Coding Assembler for Performance

SHARE 107 Baltimore, MD
Session 8192 August 2006

David Bond
Tachyon Software LLC
dbond@tachyonsoft.com

Why code in assembler?

- High performance
- Access to OS services
- Your boss said so
- Job security

In this session, we will look at coding techniques that can improve performance, or at least not sabotage performance on modern processors. If you have been coding assembler for a long time, you may not realize how much the rules have changed.

Not your parent's mainframe

Many people still think of mainframes as they were in the 1960's:

- Some or all instructions were microcoded
- Memory access speed was constant
- Instruction timings were published
- Instruction speed was independent of adjacent instructions

On modern System z machines:

- Simple instructions are in hardware
- Complex instructions are millicoded
- Multiple processors and LPARs
- Processors are piplined and superscalar
- Each processor has a 2-level memory cache
- Instructions and data are cached separately

Processor and Memory Organization

Processors are wired together into MultiChip Modules. Each MCM has a bank of main memory and 40MB of Level-2 cache.

MCMs are also connected and processors on each MCM can access the L2 cache and main memory of the other MCMs at the cost of some speed.

Each processor has 256KB of Level-1 instruction cache (I-cache) and 256KB of Level-1 data cache (D-cache). Both L1 and L2 cache is organized into *cache lines* of 256 consecutive bytes on a 256-byte boundary. Each I-cache line is accessed as 16 quadwords. Each D-cache line is accessed as 32 doublewords.

Each 256 byte *line* of memory can be in multiple caches if not updated, but only in only one cache if updated.

Memory Access Times

Cache/Memory	Access Time
L1 cache	0
L2 cache (local)	20 ns
L2 cache (non-local)	50-100 ns
Local memory	150 ns
Non-local memory	200 ns
Paged-out storage	I/O speeds

z990 cycle time is 0.8 ns. z9-EC cycle time is 0.58 ns.

Note: Cache/memory access times from SHARE-105 session 2835.

Real Storage isn't Real

Virtual storage addresses must be translated to actual (absolute) memory addresses. Virtual storage addresses are translated to real storage addresses using the Region First Table, Region Second Table, Region Third Table, Segment Table and Page Table, but ...

The real address is only real to your operating system. At a minimum, your operating system is running in a LPAR. It may also be running under z/VM, so there are one or two more levels of translation before a real address can be translated to and absolute address.

Translation Costs

Virtual and real addresses must be translated to absolute addresses. The cost of translation is high, so recent translations are saved in a hierarchy of tables.

Page Absolute Address History Table
- PAAHT miss costs 2 cycles if subsequently found in TLB-1.

Translation Lookaside Buffer-1
- TLB-1 miss costs 7 cycles if subsequently found in TLB-2.

Translation Lookaside Buffer-2
- TLB-2 miss cost is variable and large.

Why data might not be where you want it

- Another processor or MCM has an updated copy in its L1 cache. Or the updated line is in D-cache and it is needed in I-cache. Must be written back to L2 cache and refetched.
- Your task already accessed the memory, but was then dispatched on another processor by the OS.
- Your OS (LPAR) was redispatched on another processor.
- Your cached data was flushed by another task or LPAR that was dispatched on your processor.

Virtual storage concepts of working set size and thrashing apply equally to cache.

Memory – What can you do?

- Separate updated memory areas from read-only memory areas.
  Remember, there can be multiple cached copies of read-only data but only one cached copy of updated data.
  This is automatic for reentrant code.

- Never mix updated data with instructions in the same 256 bytes.
  This is the cause of major performance problems when migrating code from older machines to System z machines.
  Updating a mixed line will flush the instructions from the I-cache, causing refetch of the instructions and restart of the pipeline.

- Separate data from instructions.
  Don't waste I-cache with data and D-cache with instructions.

There isn't much else you can do to control cache effects.

Alignment

- Align data with preferred boundaries.
  Halfword, Fullword and Doubleword sized data should be aligned to 2-byte, 4-byte or 8-byte boundaries and multi-byte areas should cross as few doubleword boundaries as possible.
  Remember, D-cache is accessed in doublewords.

- Keep small data areas that are accessed together in the same doubleword and cache line. To align on cache line, use:
  GETMAIN/STORAGE STARTBDY=8

- Align the start of instruction groups on doubleword or quadword boundaries, especially the start of heavily executed loops.
  Cache-line alignment can even be better.
  I-Cache is accessed in quadwords.

- STM and LM will execute faster if the storage operand is doubleword aligned and even/odd register pairs are referenced.

Processor Pipeline

Instructions can be “executing” at any of these stages:

- Fetch into I-cache
- Fetch into processor
- Decode and Branch Prediction
- Address Generation
- Grouping
- Operand fetch
- Execution
- Put Away

Optimally, the processor can execute two simple instructions per cycle. But, to keep the pipeline flowing, there must be no delays (or “stalls”). The biggest impact of any delay is from accessing memory that is not in the Level-1 cache. It can take over 100 cycles to fetch from Level-2 cache and even longer from main memory.

Branch Prediction

The processor must guess the direction and location of any branch to keep the pipeline filled with instructions. The processor knows that instructions like BCR 15,14 always branch and BASR 14,0 never does, but often it must simply guess. The most recent 8K successful branches are saved in the Branch Target Buffer. If a branch instruction is not found in the BTB, the processor guesses that the branch will not be taken.

For non-relative branches, the target location must also be saved. This assumes that the branch or base register has not changed. In future processors, relative branch instructions may be faster than RR or RX branch instructions.

Branch Prediction

The processor will prefetch instructions on the path that it guesses will not be taken, but those instructions will not be loaded into the pipeline. For this reason, “surprise” branches have a big impact on performance.

Recommendations:

- Use the relative branch instructions whenever possible.
- Normal logic flow should rarely branch except to loop.
- Successful branches should be used for exception logic.

```
LTR     R15,R15     Success?
JNZ     ERROR       No, branch
```

Address Generation Interlock

Address Generation Interlock happens when an operand address cannot be computed until a prior instruction that sets a needed register is executed. Because of the pipeline, register values are usually not available for address generation for up to 5 cycles, causing the pipeline to stall.

There are exceptions for LOAD and LOAD ADDRESS instructions.

Instruction	z900 delay	z990 and z9 delay
LOAD ADDRESS	2 cycles	3 cycles
LOAD	3 cycles	4 cycles
other	5 cycles	5 cycles

Use the LA instruction to increment addresses (and LAY to decrement addresses) instead of add or subtract instructions.

Address Generation Interlock Avoidance

Try to rearrange instructions so that there are 3 instructions between a LOAD ADDRESS instruction and an instruction that uses the register in an address. For LOAD, try to have 4 instructions between and for other instructions such as ADD, try to have 5 instructions between.

```
LA      R1,AREA(R15)
SR      R2,R2
SR      R3,R3
SR      R4,R4
MVC   0(4,R1),=F'1'
```

There will be a 3 cycle delay between the LA and MVC, so the SR instructions will execute “for free”.

Address Generation Interlock Avoidance

SS instructions such as MVC and CLC have two operands for which Address Generation is required. Each requires one cycle. Sometimes the operands can be swapped to reduce the AGI delay.

On a z9, there will be a 3 cycle delay between these instructions:

```
LA      R1,AREA(R15)
CLC     0(4,R1),=F'1'
```

but only a 2 cycle delay between these:

```
LA      R1,AREA(R15)
CLC     =F'1',0(R1)
```

Instruction Grouping

Up to 3 instructions can be grouped together for execution in a single cycle. If a branch instruction is included, it must be the first instruction in a group, otherwise 2 instructions can be grouped. Normally, the instructions in the group cannot depend on the results of any other instruction in the group. (Read After Write dependency)

These will not be grouped together:

```
AHI     R1,1
MHI     R1,8        RAW dependency on R1
```

These can be grouped together:

```
BZR     R14
LR      R1,R2
AHI     R1,1        Operand forwarding avoids RAW
```

Pipeline Aware Coding

For best performance, interleave instructions so that there are as few dependencies between nearby instructions as possible. When coding, intermix the next logical set of instructions with the ones you are working one. Set up addressability as far in advance of use as possible.

BUT: This can make programs hard to read and maintain when instructions are not in logical sequence. This is an area where pipeline-aware compilers such as GCC can produce faster code than hand-coded assembler because the compiler can interleave instructions without worrying about readability.

Analyze This!

```
        LA      R2,STRING
        LHI     R3,L'STRING
FIND1   CLI     0(R2),0          5 cycles per iteration
        JE      FOUND
        LA      R2,1(,R2)
        BRCT    R3,FIND1

        -vs-

        LA      R2,STRING
        LHI     R3,L'STRING
FIND2   CLI     0(R2),0          4 cycles per iteration
        LA      R2,1(,R2)
        JE      FOUND
        BRCT    R3,FIND2
```

Don't mix Code and Read/Write Data!

Nothing hurts the pipeline worse than mixing instructions and updated data areas in the same cache line. Each time the data is updated, the pipeline is halted, the D-Cache line is written to Level-2 cache, the I-Cache line is reloaded, and the pipeline is refilled. This causes instructions which normally execute in a cycle or two to require hundreds of cycles.

The normal form of most z/OS macros generate an in-line parameter list. If the parameter list must be modified, use a remote parameter list (MF=L) and reference it with the MF=(E,parmlist) form of the macro to avoid modifying the I-Cache line. This is required for reentrant coding anyway.

Use the RSECT instruction to tell the assembler to check for non-reentrant code in a control section.

What is Millicode?

Complex instructions are implemented in millicode on System z. Millicode routines are loaded into the HSA and are automatically invoked when a millicoded instruction is executed. IBM can change hardware instructions to be millicode instructions in order to correct an error in the hardware implementation.

Millicode routines are coded using normal non-millicoded z/Architecture instructions as well as some millicode-only instructions. There is a full set of general, access and control registers usable by millicode routines. Millicode-only instructions exist to transfer values between the z/Architecture registers and the millicode registers and to alter the PSW, including the condition code.

System z millicode is prepared using the Tachyon z/Assembler.

Why are some instructions implemented in Millicode?

Many instructions are too complex to reasonably implement in hardware.

Millicode is changeable, so errors can be corrected and new facilities can be added after the machine is installed.

Millicode routines cannot be interrupted, so atomic operations such as the PLO instruction are possible.

Millicode routines can access normally protected memory, so instructions such as BAKR and PR can execute in problem state.

Millicode routines can access special hardware facilities such as the cryptographic coprocessor.

Why care about Millicoded Instructions?

Unless special hardware facilities are used, millicoded instructions are necessarily slower than coding the same function using normal instructions. Extra cycles are needed to invoke and return from the millicode routine, to retrieve and set register values, to set the condition code, etc.

For example, MVCL, MVCLE, MVCP, MVCS and MVCK are all implemented in millicode using the MVC instruction. Except for the special case of moving whole 4K pages, these instructions can never be as fast at moving data than using MVC instructions. But, considerable I-cache storage can potentially be saved by using MVCL over coding hundreds of MVC loops. Use MVCL whenever it is the easiest instruction to code except in extremely performance-sensitive routines.

TR and TRT

The millicode routines behind the TR and TRT instructions:

- Performs a trial execution if the translation table crosses a page boundary (TR).
- Loads the translation table into I-Cache.
- Instructs the Translation Processor to perform the operation.
- Sets the condition code and result registers (TRT).

The overhead is large enough that these instructions are slower than the equivalent instructions for short operands even though the Translation Processor can process two bytes per cycle. This is especially true when the length is variable and set using the EX instruction.

Avoid using TRT to search for a single value. Instead, code the equivalent CLI loop or use the SRST instruction.

References

Even More of What You Do When You're a CPU?
SHARE 105 (Boston) session 2835 by Bob Rogers

The IBM eServer z990 microprocessor
IBM Journal of Research and Development Vol. 48 No. 3/4

Millicode in an IBM zSeries processor
IBM Journal of Research and Development Vol. 48 No. 3/4

The microarchitecture of the IBM eServer z900 processor
IBM Journal of Research and Development Vol. 46 No. 4/5

Tachyon z/Assembler:
http://www.tachyonsoft.com