2007 Tel-Aviv Roadshow

Core™2 Micro Architecture: The Practical View with VTune™

Software and Solutions Group

Koby Gottlieb, Principle Engineer

Anat Shemer, Senior Engineer, MSP/SPD

Zvi Danovich, Senior Application Engineer

November 21, 2007

Agenda

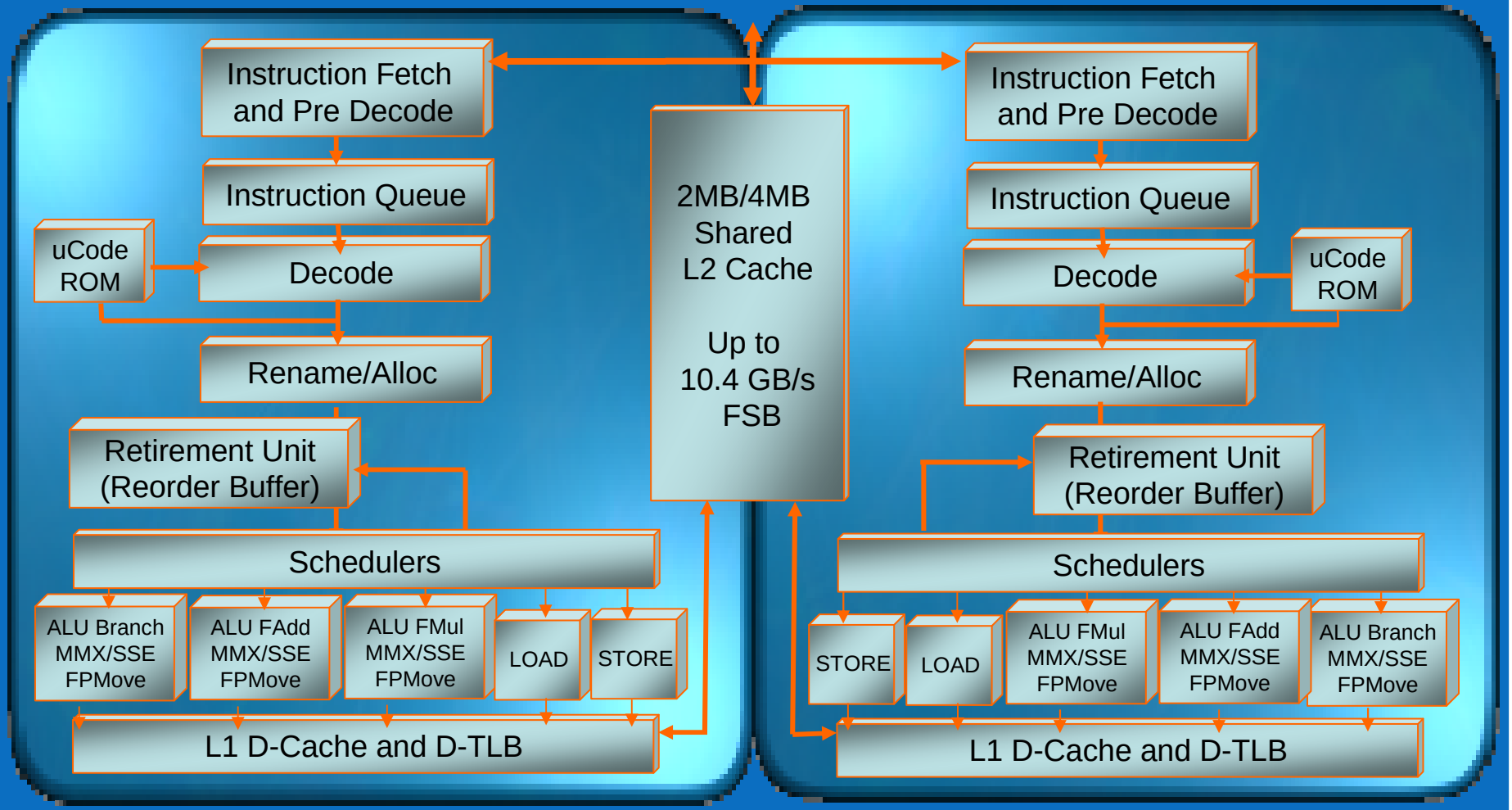
- Core™2 Duo overall micro architecture
- Front end
  - description
  - main events
  - Performance issues
- Execution
  - description
  - main events
  - Performance issues
- Memory Sub-System
  - description
  - main events
  - Performance issues
- Summery and Call for action.

Your need to understand micro architecture

- The Intel® Core™2 micro architecture gives you a big technological advantage in most of the applications (in comparison with previous one).
- Still, good understanding of the micro architecture with the right tools to analyze it's behavior is critical if you want to improve your competitive advantage.
- By tuning your favorite code to the Core™2 and avoiding some of the micro architecture pitfalls you might increase the comparative edge of your products compared to your competitor.
- The takeaway from this lecture should be better understanding of the Core™2 micro architecture and understanding of VTune™ events capabilities in order to debug and understand performance issues.

Micro architecture tuning is the key for differentiation

Core™2 Micro architecture



Microarchitecture main components

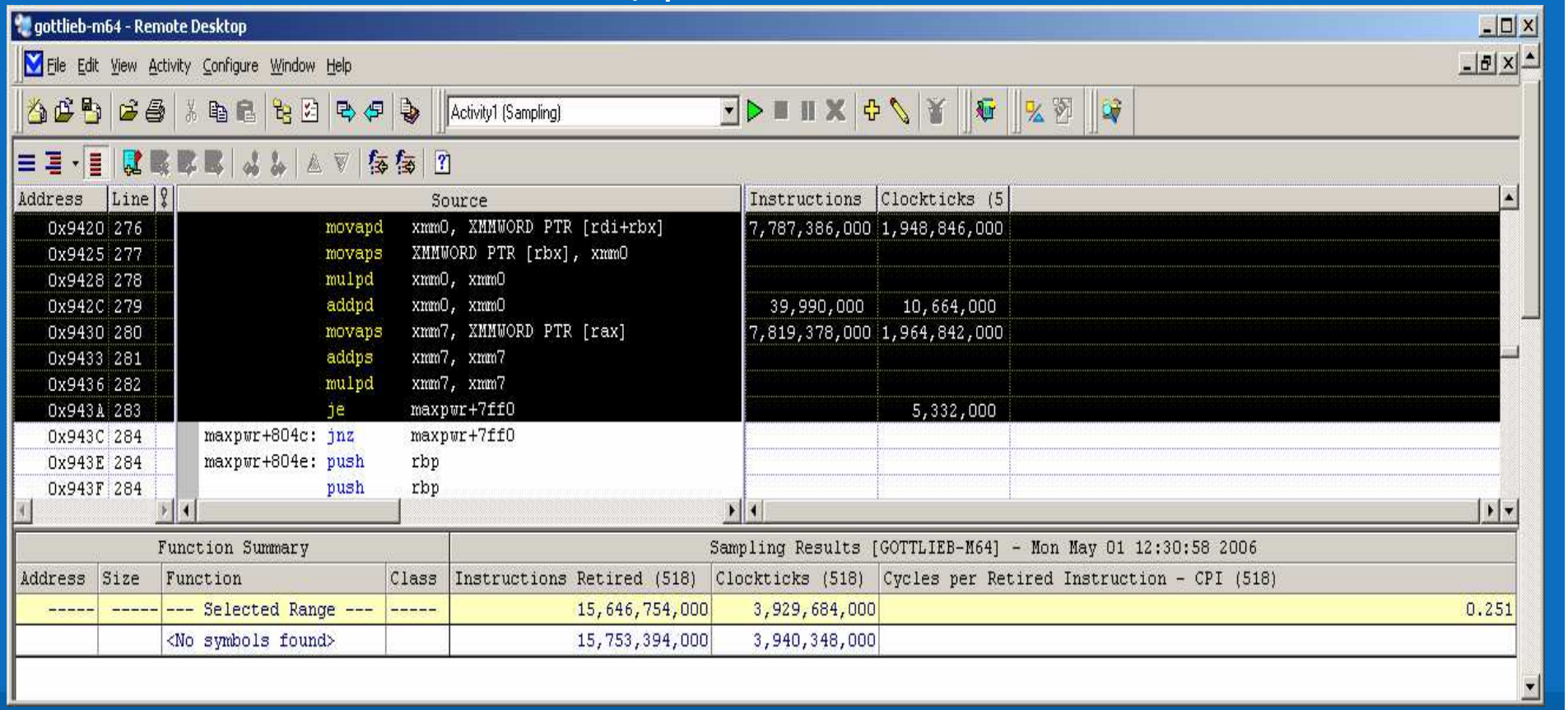
- Intel® Wide Dynamic Execution
  - Each core can fetch, dispatch, execute and retire up to 4 instructions per cycle
  - Up to 4-wide instruction decode unit
  - 14-stage efficient pipeline
  - System bus of up to 1333Mhz
- Front end
  - Macro-fusion, Micro-fusion
  - Stack pointer tracker
  - Advanced branch prediction
- Execution:
  - Three full arithmetic logical units (ALUs)
  - Single-cycle throughput of 128-bit SIMD instructions, up to eight floating point operations per cycle (1x4 Add + 1x4 Mult)

Microarchitecture main components (cnt.-ed)

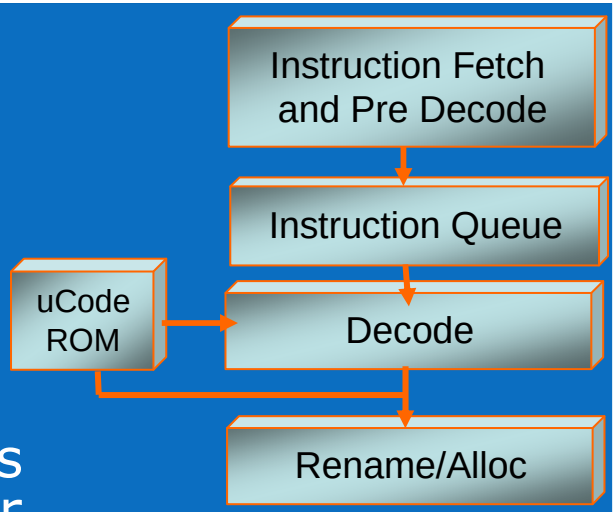
- Memory and multi core:
  - Intel® Advanced Smart Cache
  - Shared second level (L2) 2MB 8-way or 4MB 16-way instruction and data cache
  - Higher bandwidth from the L2 cache to the core
  - Intel® Smart Memory Access
  - Improved prefetching
  - Memory disambiguation
- Intel® Intelligent Power Capability
  - Advanced power gating
  - Split bus arrays
  - Dynamic power coordination

The perfect example, No stalls

- 4 instructions every cycle:
  - Cycle 1: load, store, mulpd and addps
  - Cycle 2 load, addps, mulps and JE
- Zero stalls in all events, perfect balance



Front end description



- 32 Kb Instruction Cache and Instruction Translation Lookaside Buffer (ITLB).
- A hit in the instruction cache causes 16 bytes to be delivered to the instruction predecoder.
- Instruction length decoder marking to split the byte stream into discrete instructions
- As many as six instructions per cycle can be sent to the instruction queue (IQ)
  - Typical programs average slightly less than 4 bytes per instruction, depending on the code being executed.
  - Since most instructions can be decoded by all decoders, an entire fetch can often be consumed by the decoders in one cycle.
- Front end main events:
  - L1 I Cache (32kb): L2_IFETCH. SELF
  - Decoding events: MACRO_INSTS.DECODED

Loop Stream Detector (LSD)

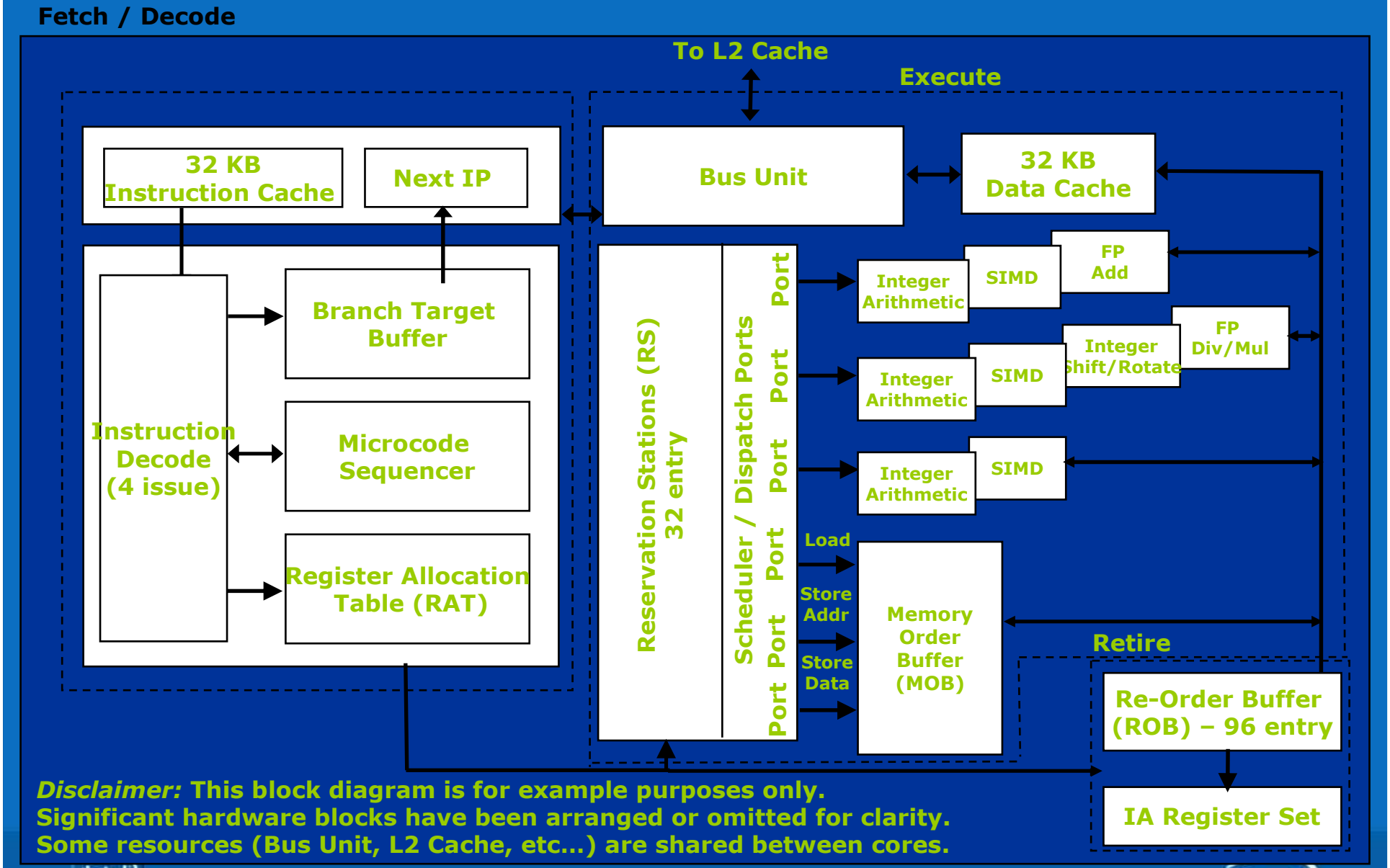
- Detects short loops and locks them in the instruction queue (IQ)
- Sends (replays) instructions from the IQ to the decoder
  - Instruction cache, Branch Prediction Unit (BPU) and pre-decode units are idle
- Benefits:
  - Reduced front end power consumption - total saving of up to 14%
  - Performance gain only if the code is front end bound.
    - Low RESOURCE_STALLS.RS_FULL count and high Cycles per Instruction (CPI)
  - No loss of bandwidth due to
    - taken branches
    - misaligned instructions
    - Length Changing Prefixes (LCP) penalties
- Loops detected by the LSD
  - Require up to 4x16-byte fetches (of code)
  - Have up to 18 instructions.
  - Contain up to 4 taken branches, none of them is RET
  - Executed for more than 64 iterations.

Front end issue debugging

- In order to identify front end issues look at RESOURCE_STALLS.RS_FULL.
  - Reservation station (RS) is the front end and allocation target.
  - If there are no issues in the front end the RS should be full above 30% of the time or the CPI should be very high (perfect execution)
  - Small number of RESOURCE_STALLS.RS_FULL events and poor CPI should be debugged as front end issue
- Front end typical issues:
  - Code is too big to fit in the L1: L2_IFETCH. SELF high number.
    - If there is an event every 10-15 instructions the loop work set is bigger than 32K. Code that could have been with CPI 1 will be around 2.
      - The instruction prefetcher works only every other line in average. 14 cycles penalty for L1 demand miss.
  - Average instruction size above 6 bytes.
    - Happens typically with SSE code and more with EM64T.
      - Typical EM64bit memory instruction:
        - » REX F2 0F XX MOD/RM SID XX XX XX XX → 10 bytes .
    - Can have impact only in case of otherwise excellent CPI
  - Code with length changing prefix issues (LCP)
    - Look at ILD_STALL. This event counts the number of cycles during which the instruction length decoder requires 6 cycles. Caused by operand override prefix (66h) preceding an instruction with immediate data.

Front should not be the bottleneck.
Focus on front end issues only if it is the issue.

Architecture Block Diagram: Execute stage



The major components of execution stage

- **Renamer:**
  - Moves instructions from the front end to the execution core.
  - Architectural registers are renamed to a larger set of micro-architectural registers.
  - Renaming eliminates false dependencies known as read-after-read and write-after-read hazards.
  - Allocates uOps into the ROB (Reorder Buffer) to track the progress of a uOp from issue to completion/cancellation
- **Reorder buffer (ROB):**
  - Holds micro-ops in various stages of completion.
  - Buffers completed micro-ops, updates the architectural state in order, and manages ordering of exceptions.
- **Reservation station (RS)**:
  - Queues micro-ops until all source operands are ready.
  - Schedules and dispatches ready micro-ops to the available execution units in as close to a first in first out (FIFO) order as possible.
- **Ports and execution units:**
  - Execute the instructions and pass the results.
  - Write back the ROB

Execution micro architecture

- Up to six uOps can be dispatched per clock
- Up to four results can be written back per clock
- Single clock latency operations are best
  - Differing latency operations can create writeback conflicts → Creating bubble in the port
  - Use RS_UOPS_DISPATCHED.PORT_*[0-5]* to understand port utilization.

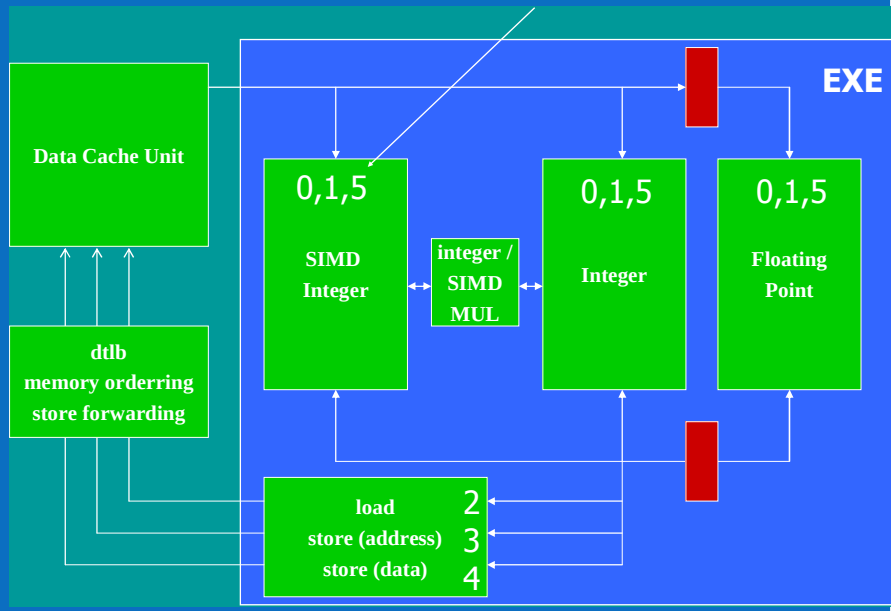
  The busiest port should determine the potential execution speed.
- Look at the dependency chains to see the potential parallelism.
  - Remember that the RS has only 32 entries and only those instructions are candidates for scheduling to the execution ports.
  - RESOURCE_STALLS.RS_FULL should have high percentage if the code is latency bound
  - The ROB has 96 entries so RESOURCE_STALLS.ROB_FULL should have high percentage only if the code had long latency instructions (L2 misses) and other code that can be executed while waiting.

Execution stage: The key for good performance. Focus on port utilization and dependency chains

Software

Execution micro architecture

- The Divider is a big potential stall source.
  - Look at DIV for Divide operations executed and even more at IDLE_DURING_DIV counts cycles of no port issue while the divider is busy.
    - Try to find some useful work to do in parallel with those divide operations
- Be aware of the instructions with several uOps
  - Typical instructions here:
    - ADC/SBB.
    - RWM, CMOVcc
    - shuffle and pack/unpack. floating point float to double converts and more.
- Extra cycle latency for bypass *between* execution domains
  - For example: FP (ADDPS) and logical ops (PAND) on XMM*n*
    - DELAYED_BYPASS.FP
    - DELAYED_BYPASS.LOAD
    - DELAYED_BYPASS.SIMD



Register Allocation Table (RAT) potential issues

- There is a general event RAT_STALLS.ANY for all the RAT stalls.
- In case of high number we should look at:
  - Flag stall can be the most critical issue as the penalty is very high. Look at the events:
    - RAT_STALLS.FLAGS Flag stall cycles.
    - RAT_STALLS.FLAGS_COUNT Flag stall events.
  - Partial stall cause a 3 cycle bubble in the RAT and adition of merge uop to combine the register parts. Look at the events:
    - RAT_STALLS.PARTIAL_COUNT Partial register stall events.
    - RAT_STALLS.PARTIAL_CYCLES Partial register stall cycles.
  - In case of code that uses many unchanged registers or too aggressive unrolling with good scheduling between the iterations we can see big number of ROB stalls
    - RAT_STALLS.ROB_READ_PORT ROB read port stalls cycles.

Flag stall example

Source	RAT_STALLS.FLAGS	INST_RETIRED.	CPU_CLK_UNHALT
add eax,0 // clear carry			
L:			
mov edx,[esi+ecx*4]	822,000,000	997,084,000	1,055,736,000
adc edx,[edi+ecx*4]			2,666,000
mov [ebx+ecx*4],edx	9,487,000,000	4,036,324,000	12,055,652,000
inc ecx	1,176,000,000		930,434,000
jle L			12,055,652,000

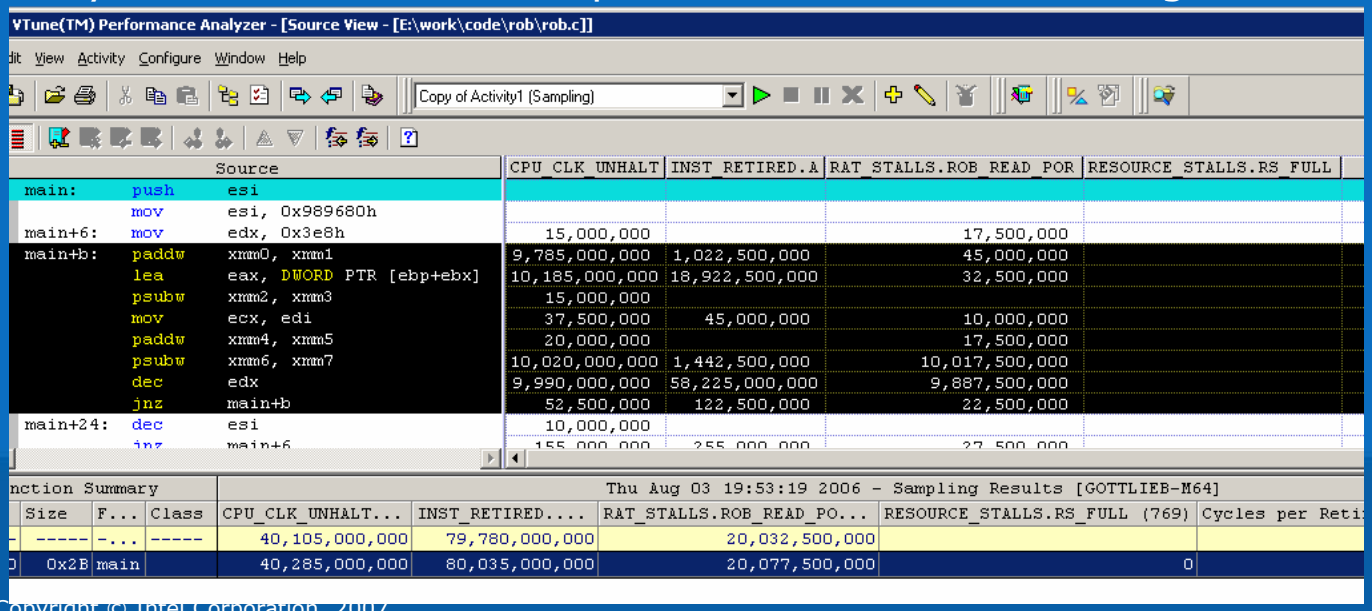
- CPI 2.78 and 14 cycles per iteration of a long add as a result of flag stall between the inc and the adc
- The fixed code is only 4.3 cycles per iteration (CPI 0.61) with no flag stalls.

```
 align 16
L:
   add dl,0xff
   mov eax,[esi+ecx*4]
   adc eax,[edi+ecx*4]
   mov [ebx+ecx*4],eax
   setc dl
   inc ecx
   jle L
```

Source	RAT_STALLS.FLA	INST_RETIRED.	CPU_CLK
align 16			
L:			
add dl,0xff		2,167,458,000	991,752,000
mov eax,[esi+ecx*4]		77,314,000	47,988,000
adc eax,[edi+ecx*4]			
mov [ebx+ecx*4],eax		2,604,682,000	2,111,472,000
setc dl		2,034,158,000	1,042,406,000
inc ecx		90,644,000	53,320,000
jle L			
pop ebx			

Execution Performance Issues

- Limited amount of ROB ports:
  - Only 2 regular plus one index registers can be read from the ROB
    - Bypass register "access" preferred to register reads
    - Registers that have become cold and require a ROB read port because execution units are doing other independent calculations.
  - Look at RAT_STALLS.ROB_READ_PORT event. This event counts the number of cycles when ROB read port stalls occurred, which did not allow new micro-ops to enter the out-of-order pipeline.
    - Look at RESOURCE_STALLS.RS_FULL. ROB port should not be an issue if the RS is full anyway
  - In these cases the micro-ops will retry entering the execution pipe in the next cycle and the ROB-read-port stall will be counted again.



Memory architecture

- 32K DCU (data cache unit, L1) per core
  - Load latency 3, throughput 1
    - 20 store buffer entries
    - 32 load buffer entries
      - RESOURCE_STALLS.LD_ST – uop allocation stalls due to lack of entries
    - 8 Line fill buffer, for L1 miss (mainly)
      - LOAD_BLOCK.L1D – load is blocked when LFB is full
  - MEM_LOAD_RETIRED.L1D_MISS – Load operations that missed the DCU
  - 2 HW prefetchers (to be continued…)
  - Disambiguation mechanism (to be continued…)
- 2 level DTLB (Data translation lookaside buffer)
  - L0 for loads only – 8 entries (32K)
    - L0 load miss / L1 hit penalty – 2 cycles
      - DTLB_MISSES.L0_MISS_LD
  - L1 – 256 entries (1M) + 8 entries for large pages
      - DTLB_MISSES.[MISS_LD | MISS_ST | ANY]
      - MEM_LOAD_RETIRED.DTLB_MISS
      - PAGE_WALKS.[COUNT | CYCLE]
- 4M shared L2
  - Load latency 14, throughput 5
  - 2 HW prefetchers (DPL - Data Prefetch Logic + streamer)
  - MEM_LOAD_RETIRED.L2_MISS – Load operations that missed the L2 cache

DCU (L1 cash) Enhancements

- Instruction Pointer (IP) Prefetcher
  - Identifies stride loads associated with the same IP
  - Predicts next loaded address and prefetch if not in the DCU
    - History array: 256 entries, indexed by 8 Least Sign Bits (LSB) of load IP
    - Possible reasons for not working
      - Dense stream of loads doesn't allow prefetches
      - Address collision with another load(s)
      - Use MEM_LOAD_RETIRED.L1D_MISS to identify misses when expected prefetches
    - L1D_PREFETCH.REQUESTS – prefetch (IP+HW) requests to the L2
- Memory Disambiguation
  - Allows load to precede stores with unknown address based on prediction
  - Machine nuke and load restart if disambiguation fails
  - Disambiguation is disabled if many disambiguation frequently fails
    - MEMORY_DISAMBIGUATION.[SUCCESS | RESET]
  - Possible reasons for not working
    - Address collision with another load(s)
      - LOAD_BLOCK.STA

Improved load latency for stride and close accesses

Other Opportunities for Performance Gain in the memory sub-system

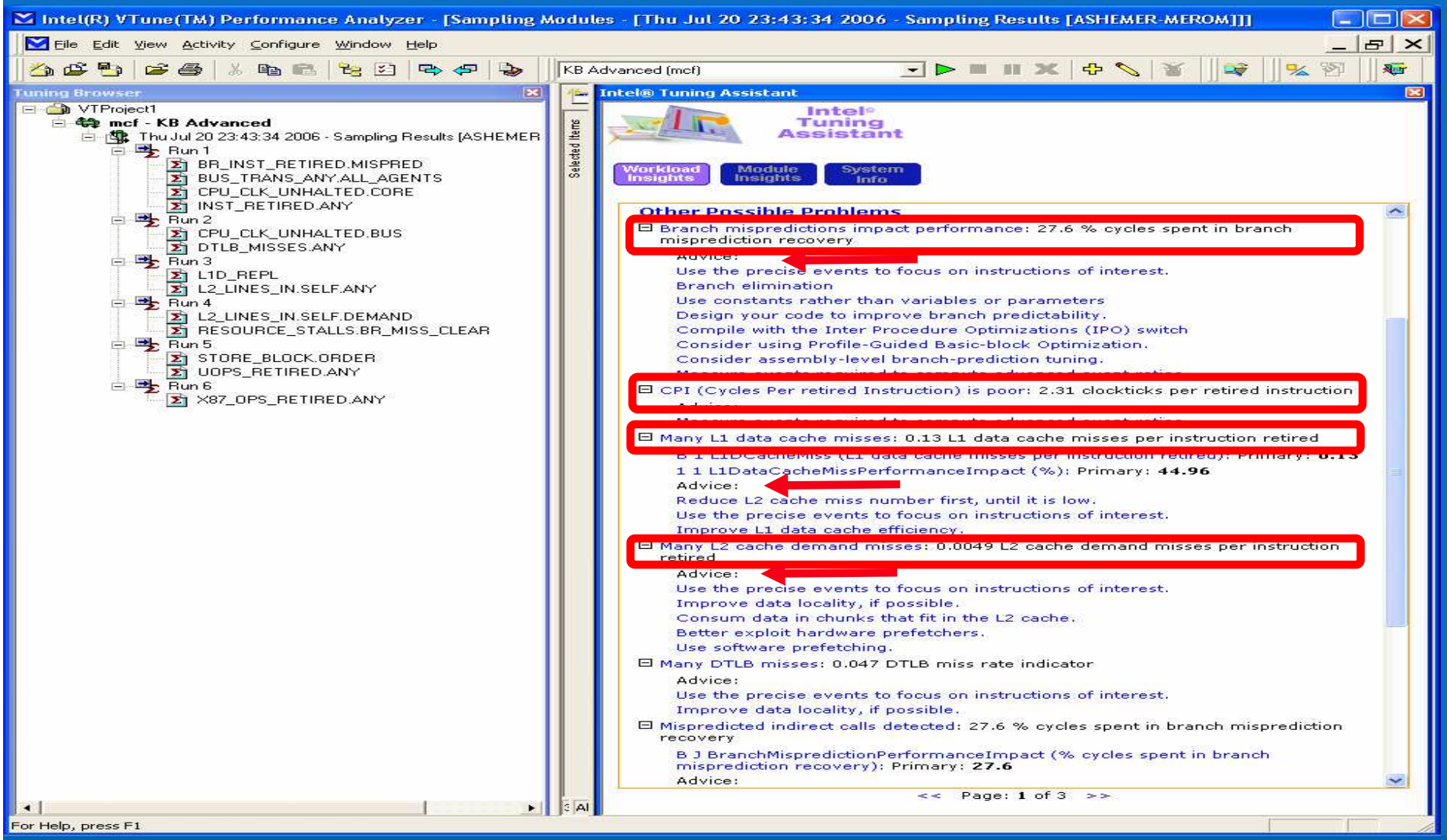
- Store L2 misses
  - When a senior store misses the L2 following stores are blocked
    - Until all other bus agents invalidate the stored cache line
      - STORE_BLOCK.ORDER
- Load block cases
  - Store address unknown - LOAD_BLOCK.STA
  - Store data unknown - LOAD_BLOCK.STD
  - 4K aliasing and unaligned load after store LOAD_BLOCK.OVERLAP_STORE
  - Split loads and uncacheable LOAD_BLOCK.UNTIL_RETIRE
  - Many cache line missed: LOAD_BLOCK. L1D
- SW Prefetching
  - Can be effective upon many L2 demand misses
    - L2_LD.SELF.DEMAND.MESI / INST_RETIRED.ANY > 0.004
  - Verify that SW prefetch instruction is located properly
    - Prefetched data is not in memory:
      - SSE_PRE_MISS.[NTA | L1 | L2] / SSE_PRE_EXEC.[NTA | L1 | L2] == 1
    - Load operation accesses data only when it's in the cache already
      - LOAD_HIT_PRE / CPU_CLK_UNHALTED == 0

Many new events help mapping issues

Precise Events

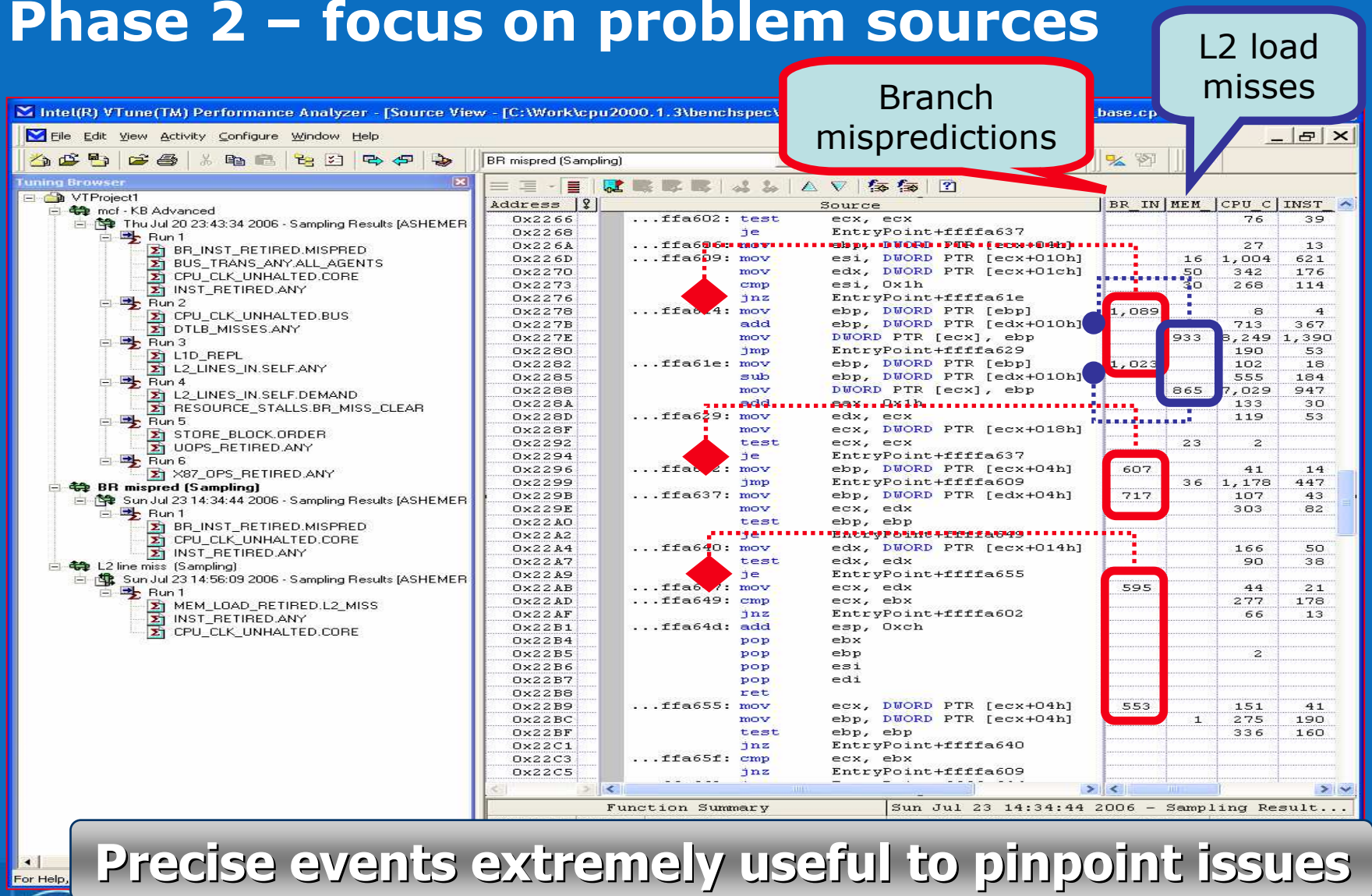
- *Definition: Event is associated with the instruction that caused it*
- Collected state contains
  - Address of the instruction following the one that caused the event
  - GP register + RFLAGS state
- Counted by programmable counter 0 only
- Events:
  - MEM_LOAD_RETIRES.[L1D_MISS | L1D_LINE_MISS | L2_MISS | L2_LINE_MISS | DTLB_MISS]
  - BR_INST_RETIRED.MISPRED – point to branch target!
  - INST_RETIRED.ANY_P
  - X87_OPS_RETIRED.ANY
  - SIMD_INST_RETIRED.ANY

Example 1 –Traversing a Tree
Phase 1 – identify main slow-down reasons



Tuning assist should help identifying issues

Example 1 –Traversing a Tree
Phase 2 – focus on problem sources



Precise events extremely useful to pinpoint issues

Summary/Call to Action

- In most cases front end should not be the bottleneck. Focus on front end issues only if it is the issue.
- Execution stage: The key for good performance. Focus on port utilization and dependency chains.
- Memory: Improved load latency for stride and close accesses
- VTune™:
  - Precise events extremely useful to pinpoint issues
  - Tuning Assist should help identifying issues

There are many new and existing events that can help tuning when you have good understanding of the micro architecture.
Learning them can help gaining
more performance out of the Core™2–based CPU's.

Useful link

There are several well detailed manuals describing micro architecture, programming set and optimization issues here:

http://www.intel.com/products/processor/manuals/index.htm


intel®
Leap ahead™