Max Haughton

The how and why of profiling D code

### What is a profiler Nullius in verba

- (machine) code that makes up the program.
- The time taken is not the only thing that can be profiled: Memory may be solely interested in contention.

• Produces a report about how the program spends its time for a given input. Although the dependence on the input is trivial, it is worth keeping in mind as performance is often dominated by patterns not inherent to the

• Information produced is typically fairly dumb. The profiler can't tell you to change algorithm, but can tell you how to make your algorithm faster.

allocation may be more important, or in a multithreaded environment one

### When to profile? Nullius in verba

- that can be resolved.
- of the speedup.
- Although profiling is typically (and mostly should be) reserved for program was designed around.

• The program is running slowly -> Profiling should hopefully reveal issues

• There is an (informal) Pareto Principle involved: 20% of the work yields 80%

diagnosing performance issues, it can also yield important understand of a program: A program may be bug free, but misunderstood in that the ratios of different work within the program may be different to the assumptions the

• Benchmarking and profiling aren't the same thing, but it's worth mentioning that doing the former regularly and the latter every now and again can do wonders in terms of keeping track of just how "fast" your code really is.



## Profiler taxonomy

What kinds of profilers are available?

- go about data acquisition?
- YMMV.
- Sampling: Interrupt process, collect data, keep going, repeat. Much lower overhead.
- More feature complete sampling profilers (e.g. VTune) provide APIs for instrumentation and tracing. Other's follow a hybrid approach e.g. Tracy.

• After identifying what we want to measure (e.g. time or memory), how to do we

• Instrumentation: Add a hook to measure the data we're interested in, which is then stored and processed later. The naive approach (instrument everywhere), can lead to very complete but contextless data. And potentially a lot of overhead,

Emulation: Run the program in an emulated environment, collect very fastidious but synthetic data. Valgrind's callgrind and cachegrind are famous examples.

#### Samping what exactly While better than nothing, -profile leaves much to be desired. • We mentioned sampling before, but what will we sample.

- We need to measure the quantity we're interested in, obviously, but we also need to save where we got that data.
- Saving the instruction pointer is easy, but we need the full context so the callstack is superior.

#### Caveman profiling Jesus' Blood Never Failed Me Yet

- Just use your debugger, get a few backtraces.
- on).
- blocked on some device, service, or library then this can be very useful.

#### • Before looking at a *true* scotsman's profiler let's reject modernity and return to monke basics by thinking about how we could approximate a profiler with a humble debugger.

• A few samples and your brain can go a very long way, however the practical utility of this method is very limited. Data acquisition is annoying, data processing even more so.

• This method's utility is much better on program with some notion of progress e.g. a simple % completion metric or even verbose output (so you know what is actually going

If, however, the program is slow enough to be considered faulty in some way, i.e.



#### Frame pointers Something to keep in mind

- to walk the call stack. First part is easy, second part not so much.
- register at the expense of debugability.
- Debug info now means these frame pointers are not necessary, however a profiler might get this wrong (so worth keeping in mind)
- Always profile with debug symbols if possible.
- pointer=all" for ldc

# • Sampling the call stack requires getting the instruction pointer, and being able

• On X86, omission of the frame pointer can let the compiler play with one more

If needed, use "-gs" for dmd, "-fno-omit-frame-pointer" for GDC, or "--frame-

### A simple instrumenting profiler Profiling a contrived example using dmd's builtin profiler

```
int add(int x, int y)
                           The compiler turns
                             LHS into RHS
 return x + y;
                               (simplified)
```

The pair of functions (prolog and epilog, to be clear) are in druntime, they collect timing information, which is then stored and printed upon program exit.

The data is outputted to a file called trace.log, or if this file already exists the new data will be merged. For this reason, delete the log file on each run.

See D & Digital Mars website for history of the feature.

```
int add(int x, int y)
 char[] loc = "add";
 trace_pro(loc);
 const res = x + y;
 _c_trace_epi();
  return res;
```

## s that it then?

While better than nothing, -profile leaves much to be desired.

- Only instrumented functions are seen in the profile. This is potentially catastrophic for some programs, e.g. IO bound workloads, calls into nonroot module functions etc
- The feature makes a valiant attempt to sample the call graph, but not the call stack. This is not ideal - more on that next.
- Data is only collected at the function-level.
- Potentially very high overhead, especially if a function.
- We can do better.

#### Profiling allocations with dmd Don't fear the reaper - how to easily profile GC allocations Instrumentation is not great for profiling time, but for profiling allocations

- it's very useful.
- not an issue.
- Compile with "-profile=gc"
- helpful to know the exact type of an allocation.

Overhead? Recorded data unaffected, allocation is slow anyway so time

• Heap profiling does not have to be integrated with the language, but it's

#### **GC profiling example** Dirty deeds done dirt cheap (But still really useful)

- 1. Compile with -profile=gc
- 2. Run program
- 3. Inspect the log (located at profilegc.log)

bytes allocated, allocations, type, function, file:line1840050 ubyte[] D main alloc.d:11160050 alloc.Data

Really useful? Allocations can (and usually are) very slow - a good malloc implementation on the hot path might still be hundreds of instructions (note: instructions, this measure doesn't even take the cache into account!)



Visual Studio Code integration! (Say thanks to WebFreak)

#### **Beyond contrived examples.** Towards "real" code

- Basics first: You can't really optimize add, and it was probably inlined anyway.
- Programming practice can lead to a relatively obscure mapping of name to task. OOP: horses vs. chickens.
- Let's look at a profile of dmd compiling hello world.

# An informative iota

#### A small profile of non-trivial program

- int dmd.mars.tryMain(ulong, const(char)\*\*, ref dmd.globals.Param)
  - + 26.75% Module::importAll
  - 9.10% DsymbolSemanticVisitor::visit DsymbolSemanticVisitor::visit
    - DsymbolSemanticVisitor::visit
    - 3.55% DsymbolSemanticVisitor::visit
      - 2.10% DsymbolSemanticVisitor::funcDeclarationSemantic typeSemantic + dmd.mtype.Type dmd.typesem.merge(dmd.mtype.Type)
- Collected using call stack sampling.
- is basically lost without CSS.
- great.

dmd.mtype.Type dmd.typesem.typeSemantic(dmd.mtype.Type, ref const(dmd.globals.Loc), dmd.dscope.Scope\*).visitFunction(dmd.mtype.TypeFunction)

#### Due to a quirk of c++ demangling and the visitor pattern, this information

#### Alternative is just a list of functions that appeared in samples, which isn't



#### So, what are we looking for? We know what we want, we don't know (yet) how to get it. • Low overhead - zero is impossible, but we can get close.

- Call stack sampling is a must-have.
- Source level profiling is very nice to have (but requires debug info)
- Cross-platform?

• Full complement of information from the hardware (more on that later).

#### Profiling with perf The Second Best Secret Agent In The Whole Wide World

- platform agnostic way, perf is the canonical frontend to it.
- presented. Page faults, for example, are a very handy thing to keep an eye on.
- optimizers etc.). A jack of all trades, master of some.
- some architectures are more equal than others.
- use in anger unless you work for Netflix.

• The Linux Kernel exposes a subsystem called perf event to read performance counters in a mostly

• perf list reveals more than just measuring time. A long list of hardware and software events are

• perf has a lot of functionality out of the box, also serves as the basis for several other tools (profilers,

• Although the tool aims to be platform agnostic (and for the first 80% of performance problems it is),

• perf is also part of (and can act as a frontend for) a rich set of tracing utilities covering both userspace and the kernel itself. These are a talk all by themselves (more for profiling systems than code), so I won't cover them. See Brendan Gregg's excellent website to learn more about it than you'll ever get to

#### Perf workflow A basic recipe for using perf

- collected
- very ichthyomorphically) with collected data.

1. Start with "perf record -g" to collect data and sample callstacks - use "-e ..." to enable specific performance counters (as elaborated on later)

2. "perf report" will open a fairly nice TUI for you to navigate the data

3. "perf annotate" annotates the assembly and source code (albeit not

#### GUI profilers (and windows...) The Second Best Secret Agent In The Whole Wide World

- not the best for exploring the data.
- graphs), but doesn't quite compare to the following tools.
- uProf.
- aforementioned tools are quite a bit better.

• perf is very good at getting data onto your screen, however the interface is

• There is a pretty good GUI for perf called hotspot (See slide about flame

• CPU designers provide tooling for getting the most out of their processors when profiling: Intel has vTune (amongst many other tools), AMD has

Perf's source annotation tool is functional and useful but a bit 1980s. The

## Flame Graph

#### I can see clearly now

- merged.
- in larger projects.
- generate one standalone, Brendan Gregg has a popular tool.

• A really handy way of looking at a profile's callstack data without going insane. A flame graph is a big stack of boxes: the x-axis indicates frequency, the y-axis is stack depth.

• The x-axis is ordered alphabetically, NOT by time. This is so identical frames can be

• Simple code will likely have a very simple flame graph, the utility of the technique comes

## How to generate one? Available from the Profilers Hotspot, vTune, uProf, a few others. To

## Flame Graph

Function HotSpots

Flame Graph

Call Graph

Metrics

#### Process IDs: [3168393] dmd Zoom Entire Graph

Click on any block in Flame Graph to focus on it's children.

Counters: CYCLES\_NOT\_IN\_HALT

_D3dmd7dsymbol14foreachDsymbolFPSQBf4root5array_T5ArrayTCQCeQCd7DsymbolZQxMDFQvZvZv	QBaZv
DsymbolSemanticVisitor::visit(StructDeclaration*)	
StructDeclaration::accept(Visitor*)	
dsymbolSemantic(Dsymbol*, Scope*)	
DsymbolSemanticVisitor::attribSemantic(AttribDeclaration*)	
DsymbolSemanticVisitor::visit(AttribDeclaration*)	
ParseTimeVisitor <astcodegen>::visit(VisibilityDeclaration*)</astcodegen>	
VisibilityDeclaration::accept(Visitor*)	[ [
dsymbolSemantic(Dsymbol*, Scope*)	ds
DsymbolSemanticVisitor::attribSemantic(AttribDeclaration*)	Ds
DsymbolSemanticVisitor::visit(AttribDeclaration*)	Ds
ParseTimeVisitor <astcodegen>::visit(StorageClassDeclaration*)</astcodegen>	Pa
StorageClassDeclaration::accept(Visitor*)	Vis
dsymbolSemantic(Dsymbol*, Scope*)	
DsymbolSemanticVisitor::attribSemantic(AttribDeclaration*)	
DsymbolSemanticVisitor::visit(AttribDeclaration*)	
ParseTimeVisitor <astcodegen>::visit(ConditionalDeclaration*)</astcodegen>	
ConditionalDeclaration::accept(Visitor*)	
dsymbolSemantic(Dsymbol*, Scope*)	
DsymbolSemanticVisitor::attribSemantic(AttribDeclaration*)	
DsymbolSemanticVisitor::visit(AttribDeclaration*)	
ParseTimeVisitor <astcodegen>::visit(VisibilityDeclaration*)</astcodegen>	
VisibilityDeclaration::accept(Visitor*)	
dsymbolSemantic(Dsymbol*, Scope*)	
D3dmd10dsymbolsem22DsymbolSemanticVisitor5visitMRCQBx7dmodule6ModuleZ_T9_lambda3TCQDf7dsymbol7DsymbolZQBgMFQB	aZv
_D3dmd7dsymbol14foreachDsymbolFPSQBf4root5array_T5ArrayTCQCeQCd7DsymbolZQxMDFQvZvZv	
DsymbolSemanticVisitor::visit(Module*)	
Module::accept(Visitor*)	
dsymbolSemantic(Dsymbol*, Scope*)	
DsymbolSemanticVisitor::visit(Import*)	
Import::accept(Visitor*)	
dsymbolSemantic(Dsymbol*, Scope*)	
DsymbolSemanticVisitor::attribSemantic(AttribDeclaration*)	
DsymbolSemanticVisitor::visit(AttribDeclaration*)	
ParseTimeVisitor <astcodegen>::visit(VisibilityDeclaration*)</astcodegen>	
VisibilityDeclaration::accept(Visitor*)	
dsymbolSemantic(Dsymbol*, Scope*)	
_D3dmd10dsymbolsem22DsymbolSemanticVisitor5visitMRCQBx7dmodule6ModuleZ_T9_lambda3TCQDf7dsymbol7DsymbolZQBgMF0	(BaZv
_D3dmd7dsymbol14foreachDsymbolFPSQBf4root5array_T5ArrayTCQCeQCd7DsymbolZQxMDFQvZvZv	
DsymbolSemanticVisitor::visit(Module*)	
Module::accept(Visitor*)	
dsymbolSemantic(Dsymbol*, Scope*)	
DsymbolSemanticVisitor::visit(Import*)	
Import::accept(Visitor*)	
dsymbolSemantic(Dsymbol*, Scope*)	
D3dmd10dsymbolsem22DsymbolSemanticVisitor5visitMRCQBx7dmodule6ModuleZT9lambda3TCQDf7dsymbol7DsymbolZQBgMFC	(BaZV
_D3dmd7dsymbol14foreachDsymbolFPSQBf4root5array_T5ArrayTCQCeQCd7DsymbolZQxMDFQvZvZv	
DsymbolSemanticVisitor::visit(Module*)	
Module::accept(Visitor*)	
dsymbolSemantic(Dsymbol*, Scope*)	
DsymbolSemanticVisitor::visit(Import*) Import::accept(Visitor*)	
dsymbolSemantic(Dsymbol*, Scope*)	
dsympolsemantic(Usympol", SCODe")	18-7-
	(DaLV
_D3dmd10dsymbolsem22DsymbolSemanticVisitor5visitMRCQBx7dmodule6ModuleZ_T9_lambda3TCQDf7dsymbol7DsymbolZQBgMFC	
_D3dmd10dsymbolsem22DsymbolSemanticVisitor5visitMRCQBx7dmodule6ModuleZ_T9_lambda3TCQDf7dsymbol7DsymbolZQBgMF0 _D3dmd7dsymbol14foreachDsymbolFPSQBf4root5array_T5ArrayTCQCeQCd7DsymbolZQxMDFQvZvZv	
_D3dmd10dsymbolsem22DsymbolSemanticVisitor5visitMRCQBx7dmodule6ModuleZ_T9_lambda3TCQDf7dsymbol7DsymbolZQBgMFC	

Semantic2Visitor::visit(AttribDeclaration*)	D3dmd5parse_T6Pa D3dm		
ParseTimeVisitor <astcodegen>::visit(StorageClassDeclaration*) StorageClassDeclaration::accept(Visitor*)</astcodegen>	_D3dmd5parse_T6ParserT D3dmd D3dmd5parse_T6ParserT D3dmd	1 1 1 1 1	
semantic2(Dsymbol*, Scope*)	D3dmd5parseT6ParserTD3dmd5	1 1 1 1 1 1	
Semantic2Visitor::visit(Module*)	D3dmd5parseT6ParserTD3dmd5p		
Module::accept(Visitor*)	D3dmd5parse T6ParserT D3dmd5par	1   1       /	
semantic2(Dsymbol*, Scope*)	D3dmd5parse_T6ParserT D3dmd5pars	1           /	
Semantic2Visitor::visit(Import*)	D3dmd5parse_T6ParserTS D3dmd5pars	1 11 11 11 1	<b>i i i i</b>
Import::accept(Visitor*)	_D3dmd5parse_T6ParserTSD	1 11 11 11 1	
semantic2(Dsymbol*, Scope*)	_D3dmd5parseT6ParserTSD3dmd5parseD3dmd		
Semantic2Visitor::visit(AggregateDeclaration*)	_D3dmd5parse_T6ParserTSDDD3dmd5parseD3dmd5le	1, 11 11 1,1 1 /	
ParseTimeVisitor <astcodegen>::visit(StructDeclaration*)</astcodegen>	_D3dmd5parse_T6ParserTSQD3dmD3dD3dmd5parse_T6ParserTSQw10astcode		
StructDeclaration::accept(Visitor*)	D3dmd5parse_T6ParserTSQD3dmd5lexer5D3dmd5parse_T6ParserTSQw10astcodege		
semantic2(Dsymbol*, Scope*)	D3dmd5parse_T6ParserTSQw10astcodegen10A D3dmd5parse_T6ParserTSQw10astcodegen		
Semantic2Visitor::visit(AggregateDeclaration*) ParseTimeVisitor <astcodegen>::visit(StructDeclaration*)</astcodegen>	_D3dmd5parse_T6ParserTSQw10astcodegen10A D3dmd5parse_T6ParserTSQw10astcodegen D3dmd5parse_T6ParserTSQw10astcodegen10A D3dmd5parse_T6ParserTSQw10astcodegen		s
StructDeclaration::accept(Visitor*)	_D3dmd5parse_T6ParserTSQw10astcodegen10A D3dmd5parse_T6ParserTSQw10astcodegen		
semantic2(Dsymbol*, Scope*)	D3dmd5parseT6ParserTSQw10astcodegen10AD3dmd5parseT6ParserTSQw10astcodegen	<b>ilii 1</b> 10 111 111 111 1	s
Semantic2Visitor::visit(AttribDeclaration*)			s
ParseTimeVisitor <astcodegen>::visit(VisibilityDeclaration*)</astcodegen>	_D3dmd5parse_T6ParserTSQw10astcodegen10AD3dmd5parse_T6ParserTSQw10astcodegen		S
VisibilityDeclaration::accept(Visitor*)	_D3dmd5parse_T6ParserTSQw10astcodegen10ASTCodegenZQBk13parseDeclDefsMFiPCQCu7dsy		P
semantic2(Dsymbol*, Scope*)	_D3dmd5parse_T6ParserTSQw10astcodegen10ASTCodegenZQBk10parseBlockMFPCQCq7dsymb		S
Semantic2Visitor::visit(AttribDeclaration*)	_D3dmd5parse_T6ParserTSQw10astcodegen10ASTCodegenZQBk13parseDeclDefsMFiPCQCu7dsy		S
ParseTimeVisitor <astcodegen>::visit(StorageClassDeclaration*)</astcodegen>	_D3dmd5parse_T6ParserTSQw10astcodegen10ASTCodegenZQBk10parseBlockMFPCQCq7dsymb		S
StorageClassDeclaration::accept(Visitor*)	_D3dmd5parse_T6ParserTSQw10astcodegen10ASTCodegenZQBk13parseDecIDefsMFiPCQCu7dsy		P
semantic2(Dsymbol*, Scope*) Semantic2Visitor::visit(AttribDeclaration*)	_D3dmd5parse_T6ParserTSQw10astcodegen10ASTCodegenZQBk10parseBlockMFPCQCq7dsymb _D3dmd5parse _T6ParserTSQw10astcodegen10ASTCodegenZQBk13parseDeclDefsMFiPCQCu7dsy		V
ParseTimeVisitor <astcodegen>::visit(ConditionalDeclaration*)</astcodegen>			se Se
ConditionalDeclaration::accept(Visitor*)			Pa
semantic2(Dsymbol*, Scope*)	D3dmd5parse_T6ParserTSQw10astcodegen10ASTCodegenZQBk10parseBlockMFPCQCq7dsymb		Vis
Semantic2Visitor::visit(AttribDeclaration*)	D3dmd5parse_T6ParserTSQw10astcodegen10ASTCodegenZQBk13parseDeclDefsMFiPCQCu7dsy		se
ParseTimeVisitor <astcodegen>::visit(VisibilityDeclaration*)</astcodegen>	D3dmd5parse_T6ParserTSQw10astcodegen10ASTCodegenZQBk11parseModuleMFZPSQCs4root		Se
VisibilityDeclaration::accept(Visitor*)	_D3dmd7dmodule6Module_T11parseModuleTSQBI10astcodegen10ASTCodegenZQBrMFZCQCuQ		Pa
semantic2(Dsymbol*, Scope*)	Module::parse()		Vis
Semantic2Visitor::visit(Module*)	_D3dmd7dmodule6Module4loadFKxSQBc7globals3LocACQBt10identifier10IdentifierQBcZCQCzQC		se
Module::accept(Visitor*)	Import::load(Scope*)		Se
semantic2(Dsymbol*, Scope*)	Import::importAll(Scope*)	_D3dmd5parD3d _D3dmd5parD3dm	Par
Semantic2Visitor::visit(Import*) Import::accept(Visitor*)	_D3dmd6attrib17AttribDeclaration9importAllMRPSQBs6dscope5ScopeZ_T9_lambda4TCQCy7ds D3dmd7dsymbol14foreachDsymbolFPSQBf4root5array_T5ArrayTCQCeQCd7DsymbolZQxMDFQv		se
semantic2(Dsymbol*, Scope*)	AttribDeclaration::importAll(Scope*)		Se
Semantic2Visitor::visit(AttribDeclaration*)	_D3dmd6attrib17AttribDeclaration9importAllMRPSQBs6dscope5ScopeZ_T9_lambda4TCQCy7ds		Te
ParseTimeVisitor <astcodegen>::visit(VisibilityDeclaration*)</astcodegen>	D3dmd7dsymbol14foreachDsymbolFPSQBf4root5array_T5ArrayTCQCeQCd7DsymbolZQxMDFQv		
VisibilityDeclaration::accept(Visitor*)	AttribDeclaration::importAll(Scope*)	_D3dmd5parseT6ParserTSQw10a	_D3
semantic2(Dsymbol*, Scope*)	_D3dmd6attrib17AttribDeclaration9importAllMRPSQBs6dscope5ScopeZ_T9_lambda4TCQCy7ds	_D3dmd5parse_T6ParserTSQw10a	_D3
Semantic2Visitor::visit(Module*)	_D3dmd7dsymbol14foreachDsymbolFPSQBf4root5array_T5ArrayTCQCeQCd7DsymbolZQxMDFQv	_D3dmd5parse_T6ParserTSQw10a	
Module::accept(Visitor*)	AttribDeclaration::importAll(Scope*)	_D3dmd5parse_T6ParserTSQw10a	
semantic2(Dsymbol*, Scope*)	_D3dmd6attrib17AttribDeclaration9importAlIMRPSQBs6dscope5ScopeZ_T9_lambda4TCQCy7ds	_D3dmd7dmodule6Module_T11pa	
Semantic2Visitor::visit(Import*)	_D3dmd7dsymbol14foreachDsymbolFPSQBf4root5array_T5ArrayTCQCeQCd7DsymbolZQxMDFQv AttribDeclaration::importAll(Scope*)	Module::parse() D3dmd7dmodule6Module4loadFK	Stat S
Import::accept(Visitor*) semantic2(Dsymbol*, Scope*)	Module::importAll(Scope*)		stat s
Semantic2Visitor::visit(Module*)	ImportAll(Scope*)	•	Stat S
Module::accept(Visitor*)	D3dmd6attrib17AttribDeclaration9importAllMRPSQBs6dscope5ScopeZT9_lambda4TCQCy7		Com F
semantic2(Dsymbol*, Scope*)	D3dmd7dsymbol14foreachDsymbolFPSQBf4root5arrayT5ArrayTCQCeQCd7Dsyr		stat s
Semantic2Visitor::visit(Import*)	AttribDeclaration::importAll(Scope*)		Sem Se
Import::accept(Visitor*)	Module::importAll(Scope*)		Func Te
semantic2(Dsymbol*, Scope*)	Import::importAll(Scope*)		semantic
Semantic2Visitor::visit(Module*)	Module::importAll(Scope*)		Semantic3
Module::accept(Visitor*) semantic2(Dsymbol*, Scope*)	Import::importAll(Scope*) Module::importAll(Scope*)		Module::a
	Module::importAll(Scope*)		semantic3
D2rt6dmain212_d_run_main2UAAamPUQgZiZ6runAllMFZ9_lambda2MFZv			
D2rt6dmain212_d_run_main2UAAamPUQgZiZ7tryExecMFMDFZvZv			
D2rt6dmain212_d_run_main2UAAamPUQgZiZ6runAllMFZv			
_D2rt6dmain212_d_run_main2UAAamPUQgZiZ7tryExecMFMDFZvZv			
d run main			

\_d\_run\_main main \_libc\_start\_main

[ROOT]



### Reliable data You can tame the chaos



- This is mostly something to keep in mind when benchmarking rather than profiling, but the techniques can nonetheless be useful.

- looks better than previously.
- Should you bother? Depends on who owns the computers your code is going to run on.

Random snapshot of CPU Utilization while writing this slide

• If you are going to go to war based on a profile, you may want to make sure your data is consistent.

• Some metrics are reliable (i.e. an instruction is always an instruction regardless of how long it takes), but other measurements can be dependent on transient (extrinsic) properties of a particular run.

• Power management (laptops especially) is something processor companies beat each other to death over, so the processor will often be extremely aggressively turned down / turned-off to save power).

• Example: If the CPU Freq is turned down relative to the (say) speed of memory, memory latency now

## A little microarchitecture

Nowhere near enough time to go into detail, but enough time to build intuition (hopefully)

Stages of an instruction's execution: Fetch, Decode, Execute, Writeback (ignore memory for now)

IF	ID	EX	WB	IF	-	ID	EX	WB			
Can we overlap them? Yes, in many cases we can overlap their executions											
IF	ID	EX	WB	Instruction 1	We have just decreased our						
	IF	ID	EX	WB	Instru	iction 2	CPI from 4 cycles per instructions to 1!				
		IF	ID	EX	Μ	/B Ins	Instruction 3				

IF	ID	EX	WB		IF		I	)	EX	WB			
Can we overlap them? Yes, in many cases we can overlap their executions													
IF	ID	EX	WB	Inst	truction 1	_		We have just decreased our					
	IF	ID	EX	١	WB	Instru	iction 2		er				
		IF	ID		EX	٨	/B	Instruction 3					

When can't we? Hazards. If we have a write-after-read dependency, then we will have to induce a stall - i.e. wait for the result of instruction 1 so instruction 2 can use it.



## A little microarchitecture

Nowhere near enough time to go into detail, but enough time to build intuition (hopefully)

- We have CPI >= 1, i.e. a scalar processor.
- backend).
- deep speculation possible.
- them)
- monograph on processor architecture.

• If we have CPI < 1, then we have a superscalar processor. A modern processor is very superscalar.

• At the expense of complexity and power usage, we can have a processor be out-of-order: The processor can do independent work independently (ideally in parallel by using a superscalar

• Speculation: In a modern OOO superscalar processor, speculation (doing work based on a guess rather than a guarantee) is the default state of being. Branch Prediction is very successful, making

• See Tomasulo's algorithm for how this actually works, in any computer architecture book (all 2 of

• To learn how these are techniques are actually implemented, Agner Fog produces a detailed

#### The memory hierarchy It's the memory stupid!

- Processors have become much faster than their memory.
- The techniques mentioned previously allow the processor to alleviate some of that, e.g. by doing other work while waiting for memory.
- Despite ever-increasing amounts of memory allocated to programs, memory access remains predictable and local - spatial locality, temporal locality.
- The chip designer has a choice between big and slow, or small and fast. Rather than choosing one, your processor has a memory hierarchy - multiple levels of tradeoffs between latency, bandwidth, and size (and power usage).
- You can do some serious work in the time taken by missing a level of this memory hierarchy, so memory is practically the number one thing to keep an eye on.



## **Historical memory latencies**

#### A classic latency test

- Random selection of old and new processors
- 12900K is the brand new Intel chip.
- Not much has changed. Things have been getting faster, of course, but not quite
- Notice the straight line, then a bump then a (slightly mangled) straight line, these are the gradations between different levels of the (data) caches.



chipsandcheese.com

### There's always leaks - Spectre and Meltdown Famous proof that speculation is not all good.

- Speculation: Great when it's right, what happens when the processor guessed wrong?
- Processor guesses wrong, bails out, end of story? Not quite.
- If we can find a side channel and make the CPU touch it in a speculative/transient operation (nomenclature varies in lit.), we can extract sensitive information.
- We can! Trick the processor into doing an operation speculatively, use that result as an index into an array.
- Time the accesses to this array, do some basic arithmetic, you know which one was transiently accessed, that's the result of the work you made the processor do.
- You can now access any virtual memory. Meltdown (same rough idea) let's you access any physical memory!



#### **Performance counters** Smarter profiling.

- As the processor goes about running your code, it keeps track of the statistics of execution types, stalls, etc.
- Using the techniques mentioned previously, we can relate code to how long it took and why it took so long.
- On Linux, using these is easy (you may need to set your perf\_event\_paranoid setting) just use perf with the "-e" flag as mentioned previously.
- On Windows, you need a profiler like vTune or uProf and a special driver which will be installed with that profiler.
- Example: Port utilization the CPU dispatches work to execution ports, you can use performance counters to track the frequencies of how many your code was able to utilize (a low number is an indication you can't pull in data fast enough).



Figure 2-1. Processor Core Pipeline Functionality of the Ice Lake Client Microarchitecture<sup>1</sup>



## vTune

#### Smarter profiling.

- Thanks to TMAM (Top-down Microarchitecture Analysis Method) we can synthesize all these counters into a cohesive view like shown on the RHS
- This is an example of memory bound code.
- Only vTune is able to do this well.
- Note that 65% of memory accesses are missing the cache entirely, so we have 0-ports exercised most of the time.

Elapsed Time<sup>®</sup>: 43.609s  $\odot$ Clockticks: Instructions Retired CPI Rate <sup>(2)</sup>: MUX Reliability <sup>(2)</sup>: Retiring <sup>(2)</sup>: Front-End Bound <sup>®</sup> Bad Speculation <sup>(2)</sup>: Back-End Bound <sup>(2)</sup> Memory Bound L1 Bound <sup>(2)</sup> L2 Bound <sup>③</sup> O L3 Bound <sup>®</sup> Contest Data Sha L3 Later SQ Full ORAM Boui Memory Memory Store Bound O Core Bound <sup>(2)</sup>: Divider <sup>(2)</sup>: O Port Utilizat O Cycles of Cycles of Cycles of O Cycles of C Vector C Average CPU Frequ Total Thread Count Paused Time <sup>(2)</sup>:

	1,209,171,600,000			<b>`</b>	
ed:	321,081,600,000				The metric value is hi
	3.766 🖻				This can indicate that significant fraction of
:	1.000				execution pipeline slo
	13.0%	of Pipeline Slots			could be stalled due t
®:	3.4%	of Pipeline Slots			demand memory load stores. Use Memory
D:	0.3%	of Pipeline Slots	76.6% - Memory Bound		Access analysis to have
? <mark>:</mark>	83.3% 🖻	of Pipeline Slots			the metric breakdowr
nd <sup>@</sup> :	76.6% 🖻	of Pipeline Slots			memory hierarchy,
?:	4.7%	of Clockticks			memory bandwidth information, correlation
?:	1.0%	of Clockticks			memory objects.
?:	13.1% 🏲	of Clockticks			<u>`</u>
sted Accesses <sup>®</sup> :	0.3%	of Clockticks	13.0% - Retiring		
haring <sup>(†)</sup> :	1.6%	of Clockticks			
ency 🔊:	8.1% 🏲	of Clockticks		/	
1.0:	0.0%	of Clockticks	-	μPipe	
und <sup>@</sup> :	65.3% 🖻	of Clockticks	This diagram represents ineffic		as a nine with an output
ry Bandwidth <sup>@</sup> :	32.0% 🖻	of Clockticks		y" ratio: (Actual Instructions Re	
ry Latency 💿:	58.7% 🖻	of Clockticks	Instruction Retired). If there	are pipeline stalls decreasing t	he pipe efficiency, the p
nd ®:	0.0%	of Clockticks		shape gets more narrow.	
	6.7%	of Pipeline Slots			
	0.0%	of Clockticks			
ation <sup>®</sup> :	7.3%	of Clockticks			
of 0 Ports Utilized <sup>(*</sup> ):	71.4%	of Clockticks			
of 1 Port Utilized <sup>(*</sup> ):	11.2%	of Clockticks			
of 2 Ports Utilized <sup>(*</sup> ):	8.2%	of Clockticks			
of 3+ Ports Utilized <sup>(*)</sup> :	9.0%	of Clockticks			
Capacity Usage (FPU) 💿	: 25.0% 🏲				
quency <sup>®</sup> :	4.4 GHz				
nt:	11				
	Os				





#### **VTune** Pause the video and take a look

ntel VTune Profiler						- 0	>		
τ Project Navigator + 🗅 🛃	Welcome × r003ue ×								
▶ ■ chh	Microarchitecture Exploration Microarchitecture Exploration - 🕐 📫				I	<b>TEL VTUNE PR</b>	OFILE		
▼ ■ dtest2	Analysis Configuration Collection Log Summary Bottom-up Event Count Platform multiply.c ×								
r000ps	Source Assembly II = $\delta \overline{r}$ $\delta \delta \phi$ $\delta \phi$								
r001ue		1					Loca		
r002ue	Source Line A Source	敊 Clockticks	Instructions Retired	CPI Rate	Retiring 💿	Front-End Bound	» B		
> r003hs	11 Intel's prior express written permission.								
r004hs									
r005ue	13       No license under any patent, copyright, trade secret or other         14       intellectual property right is granted to or conferred upon you by								
r006ue	15     disclosure or delivery of the Materials, either expressly, by								
r007ps	16 implication, inducement, estoppel or otherwise. Any license under such								
r008ps	17 intellectual property rights must be express and approved by Intel in								
r009ps	18 writing.								
r010ps	<u>19</u> */ 20								
r011hs	20 21 // matrix multiply routines								
r012hs	22 #include "multiply.h"								
r013hs	23								
r014macc	24 #ifdef USE_MKL								
r015hs	25 #include "mkl.h"								
r016macc	26								
✓ ■ raybencher	27 28 #ifdef USE THR								
r000ps	<pre>29 void multiply0(int msize, int tidx, int numt, TYPE a[][NUM], TYPE b[][NUM], TYPE c[][NUM], TYPE t[][NUM])</pre>								
r001ue r002ue	30 {								
r003ue	31 int i, j, k;								
r003ue r004ue	32 22 // Prote control implementation								
r004ue	33     // Basic serial implementation       34     for(i=0; i <msize; i++)="" td="" {<=""></msize;>								
■ sample (matrix)	34     For(1-0; 1 <msize; 1++)="" th="" {<="">       35     for(j=0; j<msize; j++)="" td="" {<=""></msize;></msize;>								
r000hs	36     for(k=0; k <msize; k++)="" td="" {<=""></msize;>								
r001ue	37 c[i][j] = c[i][j] + a[i][k] * b[k][j];								
r002ps	38 }								
r003ue	39 }								
	$40$ }								
	42								
	43 void multiply1(int msize, int tidx, int numt, TYPE a[][NUM], TYPE b[][NUM], TYPE c[][NUM], TYPE t[][NUM])								
	44								
	45 int i,j,k;								
	46								
	47 // This naive implementation of matrix multiply contains an inefficient memory access pattern. 49 // Each iteration of the inner large strides access the full width of a new from matrix this.								
	48       // Each iteration of the inner loop strides across the full width of a row from matrix 'b'         49       // because the iterator 'k' is used in the first dimension of b[k][j].								
	50 // This leads to bad cache reuse and significant memory stalls.								
	51 // Use Microarchitecture and Memory access analysis to estimate an impact of this performance bottleneck.								
	52 // See the 'multiply2' function implementation to overcome the issue.								
	53 for(i=tidx; i <msize; i="i+numt)" td="" {<=""><td></td><td></td><td></td><td></td><td></td><td></td></msize;>								
	54 for(j=0; j <msize; j++)="" {<br="">55 for()=0: k(msize: k++) (</msize;>	9 706 600 000	.,				2%		
	55         for(k=0; k <msize; k++)="" th="" {<="">           56         c[i][j] = c[i][j] + a[i][k] * b[k][j];</msize;>	8,706,600,000 1,178,927,400,000					2% )%		
	57 }	8,635,200,000					2%		
	58 }	0,000,200,000	0 0				2%		
	59 }								
	60 }								
	61								
	62 void multiply2(int msize, int tidx, int numt, TYPE a[][NUM], TYPE b[][NUM], TYPE c[][NUM])								
	$\frac{03}{64}$ intit k.								
	64 int i,j,k; 65								
	66     // This implementation interchanges the 'j' and 'k' loop iterations.								
	67     // The loop interchange technique removes the bottleneck caused by the inefficient								
				İ					

#### This example is C, works absolutely fine with D bar demangling.

#### vTune Threads

	D: 🕇 = 🖍	r			1s		1.5s		2s	
Thread	Thread (TID: 18160)		and the second				and a second second	and the second second		
Ę	Thread (TID: 24796)									
	Thread (TID: 26984)									
	Thread (TID: 20272)									
	Thread (TID: 23124)									
	Thread (TID: 27508)		-							
	Thread (TID: 14144)									
	Thread (TID: 22244)									
	Thread (TID: 17816)									
	Thread (TID: 1996)									
	Thread (TID: 28276)									
	Thread (TID: 24064)									
	Thread (TID: 27848)									
	Thread (TID: 28484)									
μ	CDU T	_								
_	CPU Time									
	System Bandwidth 34.0	000								_
F	FILTER 🝸 100.0% 🏹	;	Process Any Process	Thread Any 1	Thread	~	Module Any M	odule	~	Utilizat

#### 2.5s 3s 3.5s 4s 4.5s Thread 🗹 🔙 Running 🛃 📥 CPU Time 🗹 CPU Time 🗹 📥 CPU Time System Bandwidth 🖦 Total, GB/sec ✓ Inline Mode Show inline functions ✓ n Any Utilization V Call Stack Mode User functions + 1 V Loop Mode Functions only

#### ~ 🗹 📥 Spin and Overhead Ti... Clocktick Sample 🗹 📥 Spin and Overhead Ti...

#### **VTUNE** Latency

#### Scheme Latency Histogram

This histogram shows a distribution of loads per latency (in cycles).



						_			
		1				1		I	
180	200	220	240	260	280	300	320	340	360
	Latency								



# Coz - A causal profiler

A very different way of doing things

- Previously we emphasized sampling profilers as the way to go.
- Coz is a bit different. It performs performance experiments. We try to measure how much a given line contributes to the speed of a program by slowing the rest of the program down.
- Mainly intended for profiling multithreaded code.

# Tracy - A frame profiler

- Tracy is a profiler intended for profiling video games
- It's linked with the program being profiled, and is activated on a per-frame basis
- Data is collected externally via a socket
- Cross-platform.
- Rapidly gaining features.
- Game development exposes it to new ideas in concurrency and parallelism



## The end

• Questions?