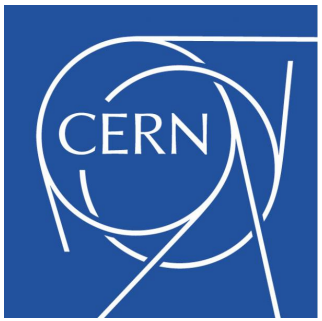
Measuring code performance

Monir Hadji

9th LHCb Computing Workshop

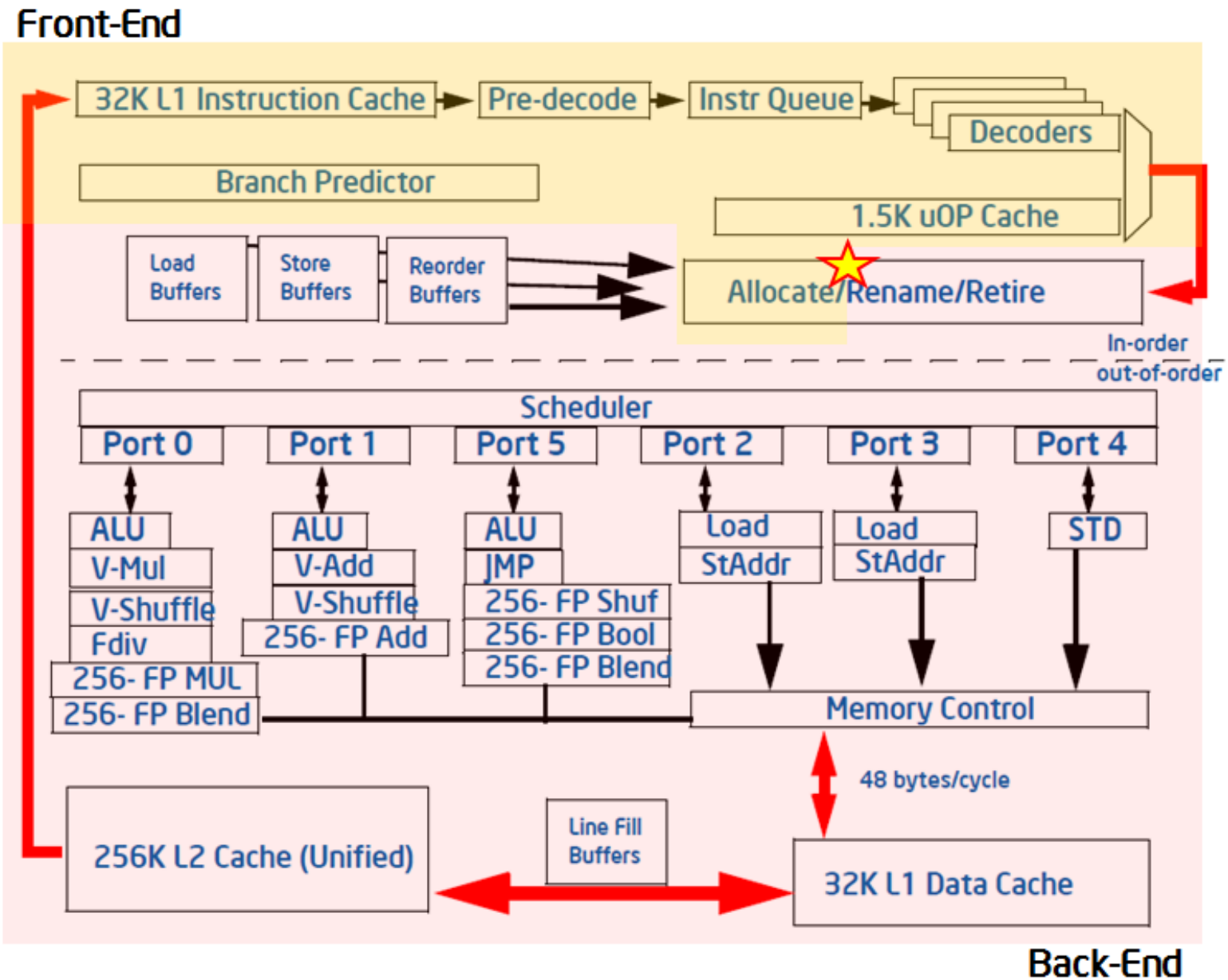
June 19, 2017



Overview

Overview

Architecture



Get useful info about OS and CPU

Get configuration of system values

```
getconf -a | grep CACHE # grep what you want

# Size unit: Bytes
LEVEL1_ICACHE_SIZE                 32768
LEVEL1_ICACHE_ASSOC                8
LEVEL1_ICACHE_LINESIZE             64
LEVEL1_DCACHE_SIZE                 32768
LEVEL1_DCACHE_ASSOC                8
LEVEL1_DCACHE_LINESIZE             64
LEVEL2_CACHE_SIZE                  262144
LEVEL2_CACHE_ASSOC                 8
LEVEL2_CACHE_LINESIZE              64
LEVEL3_CACHE_SIZE                  20971520
LEVEL3_CACHE_ASSOC                 20
LEVEL3_CACHE_LINESIZE              64
LEVEL4_CACHE_SIZE                  0
LEVEL4_CACHE_ASSOC                 0
LEVEL4_CACHE_LINESIZE              0
```

Get informations about your CPU

```
lscpu

cat /proc/cpuinfo
# you can then check details of the model on ark.intel.com
```

Overview

Outline

Sampling

Principle

- Snapshot of the state of process at several moments depending on the tool
- Get data from kernel and Performance Monitor Counters (PMC)
  - "Counting" counters
    - ↪ Just counting the numbers of events that occur
  - Sampling counters
    - ↪ Set up to generate an interrupt every N events
    - ↪ Interruption comes with some metadata like instruction address, thread id, ...
- Give statistical information

Emulation

Principle

- Layer between process and OS
- Process run on a **virtual CPU** provided by the framework
- Insert its own instructions to do advanced debugging and profiling
- No access to registers provided by **P**erformance **M**onitor **U**nit (PMU) to get data, just by software analysis and execution
- Some results could be far from real execution
  ↪ especially for **B**ranch **P**redictor **U**nit (BPU)
- Instruction granularity

Outline

Valgrind tool suite

emulation

Brief

- GPL programming tool suite
- A lot of tools
  - ↪ memcheck: memory checker (default tool when you run valgrind)
  - ↪ cachegrind: cache and branch prediction profiler
  - ↪ callgrind: profile calls, jump, CPU consumption
  - ↪ helgrind: detecting race conditions
  - ↪ ...
- Cachegrind is a kind of subset of Callgrind
  - ↪ Can do call profiling and cache/branch profiling at the same time with callgrind

How it works

- Emulation on virtual CPU

Usage

emulation

Recording

```
# I want cache and calls profiling by instruction granularity
  valgrind --tool=callgrind --cache-sim=yes --dump-instr=yes ./a.out
# this command ↗ create a callgrind.out.PID.THREAD_NUMBER
```

Reporting

```
# see data with the GUI kcachegrind by giving report file to him
  kcachegrind callgrind.out.PID.THREAD_NUMBER
```

More info

```
man valgrind
```

Outline

Linux-perf

sampling

Brief

- Initially, it's a module of linux kernel
  - ↪ Accessing counters of the CPU Performance Monitor Unit
- CLI tool using this module : `perf`

How it works

- Execution granularity with the CLI tool (just a global summary of the execution)
  - ↪ but, you can use linux-perf API directly on your code
- Get data from PMU counters

`https://github.com/torvalds/linux/blob/master/tools/perf/design.txt`

Usage

sampling

Basic profile (cycles, instructions)

```
perf stat ./prog
```

A little more advanced (add memory profile) sufficient in average

```
perf stat -ddd ./prog
```

More info

```
perf help
perf help stat
```

Outline

intel® VTune™ Amplifier

sampling

Brief

- Intel profiler
- Allows to profile time, cycles, PMC events, ...

How it works

- Try to do instruction granularity with sampling
- Get data from PMC sampling counters (Event Based Sampling)
  - ↪ Interrupt when number of events greater than **S**ample **A**fter **V**alue
  - ↪ then, increment amount of sample

intel® VTune™ Amplifier

sampling

Count L2 cache miss with a SAV of 2000

When the 2000th cache miss occurs

- ↪ Interruption
- ↪ Get current Instruction Pointer
- ↪ Associate all of these 2000 L2 cache misses to this single instruction
  - ↪ unlikely these cache misses have been caused by this single instruction
  - ↪ could introduce bias by assigning a large amount of events to a single instruction
  - ↪ statistically it's ok if the execution is long enough

Usage

sampling

To be able to use Intel tools

```
source /cvmfs/projects.cern.ch/intelsw/psxe/linux/all-setup.sh      # for bash; sh
source /cvmfs/projects.cern.ch/intelsw/psxe/linux/all-setup.csh     # for csh; tcsh
```

Recording with CLI (can do it with the GUI as well)

```
amplxe-cl -collect general-exploration -- ./prog
# this command ↗ create a directory following this default pattern :
#         r@@@{at}, where:
#    @@@ is an increasing numeric sequence automatically assigned by amplxe-cl
#    {at} is an abbreviation of the analysis type.

# See other available collections
amplxe-cl -help collect
```

Reporting with GUI (can do it with amplxe-cl and -report option way as well)

```
amplxe-gui <path to r@@@{at}>
```

Overview

Outline

Time

- We want to be fast
- Where do we spend our time?
  - in which function?
  - called by whom?
  - which instructions take time?

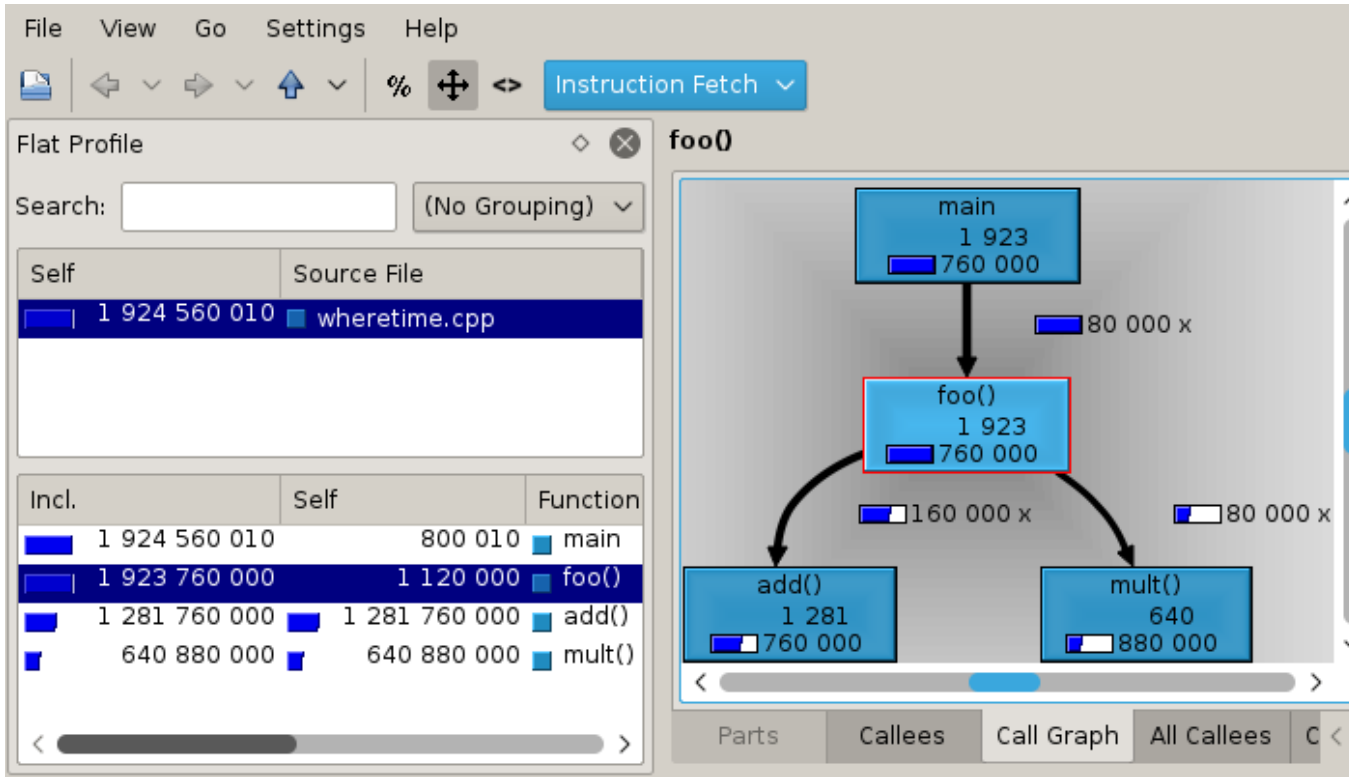
Code example

```
1  int add() {
2    int val = 0;
3    for(int i = 0; i < 800; ++i)
4      val += i;
5    return val;
6  }
7
8  int mult() {
9    int val = 1;
10   for(int i = 0; i < 800; ++i)
11     val *= i;
12   return val;
13 }
14
15 int foo() {
16   return add() + add() + mult();
17 }
18
19 int main(void) {
20   int dum = 0;
21
22   for(int i = 0; i < 80000; ++i)
23     dum += foo();
24
25   return 0;
26 }
```

Callgrind and kcachegrind

```cpp
#include <valgrind/callgrind.h>

int add() {
  int val = 0;
  for(int i = 0; i < 800; ++i)
    val += i;
  return val;
}

int mult() {
  int val = 1;
  for(int i = 0; i < 800; ++i)
    val *= i;
  return val;
}

int foo() {
  return add() + add() + mult();
}

int main(void) {
  int dum = 0;

  CALLGRIND_START_INSTRUMENTATION;
  for(int i = 0; i < 80000; ++i)
    dum += foo();
  CALLGRIND_STOP_INSTRUMENTATION;

  // other computation
  return 0;
}
```

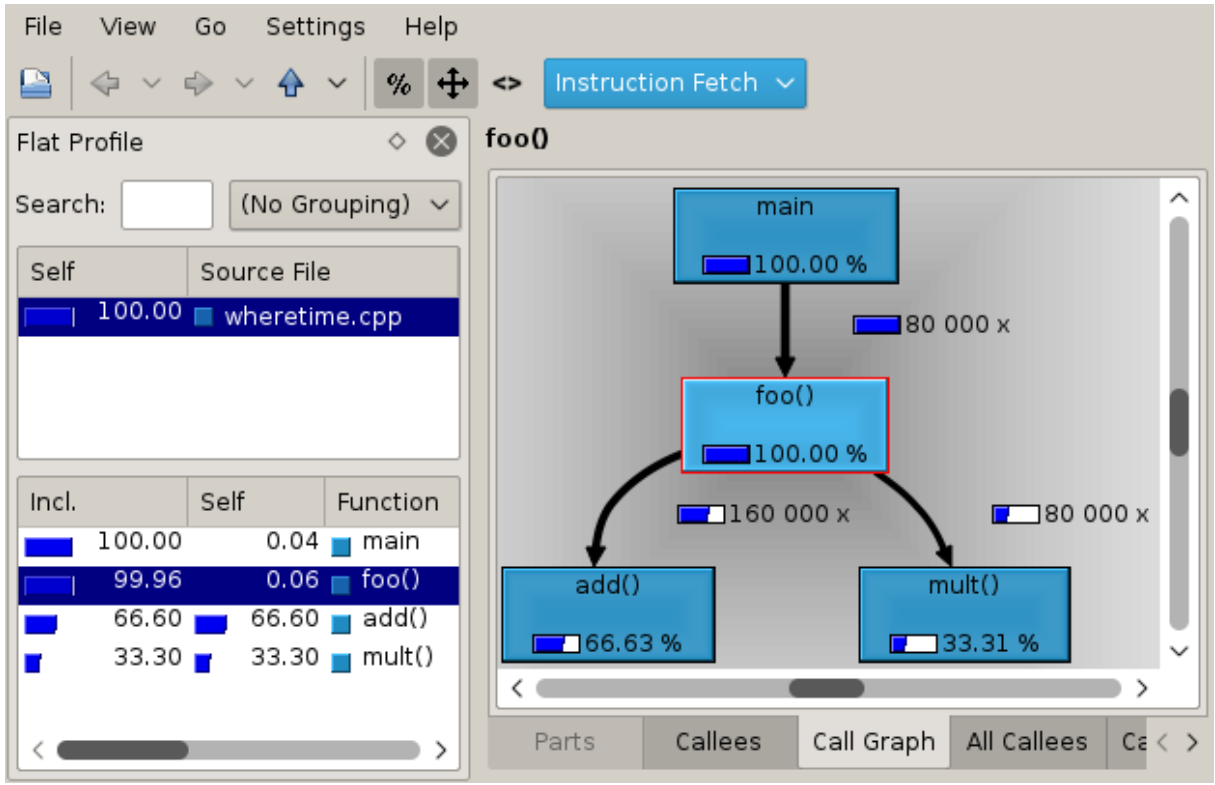
```
$ valgrind --tool=callgrind --dump-instr=yes --instr-atstart=no ./prog
$ kcachegrind callgrind.out.pid
```



Callgrind and kcachegrind

```cpp
#include <valgrind/callgrind.h>

int add() {
  int val = 0;
  for(int i = 0; i < 800; ++i)
    val += i;
  return val;
}

int mult() {
  int val = 1;
  for(int i = 0; i < 800; ++i)
    val *= i;
  return val;
}

int foo() {
  return add() + add() + mult();
}

int main(void) {
  int dum = 0;

  CALLGRIND_START_INSTRUMENTATION;
  for(int i = 0; i < 80000; ++i)
    dum += foo();
  CALLGRIND_STOP_INSTRUMENTATION;

  // other computation
  return 0;
}
```

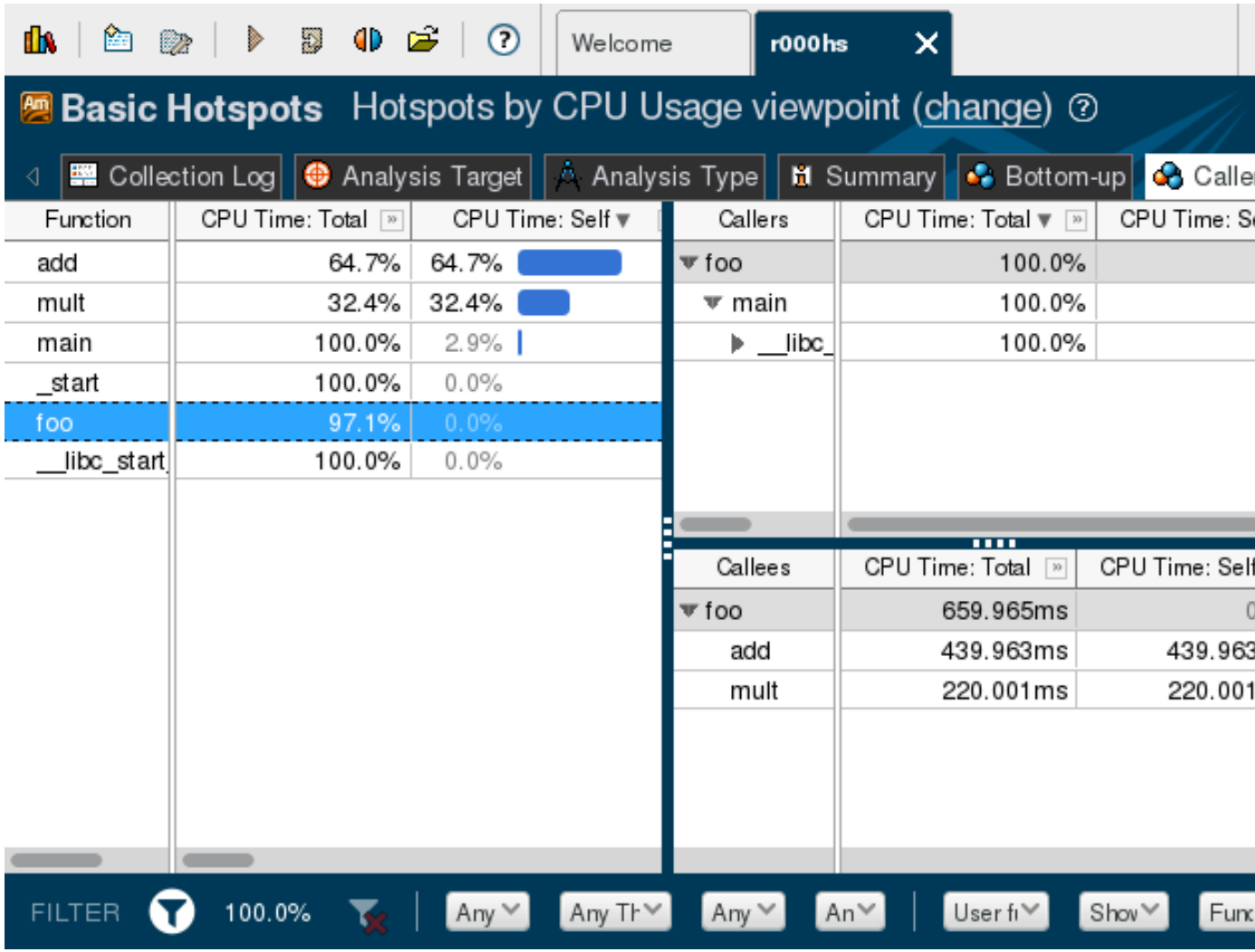
```
$ valgrind --tool=callgrind --dump-instr=yes --instr-atstart=no ./prog
$ kcachegrind callgrind.out.pid
```



intel® VTune™ Amplifier

```c
#include "ittnotify.h"

int add() {
  int val = 0;
  for(int i = 0; i < 800; ++i)
    val += i;
  return val;
}

int mult() {
  int val = 1;
  for(int i = 0; i < 800; ++i)
    val *= i;
  return val;
}

int foo() {
  return add() + add() + mult();
}

int main(void) {
  int dum = 0;

  __itt_resume();
  for(int i = 0; i < 80000; ++i)
    dum += foo();
  __itt_pause();

  // other computation
  return 0;
}
```

```
$ amplxe-cl -collect hotspots -start-paused -allow-multiple-runs -- ./prog
$ amplxe-gui r000ge/
```

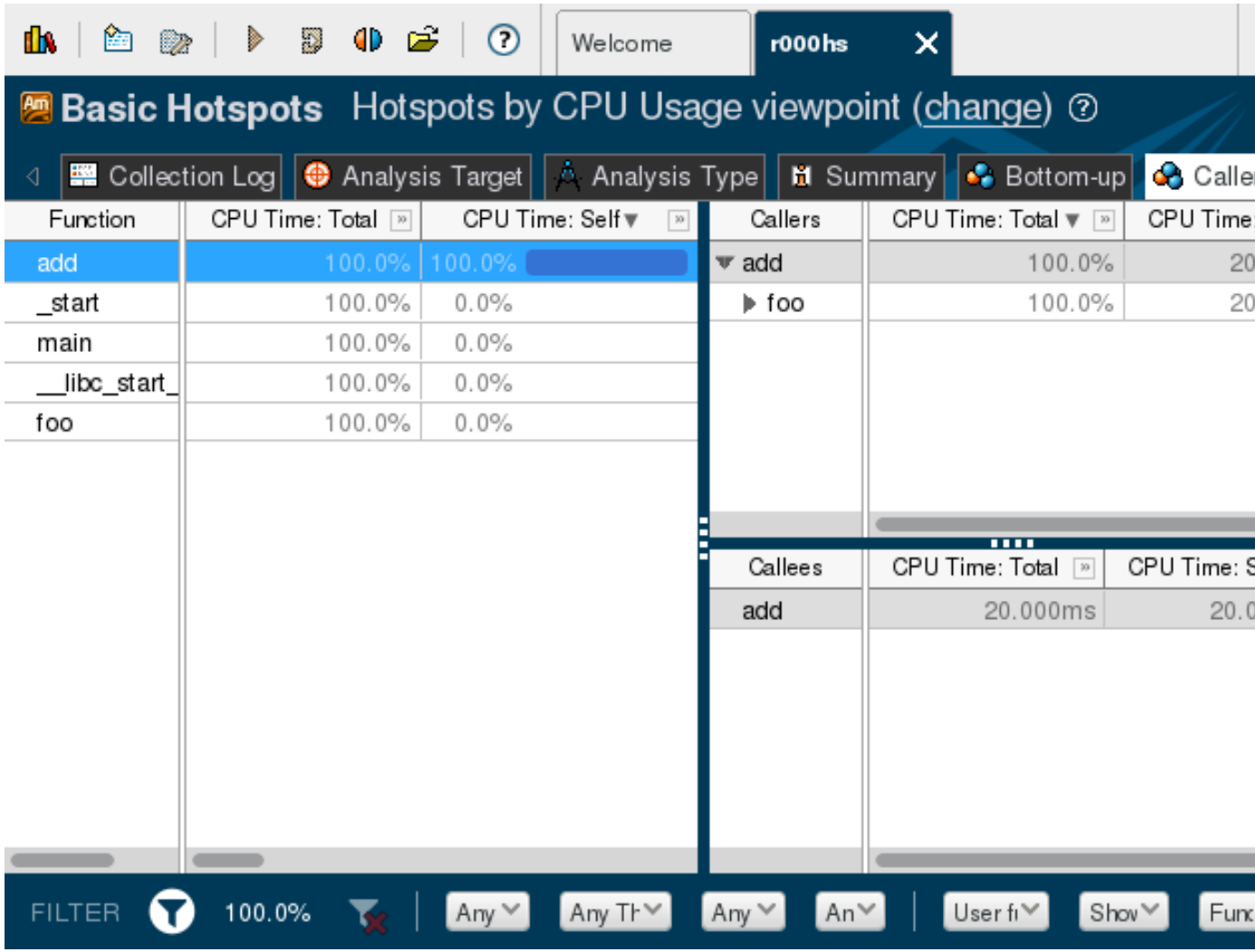


intel® VTune™ Amplifier

gray $\Longrightarrow$ no sufficient data

```
1  int add() {
2    int val = 0;
3    for(int i = 0; i < 800; ++i)
4      val += i;
5    return val;
6  }
7
8  int mult() {
9    int val = 1;
10   for(int i = 0; i < 800; ++i)
11     val *= i;
12   return val;
13 }
14
15 int foo() {
16   return add() + add() + mult();
17 }
18
19 int main(void) {
20   int dum = 0;
21
22   for(int i = 0; i < 80000; ++i)
23     dum += foo();
24
25   return 0;
26 }
```

```
$ amplxe-cl -collect hotspots -- ./prog
$ amplxe-gui r000ge/
```



Outline

Profile memory layout

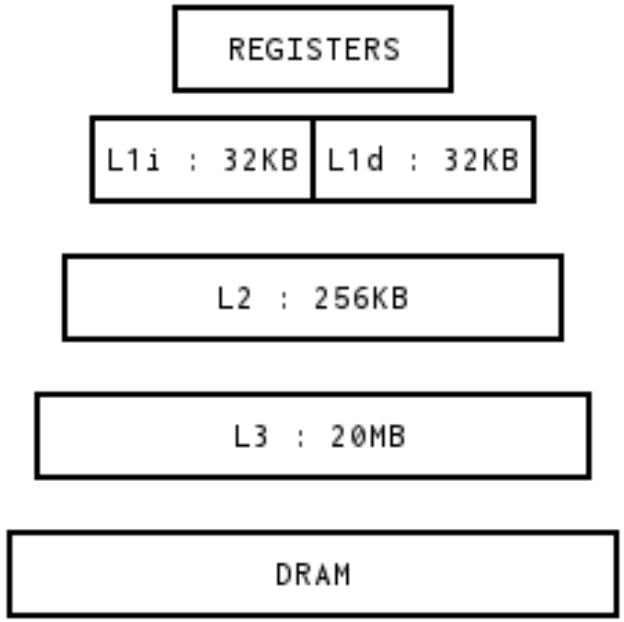


Figure: Memory hierarchy

Cache miss/hit

Access to memory is not free

- Access to memory is not free: latency
- If you try to access to a data
  - it will check if it's on L1 cache ∼4 cycles
  - if it's not on L1 $\Longrightarrow$ cache miss (else it's a hit)
  - if L1 cache miss
    it will check on L2 (∼10 ∼12 cycles)
  - if L2 cache miss
    it will check on L3 (∼30 ∼70 cycles)
  - if L3 cache miss
    it will check on DRAM (∼100 ∼150 cycles)

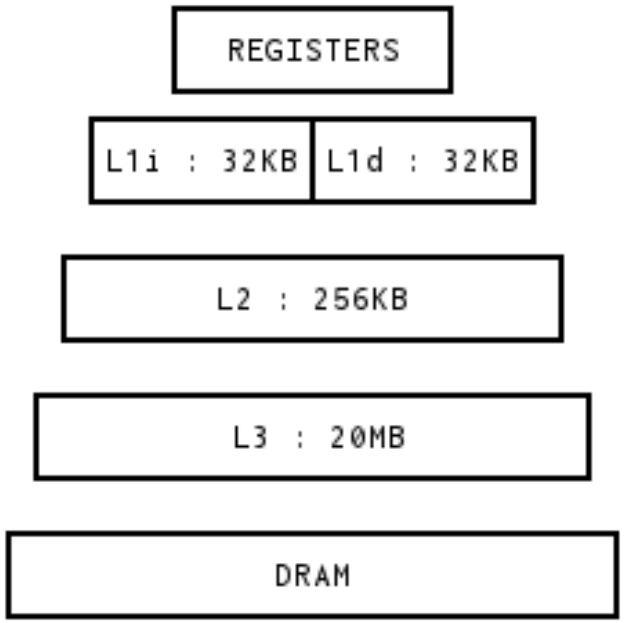
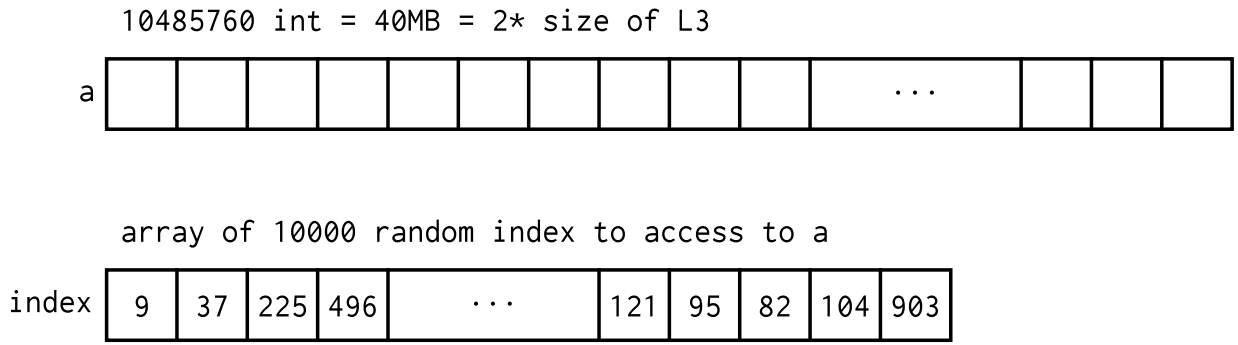


Figure: Memory hierarchy

Random access

```
1  #define SIZE 10485760
2  #define NBR 10000
3
4  int main() {
5    // array of data
6    int* a = new int[SIZE];
7    for(int i = 0; i < SIZE; i++)
8      a[i] = i;
9
10   // array of random index
11   int* index = new int[NBR];
12   for(int k = 0; k < NBR; ++k)
13     index[k] = rand() % SIZE;
14
15   int d = 0;
16   for(int k = 0; k < 1000; ++k) {
17     int c = 0;
18
19     // random fashion access
20     for(int n = 0; n < NBR; ++n)
21       c += a[index[n]];
22
23     d += c;
24   }
25
26   printf("%d\n", d);
27   return 0;
28 }
```



iteration n

↪ get a[index[n]]

Random access

perf

```
1  #define SIZE 10485760
2  #define NBR 10000
3
4  int main() {
5    // array of data
6    int* a = new int[SIZE];
7    for(int i = 0; i < SIZE; i++)
8      a[i] = i;
9
10   // array of random index
11   int* index = new int[NBR];
12   for(int k = 0; k < NBR; ++k)
13     index[k] = rand() % SIZE;
14
15   int d = 0;
16   for(int k = 0; k < 1000; ++k) {
17     int c = 0;
18
19     // random fashion access
20     for(int n = 0; n < NBR; ++n)
21       c += a[index[n]];
22
23     d += c;
24   }
25
26   printf("%d\n", d);
27   return 0;
28 }
```

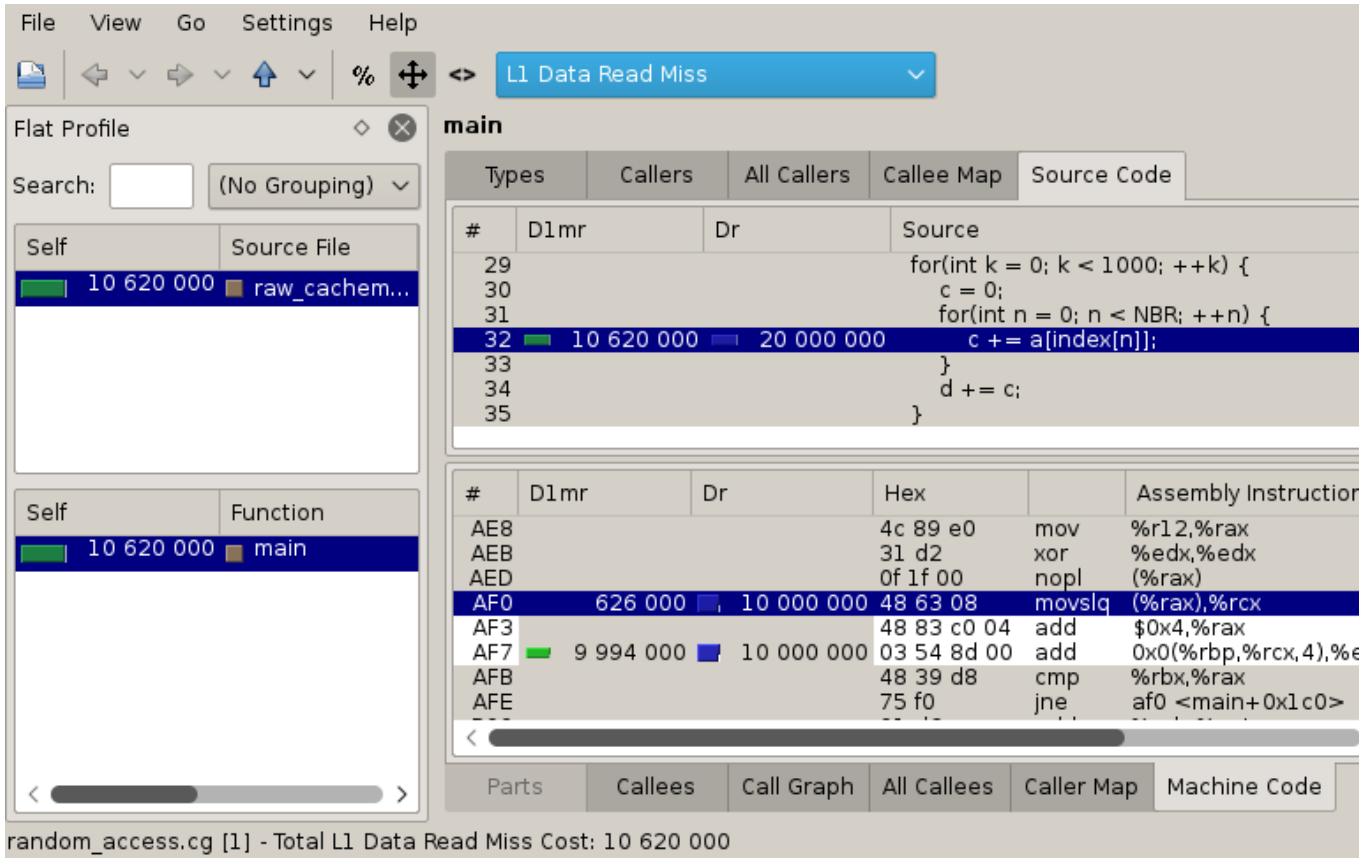
```
$ perf stat -ddd ./b_rand_access

  Performance counter stats for './b_rand_access':

  75,016,951      cycles                    #    1.731 GHz
    :
    .
  22,938,912      L1-dcache-loads           #  529.436 M/sec
  11,859,449      L1-dcache-load-misses     #   51.70% of all L1-dcache hits

0.042534358 seconds time elapsed
```

Random access

callgrind

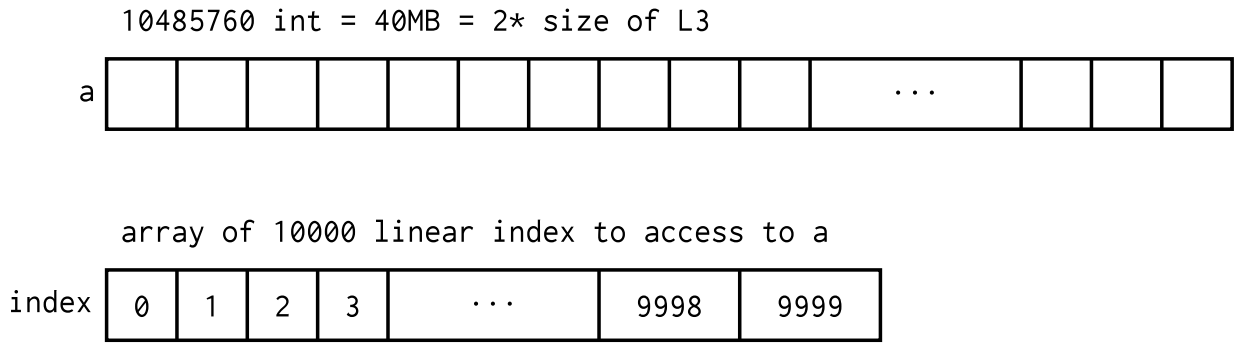
```
#define SIZE 10485760
#define NBR 10000

int main() {
  // array of data
  int* a = new int[SIZE];
  for(int i = 0; i < SIZE; i++)
    a[i] = i;

  // array of random index
  int* index = new int[NBR];
  for(int k = 0; k < NBR; ++k)
    index[k] = rand() % SIZE;

  int d = 0;
  for(int k = 0; k < 1000; ++k) {
    int c = 0;

    // random fashion access
    for(int n = 0; n < NBR; ++n)
      c += a[index[n]];

    d += c;
  }

  printf("%d\n", d);
  return 0;
}
```

```
$ valgrind --tool=callgrind --dump-instr=yes --cache-sim=yes \
           --instr-atstart=no ./b_rand_access
```



Linear access

```
1  #define SIZE 10485760
2  #define NBR 10000
3
4  int main() {
5    // array of data
6    int* a = new int[SIZE];
7    for(int i = 0; i < SIZE; i++)
8      a[i] = i;
9
10   // array of linear index
11   int* index = new int[NBR];
12   for(int k = 0; k < NBR; ++k)
13     index[k] = k;
14
15   int d = 0;
16   for(int k = 0; k < 1000; ++k) {
17     int c = 0;
18
19     // linear fashion access
20     for(int n = 0; n < NBR; ++n)
21       c += a[index[n]];
22
23     d += c;
24   }
25
26   printf("%d\n", d);
27   return 0;
28 }
```



iteration n

↪ get a[index[n]]

Linear access

see diff with perf

```
1  #define SIZE 10485760
2  #define NBR 10000
3
4  int main() {
5    // array of data
6    int* a = new int[SIZE];
7    for(int i = 0; i < SIZE; i++)
8      a[i] = i;
9
10   // array of linear index
11   int* index = new int[NBR];
12   for(int k = 0; k < NBR; ++k)
13     index[k] = k;
14
15   int d = 0;
16   for(int k = 0; k < 1000; ++k) {
17     int c = 0;
18
19     // linear fashion access
20     for(int n = 0; n < NBR; ++n)
21       c += a[index[n]];
22
23     d += c;
24   }
25
26   printf("%d\n", d);
27   return 0;
28 }
```

Random access

```
$ perf stat -ddd ./b_rand_access

   Performance counter stats for './b_rand_access':

    75,016,951      cycles                    #    1.731 GHz
     :
     :
    22,938,912      L1-dcache-loads           #  529.436 M/sec
    11,859,449      L1-dcache-load-misses     #   51.70% of all L1-dcache hits

   0.042534358 seconds time elapsed
```

Linear access

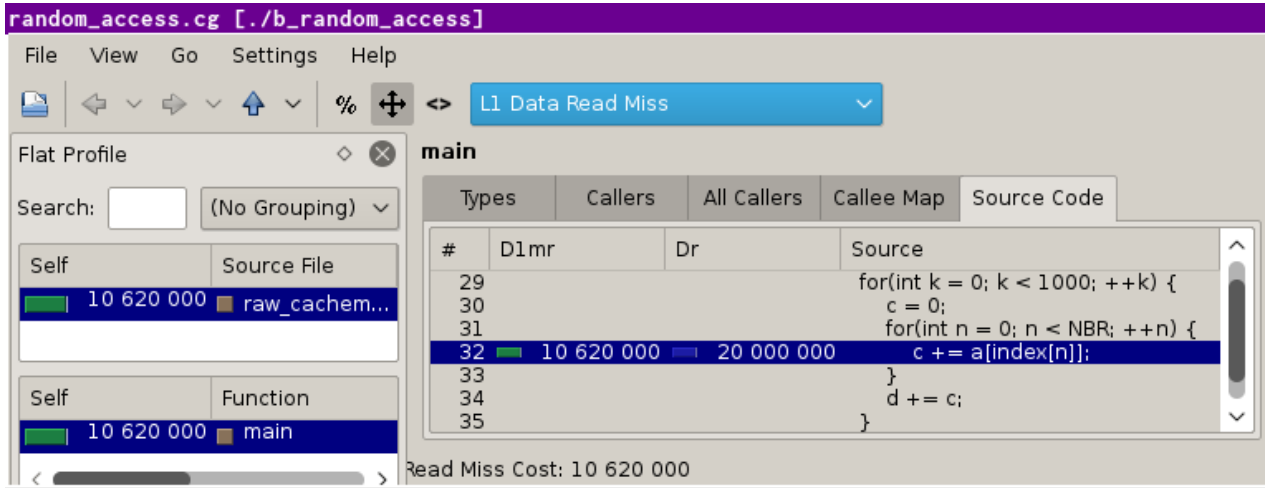
```
$ perf stat -ddd ./b_linear_access

   Performance counter stats for './b_linear_access':

    38,158,022      cycles                    #    2.978 GHz
     :
     :
    27,025,656      L1-dcache-loads           # 2109.142 M/sec
     1,227,040      L1-dcache-load-misses     #    4.54% of all L1-dcache hits

   0.019500381 seconds time elapsed
```

Linear access

see diff with callgrind

```
1  #define SIZE 10485760
2  #define NBR 10000
3
4  int main() {
5    // array of data
6    int* a = new int[SIZE];
7    for(int i = 0; i < SIZE; i++)
8      a[i] = i;
9
10   // array of linear index
11   int* index = new int[NBR];
12   for(int k = 0; k < NBR; ++k)
13     index[k] = k;
14
15   int d = 0;
16   for(int k = 0; k < 1000; ++k) {
17     int c = 0;
18
19     // linear fashion access
20     for(int n = 0; n < NBR; ++n)
21       c += a[index[n]];
22
23     d += c;
24   }
25
26   printf("%d\n", d);
27   return 0;
28 }
```

```
valgrind --tool=callgrind --dump-instr=yes --cache-sim=yes \
         --instr-atstart=no ./b_linear_access
```



Linear access

see diff with intel® VTune™ Amplifier

```
1  #define SIZE 10485760
2  #define NBR 10000
3
4  int main() {
5    // array of data
6    int* a = new int[SIZE];
7    for(int i = 0; i < SIZE; i++)
8      a[i] = i;
9
10   // array of linear index
11   int* index = new int[NBR];
12   for(int k = 0; k < NBR; ++k)
13     index[k] = k;
14
15   int d = 0;
16   for(int k = 0; k < 1000; ++k) {
17     int c = 0;
18
19     // linear fashion access
20     for(int n = 0; n < NBR; ++n)
21       c += a[index[n]];
22
23     d += c;
24   }
25
26   printf("%d\n", d);
27   return 0;
28 }
```

See all supported events

```
amplxe-cl -collect-with runsa  -k event-config=?
```

Grep what you want

```
amplxe-cl -collect-with runsa  -k event-config=? | grep -A3 L1_

   :
MEM_LOAD_UOPS_RETIRED.L1_MISS_PS Retired load uops misses in L1 cache as data
                                 sources.

MEM_LOAD_UOPS_RETIRED.L1_HIT_PS  Retired load uops with L1 cache hits as data
                                 sources.
```

Linear access

see diff with intel® VTune™ Amplifier

```
1  #define SIZE 10485760
2  #define NBR 10000
3
4  int main() {
5    // array of data
6    int* a = new int[SIZE];
7    for(int i = 0; i < SIZE; i++)
8      a[i] = i;
9
10   // array of linear index
11   int* index = new int[NBR];
12   for(int k = 0; k < NBR; ++k)
13     index[k] = k;
14
15   int d = 0;
16   for(int k = 0; k < 1000; ++k) {
17     int c = 0;
18
19     // linear fashion access
20     for(int n = 0; n < NBR; ++n)
21       c += a[index[n]];
22
23     d += c;
24   }
25
26   printf("%d\n", d);
27   return 0;
28 }
```

Rand access

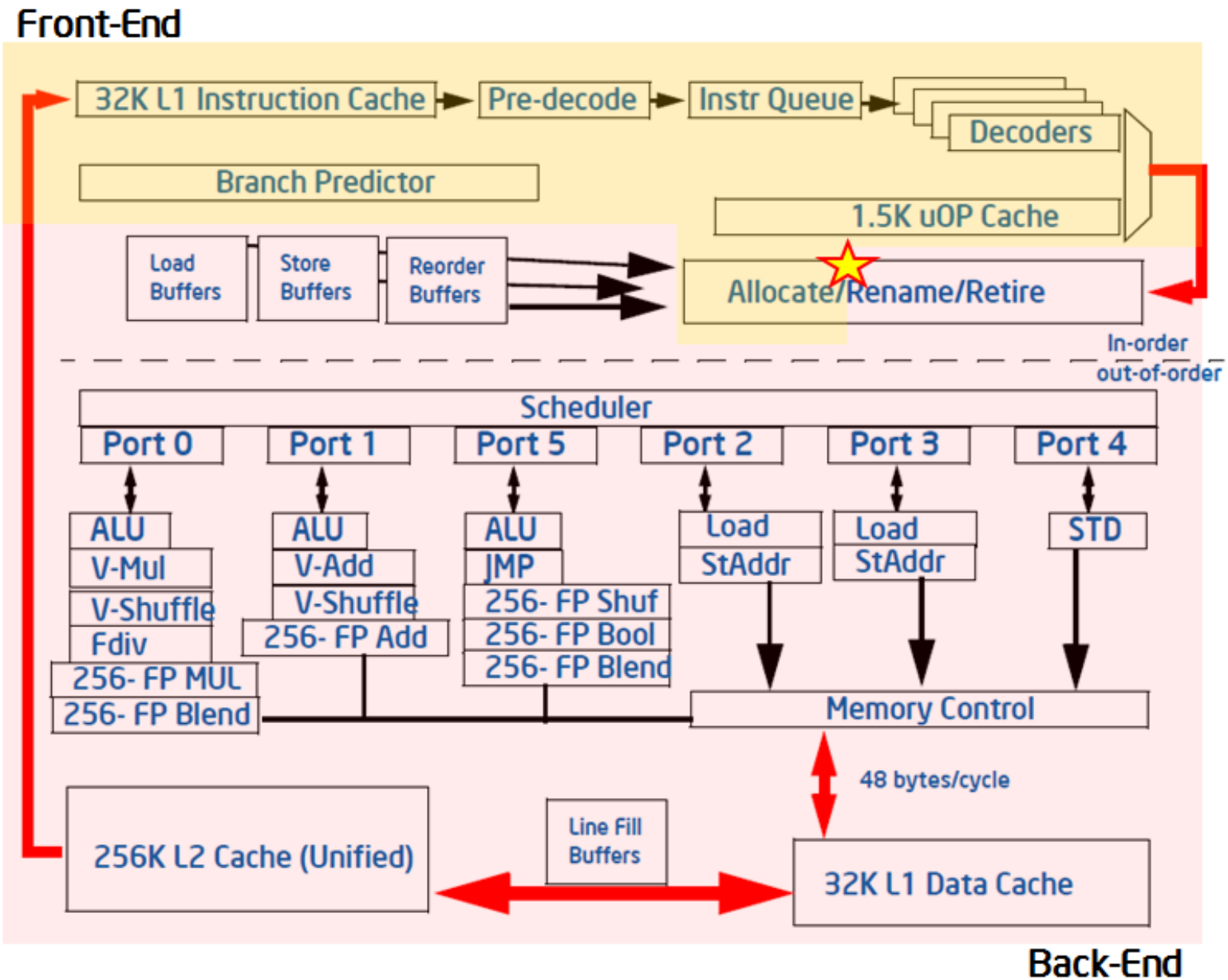
```
amplxe-cl -collect-with runsa
          -k event-config="MEM_LOAD_UOPS_RETIRED.L1_HIT_PS,  \
                           MEM_LOAD_UOPS_RETIRED.L1_MISS_PS" ./b_rand_access


-------------
Hardware Event Type                Hardware Event Count:Self
-------------------------------  -------------------------
MEM_LOAD_UOPS_RETIRED.L1_HIT_PS                      8000012
MEM_LOAD_UOPS_RETIRED.L1_MISS_PS                    10400312
```

Linear access

```
amplxe-cl -collect-with runsa
          -k event-config="MEM_LOAD_UOPS_RETIRED.L1_HIT_PS,  \
                           MEM_LOAD_UOPS_RETIRED.L1_MISS_PS" ./b_linear_access


-------------
Hardware Event Type                Hardware Event Count:Self
-------------------------------  -------------------------
MEM_LOAD_UOPS_RETIRED.L1_HIT_PS                     20000030
MEM_LOAD_UOPS_RETIRED.L1_MISS_PS                     1000030
```

Outline

Pipeline


Front-End
32K L1 Instruction Cache
Pre-decode
Instr Queue
Decoders
Branch Predictor
1.5K uOP Cache
Load Buffers
Store Buffers
Reorder Buffers
Allocate/Rename/Retire
In-order
out-of-order
Scheduler
Port 0
Port 1
Port 5
Port 2
Port 3
Port 4
ALU
V-Mul
V-Shuffle
Fdiv
256- FP MUL
256- FP Blend
ALU
V-Add
V-Shuffle
256- FP Add
ALU
JMP
256- FP Shuf
256- FP Bool
256- FP Blend
Load
StAddr
Load
StAddr
STD
Memory Control
48 bytes/cycle
256K L2 Cache (Unified)
Line Fill Buffers
32K L1 Data Cache
Back-End

Pipeline and CPI

- IF: Instruction Fetch
- ID: Instruction Decode
- EX: Execution
- WB: Write Back

Superscalar = multiple pipelines

Instr.	Pipeline						
1	IF	ID	EX	WB			
2	IF	ID	EX	WB			
3		IF	ID	EX	WB		
4		IF	ID	EX	WB		
5			IF	ID	EX	WB	
6			IF	ID	EX	WB	

Cycle	1	2	3	4	5	6	7

Figure: Simplified view of the pipeline with CPI = 0.5

CPI, number of cycles to wait in average before executing a new instruction
IPC, instructions per cycle, number of instructions in average for one cycle (reciprocal)

Pipeline and CPI

- IF: Instruction Fetch
- ID: Instruction Decode
- WB: Write Back

Example

```
1 E  = A * B
2 F  = A * C
3 G  = A + F    // need F
4 Q1 = A / B
5 Q2 = A / C
```

Instr.	Pipeline									
1	IF	ID	MUL 1/2	MUL 2/2	WB E					
2		IF	ID	MUL 1/2	MUL 2/2	WB F				
3			IF	ID	need F	need F	ADD 1/1	WB G		
4				IF	ID	DIV 1/3	DIV 2/3	DIV 3/3	WB Q1	
5					IF	ID	wait port	DIV 1/3	DIV 2/3	DIV 3/3

Cycle	1	2	3	4	5	6	7	8	9	10

Figure: Example of bubbles in the pipeline impacting CPI

Thanks to Florian Lemaitre for this example

Random access

what is the CPI?

```
1  #define SIZE 10485760
2  #define NBR 10000
3
4  int main() {
5    // array of data
6    int* a = new int[SIZE];
7    for(int i = 0; i < SIZE; i++)
8      a[i] = i;
9
10   // array of random index
11   int* index = new int[NBR];
12   for(int k = 0; k < NBR; ++k)
13     index[k] = rand() % SIZE;
14
15   int d = 0;
16   for(int k = 0; k < 1000; ++k) {
17     int c = 0;
18
19     // random fashion access
20     for(int n = 0; n < NBR; ++n)
21       c += a[index[n]];
22
23     d += c;
24   }
25
26   printf("%d\n", d);
27   return 0;
28 }
```

Recording

```
amplxe-cl -collect general-exploration -q -- ./b_cpi

Clockticks: 84,000,000
Instructions Retired: 88,800,000
CPI Rate: 0.946                    # CPI for the whole process
 :
 :
```

Reporting

```
amplxe-cl -report hotspots -r r000ge/     \
          -column='cpi' -filter="function=main"

Column filter is ON.
Function  CPI Rate
--------  --------
main         1.095
```

Random access

adding big operation, what is the CPI?

```
1  #define SIZE 10485760
2  #define NBR 10000
3
4  int main() {
5    // array of data
6    int* a = new int[SIZE];
7    for(int i = 0; i < SIZE; i++)
8      a[i] = i;
9
10   // array of random index
11   int* index = new int[NBR];
12   for(int k = 0; k < NBR; ++k)
13     index[k] = rand() % SIZE;
14
15   int d = 0;
16   for(int k = 0; k < 1000; ++k) {
17     int c = 0;
18     double bigop = 0.;
19
20     // random fashion access
21     for(int n = 0; n < NBR; ++n) {
22       c += a[index[n]];
23       // adding an sqrt here
24       bigop += sqrt(n);
25     }
26
27     d += c + bigop;
28   }
29
30   printf("%d\n", d);
31   return 0;
32 }
```

Recording

```
amplxe-cl -collect general-exploration -q -- ./b_cpi

Clockticks: 86,400,000
Instructions Retired: 144,000,000
CPI Rate: 0.600                    # CPI for the whole process
 :
 :
```

Reporting

```
amplxe-cl -report hotspots -r r001ge/     \
          -column='cpi' -filter="function=main"

Column filter is ON.
Function  CPI Rate
--------  --------
main         0.523
```

Random access

adding big operation, what is the CPI?

	Clockticks	Instructions	CPI Rate on main function
Without sqrt	84,000,000	88,800,000	1.095
With sqrt	86,400,400	144,000,000	0.523

Figure: With vs. without adding big op in parallel of cache misses, what happened?

What could we think about that?

CPI improved by two, and cache misses still there

- ↪ good CPI doesn't mean necessarily all is ok
- ↪ bad CPI doesn't mean necessarily you can improved it
- ↪ CPI metric needs a lot of context to be interpreted

Overview

Usage

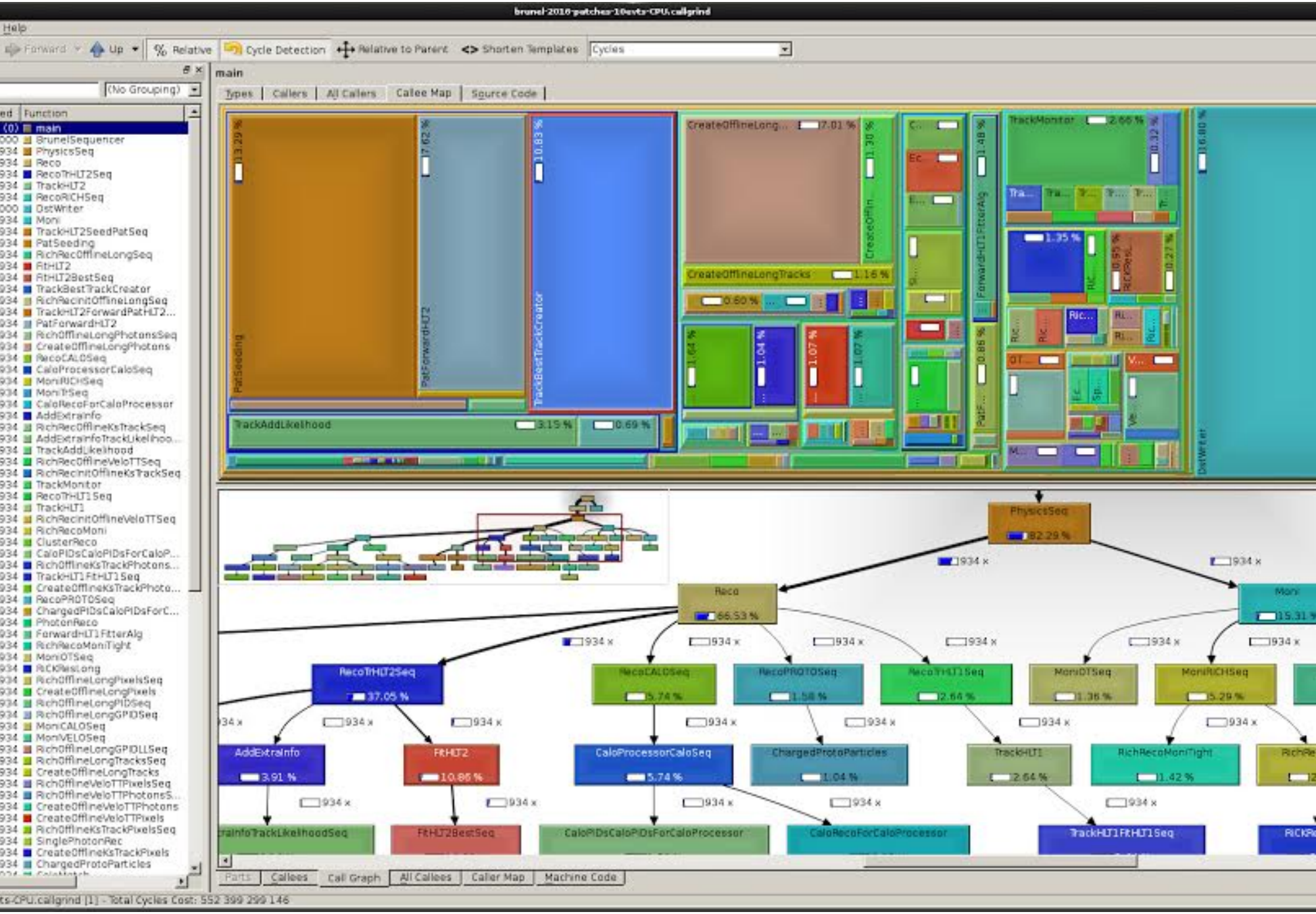
```
./Project/run <your profiler command> -- python $(./Project/run which gaudirun.py) my_options.py
#               put these two dashes ↗  to stop option list of your profiler command
```

Callgrind example

```
./Brunel/run valgrind --tool=callgrind -- python $(./Project/run which gaudirun.py) my_options.py
```

VTune example

```
./Brunel/run amplxe-cl -collect general-exploration -- python $(./Project/run which gaudirun.py) my_options.py
```


brunel-2016-patches-10evts-CPU.callgrind
Help
Forward
Up
Relative
Cycle Detection
Relative to Parent
Shorten Templates
Cycles
(No Grouping)
Function
main
BrunelSequencer
PhysicsSeq
Reco
RecoTrHLT2Seq
TrackHLT2
RecoRICHSeq
OstWriter
Moni
TrackHLT2SeedPatSeq
PatSeeding
RichRecOfflineLongSeq
FitHLT2
FitHLT2BestSeq
TrackBestTrackCreator
RichRecInitOfflineLongSeq
TrackHLT2ForwardPatHLT2...
PatForwardHLT2
RichOfflineLongPhotonsSeq
CreateOfflineLongPhotons
RecoCALOSeq
CaloProcessorCaloSeq
MoniRICHSeq
MoniTrSeq
CaloRecoForCaloProcessor
AddExtraInfo
RichRecOfflineKsTrackSeq
TrackAddLikelihood
RichRecOfflineVeloTTSeq
RichRecInitOfflineKsTrackSeq
TrackMonitor
RecoTrHLT1Seq
TrackHLT1
RichRecInitOfflineVeloTTSeq
RichRecoMoni
ClusterReco
RichOfflineKsTrackPhotons...
TrackHLT1FitHLT1Seq
RecoPROTOSeq
ChargedPIDsCaloPIDsForC...
PhotonReco
ForwardHLT1FitterAlg
RichRecoMoniTight
MoniOTSeq
RICKResLong
RichOfflineLongPixelsSeq
CreateOfflineLongPixels
RichOfflineLongPIDSeq
RichOfflineLongGPIDSeq
MoniCALOSeq
MoniVELOSeq
RichOfflineLongGPIDLLSeq
RichOfflineLongTracksSeq
CreateOfflineLongTracks
RichOfflineVeloTTPixelsSeq
RichOfflineVeloTTPhotonsS...
CreateOfflineVeloTTPhotons
CreateOfflineVeloTTPixels
RichOfflineKsTrackPixelsSeq
SinglePhotonRec
CreateOfflineKsTrackPixels
ChargedProtoParticles
main
Types
Callers
All Callers
Callee Map
Source Code
PatSeeding
PatForwardHLT2
TrackBestTrackCreator
TrackAddLikelihood
CreateOfflineLong...
CreateOfflineLongTracks
ForwardHLT1FitterAlg
TrackMonitor
DstWriter
PhysicsSeq
82.29 %
Reco
66.53 %
Moni
RecoTrHLT2Seq
37.05 %
RecoCALOSeq
5.74 %
RecoPROTOSeq
1.58 %
RecoTrHLT1Seq
2.64 %
MoniOTSeq
1.36 %
MoniRICHSeq
5.29 %
AddExtraInfo
3.91 %
FitHLT2
10.86 %
CaloProcessorCaloSeq
5.74 %
ChargedProtoParticles
1.04 %
TrackHLT1
2.64 %
RichRecoMoniTight
1.42 %
FitHLT2BestSeq
CaloPIDsCaloPIDsForCaloProcessor
CaloRecoForCaloProcessor
TrackHLT1FitHLT1Seq
934 x
Parts
Callees
Call Graph
All Callees
Caller Map
Machine Code
ts-CPU.callgrind [1] - Total Cycles Cost: 552 399 299 146

Thanks! Any question?