



Performance Improvement and Tuning on POWER8 Processors

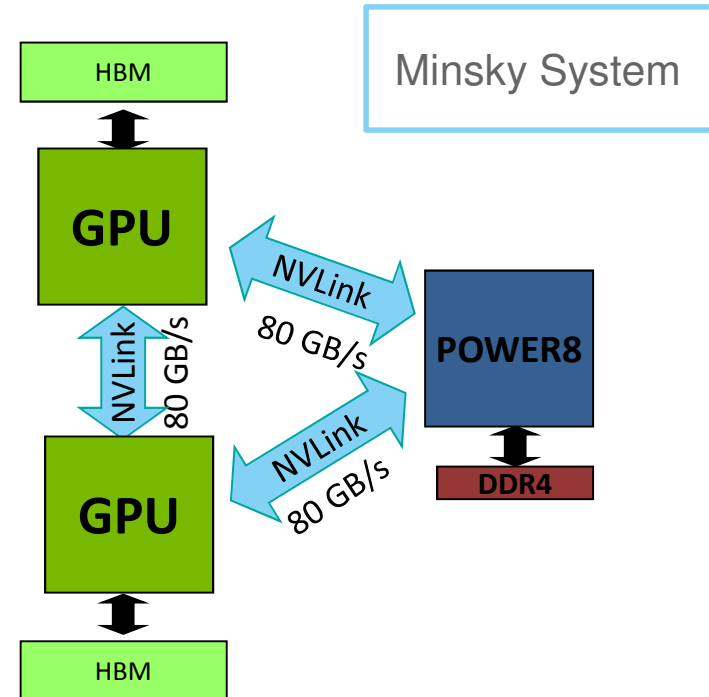
Dr. Archana Ravindar
IBM Systems
Supercomputing 2017

*For a more recent version of slides please refer to
<https://indico-jsc.fz-juelich.de/event/53/>*



Motivation for the Session

- Heterogenous Architectures consist of the CPU along with Accelerator Units such as GPU
- Performance of applications run on these systems depends on performance of CPU and GPU
- If Performance of CPU is sub-optimal it can gate the performance of the overall system
- Its worth learning about strategies to improve performance of applications on CPU



NVLink Enables Fast Unified Memory Access
between CPU & GPU Memories

Role of the Compiler, Source changes on CPU performance

- Modern day Processors are designed on a *Pipeline based architecture*
- A Compiler's job is not only to generate code that can run on a native platform but also to optimize code such that the code flows smoothly through the pipeline without causing bottlenecks
- The optimization phase is where the Compiler carries out this Job
- Compiler optimization strategies are designed to work ideally well for all cases
- To achieve extra performance in specific situations, one can use pragmas/flags
- Hand Tuning based on source and assembly changes also gives an extra boost to performance



Ways to Optimize CPU Performance

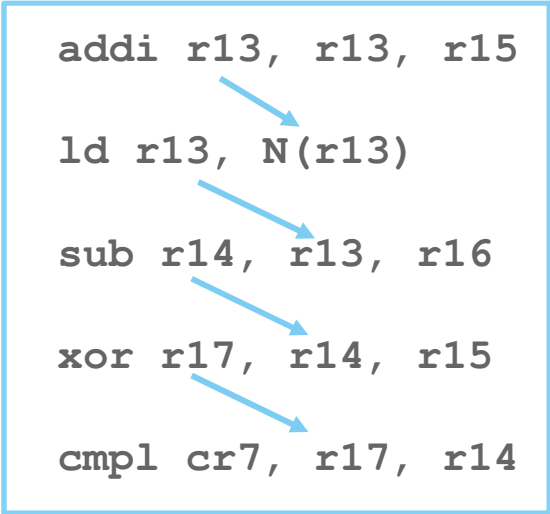
- Most common reason for low CPU Performance- Pipeline Bottlenecks-
 - Excessive Dependencies among Instructions
 - Branches / Too much decision making
 - Cache Misses
 - Load hit stores
- Additional Ways to optimize CPU Performance
 - SIMDization, Serial v/ parallel, Source code Tuning

Strategies to Work Around Pipeline Bottlenecks

- We will touch upon today how we can work around these bottlenecks -
 - Excessive Dependencies among Instructions
 - Better Scheduling- Inlining- Unrolling
 - Branches / Too much decision making
 - Inlining, Indirect Call Promotion, source code strategies
 - Cache Misses
 - Prefetching, Data structure/code rewrite
 - Load hit stores
 - Inlining, vectorization

Excessive Dependencies among Instructions

- Dependencies brakes speed of execution : Inspite of having multiple functional units and pipeline of these units, Execution becomes serial
- Solution: Scheduling
- Place non-dependent operations in between such a chain of instructions
- Compiler and the hardware does this automatically
- The following optimizations/helps the compiler do a better job of scheduling- Inlining, Unrolling
- Inlining
 - qinline=auto:level=N (XL)
 - finline-functions(LLVM, GCC)
 - __attribute__((always_inline)) Type func-name(params) {... }
- Unrolling:
- -qunroll, -funroll-loops



```
addi r13, r13, r15
ld r13, N(r13)
sub r14, r13, r16
xor r17, r14, r15
cmpl cr7, r17, r14
```

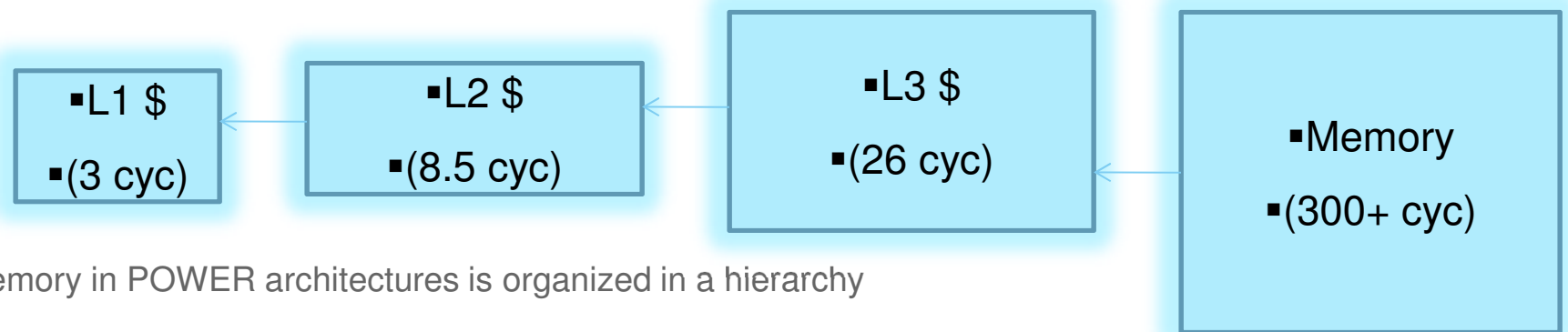
The diagram illustrates a sequence of five instructions within a light blue rectangular box. Blue arrows point from the right-hand side of one instruction to the right-hand side of the next instruction below it, indicating a chain of data dependencies. The instructions are: `addi r13, r13, r15`, `ld r13, N(r13)`, `sub r14, r13, r16`, `xor r17, r14, r15`, and `cmpl cr7, r17, r14`.

Branches

- Excessive branches can also brake execution speed
- POWER8 processor has a branch mispredictor that can predict branch direction and allow execution to go on based on predicted branch decision
- How you can help speed up performance?
 - Use ?: for one line branches
 - Compiler generates POWERpc instruction isel for one line branches (-misel)
- Provide hints in source code to indicate the expected values of expressions appearing in branch conditions (long __builtin_expect(long expression, long value);) *(hint whether branch is more likely to be taken/not)*

```
if (CAPTURED(themove) == s->sboard[FROM(themove)]) {
    prob -- 30;
}
if (s->sboard[FROM(themove)] == wpawn || s->sboard[FROM(themove)] == bpawn) {
    prob -- 30;
}
if (CAPTURED(themove) != npiece) {
    prob -- 10;
}
if (CAPTURED(themove) == wqueen || CAPTURED(themove) == bqueen) {
    prob -- 30;
}
if (PROMOTED(themove) != 0
    && PROMOTED(themove) != npiece
    && PROMOTED(themove) != wqueen
    && PROMOTED(themove) != bqueen) {
    prob += 40;
}
```

Cache Misses



- Memory in POWER architectures is organized in a hierarchy
- L1 cache : Closest memory to the processor and the fastest, followed by L2, L3 upto main memory
- Memory is most distant to the processor and slowest
- Data cache : stores data, instruction cache: stores instructions
- Data cache misses can stall instructions in the pipeline causing a cascading effect on all those instructions dependent on it data
- If the compiler has done a good job of scheduling, usually the processor can find some thing else to do in the meanwhile

Hardware Prefetching

- Power8 has the most advanced Prefetch Engine!
- Access patterns are automatically detected by the hardware (Patterns must be regular and not random to be detectable by HW prefetcher)
- Hardware Prefetching is staged
 - Data is brought to L3 first
 - Later transferred from L3 to L1
- Prefetch requests are paced by rate of consumption
 - Minimizes Pollution of cache
- To turn on Prefetching (`ppc64_cpu -dscr=0`)
- Specific levels of prefetching can be set by assigning appropriate values to DSCR by `ppc64_cpu -dscr=0xNNN`

Software Prefetching

- Under programmer's control
- Load, store address/stream can be specified (dcbt/dcbtst)
 - Requires us to include the header file <builtins.h>
 - Ex: `__dcbt((void*)address);`
- Total number of streams possible: 16
- GCC compiler: programmers can use `__builtin_prefetch(address)`
- XL compiler: `-qprefetch=aggressive` will automatically insert dcbt calls
- GCC compiler : `-qprefetch-loop-arrays`

If Prefetching does not help

- If you have to live with cache misses and cannot prevent it
- The compiler should have enough leg room to move instructions around so that it places variables that are loaded from cache far from where it is used
- Ways to increase opportunities for compilers-
 - Inlining
 - Unrolling
 - Indirect call Promotion

Load Hit Stores

```
Array[address]=value // Store  
...  
= ... Array[address] // Load
```

- Can occur while reading something immediately which you have just written
- LHS occurs when a load instruction tries to load an address to which there has been a recent store prior to it (which might not have completed yet)
- LHS causes an instruction to be rejected and reissued
- In essence, an instruction that faces load hit stores causes itself and other dependent instructions to take longer to execute
- Penalty in cycles: (Normal: 3 cycles LHS : upto 20 cycles!!)
- POWER9 architecture is designed to avoid any penalty due to Load hit stores

Pragmas/Flags to work around Load Hit Stores

If such situations are inevitable, use `#pragma unroll(4)` or `unroll(8)` (*pragma will span out instructions so that compiler can schedule them in such a way to avoid LHS*)

```
#pragma unroll(8)
for(i=0; i<N; i++) {
    for(j=0; j<n-i; j++) {
        if(array[j] > array[j+1]) {
            .....
            array[j+1]=...;
        }
    }
}
```

Use aggressive inlining to reduce LHS (*reduces save/restore operations in short routines*)

- XL: `-qinline=auto:level=N` (N ranges from 1 to 10). LLVM/GCC: `-finline-functions`
- Even at assembly level, you can insert nops between the offending store and load to reduce impact

Summary of Flags

Flag Kind	XL	GCC/LLVM	Equivalent pragma / attributes	Benefit	Drawbacks
Unrolling	"-qunroll"	"-funroll-loops"	#pragma unroll(N)	Unrolls loops ; increases opportunities pertaining to scheduling for compiler	Increases register pressure
Inlining	"-qinline=auto:level=N"	"-finline-functions"	Inline always attribute	increases opportunities for scheduling; Reduces branches and loads/stores	Increases register pressure; increases code size
Enum small	"-qenum=SMALL"	"-fshort-enums"	-	Reduces memory footprint	Can cause issues in alignment
isel instructions		"-misel"		generates isel instruction instead of branch; reduces pressure on branch predictor unit	latency of isel is a bit higher; Use if branches are not predictable easily
General tuning	"-qtune=POWER8"	"-mcpu=power8"			
64bit compilation	"-q64"	"-m64"			
Prefetching	"-qprefetch"	"-fprefetch-loop-arrays"	__dcbt/__dcbtst, builtin_prefetch	reduces cache misses	Can increase memory traffic particularly if prefetched values are not used
Generate annotation reports	"-qlist -qreport"	"-fdump-tree-all" for GCC "-Rpass=[a-z]*" for LLVM			

Additional Ways to Optimize CPU Performance

- Single Instruction Single Data (SISD) v/s Single Instruction Multiple Data (SIMD)
 - If similar operation has to be performed on multiple data, use vectorization
- Serial v/s Parallel Execution
 - If there are multiple tasks which do not have a dependency amongst each other and can be done in parallel use a framework such as OpenMP that can perform Tasks in $1/N$ th time with N threads
- Source code change
 - Ex: Costly library calls: Try writing a macro instead of making a call whenever possible

SISD v/ SIMD

- Compiler automatically vectorizes code which has repeating instructions for multiple data units when we use higher optimization flags, -qvsx -qaltivec (XL), -mvsx, -maltivec(LLVM,GCC)
- How the users can help compilers automatically vectorize code
 - Avoid branches inside loops, avoid excessive use of pointers inside hot loops
- SIMD in OpenMP: Pragmas to support SIMDization
- Following Loop gets vectorized and executes faster compared to non SIMD version

```
#pragma omp simd
for(int i=0;i<NUM;i++)
    c[i]=a[i]*b[i]-500;
```


OpenMP SIMD pragmas

- Additional helpful OpenMP SIMD pragmas : inbranch, notinbranch
- `#pragma omp simd inbranch`
- Function always called from inside an if statement
- `#pragma omp simd notinbranch`
- Function never called from inside an if statement
- If loop has parallel instructions along with a cumulative operation, use reduction attribute

```
#pragma omp simd reduction(+:loopsum)
for(int i=0;i<NUM;i++) {
    c[i]=a[i]*b[i]-500;
    loopsum+=c[i];
}
```

Advantages of SIMD-ization

- Reduces number of instructions executed as a single instruction executes on multiple data at a given time
- Uses vector registers and lessens burden on regular CPU registers
- Reduces Memory Traffic
- Improves Performance and Instructions executed per cycle (IPC/throughput)

Serial V/s Parallel Execution

If your work has several parallel components you can use OpenMP pragmas to run code in parallel

```
#pragma omp parallel for
    for( int ix = ix_start; ix < ix_end; ix++ )
    {
        A[iy*nx+ix] = Anew[iy*nx+ix];
    }
```

The parallel for pragma can be used for nested loops as well

```
#pragma omp parallel for
for( int iy = start; iy < end; iy++ ) {
    #pragma omp parallel for
    for( int ix = start; ix < end; ix++ ) {
    }
}
```

Controlling number of Threads

- We can control the number of threads at the command line when executing a parallelized version of code using omp parallel for
- OMP_NUM_THREADS=8 time ./application <params>
- OMP_NUM_THREADS=16 time ./application <params>
- OMP_NUM_THREADS=32 time ./application <params>
- The ideal number of threads depends on the application that is parallelized
- It also depends on number of cores, SMT levels, threads per core
- Large number of threads do not always help as thread book-keeping overhead can overtake benefit of parallelization

Binding Threads to CPU

- We can use `GOMP_CPU_AFFINITY="0 8 16 24 32 40 48 56" OMP_NUM_THREADS=8` time `./application <params>` to bind first thread to CPU0, second thread to CPU8, ... so on
- The ordering of CPU numbers determines performance of the application
- If all threads are bound to a single CPU execution speed slows down
- To choose the right CPU number on a POWER Linux system, we can consult the file `/sys/devices/system/cpu/cpu0/topology/thread_siblings_list`
- If the POWER Linux system is configured to be SMT=8 this file contains
0-7
Indicates threads 0-7 run on CPU0 while 8-15 run on CPU1, and so on...

Source code Tuning: Example Costly Library Function Calls

In some situations, performance of your application is gated by a costly library call

```
33.17% poisson2d poisson2d      [.] poisson2d reference
26.98% poisson2d poisson2d      [.] 000000c9.plt_call1.fmax@@GLIBC_2.17
25.82% poisson2d poisson2d      [.] main
```

Typically applications tend to use a lot of library calls

Sometimes one can replace the function call by a simple macro

For Illustration purposes-

```
// Math function call double fmax(double x, double y);
error = fmaxr( error, fabsr(Anew[iy*nx+ix]-Aref[iy*nx+ix]));
can be replaced by
double tmp = fabsr(Anew[iy*nx+ix]-Aref[iy*nx+ix]);
error = (error > tmp) ? error:tmp ;
```

In closing

- CPU performance is as important as GPU performance in a Heterogenous architecture
- We saw strategies to improve CPU performance
 - By avoiding bottlenecks possible in the pipeline
 - By vectorization strategies
 - By Parallelization strategies
 - Additional strategies to improve performance
- The Hands-on Session will touch upon examples on some of the strategies discussed in today's session

Disclaimer: This presentation is intended to represent the views of the author rather than IBM and the recommended solutions are not guaranteed on sub optimal conditions.