The Performance of FoxNet 2.0

Herb Derby

June 1999

CMU-CS-99-137

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

The goal of the `FoxNet` project is to explore the advantages (and drawbacks) of implementing networking software using an extended version of the `SML` language. In this report, we document the performance of the `FoxNet`. Using the performance results as a guide, we compare small function overhead and memory access performance of the extended `SML` to the dominate systems programming language `C`.

This research was supported in part by the Office of Naval Research and in part by the Defense Advanced Research Projects Agency, monitored by the Office of Naval Research under Contract N00014-84-K-0415, ARPA Order No. 5404.

Introduction

Researchers have made great progress in the design and implementation of programming languages in the past twenty years. However, most systems software, and in particular most networking software, is still written in `C` and `C++`. The `FoxNet` is an experiment to create an Internet networking stack using an extended version of Standard ML (`SML`) [7]. The protocols and services provided by the `FoxNet` are defined by the Request For Comments numbers 791 and 1122 [5, 8]. The extended version of `SML`, which the `FoxNet` uses, is defined by the development team of the Standard ML of New Jersey compiler [2]. For this report, we call this extended language `SML+` (leaving the term `SML` for the language defined in [7]) and we call the compiler which compiles the `SML+` language the `SML/NJ` compiler.

The goal of the `FoxNet` project is to explore the advantages (and drawbacks) of implementing networking software using `SML+` and to understand how to extend the `SML` language to accommodate systems software. Our `FoxNet` experience has shown us that `SML` provides excellent high-level support for producing systems software, but it lacks low-level support for producing efficient implementations. At the architectural level, the `SML` signatures provide concrete specifications for defining and discussing the modules of a large system. At the process level, the compilation manager of `SML/NJ` eliminates the burden of building and maintaining makefiles. At the implementation level, the type system catches programming mistakes which commonly occur when using `C` and automatic memory management eliminates a whole class of memory-allocation bugs that plague `C` and `C++`. Despite these strengths, the low-level implementation support that the `SML+` language provides has two weaknesses:

- *Small functions*: `SML+` does not provide a mechanism for efficiently implementing very small functions — functions smaller than 20 instructions. The `C` language provides the `#define` macro mechanism for efficiently using very small functions. The `C++` language gives a programmer additional control over inlining with the inline declaration and with templates.

- *Foreign memory*: `SML+` provides a mechanism for reading and writing raw memory from outside the heap managed by the `SML/NJ` system. The `FoxNet` was able to use this foreign memory system mechanism to interact with the operating system, but the foreign memory system is unacceptably slow for creating efficient networking systems.

In this report, we compare a networking stack written in `SML+` (using the term `FoxNet`) with the networking stack from Digital Unix (using the term `UNIX`), which is written in `C`. We show that the two implementations have similar throughput over Ethernet, but the `FoxNet` consumes over 10 times the CPU resources. We continue by profiling the `FoxNet` and using the results to find the major consumer of CPU resources: the checksum routine. We use the checksum routine to illustrate the two disadvantages of the `SML+` language when implementing system software. Finally, we draw our conclusions.

Performance Measurement

The performance measurements use two identical Digital Equipment Corporation Alpha 300/266 computers each with 96 Mb of memory running Digital Unix 4.0. We perform the tests with the machines started in single-user mode, and the network interfaces activated manually using the `ifconfig` command. The `FoxNet` tests are compiled using version 110.5 of the Standard ML of New Jersey compiler. In order gain access to the network, we extended the `SML/NJ` runtime system with an interface to the Digital Unix packet filter and the FORE ATM application interface. These extensions allow the `FoxNet` to access the Ethernet and ATM networks directly from a user process running the `SML/NJ` system. We use `NetPerf` version 2.1 patch level 2 to measure the performance of the networking stack from Digital Unix. `NetPerf` is a commonly used networking performance testing program maintained at Hewlett Packard [4]. For the remainder of the report, the unit MB stands for 10^6 bytes, and the unit Mb stands for 10^6 bits.

Ethernet 10 Mb/s

The performance measurements over Ethernet use the built in Ethernet adaptor of the Alphas and the machines are connected to each other using a single strand of cable forming an isolated 10 Mb/s network.

TCP Throughput

The `FoxNet` uses a test similar to the `NetPerf` TCP Stream Performance test to measure throughput. The TCP window sizes for both `UNIX` and the `FoxNet` are set to 65,536 bytes for these measurements. We measured three different payload sizes: 1 byte, 1,000 bytes and 1,000,000 bytes. In Figure 1, we show the results of the measurements for the throughput of both the `FoxNet` and `UNIX` handling different size payloads.

	Throughput in Mb/s		
Size	`FoxNet`	`UNIX`	Ratio
1 MB	6.92	6.89	1
1 KB	.50	6.88	.007
1 Byte	.0006	.79	.00007

Figure 1: Ethernet throughput measurements. The first column indicates the size of the payload. The second column is the throughput for the foxnet in Mb/s and the third column is the throughput for `UNIX` in Mb/s. The fourth column is the ratio of the `FoxNet` to `UNIX`.

TCP Latency

The `FoxNet` uses a test similar to the `NetPerf` TCP Request–Response test to measure latency. A packet with a 1 byte payload is sent to a server which responds with a packet with a 1 byte payload. The test counts the number of responses per second. In Figure 2, we show the results of the latency measurements.

Responses/s		
`FoxNet`	`UNIX`	Ratio
257	2343	.11

Figure 2: Ethernet request–response measurement. The first two columns show the responses per second for the `FoxNet` and `UNIX` respectively. The third column is the ratio of the `FoxNet` to `UNIX`.

ATM 155 Mb/s

The hardware set of the ATM performance measurements consists of a FORE ForeRunner PCA200E ATM adaptor installed in each machine and a ForeRunner ASX200WA ATM switch connecting the machines together forming an isolated network. The segment size for the network is set to 65,536 bytes.

Test	Throughput in Mb/s	Ratio
`UNIX`	79.15	—
`FoxNet`	18.4	.23
`FoxNet` no checksum	47	.59

Figure 3: ATM Throughput. This is the 1 MB test run using an ATM network instead of an Ethernet network. The first column is the system being measured. The second column is the throughput and the third column is the ratio of the `FoxNet` to `UNIX`. We include the measurement of the `FoxNet` without the checksum routine to demonstrate how much time the checksum routine consumes.

CPU Utilization and Layer Timing

The 1 Mb/s throughput test case gives us the opportunity to compare the CPU utilization of both systems sustaining a throughput of 6.9 Mb/s. The `NetPerf` program has a feature to measure the CPU utilization of the `UNIX` networking stack running the throughput test, but its accuracy is dependent on the behavior of the operating system. In the throughput test, the CPU utilization measurement uses a tight loop — called a *soaker* — to consume any CPU cycles left over by the `C` networking stack. We start the workstation in single-user mode to eliminate as many processes as possible. While the networking test is executing, the soaker loop runs in its own process at a very low priority waiting for the test to block. When the network stack blocks, the soaker gets its chance to use the CPU until the networking test is ready to run again. It executes its loop counting the number of iterations. The accuracy of the utilization measurement depends upon the policy of the scheduler allowing the soaker to run only when the networking test is blocked. Unfortunately, Digital Unix suspends the networking test and gives the soaker processor cycles to try to balance the performance of the entire system. Therefore, we will only use the measurements for a rough comparison.

In order to measure the CPU utilization of the `FoxNet`, we extended the `SML/NJ` runtime system with the exact code `NetPerf` uses to measure CPU utilization.

	UNIX	FoxNet	Ratio
CPU Usage	8%	88%	11

Figure 4: CPU utilization measurement. Both UNIX and the FoxNet are measured using the 10 MB Ethernet throughput test. The second column shows the utilization for UNIX and the third column shows the utilization for the FoxNet. The fourth column is the ratio of the FoxNet to UNIX CPU utilization. The FoxNet requires 11 times the resources of the CPU for a 6.9 Mb/s throughput.

Our measurements show that the UNIX system consumes about 8% of the CPU while the FoxNet consumes about 88%. To understand why UNIX is 11 times more efficient than the FoxNet, we use the Alpha cycle counter [10] to measure the time it takes a packet to pass through each layer of the FoxNet. We accumulate this measurement information at each layer and calculate the average number of cycles it takes to process a packet at each layer in the network stack.

From our experience working with the FoxNet, we know that the checksum routine consumes a large portion of the packet processing time. Therefore, we measured two different stacks, one with the TCP checksum calculation and one without. We show, in Figure 5, the average layer timings of the two stacks.

Layer	with checksum (cycles per layer)	without checksum (cycles per layer)
TCP	239,742	93,440
IP	47,448	48,236
Ethernet	2,458	2,500
Device	34,072	30,244
	Total 323,740	174,420

Figure 5: Profile of the layers of the FoxNet TCP stack. The second column shows the average cycle count for processing a packet at each layer. The third column shows the same measurement as the second column but with the checksum calculation removed from the TCP layer. The fluctuations of the measurements between the second and third columns for the Device, Ethernet and IP layers are caused by interrupts and other effects of a multitasking system. The ratio of the without-checksum total to the with-checksum total is 54%. Therefore, the checksum accounts for 46% of the packet processing time.

This test demonstrates that most of the time is spent in the TCP layer and that the TCP checksum calculation takes 46% of the packet processing time. The TCP checksum is calculated using part of the TCP header and the entire TCP payload. It treats these bytes as if they are an array of 16-bit words and calculates a 16-bit answer by summing the array using 16-bit one's complement arithmetic. The running time of the checksum routine is dominated by its inner loop. Therefore, this will be our focus for the remainder of the report.

Small Functions

Most systems software contains a few small functions, which have calling overhead that takes more time than executing the function itself. An example of this situation is the C Standard I/O (`stdio`) library [9]. The `stdio` library contains a type called `FILE`, which the library implementation uses to store the state of the internal buffering system. The `fgetc` is a small function, which extracts the next character from a `FILE` buffer. The calling overhead of `fgetc` is unacceptable for time critical code. A simple way to remove the overhead is to copy the body of the function to the function call site; this method of removing overhead is called manual inlining and has several problems:

- The function body for `fgetc` may not be available.
- The programmer has to understand the invariants of the `FILE` type and how it behaves with respect to the rest of the `stdio` library.
- Any maintenance programming performed on the `stdio` library must include changes to all the places that `fgetc` is manually inlined.

The `stdio` library solves this problem by introducing a `#define` macro called `getc`, which has the same behavior as `fgetc`. Using `getc`, the macro system automates the inlining of the `fgetc` body at the function call site. By using the macros system, the calling overhead has been eliminated, the internal workings of the `stdio` library are protected and there is a single point to make all maintenance changes.

In the `FoxNet`, the buffering mechanism is implemented as an abstract data type called a `Word_Array`. The `Word_Array` provides functions for extracting and manipulating the buffers in different word formats. These include 32-bit and 16-bit size words in big-endian or little-endian format. We will show that the small functions of the `Word_Array` add significant overhead to the checksum routine. We can measure the overhead by simply inlining the function `checkOneEntry` into the `Word_Array` function, `fold`. We measure the performance of the two routines by calculating the checksum of a buffer with a length of 2^{22} bytes. In Figure 6, we show the inner loop of the `FoxNet` checksum routine.

```
fun checkOneEntry(new, accumulator) =
  Word32.+ (
    Word32.+ (
      Word32.>> (new, 0w16),
      Word32.andb (new, 0wxffff),
    accumulator)
fun checksum buffer = Word_Array.W32Little.fold checkOne 0w0 buffer
```

Figure 6: The inner loop of the `FoxNet` checksum. The notations `Word32.+`, `Word32.>>` and `Word32.andb` are 32-bit two's complement addition, logical right shift and bitwise `and` operation respectively. These functions are documented in the `SML/NJ` web pages [1]. This routine takes 72.1 billion cycles to calculate checksum on the test buffer.

In a `Word_Array`, a `buffer` is a triple `(byteBuffer, first, last)` where `byteBuffer` is an array of bytes, `first` points to the first 32-bit word in the buffer and `last` points to the last 32-bit

```
fun fold f b (byteBuffer, first, last) =
  let
    fun loop (index, accumulator) =
      if index > last then accumulator
      else
        loop (index + 1, f (Pack32Little.subArr(byteBuffer, index), value))
  in
    loop (first, b)
  end
```

Figure 7: The `fold` function from the `FoxNet`. We discuss the `Pack32Little.subArr` in the Foreign Memory section.

word in the buffer. The function `Word_Array.W32Little.fold` interprets the bytes in the buffer as an array of 32-bit words and has the following definition

$$\texttt{fold}\ f\ b\ [w_1, w_2, \ldots, w_n]\ \text{returns}\ f(w_1, f(w_2, \ldots, f(w_n, b)\ldots))$$

where the w_i's are 32-bit words in little-endian format. We display the code for the `FoxNet` `Word_Array.W32Little.fold` function in Figure 7. We show the results of substituting the body of the `checkOne` function for the variable `f` of the function `fold` in Figure 8.

The original version of the `FoxNet` checksum routine, from Figure 6, calculates the checksum in 72 billion cycles while the inlined checksum routine does the calculation in 45 billion cycles. Therefore, the function `fold` introduces an overhead of 60%. From a systems software perspective, a mechanism for removing the overhead of small functions gives you more freedom in dividing a system into modules.

Foreign Memory

The `SML/NJ` runtime system manages memory structures in its own fashion in its own heap. If `SML` is to be used as a systems programming language, it must be able to access and manipulate memory structures from other languages. The `SML+` language has added several structures for doing this.

		100				104				108	
⋯	1D	AB	78	E3	44	67	03	B1	F1	23	⋯

Figure 9: Memory. In this figure, we show memory as bytes starting at location `100`.

The core of the `SML+` foreign memory system is the `Word8Array` structure [1]. This structure implements arrays of bytes (8-bit words) of fixed length. The `SML/NJ` runtime system can send and receive data by using arrays of bytes, implemented using `Word8Arrays`, as parameters to the underlying operating system calls. A program can use the functions of the `Word8Array` to manipulate the bytes of a buffer. Assuming the memory configuration in Figure 9 and a `Word8Array` starting

```
fun inlineChecksum (byteBuffer, first, last) =
   let
      fun loop (index, accumulator) =
        if index > last then accumulator
        else
           loop (
             index+1,
             let
               val new = Pack32Little.subArr(byteBuffer, index)
             in
               Word32.+ (
                 Word32.+ (
                   Word32.>> (new, 0w16),
                   Word32.andb (new, max32)),
                 accumulator)
             end
   in
     loop(first, 0w0)
   end
```

Figure 8: The checksum routine with the `CheckOne` routine manually inlined into the `fold` routine. This routine takes 45.2 billion cycles to calculate the checksum on the test buffer.

at location `100` called `buffer`, the function `Word8Array.sub(buffer,1)` will return the value `0x78` located at address `0x101`.

The `SML+` language has structures for accessing `Word8Arrays` in different formants and word lengths: `Pack32Little`, `Pack32Big`, `Pack16Little` and `Pack16Big`. For example, the structure `Pack32Little` provides functions for manipulating `Word8Arrays` as arrays of 32-bit values in little-endian format. Assuming the memory configuration from Figure 9, the call `subArr(buffer,0)` from the `Pack32Little` structure would return the 32-bit value, $\texttt{44} * 2^{24} + \texttt{E3} * 2^{16} + \texttt{78} * 2^{8} + \texttt{AB}$, and the `Pack32Little` call `subArr(buffer,1)` would return the value $\texttt{F1} * 2^{24} + \texttt{B1} * 2^{16} + \texttt{03} * 2^{8} + \texttt{67}$.

These structures provide the basic mechanisms for interacting with memory structures from outside the `SML/NJ` system. However, their design clashes with the alignment restrictions imposed by modern processors. All CPUs provide single instructions for accessing 32-bit and 16-bit words (on the Alpha, the instructions `ldl` and `stl` load and store 32-bit-little-endian values), but valid addresses are restricted to multiples of four for the 32-bit instructions and multiples of two for the 16-bit instructions. If these restrictions are not met, the CPU generates an alignment exception.

The `SML/NJ` compiler could create very efficient code if it used the multiple byte access instructions. Unfortunately, the `SML/NJ` runtime places no alignment restrictions on the addresses of `Word8Arrays`, making these fast access instructions unsafe for implementing functions like `subArr` from the `Pack32Little` structure. Therefore, the 32-bit and 16-bit values must be assembled out of several 8-bit values obtained using single byte loads.

We can measure the overhead of assembling 32-bit words from four individual bytes by per-forming the following experiment. We calculate the number of cycles the checksum routine uses to access memory by creating a copy of the checksum routine with the memory accesses removed. The difference of the running times of these two routines is amount of time spent accessing memory. We can do the same measurement with the checksum routines translated into `C` in order to calculate the number of cycles that `C` uses to access memory as 32-bit words. In Figure 10, we show the code for the checksum routine translated to `C`. In Figure 11 and Figure 12, we show the lines which we changed, to remove the memory accesses.

System	Original	No Access	Difference
`SML+`	45.2	5.6	39.6
`C`	8.3	1.0	7.3
			Ratio: 5.4

Figure 13: Foreign memory overhead. The second column shows the number of cycles that both the `C` implementation and the `SML+` implementation take to calculate the checksum on the test case buffer. The third column shows the number of cycles that both implementations take to calculate the checksum without reading memory. The forth column shows the number of cycles spent accessing memory for both implementations. The ratio is the overhead imposed by the `SML/NJ` implementation of `Pack32Little.subArr`.

We summarize the results of the experiment in Figure 13. As you can see, the foreign memory handling primitives of `SML/NJ` are 5.4 times slower than the memory accesses of `C`. The `C` checksum routine generates a single `ldl` instruction to read the 32-bit word. The `SML/NJ`compiler generates several instructions to read four bytes and concatenates them together to form a 32-bit word.

Conclusions

Matching the throughput performance of a `UNIX` system for large block sizes is a significant milestone for the `FoxNet`. Unfortunately, the `FoxNet` takes 11 times the CPU resources to obtain this performance. We have identified that the checksum calculation consumes 47% of the CPU time.

Studying the checksum routine leads us to the conclusion that there are two significant impediments to using `SML/NJ` as an efficient systems programming language. First, accessing memory from outside the `SML/NJ` system is unacceptably slow. In fact, accessing 32-bit words using `SML/NJ` is 5.4 times slower than using `C`. The following are two possible alternatives for solving the foreign memory problem:

- The current implementation of the `Word32Little.subArr` reads four individual bytes and concatenates them together forming a 32-bit word. The compiler could generate the single instruction for reading the 32-bit word. This code would execute quickly, but could cause an alignment exception. The `SML/NJ` runtime system could handle the exception and simulate the instruction.

- Instead of basing the the foreign memory system on `Word8Arrays`, `SML+` could base it on `Word32Arrays`. This new system would introduce a set of coercions from `Word32Arrays`

```
unsigned int
inlineChecksum(byteBuffer, first, last) {
  int index = first;
  unsigned int accumulator = 0;
   while (index <= last ) {
    unsigned int w32 = byteBuffer[index];
    index += 1;
    sum += ( (w32>>16) + (w32&0xFFFF) );
  }
   return accumulator;
}
```

Figure 10: C Checksum routine. The inlineChecksum routine from Figure 8 translated into C. This routine takes 8.3 billion cycles to calculate the checksum on the 2^{24} byte test buffer.

```
val dummy = ref 0
fun inlineChecksum (byteBuffer, first, last) =
        ...
             let
               val new = !dummy
             in
        ...
```

Figure 11: SML+ inlineChecksum without memory accesses. This figure shows just the portions of the inlineChecksum routine which we changed to remove memory accesses. This routine takes 5.6 billion cycles to calculate the checksum on the 2^{24} byte test buffer.

```
unsigned int dummy = 0;
unsigned int
inlineChecksum(byteBuffer, first, last) {
    ...
    unsigned int w32 = dummy;
    ...
}
```

Figure 12: The routine from Figure 10 with the memory references removed. The access to the buffer (byteBuffer[index]) is replaced with reading a variable dummy. We have declared the variable dummy as external to keep the compiler from optimizing the loop away. This routine takes 1.0 billion cycles to calculate the checksum on the 2^{24} byte test buffer.

to Word16Arrays and from Word32Arrays to Word8Arrays. This would allow access to the different word sizes while still obeying the alignment restrictions. Disallowing the reverse coercion would eliminate any unsafe code.

The second impediment to using SML+ as a systems programming language is its inability to eliminate the function call overhead for small functions. This makes it difficult to abstract low level mechanisms, such as buffer handling, into their own modules. Consequently, the programming teams are forced to manually inline the small functions throughout the system making maintenance difficult. Several research projects have addressed this problem. The CAML system from Inria has included a macro system in their build environment. Matthias Blume is working on automatic inlining for the SML/NJ system [3]. Finally, the research in the field of staged computations, such as the work on lightweight runtime-code generation by Leone and Lee [6] and the work on modal ML by Wickline, Lee, Pfenning and Davies [11], can be used to eliminate small function overhead.

In industry, performance is not the only measure of quality for systems software. Using SML+ for programming the FoxNet gives it many reliability benefits over C and C++. Also, we have found that the SML module system and higher-order functions make good building blocks for expressing systems software elegantly and directly. In this report, we have identified two impediments, which when removed could greatly improve the efficiency of using SML+ for systems software. It is no longer a question of whether SML+ can be used to create systems software, but a question of making it efficient enough for low-level programming.

Acknowledgments

I would like to thank Robert Harper and Peter Lee for leading the Fox project and for providing feedback on this report. I would like to thank Ken Cline, Elmootazbellah Elnozahy, Nick Haines, Greg Morrisett, Brian Milnes, and Eliot Moss for their assistance and contributions to the Foxnet project. I would especially like to thank Edoardo Biagioni for his support and timely reading and re-reading of this report.

Bibliography

[1] Andrew W. Appel, Nick Barnes, Dave Berry, et al. SML'97 basis library. http://cm.bell-labs.com/cm/cs/what/smlnj/doc/basis/index.html.

[2] Andrew W. Appel, Nick Barnes, Dave Berry, et al. SML/NJ. http://cm.bell-labs.com/cm/cs/what/smlnj/.

[3] Matthias Blume and Andrew W. Appel. Lambda-splitting: A higher-order approach to cross-module optimizations. Technical Report CS-TR-537-96, School of Computer Science, Princeton University, June 1996.

[4] Hewlett Packard Company. http://www.netperf.org.

[5] USC Information Sciences Institute. Internet protocol. RFC 791, September 1981.

[6] Mark Leone and Peter Lee. Optimizing ML with run-time code generation. Technical Report CMU-CS-95-205, School of Computer Science, Carnegie Mellon University, December 1995.

[7] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML(Revised)*. The MIT Press, 1997.

[8] IETF Network Working Group. Requirements for internet hosts – communication layers. RFC 1122, October 1989.

[9] Herbert Schildt. *The Annotated ANSI C Standard*. Osborne, 1990.

[10] Richard Sites. *The Alpha Architecture*. Digital Press, 1992.

[11] Philip Wickline, Peter Lee, Frank Pfenning, and Rowan Davies. Modal types as staging specifications for run-time code generation. *ACM Computing Surveys*, 20(3), September 1998.