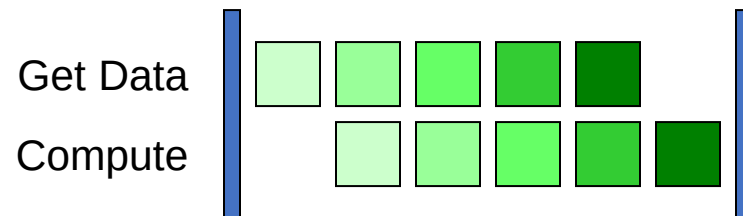
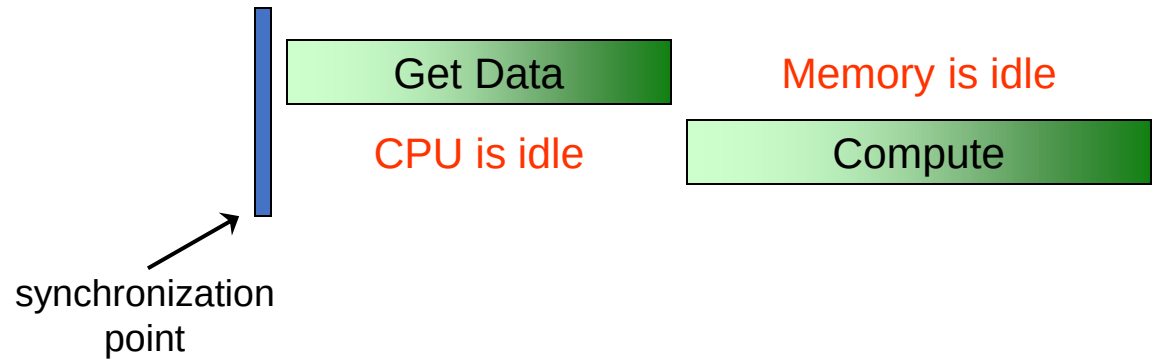




# Memory Performance Monitoring

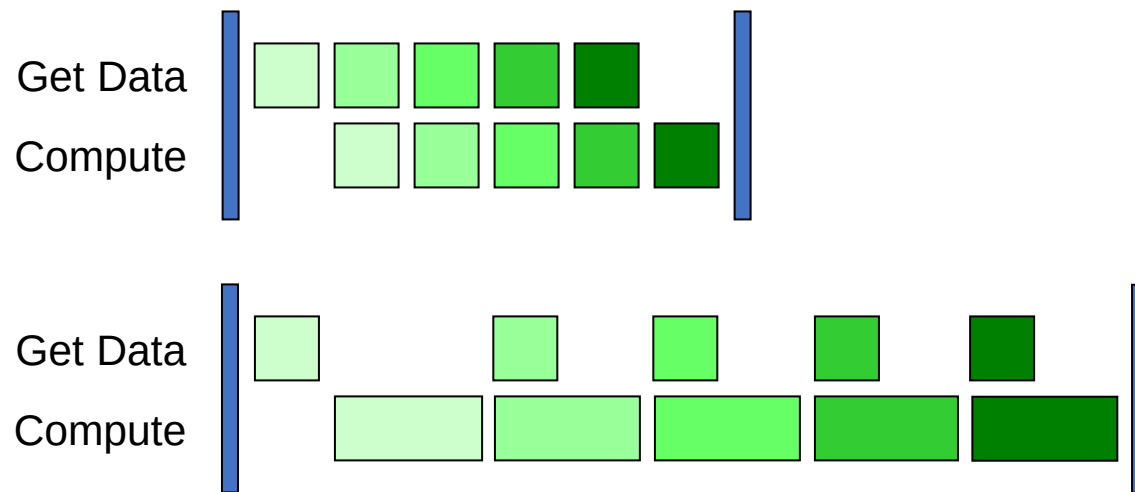
## Ozcan Ozturk – Sabancı University

# Overlapping Communication with Computation



# Limits in Pipelining Communication

- Computation to communication ratio limits performance gains from pipelining



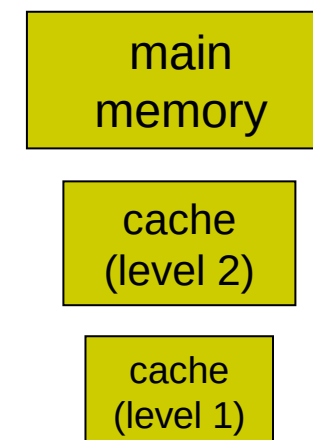
- Where else to look for performance?

# Spatial Locality

- CPU asks for data at address 1000
- Memory sends data at address 1000 ... 1064
  - Amount of data sent depends on architecture parameters such as the cache block size
- Works well if CPU actually ends up using data from 1001, 1002, ..., 1064
- Otherwise wasted bandwidth and cache capacity

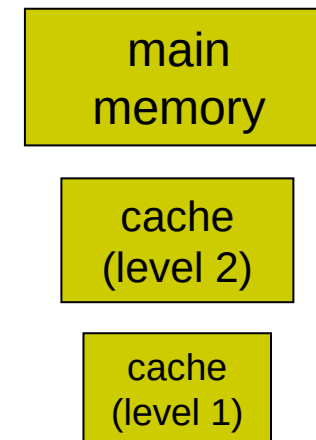
# Temporal Locality

- Main memory access is expensive
- Memory hierarchy adds small but fast memories (caches) near the CPU
  - Memories get bigger as distance from CPU increases
- CPU asks for data at address 1000
- Memory hierarchy anticipates more accesses to same address and stores a local copy
- Works well if CPU actually ends up using data from 1000 over and over and over ...
- Otherwise wasted cache capacity



# Reducing Artifactual Costs in Distributed Memory Architectures

- Data is transferred in chunks to amortize communication cost
  - DMA gets up to 16K
  - Usually get a contiguous chunk of memory
- Spatial locality
  - Computation should exhibit good spatial locality characteristics
- Temporal locality
  - Reorder computation to maximize use of data fetched



# Understanding Performance

- Need some way of measuring performance
  - Coarse grained measurements

```
% gcc sample.c
% time a.out
2.312u 0.062s 0:02.50 94.8%
% gcc sample.c -O3
% time a.out
1.921u 0.093s 0:02.03 99.0%
```
- ... but did we learn much about what's going on?

```
#define N (1 << 23)
#define T (10)
#include <string.h>
double a[N],b[N];

void cleara(double a[N]) {
    int i;
    for (i = 0; i < N; i++) {
        a[i] = 0;
    }
}

int main() {
    double s=0,s2=0; int i,j;
    for (j = 0; j < T; j++) {
        for (i = 0; i < N; i++) {
            b[i] = 0;
        }
        cleara(a);
        memset(a,0,sizeof(a)); record start time
        

---


        for (i = 0; i < N; i++) {
            s += a[i] * b[i];
            s2 += a[i] * a[i] + b[i] * b[i];
        }
        

---

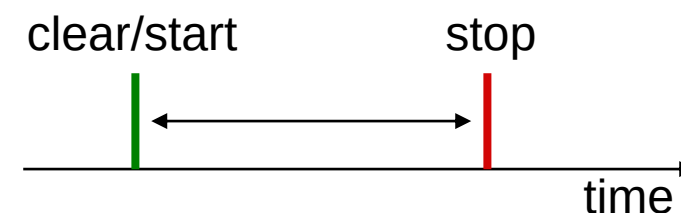

record stop time
    }
    printf("s %f s2 %f\n",s,s2);
}
```

# Hardware Performance Counters

- Special purpose registers built into the processor microarchitecture
  - Monitor hardware-related activity within the computer
- Useful for performance analysis and tuning
  - Provide low-level information that can not be obtained with software profilers

# Measurements Using Counters

- Increasingly possible to get accurate measurements using performance counters
  - Special registers in the hardware to measure events
- Insert code to start, read, and stop counter
  - Measure exactly what you want, anywhere you want
  - Can measure communication and computation duration
  - But requires manual changes
  - Monitoring nested scopes is an issue
  - Heisenberg effect: counters can perturb execution time



# Dynamic Profiling

- Event-based profiling
  - Interrupt execution when an event counter reaches a threshold
- Time-based profiling
  - Interrupt execution every  $t$  seconds
- Works without modifying your code
  - Does not require that you know where problem might be
  - Supports multiple languages and programming models
  - Quite efficient for appropriate sampling frequencies

# Counter Examples

- Cycles (clock ticks)
- Pipeline stalls
- Cache hits
- Cache misses
- Number of instructions
- Number of loads
- Number of stores
- Number of floating point operations
- ...

# Counter Examples

## Alpha EV5

| Name                | Description                                   | Reg# |
|---------------------|-----------------------------------------------|------|
| CYCLES              | Total cycles                                  | 0, 2 |
| ISSUES              | Total issues                                  | 0    |
| NON_ISSUE_CYCLES    | Nothing issued, pipeline frozen               | 1    |
| SPLIT_ISSUE_CYCLES  | Some but not all issuable instructions issued | 1    |
| PIPELINE_DRY        | Nothing issued, pipeline dry                  | 1    |
| REPLAY_TRAP         | Replay traps (ldu, wb/maf, litmus test)       | 1    |
| SINGLE_ISSUE_CYCLES | Single issue cycles                           | 1    |
| DUAL_ISSUE_CYCLES   | Dual issue cycles                             | 1    |
| TRIPLE_ISSUE_CYCLES | Triple issue cycles                           | 1    |
| QUAD_ISSUE_CYCLES   | Quad issue cycles                             | 1    |
| FLOW_CHANGE         | Flow change (meaning depends on counter 2)    | 1    |

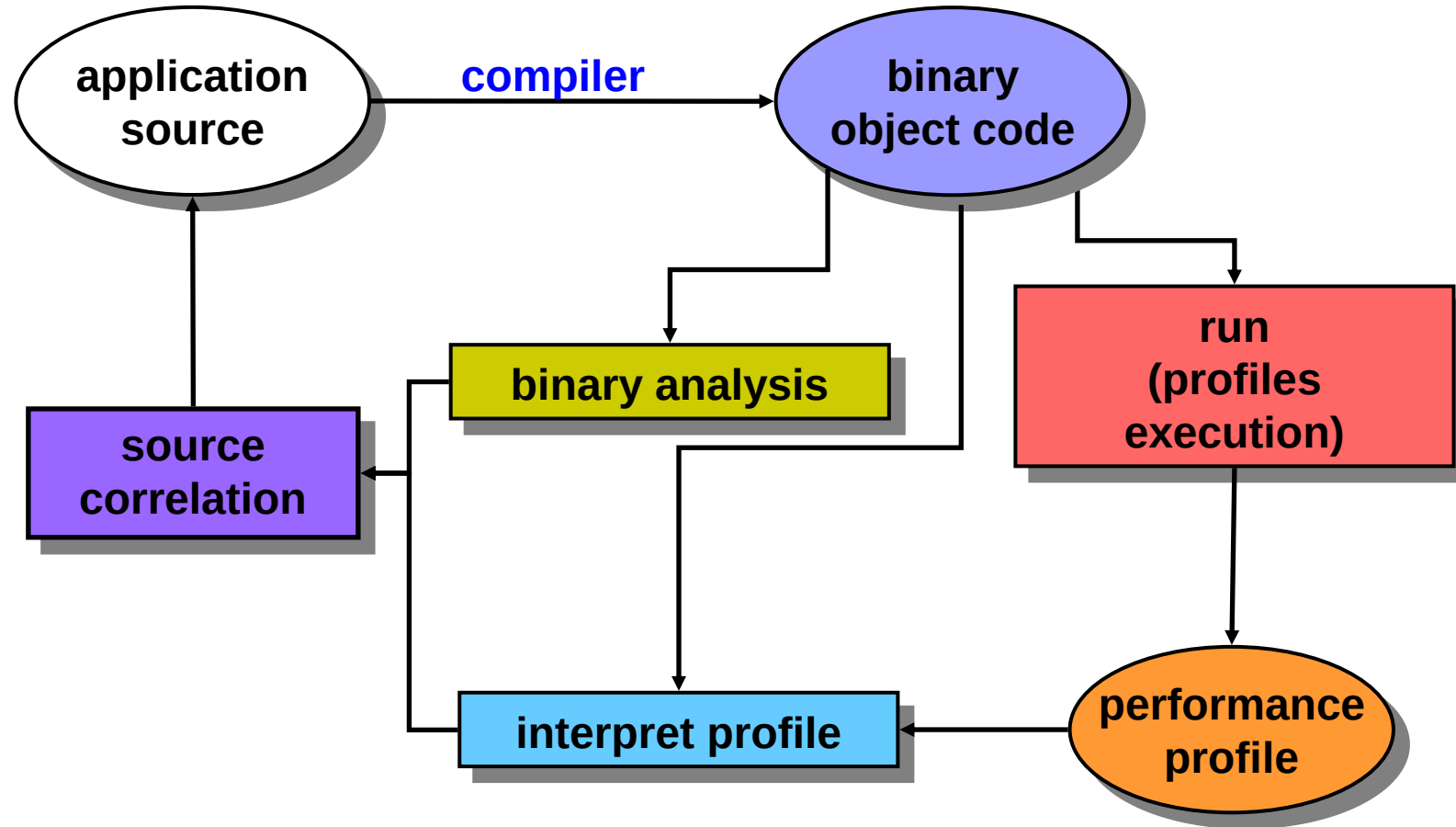
## IBM Power 5++ (759 counters total)

| Name                   | Description                           | Reg# |
|------------------------|---------------------------------------|------|
| CYCLES                 | Processor Cycles                      | 1    |
| CYCLES_RND_SMPL        | Processor Cycles with random sampling | 2    |
| PM_RUN_CYC_GRP1        | Run cycles                            | 0    |
| PM_INST_CMPL_GRP1      | Instructions completed                | 1    |
| PM_INST_DISP_GRP1      | Instructions dispatched               | 2    |
| PM_CYC_GRP1            | Processor cycles                      | 3    |
| PM_1PLUS_PPC_CMPL_GRP2 | One or more PPC instruction completed | 0    |
| PM_GCT_EMPTY_CYC_GRP2  | Cycles GCT empty                      | 1    |
| PM_GRP_CMPL_GRP2       | Group completed                       | 2    |
| PM_CYC_GRP2            | Processor cycles                      | 3    |

# Useful Derived Measurements

- Processor utilization
  - Cycles / Wall Clock Time
- Instructions per cycle
  - Instructions / Cycles
- Instructions per memory operation
  - Instructions / Loads + Stores
- Average number of instructions per load miss
  - Instructions / L1 Load Misses
- Many others
  - Cache miss rate
  - Branch misprediction rate
  - ...

# Common Profiling Workflow



# Hardware Performance Counters

- Special purpose registers built into the processor microarchitecture
  - Monitor hardware-related activity within the computer
- Useful for performance analysis and tuning
  - Provide low-level information that can not be obtained with software profilers

# Popular Runtime Profiling Tools

- GNU gprof
  - Widely available with UNIX/Linux distributions
  - `gcc -O2 -pg foo.c -o foo → ./foo → gprof foo`
- HPC Toolkit
  - <https://hpctoolkit.org/>
- PAPI
  - <http://icl.cs.utk.edu/papi/>
- Intel VTune
  - <https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html>
- AMD µProf
  - <https://www.amd.com/en/developer/uprof.html>
- Many others

# GNU gprof

- MPEG-2 decoder (reference implementation)

```
% ./mpeg2decode -b mei16v2.m2v -f -r
```

-r uses double precision inverse DCT

| %     | cumulative | self    |       | self     | total    |                          |
|-------|------------|---------|-------|----------|----------|--------------------------|
| time  | seconds    | seconds | calls | ns/call  | ns/call  | name                     |
| 90.48 | 0.19       | 0.19    | 7920  | 23989.90 | 23989.90 | Reference_IDCT           |
| 4.76  | 0.20       | 0.01    | 2148  | 4655.49  | 4655.49  | Decode_MPEG1_Intra_Block |

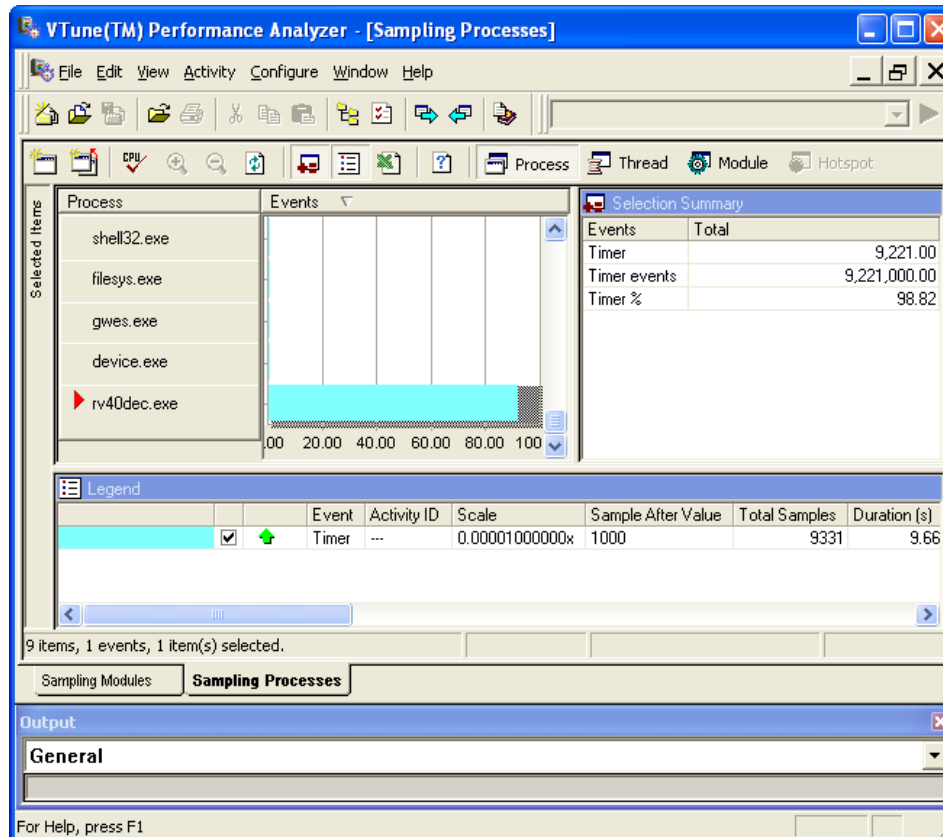
```
% ./mpeg2decode -b mei16v2.m2v -f
```

uses fast integer based inverse DCT instead

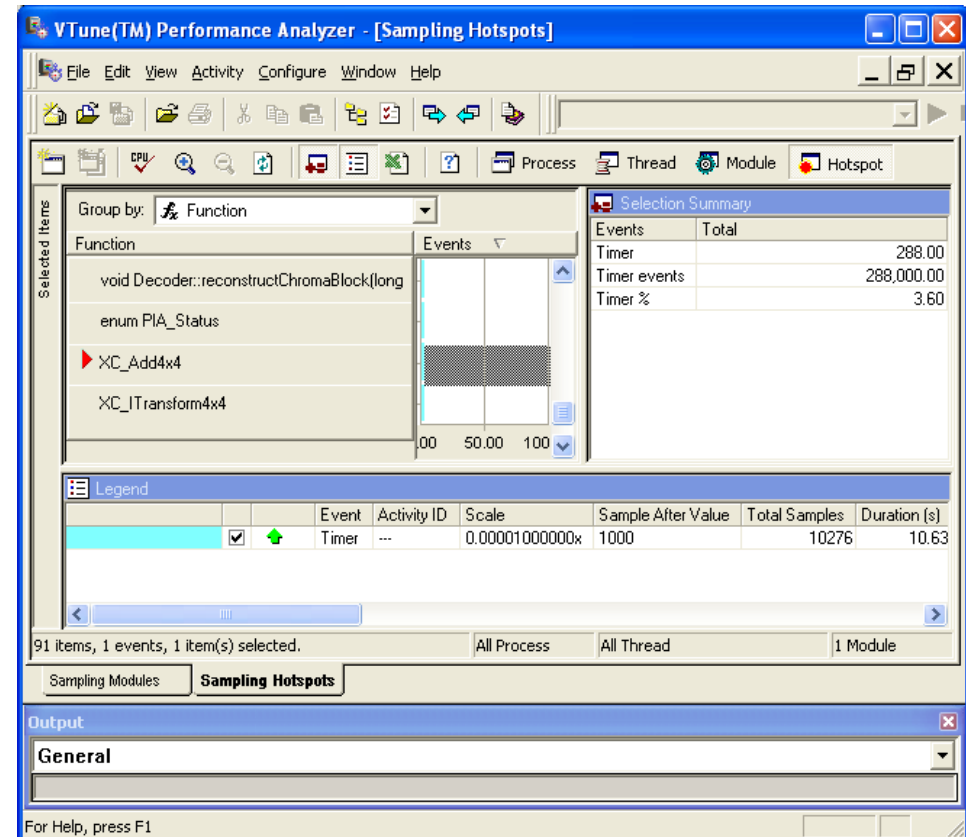
|       |      |      |       |         |         |                           |
|-------|------|------|-------|---------|---------|---------------------------|
| 66.67 | 0.02 | 0.02 | 8238  | 2427.77 | 2427.77 | form_component_prediction |
| 33.33 | 0.03 | 0.01 | 63360 | 157.83  | 157.83  | idctcol                   |

# Intel VTune

## Process View



## Drill Down



## Next Lecture

- Memory Optimizations

# References

Grama, Ananth, Anshul Gupta, George Karypis, and Vipin Kumar. 2003. *Introduction to Parallel Computing*. 2nd ed. Boston: Addison-Wesley.

# Sabancı Universitesi Thank you!



**Co-funded by  
the European Union**



**EuroHPC**  
Joint Undertaking



EuroCC 3 has received funding from the [European High-Performance Computing Joint Undertaking \(JU\)](#) under Grant Agreement No. 101306701. The JU receives support from the [European Union](#)'s Digital Europe Programme and Germany, Albania, Austria, Belgium, Bosnia and Herzegovina, Bulgaria, Croatia, Cyprus, Czechia, Denmark, Estonia, Finland, France, Greece, Hungary, Iceland, Ireland, Italy, Latvia, Lithuania, Luxembourg, Malta, Montenegro, the Netherlands, North Macedonia, Norway, Poland, Portugal, Romania, Serbia, Slovakia, Slovenia, Spain, Sweden, Türkiye, and Kosovo.

Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or EuroHPC Joint Undertaking. Neither the European Union nor the EuroHPC Joint Undertaking can be held responsible for them.