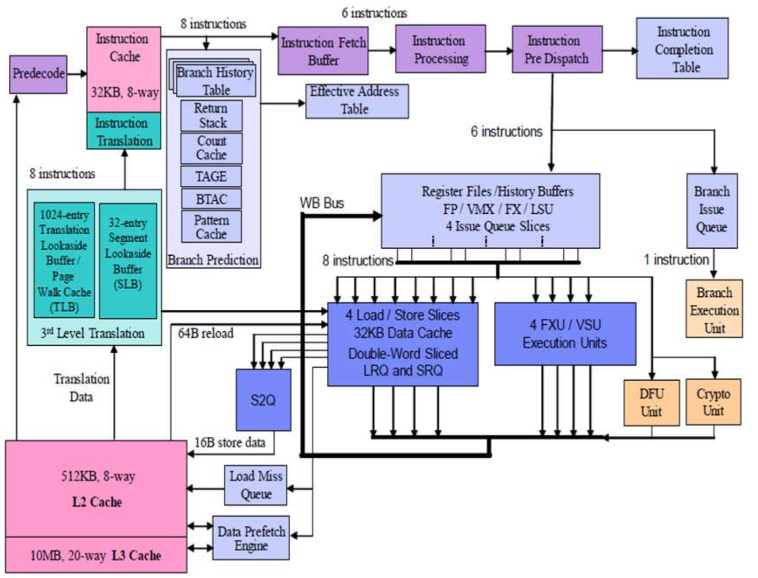
Getting the Most Bang for the Buck with Compilers on POWER

Archana Ravindar

Agenda

- Introduction to POWER9
- Tuning Strategies to get better performance using compiler flags, built-ins, pragmas on IBM XL, GCC, LLVM
- Optimization options chart for IBM XL, GCC and LLVM
- Compiler optimization remarks and their uses
  - Provides insights into the program and compiler
  - Understand what optimizations compiler could or could not do
  - How do we obtain optimization remarks on IBM XL, GCC and LLVM

Role of the Compiler



From POWER9 Processors User Manual

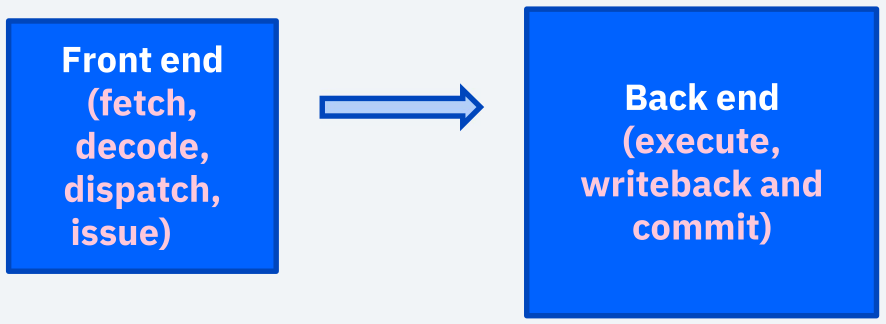
– POWER9 is a RISC architecture and has a pipeline of fetch-decode-dispatch-issue-execute-writeback-commit stages

– There are multiple load store units, vector-scalar and arithmetic units that can support up to 128 (64-SMT4) instructions per cycle

– The Compiler performs various functions to assist the processor in ensuring that the application runs most optimally

- Present an optimal sequence of instructions to the pipeline such that there are no bottlenecks
- Schedule instructions such that many of them can run in parallel
- Take advantage of multitude of functional units
- Exploit the new ISA

Processor View- Front and Backend


Front end (fetch, decode, dispatch, issue)
Back end (execute, writeback and commit)

- It helps to view the processor as constituted of two components- Front end and the Back end
- Front end ensures a smooth supply of instructions to be executed to the Backend
- The Backend is concerned with the execution of instructions
- The presence of branches poses a challenge to the front end as it needs to keep supplying the backend with a stream of instructions to be processed. When there is a branch, there is a choice as to where the instructions need to be fetched from
- Advanced Processors have Branch Predictors who predict with high accuracy which direction the branch is going to be taken and the instructions are fetched from that address and given for processing

Possible Opportunities of Improvement in Frontend

- Smoothen fetch stream by reducing branches occurring in loops and function calls
  - Loop branches- unrolling
  - Function branches – Inlining
- These strategies will also usually increase the field of play for the compiler to identify greater opportunities around duplicate computation of values, scheduling and several others

This can be done at a multitude of levels -

- Compiler flags- loop unrolling flags, inlining flags
- Pragmas and attributes

  # pragma unroll(N) just before the loop will unroll the loop

  __attribute__((always_inline)) in front of a function will

  inline the function
- Built-ins

  __builtin_expect(variable, value) will generate hints that will favour

  the path which is dictated by the variable having the value provided
- Source changes

  Converting control flow dependency into a data dependency using

  ?: operator will generate isel for branches; Useful in cases where branches are difficult to predict

Useful flags for improving performance in Frontend

Flag Kind	XL	GCC/LLVM	Can be realized in source	Benefit	Drawbacks
Unrolling	-qunroll	-funroll-loops -fno-unroll-loops(disable unrolling)	#pragma unroll(N)	Unrolls loops ; increases opportunities pertaining to scheduling for compiler	Increases register pressure
Inlining	-qinline=auto:level=N (1: least aggressive, 10: most aggressive)	-finline-functions	always_inline function attribute or manual inlining	increases opportunities for scheduling; Reduces branches and loads/stores	Increases register pressure; increases code size
isel instructions		-misel (to disable use –mno-isel)	In some cases we can rewrite source using ?: operator	generates isel instruction instead of branch; useful in cases when branches cannot be predicted easily	For easy to predict branches HW branch predictor itself can give optimal performance
Link time optimization(also improves performance in backend)	-qipo	-flto , -flto=thin		Enables interprocedural optimizations Further increases scope for inlining and other optimizations across functions even in different modules	Can increase overall compilation time

Possible Opportunities of Improvement in Backend

- Backend is concerned with executing of the instructions that were fetched and dispatched to the appropriate units
- Compiler automatically takes care of making sure dependent instructions are far from each other in its scheduling pass so that multiple instructions can execute and produce results in parallel
- Tuning performance in backend involves ensure processor resources are optimally used
- We will look at some ways we can help compiler in ensuring optimal performance in the backend

- Registers is the most important resource in a processor
- Using instructions that reduce reg usage - Vectorization / reducing pressure on GPRs/ ensuring more throughput,
- Making loops free of pointers and branches as much as possible to enable more automatic vectorization by the compiler
- Using instructions that perform multiple tasks together – such as add. which simulates the combined effect of add and cmp instructions, andc which simulates the combined effect of and and complement
- Caches - data layout optimizations that reduce footprint, using –fshort-enums
- Prefetching – hardware and software

Useful flags for improving performance in Backend

Flag Kind	XL	GCC/LLVM	Can be realized in source	Benefit	Drawbacks
Data footprint	-qenum=small	-fshort-enums		Uses only the minimally sized data type for the enum depending on its max declared value	May cause unalignment issues
ISA exploitation	-qarch=pwr9	-mcpu=power9	Using builtins new ISA instructions can be exploited in source by the user	Uses new instructions introduced to exploit newly designed hardware units in the architecture	Code compiled with this option will run on HW with architecture POWER9 and newer

Software prefetching	-qprefetch=aggressive
	-qprefetch =aggressive:dscr=<value>
		-fprefetch-loop-arrays(GCC)
		
		To disable –fno-prefetch-loop-arrays (GCC)
			XL: __dcbt(load prefetch)
			GCC:__builitin_prefetch(ptr, 0,..)
			XL:__dcbtst(store prefetch)
			GCC:__builtin_prefetch(ptr, 1,..)
				Can prefetch values into L3 so that cache misses are reduced	For the user to predict when to prefetch manually will be difficult hence its best to rely on the compiler to do it

Hardware prefetching	**ppc64_cpu –dscr=<0x value>** will set the DSCR(Data Stream Control Register) with 0x value which will determine the various parameters of prefetching setting – Depth, aggression , ramp etc Usually will require root privileges to set this; Default value is 0 and is shown to work well for most situations (prefetching on and set to default parameters) In order to disable prefetching **ppc64_cpu –dscr=0x1**				

Compiler Optimization Remarks

Compilers are usually equipped to produce a log while they are trying to optimize codes

Remarks can be generated by a compiler flag

Remarks convey the thought process of the compiler

Compiler Flags to generate optimization remarks

XL	GCC	LLVM
–qreport Generates optimization remarks in *.lst files Includes all optimizations that were done and those that could not be done including the reason why in the same report	-fopt-info-all=<filename> -fopt-info-missed=<filename> -fopt-info-all generates remarks on optimizations that were done -fopt-info-missed generates remarks on optimizations that could not be done including the reason GCC has the option to redirect remarks to a user specified filename	-Rpass=<regular expression> -Rpass-analysis=<regular expression> -Rpass-missed=<regular expression> Together -Rpass and –Rpass-analysis generate remarks for optimizations that could be done. Rpass-analysis provides more detailed explanation -Rpass-missed generates remarks for optimizations that could not be done including the reason in most cases Generates output to stdout but can be redirected to a file

Examples of Optimization Remarks

```
bubplain.cpp:
18: void bubble_sort( char * data, uint64_t n ) {
19:   int * d = (int *)data;

24:   for( uint64_t i=0; i < n; i++ )
25:    for( uint64_t j=0; j < n-i-1; j++ )
26:      if ( d[ j ] > d[ j + 1 ] ) {
27:          int t = d[ j ];
28:          d[ j ] = d[ j + 1 ];
29:          d[ j + 1 ] = t;
30:     }
32: }

34: int main() {
35:  char * data = (char*)malloc( MAX*8 );
37:  uint64_t s=0;

// repeatedly invoke bubble sort on a random array

38:  for( int  i=2; i <= MAX; i *= 2 ) {
39:    printf( "Size         : %d\n", i );
41:    for( uint64_t j=0; j < i; j++ )
42:      ((uint32_t*)data)[ j ] = rand();
43:      bubble_sort( data, i );

44:  }
45: }
```

```
1586-539 (I) Loop (loop index 7) at bubplain.cpp <line 41> was not SIMD
vectorized because it contains function calls.
```

(Remarks emitted by XL .lst file)

```
Analyzing loop at bubplain.cpp:41
bubplain.cpp:41:26: note: ===== analyze_loop_nest =====
bubplain.cpp:41:26: note: === vect_analyze_loop_form ===
bubplain.cpp:41:26: note: not vectorized: loop contains function calls or data
references that cannot be analyzed
```

(Remarks emitted by GCC)

```
bubplain.cpp:41:62: remark: loop not vectorized: call instruction cannot be
vectorized [-Rpass-analysis=loop-vectorize]
   for( uint64_t j=0; j < i; j++ ) ((uint32_t*)data)[ j ] = rand();
                                                                ^
```

(Remarks emitted by LLVM)

Examples of Optimization Remarks

```
bubplain.cpp:
18: void bubble_sort( char * data, uint64_t n ) {
19:   int * d = (int *)data;

24:   for( uint64_t i=0; i < n; i++ )
25:    for( uint64_t j=0; j < n-i-1; j++ )
26:      if ( d[ j ] > d[ j + 1 ] ) {
27:          int t = d[ j ];
28:          d[ j ] = d[ j + 1 ];
29:          d[ j + 1 ] = t;
30:     }
32: }

34: int main() {
35:  char * data = (char*)malloc( MAX*8 );
37:  uint64_t s=0;

// repeatedly invoke bubble sort on a random array

38:  for( int  i=2; i <= MAX; i *= 2 ) {
39:    printf( "Size         : %d\n", i );
41:    for( uint64_t j=0; j < i; j++ )
42:      ((uint32_t*)data)[ j ] = rand();
43:      bubble_sort( data, i );

44:  }
45: }
```

1586-552 (I) Loop (loop index 3) at bubplain.cpp <line 25> was not SIMD vectorized because it contains control flow.

(Remarks emitted by XL .lst file)

bubplain.cpp:24:24: note: === vect_analyze_loop_form ===

bubplain.cpp:24:24: note: not vectorized: control flow in loop.

bubplain.cpp:24:24: note: bad loop form.

(Remarks emitted by GCC)

bubplain.cpp:30:7: remark: loop not vectorized: control flow cannot be substituted for a select [-Rpass-analysis=loop-vectorize]

    }

(Remarks emitted by LLVM)

Examples of Optimization Remarks

In case of LLVM:

-Rpass=<regular expression>

-Rpass-analysis=<regular expression>

-Rpass-missed=<regular expression>

One can ask for any of the following categories of remarks **Loop-vectorize**(vectorization), **licm**(Loop invariant code motion), **inline**(inlining), **prologepilog**(stack prolog/epilog), **asm-printer**(count of instructions in assembly), **loop-unroll**(unrolling), **tailcallelem**(tail call elimination), **gvn**(global value numbering) * to be passed as a parameter for Rpass options

Or one can simply ask for all of them together by using a simple regular expression; The regular expression is defined in the usual sense; [a-z]* denotes all possible combination of words

**: Check LLVM documentation to find the latest supported categories*

Example:

```
bubplain.cpp:25:5: remark: unrolled loop by a factor of 4 with run-time trip count [-Rpass=loop-unroll]
   for( uint64_t j=0; j < n-i-1; j++ )

bubplain.cpp:45:5: remark: _Z15int_bubble_sortPcm inlined into main [-Rpass=inline]
   int_bubble_sort( data, i );
```

Other useful flags for improving performance

Flag Kind	XL	GCC/LLVM	Benefit	Drawbacks
Profile directed feedback	1. Compile using -qpdf1 to generate a training binary 2. Run training binary with training input 3. Compile using –qpdf2 to generate final binary	1. Compile using –fprofile-generate to generate a training binary 2. Run training binary with training input 3. (only for LLVM) : Run llvm-profdata merge default*.profraw -output default.profdata 4. Compile using –fprofile-use to generate the final binary	If your application has very specific hot paths the compiler will apply special optimizations to speed up the hot paths thereby improving performance	Requires an additional compilation and run step
Using special libraries to speedup operations	Link with –ltcmalloc (Tcmalloc libraries), -lmass (Math libraries)			

Summary

- Compiler is your powerful ally
- Today we saw ways and means in which we can use compilers on POWER to tune application performance
- We also saw how we can better understand the compiler thought process that can deepen insight about your application and the compiler itself

Useful References

https://openpowerfoundation.org/?resource_lib=power9-processor-users-manual

https://www.ibm.com/support/knowledgecenter/SSXVZZ_16.1.1/com.ibm.compilers.linux.doc/compiler.pdf

http://www.redbooks.ibm.com/redbooks/pdfs/sg248422.pdf

https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

https://www.gnu.org/software/gcc/projects/prefetch.html

https://gcc.gnu.org/onlinedocs/gccint/Guidelines-for-Diagnostics.html#Guidelines-for-Diagnostics

https://clang.llvm.org/docs/UsersManual.html

http://llvm.org/devmtg/2016-11/Slides/Nemet-Compiler-assistedPerformanceAnalysis.pdf

Thank you.

Archana Ravindar

Senior Engineer

LLVM Performance on POWER

—

aravind5@in.ibm.com

ibm.com

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