CScADS Workshop on Autotuning for Petascale Systems

Auto-tuning Compilers

Kevin O’Brien
IBM Watson Research Center

© 2008 IBM Corporation
Propositions/Questions Addressed

- **Propositions I am supporting**
  - Proposition: The focus on specialized tuning systems is too narrow, and so only compilers, which apply most broadly, are the most sensible investment.
  - Proposition: Runtime optimization will catch opportunities for improvement that neither a compiler nor a neither an autotuned library can.

- **Propositions I disagree with**
  - Proposition: Self-tuned libraries will always outperform compiler-generated code.
Main Themes

- We need compilers to fully exploit the potential of autotuning
  - Libraries don’t cover the full design space of programs
    - if they did, we wouldn’t be here talking about this
  - Programs are much more than a structured composition of calls to standard libraries
  - Compilers have a detailed view of the specific application code
    - compilers running at link-time can do whole program analysis
  - Deficiency in compiler applicability is the limitation to static analysis
    - partial (albeit unsatisfactory) resolution is profile directed feedback

- We really need a combination of compilers and run-time monitoring
  - Traditional Compiler strength in offline (static) analysis
  - Profile that tracks actual execution
    - recognition of phase changes
    - always monitoring, need smart ways to make the cost of instrumentation vanishingly small (compared to speed-ups)
  - Nimble and flexible dynamic re-adaptation of code
    - Can be based on offline pre-planning
    - Can exploit underutilized threads to asynchronously adapt program
A Digression: The Cell Processor

- Cell Architecture (CBEA)
- Cell Programming models
- XL Compiler for Cell
Cell Broadband Engine

- L2 (512KB)
- SPE
- SPE
- SPE
- SPE
- Element Interconnect Bus
- Power Processor Element
- Test & Debug Logic
- Memory Controller
- Rambus XDRAM Interface
- I/O Controller
- Rambus FlexIO™
Cell Programming

- Partition application into PPE and SPE portions
- Compile PPE and SPE portions separately
- Code streaming data portions for MFC
- Parallelize across multiple SPEs
- Exploit SIMD features

64-bit Power Architecture with VMX
Compilation Model
Programmability is the biggest problem

- Complex systems have potential for high performance
  - very few expert programmers
  - current tools require high level of expertise
- In the late ‘50s, the switch from assembler to HLLs (FORTRAN) was enabled by the development of compilers
- Today, we are in a very similar position to the pre-FORTRAN era
  - explicit parallel/SIMD/DMA
  - need the equivalent new technology to get back on track
- New languages and libraries like CUDA, ALF/DACS may help mainly the expert programmers
  - need languages that express high-level intent, not details of implementation
OpenMP Compiler for Cell

```
Seq1();
#pragma omp parallel
{
  Par();
}
Seq2();
```

- Single source code
- C/C++/Fortran
- OpenMP programming model

- Outline parallel region
- Code overlay
- Software cache
- Direct buffering
- Auto SIMD

- Parallel region runs on PPE/SPEs
- Data/task parallelism
- Runtime scheduling
Software Cache – Data Structure

TAG
- tag
- base
- dirty_bits

DATA
- [0]
- [1]
- [2]
- ...
- [511]

0-way
- 0x10008000
- [1]
- ...

1-way
- ...

2-way
- ...

3-way
- ...

• 4-way set associative
• TAG size 16K (128 x 4 x 32B)
• DATA size 64K (512 x 128B)
• SIMD for tag comparison
• Cache lookup inlined by TOBEY, ~16 cycles
Problem Statements

- Coherence problems
  - It is possible to have two copies of a variable in local memory at the same time, one in software cache and the other in direct buffer.
Solutions

- **Separate transfers**
  - A variable either goes to software cache or direct buffer
  - No redundant copy, no coherence
  - Whole program analysis

- **Hybrid transfers**
  - A variable goes to both software cache and direct buffer
  - Maintain two copies, make the values in sync
  - Compiler analysis
    - No cache access within the tiled loop
    - Make sure the value sync only happens at the loop entry & exit
  - Runtime check
    - For read buffers, update the value from software cache after DMA get
    - For write buffers, update software cache after DMA write

```c
for (ii=0; ii<N; ii+=bf) {
    n = min(ii+bf, N);
    DMA get A[ii:n] to A['];
    Coherence maintenance for A;
    for (i=ii; i<n; i++) {
        B'[i] = A'[i] * S;
    }
    DMA put B'[] to B[ii:n];
    Coherence maintenance for B;
}
```
Multi-dimensional problem

- Time of application
  - “Compile time”
  - “Execution time”
    - both of these concepts get stretched (later)
- Range of potential targets
  - memory system
  - processor pipeline
  - parallelism
  - choice of machine organization or ISA
- Aspects of the hardware that influence performance
  - number and type of execution threads
  - cache configuration
  - ...
- Aspects of application behavior that affect performance
  - phase changes
Time of Application

- **Compile Time**
  - traditionally offline, can take a lot of time
  - mainly focused on the execution environments behavior, as it intersects the particular application
  - must be aware of the target execution environment (cross compiler issue)
  - cannot take account of execution behavior except through “training runs”
  - compiler can build experiments (constructed from the source) to determine “good” values for parameters etc
    - example: tile sizes in the polyhedral model
    - example: unrolling factors
  - has some similarity to the way that autotuning of libraries is done
Time of Application

- **Execution Time**
  - traditionally online, usually constrained by requirement to speed up, rather than slow down the application
    - Java compilers like Testarossa(IBM) and HotSpot(SUN)
  - has access to profile data from the current execution of the program
    - can be aware of phase changes
    - much more data can be collected than in conventional PDF
    - interaction between compiler and monitoring system can pose questions (experiments) that reveal more information about interesting program behavior
  - in a petascale (massively parallel) system, under-utilized execution contexts can be pressed into service of the compiler
    - allows a type of “offline” dynamic compilation
Range of potential targets

- **memory system**
  - tiling parameters and unroll factors
  - delinquent load amelioration
  - complex prefetch patterns
  - dynamic control of stream hardware engines
  - remapping data-structures
    - whole program analysis, remapping dynamically for phase changes

- **parallelism**
  - speculative execution
    - based on profile data, radically optimized code can be chosen
    - need to be able to monitor and back-out
  - dynamic (in)dependance discovery
  - dynamic re-scheduling
  - choosing between alternative levels of parallelism

- **trace optimization**
  - dynamic hyperblock formation
    - online scheduling of hyperblocks
  - reducing branch mispredicts
Range of potential targets

- **choice of machine organization or ISA**
  - accelerators (either the same or different ISA to core)
    - source fragments may be compiled to multiple targets
      or to the same target but with different pipeline/frequency
    - choose which version to run
      depends on execution characteristics of the application
    - may require management of code and data transfers (eg Cell SPU)
    - need to monitor and evaluate these decisions

- **processor pipeline**
  - codes may be statically compiled to a different model
Aspects of the hardware that influence performance

- **number and type of execution threads**
  - number of cores/SMT threads
  - presence of accelerators
    - same ISA but different performance
    - different ISA
    - SIMD units (and their alignment requirements)
    - floating point compatibility between processors

- **cache configuration**
  - level of sharing between threads, cores, chips, nodes, ...
  - ... and the bandwidths, latencies and geometries

- **speculation support in hardware**
  - TLS, TM

- **Interconnect topology**
  - support for distributed memory
  - DMA engines etc
    - are they programmable?
Aspects of application behavior that affect performance

- **Execution path**
  - iteration counts (profitability of SIMDization, parallelization)
  - hyperblock formation
  - branch penalties
    - not all processors have (good) branch prediction hardware
    - (also a software/hardware tradeoff)

- **Phase changes**
  - can we recognize them?
    - fast enough?
    - can we react effectively?

- **Dynamic dependance structure**
  - for unsolvable dependances, are there patterns?
Hardware Support

- Do we need it?
- What should it look like?