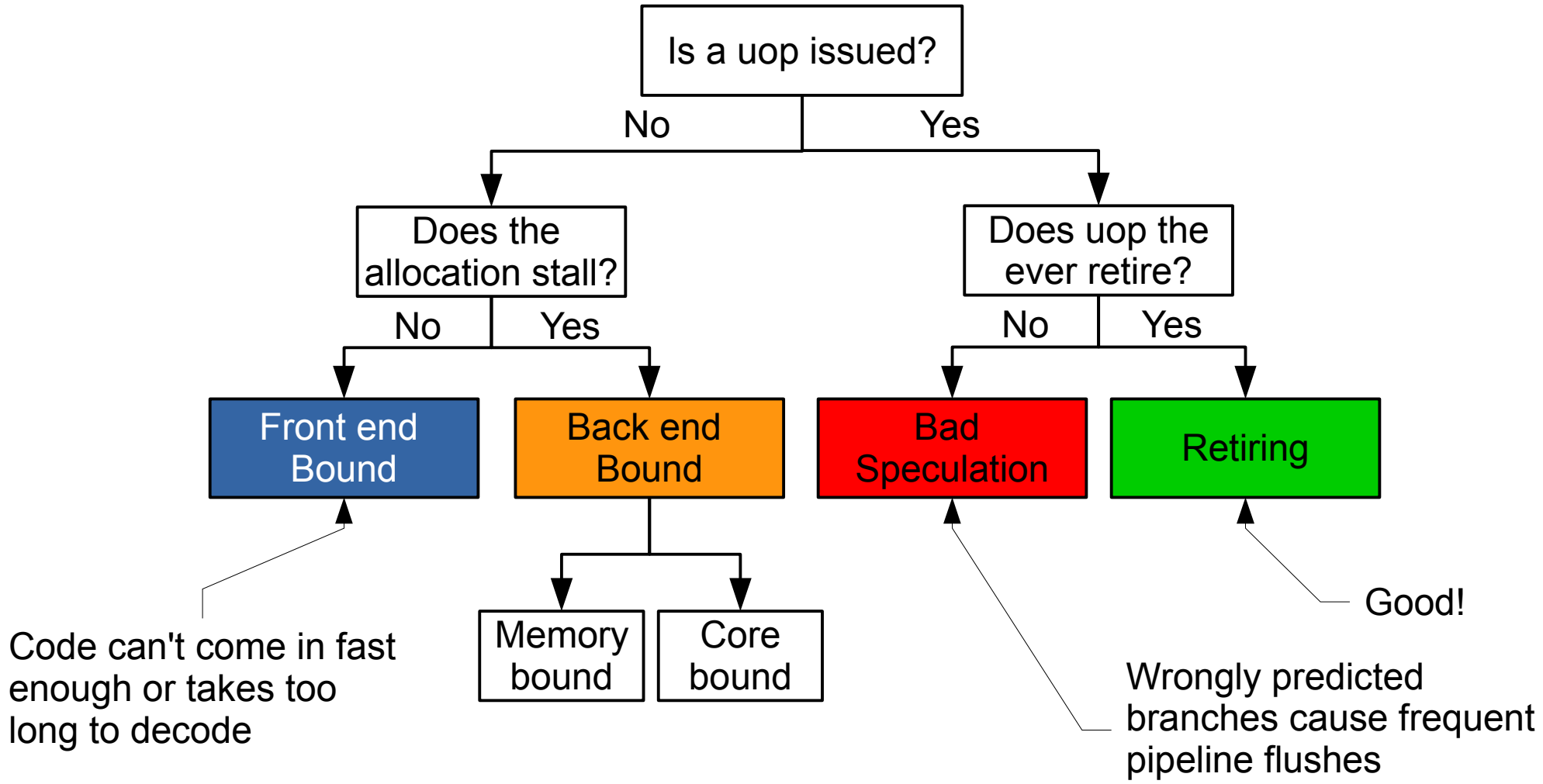
Sandy Bridge Pipeline



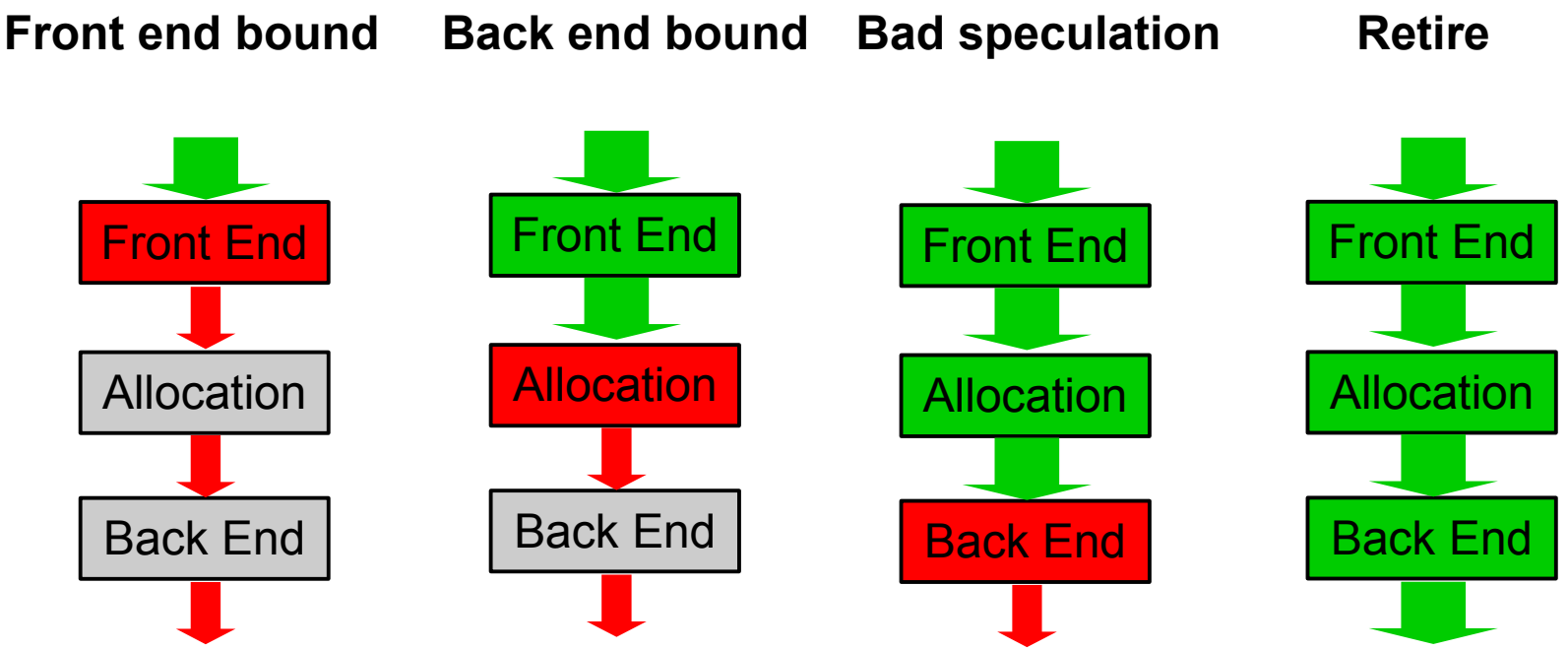
Hierarchical Top-Down Method



Hierarchical Top-Down Method



Hierarchical Top-Down Method



- Meaningful breakdown
- Remember – a fluent pipeline doesn't say something about the quality of the instructions

How to determine the impact in each category?

Normalization

N represents total execution slots opportunities.

N = 4*CPU_CLK_UNHALTED.THREAD

Sandy Bridge has a maximum of 4 instructions/cycle. So 4 times the number of cycles represents the total number of possible instruction executed, which we choose to be our 100% measure.

How to determine the impact in each category?

Percentage of **Front-End bound** execution slots

$$\%FE\_BOUND = 100 \times \frac{IDQ\_UOPS\_NOT\_DELIVERED.CORE}{N}$$

- Intel® 64 and IA-32 Architectures Software Developer's Manual, Chapter 35.9

89H	FFH	BR_MISP_EXEC.ALL_BRANCHES	Counts all near executed branches (not necessarily retired).	
9CH	01H	IDQ_UOPS_NOT_DELIVERED.CORE	Count number of non-delivered uops to RAT per thread.	Use Cmask to qualify uop b/w
A1H	01H	UOPS_DISPATCHED_PORT.PORT_0	Cycles which a Uop is dispatched on port 0.	

How to determine the impact in each category?

Percentage of **Retired** execution slots

$$\%\text{Retiring} = 100 \times \frac{\text{UOPS\_RETIRED.RETIRE\_SLOTS}}{\text{N}};$$

- All the retired instructions (max 4/cycle) divided by all possible execution slots

How to determine the impact in each category?

Percentage of **Bad Speculation** execution slots

$$\%\text{Bad\_Speculation} = 100 \times \frac{\left(\text{UOPS\_ISSUED.ANY} - \text{UOPS\_RETIRED.RETIRE\_SLOTS} + 4 \times \text{INT\_MISC.RECOVERY\_CYCLES}\right)}{N}$$

- If no bad speculation: UOPS_ISSUED.ANY = UOPS_RETIRED.RETIRE_SLOTS
- If only bad speculation: we only spend time in the recovery

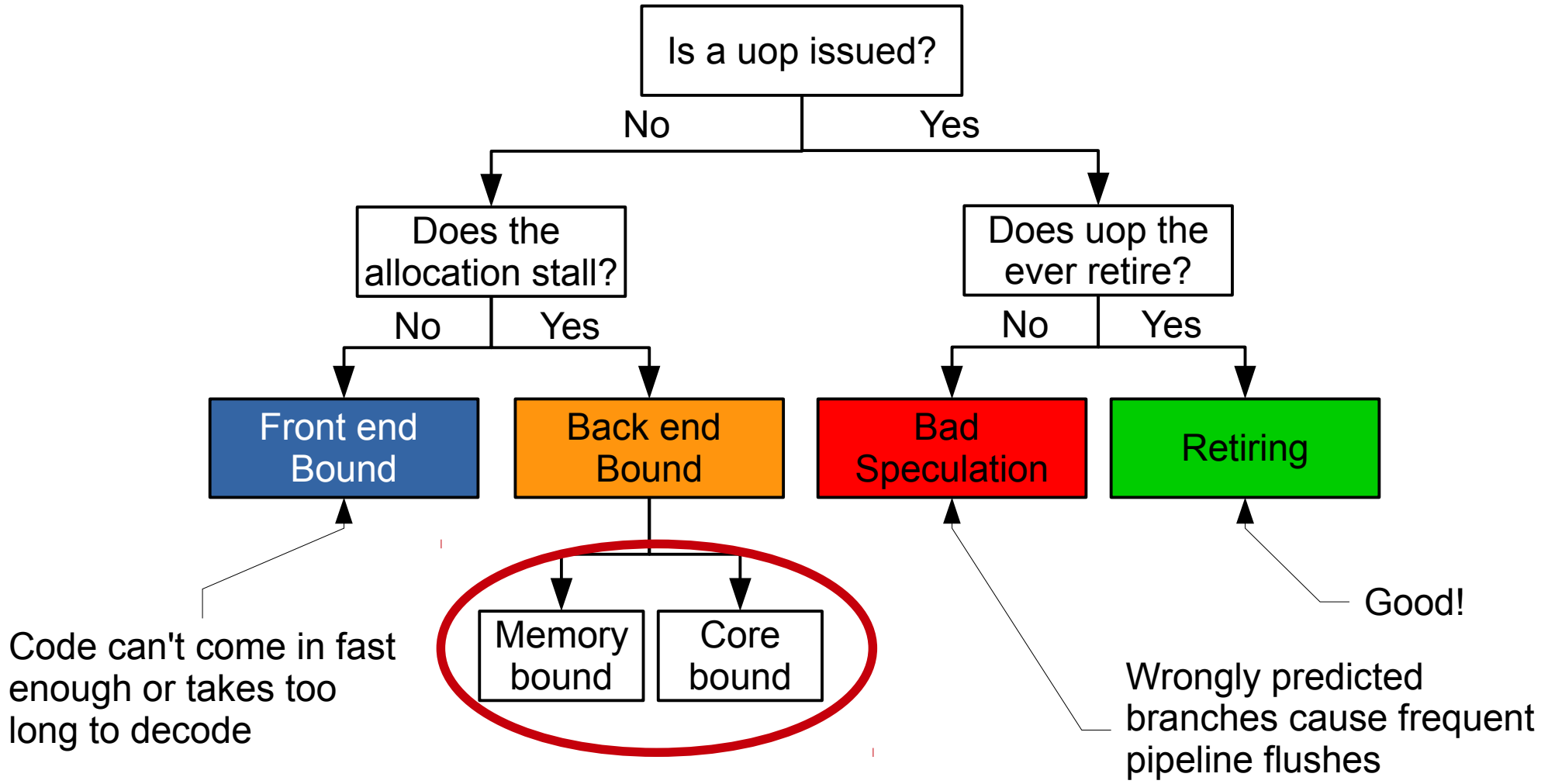
How to determine the impact in each category?

Percentage of **Back End bound** execution slots

$$\%BE\_Bound = 100 \times (1 - (FE\_Bound + Retiring + Bad\_Speculation));$$

- Sum rule: FE_Bound+Retiring+Bad_Speculation+BE_Bound =1
- Everything that is not spend somewhere else will be found in BE_Bound

Hierarchical Top-Down Method



CPU and memory bound codes

- If you are not in the memory bound category, you are CPU bound
- Advanced tools like Vtune will report this. Likwid also can give you the memory bandwidth used.
- If you are working bare metal (e.g. with wrmsr/rdmsr or perf) you can get measures for the the number of loads (perf top -e mem-loads)
- Also consider looking at cache misses via the architectural counters, e.g. cache misses/cache references

Vtune Output

Elapsed Time: 6.166s

Clockticks:	20,392,030,588
Instructions Retired:	37,218,055,827
CPI Rate:	0.548
MUX Reliability:	0.997
Paused Time:	0s

- **Filled Pipeline Slots:**
  - **Retiring: 0.639**
    - Microcode Sequencer: 0.018
    - **General Retirement: 0.628**

      This metric represents a fraction of slots during which CPU was retiring uOps not originated from the Microcode Sequencer. This correlates with the total number of instructions executed by the program. A uOps-per-Instruction ratio of 1 is expected. If the Retirement value for non-vectorized code is high, consider vectorizing your code to reduce instructions and hence this bucket.
      - Other: 0.760

        This metric represents a non-floating-point (FP) uop fraction the CPU has executed. If your application has no FP operations, this is likely to be the biggest fraction.
      - **FP Arithmetic: 0.213**

        This metric represents an overall arithmetic floating-point (FP) uops fraction the CPU has executed.
        - FP x87: 0.000
        - FP Scalar: 0.213

          This metric represents an arithmetic floating-point (FP) scalar uops fraction the CPU has executed. Analyze metric values to identify why vector code is not generated, which is typically caused by the selected algorithm or missing/wrong compiler...
        - FP Vector: 0.000
  - **Bad Speculation: 0.011**
- **Unfilled Pipeline Slots (Stalls):**
  - **Back-End Bound: 0.255**

    Identify slots where no uOps are delivered due to a lack of required resources for accepting more uOps in the back-end of the pipeline. Back-end metrics describe a portion of the pipeline where the out-of-order scheduler dispatches ready uOps into their respective execution units, and, once completed, these uOps get retired according to program order. Stalls due to data-cache misses or stalls due to the overloaded divider unit are examples of back-end bound issues.
    - **Memory Bound:**
      - **L1 Bound: 0.026**
      - **L3 Bound: 0.000**
      - **DRAM Bound: 0.030**
        - Memory Bandwidth: 0.004
        - Memory Latency: 0.143

          This metric shows how often CPU could be stalled due to the latency of the main memory (DRAM). Consider optimizing data layout or using Software Prefetches (through the compiler).
        - Local DRAM: 0.023
        - Remote DRAM: 0.000
        - Remote Cache: 0.000

Hierarchical To-Down Method

- Now you should have a rough idea where you code exhibit some weaknesses
- It is likely you will end up in the memory bound or CPU bound category after you resolved FE and bad speculation issues.
- We will look at remedies for all of this in the next module

On the quality of instructions

- As mentioned above, this method gives you only insight on the execution of instructions
- It doesn't say anything about the quality
- You could be retiring at the maximum level and still have bad code → if you are no memory CPU bound, it is a sign of alert and needs checking.
- Let's look at an example ...

What's wrong with this code

```
#include <xmmintrin.h>
int main(void){
  int size=10000;
  float* a = (float*) _mm_malloc(sizeof(float)*size,32);
  for(int i=0;i<size;i++){
    a[i]=i;
  }
  for(int i=0;i<size;i++){
    a[i]=a[i]+1;
  }
}
```

Callonacci-Effect

What's wrong with this code

- Easy code, should be perfectly vectorizable
- The devil is in the detail:
  `a[i]=a[i]+1;`
  For this code the standard assumes that 1 is a double and demands that the float array is up-converted to double, the 1 is added and then the array is down-converted to float → 2/3 of all operations are conversions!
- For such codes the IPC could potentially be 4, but the overall performance in absolute numbers is bad!

Summary

- We have introduced a sum-rule consistent hierarchical method to account for all upos
- You should be able to categorize you code and perform appropriate optimizations (to be introduced next)