Performance tools for understanding the behavior of Multicore programs

Gadi Haber
haber@il.ibm.com
Cell BE Architecture
Microprocessor Trends

- Single Thread performance power limited
- Multi-core throughput performance extended
- Hybrid extends performance and efficiency
Cell Broadband Engine

- Heterogeneous Multiprocessor
  - Power processor
  - Synergistic Processing Elements

- Power Processor Element (PPE)
  - general purpose
  - running full-fledged OSs
  - 2 levels of globally coherent cache

- Synergistic Proc. Element (SPE)
  - SPU optimized for computation density
  - 128 bit wide SIMD
  - Fast local memory
  - Globally coherent DMA
Cell Broadband Engine™: A Heterogeneous Multi-Core Architecture
IBM PowerXCell™ 8i processor


**PowerXCell 8i processor**
- 65 nm
- 9 cores, 10 threads
- 230.4 GFlops peak (SP) at 3.2GHz
- 108.8 GFlops peak (DP) at 3.2GHz
- Up to 25 GB/s memory bandwidth
- Up to 75 GB/s I/O bandwidth
- 92 Watts @ 3.2GHz
- Top frequency >4GHz (observed in lab)
SPE highlights

- **RISC like organization**
  - 32 bit fixed instructions

- **Unified register file**
  - 128 entry x 128 bit

- **256KB Local Store (LS)**
  - Combined I & D
  - No translation/protection within SPU

- **DMA Engine (MFC)**
  - Move data between LS and main memory

- **Channels:**
  - Mailbox, Signal-notification
  - Event-Management

- **VMX-like SIMD dataflow**
  - Single Instruction Multiple Data” - Data-level parallelism
  - 128 bit data path - multiple data elements per register
  - 16 bytes / 8 halfwords / 4 fullwords / 2 doublewords
  - Graphics SP-Float / IEEE DP-Float

**14.5mm² (90nm SOI)**
Programming in Multicore
Programming for Multicore platforms

- Emerging Multicore platforms introduced a challenge to the development cycle
- The sole purpose of many multicore platforms is to provide better performance
- In these platforms Performance Debugging and Performance Analysis has become essential to the development cycle
- Correctness on a single core is no longer sufficient
Performance Debugging & Optimization in Multicore

- Performance debugging is different from traditional correctness debugging

- One cannot use a regular debugger to intercept
  - Delays resulting from synchronization effects between running threads
  - Delays resulting from improper load balancing on the CPUs
  - Delays resulting from Memory affinity
  - Delays resulting from mutual cache invalidations of global shared data
  - Delays resulting from pipeline stalls of each core
  - Potential for parallelization and simultaneous execution
Performance Analysis challenges on Multicores

Scalability issues:
- Insufficient parallelism
- Stalls waiting for communication
- Hot locks
- TLB/cache thrashing

Other issues:
- Taller application stack
- App/framework configuration
- Working with accelerators

Single thread performance:
- Branch misprediction
- Cache misses
- ...
Profile information – the basis for performance analysis

- Performance debugging and optimizations for multicore is based on gathered **profile information** from a representative program execution

- Profile information is then used by:
  - **Compilers** to optimize, vectorize, parallelize and tune programs
    - Examples of profile-based optimizers:
      - FDO option in GCC
      - Intel Proton FDO
      - IBM XL PDF
      - IBM FDPR-Pro post-link optimizer
  - **Program developers** as part of the development cycle
Categories of Performance Monitoring Tools

- **Performance counting**
  - CPC (Cell Performance Counter)
  - hpmcount

- **Sample-based profiling**
  - Oprofile
  - IBM tprof
  - HP SPT/XL

- **Instrumentation-based profiling**
  - IBM FDPR-Pro instrumentation
  - Bprober
  - PIN

- **Tracing**
  - IBM Ctrace
  - PDT
Performance Analysis & Visualization Tools

- **Convenient Visualization tools**
  - Accept the output of the monitoring tools
  - Provide GUI for displaying and debugging code
  - Provide Integration to the IDE tools of the development cycle

- **Examples:**
  - IBM VPA (Visual Performance Analyzer)
  - Intel Vtune
  - Apple Shark
Performance Tools for Cell BE
Performance tools for Cell BE – relevant links

- Cell BE SDK ver. 3.1:
  http://www.alphaworks.ibm.com/tech/cellsw

- Cell IDE:
  http://alphaworks.ibm.com/tech/cellide

- VPA 6.3:
  http://alphaworks.ibm.com/tech/vpa

- IBM XL compiler for Cell BE:
  http://alphaworks.ibm.com/tech/cellcompiler

- GCC for Cell:
Cell Performance Tools

- Compilers and optimization tools
  - IBM XL compiler
  - GCC compiler
  - FDPR-Pro (Feedback Directed Program Restructurer)
    - performs global optimization at the entire executable
    - optimizes PPE/SPE executables

- Performance monitoring tools
  - oProfile
    - PPU Time and event profiling; SPU time profiling
  - Performance Debugging Tool (PDT)
    - Generalized tracing facility for instrumentation and trace data collection
    - instrumentation of DaCS/ALF, and other libraries in SDK
  - Hardware Performance Monitoring (CPC)
    - Interface to Cell performance monitoring facility to collect statistics on performance events
    - Perfmon2 support and enablement for PAPI, etc.
  - FDPR-Pro instrumentation Tool

- Performance analysis & visualization tools
  - Visual Performance Analyzer (VPA)
    - Visualize performance data collected by Cell performance tools through plug-ins of profile analyzer, count analyzer, trace analyzer, and code analyzer
    - display hierarchy views in analyzers to show hybrid elements.
    - support remote configuration and data collection in a hybrid system
  - PDTR (PDT Report)
    - Post processes PDT traces
    - Provide analysis and summary reports (Lock analysis, DMA analysis, etc.)
FDPR-Pro

• Feedback Directed Program Restructuring

• A post-link program optimizer
  • optimizations:
    ► code restructuring
    • inlining
    • loop unrolling
    ► hot-cold code motion
    ► many others optimizations

• Supports AIX, Linux and Window hosts, Power, z, and Cell targets

Diagram:
- Source files -> Compiler -> Object files -> Linker -> Executable program
- Executable program -> FDPR-Pro -> Instrumented program
- Execution workload -> Execution Profile
- Execution Profile -> Optimized program
- Debugger
VPA – Visual Performance Analyzer

- Eclipse based tool set for analyzing and visualizing gathered profiling and program code
- currently including 6 plug-in applications working collaboratively:
  - Profile Analyzer
  - Code Analyzer
  - Pipeline Analyzer
  - Counter Analyzer
  - Trace Analyzer
  - Control Flow Analyzer
Visual Performance Analyzer - architecture

Profile Analyzer  Code Analyzer  Pipeline Analyzer  Counter Analyzer  Trace Analyzer  Control Flow Analyzer

Eclipse Real Time Environment

Remote Data Collection Initiator

Windows/AIX/Linux(Intel)

Platform Specific Data Collectors

- AIX System
  - Tprof
  - hpmcount/hpmstat
  - FDPR-Pro
- Linux System
  - Perf. Inspector
  - OProfile
  - FDPR-Pro
  - pmcount
- Linux System
  - Perf. Inspector
  - OProfile
- Windows System
  - Perf. Inspector
- Linux Cell Blade
  - OProfile
  - Cell/PerfCounter
  - FDPR-Pro
  - PDT

Performance data

Other Systems

Platform specific tool(s)
Visual Performance Analyzer

- **Provide** platform independent, **easy to use integrated set of graphical application performance analysis tools**

- Leverage existing platform specific non-GUI performance analysis tools to collect a comprehensive set of data, for instance:
  - tprof and hpmcount on AIX
  - oProfile, pmcount and CellPerfCounter on Linux
  - Performance Inspector on Windows and Linux

- **Create consistent set of integrated tools to provide a platform independent drill down performance analysis experience**
Visual Performance Analyzer - current look

Using RCP allows VPA to be built with a custom appearance.

1. Windows, AIX, Linux
2. DB2 and HSQLDB support
3. Java profiling via JVMPI/JVMTI
4. Single tool framework using common data models
5. RSE for remote execution of test case and collection of data
CPC (Cell Performance Counter) - PMU features

- Four 32-bit counters for each Cell processor
  - Each can also be used as two 16-bit counters

- 1400+ events available to count

- Hardware sampling
  - Specify initial counter values and sampling time interval
  - PMU record and reset counter values after each interval
  - Samples available in hardware trace-buffer
    - Sample sequence can be annotated through writes to the PPU bookmark register
  - Thus reduce the number of calls that CPC has to make into the kernel

- Seven “logic islands”: PPU, PPSS, SPU, MFC, EIB, MIC, and BEI
  - Each island has groups of signals
  - Each signal represents an hardware event
  - PMU can monitor two signal groups
    - Monitor any number of signals within one group – up to the number of available counters
The fastest way to collect statistics about hardware events and consequently identify a delay is to use counting tools which count the number of times a certain hardware event occurred during run-time.

The counting is done in parallel to the execution of the application is always hardware supported.

Recommended when there is another version of the program to compare the statistics.

Example: IBM CPC command on Cell BE

```
cpc -e 2100,2101,2106,2109 -e 2103,2104,2111,2119 -c all --sampling-buffer-size 15 -i 100000000 -X fft_cpc2.pmf -t 10 ./fft 1 1 4 1 0
```

Resulted output file `fft_cpc2.pmf` will include sampling interval information for each of the following sampled events:

- Branch_Commit_t0
- Branch_Commit_t1
- Branch_Flush_t0
- Dispatch_Blocked_t0
- IERAT_Miss_t0
- IL1_Miss_Cycles_t0
- Instr_Flushed_t0
- PPC_Commit_t0
Counting tools cont. – example of displayed CPC
OProfile Overview

- **Periodically sample the program counter (PC)**
  - Give a statistical profile of which application, and where in the application the CPU was executing

- **Profile based on hardware performance counter events**
  - Support collecting profiles on multiple events at a time
  - Available performance counter events can be listed with
    - `opcontrol list_events`
  - `opcontrol` can also specify which event to use, frequency of sampling, and mode (user, kernel or both)

- **opreport generate the profiles**
  - XML output is supported, and is accepted by VPA

- **User's manual can be found at:**
  - OProfile home page: http://aixptools.austin.ibm.com/perf/w3_tools/oprofile/
OProfile – cont.

- **Command line Example:**
  - # opcontrol --separate=all --event=SPU_CYCLES:100000
  - # opcontrol --start
  - // Running the program to be profiled with a representative input
  - # opcontrol --stop
  - # opcontrol --dump
  - # opreport -X -g -l -d -o program.opm
Example- Oprofile displayed in ProfileAnalyzer view in VPA
PDT overview

- **The PDT is a Tracing Facility**
  - Real-time tracing of events on Opteron, PPE and SPEs at application level
  - Generic tracing API
  - All the relevant Opteron, PPE and SPE SDK functions are instrumented
  - Extensive information record on each event
  - Dynamic user-guided selective event tracking
  - Minimal interference with the application code – just a rebuild
  - Small amount of code added on the SPE
PDTR Overview

- **Command line PDT trace post-processor**
  - Support PDT's XML meta file trace format

- **Generates text output**
  - Quick PDT trace analysis before using VPA for further GUI based analysis

- **Produces various summary output reports**
  - Lock statistics
  - DMA statistics
  - Mailbox usage statistics
  - Overall event profiles

- **Also produces sequential report with time-stamped event and its parameters per line**

- **Provides full SPE and PPE, data and instruction address to name mapping, including for SPE overlays**
Trace Analyzer

- Read the PDT trace
- Fill in implicit data
  - Wall-clock time
  - Core ids
  - Context switches
- Show the trace
  - Graphical view
  - Text-based browser
  - Full record details on selection

Part of Visual Performance Analyzer
http://www.alphaworks.ibm.com/tech/vpa
Trace Analyzer - cont.

- Available as part of VPA 6.0
- Eclipse-based tool for trace processing and visualization
- Visualization from the hardware perspective
  - Data organized by core and under it by thread
- Synchronized graphical and textual views
Trace analyzer - current look

**TOP-LEFT BOX**
- Workspace
- Navigator
- Trace outline

**TOP-RIGHT BOX**
- Graphical view of the trace

**BOTTOM-LEFT BOX**
- Record details

**BOTTOM-RIGHT BOX**
- Color legend
VPA Trace Analyzer example

- **Zoom controls**
- **Time axis ruler**
- **Selection marker** – click to scroll to selection
- **Navigator view** – browse traces in your workspace, import more...
- **Graph will scroll to selection in outline view and vice versa**
- **Green background rectangle** shows the extent of an SPE thread
- **Dark border/light internals color-coding** make it easy to identify stalls
- **Click to edit the color**
- **Selected record details**
Code Analyzer - Motivations

- Architectures are becoming more and more complex
  - Some profiles are hard to understand by just looking at the source and disassembly codes

- Simulator help in detecting instruction level performance bottlenecks
  - But very hard to collect data, even harder to analyze

- Performance tool needed to statically analyze and visualize programs structure and performance characteristics for a specific platform design

- Code Analyzer perform static analysis on executable files and libraries/DLLs as well as dynamic analysis using profiling data from FDPR-PRO

- Features provided by Code Analyzer
  - Easy code navigation
    - by instructions, basic blocks, functions, hot loops, control flow graphs
  - Architectural comments generation
    - based on static analysis of code structure
  - Performance comments generation
    - when profile information is available
  - Dispatch group and pipeline functional units mapping
  - Source code mapping of executable
  - Code Analyzer
    - Analyzes profiling data from FDPR-Pro and profile analyzer
    - displays assembly code, basic blocks, functions, control flow graph, hot call graph, and annotated code, etc.
Analyzed profile output for fine tuning

- **Profile information can be further processed by static analysis tools to display**
  - Problematic code patterns which manifest potential stalls
    - Delays resulting from hardware modeling
      - Bad scheduling
      - Bad alignment
  - **Recommended code transformations**
    - Inserting compiler directives for improved performance
      - data prefetch directives
      - Method Inline directives
      - “final” directives
    - Code transformations to be done manually or by compiler flags
      - Unroll small critical loops
      - Inline dominant function calls
      - Code alignment for critically small areas
Example - Displayed analyzed code with profile in VPA
Code Analyzer – PPE sample views

Program tree of an executable file

Annotated Basic Block/Disassembly view

Annotated Source view

Detailed instruction information

Performance Comments
Code Analyzer – SPE sample views

- Program tree of an executable file
- Annotated Basic Block/Disassembly view
- Annotated Source view
- Detailed instruction information
- Performance Comments
Example of recommended tuning comment

```c
for (time=0; __builtin_expect(time<END_OF_TIME,1); time += ctx.dt) {
    ...
    for (i=0; __builtin_expect(i<ctx.particles,1); i+=PARTICLES_PER_BLOCK) {
        ...
        for (j=0; __builtin_expect(j<cnt,1); j++) {
            ...
        }
    }
}
```
Tuning code for Cell BE
Tuning code for Cell BE – Useful Links


Steps for Porting code to the Cell/B.E. platform

1. Analyze the code to find possible algorithmic improvements.
2. Identify the hot spot.
3. Port the code to run on Linux on the PPU.
4. Measure the performance of this hot spot code on the PPU.
5. Analyze the data flow to find good options for limiting bandwidth.
6. Analyze the memory alignment and memory allocation to maximize the “128-byte options”.
7. Create a structure as a "control block" to hold the SPE initialization data.
8. Determine how the work can be parallelized and off-loaded across multiple SPEs.
9. Port the code to the SPUs.
10. Establish communication strategy and if possible, multi-buffer all DMAs.
11. SIMDize and unroll the inner loops of this hot spot code.
12. Measure the performance, using static and dynamic timing tools.
Example: FFT

Chow et al., A Programming Example: Large FFT on the Cell Broadband Engine, 2005
Implementation available as part of Cell BE SDK 3.0 sample code

- Trace of an FFT execution
- Hue indicates event duration
  - Light hue means longer event
FFT, zoomed in

- DMA stalls are way too long
- Looks like the mail marks a phase change
FFT: behavior by phase

- Phase synchronization overhead negligible
- Stall time is huge!
  - 70% of phase time on average
  - Up to 85%
- Different behavior by stage
What did we do wrong?

- Let’s try the classics
- Memory accesses across multiple pages
  - Page faults
  - TLB thrashing
  - Might be avoided by using large and huge page sizes

Daniel Brokenshire, *Maximizing the power of the Cell Broadband Engine processor*
FFT with huge pages

(small page strip for comparison)
FFT with huge pages: behavior by phase

(a) Small pages

(b) Huge pages
Example: Julia set

- Like in FFT, we see phases marked by mail.
- But now phase synchronization is dominant!
Julia load balancing
Julia set – work set changes
Julia set without load balancing (low load)
Julia set without load balancing (high load)
Overhead Matters

FFT

FFT*

FFT* -- modified FFT configuration