Summary of continued work on energy profiling

Mikael Hirki

June 8, 2015

About me

- I'm a Master's degree student at Aalto (CS program)
- I have prior experience in performance profiling and optimization
- I'm doing my thesis on energy profiling

Outline

- Step 1: RAPL validation
- Step 2: ParFullCMS energy trace analysis
- Step 3: Power modeling

About the hardware

- I'm using a Haswell desktop computer
  - Intel Core i7-4770 @ 3.4 GHz
  - Intel DH87RL desktop board
  - 2 x 8 GB DDR3 1600 MHz
- Hyperthreading enabled
- Turbo Boost disabled

RAPL findings

- The time between RAPL updates is 1 ms (+/- 20 µs)
- Haswell measures energy in units of 61 µJ
- Latency of reading one RAPL counter using PAPI is about 550 nanoseconds
- Idle power consumption of my system is fairly low
  - Package (PKG): 0.95 W
  - Power plane 0 (PP0) / Processor cores: 18.4 mW
  - Power plane 1 (PP1) / Integrated graphics: 10 mW
  - DRAM: 1.6 W

STREAM benchmark

- My results were very similar to what Filip discovered
- STREAM stresses DRAM
  - Highest bandwidth 14.7 GB/s
  - DRAM power consumption is only about 5.5W

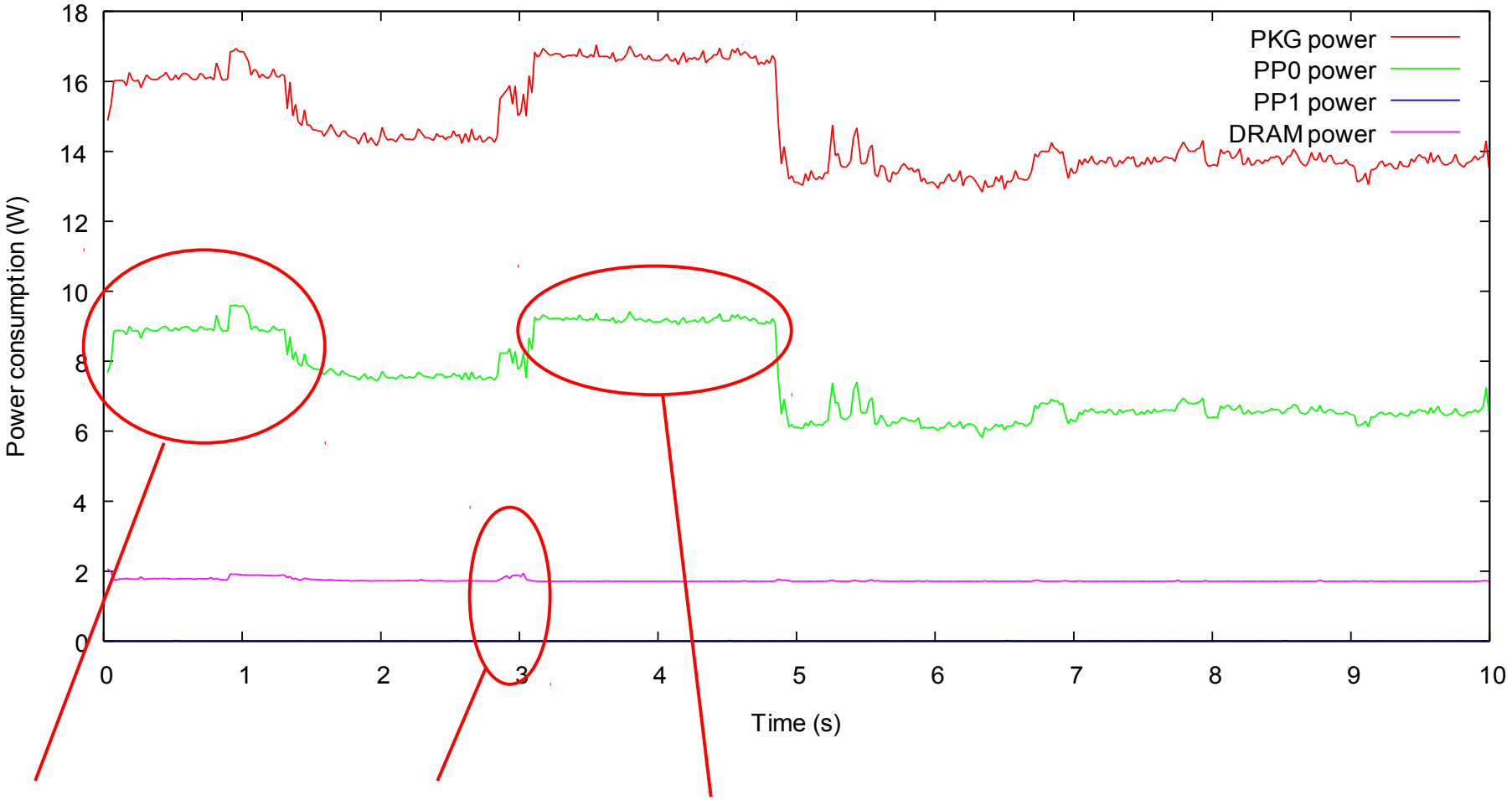
ParFullCMS

- I installed ParFullCMS from http://davidlt.web.cern.ch/davidlt/g4parfullcms/G4test-src-V11.tar.gz
- Running in single-threaded mode for energy profiling

Graphical look at ParFullCMS

- I wrote my own tool to measure power consumption every 5 ms
  - This corresponds to how the energy profiling module works
- No stack traces
- Plotted the energy trace using Gnuplot

ParFullCMS initialization (single-threaded)

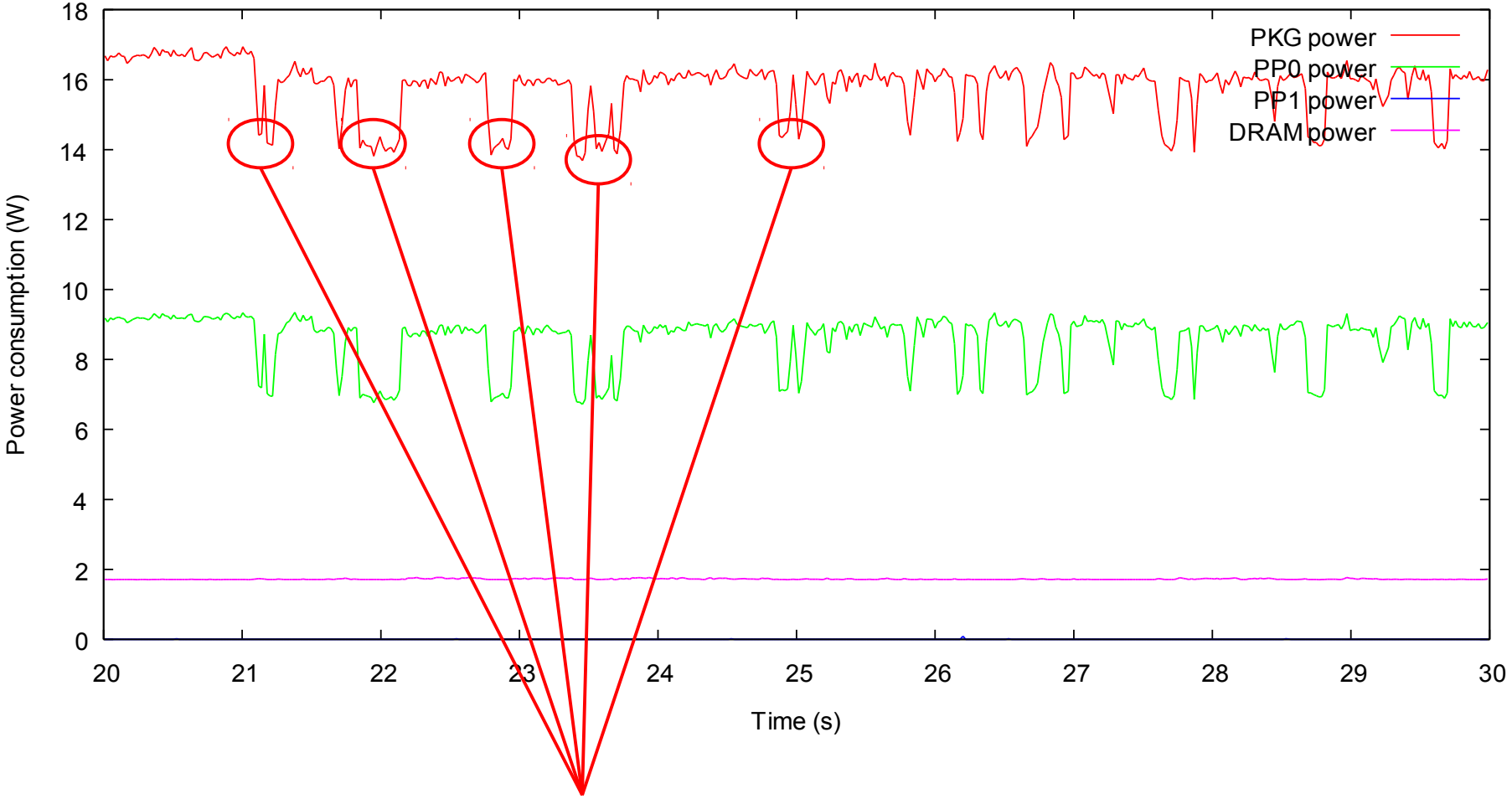


XML data parsing

DRAM access burst

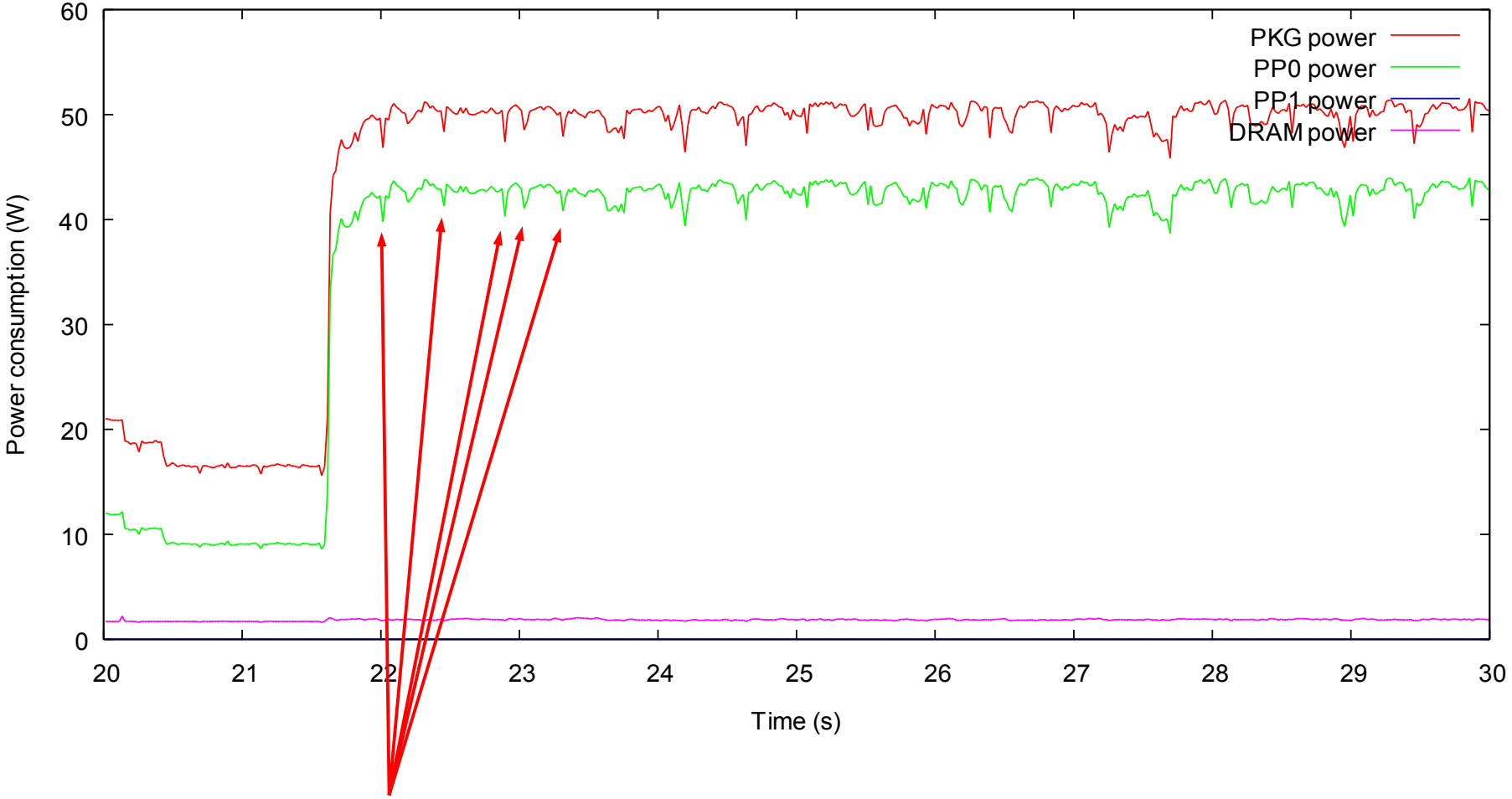
High power consumption caused by a recursive function

ParFullCMS worker thread (single-threaded)



Strange drops in power consumption

ParFullCMS worker threads (8 threads)



Similar drops in power consumption

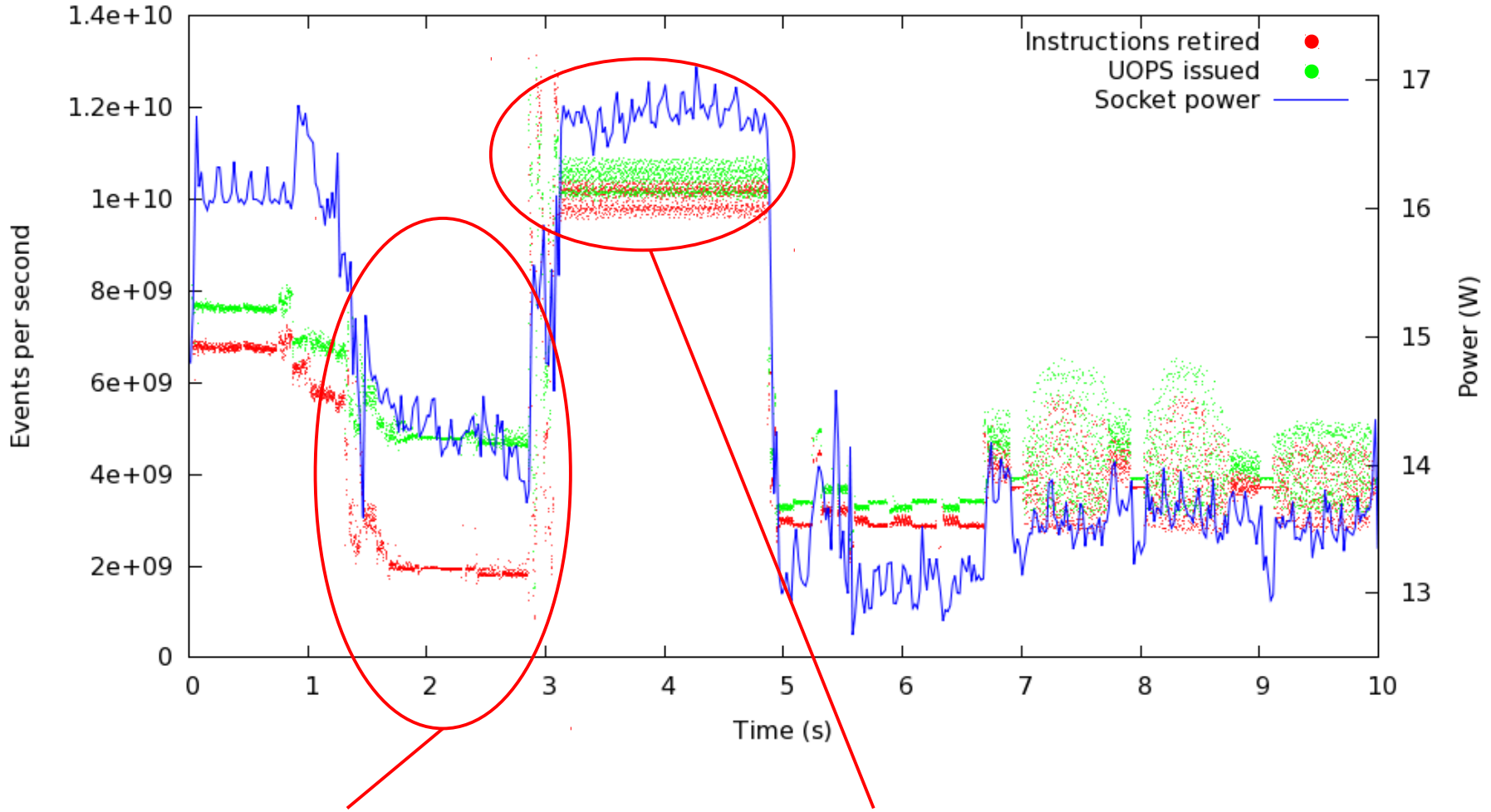
Performance counters on Intel hardware

- 8 programmable counters per core
  - 4 counters per thread when using hyperthreading
- 3 fixed-function counters
  - Unhalted core cycles
    - Unhalted = core is not sleeping
    - Core cycles is affected by frequency scaling
  - Unhalted reference cycles (not available in perf)
  - Instructions retired

µops issued

- Known as UOPS_ISSUED.ANY in Intel's manual
- The number of micro-operations issued by the front-end to the back-end inside the processor
  - UOPS_ISSUED includes speculative execution
  - Failed branch prediction increases UOPS_ISSUED
  - Conceptually, this is much closer to the actual amount of work done than the number of instructions retired

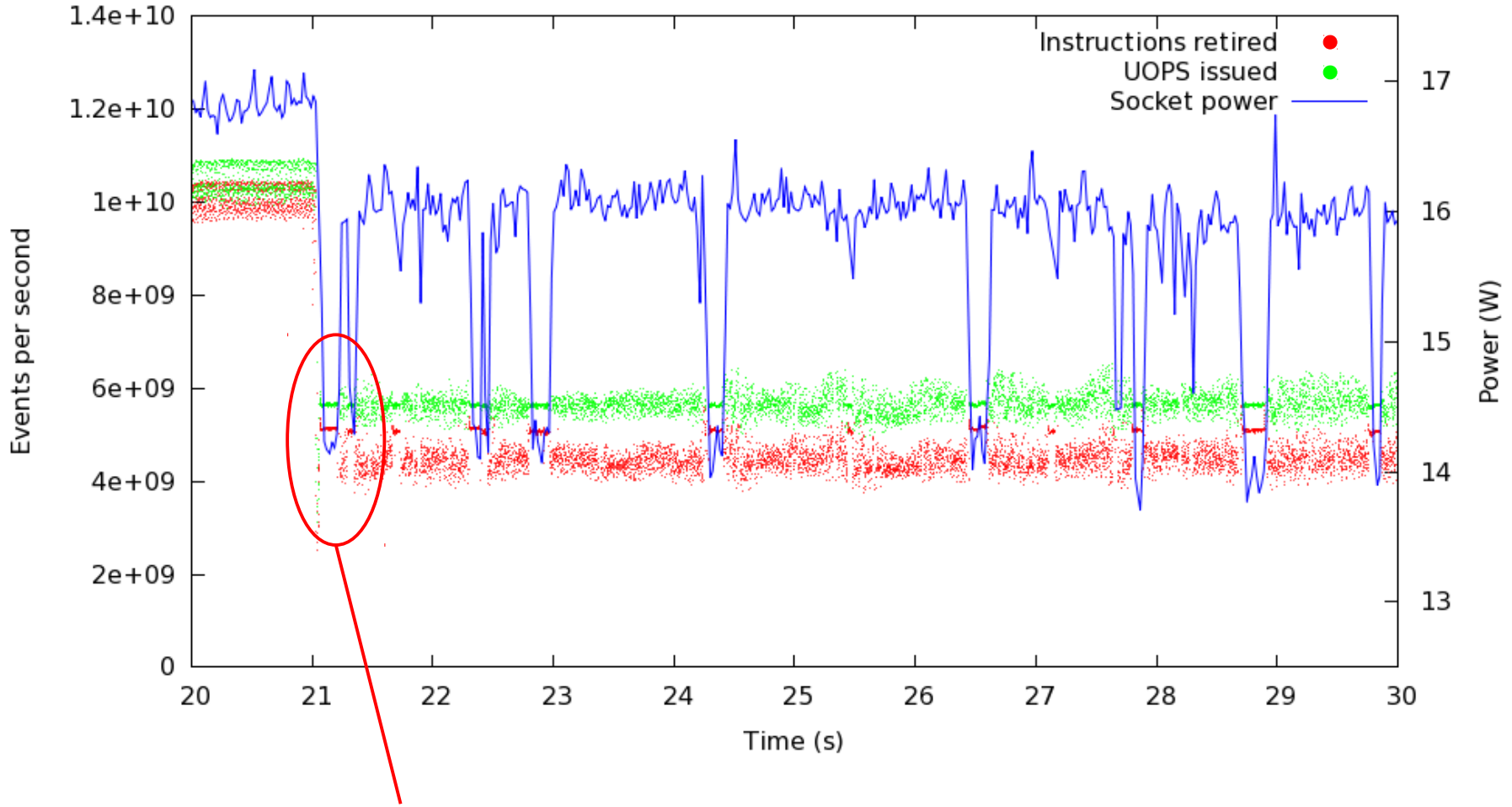
Socket power compared to instructions retired and µops issued



µops issued shows much better correlation than instructions retired

High power consumption caused by a recursive function

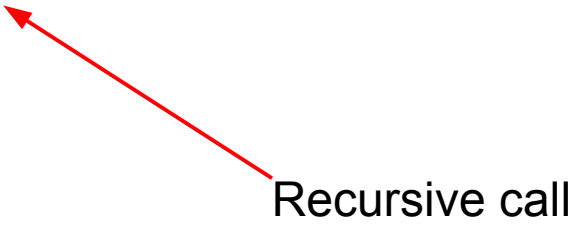
Socket power compared to instructions retired and µops issued



The mean number of µops remains the same while power consumption drops

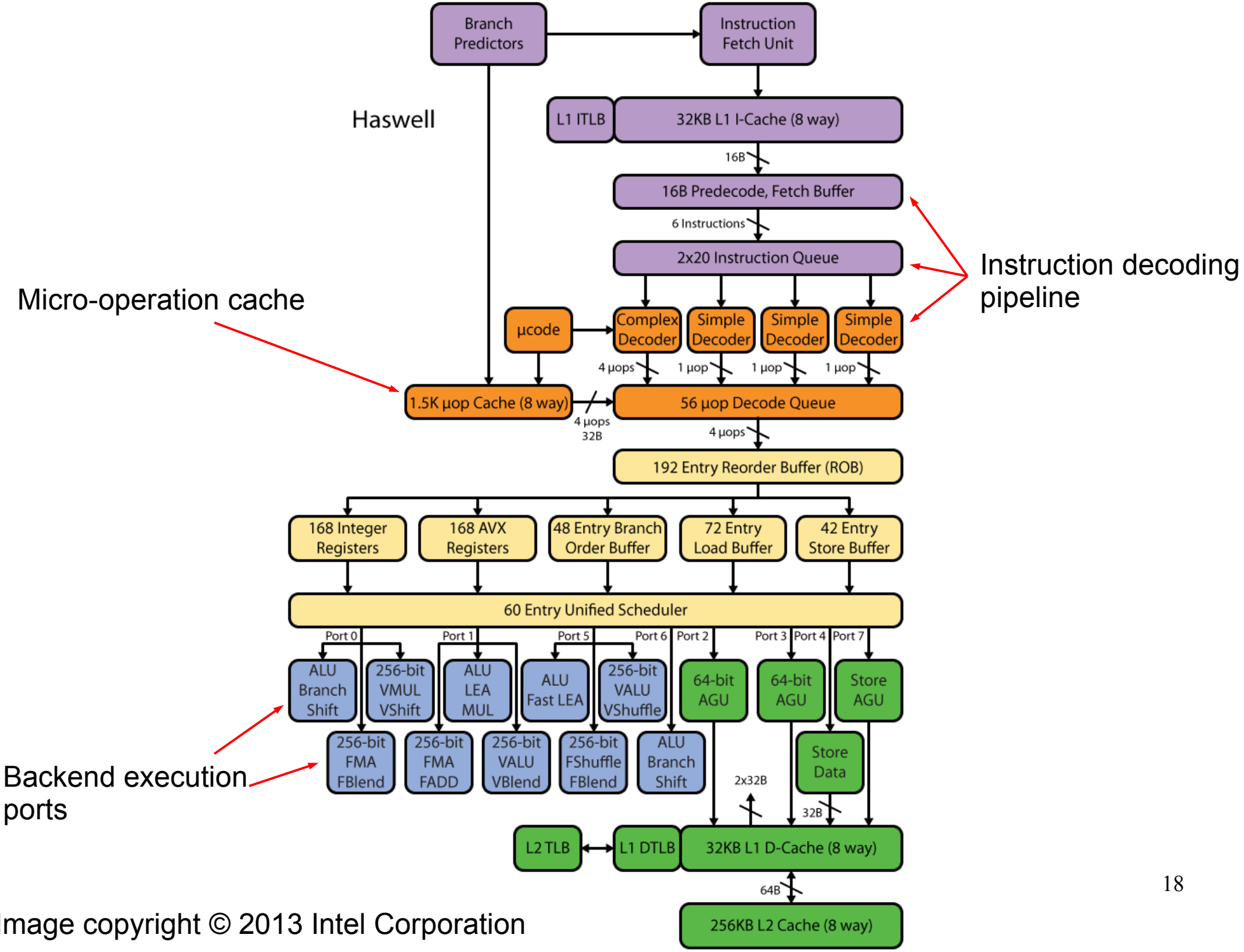
Recursive function code

```
void G4ProductionCutsTable::ScanAndSetCouple(G4LogicalVolume* aLV,
                                             G4MaterialCutsCouple* aCouple,
                                             G4Region* aRegion)
{
  //Check whether or not this logical volume belongs to the same region
  if((aRegion!=0) && aLV->GetRegion()!=aRegion) return;

  //Check if this particular volume has a material matched to the couple
  if(aLV->GetMaterial()==aCouple->GetMaterial()) {
    aLV->SetMaterialCutsCouple(aCouple);
  }

  size_t noDaughters = aLV->GetNoDaughters();
  if(noDaughters==0) return;

  //Loop over daughters with same region
  for(size_t i=0;i<noDaughters;i++){
    G4LogicalVolume* daughterLVol = aLV->GetDaughter(i)->GetLogicalVolume();
    ScanAndSetCouple(daughterLVol,aCouple,aRegion);
  }
}
```



Recursive function optimizations

- Eliminating register spilling boosted performance by 13%
  - Register pressure caused by function arguments, and implicit 64bit <-> 32bit integer casts
- Power consumption remained the same
  - Thus energy consumption was also reduced by 13%



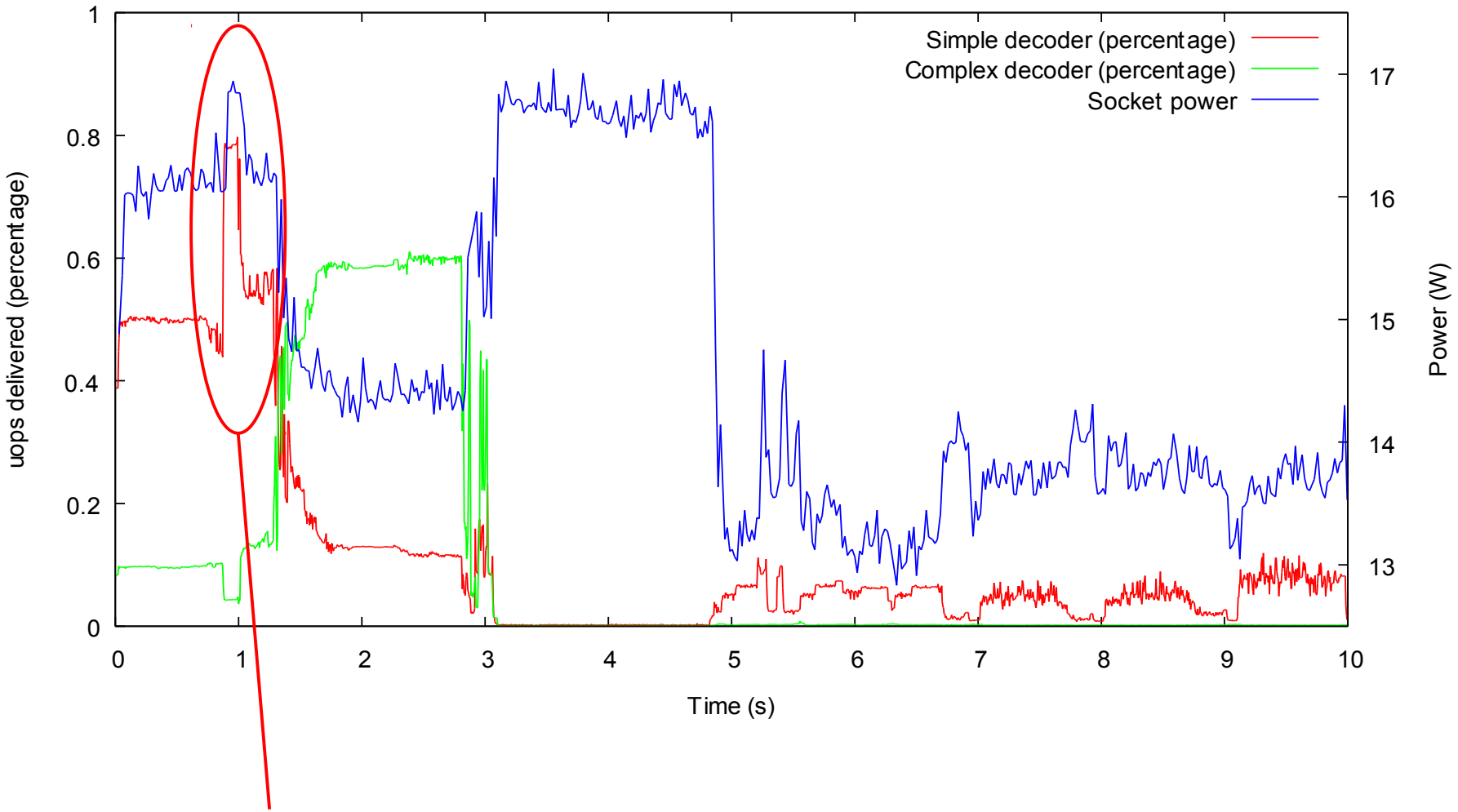
Sources of micro-operations

- Three different sources of micro-operations in the frontend
  - Simple instruction decoders
  - Complex instruction decoders
    - Micro-operations are stored in the microcode ROM
    - Instructions that generate more than 4 μops
  - Micro-operation cache
    - Complex instructions fill an entire cache line (6 μops)

Instruction decoders events

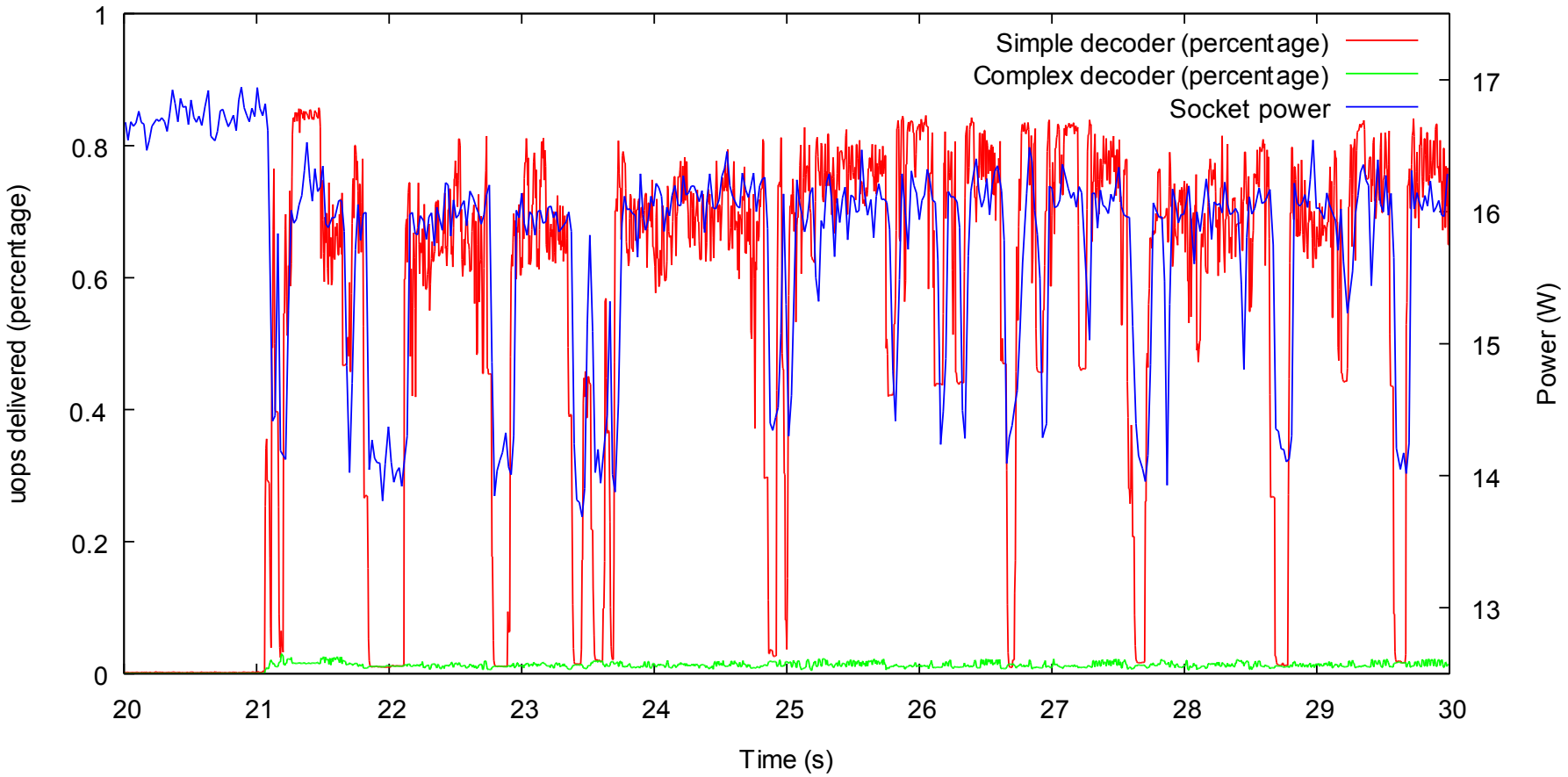
- IDQ.DSB_UOPS
  - Micro-ops delivered from the micro-operation cache
- IDQ.MITE_UOPS
  - Micro-ops delivered from the simple decoders
- IDQ.MS_UOPS
  - Micro-ops delivered from the complex decoder (microsequencer)
- Typical distribution is 80% cached micro-operations, 15% from simple decoders and 5% from complex decoder

Instruction decoder events graphically in ParFullCMS



Spike in power consumption correlates with instruction decoder activity

Instruction decoder events graphically in ParFullCMS



The extra 2 watts of power correlates with instruction decoder activity

Conclusion

- Instruction decoder consumes about 2 watts of power
- This is a power saving opportunity
  - Need to increase code locality
    - Do as much useful work as possible once code has been decoded into the µop cache
  - Need to optimize for smaller code size
    - Avoid using -O3 and -funroll-loops too much

Ideas for further work

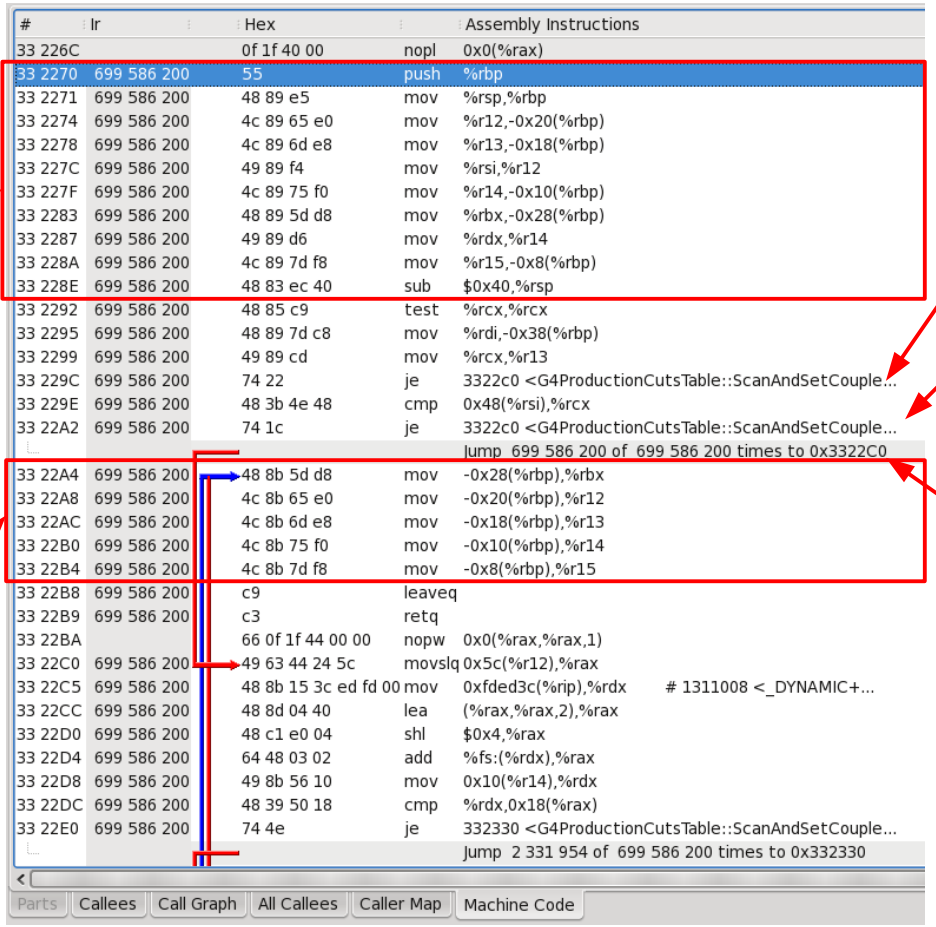
- Power modeling
  - Currently in progress
- Compiler optimization comparison
- Intel vs ARM hardware comparison

Power modeling

- ParFullCMS power consumption could be modeled using only two performance counters:
  - UOPS_ISSUED.ANY
    - This accounts for the work done by the CPU backend
  - IDQ.MITE_UOPS
    - This accounts for the (simple) instruction decoders
- This simple model fails to predict the power consumption in the PARSEC benchmark
  - Need to include L1 & L2 cache accesses and number of active cores in the model

Extra slides

Machine code view (first part)



Lots of values being pushed onto the stack

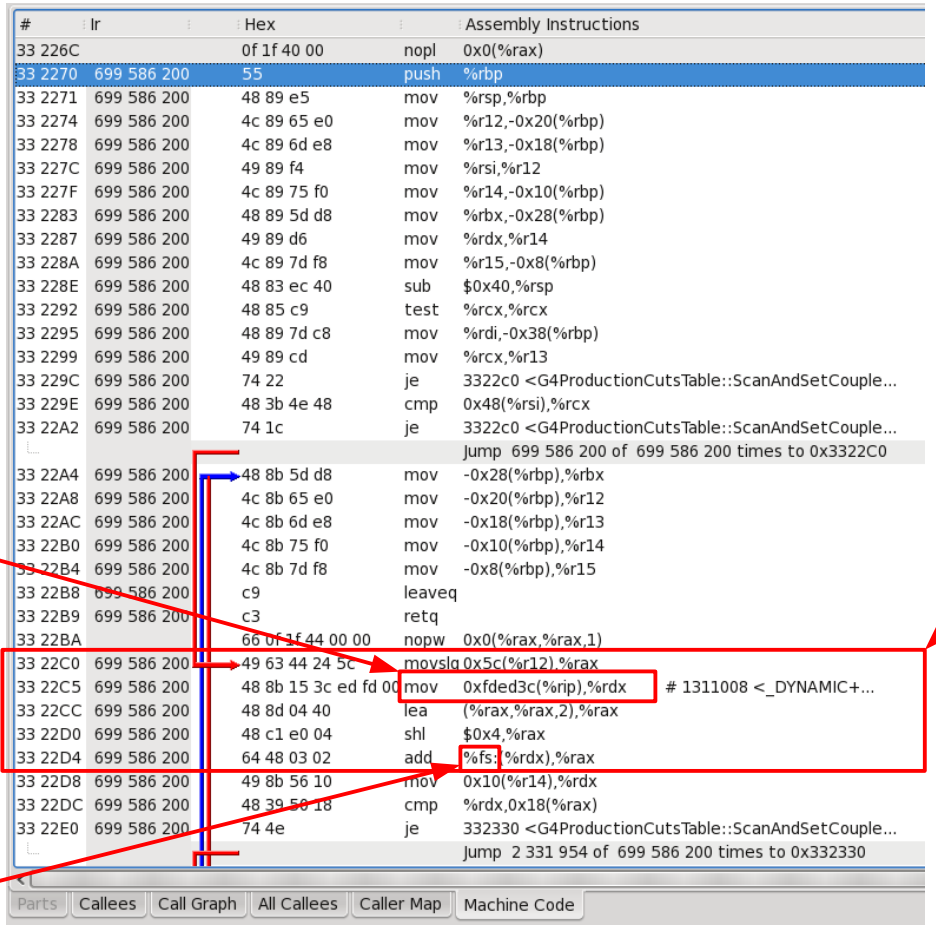
Five registers are restored before return from this function

Branch never taken

Branch always taken

Static branch prediction hint is wrong

Machine code view (first part)



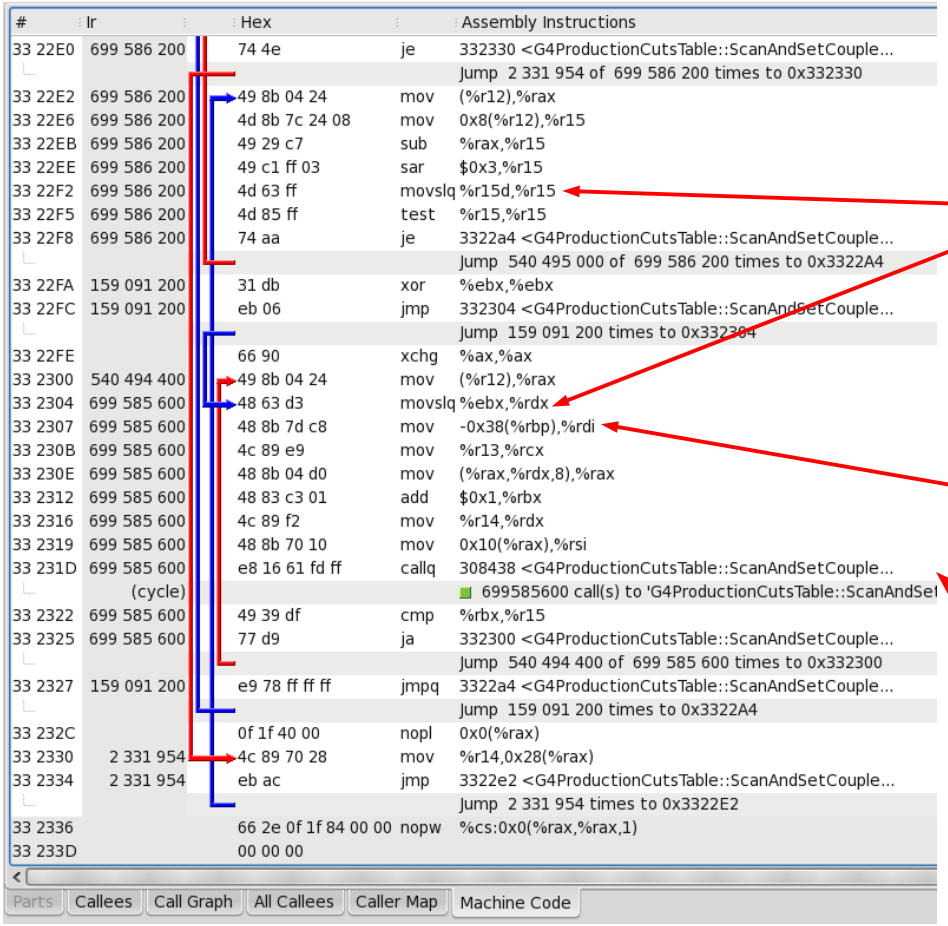
Dynamic constant fetched from memory

Code for aLV->GetMaterial()

%fs is a segmentation register! It's used for thread-local storage on x86-64

C++ tends to hide implementation details...

Machine code view (second part)



Optimization opportunities

```
void G4ProductionCutsTable::ScanAndSetCouple(G4LogicalVolume* aLV,
                                             G4MaterialCutsCouple* aCouple,
                                             G4Region* aRegion)
{
  //Check whether or not this logical volume belongs to the same region
  if((aRegion!=0) && aLV->GetRegion()!=aRegion) return;

  //Check if this particular volume has a material matched to the couple
  if(aLV->GetMaterial()==aCouple->GetMaterial()) {
    aLV->SetMaterialCutsCouple(aCouple);
  }

  size_t noDaughters = aLV->GetNoDaughters();
  if(noDaughters==0) return;

  //Loop over daughters with same region
  for(size_t i=0;i<noDaughters;i++){
    G4LogicalVolume* daughterLVol = aLV->GetDaughter(i)->GetLogicalVolume();
    ScanAndSetCouple(daughterLVol,aCouple,aRegion);
  }
}
```

Redundant check that should be done outside recursion

This is always true in the benchmark

This result should be cached since it's always the same

This early return is redundant but compiler already optimizes it nicely

Getting rid of the recursion

Optimization opportunities

```
void G4ProductionCutsTable::ScanAndSetCouple(G4LogicalVolume* aLV,
                                             G4MaterialCutsCouple* aCouple,
                                             G4Region* aRegion)
{
  //Check whether or not this logical volume belongs to the same region
  if((aRegion!=0) && aLV->GetRegion()!=aRegion) return;

  //Check if this particular volume has a material matched to the couple
  if(aLV->GetMaterial()==aCouple->GetMaterial()) {
    aLV->SetMaterialCutsCouple(aCouple);
  }

  size_t noDaughters = aLV->GetNoDaughters();
  if(noDaughters==0) return;

  //Loop over daughters with same region
  for(size_t i=0;i<noDaughters;i++){
    G4LogicalVolume* daughterLVol = aLV->GetDaughter(i)->GetLogicalVolume();
    ScanAndSetCouple(daughterLVol,aCouple,aRegion);
  }
}
```

size_t is unsigned 64-bit integer

32-bit integer returned, implicit cast to 64-bit

GetDaughter takes a 32-bit integer as the parameter, implicit cast to 32-bit