PRACE Summer School: Taking the Most Out of Supercomputers
29.08. – 01.09. 2011

Low-level (assembler) serial code optimization
1 h lecture Wed. 15.15, 1 h lab Thu. 10.45

Jan Westerholm
professor, high performance computing
http://www.abo.fi/~jan.westerholm
Assembly: When and Why?

- A typical programming process proceeds from formulating a mathematical problem, designing solution approaches, choosing algorithms and data structures, implementing and debugging.
- When everything is already running smoothly and correctly we might be interested in speeding up program execution further without program redesign.
- Enter assembly code and serial assembly code optimization!
Assembly: When and Why?

- Assembly is the lowest level programming of a microprocessor available to the programmer.
- Assembly instructions are the only instructions a microprocessor can execute.
Assembly: When and Why?

- Very few people write programs directly in assembly!
- Instead we use high level programming and a compiler/interpreter, and we design our high level programs in such a way that our data layout and the program flow comply with the architecture of the microprocessor
  - cache optimization from data layout
  - instruction pipelining
  - avoiding unpredictable jump instructions
Assembly: When and Why?

- Reasons for **not writing** assembly programs:
  - hardware specific: assembly code is usually not portable (use conditional compilation)
  - assembly code is hard to maintain and change
  - compilers already do a very nice job in optimizing assembly code
  - compiler optimizations and microprocessor features tend to alleviate the effects of non-optimized code (out-of-order execution, branch prediction, micro-fused code)
Assembly: When and Why?

- Reasons for writing assembly programs:
  - if profiling indicates that most of the time is spent in a very small part of the code: this is a potential place for bespoke assembly code
  - if program execution time is important and we need to shorten it as much as possible. Speed ups from writing assembly programs can be between 1.1 and 2.0
  - if compilers do not fully utilize the assembly instruction set or the microprocessor hardware features (eg. 256-bit registers)
Microprocessor Architecture

- What do I need to know to write assembly code?
- Microprocessor architectures
  - hardware registers
  - assembly instructions (op-codes)
- Assembly guides for Intel architectures
  - Intel® 64 and IA-32 Architectures Optimization Reference Manual (Order Number: 248966-025 June 2011)
Intel 64-bit Register Architecture

**Basic Program Execution Registers**
- Sixteen 64-bit Registers

**General Purpose Registers**
- 2^64 - 1

**Segment Registers**
- Six 16-bit Registers

**Address Space**
- 2^64 - 1

**RFLAGS register**
- 64-bits

**RCP Instruction Pointer Register**
- 64-bits

**FPU Registers**
- Eight 80-bit Floating-point Data Registers
## Intel 64-bit Register Architecture

<table>
<thead>
<tr>
<th>Register Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control Register</td>
<td>16 bits</td>
</tr>
<tr>
<td>Status Register</td>
<td>16 bits</td>
</tr>
<tr>
<td>Tag Register</td>
<td>16 bits</td>
</tr>
<tr>
<td>OpCode Register (11-bits)</td>
<td></td>
</tr>
<tr>
<td>FPU Instruction Pointer Register</td>
<td>64 bits</td>
</tr>
<tr>
<td>FPU Data (Operand) Pointer Register</td>
<td>64 bits</td>
</tr>
</tbody>
</table>

### MMX Registers

- Eight 64-bit MMX Registers

### XMM Registers

- Sixteen 256-bit XMM Registers
# Intel 32-bit General Purpose Registers

<table>
<thead>
<tr>
<th>General-Purpose Registers</th>
<th>31</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
<th>16-bit</th>
<th>32-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AX</td>
<td>EAX</td>
</tr>
<tr>
<td>BH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BX</td>
<td>EBX</td>
</tr>
<tr>
<td>BL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CX</td>
<td>ECX</td>
</tr>
<tr>
<td>CH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DX</td>
<td>EDX</td>
</tr>
<tr>
<td>CL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Intel 64-bit General Purpose Registers

- 8 new general purpose registers R8-R15
- 8 new XMM registers: XMM8-XMM15
- REX: 64-bit code, 64-bit registers

<table>
<thead>
<tr>
<th>Register Type</th>
<th>Without REX</th>
<th>With REX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte Registers</td>
<td>AL, BL, CL, DL, AH, BH, CH, DH</td>
<td>AL, BL, CL, DL, DIL, SIL, BPL, SPL, R8L - R15L</td>
</tr>
<tr>
<td>Word Registers</td>
<td>AX, BX, CX, DX, DI, SI, BP, SP</td>
<td>AX, BX, CX, DX, DI, SI, BP, SP, R8W - R15W</td>
</tr>
<tr>
<td>Doubleword Registers</td>
<td>EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP</td>
<td>EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D - R15D</td>
</tr>
<tr>
<td>Quadword Registers</td>
<td>N.A.</td>
<td>RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8 - R15</td>
</tr>
</tbody>
</table>
Assembly: Where to Apply

- Given the source code
  - compile your code using reasonable optimization flags for the compiler
  - make careful time measurements
  - using a profiler, eg. *gprof*, locate execution hot spots, preferably with only a few lines of code
  - run a disassembler to see what the compiler has put together for this source code, eg. *objdump*
  - try to write faster and correct code using inline assembly!
Assembly: Sample Code

- Sample code for assembly optimization: POY
- POY calculates cladograms:
  - ancestral relations between organisms using DNA or protein (amino acid) sequence alignment.

![Cladogram diagram]

- beetles
- wasps, bees, ants
- butterflies, moths
- flies
Assembly: Sample Code POY

- Central algorithm: Needleman-Wunsch (Smith-Waterman) for DNA or protein sequence alignment.
- 72% of the total run time is spent in `alg_fill_row()`.

In `alg_fill_row`:

1. For integers `aa, bb, cc` calculate
   1. `min(aa, bb, cc)`
   2. a three bit pattern indicating which of `aa, bb` and `cc` obtain the minimum value.

Example: 3, 8, 2 gives `min = 2` and bit pattern `001`. Similarly, 4, 5, 4 gives `min = 4` and `101`. 
for (i = st; i <= end; i++) {
    aa = pm[i - 1] + alg_row[i]; bb = mm[i - 1] + gap_row[i]; cc = pm[i] + c;
    if (cc < aa) {
        if (cc < bb) { mm[i] = cc; dm[i] = 0x1;}  // min = cc
            else if (bb < cc) { mm[i] = bb; dm[i] = 0x2;}  // min = bb
            else { mm[i] = bb; dm[i] = 0x3;}  // min = bb,cc
    } else if (aa < cc) {
        if (aa < bb) { mm[i] = aa; dm[i] = 0x4;}  // min = aa
            else if (bb < aa) { mm[i] = bb; dm[i] = 0x2;}  // min = bb
            else { mm[i] = bb; dm[i] = 0x6;}  // min aa,bb
    } else { /* aa == cc */
        if (aa < bb) { mm[i] = aa; dm[i] = 0x5;}  // min = aa,cc
            else if (bb < aa) { mm[i] = bb; dm[i] = 0x2;}  // min = bb
            else { mm[i] = bb; dm[i] = 0x7;}  // min = aa,bb,cc
    }
}
Assembly Code

- Compile POY code with `gcc -O3 -c -g`.
- Disassemble with `objdump -d -S`.
- Result in PRACELowLevel_objdump.pdf.
- Examine disassembled code to see what it does.

`objdump` uses AT&T assembler syntax:

```
mov %src,%dest ; src is copied to dest
```

Intel syntax (used in Intel’s manuals):

```
mov dest,src ; src is copied to dest
```
Why might this code be a candidate for improvement?

- It contains many if-statements which “obstruct” a continuous flow of program execution, and the execution path in this program is more or less random
  - Any branch may be taken and branching is independent of the previous values calculated.

Can we rewrite the program with less or perhaps without if-statements

- in C?
- in assembler?
Assembly: Improvement Strategy

- Consider the following jumpy code:
  \[
  \text{if ( a > b ) z = a; else z = b;}
  \]
- It will translate roughly to

  ```
  cmp a, b
  jg labelone
  mov b, z
  jmp labeltwo
  labelone: mov a, z
  labeltwo: next op code
  ```
Assembly: Improvement Strategy

- In assembly there are instructions, cmovXX (conditional moves), which are sparingly used by compilers.
- The instructions perform a move if the condition XX is true: if the corresponding condition flag XX is set, the move is executed, otherwise the move is not done.
- The trick: cmovXX is always executed, and there is no jump in the program flow.
Assembly: Improvement Strategy

- In some simple cases the high level code can obviously be translated to `cmovXX`:
  - Consider $z = (a > b) ? a : b$;
  - The assembly code for this C-statement could be:
    
    \[
    \begin{align*}
    \text{mov} & \quad a, z \quad ;\text{assume } a > b \\
    \text{cmp} & \quad a, b \quad ;\text{compare } a \text{ and } b \\
    \text{cmovl} & \quad b, z \quad ;\text{if } a < b \text{ move } b \text{ to } z
    \end{align*}
    \]
Inline Assembly

- So far we have:
  - identified a piece of code amenable to optimization: `alg_n_fill_row` in POY
  - run objdump to see what the compiler has produced (few or no cmovXX’s)
  - presented an idea on how to achieve a speed up (using cmovXX’s)
- How do we insert our assembly code?
  - Answer: use *inline* assembly
Inline Assembly

- Basic template for gcc inline assembly:

```
__asm__ __volatile__ (  
    assembler program  
    : output operands /* optional */  
    : input operands /* optional */  
    : list of clobbered registers /* optional */  
);
```
Inline Assembly: Simple Case

```c
int main(void) {
    int foo = 10, bar = 15;

    __asm__ __volatile__(
        "add %2,%0\n\t" : "=a"(foo) : "a"(foo), "b"(bar);
        printf("foo+bar=%d\n", foo);
        return 0;
    }
```
Lab Assignment

- Given three integers, design and implement a subroutine in C that finds
  1. the minimum value of three integers
  2. a bit pattern xxx indicating which integers obtain the minimum value
- Compile and run objdump
Lab Assignment

- Design and implement a subroutine in assembly!
- Compile and run objdump
Extra Lab Assignment

- Design and implement a subroutine in C which finds the minimum value and the corresponding bit pattern with absolutely no if-statements!
- Hint: Comparisons are allowed!
Low Level Optimization: Assembly

- Final comments:
- General rule:
  When in doubt, don’t use assembly!
- If there is a small piece of code then profile the code to see if it is worth the effort to write it in assembly.
- Document profusely: you will forget how the program works in less than 6 months!
Lab Assignment

```c
__asm__ __volatile__(
  "cmp %2,%1\n\t"
  "mov $0x2,%3\n\t"
  "mov $0x0,%4\n\t"
  "cmovl %1,%2\n\t"
  "cmovl %4,%3\n\t"
  "setle %b4\n\t"
  "add %4,%3\n\t"
  "mov $0x0,%1\n\t"
  "add %3,%3\n\t"
  "cmp %2,%0\n\t"
  "cmovl %1,%3\n\t"
  "cmovl %0,%2\n\t"
  "setle %b1\n\t"
  "add %1,%3\n\t"
  : "+r" (aa), "+r" (bb), "+r" (cc), "+r" (dd), "+r" (ee) 
); //min value in cc, bit pattern in dd
```
inline void algn_fill_row (int *mm, const int *pm, const int *gap_row, \
const int *alg_row, char *dm, int c, int st, int end) {
    int i, index, tmpa[8];
    const int index2tmp[] = {0,1,2,1,3,1,2,1};
    const char DIA[] = {0x0, 0x1,0x2,0x3,0x4,0x5,0x6,0x7};
    for (i = st; i <= end; i++) {
        tmpa[1] = pm[i] + c;
        tmpa[2] = mm[i - 1] + gap_row[i];
        tmpa[3] = pm[i - 1] + alg_row[i];
        mm[i] = tmpa[index2tmp[index]];
        dm[i] = DIA[index];
    }
    return;}

Disassembly of section .text:

```
align_fill_row (int *mm, const int *pm, const int *gap_row, const int *alg_row, int c, int st, int end) {
    int i, tmp1, tmp2, tmp3;
    for (i = st; i <= end; i++) {
        if (tmp1 < tmp3) {
            if (tmp1 < tmp2) {
                mm[i] = tmp1;
                dm[i] = DELETE;
            } else if (tmp2 < tmp1) {
                mm[i] = tmp2;
                dm[i] = INSERT;
            } else {
                mm[i] = tmp2;
                dm[i] = (INSERT | DELETE);
            }
        } else if (tmp3 < tmp1) {
            if (tmp3 < tmp2) {
                mm[i] = tmp3;
                dm[i] = DELETE;
            } else {
                mm[i] = tmp2;
                dm[i] = (INSERT | DELETE);
            }
        } else if (tmp2 < tmp3) {
            if (tmp2 < tmp1) {
                mm[i] = tmp2;
                dm[i] = INSERT;
            } else if (tmp1 < tmp2) {
                mm[i] = tmp1;
                dm[i] = DELETE;
            } else {
                mm[i] = tmp2;
                dm[i] = (INSERT | DELETE);
            }
        } else if (tmp1 < tmp2) {
            if (tmp1 < tmp3) {
                mm[i] = tmp1;
                dm[i] = DELETE;
            } else if (tmp3 < tmp1) {
                mm[i] = tmp3;
                dm[i] = DELETE;
            } else {
                mm[i] = tmp2;
                dm[i] = (INSERT | DELETE);
            }
        }
    }
}
```
# Code Snippet

```c
    6d: 41 c6 04 10 02       movb   $0x2,(%r8,%rdx,1)
    72: 41 39 f5             cmp    %esi,%r13d
    75: 7d c1                jge    38 <algn_fill_row+0x38>

    else if (tmp2 < tmp3) {
        mm[i] = tmp2;
        dm[i] = (ALIGN | INSERT);
    }

    else { /* tmp3 == tmp1 */
        if (tmp3 < tmp2) {
            mm[i] = tmp3;
            dm[i] = (ALIGN | DELETE);
        }
        else if (tmp2 < tmp3) {
            mm[i] = tmp2;
            dm[i] = INSERT;
        }
        else { /* tmp2 == tmp1 */
            mm[i] = tmp2;
            dm[i] = (DELETE | INSERT | ALIGN);
        }
    }
}

    return;
}
```

---

```assembly
77: 5b                   pop    %rbx
78: 5d                   pop    %rbp
79: 41 5c                pop    %r12
7b: 41 5d                pop    %r13
7d: c3                   retq
```

---

```assembly
80: 7f 1e                jg     a0 <algn_fill_row+0xa0>
82: 44 89 14 0f          mov    %r10d,(%rdi,%rcx,1)
86: 41 c6 04 10 05       movb   $0x5,(%r8,%rdx,1)
8b: eb a3                jmp    30 <algn_fill_row+0x30>
8d: 66                   data16
8e: 66                   data16
90: 90                   nop
93: 7e 2b                jle    c0 <algn_fill_row+0xc0>
95: 89 04 0f             mov    %eax,(%rdi,%rcx,1)
98: 41 c6 04 10 06       movb   $0x6,(%r8,%rdx,1)
9d: eb 91                jmp    30 <algn_fill_row+0x30>
9f: 90                   nop
a0: 44 89 14 0f          mov    %r10d,(%rdi,%rcx,1)
a4: 41 c6 04 10 01       movb   $0x1,(%r8,%rdx,1)
a9: eb 85                jmp    30 <algn_fill_row+0x30>
ab: 66                   data16
ac: 66                   data16
ad: 90                   nop
ae: 66                   data16
af: 90                   nop
b0: 7c ee                jli    a0 <algn_fill_row+0xa0>
b2: 44 89 14 0f          mov    %r10d,(%rdi,%rcx,1)
b6: 41 c6 04 10 03       movb   $0x3,(%r8,%rdx,1)
bb: e9 70 ff ff ff       jmpq    30 <algn_fill_row+0x30>
c0: 7c de                jli    a0 <algn_fill_row+0xa0>
c2: 44 89 14 0f          mov    %r10d,(%rdi,%rcx,1)
c6: 42 c6 04 02 07       movb   $0x7,(%rdx,%r8,1)
cb: e9 60 ff ff ff       jmpq    30 <algn_fill_row+0x30>
```