

Getting the Most out of the Intel® Itanium® Architecture

Compiler Optimization and Performance Tuning

Dr. Gernot Hoyler
Technical Marketing
Intel® EMEA

<mailto:Gernot.Hoyler@intel.com>



Agenda

- Optimizing with the Intel® Compilers
- Using Intel Performance Libraries
- Performance Tuning with Intel VTune™
- Other Optimization Opportunities
- Summary

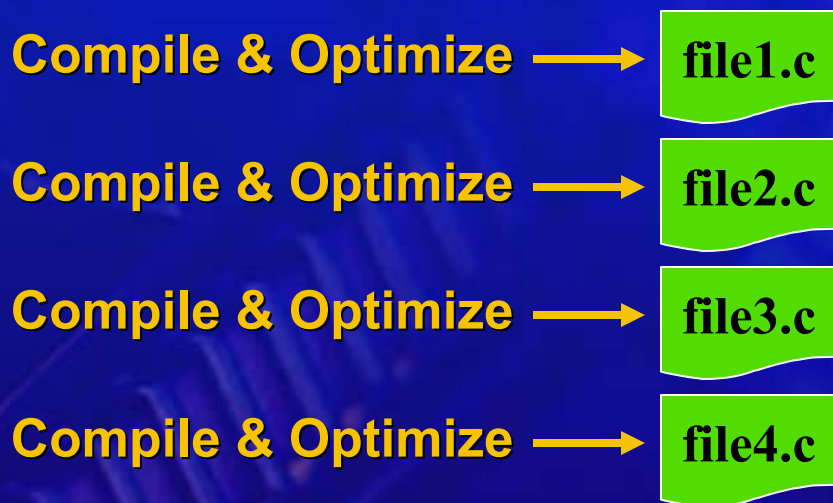
Compiler Optimization Flags

- **-O0**: disables optimization
- **-O1**: optimizes for speed without increasing code size
- **-O2**: optimizes for speed (default)
- **-O3**: enables -O2 plus more aggressive optimizations, may not improve performance for all programs
- **-tpp2**: Itanium® 2 Code Generation (instruction mix)
- **-fno-alias**: assumes no aliasing in program (may be unsafe)
- **-align**: analyzes and reorders memory layout for variables and arrays (FTN only)
- **-pad**: enables changing variable and array memory layout (FTN only)

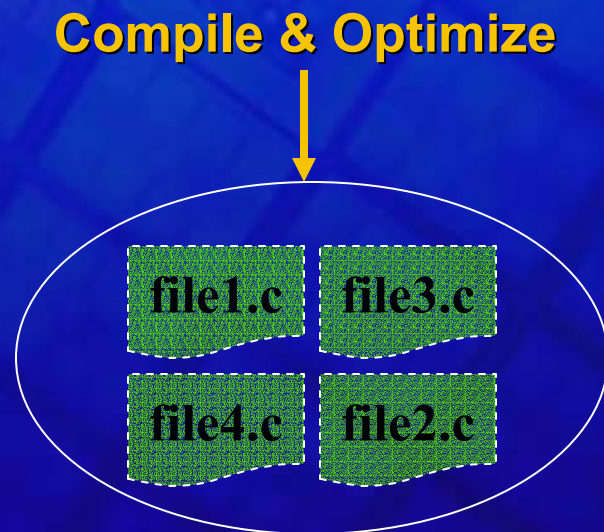
Interprocedural Optimization

Extends optimizations across file boundaries.

Without IPO (or with -ip)



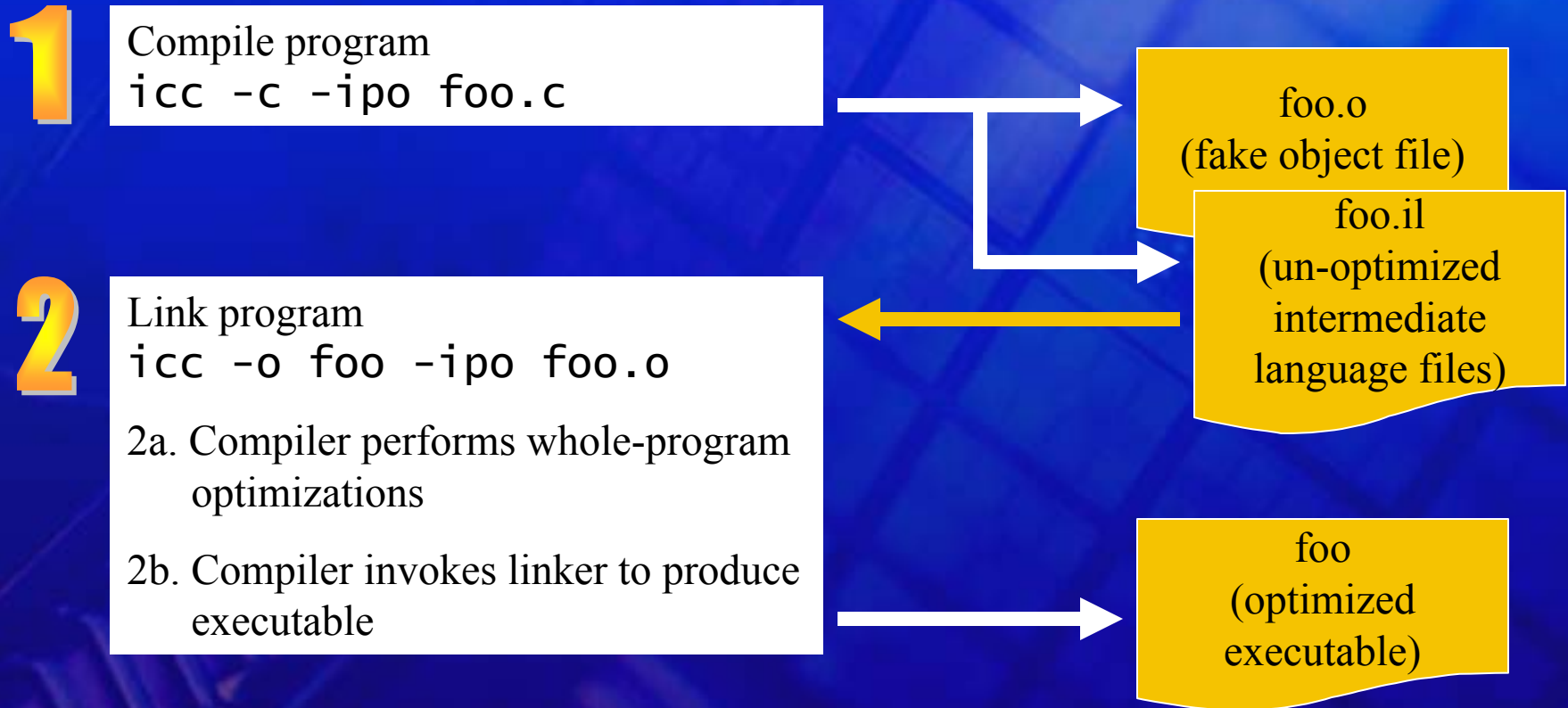
With IPO



IPO Benefits

- Exposes more optimization opportunities (constant propagation, function/data promotion, data layout optimizations, etc.)
- Optimizes function ordering
- Reduces function call overhead by inlining functions across file boundaries
- Can significantly reduce code size through dead code elimination
- Available for both Itanium® and IA-32

How IPO Works



Programs that Benefit from IPO

- Many small utility functions
- Frequent constructor/destructor invocation
- One-liner member functions
- Lynx success story
 - Intel® Spice-like circuit simulator
 - Highly tuned algorithmically
 - Intel compiler (icc) with O2 & IPO:
 - 1.2x - 5.2x speedup over gcc -O (2x typical)
 - 1x - 2.4x speedup over icc -O2 (1.2x typical)

Profile-Guided Optimization

Feed back of profile data gathered during program execution to improve subsequent builds.

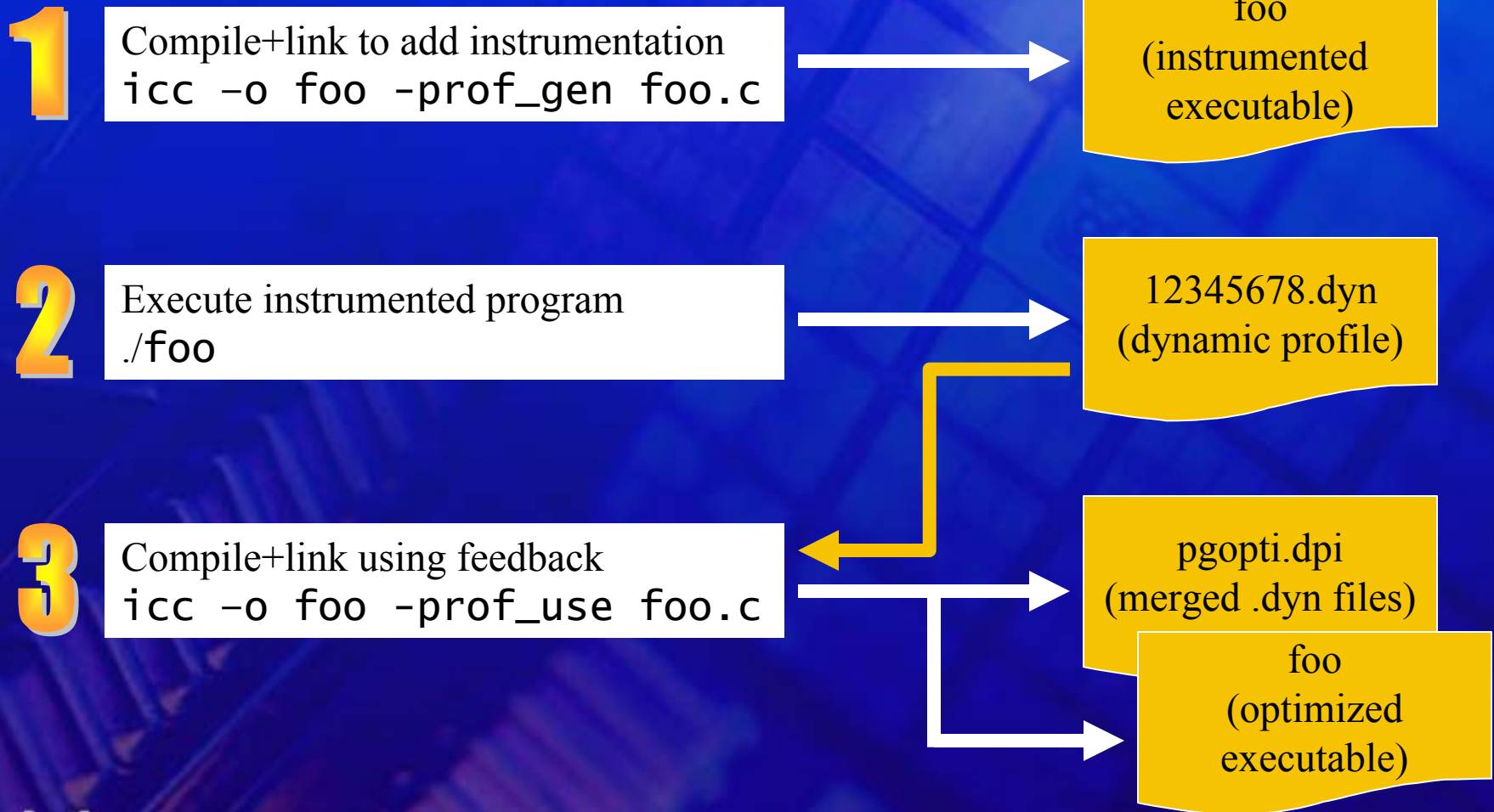
Benefits:

- More accurate branch prediction
- Better register allocation
- Improved IPO inlining
- Basic block movement

```
status = UtilityFunc (arg1, arg2, arg3);  
if (status != 0) // Not expected to fail  
    HandleErr (status);
```

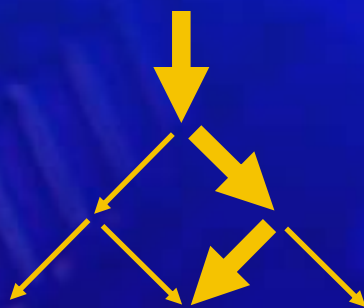
- Improves I-cache behavior
- Available for Itanium® and IA-32

How PGO Works

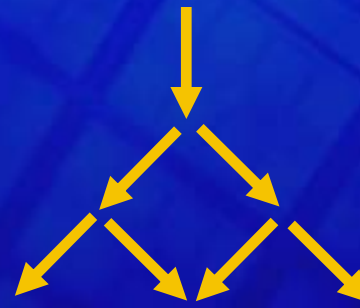


Programs that Benefit from PGO

- Consistent hot paths
- Many if statements or switches
- Nested if statements or switches



VS.



**Significant
Benefit**

**Little
Benefit**

Practical Considerations

- It is not necessary to regenerate profile with every build
- Performance benefits degrade gradually as source code changes
- Regenerate profile when justified
- Source files must stay in same dir from -prof_gen to -prof_use (full source paths recorded in profile files)

PGO makes IPO work better

Compiler Directives/Pragmas

- **!DIR\$ IVDEP** : no loop-carried dependency
- **!DIR\$ SWP** : software-pipeline the loop
- **!DIR\$ NOSWP**
- **!DIR\$ UNROLL (*n*)** : unroll a counted loop
- **!DIR\$ NOUNROLL**
- **!DIR\$ LOOP COUNT (*n*)** : loop count is likely to

Use in conjunction with optimization report flags (-opt_report)

■ **!DIR\$ DISTRIBUTE POINT** : perform loop
intedistribut.

Compiler Directives/Pragmas

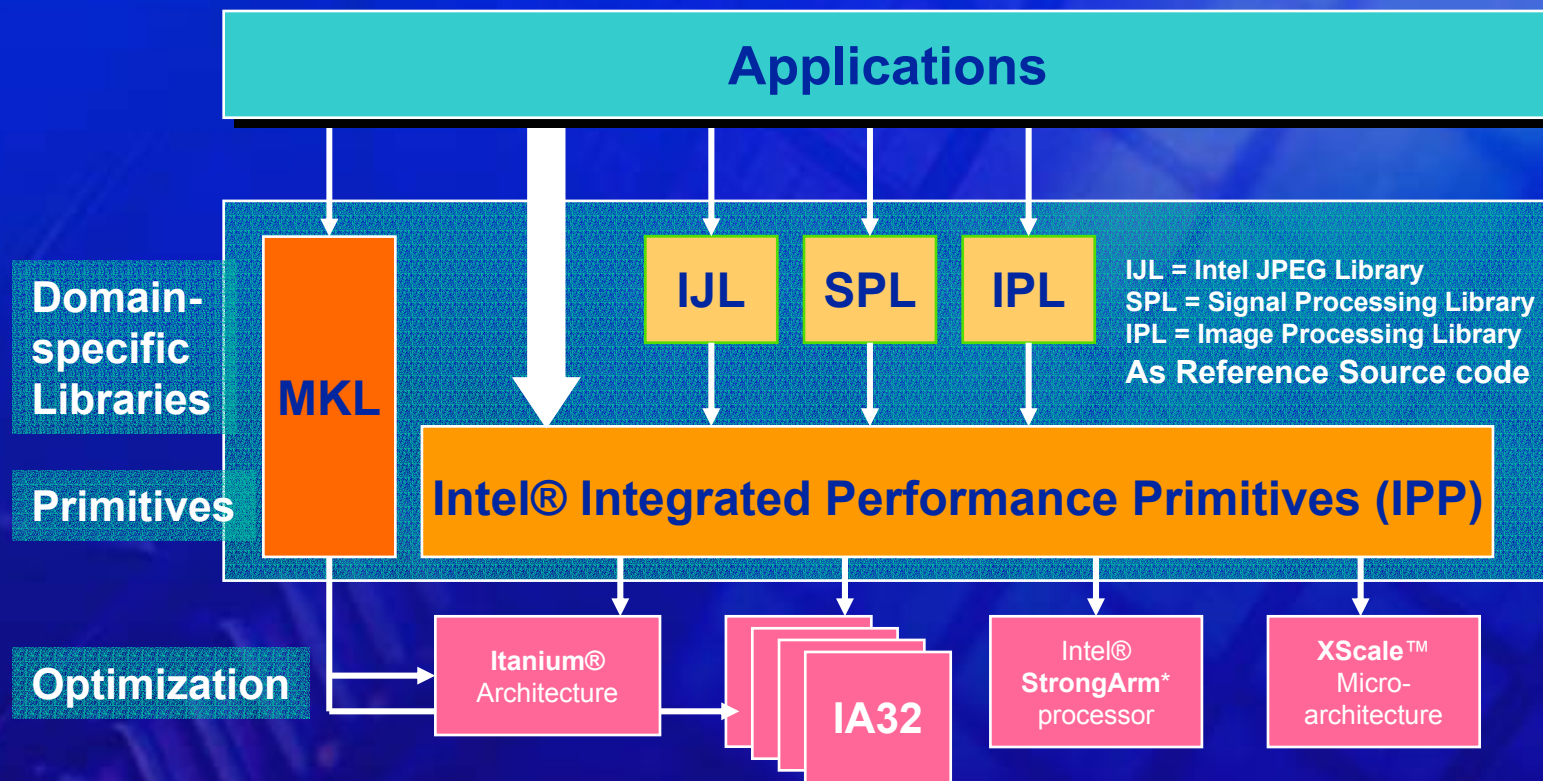
- **!DIR\$ PREFETCH *a*** : insert prefetch instructs.
- **!DIR\$ NOPREFETCH *a***
- **!DIR\$ PARALLEL** : disambiguates assumed data dependencies
- **!DIR\$ NOPARALLEL**

Use in conjunction with optimization report flags (-opt_report , -par_report)

Agenda

- **Optimizing with the Intel® Compilers**
- **Using Intel Performance Libraries**
- **Performance Tuning with Intel VTune™**
- **Other Optimization Opportunities**
- **Summary**

Intel® Performance Libraries



Expanded functionality & full platform integration

Intel® Performance Libraries

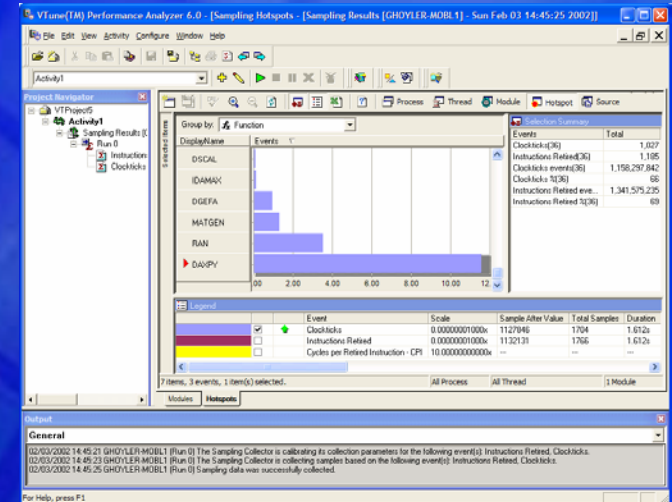
- Performance: raison d'être
- Resisted adding software only for functionality
- Finds ways to exploit every part of system: processor, memory, multiple processors, multiple nodes
- Makes high performance easy on lots of scientific, engineering and math codes
- Example: DGEMM (part of MKL) achieves
 - 95 % of theoretical peak performance on Itanium® 2
 - 75-77 % of theoretical peak performance on Xeon™

Agenda

- Optimizing with the Intel® Compilers
- Using Intel Performance Libraries
- Performance Tuning with Intel VTune™
- Other Optimization Opportunities
- Summary

VTune™ Overview

- Graphical performance tool
- Various experiments
 - Counter Monitoring (OS counters)
 - Call Graph (who calls who)
 - Sampling (standard hot spot profiling & HW counters)
- Visualization of results as graphs and/or tables
- GUI running under Windows* (32 or 64 bit)
- Remote collectors for Windows & Linux*
- Commercial product, list price: 699 US\$
- Current Version 6.1 adds Itanium® 2 support
<http://developer.intel.com/software/products/vtune/vtune61/index.htm>



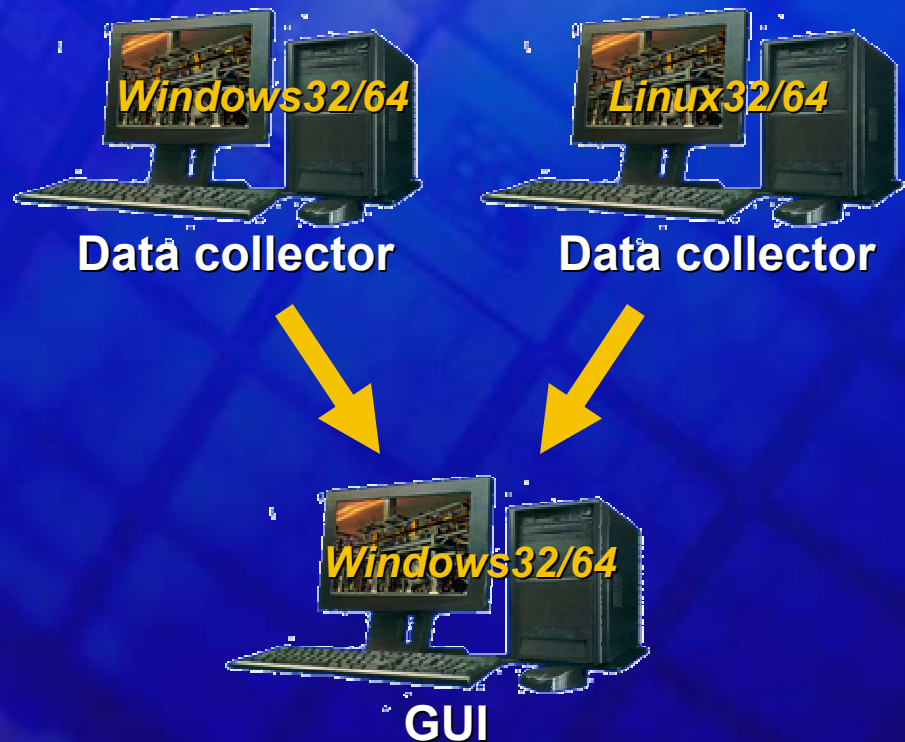
VTune™ Architecture

Local Analysis



Both data collector and
GUI on the same system

Remote Analysis



Linux* Data Collectors

- VTune™ includes binary kernel modules which are precompiled for various 2.4 and 2.5 Linux kernels
- Need to recompile kernel for Linux-64 (minor source code changes necessary)
- Install VTune™ 6.1 on a Windows client
- Install VTune Remote Collector package on the remote Linux system
- Copy or share the source code with client
- Open a new project on the Windows client and connect to the remote Linux system

The VTune™ GUI

VCR like controls

Data window

Project navigator

Graph window

The screenshot displays the VTune Performance Analyzer 6.0 interface. At the top, a menu bar includes File, Edit, View, Activity, Configure, Window, and Help. Below the menu is a toolbar with various icons, including a red circle highlighting the play, pause, and stop buttons. The main window is titled "VTune(TM) Performance Analyzer 6.0 - [Call Graph - [Call Graph Results - Sun Feb 03 13:12:50 2002]]".

On the left, the "Project Navigator" shows a tree view of the project structure, including "Activity1 [Counter Monitor]" and its sub-items like "Counter Monitor Results" and "Call Graph Results".

The central "Data window" contains a table with the following data:

| Thread (18) | Function (18) | Calls (18) | Self Time (18) | Total Time (18) | Callers (18) | Callees (18) |
|--------------------|---------------|------------|----------------|-----------------|--------------|--------------|
| Thread_8CC - Total | | 3174186 | 7,524,864 | | | |
| Thread_8CC | daxpy | 1060694 | 4,440,382 | 4,440,382 | 2 | 0 |
| Thread_8CC | ran | 2070000 | 1,576,102 | 1,576,102 | 1 | 0 |

Below the table is a "Graph window" showing a call graph. The graph starts with "CRTStartup" on the left, which branches into several system and application functions like "main_program", "daxpy", "dscal", and "idamax". Each function node is represented by a colored box with a plus sign, and arrows indicate the flow of calls between them.

At the bottom, the "Output" window shows "Instrumentation Results" with a log of events: "02/03/2002 13:12:50 Data collection finished...", "02/03/2002 13:12:50 Updating call graph database...", and "02/03/2002 13:12:51 Done.".

Output log



VTune™ Sampling

The screenshot displays the VTune Performance Analyzer interface. The main window shows a bar chart of sampling results grouped by function. The 'DAXPY' function is the most prominent, with a value of approximately 11.5. Other functions include RAN, MATGEN, DGEFA, IDAMAX, and DSCAL.

Selection Summary

| Events | Total |
|-----------------------------|---------------|
| Clockticks(36) | 1,027 |
| Instructions Retired(36) | 1,185 |
| Clockticks events(36) | 1,158,297,842 |
| Clockticks %(36) | 66 |
| Instructions Retired eve... | 1,341,575,235 |
| Instructions Retired %(36) | 69 |

Legend

| Event | Scale | Sample After Value | Total Samples | Duration |
|---|-----------------|--------------------|---------------|----------|
| <input checked="" type="checkbox"/> Clockticks | 0.00000001000x | 1127846 | 1704 | 1.612s |
| <input type="checkbox"/> Instructions Retired | 0.00000001000x | 1132131 | 1766 | 1.612s |
| <input type="checkbox"/> Cycles per Retired Instruction - CPI | 10.00000000000x | --- | --- | --- |

7 items, 3 events, 1 item(s) selected. All Process All Thread 1 Module

Output

General

```
02/03/2002 14:45:21 GHYOYLER-MOBL1 (Run 0) The Sampling Collector is calibrating its collection parameters for the following event(s): Instructions Retired, Clockticks.
02/03/2002 14:45:23 GHYOYLER-MOBL1 (Run 0) The Sampling Collector is collecting samples based on the following event(s): Instructions Retired, Clockticks.
02/03/2002 14:45:25 GHYOYLER-MOBL1 (Run 0) Sampling data was successfully collected.
```



Sampling Methods

■ Time-based sampling (TBS)

- Uses OS timer
- Delivers hotspot profile
- Available on almost every CPU, including mobile Intel® & AMD* processors

■ Event-based sampling (EBS)

- Uses processor counters
- Much more powerful than simple time-based sampling
- Delivers hotspot profile + CPU specific data (FP instructions, cache misses, bus activity, etc.)

Configure Sampling

General | Event Ratios | Events

Sampling Mechanism

Time-based sampling (TBS)

TBS based on: OS Services

Event-based sampling (EBS)

Calibrate Sample After value

Sampling Collection Options

Sampling interval: 1 millisecond(s)

Sampling buffer size: 2000 KB

Delay sampling: 1 second(s)

Track thread creation

Terminate application when Activity ends

Stop Collection

When application terminates (before duration completes)

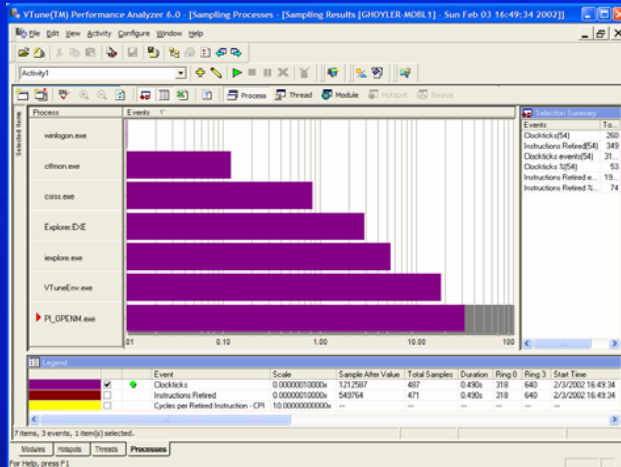
Maximum samples collected: 1

Hint:

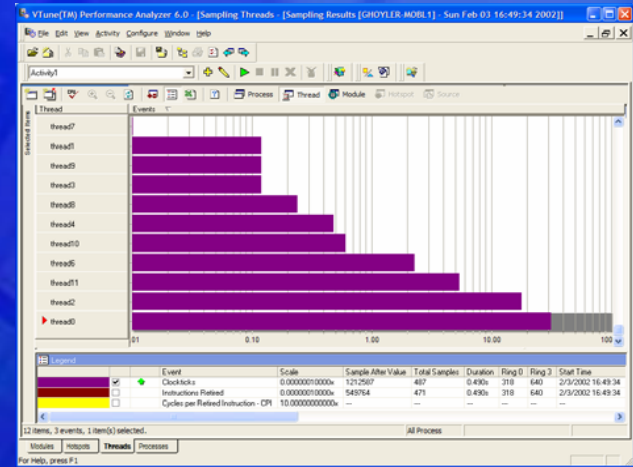
For EBS you can select events to monitor from the Events tab, and event ratios from the Event Ratios tab. If When application terminates (before duration completes) is checked, the VTune analyzer stops data collection when your application terminates even if the specified duration is not

OK Cancel Apply Help

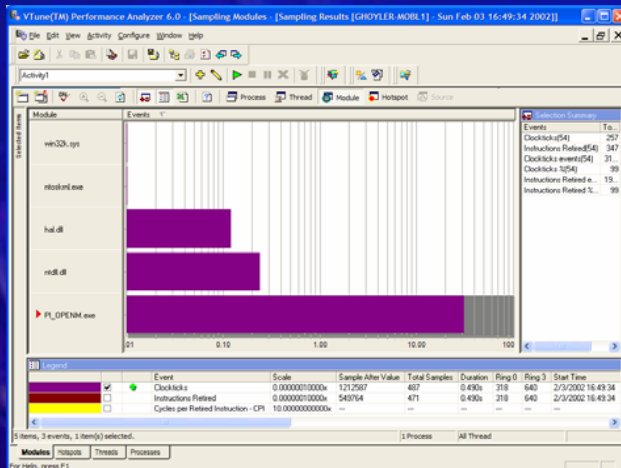
Multilevel Views ...



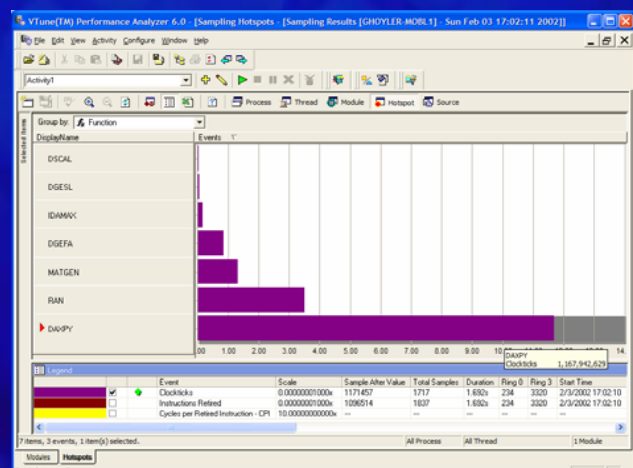
Process level



Thread level



Module level



Function level



... down to source & assembly!

VTune(TM) Performance Analyzer 6.0 - [Source View - [C:\Gernot\ftproot\linpack_tschiedel.f]]

```
Address Line Source
0x75C0 447 subroutine daxpy(n,da,dx,incx,dy,incy)
448 c
449 c constant times a vector plus a vector.
450 c jack dongarra, linpack, 3/11/78.
451 c
452 c double precision dx(1),dy(1),da
453 c integer i,incx,incy,ix,iy,n
454 c
0x762F 455 if(n.le.0)return
0x763C 456 if (da .eq. 0.0d0) return
0x7659 457 if(incx.eq.1.and.incy.eq.1)go to 20
458 c
459 c code for unequal increments or equal increments
460 c not equal to 1
461 c
0x7671 462 ix = 1
0x7679 463 iy = 1
0x767C 464 if(incx.lt.0)ix = (-n+1)*incx + 1
0x7697 465 if(incy.lt.0)iy = (-n+1)*incy + 1
0x76B2 466 do 10 i = 1,n
0x76CC 467 dy(iy) = dy(iy) + da*dx(ix)
```

| Address | Size | Function | Class | Clockticks (76) | Instructions Retired (76) | Cycles per Retir... |
|---------|--------|----------|-------|-----------------|---------------------------|---------------------|
| 0x06BDD | 0x04CD | DGEFA | | 37 | 22 | 1.797 |
| 0x0709D | 0x053D | DGESL | | 5 | 5 | 1.068 |
| 0x075C0 | 0x01B0 | DAXPY | | 997 | 1,222 | 0.872 |
| 0x07770 | 0x01A0 | DDOT | | 0 | 0 | |
| 0x07910 | 0x0130 | DSCAL | | 2 | 10 | 0.214 |
| 0x07740 | 0x0160 | TDAXPY | | 14 | 17 | 0.820 |

VTune(TM) Performance Analyzer 6.0 - [Source View - [C:\Gernot\ftproot\linpack_tschiedel.f]]

```
Address Line Source
0x7718 476 20 continue
0x7719 476 do 30 i = 1,n
DAXPY+158: mov eax, DWORD PTR [ebx+08h]
0x771B 476 mov eax, DWORD PTR [eax]
0x771D 476 mov DWORD PTR [ebp-8], eax
0x7720 476 mov DWORD PTR [ebp-12], 01h
0x7727 476 mov eax, DWORD PTR [ebp-8]
0x772A 476 test eax, eax
0x772C 476 jle DAXPY+1a2
0x772E 477 dy(i) = dy(i) + da*dx(i)
0x772E 477 DAXPY+16e: mov eax, DWORD PTR [ebp-12]
0x7731 477 mov edx, DWORD PTR [ebp-56]
0x7734 477 mov ecx, DWORD PTR [ebx+0ch]
0x7737 477 fld QWORD PTR [ecx]
0x7739 477 mov ecx, DWORD PTR [ebp-12]
0x773C 477 mov esi, DWORD PTR [ebp-96]
0x773F 477 fld QWORD PTR [esi+ecx*8-8]
0x7743 477 fmulp st(1), st(0)
0x7745 477 fld QWORD PTR [edx+ecx*8-8]
0x7749 477 faddp st(1), st(0)
0x774B 477 mov eax, DWORD PTR [ebp-12]
```

| Address | Size | Function | Class | Clockticks (76) | Instructions Retired (76) | Cycles per Retir... |
|---------|--------|----------|-------|-----------------|---------------------------|---------------------|
| 0x06BDD | 0x04CD | DGEFA | | 37 | 22 | 1.797 |
| 0x068D0 | 0x04C0 | DGEFA | | 95 | 95 | 0.821 |
| 0x07090 | 0x0530 | DGESL | | 5 | 5 | 1.068 |
| 0x075C0 | 0x01B0 | DAXPY | | 997 | 1,222 | 0.872 |
| 0x07770 | 0x01A0 | DDOT | | 0 | 0 | |
| 0x07910 | 0x0130 | DSCAL | | 2 | 10 | 0.214 |
| 0x07740 | 0x0160 | TDAXPY | | 14 | 17 | 0.820 |

Source code level



Assembly level

Source code level analysis requires use of debugging flag /Zi,-g in order to include symbolic source line info in the binary!

Intel® Tuning Assistant

The screenshot shows the VTune(TM) Performance Analyzer 6.0 interface. The main window displays source code for a file named [C:\Gernot\ftprot\linpack_tschiedel.f]. The code includes initialization of variables and a loop structure. A yellow arrow points to line 214, which is highlighted in the source view. The Intel Tuning Assistant window is open, displaying tuning advice for the selected code. It indicates that 100% of lines 214 to 214 in linpack_tschiedel.f are candidates for SIMD optimization. The advice suggests using SIMD technology for vector operations.

| Address | Size | Function | Class | Sampling Results [...] |
|----------------------------|---------|----------|-------|------------------------|
| ----- Selected Range ----- | | | | |
| 0x0508D | 0x0188D | program | main | 45 |
| 0x0690D | 0x02DD | MATGEN | | 113 |
| 0x06BD0 | 0x04CD | DGEFA | | 73 |
| 0x07090 | 0x053D | DGESL | | 5 |
| 0x075C0 | 0x01B0 | DAXPY | | 997 |
| 0x07770 | 0x0120 | DPOT | | 0 |

Line 214: Vector operations using SIMD technology

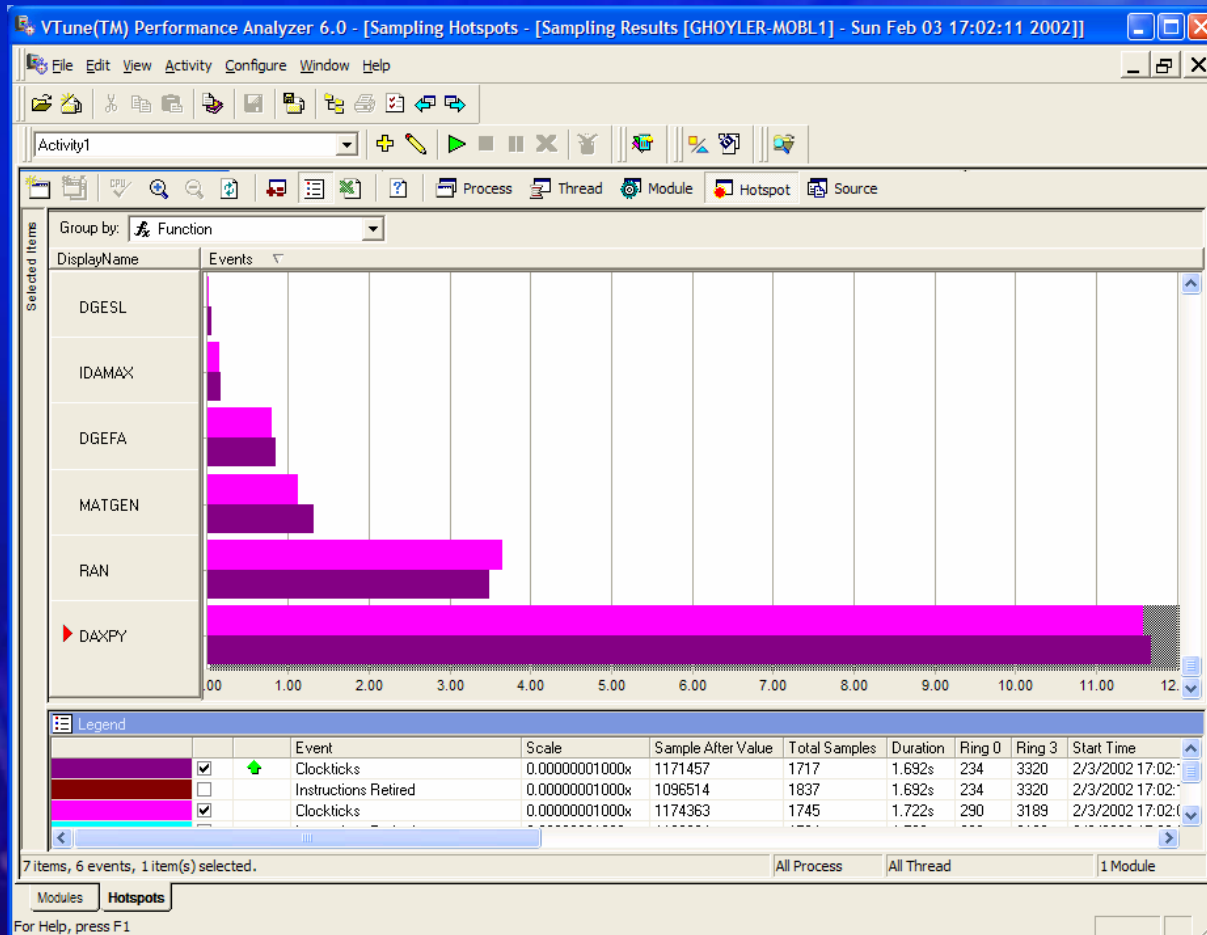
The assignment operation on line 214 is a candidate for a performance boost using SIMD technology. The following SIMD options are available:

- SIMD Class Library:** Using C++, declare the arrays of this statement as objects of the `F64vec2` class defined in Intel's SIMD Class Library, and recompile your program using the Intel® C/C++ Compiler. This and other statements like it will be compiled using SIMD code ("vectorized").
- Vectorizer:** Use the Intel® Fortran Compiler vectorizer to automatically generate highly optimized SIMD code. The statement on line 214 and others like it will be vectorized.
- Performance Libraries:** Replace your code with calls to functions in the Intel® Performance Library Suite. Some of its functions that are possibly useful in your application appear below (use the F1 key with this advice for a more informative summary):
`ippsAbs_64f_I()`
`ippsMax_64f()`
- Intrinsic Functions:** Rewrite this loop as a C-language subroutine, and use the SIMD intrinsic functions recognized by the Intel® C/C++ Compiler. C-style pseudocode for the intrinsics suggested for this statement (click on any function name for a brief description):

```
*((__m128d*) &norma(..)) = _mm_max_pd(  
    _mm_max_pd(  
        *((__m128d*) &a), _mm_sub_pd(  
            (__m128d)_mm_set1_pd(0.0000000000000000e+000),  
            tmp0 = *((__m128d*) &a))),  
    _mm_set1_pd((double)norma));
```

Processor specific expert system –
based on source code analysis!

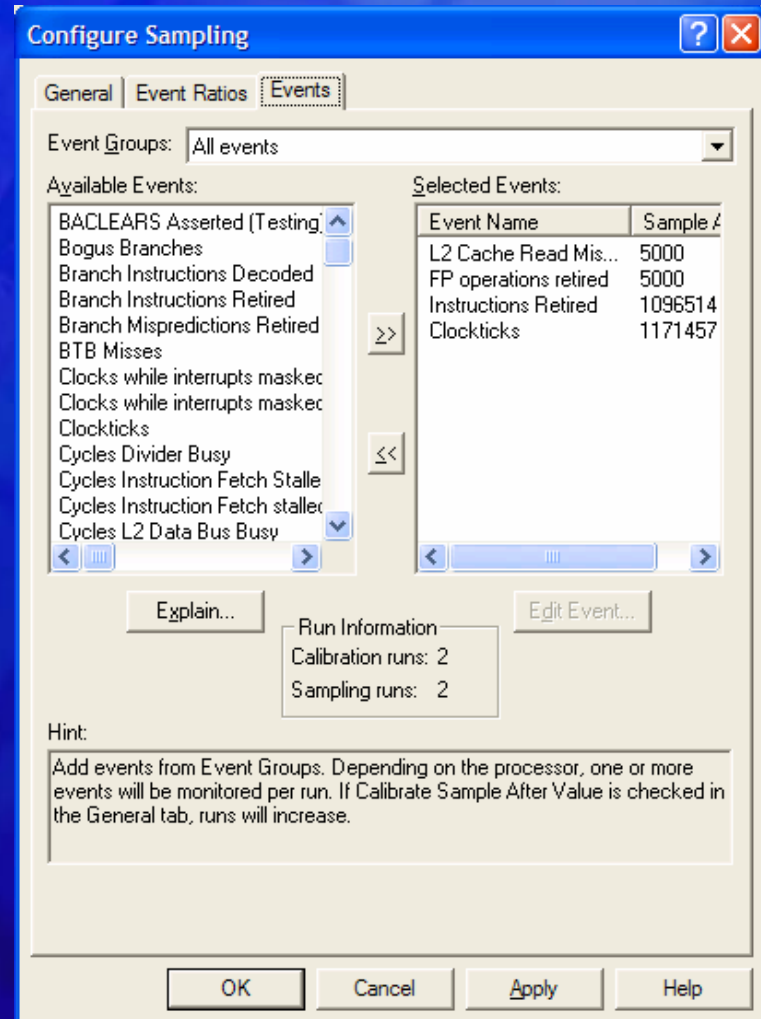
Multiple Sampling Activities



Allows direct comparison of multiple profiles

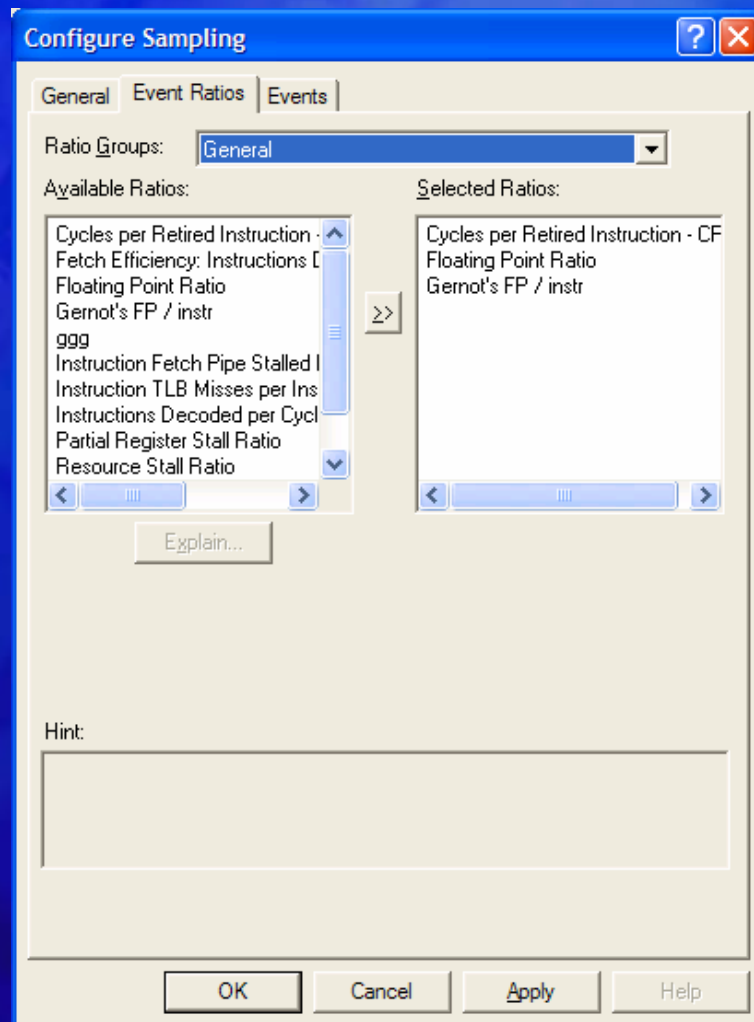
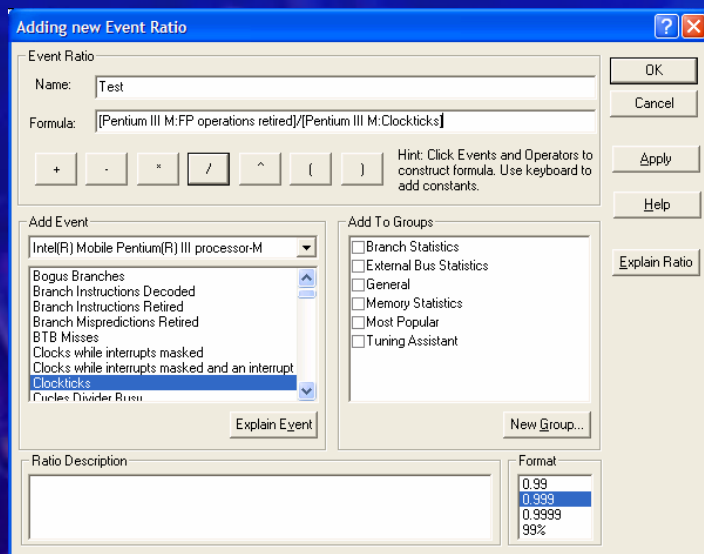
CPU Specific HW Counters

- Event based sampling allows programming of CPU specific counters
- Useful to look at cache, memory and TLB activities, FP throughput as well as other CPU characteristics
- Can require multiple runs to collect all data!



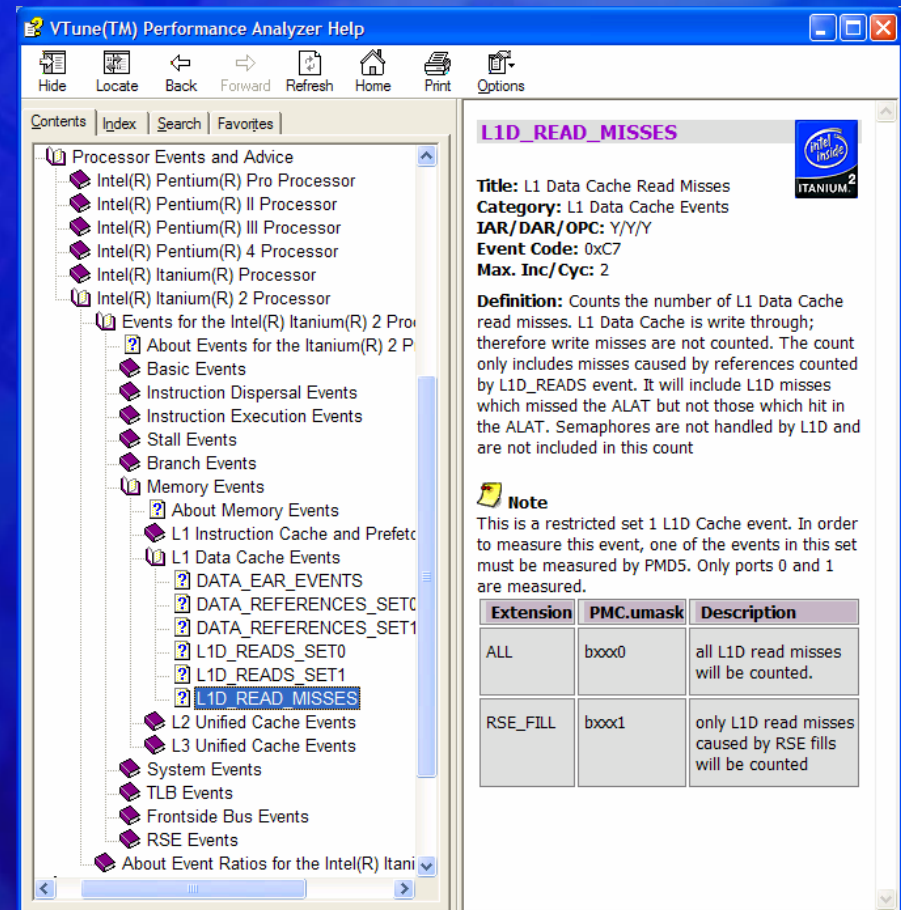
Counter/Event Ratios

- Predefined standard ratios available
 - e.g. CPI (clocks per instr.)
- User defined ratios can be easily added
 - even complex formulae possible



Which Events and Event Ratios should be Monitored?

- Consult the VTune™ online manual for a list of primary events and event ratios to start with, afterwards extend and refine as needed
- Itanium®: start with Basic, Instruction, Execution, TLB, Memory and Branch Events and use Stall Events for cycle accounting
- NetBurst™: start with „Primary Performance Tuning Events“ and ratios, note that an impact of > 2 might already be critical



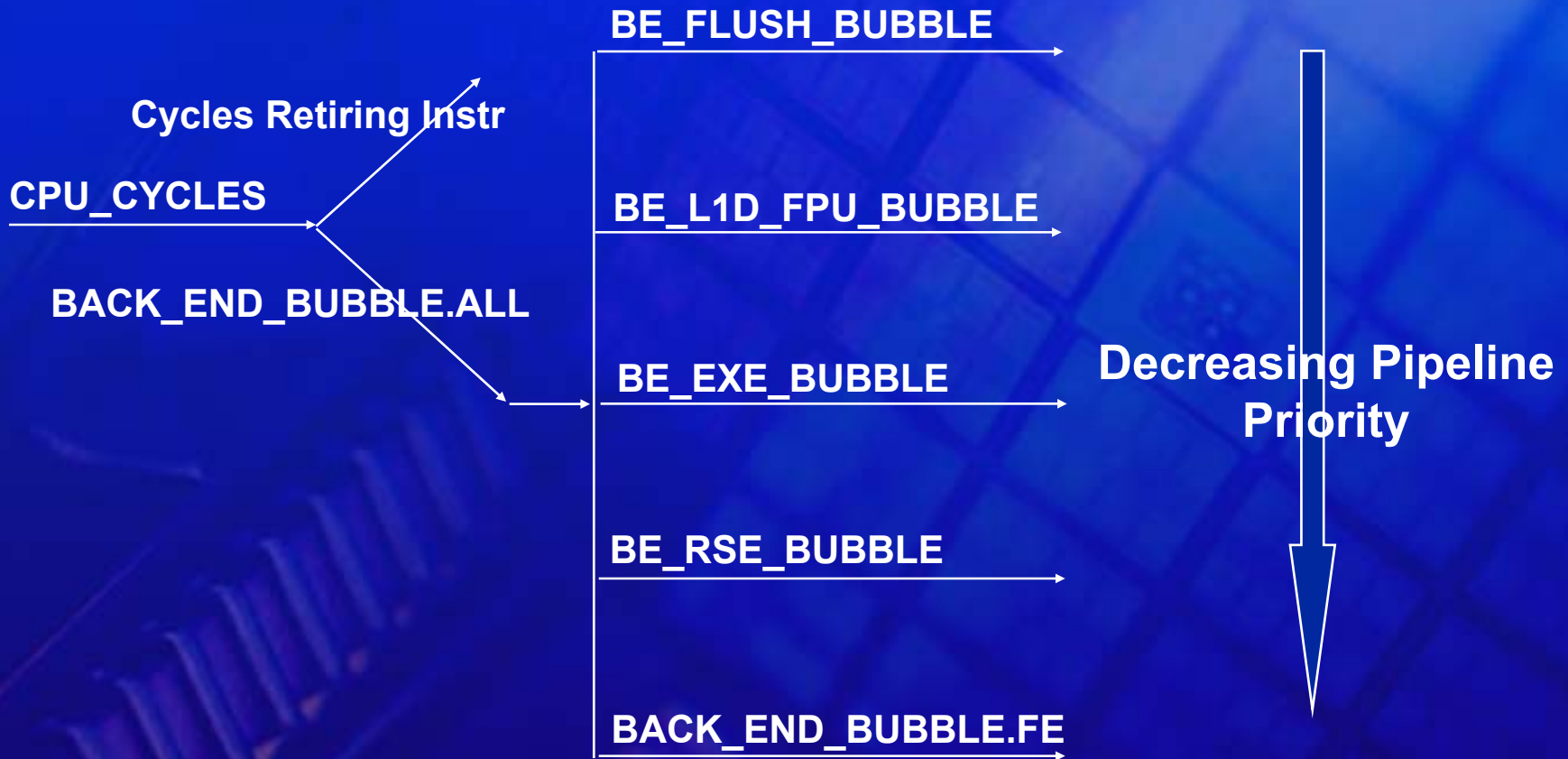
The screenshot shows the VTune(TM) Performance Analyzer Help window. The left pane displays a tree view of processor events and advice, with 'L1D_READ_MISSES' selected under the 'Intel(R) Itanium(R) 2 Processor' > 'L1 Data Cache Events' > 'L1D_READS_SET1' path. The right pane provides detailed information for the 'L1D_READ_MISSES' event, including its title, category, IAR/DAR/OPC, event code, and maximum increment per cycle. A definition explains that it counts L1 Data Cache read misses, excluding write misses. A note specifies that this is a restricted set of L1D Cache events measured by PMD5. A table at the bottom lists extensions and their descriptions.

| Extension | PMC.umask | Description |
|-----------|-----------|--|
| ALL | bxxx0 | all L1D read misses will be counted. |
| RSE_FILL | bxxx1 | only L1D read misses caused by RSE fills will be counted |

Cycle Accounting on Itanium®

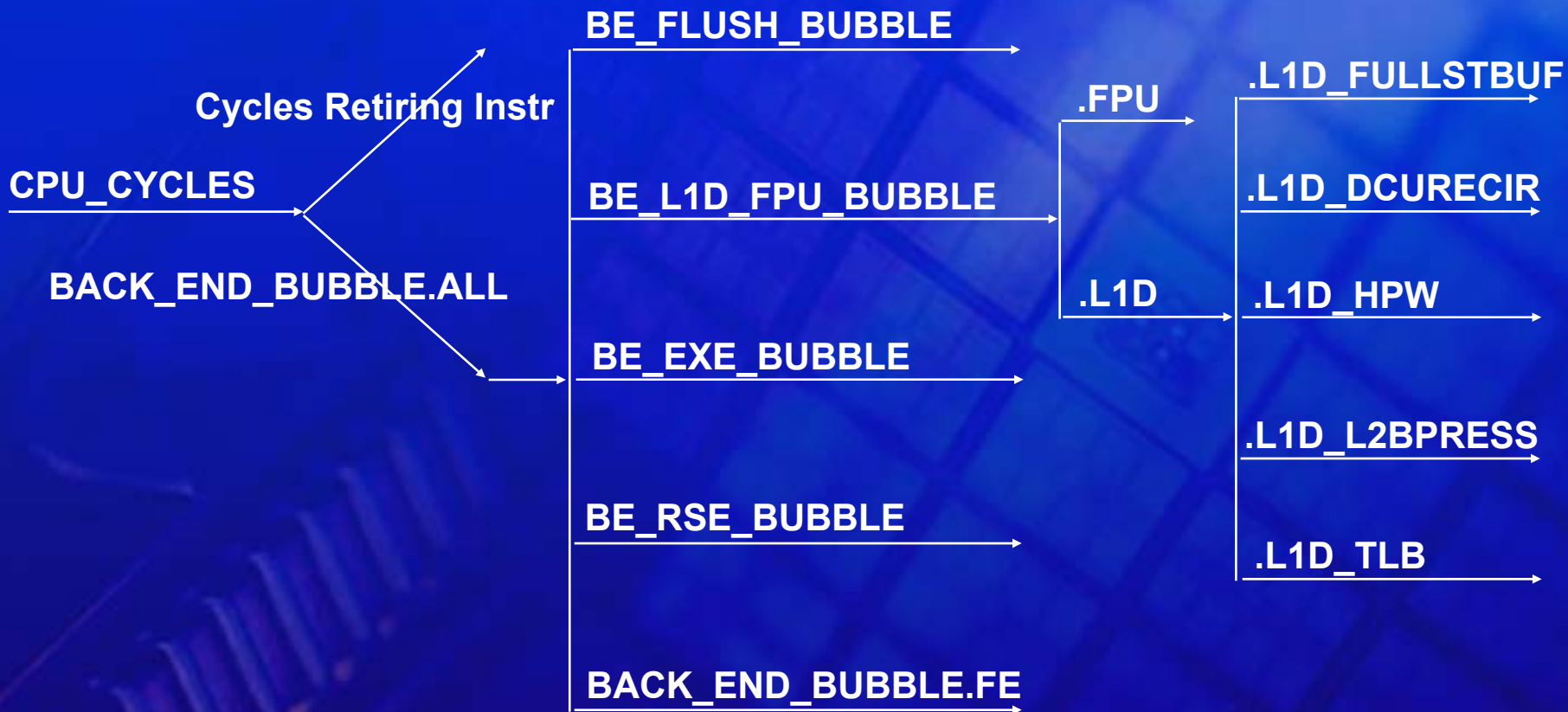
- In general pipeline is stalling and not issuing instructions to functional units on every cycle; Cycles are wasted waiting for resources like e.g.
 - data from memory or cache
 - instructions to be processed
 - register availability due to excessive RSE activity
- Use sub-set of PMU counters to find main stall source
- Hierarchical tree structure enables systematic performance methodology

Tree Structure of Stall Cycles



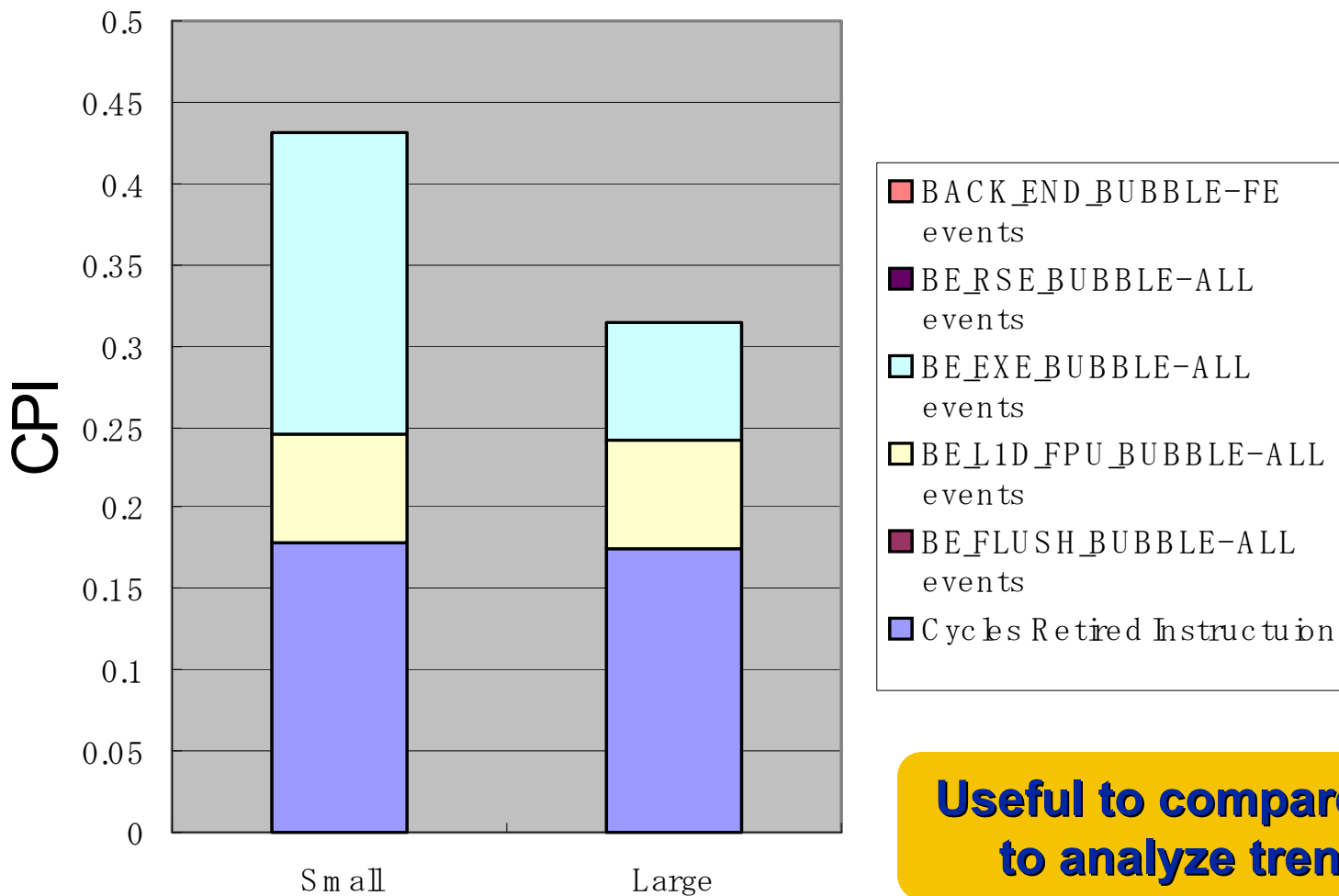
Cycle accounting sum rule starts the decomposition

E.g.: Components of BE_L1D_FPU_BUBBLE



Sub-events of BE_L1D_FPU_BUBBLE have multiple levels

Cycle Accounting Chart



Useful to compare and to analyze trends

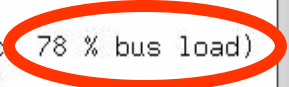
Agenda

- **Optimizing with the Intel® Compilers**
- **Using Intel Performance Libraries**
- **Performance Tuning with Intel VTune™**
- **Other Optimization Opportunities**
- **Summary**

Other Optimization Opportunities

- Use of command line tools such as `pfmon`, `hpcmon`, `vt1` etc.

```
Konsole - gernot@localhost:~ - Konsole
=====
HPCmon Ver. 1.1 - (c) 2002 Gernot Hoyler, Intel GmbH
[pfmon Ver. 1.1 - (c) 2001-2002 Hewlett-Packard Company]
=====
CPU speed          :      897512179 Hz
Elapsed time       :      8036620308 cycles (8.95 sec)
Monitored clockticks :      8035906015 cycles (8.95 CPU sec)
Instructions retired :      6657720795 events (743.52 Mev/sec, 0.83 IPC)
Nops retired       :      415678786 events (46.42 Mev/sec, 0.05 NPC)
Off predicated instr. :      304378 events (0.03 Mev/sec, 0.00 NPC)
FP instr. retired  :      1608094008 events (179.59 Mev/sec or MFLOPS)
FP results FTZ'ed  :      0 events (0.00 Mev/sec)
2nd lev. cache refs. :      3071626791 events (343.03 Mev/sec)
2nd lev. cache misses :      252098578 events (28.15 Mev/sec, 91 % hit rate)
3rd lev. cache refs. :      353178614 events (39.44 Mev/sec)
3rd lev. cache misses :      252082821 events (28.15 Mev/sec, 28 % hit rate)
L2 DTLB misses     :      495705 events (0.06 Mev/sec)
L2 ITLB misses      :      62 events (0.00 Mev/sec)
Bus data cycles     :      1412740182 events (157.77 Mev/sec)
Bus snoop stall cycles :      223012470 events (24.91 Mev/sec)
Misaligned loads    :      0 events (0.00 Mev/sec)
Stores to shared line :      9366 events (0.00 Mev/sec)
Branches retired    :      302463901 events (33.78 Mev/sec)
Mispredicted branches :      82394 events (0.00 Mev/sec, 99 % hit rate)
% █
```



Can help to identify obvious bottlenecks at a first glance

Other Optimization Opportunities

- **Use of hand coded assembly**
 - Quite tricky to apply to larger code portions
 - Consult IA-64 guides for instruction mix and latencies
 - GCC inline assembly syntax not supported, use asm intrinsics instead
- **Give different OS & Compilers a chance**
- **Submit the code to specialists at your OEM or at Intel® (e.g. Solution Centers)**

Further information & tricks can be found in the Itanium® 2 Processor Reference Manual for Software and Optimization

<http://developer.intel.com/design/itanium/manuals.htm>

Summary

- The Intel® Itanium® Architecture is still quite a new march with a huge performance potential
- Always use the latest versions of Intel® tools
- Apply standard compiler optimization flags plus IPO & PGO
- Use performance libraries wherever possible
- For advanced optimizations take the Intel® VTune™ Performance Analyzer and identify the bottlenecks
- Use hand coded assembly only as a last resort

Thank You !

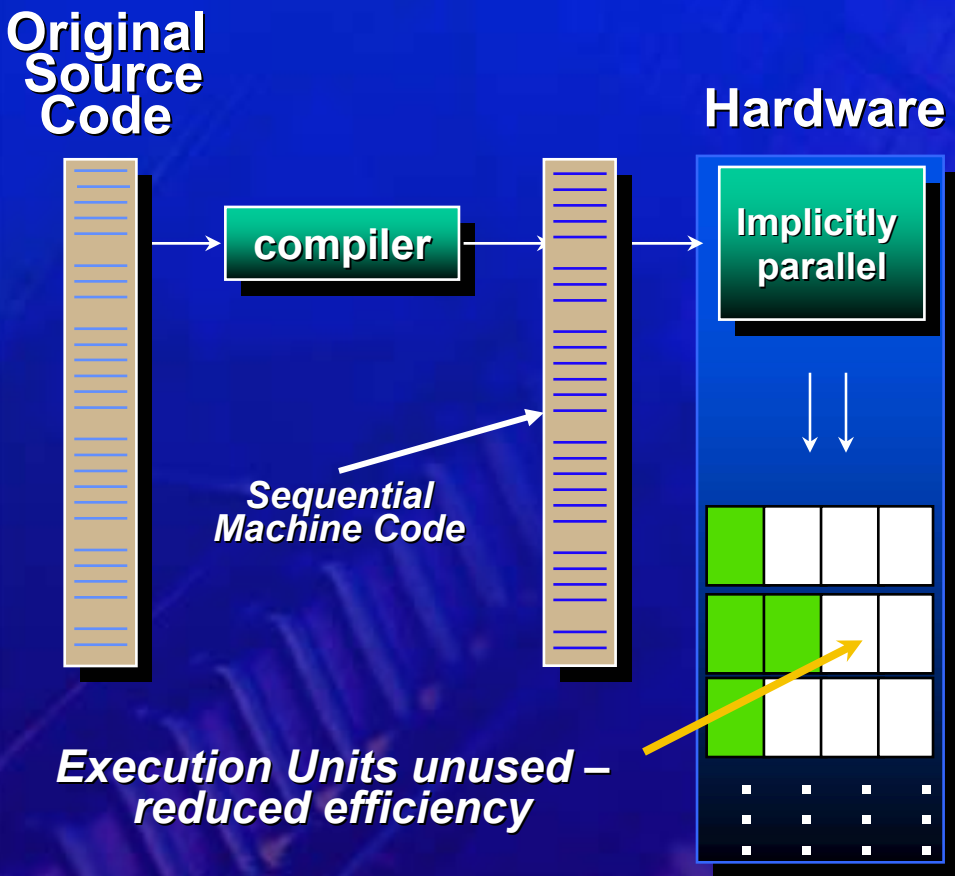


www.intel.com/go/hpc

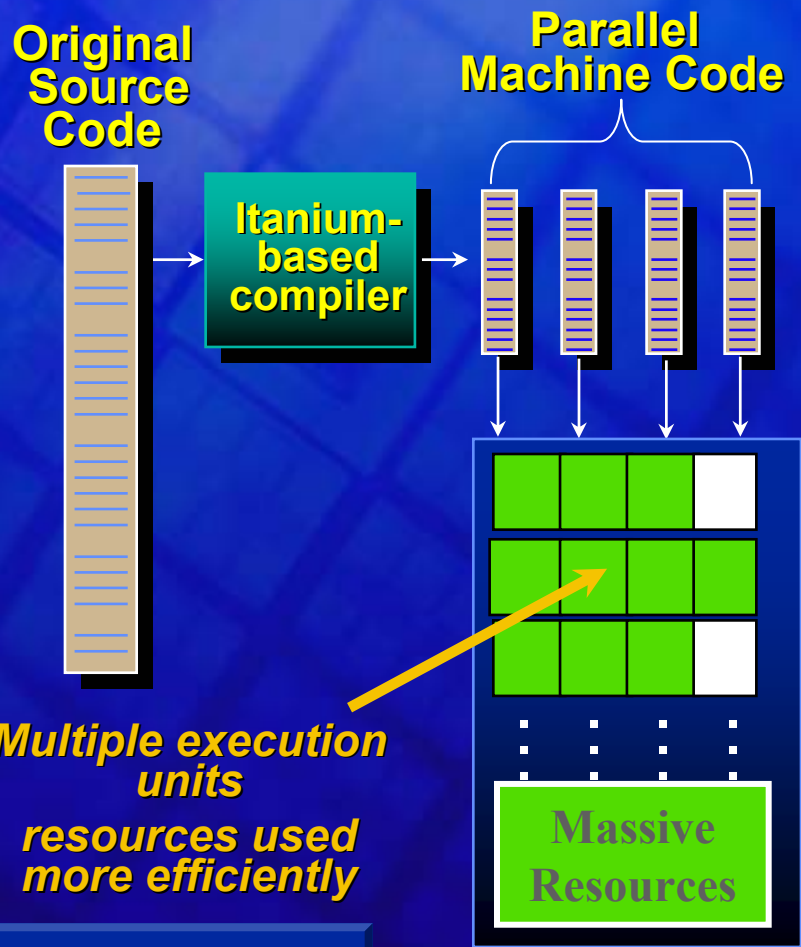
Itanium™ Architecture

Explicit Parallelism

Traditional



Itanium™ Architecture



Performance through Parallelism

Block Diagram (Itanium 2)

