# High-Performance Tracing with C++11

Travis Gockel
SolidFire, Inc.
travis@solidfire.com

Marshall McMullen
SolidFire, Inc.
marshall@solidfire.com

**Abstract**

*Standard logging statements have a negative impact on operation, making it more difficult to debug high-performance multithreaded code. This paper gives an overview of a high-performance system for logging values, while maintaining the convenience of ad-hoc logging statements. SolidFire's system exploits a number of C++11 features, most notably variadic templates. This system allows SolidFire engineers to debug* any *function without having to worry about the impact on the system.*

## I. Problem

Let us start with an assumption: No software is written perfectly. At some point, a problem will occur and you will be left asking: What went wrong with the system? Few application developers will debate the benefit of log files for the purposes of diagnosis and debugging. But what if your application is so high-performance that introducing additional textual log statements to the system yields unacceptable performance degradation, or worse, triggers a change in behavior? One could choose to forego logging in certain areas and trust that the application will never break in those places, but hope is not a solid building block for reliable software.

### I.1 Standard Logging

At SolidFire, the most high-performance components of the system are also the most critical to the system's correctness. The ideal latency for reading a block of data is one millisecond (which includes network accesses); this leaves very little room for overhead. The most demanding logging area of the system is called the "LBA Trace" which, under full load, can end up logging a million messages each second. With an average log statement output of 217.2 bytes, logging alone could demand 200 MB/s, which is slightly lower than the sustained sequential write of a standard solid-state drive (as of the time of writing). Given that the SolidFire application needs to do other things with the drive bandwidth besides writing logs, standard text logging will not work.

Curiously enough, in an all-SSD system, writing the logs to the drive is *not* the long pole. When encoding only basic integer types (`std::uint64_t`), 85% of the time is spent simply converting the number to a string. This number only increases with the complexity of the encoding, `boost::format` being an egregious offender. In our experience, the speed of a solid-state drive is rarely a bottleneck.

The first solution to the encoding problem was to defer it and put "stringification" into another thread. Many known types were placed into a `boost::variant` and the unknown types were covered by a `boost::function<void (std::ostream&)>`. The `boost::variant`s were copied onto a queue so that a dedicated thread could stringify and write them out to the drive.

```
boost::variant<char,
               bool,
               int16_t,
               int32_t,
               uint8_t,
               uint16_t,
               uint32_t,
               uint64_t,
               unsigned long long int,
               long long int,
               float,
               double,
               const char *,
               std::string,
               boost::function<void(std::ostream&)>
              >
```

This solution decreased the overhead of logging and worked fairly well for a period of time. As the system grew, the log message output grew exponentially, which made the queue of log messages to process grow unbounded, leading to memory issues. To prevent the system from running out of memory, we had to implement a system for blocking when the queue was too full. That solution was unacceptable, as any arbitrary logging statement could lead to program stalls. Ultimately, queuing things with a `boost::variant` could not keep up with the amount of information we wanted to throw at it.

## I.2 Goals

**Minimal overhead.** Should be able to *sustain* a million statements per second – queuing is fine to handle bursts, but we must be able to get back to near empty queues. Beyond that, overhead in the system should not be placed on the thread issuing the statements to log, which helps avoid Heisenbugs (errors that, when observed, go away).

**No compile-time static strings in the output.** Static strings are the same throughout the lifetime of the application, so repeatedly writing them to the output is needless overhead. To reduce the bandwidth needed for logging, a solution should minimize or eliminate repeated chunks of statically-known data.

**Use `operator<<`.** All statements in the system use a convenient free-form `TRACE(area)<< ...` syntax, the C++ way. An ideal solution should look and feel exactly like standard text logging, yet be an order of magnitude faster.

# II. Proof of Concept

## II.1 Demonstration Prototype

After furious debate on how best to implement a system for high-performance logging, a prototype was written [1]. The look and feel should be familiar to anyone who has written to `std::cout`.

```
int main()
{
    TRACE(Area1) << "Hello, " << "world!";
    TRACE(Area2) << "I like to write " << 5 << " numbers: "
                 << 1 << 2 << 3 << 4 << 5;
    int x = 4;
    float f = 3.1415;
    const char blah[] = "Yo!";
    TRACE(Area1) << "x=" << x << " f=" << f << " blah=" << blah;
    TRACE(Really just a char*)
                 << "but not so in the real implementation"
                 << "...sound good?";

    // Time passes
    LoadBuffer(BinaryTracer::storage.data(),
               BinaryTracer::storage.size()
              );
}
```

The program output looks like:

```
Area1 -- logbuffers.cpp:151]] Hello, world!
Area2 -- logbuffers.cpp:152]] I like to write 5 numbers: 12345
Area1 -- logbuffers.cpp:157]]  x=4 f=3.1415 blah=Yo!
Really just a char* -- logbuffers.cpp:158]] but not so in the real
   implementation...sound good?
```

## II.2 Explanation

Tracing relies on being strongly-typed from the point of capture until we stringize the output. Functions are statically-typed, all information about the types passed to it *must* be known. Functions exist at different addresses, those addresses are sufficient to encode *all* necessary type information for decoding the `std::tuple<T...>`. Moreover, because template functions are generated at compilation time, the address will remain the same between runs [2]. These static assumptions allow the system to write out the compact binary format to the drive and post-process it at a later time (in a different process).

[1] The full source code for the prototype is available on GitHub.

[2] This limits you to compilers that do *not* do function address randomization. Check your compiler's settings!

### II.2.1 `Trace`: Encoding the Values

The `Trace` function is responsible for "encoding" provided variables into a small chunk of memory. It ties the variable payload to the static data by prefixing the payload portion of the data with a function pointer to a template function which understands the static portion. In the prototype, this is accomplished with an `std::tuple` and relies on `std::make_tuple` to decay the provided arguments to their values.

```
0 template <typename... T>
1 void Trace(const T&... x)
2 {
3     auto vals = std::make_tuple(x...);
4     size_t (*fn)(void*) = &Print<decltype(vals)>;
5     char buffer[sizeof fn + sizeof vals];
6     std::memcpy(buffer, &fn, sizeof fn);
7     std::memcpy(buffer + sizeof fn, &vals, sizeof vals);
8     BinaryTracer::Write(buffer, sizeof buffer);
9 }
```

Line 4 is the most subtly interesting line of the `Trace` function. Grabbing the address of a function with `&Print<decltype(vals)>` counts as "use," so the compiler emits code for a `template <std::tuple<T...>&> size_t Print(void* input)`.

The call to `Trace(0xdeadbeef, "cow", 0x1234feedbabeUL)` would get encoded to this [3]:

```
0 1 2 3  4 5 6 7  8 9 0 1  1 2 3 4  5 6 7 8  9 0 1 2  3 4 5 6
&Print<tuple<i... efbeadde &"cow"..... bebaedfe 34120000
```

There are two important things to take away from the output of `Trace`:

- Values are simply `memcpy`-ed – no translation
- Type information is preserved in the function pointer (first 8 bytes).

The `Print` function is covered in more detail in the post-processing section – the important thing is that information about the types is stored into the template parameters of the function pointer, which is static. The key optimization detail is that we do not have to write out the types of each parameter, which is a great deal of overhead.

The static string works because the address of the string, does not change when the `Print` function is called later. This is a decent amount of compression – a string of arbitrary length will always take up `sizeof(char*)` bytes of output. Multiple strings on the same line will independently take up `sizeof(char*)` bytes, even though the message for the line is static. This problem is dealt with in the real implementation.

### II.2.2 `Print`: Human-Readable Output

The output of the tracing logger is post-processed into plain-text human readable trace message by simply calling the saved function pointer with the encoded data. Creating human-readable text is

[3] Thank x86 for the little-endianness

accomplished with the `Print` function. It prints out all the values in the `TTuple` with `PrintTuple` and returns the size of data read from the buffer, which is used to know where the next piece of data is.

```
template <typename TTuple>
size_t Print(void* input)
{
    TTuple* p = reinterpret_cast<TTuple*>(input);
    PrintTuple(*p);
    return sizeof *p;
}
```

Ultimately, the post-processor can be described in a simple loop:

```
void LoadBuffer(char* buffer, size_t sz)
{
    for (size_t pos = 0; pos < sz; )
    {
        size_t (*fn)(void*);
        std::memcpy(&fn, buffer + pos, sizeof fn);
        pos += sizeof fn;
        pos += fn(buffer + pos);
    }
}
```

#### II.2.3 Variable Capture

To keep the familiar feeling of `operator <<` while maintaining the need for statically tracking the involved types, the system overloads the `operator <<` in a different way. It all starts with the `TRACE` macro:

```
TraceEntryContainer<> TraceFor(TraceArea area)
{
    return TraceEntryContainer<>(area);
}

#define TRACE_PREFIX (__FILE__ ":" STRINGVAL(__LINE__) "]]␣")

#define TRACE(area)                                                 \
    TraceEntryPusher(),                                             \
        TraceFor(#area) << TRACE_PREFIX
```

This starts us with an empty `TraceEntryContainer` and puts the standard trace prefix string after it. The next step is to use `operator<<` to capture the values for the trace.

```
template <typename T, typename... TCameFrom>
TraceEntryContainer<T, TCameFrom...>
```

```
operator <<(const TraceEntryContainer<TCameFrom...>& cameFrom,
            const T& x)
{
   return TraceEntryContainer<T, TCameFrom...>(x, cameFrom);
}
```

`TraceEntryContainer` is a heterogeneous recursive data structure which captures a single value `T` by const reference and a link back to where it came from (think `std::tuple<T...>` meets singly-linked list). Each application of `operator<<` returns another container (g++ is good at optimizing out these temporaries). Finally, `TraceEntryPusher`'s `operator,` collects the captured references and calls the `Trace` function to encode.

**Why** `operator,`**?** `operator,` is the lowest precedence with left-to-right associativity, which means that it will only see the final `TraceEntryContainer`'s type. An alternative approach would be to have the application of `operator<<` invalidate the previous container and rely on the final `TraceEntryContainer`'s destructor to invoke `Trace`. However, g++ does not understand that all but one of the `TraceEntryContainers` will not do anything on destruction, which leads to massive code bloat.

## III. Real Implementation

The real implementation of binary tracing required a number of changes to make it fit into our build system and for a few added features. The most important added feature was making all static formatting information available with only the function pointer. Another was the ability to extend the encoding system for arbitrary types, such as `std::vector` and `std::string`.

### III.1 Code Generation

The first step in the process is to parse all the source code for uses of the `TRACE` macro and extract the strings from them. As a build step, we generate a file called `TraceFormats.cpp` with an `awk` script run over the post-processed source. The file is a mapping of a `TraceUniqueType` to static `boost::format` strings, encoded as a fully-specialized template function named `TraceFormatFor`. The `TraceUniqueType` is a template class over two values: the file hash and line number. The file hash is unique to each translation unit[4] and the line number is the line number the `TRACE` statement appears on. This provides a way to associate an entry in the trace file to its format string [5]. The Makefile passes in `FILEHASH` to each translation unit as a macro.

```
template <size_t FileHash, size_t LineNumber>
struct TraceUniqueType
{
    // "Safety" -- an ad-hoc defined contract
    static void IsTraceUniqueType() { }
};
```

[4] it is generated from the MD5 sum of the CPP file name

[5] ...and restricts `TRACE` statements to CPP files.

For example, the trace statement on line 312 of the file "FakeFile.cpp":

```
TRACE(Foo) << "Writing␣to␣lba=" << lba
           << "␣key=" << key
           << "␣length=" << dataLength;
```

Would generate the code:

```
template <>
const boost::format&
TraceFormatFor<TraceUniqueType<0x683a62df909de747, 312>>
{
   static boost::format instance("FakeFile.cpp:312|Writing␣to␣
      lba=%1%␣key=%2%␣length=%3%");
   return instance;
}

template
const boost::format&
TraceFormatFor<TraceUniqueType<0x683a62df909de747, 312>>();
```

## III.2 Encoding

The most major change from a coding standpoint is that all free functions in the prototype are now inside of `template <typename UniqueType> struct TraceControl`, since the `Trace` and `Print` operations work on a specific `TraceUniqueType`. Another change is wrapping all values in a `TraceDecoratedValue`, whose use should become apparent in the next section. The smallest change is making the printing function take an `std::ostream` so that we can output to more than `std::cout`: `size_t (*TracePrinterFunc)(std::ostream&, const char*)`.

```
template <typename T, typename... TRest>
static void TraceStripEncode(char* buffer,
                             size_t bufferLen,
                             size_t idx,
                             const TraceDecoratedValue<T>& x,
                             const TraceDecoratedValue<TRest>&...
                                rest
                            )
{
    x.Encode(buffer + idx);
    TraceStripEncode(buffer, bufferLen, idx + x.GetSize(), rest...);
}

static void TraceStripEncode(char*, size_t bufferLen, size_t idx)
{
    ASSERT(bufferLen == idx);
```

```
}

template <typename TTracer, typename... T>
static void TraceImpl(TTracer& tracer,
                      const TraceDecoratedValue<T>&... x
                     )
{
    size_t bufferSize = TraceDecoratedValueSizeSum(x...);
    std::vector<char> buffer(bufferSize);
    TraceStripEncode(buffer.data(),
                     buffer.size(),
                     0,
                     x...
                    );
    tracer.WriteBuffer(buffer.data(), buffer.size());
}

template <typename TTracer, typename... T>
static void Trace(TTracer& tracer, const T&... x)
{
    // "safety"
    UniqueType::IsTraceUniqueType();
    TracePrinterFunc fn = &Print<T...>;
    TraceImpl(tracer, TraceDecorate(fn), TraceDecorate(x)...);
}
```

### III.2.1 `TraceDecoratedValue`

The responsibility of `TraceDecoratedValue` is to encode single values into a buffer and to extract a single value from a buffer. One can define their own `TraceDecoratedValue` through partial or full template specialization of the type.

```
template <typename T>
struct TraceDecoratedValue
{
    /*
    static_assert(SomeProperty<T>::value,
                  "Good luck safely defining SomeProperty"
                 );
    */

    explicit TraceDecoratedValue(const T& ref) :
            mRef(ref)
    { }
```

```
    size_t GetSize() const
    {
        return sizeof(T);
    }

    void Encode(char* buffer) const
    {
        memcpy(buffer, &mRef, sizeof(T));
    }

    static constexpr bool kHasDecode = true;

    static const T& Decode(const char* buffer, OUT size_t& size)
    {
        size = sizeof(T);
        return *reinterpret_cast<const T*>(buffer);
    }

    static size_t SizeOf(const T&)
    {
        return sizeof(T);
    }

private:
    const T& mRef;
};
```

This technique is quite powerful. The rule "do not encode static strings" is defined through partial template specialization:

```
template <size_t N>
struct TraceDecoratedValue<char[N]>
{
    explicit TraceDecoratedValue(const char (&)[N])
    { }

    size_t GetSize() const
    {
        return 0;
    }

    void Encode(char*) const
    { }

    static const bool kHasDecode = false;
};
```

This also supports dynamically-sized containers such as `std::vector`:

```
template <typename T>
struct TraceDecoratedValue<std::vector<T>>
{
    typedef std::vector<T>  VectorType;
    typedef uint16_t        SizeType;

    explicit TraceDecoratedValue(const std::vector<T>& vec) :
            mVector(vec)
    { }

    size_t GetSize() const
    {
        return sizeof(SizeType) + sizeof(T) * mVector.size();
    }

    void Encode(char* buffer) const
    {
        ASSERT(mVector.size() <
           std::numeric_limits<SizeType>::max());
        uint16_t size = static_cast<SizeType>(mVector.size());
        memcpy(buffer, &size, sizeof(SizeType));
        memcpy(buffer + sizeof(SizeType),
               mVector.data(),
               sizeof(T) * mVector.size()
              );
    }

    static const bool kHasDecode = true;

    static VectorType Decode(const char* buffer, OUT size_t& size)
    {
        uint16_t numElements;
        memcpy(&numElements, buffer, sizeof(SizeType));

        vector<T> vec(numElements);
        memcpy(&vec[0],
               buffer + sizeof(SizeType),
               sizeof(T) * numElements
              );
        size = sizeof(SizeType) + sizeof(T) * numElements;
        return vec;
    }
```

```
    const vector<T> & mVector;
};
```

### III.3 Buffering and Flushing to Disk

All encoded trace writes are placed into a buffer in memory. When the current buffer is filled, it is pushed onto a queue to write to the drive. We will never attempt to straddle a write across two buffers – the remainder of the buffer is all zero. By changing the buffer size, we can optimize writes for the output medium.

```
void Trace::WriteBuffer(const char* buffer, size_t sz)
{
    ASSERT(sz <= kBufferSize);

    SpinGuard ax(mCurrentBufferMutex);

    if (mCurrentBufferOffset + sz > kBufferSize)
        RollNextBuffer();

    std::memcpy(mCurrentBuffer + mCurrentBufferOffset, buffer, sz);
    mCurrentBufferOffset += sz;
}

void Trace::RollNextBuffer()
{
    mBuffersToWrite.push(std::move(mCurrentBuffer));
    mCurrentBuffer.reset(CreateAlignedBuffer(kBufferSize));
    mCurrentBufferOffset = 0;
    ++mNumBuffers;
}

void Trace::Run()
{
    std::unique_ptr<char[]> buffer;

    while (mBuffersToWrite.pop(buffer))
    {
        WriteToFile(buffer);
        buffer.reset();
        ++mNumWritten;
    }
}
```

**Spin Locking in `WriteBuffer`.** One could imagine an implementation of `Trace::WriteBuffer` which uses the Lamport bakery algorithm to reserve space instead of relying on a mutex for synchronization. The original implementation was done this way, but it was slower than the locking implementation. The code around rolling to the next buffer was more awkward and, perhaps more importantly, the code to ensure that buffers are output to the queue in the same order required extra space, which prevents allocating exactly 1 aligned page. Spin-locking on x86 is incredibly optimized and the contention on the mutex is fairly low, since the `memcpy` operation is cheap. Even an optimistic `atomic_fetch_add` does not improve performance, since the `mCurrentBufferMutex` and `mCurrentBufferOffset` reside on a single cache line.

## III.4 Post-Processing

The `Print` function changed significantly to support `TraceDecoratedValue` and `TraceUniqueType`:

```
template <typename T>
static size_t DecodeSingle(boost::format& fmt,
                           const char* ptr,
                           const std::true_type& hasDecode
                          )
{
    size_t sz;
    const T& val = TraceDecoratedValue<T>::Decode(ptr, OUT sz);
    fmt = fmt % val;
    return sz;
}

template <typename T>
static size_t DecodeSingle(boost::format& fmt,
                           const char* ptr,
                           const std::false_type& hasDecode
                          )
{
    return 0;
}

template <typename T>
static size_t PrintImpl(boost::format& fmt,
                        const char* ptr
                       )
{
    typedef TraceDecoratedValue<T> DecoratedType;
    return DecodeSingle<T>(fmt,
                           ptr,
                           std::integral_constant<bool,
                               DecoratedType::kHasDecode>()
                           );
```

```
}

template <typename T, typename TNext, typename... TRest>
static size_t PrintImpl(boost::format& fmt, const char* ptr)
{
    size_t sz = PrintImpl<T>(fmt, ptr);
    return sz + PrintImpl<TNext, TRest...>(fmt, ptr + sz);
}

template <typename... T>
static size_t Print(std::ostream& stream, const char* buffer)
{
    boost::format format = TraceFormatFor<UniqueType>();
    size_t sz = PrintImpl<T...>(format, buffer);
    stream << format.str() << std::endl;

    return sz;
}

static size_t PrintUnknown(std::ostream& stream,
                           const char* buffer
                          )
{
    TracePrinterFunc fn;
    std::memcpy(&fn, buffer, sizeof fn);
    return sizeof fn + fn(stream, buffer + sizeof fn);
}
```

The only change to the decoding loop is understanding that encoded values will not straddle a buffer. So if `offset + sizeof(PrintFunction)> bufferSize`, it should skip on to the next buffer.

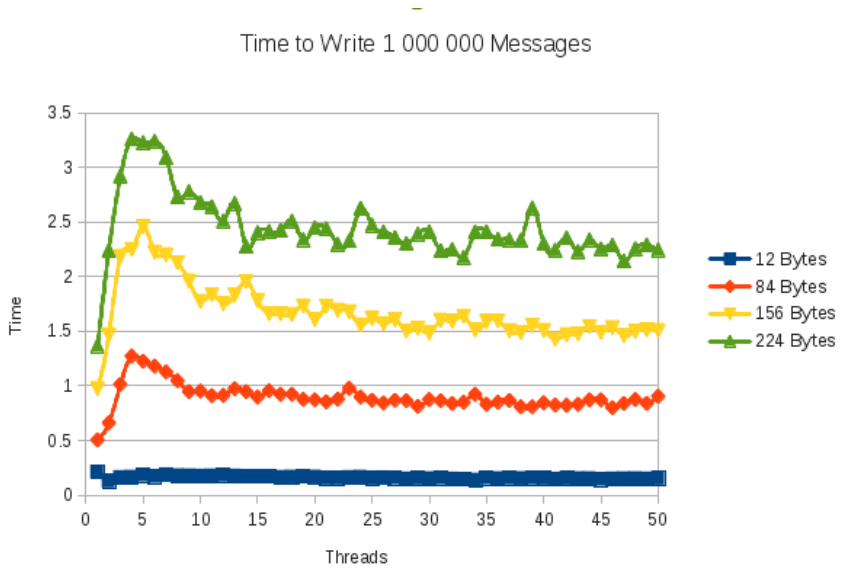
### III.4.1 Multithreaded Post-Processing

Interestingly, the decoding of chunks is completely independent. Since a write will never straddle a buffer, each `bufferSize` chunk can be processed independently. The implementation uses `tbb::pipeline` to decode chunks in parallel and then print them out in the file order.

## IV. Performance

The most significant performance gain is not in how fast we can spit out log messages to the drive, but in how little overhead there is for the application. Memory use is minimal as the binary tracer only uses enough to copy the variable data and to put it into a queue. CPU use is also very minimal since `memcpy` is a well-optimized function. This also scales well for multiple threads, as the highly volatile shared data consists only of a spin lock and an offset value, both of which fit

into a single cache line. Lowering the amount of data we write to the drive (in comparison to logging the text) is an obvious performance benefit, but the write pattern of directly writing to page-aligned blocks from page-aligned memory buffers means the writes themselves are more efficient.




The above graph shows the time it takes to write a million messages of various sizes with $N$ threads along the X axis [6]. The single-threaded "zero-contention" is the fastest, as there were no other processes running on the machine, so the Linux kernel's scheduler had nothing to do. After the scheduler starts context-switching, there is a fairly steady decline until we hit 12 threads, which is the number of cores the profiling machine has, after which the time evens out.

The rate at which we can log messages is highly correlated with the size of the message we are trying to log. We can sustain writing messages at just under 100 $MB/s$, so the number of messages we can write depends on the size of the average payload. If the average payload is 29.47 bytes, we can log over 3 million messages every second.

Performance "hammer" tests are somewhat meaningless, since the actual application does things besides logging. With binary tracing enabled, it is rare to see any impact on overall system performance.

[6] The number of messages each thread writes is $1000000/N$ so the total message count is always 1000000.

# V. Conclusion

## V.1 Limitations

There is no such thing as a free lunch and SolidFire's binary tracing is no exception. For the massive gains in efficiency, there are significant trade-offs in terms of safety and ease of use.

**The safety is off on your types.** Briefly alluded to in the definition of `TraceDecoratedValue` by a commented-out `static_assert` is the fact that the safety is off on your types. We could not find an existing type trait or create a combination thereof in order to define `SomeProperty`. Things like `std::is_trivially_copyable` and `std::is_standard_layout` have false positives, since a type consisting of a private `int[4]` is safe to log; and they have false negatives, since `int*` is not safe. Since `SomeProperty` is not defined, the general `TraceDecoratedValue` will happily accept traces of types which cannot be deserialized properly, when it would be ideal to not compile such code. Even worse than allowing the code to compile is the way the issue manifests: When post-processing, the program will segmentation fault or worse.

**The safety is off on binary compatibility.** Since we are using function pointers as data, the binary which generated the data must be the *exact* same as the one decoding it. We attempt to enforce safety using file system extended attributes. On output file creation, we add the MD5 sum as an `xaddr` and throw up boatloads of warnings if someone attempts to decode using a binary with a different MD5 sum. This forces people to use tools which preserve these attributes (`cp -a` and `rsync -X`) when copying.

**Difficult to get events as they occur.** With regular text-based logging with `syslog`, one can very easily `tail -f` a log file to see the messages as they are generated. Even if you output to a named pipe, binary tracing *requires* output buffering, which means there will be blocks of time when nothing is happening while the in-application buffer fills up.

**Specializing `operator <<` is painful at best.** Consider `class Socket`, which wraps an `int` file descriptor with some user-friendly construction, destruction and methods:

```
class Socket
{
public:
    explicit Socket(IPAddress addr);

    ~Socket();

    bool IsConnected() const;

    IPAddress GetBoundIP() const;

    IPAddress GetRemoteIP() const;
```

```
private:
    int mFd;
};
```

Socket's operator << is defined as:

```
std::ostream& operator<<(std::ostream& stream, const Socket& sock)
{
    if (!sock.IsConnected())
        return stream << "Not␣Connected";
    else
        return stream << sock.GetBoundIP()
                      << "␣->␣" << sock.GetRemoteIP();
}
```

So how does one overload operator << to work on a TraceEntryContainer? To start with, the overload is rather awkward to write out:

```
template <typename... TSrc>
TraceEntryContainer<IPAddress, IPAddress, bool, TSrc...>
operator <<(TraceEntryContainer<TSrc...>&& src, const Socket& x)
{
    return src << x.IsConnected() << x.GetBoundIP() <<
       x.GetRemoteIP();
}
```

There is not a convenient way to write ad-hoc char[]s inside of the definition of operator <<. Furthermore, there is not a convenient way to not return the two IPAddresses when the Socket is not connected, since C++ does not statically support types based on values. This can all be worked around with specializations of TraceDecoratedValue, but it is markedly more annoying than overloading the operator << free function.

Another solution is to add a function to convert a given type to an std::string and call that function when tracing it.

```
TRACE(Area) << "Something␣happened␣on␣socket␣" << ToString(socket);
```

There are two disadvantages to this solution. Encoding the value into an std::string throws away many of the advantages of binary tracing. Another disadvantage is that it pushes additional burdon onto the users of Socket. While certainly not a permanent solution, it is a great temporary workaround.

**No good way to TRACE in a header file.** The template parameters of TraceUniqueType are filled with the MD5 sum of the current translation unit and the line number the statement appears on. This leads to three curious consequences. First, format strings for TRACE statements written in header files will be written multiple times in the TraceFormats.cpp file, since the preprocessor will see them multiple times. Second, the output for a TRACE statement will appear to come from a line of *one* of the translation units which included that header, as the linker will merge

the definitions of the `inline` function the `TRACE` appears in into a single definition in the final binary. Third, if a `TRACE` statement is on line $N$ of a header file, no translation unit which includes that file can have a `TRACE` statement on its own line $N$, since they will share the same template parameters for their `TraceUniqueType` and cause our preprocessor to emit the same specialization of `TraceFormatFor`.

## V.2 Use at SolidFire

Binary tracing has been immensely helpful at SolidFire and fills two roles, one it was designed for and one which was an accidental consequence. The first is what it was designed for: Binary tracing enables us to monitor sections of code where normal logging would give unacceptably poor performance. The fact that the output of binary tracing is significantly smaller than its full-text counterpart enables the second use: Logging over extended periods of time.

### V.2.1 Use Case: Inside the RPC System

As a distributed storage system, network performance is super critical to the overall system. Each service can process hundreds of thousands of requests per second, all while trying to keep latency below 1 millisecond. If remote service latencies are too high for too long, we treat them as failed services. Any introduction of latency to the system which puts packets on the socket will be extremely detrimental to the point that a cluster will be unusable.

This was the perfect use case for binary tracing. With regular logging, any message meant a tiny performance hit, so we were extremely limited in where we could output, which meant only logging major events. After a major refactor, we ran into problems which only occurred under very high loads and did not happen in debug builds and seemed to disappear when adding even minimal logging – a Heisenbug. By adding binary trace points along the network communication path, we were able to quickly diagnose the issue.

### V.2.2 Use Case: Long Running Block Tracking

In testing the SolidFire application, errors which occur on the data path are extremely difficult to track down. Issues are often difficult to reproduce, as they require a number of interacting components to fail with the exact right timing. Recently written pieces of data are in cache, so a block might be correct for a period of time and then disappear as it leaves the cache some time later. This means monitoring the system for an extended period of time. Without knowing which piece of data might go missing, everything must be logged.

Out of this need, the "LBA trace" was born. This trace point enables us to track when every single piece of data in the system moves or thinks about moving to any other place in the system. Even with the tightly compacted output, it is reasonable for this trace point to write out a gigabyte of data each minute. Attempting to use text-based logging for this information is out of the question due to the enormous bandwidth requirements.

## V.3 Enabling Technology

Our binary tracing system has enabled us to track information we could not have considered before. We spend less time attempting to recreate problems and more time analyzing the data from a test

run. Furthermore, we can monitor sections of the code which are extremely performance-sensitive without worrying about the impact. While it took some getting used to dealing with the trade-offs, binary tracing has been a boon to productivity.