18-344 Recitation 1

09/05/2025

Outline

1. Logistics
2. Review
3. Tool Primer
4. Labs + Lab Reports
    a. Doing the lab
    b. Writing the report
    c. Presenting the data

Logistics

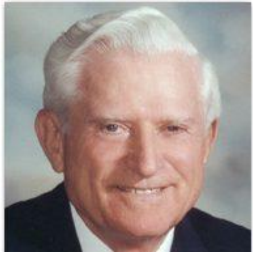
- Lab 0 due September 15 (next next Monday) (in 10 days)
  - Review last week's recitation if needed
- Homework 1 due September 14 (next next next Sunday) (in 9 days)
  - Covers materials from lectures 1-4

Review

Speedup

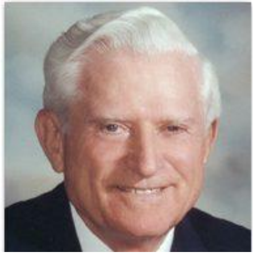
$$S = \frac{t_{\mathrm{base}}}{t_{\mathrm{improved}}}$$

Amdahl's Law v1



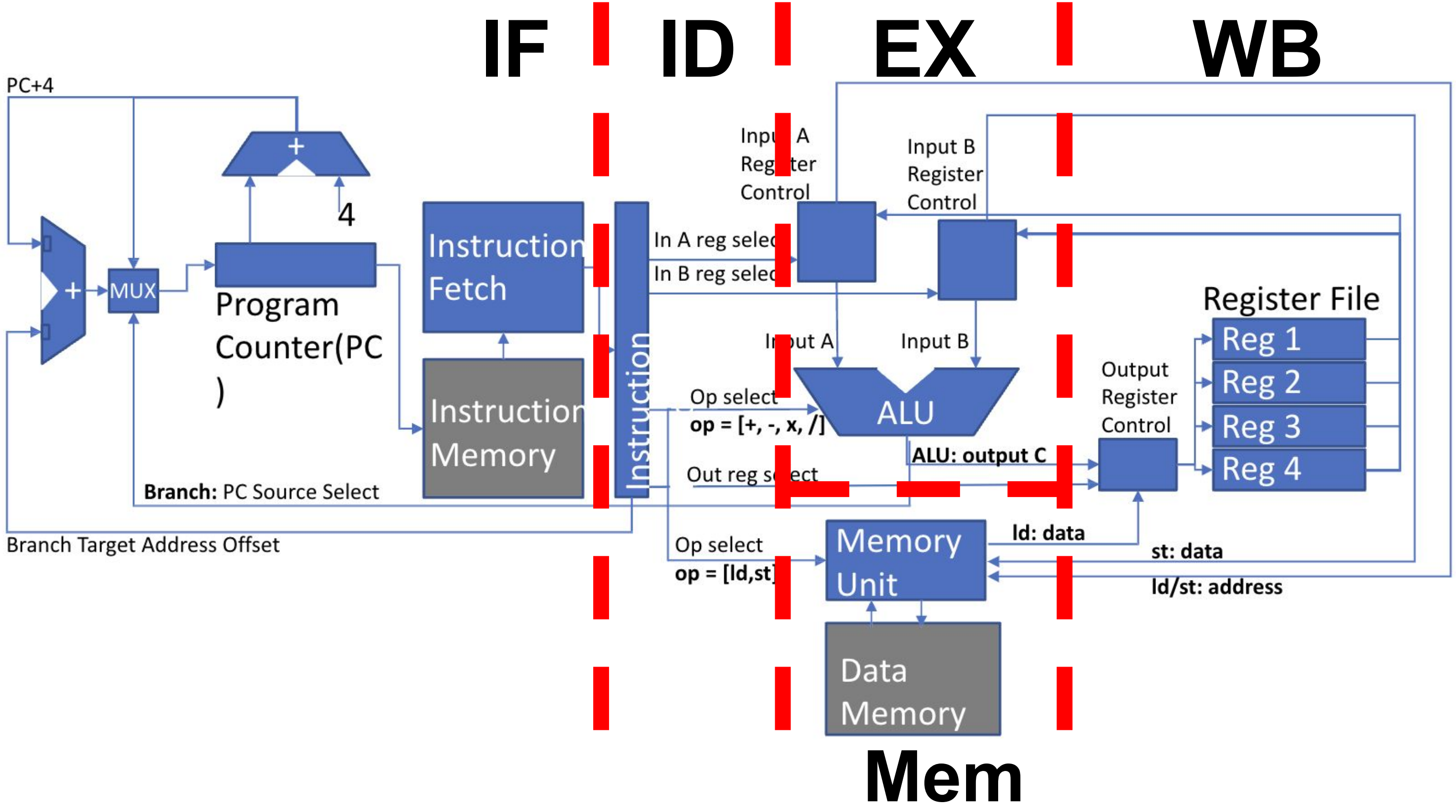
$$t_{\text{overall}} = \underbrace{\left(\frac{(1-f)\times t_{\text{base}}}{1.0}\right)}_{\text{unchanged fraction}} + \underbrace{\left(\frac{f \times t_{\text{base}}}{S}\right)}_{\text{improved fraction}}$$

Amdahl's Law v2



$$S_{\text{overall}} = \frac{1}{\frac{(1-f)}{1.0} + \frac{f}{S}}$$

Basic RISC-V Datapath



Tool Primer

SPEC

- Standard Performance Evaluation Corporation
  - Provides suites of benchmarks
  - SPEC CPU 2017 Integer is just one of them!

SPECspeed®2017 Integer	Language[1]	KLOC[2]	Application Area
600.perlbench_s	C	362	Perl interpreter
602.gcc_s	C	1,304	GNU C compiler
605.mcf_s	C	3	Route planning
620.omnetpp_s	C++	134	Discrete Event simulation - computer network
623.xalancbmk_s	C++	520	XML to HTML conversion via XSLT
625.x264_s	C	96	Video compression
631.deepsjeng_s	C++	10	Artificial Intelligence: alpha-beta tree search (Chess)
641.leela_s	C++	21	Artificial Intelligence: Monte Carlo tree search (Go)
648.exchange2_s	Fortran	1	Artificial Intelligence: recursive solution generator (Sudoku)
657.xz_s	C	33	General data compression

Pin

- A *dynamic binary instrumentation* tool
    - **Dynamic** (as opposed to static). Tool operates at runtime / during execution.
    - **Binary** (what is being instrumented). Tool operates on the raw binary.
    - **Instrumentation**. Insertion of additional code to monitor/analyze the program

- Pin dynamically inserts code (your pintool) while executing a program!

- Count number of branches executed while running a binary!
    - Can’t do this statically!

Pintools

- Pin executes the instrumented binaries
- The pintool defines how the binary is instrumented

```
pin -t obj-intel64/mem-count.so -o curl.stats -- curl cmu.edu
```

pintool

output knob defined by the pintool

binary/program to instrument & its parameters

- The pintool defines
  - Which instructions to instrument
    - loads, stores, branches?
  - How the binaries are instrumented
    - what code runs upon seeing the instruction?
  - How it interacts with the user
    - where/how to write output?
    - how to change its behavior? e.g. making the pintool count loads/stores for specific addresses

Components of a Pintool

1. **Knobs**
    a. Command-line arguments to the Pintool

2. **AddInstrumentFunction**
    a. A function (code) that instruments instructions

3. **AddFiniFunction**
    a. Code that runs when the program finishes

Knobs

- Knobs allow passing command-line arguments to the pintool.
- Useful for dynamically configuring the pintool from the command-line without recompiling
  - e.g. a different branch prediction algorithm

```
KNOB<datatype> KnobName ( KNOB_MODE, KNOB_FAMILY, PREFIX, DEFAULT_VALUE, PURPOSE );
```

`datatype` - type of the data being read from the command line (e.g. `string`, `UINT32`, etc)

`KnobName` - Name for the Knob, will be used to refer to the Knob through the rest of the program

`KNOB_MODE` - Indicates how multiple arguments for the same Knob are handled (e.g. if `KNOB_MODE_WRITEONCE` is used, only the first argument will be read into the Knob)

`KNOB_FAMILY` - Name for the family that the Knob belongs to, you can turn on/off Knobs by their families.

`PREFIX` - The flag that will be used on the command line for this Knob (e.g. `-o` for output file name)

`DEFAULT_VALUE` - Default value of the Knob if nothing specified on command line

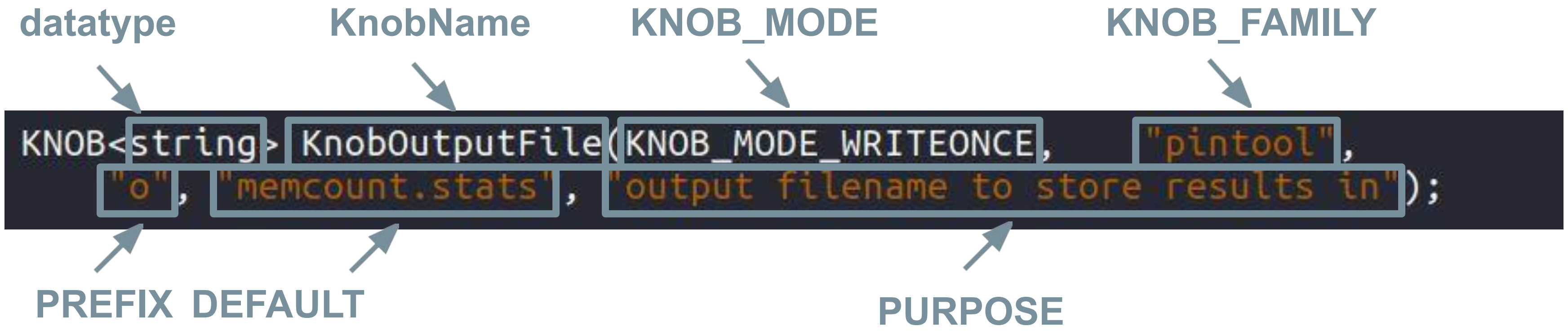
`PURPOSE` - String description that explains what the Knob does

Fields you will use in this class

Knobs - Command Line Arguments

```
KNOB<datatype> KnobName ( KNOB_MODE, KNOB_FAMILY, PREFIX, DEFAULT_VALUE, PURPOSE );
```

Example:



Passing a command-line argument to this Knob:

```
pin -t obj-intel64/mem-count.so -o test.stats -- curl cmu.edu
```

Using Knobs in the pintool

```
KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE,    "pintool",
    "o", "memcount.stats", "output filename to store results in");
```

The `KnobName.Value()` function gives the value stored in the Knob.

```
std::ofstream out(KnobOutputFile.Value().c_str());
```

Here, `KnobOutputFile.Value()` returns the string stored in the Knob. If nothing is specified on the command-line, this will return the default value, which is "memcount.stats" in this case.

You can define similar Knobs to read integer values, e.g. to specify the size of a cache, TLB, etc.

INS_AddInstrumentFunction

Allows you to specify a function that is called for every single instruction

```
// Insert call to function that runs for every instruction
```

```
INS_AddInstrumentFunction(Instruction, 0);
```

Here, `Instruction` is the function that is called for each instruction.

Example `Instruction()` - Instrument Mem. Accesses

Count memory operands in the instruction

If memory operand is read, call the `Load()` function.

If memory operand is written, call the `Store()` function.

```
// Runs for every instruction

VOID Instruction(INS ins, void * v)
{
    UINT32 memOperands = INS_MemoryOperandCount(ins);

    // Instrument each memory operand. If the operand is both read and written
    // it will be processed twice.
    // Iterating over memory operands ensures that instructions on IA-32 with
    // two read operands (such as SCAS and CMPS) are correctly handled.
    for (UINT32 memOp = 0; memOp < memOperands; memOp++)
    {
        const UINT32 size = INS_MemoryOperandSize(ins, memOp);

        if (INS_MemoryOperandIsRead(ins, memOp))
        {
                // map sparse INS addresses to dense IDs
          const ADDRINT iaddr = INS_Address(ins);

          INS_InsertPredicatedCall(
                         ins, IPOINT_BEFORE,  (AFUNPTR) Load,
                         IARG_MEMORYOP_EA, memOp,
                         IARG_UINT32, size,
                         IARG_ADDRINT, iaddr,
                         IARG_END);
        }

        if (INS_MemoryOperandIsWritten(ins, memOp))
        {
          const ADDRINT iaddr = INS_Address(ins);

          INS_InsertPredicatedCall(
                         ins, IPOINT_BEFORE,  (AFUNPTR) Store,
                         IARG_MEMORYOP_EA,memOp,
                         IARG_UINT32, size,
                         IARG_ADDRINT, iaddr,
                         IARG_END);
        }
    }
}
```

We will provide you with the `Instruction()` required for the lab!

Example Instruction() - Instrument Mem. Accesses

```
VOID Load(ADDRINT addr, UINT32 size, ADDRINT instAddr)
```

```
VOID Store(ADDRINT addr, UINT32 size, ADDRINT instAddr)
```

Here,

`addr` **- address being read/written to**

`size` **- size of the data being read/written**

`instAddr` **- address of the instruction itself**

For the labs, these arguments are all you need to know about. You will write functions that use these arguments for solving the labs.

We will provide you with the `Instruction()` required for the lab!

```
// Runs for every instruction

VOID Instruction(INS ins, void * v)
{
    UINT32 memOperands = INS_MemoryOperandCount(ins);

    // Instrument each memory operand. If the operand is both read and written
    // it will be processed twice.
    // Iterating over memory operands ensures that instructions on IA-32 with
    // two read operands (such as SCAS and CMPS) are correctly handled.
    for (UINT32 memOp = 0; memOp < memOperands; memOp++)
    {
        const UINT32 size = INS_MemoryOperandSize(ins, memOp);

        if (INS_MemoryOperandIsRead(ins, memOp))
        {
                // map sparse INS addresses to dense IDs
          const ADDRINT iaddr = INS_Address(ins);

          INS_InsertPredicatedCall(
                          ins, IPOINT_BEFORE,  (AFUNPTR) Load,
                          IARG_MEMORYOP_EA, memOp,
                          IARG_UINT32, size,
                          IARG_ADDRINT, iaddr,
                          IARG_END);
        }

        if (INS_MemoryOperandIsWritten(ins, memOp))
        {
          const ADDRINT iaddr = INS_Address(ins);

          INS_InsertPredicatedCall(
                          ins, IPOINT_BEFORE,  (AFUNPTR) Store,
                          IARG_MEMORYOP_EA,memOp,
                          IARG_UINT32, size,
                          IARG_ADDRINT, iaddr,
                          IARG_END);
        }
    }
}
```

Example Instruction() - Instrument Branch Instructions

Check if instruction is a branch

TRUE for only conditional branches

```
void InstrumentInstruction(INS ins, void *v) {
  if (INS_IsBranch(ins) && INS_HasFallThrough(ins)) {
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR) branch,
        IARG_INST_PTR, IARG_BRANCH_TAKEN, IARG_END);
  }
}
```

```
void branch(ADDRINT pc, bool brTaken)
```

Here,

`pc` **- address of the instruction**

`brTaken` **- whether the branch is taken or not**

We will provide you with the `Instruction()` required for the lab!

INS_AddFiniFunction

Allows you to specify a function that is called <u>after</u> the original binary has completed execution

`INS_AddFiniFunction(Fini, 0);`

```
VOID Fini(INT32 code, VOID * v)
{
    std::ofstream out(KnobOutputFile.Value().c_str());

    //Output your results here

    out.close();
}
```

tmux

- Super powerful “terminal multiplexer”
  - Manage windows, tabs, panes

- For this class, useful for keeping your session alive even after disconnecting
  - SPEC+Pin runs take a long time!

- tmux demo
  - tmux
  - <PREFIX>+d
  - tmux attach

  - <PREFIX> is Ctrl-b by default

Doing the lab

Grading

- Small portion of your lab grade comes from your code.
- **Most of your lab grade comes from your report.**
- Lab report is where you make your work shine!

Phases of a Lab

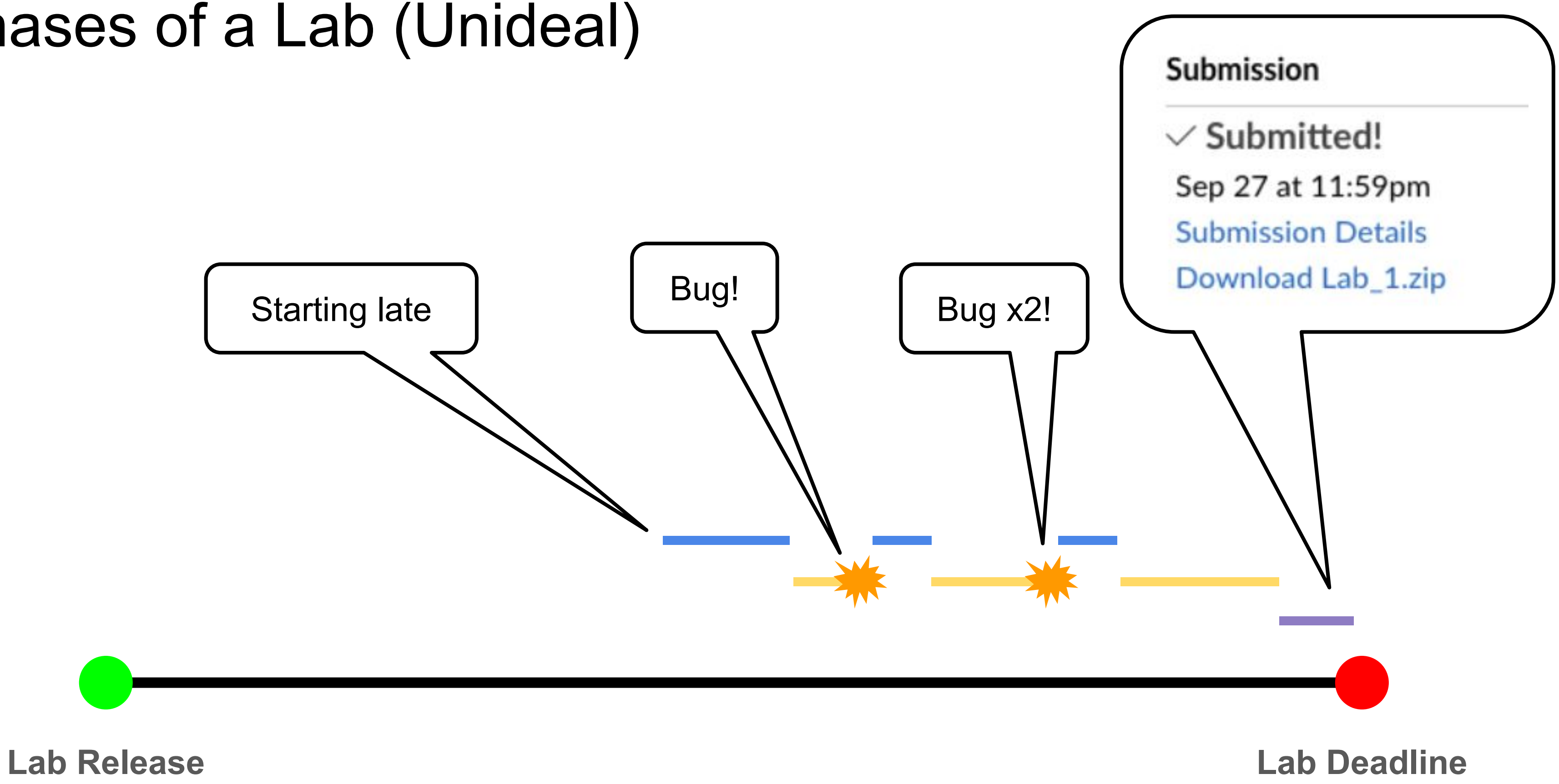
1. Implementation
   a. Implement for correctness first
      i. Ensures correct data + counts towards grading
   b. Then implement stats/counters for data collection

2. Data Collection
   a. Running your implementation to gather results
   b. Varying parameters to see how results change

3. Writing the Report
   a. Majority of lab grade is influenced by the report

Phases of a Lab (Ideal)

Phases of a Lab (Unideal)



Lab Tips

- Aim for correctness the first/second time around
    - Verify with data—”does the data make sense?”
    - Discuss with us (OH/Slack) to check your understanding!

- If you realize too late that your implementation is wrong
    - Don’t worry—mention what is wrong in your report
    - Explain how it affects your results
    - We care more about **report + reasoning** than implementation + results

More Lab Tips

- Ideally, collect data as few times as possible
    - SPEC runs take a long time!
    - Have idea of what data is important
    - Influences the stats/counters you implement
    - Influences the graphs you put in your report

- Don't leave the report to the last minute
    - Aim to finish collecting data a few days before the deadline

Writing the report

Report Structure

1. Introduction
2. Methodology
3. Results
4. Discussion
5. Conclusion

1. Introduction

- **Briefly**, what is this report about?
- What topic are you studying?
- Why is it important?

2. Methodology — *How?*

- **How** did you explore this topic?
- **How** did you do your experiments?
- Design/implementation decisions?
- What assumptions did you make?

3. Results — *What?*

- **What** results did you see?

- Show your data
    - **Warning**: Not literally *all* of your data. More on this later.

- Data should tell a story, show trends

4. Discussion

- **Why** does the data look the way it does?

- Point out the trends, then explain the trends

- Connect the trends to concepts

5. Conclusion

- **Briefly**, what are your takeaways from your experiments?
  - Key learnings
  - Ideal configuration/parameters

- What are the limitations of your experiments?

- Given more time/resources, what would be worth exploring next?

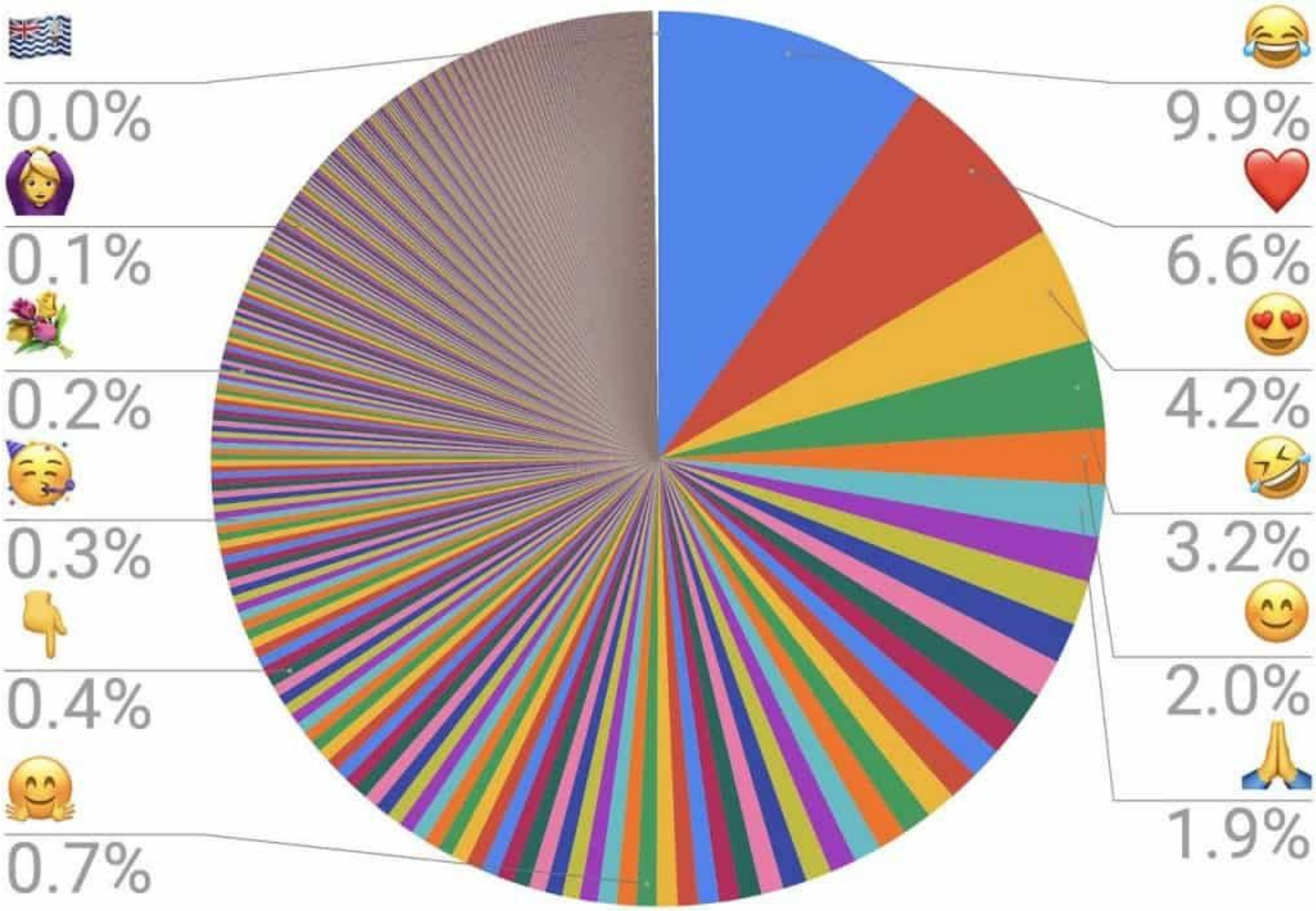
Format & Style

- Make your reports legible and pretty!
- Explicit sections and headers
- Brownie points for LaTeX (Overleaf)!

Presenting the data

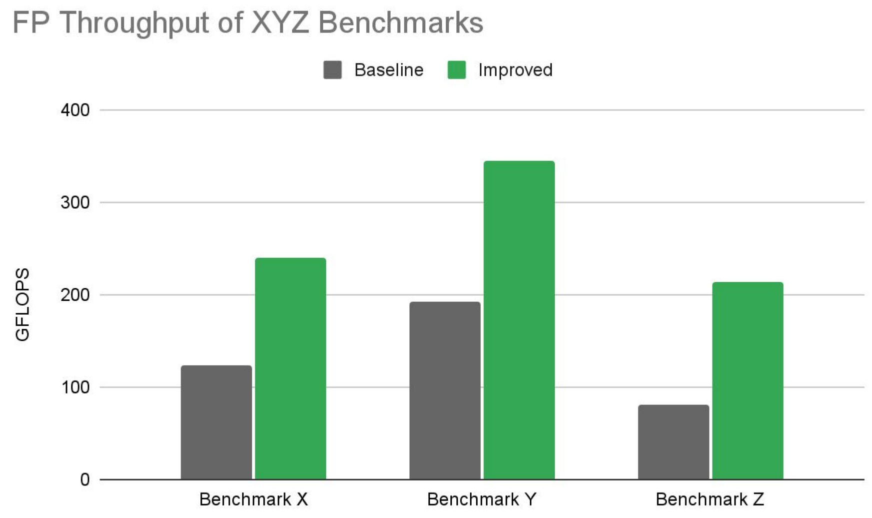
(Graphs and tables)

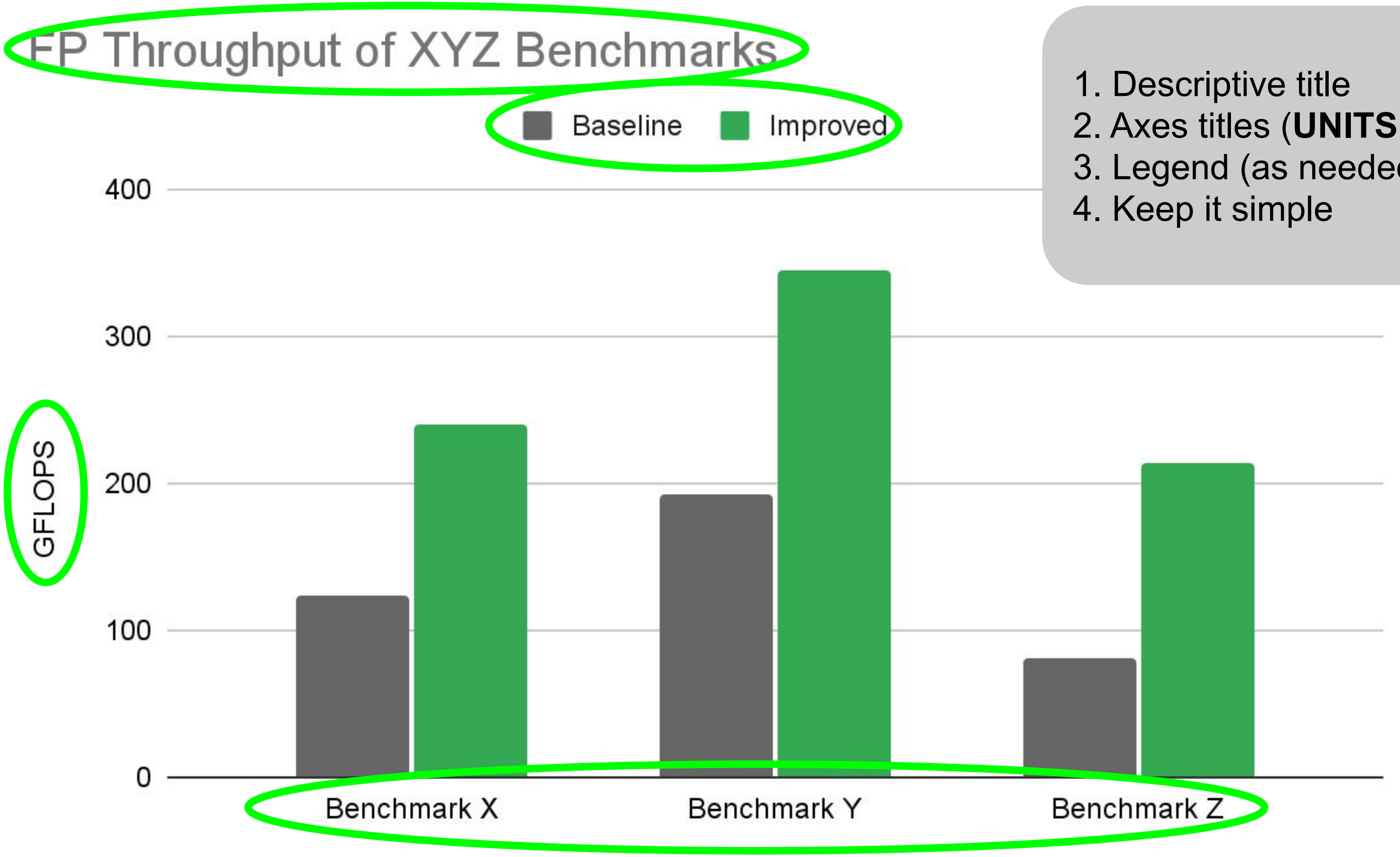
	A	B	C	D	E	F	G	H	I	J	K
1	LYLTY_CAR	DATE	STORE_NBF	TXN_ID	PROD_NBR	PROD_NAM	PROD_QTY	TOT_SALES	LIFESTAGE	PREMIUM_CUSTOMER	
2	1000	43390	1	1	5	Natural Chi	2	6	YOUNG SIN	Premium	
3	1002	43359	1	2	58	Red Rock D	1	2.7	YOUNG SIN	Mainstream	
4	1003	43531	1	3	52	Grain Wave	1	3.6	YOUNG FAI	Budget	
5	1003	43532	1	4	106	Natural Chi	1	3	YOUNG FAI	Budget	
6	1004	43406	1	5	96	WW Origina	1	1.9	OLDER SIN(	Mainstream	
7	1005	43462	1	6	86	Cheetos Pu	1	2.8	MIDAGE SII	Mainstream	
8	1007	43439	1	8	10	RRD SR Slo	1	2.7	YOUNG SIN	Budget	
9	1007	43438	1	7	49	Infuzions S	1	3.8	YOUNG SIN	Budget	
10	1009	43424	1	9	20	Doritos Che	1	5.7	NEW FAMIL	Premium	
11	1010	43352	1	10	51	Doritos Me	2	8.8	YOUNG SIN	Mainstream	
12	1010	43448	1	11	59	Old El Paso	1	5.1	YOUNG SIN	Mainstream	
13	1011	43453	1	15	1	Smiths Crin	1	2.9	OLDER SIN(	Mainstream	
14	1011	43435	1	14	49	Infuzions S	1	3.8	OLDER SIN(	Mainstream	
15	1011	43310	1	12	84	GrnWves Pl	2	6.2	OLDER SIN(	Mainstream	
16	1011	43412	1	13	59	Old El Paso	1	5.1	OLDER SIN(	Mainstream	
17	1012	43635	1	17	3	Kettle Sens	1	4.6	OLDER FAN	Mainstream	
18	1012	43539	1	16	20	Doritos Che	1	5.7	OLDER FAN	Mainstream	
19	1013	43528	1	18	93	Doritos Cor	1	3.9	RETIREES	Budget	
20	1013	43531	1	19	91	CCs Tasty (	2	4.2	RETIREES	Budget	
21	1016	43574	1	20	74	Tostitos Sp	1	4.4	OLDER FAN	Mainstream	
22	1016	43625	1	21	63	Kettle 135g	1	4.2	OLDER FAN	Mainstream	
23	1018	43636	1	24	38	Infuzions M	1	2.4	YOUNG SIN	Mainstream	
24	1018	43346	1	22	3	Kettle Sens	1	4.6	YOUNG SIN	Mainstream	
25	1018	43432	1	23	97	RRD Salt & '	1	3	YOUNG SIN	Mainstream	
26	1019	43492	1	25	84	GrnWves Pl	1	3.1	OLDER SIN(	Premium	
27	1020	43328	1	26	19	Smiths Crin	1	2.6	YOUNG SIN	Mainstream	
28	1020	43375	1	27	7	Smiths Crin	1	5.7	YOUNG SIN	Mainstream	
29	1020	43587	1	28	84	GrnWves Pl	1	3.1	YOUNG SIN	Mainstream	
30	1022	43397	1	29	3	Kettle Sens	1	4.6	OLDER FAN	Budget	


0.0%
0.1%
0.2%
0.3%
0.4%
0.7%
9.9%
6.6%
4.2%
3.2%
2.0%
1.9%

Graphs show trends

- Give the reader a comparative sense of scale
  - “My cache is ~2x faster across all benchmarks”
  - **Not** “my cache is 2.386x faster on benchmark X, 1.893x faster on benchmark Y…”



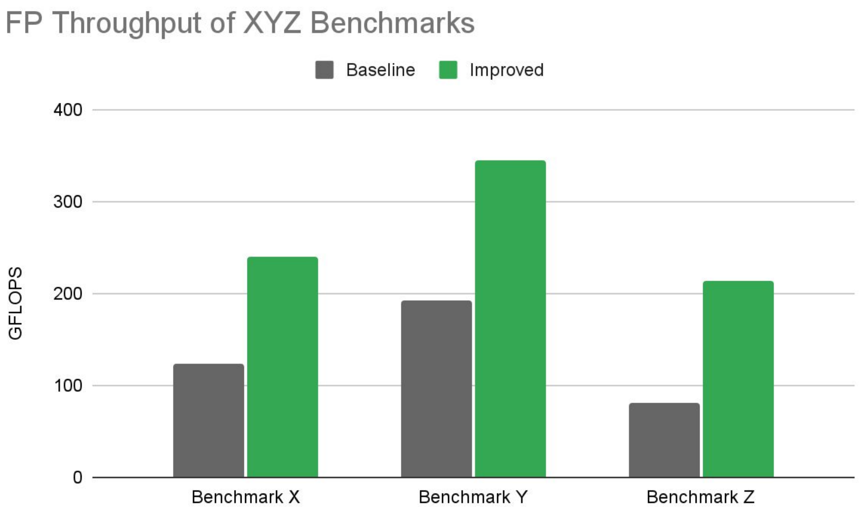


1. Descriptive title
2. Axes titles (**UNITS**)
3. Legend (as needed)
4. Keep it simple

Grouped Bar Chart

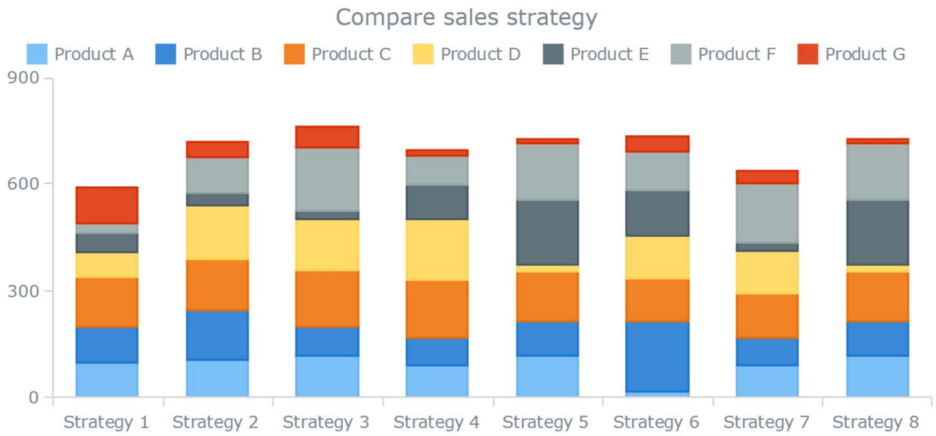
- Categorical data
    - x-axis categorical (benchmarks)
    - legend categorical (baseline vs. improved)

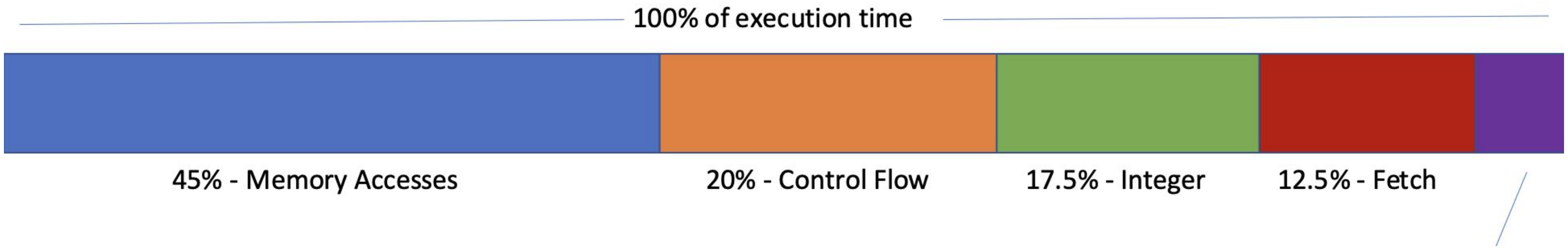
- y-axis continuous (throughput)



Stacked Bar Chart

- Gives further breakdown
- Overall composition

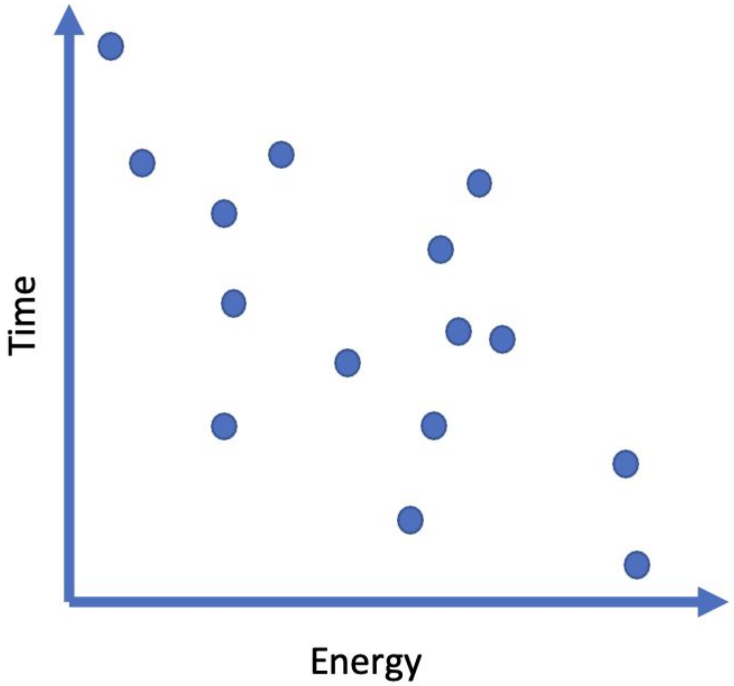




Scatter Plot

- Shows relationship between two continuous variables

- As you vary one, what happens to the other?



Line Graph

- Similar to scatter plot, but emphasizes a trend over the points

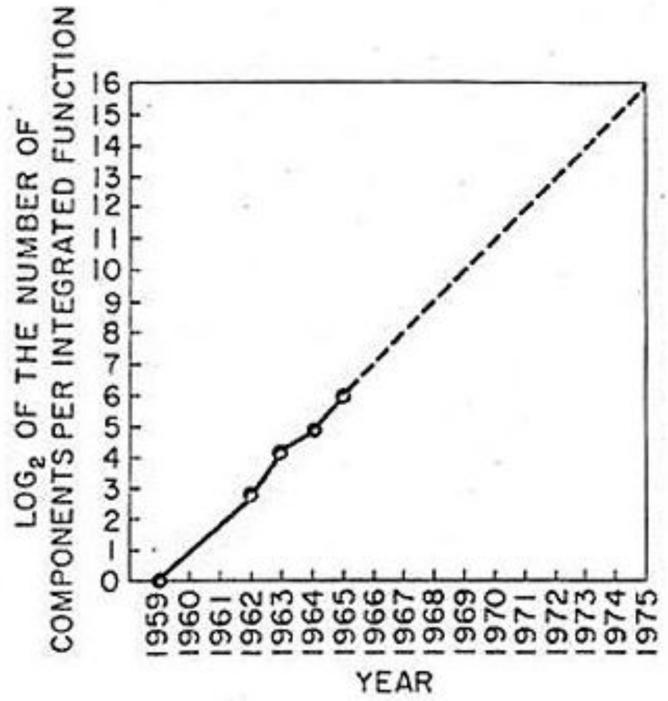


Fig. 2 Number of components per integrated function for minimum cost per component extrapolated vs time.

Tables

- Graphs for trends, tables for quantities
    - *Most of the time, trends are more useful than quantities!*

- Keep tables short and brief

Speedup		
Benchmark X	**Benchmark Y**	**Benchmark Z**
2.386	1.893	2.102

Making the Graphs

- Get comfortable with a text data format!
  - e.g. CSV, TOML, JSON … your choice!
  - Your code should dump the text data format of your choice
  - Makes it easier to graph your data

- Processing the data
  - Write scripts for aggregating your data across multiple output files
  - Write scripts for pre/post-processing your data

- Spreadsheets
  - Google Sheets, Excel

- Plotting Libraries
  - Matplotlib

Lab Report Guide

On course website -> Lab Details -> Lab Resources -> Lab Report Guide

Or use this url:
https://course.ece.cmu.edu/~ece344/course_documents/Lab_Report_Guide.pdf