Keys to Successful IA-64 Software Development

Mike Martell - Staff Software Engineer
David Mackay - Applications Analysis Leader
Software Performance Lab
Intel Corporation

February 15-17, 2000
Agenda: Key IA-64 Software Issues

- Do I Need IA-64?
- Coding for Portability
- Coding for Performance and Maintainability
- Mixing IA-32 and IA-64
- Planning for an Optimal Product
Need IA-64 Yet?

- **Want More Performance?**
  - IA-64 removes performance bottlenecks
  - IA-64 is a parallel/scalable architecture

- **Need More Than 2 or 3 Gigabytes?**
  - Technology exponential (feeds on itself)
  - Data space growing at ~.7 bits/yr
  - Hard to anticipate market 5 years out

Start IA-64 Development Now
Coding for Portability

- Abstract the OS API
  - Extra layer to encapsulate OS calls
- Use a Unified Data Model
  - Polymorphic types to support 32 and 64 bit processors
- Minimize Processor Dependencies
  - Minimize Assembly (use intrinsics)

OS and Processor Independent Coding Is Easy
Old Hat: Abstract the OS’ API

- If you support Unix and NT, you probably already do this
- C++ and OOP have taught us how
- Some APIs already standard: OpenGL
The New Challenge: Abstracting the Data Model

- Not harder, but perhaps new
- Single source for 32 and 64 bit processors very desirable:
  - Termed Unified Data Model (UDM)
- OS vendors have done it, now it’s your turn
Terms to Know

- **Code Clean**
  - Revising source code to be compilable in *both* 64-bit and 32-bit environments.

- **Polymorphic**
  - One data item having a different type to different users/viewers. A key source of code-clean problems.

- **“Data Bloat”**
  - Increase in size of data in 64-bit applications.

- **Cardinality**
  - The range of numbers a data item can count.

Topics that could previously be ignored in the 32-bit world
Windows & UNIX Use Different Data Models

- OS vendors tried hard to minimize changes
  Windows chose P64, Unix chose LP64

<table>
<thead>
<tr>
<th>OS</th>
<th>Data Model</th>
<th>int</th>
<th>long</th>
<th>pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIX/64</td>
<td>I32,LP64</td>
<td>32</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Win64</td>
<td>IL32,P64</td>
<td>32</td>
<td>32</td>
<td>64</td>
</tr>
</tbody>
</table>

- If using `long`, you must change it to maintain platform independence!
  - Suggestion: Replace with new type defined in a `<compatible.h>` file.

See IDF Tracks: Porting to Monterey*/Linux* on IA-64

Third party marks are properties of their owners
Use ANSI Types for Platform Independence

- ANSI* types with universal meaning on Unix, Win32, Win64:
  - New polymorphic integer: intptr_t
  - Old polymorphic cardinal type: size_t
  - Good old 32 bit integers: int

- Try to avoid OS dependent types:
  - INT_PTR, SIZE_T, (Windows only types)

Code Cleaning Steps

- **Switch to the Unified Data Model**
  - use polymorphic types

- **Perform required API tweaks**
  - typically, small number of semantic changes (can be automated)

- **Compile with code clean switch to catch problems**
  - pointer truncations, etc. Not easily automated

---

Single Source Code for IA-64 / IA-32

See IDF Sessions: Porting to Win64*/Monterey*/Linux* on IA-64
Use a Portability Header File

#include <portatyp.h>

- Used by all your source files
- Write your source using these standard-inspired types

/* portatyp.h */
#if defined(_WIN32)
    ... // stuff related to Win32
#else ifdef(_WIN64)
    ... // Win32 without Win64 (regular Win32)
#endif /* _WIN64 ? */
#else /* some other OS */
#error Unhandled OS;
#endif

Enhance Portability & Readability
Processor Dependencies

- Use significant architectural features for performance

... but

- Use pragmas, macros, and intrinsics to minimize platform dependencies

- Rely on the compiler as much as possible
Processor Dependencies (continued)

- **SIMD instructions**
  - Use intrinsics, encapsulate in a class

- **Rely on the compiler for:**
  - Loop unrolling
  - Vectorizing
  - Proper insertion of prefetch intrinsics
Coding for Performance and Maintainability

- Modular programming (using DLLs, classes, and small functions) promotes maintainability!

  ... but it also
  - adds overhead and
  - limits what a compiler can do

- IA-64 promotes modular programming without the performance loss!
Special Support for Modular Programming

**Hardware:**
- Register Stack Engine (minimize stack frame save/restore)
- lots of registers (schedule more operations)

**Software:**
- Interprocedural Compiler Optimizations (IPO):
  - let the compiler see more code
- Profile Guided Compiler Optimizations (PGO):
  - drive IPO decisions with real performance data
- High Level Optimizations (HLO):
  - loop unrolling, vectorization, prefetch, blocking
New Compiler Usage Model

- IA-64 offers more opportunity for compiler optimizations

<table>
<thead>
<tr>
<th></th>
<th>Debug</th>
<th>Release</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA-32 Model</td>
<td>Od</td>
<td>Ox</td>
</tr>
<tr>
<td>IA-64 Model</td>
<td>Od</td>
<td>Ox</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Oa</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HLO</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PGO</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IPO</td>
</tr>
</tbody>
</table>
Software Pipelining Is A BIG Performance Deal

- Dynamic memory (pointers) allows SW to support wide range of HW configurations!
  - but it also
    - limits the compiler’s ability to software pipeline

- Disambiguate pointers with:
  - restrict keyword (loop level control, now in ANSI C)
  - #pragma optimize(“a”) (function level control)
  - No aliasing flag (Oa, file level control)

Help the Compiler Pipeline your Loops
Using `restrict` To Allow Software Pipelining

```c
void VSquare
{
    double* restrict pX, double* restrict pY, int n

    for ( int i=0; i<n; i++ ) pX[ i ] = pY[ i ] * pY[ i ];
}
```

Use `restrict` when you know `pX` and `pY` point to distinct objects (i.e., do not overlap).
Software Pipelining (SWP) some examples

- SSL Kernel Loop (3 cycle, 9 stage)
  - 26 cycle inner loop
  - 3 cycle loop
  - Compiler Can Do This For You

- MemCopy (1 cycle, 3 stage)
  - 3 cycle loop
  - Compiler Can Do This For You
Software Pipelining (SWP) some more examples

- **Vector Scale (1 cycle, 18 stage)**
  - Without SWP: 18 cycle inner loop
  - With SWP: 1

- **Dot Product (1 cycle, 10 stage)**
  - Without SWP: 10 cycle inner loop
  - With SWP: 1

**Compiler Can Do This For You**
Interprocedural Optimization – IPO

Compiler looks at entire program:

- Inlining, partial inlining
  - HW promotes greater benefits from
- Interprocedural constant propagation
- Circumvent legacy calling convention
- Enables data layout
  - better data alignment and d-cache hits
- Dead call, partial dead call removal
Profile Guided Optimization – PGO

Drive the compilation process with real performance data:

- Code layout for i-cache
- Static and dynamic profile information
- Improve usage of Resister Stack Engine
- Tune usage of predication and speculation!
- Improve branch prediction
- Inline frequently used functions only
Using IPO and PGO

- Available now on IA-32
- Greater impact is available on IA-64
- Very synergistic optimizations
  - Plan to use PGO and IPO together
- Plan to utilize in build and QA cycle

See IDF Session: Optimizing IA-64 Software Performance
PGO/IPO Speedups

PGO/IPO Benefits
Much Greater on IA64
Mixing IA-64 and IA-32

- Use standard IPC between IA-64 and IA-32 processes (further details in backup slide)
- May be useful for dealing with 3rd party dependencies
- **However ...**
  - no support for in-process mixing
  - should only be used when performance isn’t critical
Mgt Planning for IA-64

- Start UDM practices Now!
  - Don’t wait for IA-64 release

- QA stress tests much harder/longer:
  - Fatter machines, bigger datasets, longer runs
  - New level of functionality testing (never had a 5 GB assembly to test)
  - Plan to allow PGO – dual build cycle

- Delivery plans
  - New binary (new CD? Or coexist with IA-32?)
For More Information

Intel IA-64 information:

• Primary portal: http://developer.intel.com/design/IA-64
• “IA-64 Application Developer’s Guide,” Literature #245188-00
• IA-64 Computer Based Tutorials
  http://developer.intel.com/vtune/cbts/cbts.htm

Other IA-64 information:

• IDF Tracks: Porting to Win64*/Monterey*/Linux* on IA-64
• Unified Data Model Rationale:
  http://www.opengroup.org/public/tech/aspen/1p64_wp.htm
What is an Intel® Application Solution Center (ASC)?

ASC is a premier technical center that offers software developers consultation services, tools, and support for server and workstation applications performance-tuning and porting assistance on current and future Intel® Architecture-based systems.
ASC Strategic Importance for IA-64

- ASCs provide critical *technical support* to customers during transition to IA-64
- ASCs provide *early access* to SDVs for customers to execute, debug IA-64 software
- ASCs offer software developers a *competitive edge* with leading applications built to take advantage of all IA-64 robust features
Call to Action

- **Start** using a Unified Data Model
- **Make** a plan to address key issues
- **Seek assistance** through an IA-64 Application Solution Center
  
Backup
COM or RPC for Mixed Binary Communication on Win64

- Use COM or RPC (which COM is built on) for interprocess communications
- Move IA-32 DLLs to another process
- Easy to do. Can be automated.
- Use only when RPC overhead is well amortized

Ref: Search for “64-bit and 32-bit Processes” in MSDN/Platform SDK/Win64 Programming Preview