

Application Performance Tuning on POWER9- with Compilers

—

Dr. Archana Ravindar
(@aravind5@in.ibm.com)

Contents

Part One

Introduction and scope	03
Salient Features of POWER9	04
Tools used in the Presentation	06

Part Two

Optimizing Front End Performance	07
----------------------------------	----

Part Three

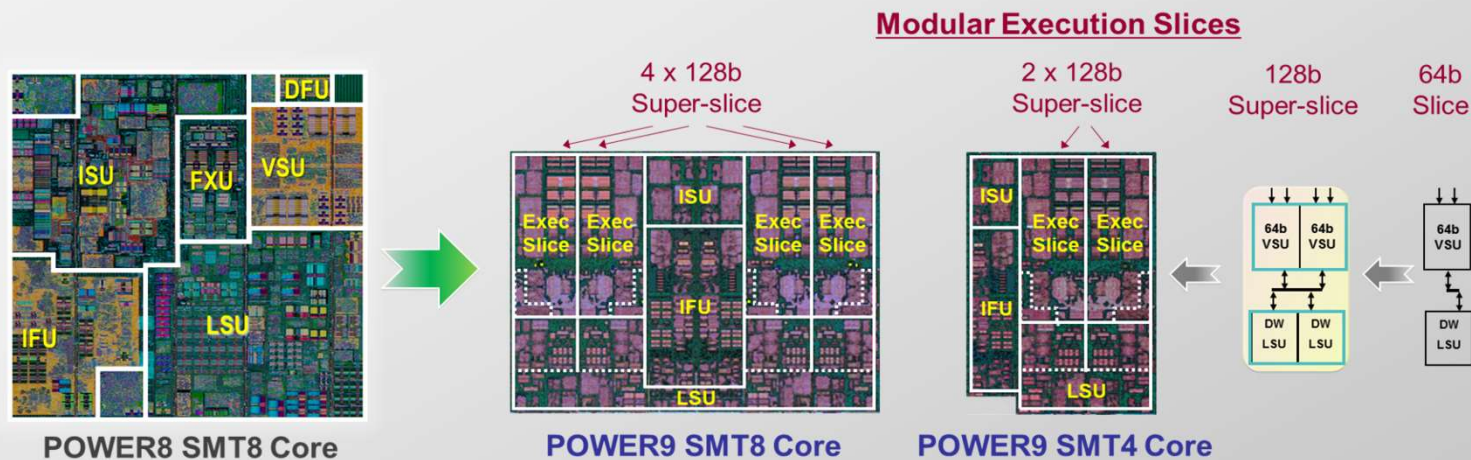
Optimizing Back End performance	11
Additional ways of Tuning Performance	18
Compiler Flag Comparison – XL, GCC, Clang	19

Summary	20
----------------	-----------

Scope of the Presentation

- Outline Tuning strategies to improve performance of programs on POWER9 processors
- The strategy is directed by program characteristics which can be assessed by hardware performance counters
- These strategies can take the form of compiler flags, source code pragmas/attributes
- This talk addresses overall summary of options supported by open source compilers such as GCC, LLVM and IBM proprietary compilers such as XL
- Tools used to measure performance counters- perf / PAPI

POWER9 Processor

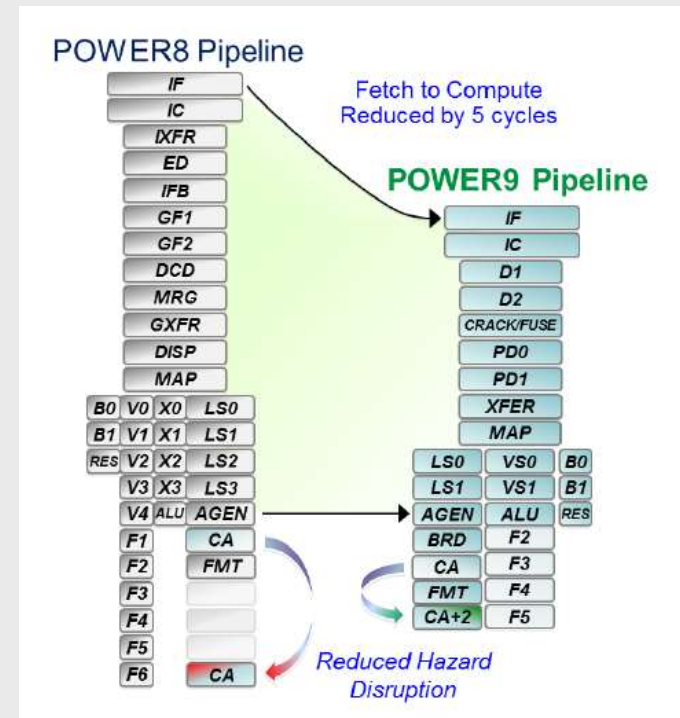


Reference: IBM Power9 Processor Architecture, S. Sadasivam, et al, IEEE Micro, Volume 37, Issue : 2, Mar-Apr 17

- Optimized for Stronger Thread Performance and Efficiency
- Increased Execution Bandwidth efficiency for a range of workloads including commercial, cognitive and analytics
- Sophisticated instruction scheduling and branch prediction for unoptimized applications and interpretive languages

POWER9 Core Pipeline Efficiency

- Shorter Pipelines with reduced disruption
- Improved Application Performance for Modern Codes
- Advanced Branch Prediction
- Higher Performance and Pipeline Utilization
 - Removed instruction grouping
 - Enhanced instruction fusion
 - Pipeline can complete upto 128 (64-SMT4) instructions /cycle
- Reduced Latency and Improved Scalability
 - Improved pipe control of load/store instructions
 - Improved hazard avoidance



Reference: IBM Power9 Processor Architecture, S. Sadasivam, et al, IEEE Micro, Volume 37, Issue : 2, Mar-Apr 17

Tools Used in the Discussion

- Open source compilers such as GCC, LLVM and Proprietary compilers such as XL
 - `[gcc| clang | xlc] -O[n] program.c -o program` for C programs
 - `[g++| clang++| xlc] -O[n] program.cc -o program` for C++ programs
 - Optimization level ranges from 0 to 3, Ofast for GCC, LLVM and upto O5 for XL
 - `-mcpu=power9` for targeted code generation on POWER9
 - Profile directed feedback
- Perf tool
 - To record hotspots/profile application
 - `perf record -e r<code> ./binary args > out` (produces perf.data)
 - `perf report` (opens profile report stored in perf.data)
 - To measure hardware events
 - `perf stat -e r<code> ./binary args > out`
 - These counters can be read using PAPI API as discussed in the previous session

Performance Tuning in the Front-End

- Front end fetches and decodes the successive instructions and passes them to the backend for processing
- POWER9 is a superscalar processor and is pipeline based so works with an advanced branch predictor to predict the sequence and fetch instructions in advance
- However if there is a misprediction, it causes wrong path instructions to be fetched and introduces additional penalty as these instructions need to be flushed from the pipeline and correct instructions need to be fetched and processed
 - Counters to detect this: **PM_BR_MPRED***
- Branches are caused even by function calls, Such branches affect instruction cache locality and increase instruction cache misses; Indirect function calls with no patterns make it difficult to predict with accuracy and can cause Instruction Cache Misses
 - Counters to detect this: **PM_L1_ICACHE_MISS**

Tuning Strategies to improve Front End Performance

- Unrolling loops (will reduce loop branches and in some cases branches within loop)
- Manual unrolling in source
- Language support to do unrolling in source
 - Place `#pragma unroll(N)` before the for loop which needs to be unrolled
- Compiler support for controlling Unrolling
 - Enable Loop Unrolling: `-funroll-loops`: Leave it to the compilers judgement to decide optimal unrolling for each loop
 - Disable Loop unrolling : `-fno-unroll-loops`

```
int x;  
for (x=0;x<100;x++)  
{  
    func(x);  
}
```

```
int x;  
for (x=0;x<100;x+=5)  
{  
    func(x);  
    func(x+1);  
    func(x+2);  
    func(x+3);  
    func(x+4);  
}
```


Tuning Strategies to improve Front End Performance

- Shorter routines are best to be inlined to avoid call overhead and reduce branches
- Broadens the window available for better scheduling
- Manual function inlining in source
- Language Support– use inline `__attribute__((always_inline))` in front of a function definition
- Compiler Support
 - `-finline-functions`(GCC, LLVM), `-qinline(XL)` : Inline suitable functions
 - Compiler supports inlining thresholds : number of instructions in a function before it can be considered for inlining
- Help Compiler to Inline more Functions:
 - Convert Indirect calls to direct calls

Tuning Strategies to improve Front End Performance

```
If (val==M)  a=b;  
            else a=c;
```

```
a= (val==M) ?b:c;
```

- If you have a branch that assigns a different value depending on a condition we can convert it to a one line assignment statement with the ?: operator
- Compiler generates isel instructions for such branches that essentially converts a control dependency into a data dependency
- GCC/LLVM option to generate isel : -misel. To Disable generation of isel: -mnoisel
- For simple branches as below, we may leave the branch statement as it is; The branch predictor will automatically take care of the performance

```
If (val>M)  M=val;
```

- Other techniques to improve performance: 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*)

Tuning Strategies to Improve Backend Performance

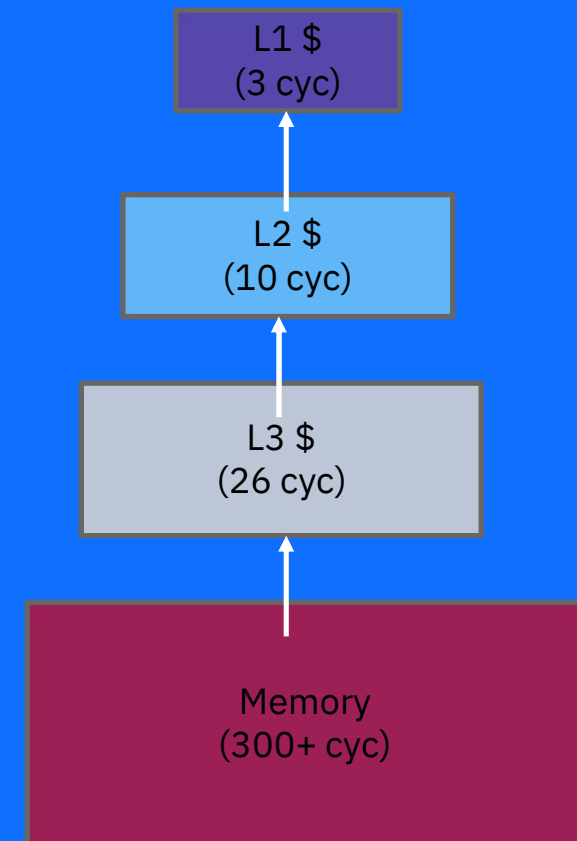
- Backend of the processor is concerned with execution of the instructions
- There are numerous ways we can tune performance in the backend
 - Using Processor Resources effectively-
 - Registers,
 - Caches,
 - Prefetching
 - Vectorization

Using Processor Registers Effectively

- CPU Registers are an important resource for execution and are finite in number
- Compiler has to judge and allocate variables into registers to avoid spilling as much as possible
- Spill is a mechanism when a register value is saved on to memory for later use as the register is required to do some thing else
- Each spill will require one store + a future load (to use that value from memory)
- If distance between store and load is short it may cause a dependency chain in the pipeline
- Stalls due to spills can be detected using the following counters- **PM_LSU_FIN, PM_LSU_FLUSH,**
- Use other register resources like SIMD registers if applicable (Vectorization)
- Use multipurpose instructions such as andc (logical AND complement), orc (logical OR complement) which combines multiple math operations in a single instruction saving a register,
- Record instructions such as addi. That does the operation and also set the CR fields

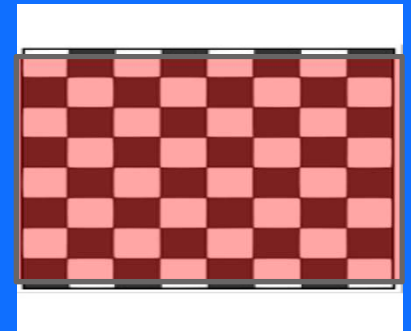
Using Caches Effectively

- Memory is organized in a hierarchy
- L1 cache : Closest memory to the processor and the fastest, followed by L2, L3 upto Memory
- Memory is most distant to the processor and slowest
- Data cache : stores data, instruction cache: stores instructions
- Data cache misses can stall load instructions in the pipeline causing a cascading effect on all those instructions dependent on it data
- Counters- **PM_LD_MISS_L1**, **PM_CMPLU_STALL_DCACHE_MISS**, **PM_ST_MISS_L1**, **PM_CMPLU_STALL_DMISS_L2L3**, **PM_CMPLU_STALL_DMISS_LMEM** etc



Memory footprint reduction (enum=small)

- Use only as much memory as required
- Game applications such as chess, GO contain data structure to represent board
- Each square has a fixed number of states, n , which is usually a single digit number
- Usually such a data type is defined as an enumerated data type such as
 - `typedef enum {BLACK=0, WHITE=1,...} square;`
- Typically each element of square will be allocated as
- type “int” (4 bytes)
- Compile the application with `-qenum=small(XL)` or `-fshort-enums(LLVM, GCC)` that allocates only minimum required memory to store each
- element of square (1 byte),
- $1/4^{\text{th}}$ memory will be used



Prefetching

- Hardware prefetching
 - Controlled by DSCR (data stream control register) settings;
 - `-ppc64_cpu --dscr=<n>`
 - Common DSCR configurations: n=0 (moderate depth, ramp) : By default the HW prefetcher is “ON”
 - n=0x1D7 (Achieve most aggressive depth, most quickly, enable stride N prefetch),
 - n=0x1 (no prefetch)
 - Reference: <https://developer.ibm.com/linuxonpower/docs/linux-on-power-application-tuning/>

Software Prefetching

- Programmer inserted prefetch instructions `__dcbt` (load prefetch), `__dcbtst` (store prefetch)
- If you want to explicitly control prefetching via software **only**, you can turn off hardware prefetching using
 - `ppc64_cpu -dscr=1`
- Compilers provide an option to enable compiler to insert prefetch instructions wherever applicable. However this is only a directive that can be ignored by the compiler if it does not see adequate benefit
- XL: (`-qprefetch/-qnoprefetch`) and GCC (`-fprefetch-loop-arrays/-fno-prefetch-loop-arrays`); Some compilers may include further loop optimizations when `-fprefetch-loop-arrays` is invoked.
- XL/POWER9 supports setting DSCR values at compile time for an application

Example: `-qprefetch=dscr=<value>`

Compiling with the option `-qprefetch=dscr=7` sets the prefetch level to 7

Compiling with the option `-qprefetch=dscr=1` turns off hardware prefetching and is equivalent to

`ppc64_cpu -dscr=1`

- Useful in cases where we cannot obtain root privileges to play with prefetch settings on the command line

Vectorization

- Compute intensive programs whose computations can be parallelized can take advantage of vector instructions on POWER
 - Advantages- reduces loads, stores and hence pathlength, reduces register pressure on GPRs, effective use of resources, Faster throughput
- At a time- Vector instructions can work on **4 32 bit words**, **8 half-words** and **16 bytes**
 - Amount of work done per unit time correspondingly becomes faster
- Clang/GCC: -ftree-loop-vectorize, -mvsx, -maltivec, -mllvm -force-vector-width=n(Clang only),
- Help the Compiler to automatically vectorize loops
 - Keep the loop simple
 - Avoid extensive branches, pointer references within loops (use restrict wherever applicable)
- GPU codes can scale really well with SIMDization for performance
 - Structure of arrays are more amenable to vectorization than array of structures

Additional Ways to Improve Performance

- Serial v/s Parallel Execution: tasks that don't have dependencies can be done in parallel using a framework such as OpenMP that can perform Tasks in $1/N$ th time with N threads
- If Mathematical accuracy is not important, use -Ofast
 - This automatically substitutes expensive library calls to native implementation of the math function using target ISA
- Thread Binding
 - We can use `GOMP_CPU_AFFINITY="0 5 10 15" OMP_NUM_THREADS=4 time ./application <params>` to bind first thread to CPU0, second thread to CPU1, ... 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`
- Large Pages- allow TLB to map to a larger virtual memory page thereby reducing TLB misses. Memory intensive applications that use large amounts of virtual memory can benefit with using largepages

Flag Kind	XL	GCC/LLVM	Can be simulated in source	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	-qarch=pwr9", -qtune=pwr9"	-mcpu=power8, -mtune=power9			
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
Link time optimization	-qipo	-flto , -flto=thin		Enables Interprocedural optimizations	Can increase overall compilation time
Profile directed feedback	-qpdf1, -qpdf2	-fprofile-generate and -fprofile-use LLVM has an intermediate step llvm-profdata		Enables hot path optimizations	Requires a training run

Summary

- Today we talked about
 - Tuning strategies pertaining to the various units in the POWER9 HW –
 - Front-end, Back-end
 - Some of these strategies were compiler flags, source code pragmas that one can apply to see improved performance of their programs
 - We also saw additional ways of improving performance such as parallelization, binding etc
 - We saw that POWER9 has the most comprehensive set of hardware counters that enable analysts to understand applications of performance and get to the bottlenecks quickly
 - Get counter data either using perf stat or PAPI APIs
 - We concluded with a comparison of compiler flags on open source compilers such as GCC, LLVM with IBM XL compilers

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