CSC2227 Project Design Document: Fletcher

Eric LaForest

March 6, 2008

1 System Design

This project involves the creation of a simple operating system, named *Fletcher*, whose purpose is to evaluate the impact on system design of cheap context switches and flat, unvirtualized memory. The effects of dynamic code generation are also evaluated to a lesser extent. The building blocks of Fletcher are a simulator for the Gullwing microprocessor architecture and the Flight programming language. Both originate in prior undergraduate work [LaF07].

1.1 Memory Map

Figure 1 shows the overall system memory map. Given a machine word width of n bits, the address space spans the entire 2^n addressable range. All memory locations are word-wide. There are no half-words or bytes. The actual usable RAM is mapped to the bottom of the address space.

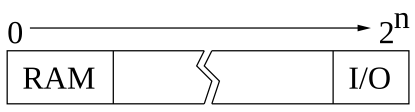

Figure 1: System Memory Map

1.1.1 Input/Output

At the top end of the address space lie memory-mapped I/O ports. These are the only means of communicating with the outside world. The primary I/O channels are the console input and output ports. For simplicity, reads or writes to these memory locations are made blocking by the simulator.

In order to measure the passage of time, the simulator will need to be augmented with a settable cycle counter. After each cycle, the value stored in that memory location is incremented by one. A new value may be written at any time. A settable counter avoids having to deal with counter wraparound in most cases. For example, the counter could be to set to a negative value, followed by polling and a jump-on-positive branch.

The simulator has no means of storage at the moment, but simulated access to a disk drive could be done through the addition of read and write ports along with associated status ports. To read a sector, the sector number is first written to the read port. The read status port will take on a non-zero value once the data is ready to be read out through the read port. The write port operates in the same manner, except the write status port indicates when the data port is free. Only one read and write may be in progress at any time. The simulator may delay the setting of the status ports to emulate the latency of disk accesses, or the status ports may be omitted altogether if disk latency is not significant to benchmarks.

1.2 Gullwing Microprocessor Architecture

The Gullwing processor [LaF07] is a small, second-generation stack architecture, implemented here as a cycle-accurate simulator. Figure 2 shows the major blocks of Gullwing:

- The Instruction Shift Register (ISR) holds the current group of instructions.
- The Program Counter (PC) holds the next sequential address to fetch from.
- The Memory Address Register (MAR) controls the memory address bus.
- The ALU operates exclusively on the top of the Data Stack (DS) and TOP register.
- The Return Stack (RS) and R Register hold the return addresses of subroutine calls.
- The Address register (A) holds addresses for loads and stores.
- The incrementer (+1) is used for post-incrementing loads/stores and for the PC.

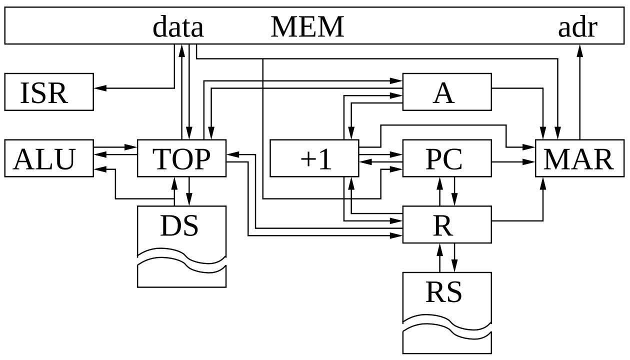

Figure 2: Gullwing Microprocessor Architecture Block Diagram

The instruction set is simple and stack-oriented. Arithmetic and logical instructions operate on TOP and the Data Stack. Branches, calls, and returns use the Return Stack. Loads and stores use the A or R registers to hold an address, with the option of post-incrementing, and use the Data Stack for data. Data can be moved between the two stacks.

It is important to note that the stacks are not in memory, nor randomly accessible. The crucial benefit of this design is that procedure calls and returns are extremely fast, taking only two cycles: one to place the return address on the Return Stack, the other to load the first instruction group of the called procedure. There is no conventional call stack in memory. Parameters, intermediate computations, and return values reside entirely on the Data Stack.

1.2.1 Virtualization Support

In order to properly support virtualization as per Popek and Goldberg [PG74], a supervisor mode bit and program counter must be added, along with memory range checkers, and a privileged instruction to change modes.

Range Checkers Two registers will be added to contain the lower bound (LB) and the upper bound (UB) of the portion of address space accessible to a user process. A pair of subtractors will continually compare these to the current memory address being accessed. If either bound is exceeded, a TRAP bit will be set. This bit will cause loads, stores, and branches to execute a TRAP sequence instead of completing their operation.

TRAP sequence If the TRAP bit is set during the first cycle of an instruction which accesses memory, the second cycle of the instruction is pre-empted by a TRAP sequence:

Cycle 1	Push LB onto DS. Set LB to 0. Push PC onto RS. Load PC from TRAP Program Counter (TPC).
Cycle 2	Push UB onto DS. Set UB to 2^n. Increment PC and MAR. Set mode to Supervisor. Clear TRAP bit. Push ISR onto RS. Load ISR from memory.

This places the memory bounds, the current PC, and the current ISR of the user process on the stacks, making them available to the supervisor software. Note that the new LB and UB values make it impossible to cause a TRAP sequence while in supervisor mode.

Return To User (RTU) Instruction Once in supervisor mode, the reverse of a TRAP sequence is the Return To User (RTU) instruction:

Cycle 1	Pop DS into UB. Store pre-incremented PC in TPC. Pop RS into PC and MAR.
Cycle 2	Pop DS into LB. Pop RS into ISR. Set mode to User.

This causes a return to User mode, setting the entry point, the instruction group, and the memory bounds. The next trap will return to the point right after the RTU instruction, as stored in the TRAP Program Counter (TPC). Attempting to execute RTU while in User mode will cause a TRAP sequence.

1.3 Flight Programming Language

The core software of Fletcher is the Flight programming language [LaF07]. It is a minimal, Forth-like language. Figure 3 shows the layout of the Flight language kernel.

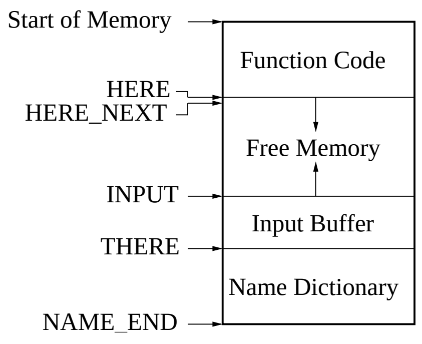

Figure 3: Flight Language Kernel Memory Map

The Function Code area contains compiled machine code. The current and next address at which code is to be compiled are denoted by the HERE and HERE_NEXT variables. The Name Dictionary contains the names and addresses of compiled functions. It is a linear list traversed backwards during look-ups, so the most recent code by a given name is found first. The head of the Name Dictionary is pointed to by the THERE variable. It is also the end of the Input Buffer, whose head is indicated by the contents of INPUT. The end of the Name Dictionary is pointed to by the NAME_END variable.

The Flight kernel is composed of functions which read input, look-up names, define names, and compile Gullwing opcodes. The main loop of the Flight kernel consists of reading the name of a function into the Input Buffer, looking it up in the Name Dictionary, and executing the associated Function Code. If the executed code defines a function name, the name of the function is the contents of the Input Buffer, which is altered in place to extend the Name Dictionary. If the executed code compiles code, it is appended to the Function Code area. There is no further syntax.

1.3.1 Metacompiler

As described, Flight is limited to compiling code within itself. However, it is possible to temporarily redirect HERE, THERE, and the other kernel variables to point to another area of memory. This will create a separate memory map like the one in Figure 3 which the original Flight kernel can then manipulate as if it was itself.

Furthermore, if a function name is not found in the new dictionary, the look-up function can fall back to the original dictionary but still execute the found code in the new memory map, simply by saving and restoring the kernel variables at the appropriate times during look-up.

In Forth lingo, this process is called *metacompilation*, since it allows the full power of the language to be used to create a separate piece of software, including a new copy of itself.

1.4 Processes

Using the metacompiler, the supervisor-mode Flight language kernel can create new user processes with a structure similar to Figure 4.

A user process is divided into two parts: kernel state and user space. The kernel state contains pointers to the next and previous processes in a ring, the unique process number (PID), and its upper and lower user memory bounds (UB and LB). These last two define a user area immediately following the kernel state, which is the range of memory accessible by this process.

In theory, the user area can contain arbitrary code, but for flexibility and security, the kernel will initialize it with a minimal user-version of itself so no user code is run by the kernel. The new process, being a Flight language kernel, can then compile the actual application code for the process. This will also simplify IPC, as shown later.

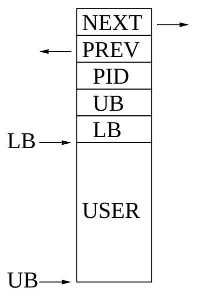

Figure 4: Structure of a Fletcher process

1.5 OS Services

Syscalls A fast syscall mechanism can be built from the fact that the RTU instruction causes a TRAP sequence when executed in User mode. A process can place a syscall number and its arguments on the Data Stack, then execute RTU. The kernel, seeing that the trapped instruction was RTU, can then call the appropriate syscall function, leave its return value on the Data Stack, and execute RTU to return to the calling process.

Traps Another means of entering the kernel is through a load/store memory trap. After the TRAP sequence, the kernel can find the load/store address either in the R or the A register and take action based on the memory range. For example, if the address is within the processes' kernel state, then a load would be allowed, but a store would be discarded. For example, this scheme could implement the functionality of the UNIX *getpid()* syscall as a simple load from the perspective of the process.

Storage I/O The trap mechanism can also be used to implement a uniform abstraction for storage. If a trapped load address is beyond RAM, but below I/O (Figure 1), then the kernel could translate the address to a physical disk block number, fetch the disk block via the I/O ports, buffer the block, and return the particular word within the block to the process. A subsequent access would return the buffered data instead. This translation erases the distinction between disk, I/O, and memory for processes. Some memory areas simply appear to be persistent or change after each read.

IPC Since each process is an instance of the Flight language, it can run the read-lookup-execute loop to receive and process messages. A sending process can place the name of the command in its Input Buffer, place the PID of the recipient process on the Data Stack, and perform a 'send' syscall. The kernel will then copy the name into the input buffer of the recipient process, which will read the name when it is next scheduled.

2 Evaluation Plan

The Fletcher system begins with the proposition that the benefits of faster context switching gained from a stack architecture and from avoiding paged virtual memory will outweigh the loss in flexibility, and that this flexibility can be mostly regained through the use of dynamic code generation.

The overall hypothesis to be tested is: Can OS services with a performance that is equal or better than shown in past research be implemented in such a system, given the low overhead of context switches and the possibility of generating optimized code?

Test Framework: Cycle Counter The key component of the test framework is the settable cycle counter. All micro-benchmark results will be measured in elapsed processor cycles. Given the wide variety of machines used in past research, a cycle count should be a fairly uniform measure of performance since it abstracts away details like actual processor speed, the types of instructions, and the level of instruction parallelism. Since micro-benchmarks are small and repetitive, the code locality is assumed to be very high, and thus mostly working out of a warm cache, allowing memory latency to be ignored.

Syscalls and Traps Since it is the simplest syscall, the basic test of context switch performance is *getpid()*. The test will consist of a single process performing the syscall. The equivalent function can also be implemented using memory traps and tested in the same manner.

Process Scheduling Another test of context switch overhead is the time taken to switch from one process to the next. The test-bench will consist of a pair of processes, one immediately yielding to the other. This can be extended to a ring of many processes, chiefly for the purpose of measuring the time taken to yield to a arbitrary process (identified by PID) instead of to the next one in the ring.

Process Management The time taken to create or destroy a process is significantly affected by the use of virtual memory. One process will perform the syscalls to create then destroy a number of empty processes, reporting the time for each. The same test will be repeated for processes containing the initial user-version of the Flight language kernel. For the second version, the expected main source of overhead will be the time taken to compile each process from source, which is the trade-off from creating new virtual address spaces.

Inter-Process Communication IPC is strongly affected by context switch overhead and virtual memory since it involves crossing process domains. The test will consist of one process sending a command to another to request a number, then descheduling itself to wait for a reply. The second process, which was waiting for a message, will send a return message containing the number.

A second version of the IPC test will show the benefits of dynamically generated code. The first process will ask the second for the address of its function which returns a number. The first process will then instruct the second to run a special message processing loop that uses command addresses instead of names. The first process can then compile and run a new version of the IPC test which uses the leaner message format.

Storage Rather than have each process go through a read/write syscall to access storage, the fact that memory traps are just as fast as syscalls might make it practical to simply map in software the disk blocks to the memory area beyond RAM, reducing the user interface to loads and stores. The kernel could translate the address of the trapped load/store to a disk block number, fetch the disk block, and return the contents of the accessed word to the process. The performance can be compared to that of buffered file and *mmap()* reads in past research.

The naive version of this test would fetch the disk block at every access. An improved version would buffer the last accessed block. Finally, these can be compared to a syscall which returns the contents of the disk block to the user process. The test application will sequentially read a given amount of disk data into process memory.

References

[LaF07] Charles Eric LaForest, *Second-generation stack computer architecture*, Independent Studies thesis, University of Waterloo, April 2007.

[PG74] Gerald J. Popek and Robert P. Goldberg, *Formal requirements for virtualizable third generation architectures*, Commun. ACM **17** (1974), no. 7, 412–421.