

Lightning Talk

When smart programmers write bad programs

Steven Munroe
Linux Technology Center, IBM

Introduction

- While working customers to test and tune their code I have noticed some recurring examples of really bad (performing, or misleading) code. The interesting thing was that these programmers did not understand how their code could be the source of the problems they saw.

Some examples

- The “micro-benchmark”
- The “early exit”
- The “dollars and cents”
- The “over objectification”

The “micro-benchmark”

```
startt = getTimeOfDayMsec();

for(j=0;j<iterations;j++)
{
    result = log(value);
}
// gresult = result;

endt   = getTimeOfDayMsec();
deltat = endt - startt;
mill_sec = (double)deltat / 1000.0;

printf ("%d log(%f) %6.3f seconds\n",
        iterations, value, mill_sec);
```

Results

- For earlier versions of the GCC (4.1 – 4.3) there was little change from -O1 to -O2 or -O3
 - This is expected because most of the execution is in the library and the loops is just the driver.
- But starting with GCC-4.4 we see big jump in performance at -O2 and -O3?

```
gcc version 4.3.5
Log_t -O1
100000000 log(123.456000) 14.452 seconds
Log_t -O2
100000000 log(123.456000) 14.288 seconds
Log_t -O3
100000000 log(123.456000) 14.329 seconds
```

```
gcc version 4.4.4
Log_t -O1
100000000 log(123.456000) 14.630 seconds
Log_t -O2
100000000 log(123.456000) 0.501 seconds
Log_t -O3
100000000 log(123.456000) 0.476 seconds
```

What's going on?

- The performance gain is not real!
 - GCC detects the results are not used and the input values don't change so makes only one call to $\log(123.456)$.
 - So the measurement is only for the value compare (within the loop) and the loop overhead.
 - I am surprised that GCC did not simply generate the constant for $\log(123.456)$ and perhaps the in future it will (LTO and full program IPA)

The “temporary fix”

```
startt = getTimeOfDayMsec();

for(j=0;j<iterations;j++)
{
    result = log(value);
}
/* This seems to disable the one-time logic
But not sure for how long. */
gresult = result;

endt    = getTimeOfDayMsec();
deltat  = endt - startt;
mill_sec = (double)deltat / 1000.0;

printf ("%d log(%f) %6.3f seconds\n",
        iterations, value, mill_sec);
```

Why should we care?

- Somebody is going to come to the wrong conclusion based on this data.
 - It is easy to run a quick benchmark on your laptop (with the latest Ubuntu and GCC-4.4 compiler) against whatever Enterprise server (RHEL5.5 with GCC-4.1) and assume that the difference is meaningful
 - They do and they have!
 - And as GCC gets better, especially with LTO and IPA, this will only get more complicated.
- GCC should issue warnings for these cases!
 - *Warning function xyz, at line ### may only be executed once.*

The “early exit”

```
// Logger.H
class custLogger {
public:
    custLogger ();
    custLogger (const char *AppID);
    ~custLogger () {};
    void LogEnable ();
    void LogDisable ();
    void LogTimeStamp (int event, char *suffix);
    void LogDebug (char *format, ...);
protected:
    const char *IDstr;
    int logEnabled;
};
```

The “early exit”

```
// Logger.cpp

void custLogger::LogDebug (char *format, ...)
{
    if (!logEnabled) return;

    char buf [1024];
    va_list  arg;
    va_start (arg, format);
    vsnprintf(buf, 1024, format, arg);
    va_end (arg);

    fprintf (stderr, "%s-LOG: %s\n", IDstr, buf);
}
```

The “early exit”

```
// Logger.cpp cont
```

```
void custLogger::LogTimeStamp (int event, char *suffix)
{
    long result, msec;
    struct timeval time;
    int rc;

    if (!logEnabled) return;

    rc = gettimeofday (&time, NULL);
    if (rc == 0)
    {
        msec = (long) ((time.tv_usec + 500) / 1000);
        result = (long) (time.tv_sec);
        result = (result * 1000) + msec;
    }

    fprintf (stderr, "%s-LOGTime: ID=%d %ld.%3ld %s\n",
            IDstr, event, result, msec, suffix);
}
```

Results

- Seeing very high overhead even when ***logEnabled*** is false
 - The parameter list is always built in the calling code
 - Even for register rich architecture that can pass parameters efficiently This interferes with register allocation,
 - When varargs are use, parameters are spilled to the parameter save area even if they are not used
 - For the PowerPC ABI float/double values have to be copied into GPRs
 - Still executing the prologue/epilogue every time
 - Register rich architecture are impacted more then register poor architectures
 - If the Logger function is in a separate dynamic library then include PLT call stub overhead for every call.

-

The “temporary fix”

```
// Logger.H
class optLogger {
public:
    optLogger ();
    optLogger (const char *AppID);
    ~optLogger () {};
    void LogEnable ();
    void LogDisable ();
    inline void LogTimeStamp (int event, char *suffix) {
        if (logEnabled) {
            internalLogTimeStamp (event, suffix);
        }
    }
    inline void LogDebug (char *format, ...) {
        va_list ap;
        if (logEnabled) {
            va_start (ap, format);
            internalLogDebug (format, ap);
            va_end(ap);
        }
    }
protected:
    void internalLogDebug (char *format, va_list val);
    void internalLogTimeStamp (int event, char *suffix);
    const char *IDstr;
    int logEnabled;
};
```

The “temporary fix”

```
// Logger.cpp
void optLogger::internalLogDebug (char *format, va_list val)
{
    char buf [1024];
    vsnprintf(buf, 1024, format, val);

    fprintf (stderr, "%s-LOG: %s\n", IDstr, buf);
}

void optLogger::internalLogTimeStamp (int event, char *suffix)
{
    long result, msec;
    struct timeval time;
    int rc;

    rc = gettimeofday (&time, NULL);
    if (rc == 0)
    {
        msec = (long) ((time.tv_usec + 500) / 1000);
        result = (long) (time.tv_sec);
        result = (result * 1000) + msec;
    }

    fprintf (stderr, "%s-LOGTime: ID=%d %ld.%3ld %s\n",
            IDstr, event, result, msec, suffix);
}
```

Results after changes

- Better for fixed parameter case but still need to work on the varargs case.
 - The optimized version does eliminate the PLT stub overhead.
 - But inline member functions with varargs are not being inlined!
 - To call the not-inlined member function requires that parameters are spilled (and FPRs copied to GPRs) unconditionally
 - So forced to use macros in the final customer implementation
-

Static link of Logger

100000000 custLogger va_list 5.292
seconds

100000000 optLogger va_list 5.373 seconds

100000000 custLogger 4.339 seconds

100000000 optLogger 0.151 seconds

Dynamic link of Logger

100000000 custLogger va_list 7.834
seconds

100000000 optLogger va_list 5.359 seconds

100000000 custLogger 6.966 seconds

100000000 optLogger 0.151 seconds

Why should we care?

- Very common in Banking and Financial Markets (regulatory requirement).
- Exposes the need for Prologue/Epilogue shrink-wrap.
 - At least hoist the early exit “if” up into the prologue
 - Or version epilogues based on actual non-volatile spillage.
- Also need to understand why varargs forces inline members to no be inlined (at least for x86_64 and PPC32/64).

The “dollars and cents”

```
double d_Precision[] = {1, 10, 100, 1000, 10000};

bool D_EQUAL(double dVal1, double dVal2, int iPrecision)
{
    long long lVal1 = (long long )
        rint(dVal1 * d_Precision[iPrecision]);
    long long lVal2 = (long long )
        rint(dVal2 * d_Precision[iPrecision]);

    if (lVal1 == lVal2)
        return true;
    else
        return false;
}
```

What's going on?

- The rounded decimal compare is showing hot in the profile
 - Each compare requires 2 each:
 - External library call to rint()
 - round double to integer in current rounding mode, returning a double
 - Inline truncation from double to 64-bit integer (long long)
- Should avoid the library call, the double convert or both

The “fix” 1st attempt

```
bool D1_EQUAL(double dVal1, double dVal2, int iPrecision)
{
    long long lVal1 = (long long )
        (dVal1 * d_Precision[iPrecision] + 0.5);
    long long lVal2 = (long long )
        (dVal2 * d_Precision[iPrecision] + 0.5);

    if (lVal1 == lVal2)
        return true;
    else
        return false;
}
```

Results after changes

- Eliminated the library call but:
 - since the double to long long cast truncates need to explicitly round by adding a constant 0.5.
 - Ok on architectures that support FMA
 - This still requires a convert to integer operation
 - Which implies a FPR to GPR transfer
 - And may still require a library call on some architectures

10000000 D_round 3.189 seconds
10000000 DI_round 1.960 seconds

The “fix” 2nd attempt

```
bool Dfb_EQUAL(double dVal1, double dVal2, int iPrecision)
{
    double lVal1 = __builtin_round
                   (dVal1 * d_Precision[iPrecision]);
    double lVal2 = __builtin_round
                   (dVal2 * d_Precision[iPrecision]);

    return (lVal1 == lVal2);
}
```

Results after 2nd change

- Eliminated the library and the conversion:
 - Using the POSIX round() we get correct rounding without extra steps.
 - Using __builtin_round() generates a single instruction for many architectures
 - Using FPU operations throughout”
 - Avoids FPR to GPR transfers
 - Avoids extra round/truncat operations that can effect the result

```
10000000 D_round 3.189 seconds
10000000 DI_round 1.960 seconds
10000000 Dfb_round 1.237 seconds
```

Still not a good idea

- We are trying to convert a binary approximation of a decimal value, to an another binary approximation of a decimal value.
 - With two or more roundings (the multiple and the rint/round) we are exposed to rounding errors.
- In some countries using binary floating point for financial transactions is illegal

The better “dollars and cents”

```
_Decimal64 d_Precision[] = {1.0DD, 10.0DD, 100.0DD};

bool D_EQUAL(double dVal1, double dVal2, int iPrecision)
{
    _Decimal64 lVal1 = quantized64(dVal1,
d_Precision[iPrecision]);
    _Decimal64 lVal2 = quantized64(dVal1,
d_Precision[iPrecision]);

    if (lVal1 == lVal2)
        return true;
    else
        return false;
}
```

“over objectification”

- C++ with its pension for small member member functions is already a difficult problem for the compilers.
 - Mind sets about information hiding using “accessor function” makes it worse
 - When putting these classes in a (share object) library is really bad
- In a shared object every public function is exported by default
 - And SystemV linkage rules require even intra-library call to those functions to go through the PLT
 - Unless explicitly overridden with `-Bsymbolic`

“over objectification” the bad

- This the compiler has few opportunities for optimization
 - Unless the user explicitly inlines the small member functions
 - Bsymbolic is a linker option, tool late for the compiler
- LTO and whole program IPA is heading the right direction, but:
 - Creating one big statically linked application may not be acceptable
 - And inlining can create versioning problems as C++ libraries upgrade and applications don't.
- Net:
 - need to think about getting larger blocks of C++ code to optimize
 - While addressing the shared library version issues.