Use hardware counters in performance tuning

Different view of performance

- Time

Different view of performance

- Cache miss

Different view of performance

- L2 cache miss

Different view of performance

- TLB miss

Measure time accurately

- Initializing application state
  - Flushing cache, TLB (non-trivial)
  - Read/write large chuck of memory
  - Pay attention to cache associativity, cache line size
  - Variable TLB configuration
- Execute programs multiple times

Accurate timer

- Cycle-accurate timer depends on architecture
- OS-level timer usually is accurate down to 1ms
  - gettimeofday: has usec field. But aligned to ms
  - Perturbed by system call overhead

High precision timer on IA32

- Need 64-bits numbers
- Record number of ticks since processor reset
- Caution: TSCs on multi-cores in a processor do not guarantee to be in sync

TSC example

- typedef union
- {
  - unsigned long long u64;
  - struct
  - {
    - unsigned long u32lo;
    - unsigned long u32hi;
  - };
  - double d;
- } Big;

- #define RDTSC(a) __asm__ __volatile__ ("rdtsc" : "=a" (a.u32lo), "=d" (a.u32hi))

TSC example

- Big begin, end
- RDTSC(begin);
- For(…).
  - …
- RDTSC(end);
- Wall_clock_time = (end-begin)/cpu_freq

Hardware counters

- Small set of registers that count events, specific signals in a processor
- Almost all modern microprocessors and computer systems support hardware counters
  - Usually 2-8 counters
- Low-overhead access to detailed observations for program execution
  - Pipelined function units
  - Multiple functional units
  - Speculative execution: branch prediction, …
  - Memory hierarchy,
  - Cache lines shared between SMP
  - I/O, …

Access hardware counters

- Platform-specific
  - AIX/Power: hpmcount, libhpm, PMAPI
  - Solaris: collect/analyzer,
  - Linux: perfctr, oprofile, perf
  - Assembly instructions
- Portable solution
  - PAPI
  - PCL (no longer updated?)
- Have own limitations
  - System specific
  - Difficult to compare cross-platform

Instruction count and functional unit status

- Total cycles
- Total instructions
- Floating point operations
- Load/store instructions
- Cycles functional units are idle
- Cycle stalled
  - Waiting for memory access
  - Waiting for resource
- Conditional branch instructions
  - Executed
  - Mispredicted

Counters for memory hierarchy

- Register
- L1: instruction and data
- L2:
- Shared cache
- Main memory
- Remote memory

Relevant hardware counters

- Cache line requested
- Cache misses
- Cache hit
- Cache line loaded
- Cache line prefetched
- Cache line invalidated
- TLB misses

Usage of hardware counters

- Lowest level: assembly instructions
  - Example 1: Resource stall on
  - 0xA2 | (1 << 16) | (1 << 22) Pentium-M
  - Example 2: Total number of instructions
  - 0xC0 | (1 << 16) Pentium-M
  - (2 << 25) | (1 << 9) | (1 << 2) Intel P4 Xeon
- High-level interface
  - OS-level library: perfctr, libpmc
  - Cross-platform library: PAPI

PAPI (From PAPI tutorial)

- Goal:
  - Solid foundation for cross platform performance analysis tools
  - Standardization between vendors, academics and users
- A portable and efficient API to access the hardware performance monitor counters found on most modern microprocessors.

Operating System	Processor	Driver	Notes
AIX 5.x	IBM POWER5, POWER6	bos.pmapi	xlc 6+
AIX 6.x	IBM POWER7	bos.pmapi	under development
Cray Linux Environment 2.x, 3.x	Cray XT{3 - 6}, XE{5, 6}	none	
Compute Node Kernel	IBM Blue Gene P	none	
FreeBSD	x86, x86_64 (Intel, AMD)	HWPMC driver	
Linux	x86, x86_64 (Intel, AMD)	PerfCtr 2.6.x	kernel 2.6.x
	x86, x86_64 (Intel, AMD)	Perfmon2	kernel 2.6.30 and below
	x86, x86_64 (Intel, AMD), ARM, MIPS	perf_events	kernel 2.6.32 and above
	Intel Itanium II, Montecito, Montvale	none	
		Perfmon2	Linux 2.6.30 and below
	IBM POWER4, 5, 6, 7	PerfCtr 2.7.x	
		perf_events	kernel 2.6.32 and above
	IBM PowerPC970, 970MP	PerfCtr 2.7.x	
Solaris 8, 9	UltraSparc I, II & III	none	
Solaris 10	Niagara 2	none	

Three levels of interface

- The low level interface manages hardware events in user defined groups called *EventSets.*
- The high level interface simply provides the ability to start, stop and read the counters for a specified list of events.
- Graphical tools to visualize information.

Low-level interface

- Increased efficiency and functionality over the high level PAPI interface
- About 40 functions
- Obtain information about the executable and the hardware
- Thread-safe
- Fully programmable
- Callbacks on counter overflow

Low-level interface example

```
#include "papi.h“
#define NUM_EVENTS 2
int Events[NUM_EVENTS]={PAPI_FP_INS,PAPI_TOT_CYC}, EventSet;
long_long values[NUM_EVENTS];
/* Initialize the Library */
retval = PAPI_library_init(PAPI_VER_CURRENT);
/* Allocate space for the new eventset and do setup */
retval = PAPI_create_eventset(&EventSet);
/* Add Flops and total cycles to the eventset */
retval = PAPI_add_events(&EventSet,Events,NUM_EVENTS);
/* Start the counters */
retval = PAPI_start(EventSet);

do_work();  /* What we want to monitor*/

/*Stop counters and store results in values */
retval = PAPI_stop(EventSet,values);
```

Multiplexing

- Multiplexing allows simultaneous use of more counters than are supported by the hardware.
- PAPI_multiplex_init()
  - should be called after PAPI_library_init() to initialize multiplexing
- PAPI_set_multiplex(int *EventSet);
  - Used after the eventset is created to turn on multiplexing for that eventset
- Then use PAPI like normal
- Lower precision than non-multiplexed events
- Also used by Sun Collector/Analyzer

High-level interface

- Meant for application programmers wanting coarse-grained measurements
- Not thread safe
- Calls the lower level API
- Allows only PAPI preset events
- Easier to use and less setup (additional code) than low-level

High-level functions

- `int PAPI_start_counters(int *events, int len)`
  - Initializes PAPI (if needed)
  - Sets up an event set with the given counters
  - Starts counting in the event set
- `PAPI_stop_counters(long_long *vals, int  alen)`
  - Stop counters and put counter values in array
- `PAPI_accum_counters(long_long *vals, int  alen)`
  - Accumulate counters into array and reset
- `PAPI_read_counters(long_long *vals, int  alen)`
  - Copy counter values into array and reset counters

Example

```
long long values[NUM_EVENTS];
 unsigned int
Events[NUM_EVENTS]={PAPI_TOT_INS,PAPI_TOT_CYC};
  /* Start the counters */
  PAPI_start_counters((int*)Events,NUM_EVENTS);
  /* What we are monitoring? */
  do_work();
  /* Stop the counters and store the results in values */
  retval = PAPI_stop_counters(values,NUM_EVENTS);
```

High-level vs. low-level

- Better defined events in low-level
  - High-level events may have different semantics on different platforms
- Unsupported events can only be used in low-level interface
- Useful links:

PAPI Overview:

Useful Links

- PAPI Overview:
  http://icl.cs.utk.edu/projects/papi/wiki/PAPIC:Overview
- PAPI Preset Event

  http://icl.cs.utk.edu/projects/papi/wiki/PA

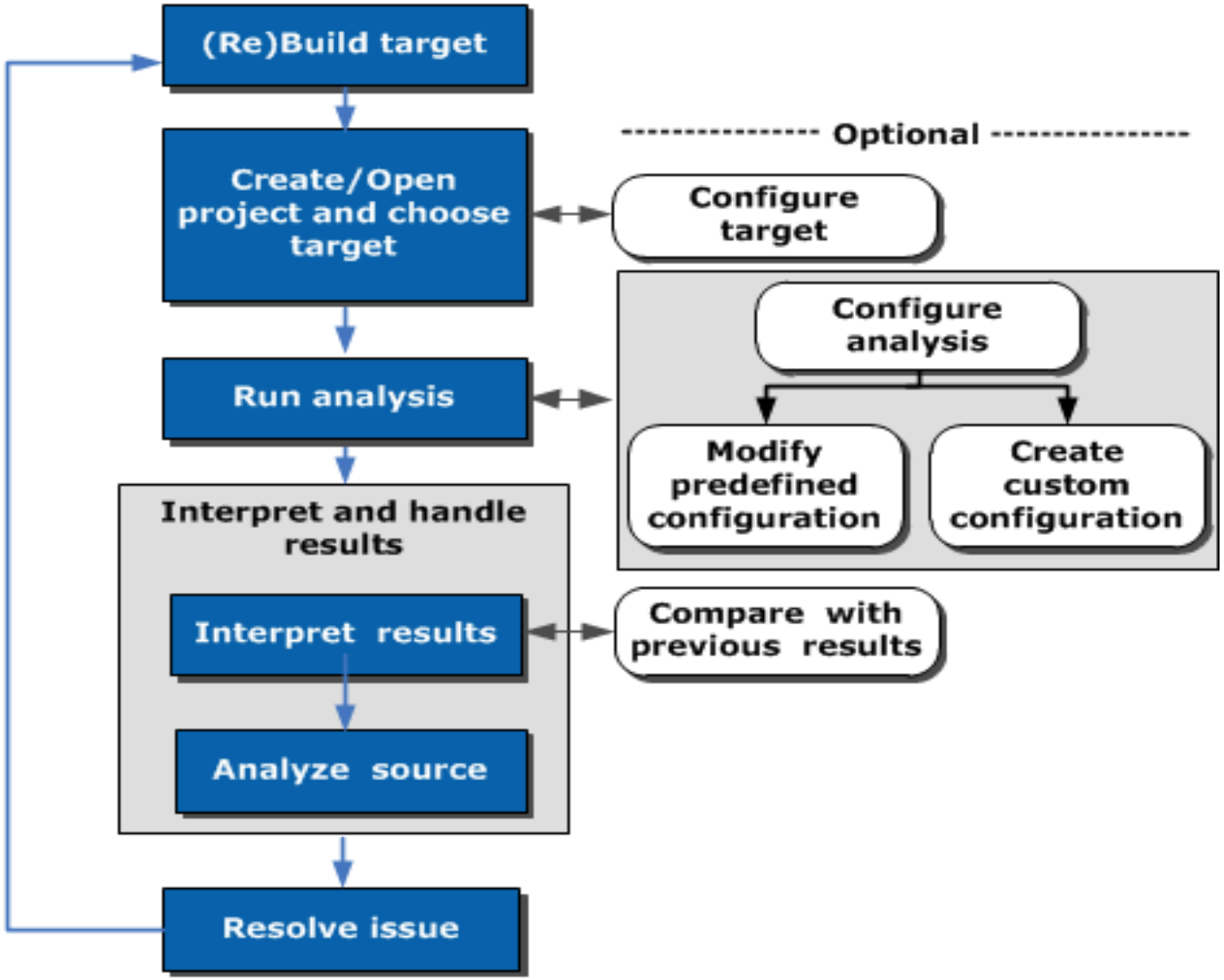
  PIC:Preset_Event_Definitions
- PAPI Documentation

  http://icl.cs.utk.edu/papi/docs/index.html

Intel Vtune: Functionality

- The most time-consuming (hot) functions in your application and/or on the whole system
- Sections of code that do not effectively utilize available processor time
- The best sections of code to optimize for sequential performance and for threaded performance
- Synchronization objects that affect the application performance
- Whether, where, and why your application spends time on input/output operations
- The performance impact of different synchronization methods, different numbers of threads, or different algorithms
- Thread activity and transitions
- Hardware-related bottlenecks in your code

Workflow


(Re)Build target
Create/Open project and choose target
Run analysis
Interpret and handle results
Interpret results
Analyze source
Resolve issue
Optional
Configure target
Configure analysis
Modify predefined configuration
Create custom configuration
Compare with previous results

Data Collection Mode

All analysis types in the VTune Amplifier are based on one of the following data collection types:

- User-mode sampling and tracing collection
- Hardware event-based sampling collection, optionally extended with the stack collection
- Hardware event-based sampling collection for Intel Xeon Phi™ coprocessor

User-Mode Sampling and Tracing(UMST)

- UMST collector embeds an agent library into the profiled application using the LD_PRELOAD variable.

- Average overhead of the UMST collector is about 5% when sampling is using the default interval of 10ms.

- Vtune use UMST for the analysis types:
  *Hotspots
  *Concurrency
  *Locks and Waits

Hardware Event-based Sampling(EBS)

- Using the counter overflow feature of the Performance Monitoring Unit (PMU)
- Average overhead of the EBS collector is about 2% on a 1ms sampling interval.
- Number of simultaneous hardware events is limited by CPU capability. Multiplex may be needed.

Algorithm Analysis vs Hardware-level Analysis

- Algorithm analysis includes the following analysis types: Hotspots, Concurrency, Locks and Waits
- Hardware-level Analysis introduces a set of analysis types based on the EBS collection and targeted for specific microarchitecture/cpu.

Key Concepts

- Data of Interest
- CPU Time and Wait Time
- Hardware Event Skid
- Precise Events
- Sample After Value
- Self Time and Total Time
- Bottom-up Analysis and Top-down Analysis

Data of Interest

The data in the Data of Interest column is used by

- The Call Stack pane calculates the contribution
- The Filter bar uses the data of interest data to calculate the percentage indicated in the filtered option.
- The Source/Assembly window uses this column for hotspot navigation.

CPU Time and Wait Time

- CPU time is the amount of time a thread spends executing on a logical processor. For multiple threads, the CPU time of the threads is summed. The application CPU time is the sum of the CPU time of all the threads that run the application.
- Wait time is the amount of time that a given thread waited for some event to occur, such as synchronization waits and I/O waits.

Hardware Event Skid

- Event skid is the recording of an event not exactly on the code line that caused the event. Event skids may even result in a caller function event being recorded in the callee function.

Reasons include:

- The delay in propagating the event out of the processor's microcode through the interrupt controller (APIC) and back into the processor.
- The current instruction retirement cycle must be completed.
- When the interrupt is received, the processor must serialize its instruction stream which causes a flushing of the execution pipeline.

Precise Events

- Precise events are events for which the exact instruction addresses that caused the event are available.

Details

Events configured for CPU: Intel(R) Core(TM) Processor Family

NOTE: For analysis purposes, Intel VTune Amplifier XE 2013 may adjust the Sample After values in the table below by a multiplier. The multiplier depends on the value of the Duration time estimate option specified in the Project Properties dialog.

Event Name	Sample After	LBR Filter	Event Description
INST_RETIRED.ANY	2000000		Instructions retired (fixed counter)
ITLB_MISS_RETIRED	200000	None	Retired instructions that missed the ITLB (Precise Event)
L1I.CYCLES_STALLED	2000000	None	L1I instruction fetch stall cycles
L1I.MISSES	2000000	None	L1I instruction fetch misses
LOAD_HIT_PRE	200000	None	Load operations conflicting with software prefetches
MACHINE_CLEARS.CYCLES	20000	None	Cycles machine clear asserted
MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD_32	5000	None	Memory instructions retired above 32 clocks (Precise Event)
MEM_INST_RETIRED.LOADS	2000000	None	Instructions retired which contains a load (Precise Event)
MEM_INST_RETIRED.STORES	2000000	None	Instructions retired which contains a store (Precise Event)

Sample After Value(SAV)

- SAV is the frequency with which Vtune interrupt the processor the collect a sample during hardware event-based data collection.
- SAV is measured as the number of events it takes to trigger a sample collection.
- Choose SAV carefully to have best overhead/precision balance.

Bottom-up vs Top-down

- Investigating the impact of single functions is also known as bottom-up analysis.
- Investigating the impact of functions together with their callees is also known as top-down analysis.

Self Time and Total Time

- Self time is the time spent in a particular program unit.

- Total time is the accumulated time that a program unit incurs.

```
Void test() {
  int a = 0;
  a++
  foo(bar());
  a--;
}
```

Vtune API Support

- Timeline panel annotation: Task API, User Event API
- Video apps: Frame API
- Java, .NET, Javascript: JIT Profiling API
- Extension to support analyzing user-defined synchronization object: User-Defined Synchronization API
- Fine control data collection: Collection Control API

Reference

- http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/amplifierxe/lin/lin_ug/index.htm