Tackling Computer Security Issues Through Lazy, Stackless Functional Programming Architectures

Cecil Accetti, Eshton Robateau, Peilin Liu
School of Electronic Information and Electrical Engineering - Shanghai Jiaotong University
Shanghai, China
cecaccetti@sjtu.edu.cn

1. Problem And Motivation

As transistor scaling hits its limits, a new challenge for general purpose computing arises: the need for efficient structures both in architecture and microarchitecture levels. Until recently, gains in transistor performance justified complex architectures, while computer systems security and safety has been dealt with as a software-only problem. The result is that current computer architectures are plagued with security vulnerabilities, intentional and unintentional backdoors. As a response to this, we propose a fresh look on functional programming architectures, based on combinatory logic. A functional processor has been designed, enabling the development of safe, purely functional operational systems and device drivers.

Traditional von Neumann architecture computers follow an imperative model of computation based on sequential instructions. Imperative programs are essentially time-dependent (stateful) sequences of variable and pointer assignments. Furthermore, state and side-effects are central to the imperative model, and are fundamental sources of security flaws, as both the well-intentioned programmer and the malicious attacker can directly control the execution flow of programs, and have direct access to read and modify memory locations.

Out-of-order execution and branch speculation introduce more complications to this design, by breaking the sequential nature of the imperative model. These performance-improving strategies deepen the dependency on state, both from the program (variables) and from the machine (registers, caches), which has long been exploited by malicious software, culminating with the recent Meltdown and Spectre-class attacks [1]. The main difference now, is that architecture and microarchitecture flaws are known to be the culprits.

Functional programming (FP) languages are known for their safe typing systems, high modularity, and its use on distributed systems [2]. FP languages are usually stateless and forbid or strictly control side-effects. However, they tipically rely on virtual machines or other runtime environments for execution in imperative architectures. When security and safety is concerned, their benefits might be lost if the host machine is flawed.

Motivated by these issues, this work makes the following contributions:

- We define a combinator-set architecture (CSA) for lazy evaluation of functional programs;
- The CSA supports controlled side-effects, input/output and interrupt handling combinators that allow the design of device drivers and operational-system tasks.
- A stackless processor following this architecture is presented. A distributed memory subsystem allows up to 4 read/write accesses per clock cycle.
- Hardware garbage collection, by a combination of pauseless and stop-the-world algorithms.

Much has changed since functional programming architectures were first proposed in the 1980s [3,4]. Current EDA tools, compiler algorithms, and the cost of semiconductor manufacturing allow to explore the concept further than previous implementations. Recent works, such as the Reduceron, targeted FPGA platforms, following design concepts derived from the Glasgow Haskell Compiler (GHC) [5]. The main drawback of this approach is that it tries to mimic the behavior of the compiler's abstract machine (stack-based) into a physical machine.

2. Proposed Architecture and Experiments

Our approach is based on graph reduction of combinator expressions (Figure 1). Combinators are simple generalized functions, proven to be Turing-complete . Their application to compilers was proposed in [6] but their use in programming languages declined in favor of supercombinators and abstract machines targeting imperative computers. New combinators were obtained through profiling of common higher-order functions (Table 1). Figure 2 illustrates the CSA and the reduction rules for S, as an example. In average, compiled code for the CSA is 10% smaller than code compiled with Turner combinators only.

The presented processor employs a stackless microarchitecture called *fun*, shown on Figure 3. It performs graph-traversal in normal order, issuing one node on each clock cycle, until a reducible combinator (redex) is found. While G-machine-based graph reducers depend on one or more stacks to store issued graph nodes [3], a key idea in the fun architecture is to store the 5 most recently issued nodes in registers (graph-spine window, GW), this way only one clock cycle is needed to reduce any combinator. For a lower graph-traversal time, and memory accesses, the GW is updated at every reduction.

The memory architecture is implemented in a network of small memory banks, allowing for up to 4 read/write accesses per clock cycle. Memory management is performed by a garbage collector that integrates dynamically reuse of dereferenced cells during graph reduction, similar to the strategy used in [4], and freeing memory through a mark-compact algorithm [7].

The proposed architecture was evaluated by benchmarking of common higher-order functions and selected applications (Table 1). We compare the evaluation of the benchmarks with the fun CSA and Turner combinators. As a reference, the evaluation times for the GHC are included.

As measured during benchmarking, the fun graph reducing processor averages 3.4 cycles/ combinator reduction, for which 2.26 cycles are spent in graph traversal. The RTL model was described in VHDL, with synthesis targeted a Cyclone IV FPGA, in less than 8000 logic elements. For this device it achieves a maximum clock frequency of 125 MHz.

3. Conclusions

Further work has to be done to eliminate Backus' "von Neumann bottleneck". A high memory bandwidth architecture should allow the use of complex combinators, which could read and modify more than five memory locations in a single clock cycle. Meanwhile, we show that functional programming as a basis for architecture design is a good candidate to solving some of the issues of traditional architectures, controlling state and timing dependencies, thus being suitable for implementing security-critical tasks, such as in OS kernels and embedded systems.

4. References

[1] M.Lipp, M.Schwarz, D.Gruss, T.Prescher, W.Haas, S.Mangard, P.Kocher, D.Genkin, Y.Yarom, M.Hamburg, "Meltdown", https://meltdownattack.com/

[2] John Hughes, "Why Functional Programming Matters", Research Topics in Functional Programming, pp. 17-42, 1990.

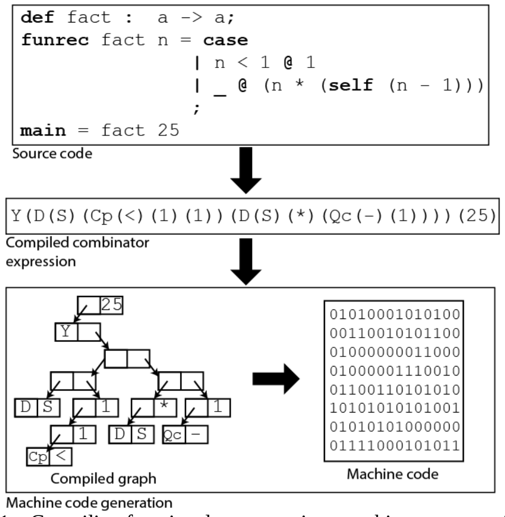
[3] R.Kieburtz, "A RISC Architecture for Symbolic Computation",ACM SIGARCH Computer Architecture News: Volume 15 Issue 5, Oct. 1987

[4] W.Stoye, T. Clarke, A. Norman "Some Practical Methods for Combinator Reduction" LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming

[5] M.Naylor, C. Runciman, "The Reduceron Reconfigured", Proceedings of the International Conference on Functional Programming, ICFP 2010

[6] D.Turner,."A New Implementation Technique for Applicative Languages ",Software-Practice and Experience, vol 9, no. 1, pp 31-49, 1979

[7] M.Maas,K.Asanovic, J.Kubiatowicz, "A Hardware Accelerator for Tracing Garbage Collection", The 45th ACM/IEEE International Symposium on Computer Architecture



```
def fact :  a -> a;
funrec fact n = case
                | n < 1 @ 1
                | _ @ (n * (self (n - 1)))
                ;
main = fact 25
```

```
Y(D(S)(Cp(<)(1)(1))(D(S)(*)(Qc(-)(1))))(25)
```

```
01010001010100
00110010101100
01000000011000
01000001110010
01100110101010
10101010101001
01010101000000
01111000101011
```

Figure 1 – Compiling functional programs into combinator expressions for graph reduction

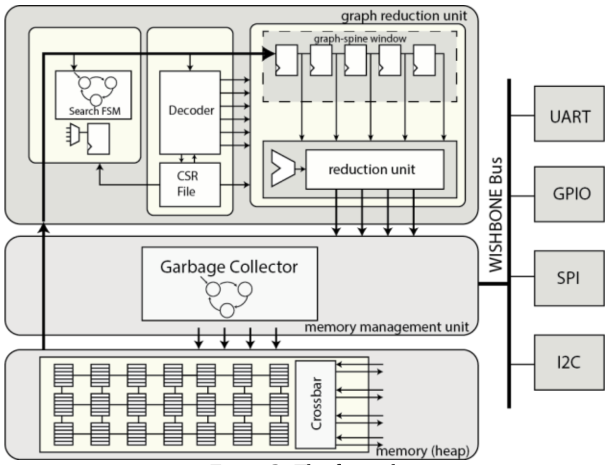


Figure 3- The *fun* architecture

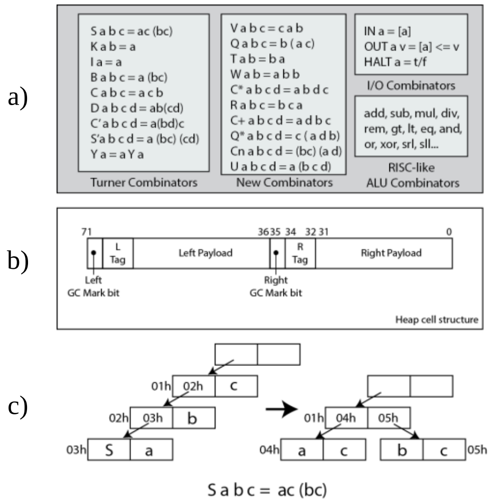


Figure 2 – a)The fun combinator set. b) Graph node structure in the heap. c)Combinator reduction rule for S.

Table 1 – Benchmarking of the fun architecture. GHC compiler times for reference. Comparison with Kieburtz (microcoded G machine). All values in μs.

Program	fun (Cyclone IV FPGA, 100 MHz)	fun (Turner) (Cyclone IV FPGA, 100 MHz)	GHC 7.10.3, (Intel Core i5, 3.4GHz)	Kieburtz [3] 10 MHz (1987)
map	725	755	100	-
foldl	1452	1531	200	-
scanl	2008.2	2087	300	-
zipWith	3043.6	3043.6	300	-
filter	45.2	46.0	20	-
transpose	352.1	364.2	100	-
take	19.2	19.2	10	-
mersenne	1600.0	1632.3	800	-
rsa	1102	1111	700	-
qsort	1136.2	1167.4	400	-
hamming	101.5	102.5	40	-
fiblist	69.97	72.0	30	2,920
sieve	148,989.6	149,344	7000	798,000
tak	47,032.7	47,032.7	1500	1.691s