Keeping the PilGRIM at a steady pace

Avoiding pipeline stalls in a lazy functional processor

Arjan Boeijink

University of Twente
Enschede

NL-FP dag 2014

PilGRIM introduction

Pipelined Graph Reduction Instruction Machine

- A processor specialized for lazy functional languages
- Executes Instruction set that is close FP core language
- Deep pipeline (about 10 stages) for competitive performance
- High level stack based instruction set working on wide data
- Moving a lot of data in parallel but only locally
- Use extra available transistors for runtime optimizations

Why design a processor for lazy functional languages again?

- Exploring 20 years of alternative history in hardware
- Can overhead of FP be eliminated by using extra hardware?
- Functional languages are a difficult workload for current processors
- Hardware technology changes slowing down gives time to catch up

The instruction set of the PilGRIM

Derived from GRIN (Graph Reduction Intermediate Notation)

- High level instructions with builtin support for eval, apply and case
- Modified with focus on parallelism within a single instruction

PilGRIM instructions work on whole nodes

- Nodes are: Constructors, Functions and Partially applied functions
- Each node has tag/header word for all meta data and a sequence of either references or unboxed primitives
- Whole nodes are moved between stack and heap at once

Instruction format (store/return)

instruction	node tag	node arguments	stack cleanup
Store	$\mathtt{F}_{foldr}$	f z xs	optional
Return	$\mathtt{C}_{Pair}$	x y	pop mask

Functional language support and example

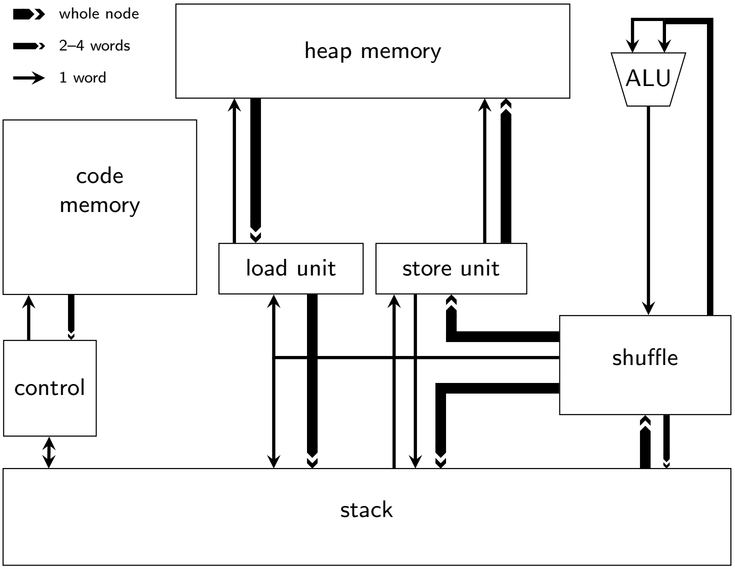
Call-like instructions are combinations of three aspects

control part	what is called	'application'	stack cleanup
`Call`	`Eval x`	`Apply` $\vec{a}$	⋮
`Jump`	*function* $\vec{x}$	`Select n`	pop mask
`Case` [jump table]	`Receive`	()	⋮

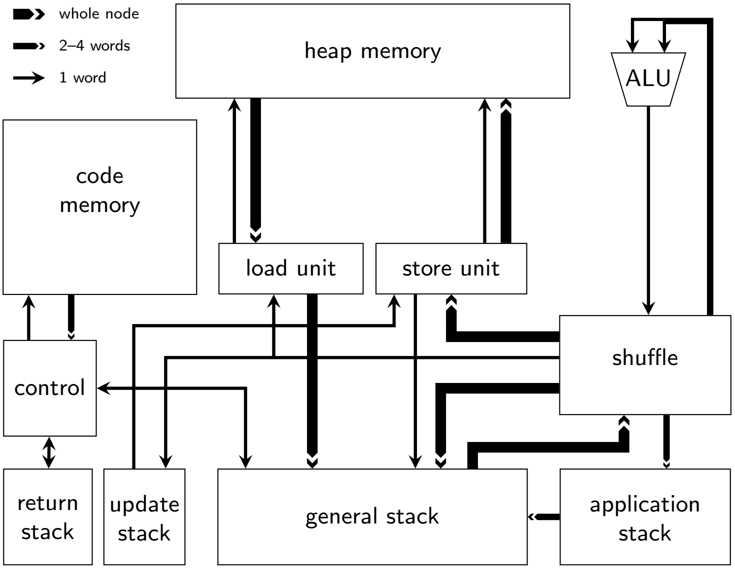
`foldr` expressed in 4 PilGRIM instructions

```
foldr $f$ $a$ $ys$ =
  Case [$nil$, $cons$] (Eval $ys$) ()
     $C_{Nil}$ →
        Jump (Eval $a$) ()
     $C_{Cons}$ $x$ $xs$ →
        $rs$ ← Store ($F_{foldr}$ $f$ $a$ $xs$)
        Jump (Eval $f$) (Apply $x$ $rs$)
```

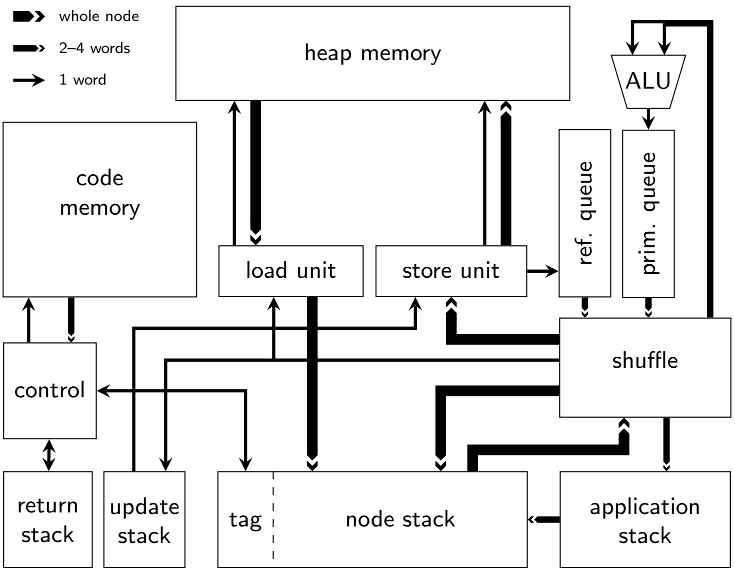
Overview of a simplified architecture



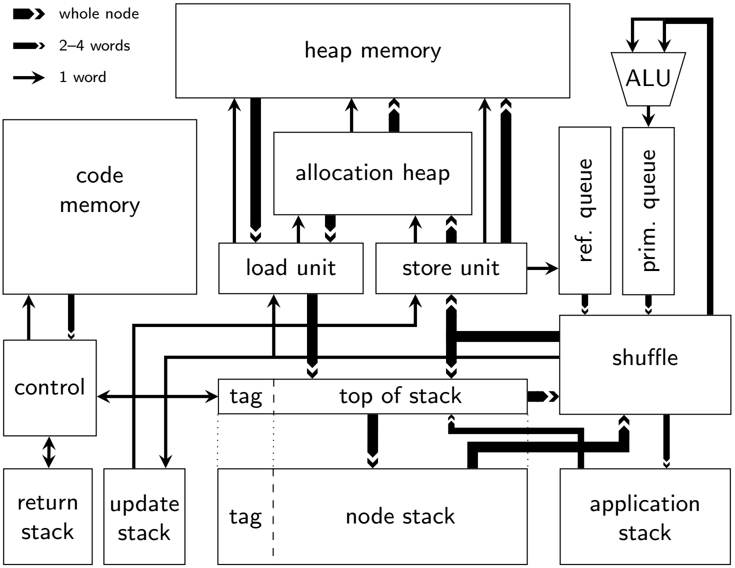
Splitting the stack



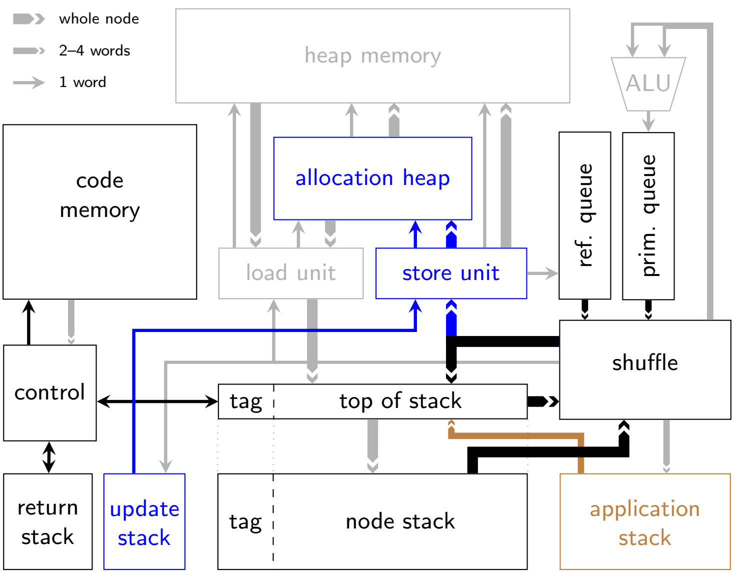
Queues for temporary references and primitive values



Adding the allocation heap and top of the stack



Executing a Return instruction

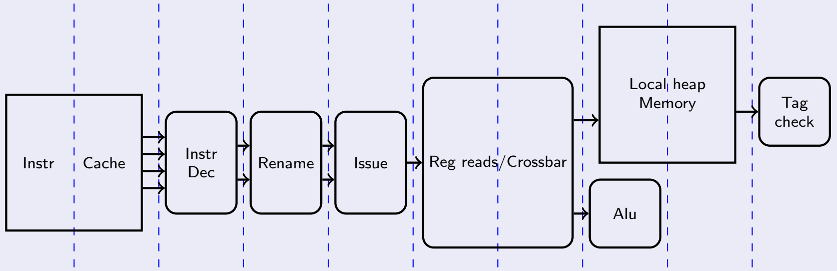


Pipelining in the PilGRIM

Reasons for pipelining

- Wide instructions doing 10x the work is not enough
- Need ∼1Ghz frequency to be competitive
- Trying to address latency related bottlenecks
- Exploring what makes FP code tricky to execute

Simplified pipeline structure

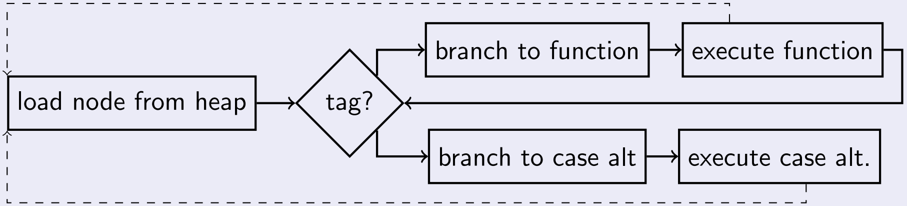


The problem of keeping a steady pace

Lazy functional languages stress bottlenecks

- Very memory intensive due to immutable data
- Loading a node from the heap has implicit control flow
- About 40% of all PilGRIM instructions contain control flow

Executing a simple case expression



- Need to wait on load result to determine next instruction to fetch
- Can start next load only after instructions are fetched and decoded

Keeping more data local

Allocation heap

- Local memory that supports a node load or store every cycle
- Can be viewed as a fast tagless first level cache
- Reducing bandwidth requirement for the other caches

Local reference counting

- Each allocation heap entry has a reference counter
- Acts as garbage collection filter for short lived data
- Avoid polluting the cache with temporary data
- Also reduces frequency of real garbage collection

Uniqueness bit on each reference

- Allows for destructive reads from heap
- Marking reference shared is a cheap local operation

Making decisions while waiting on a load

Pointer tagging extreme

- Exploiting the abundance of bits in a 64 bit architecture
- Every reference is split into 16 bits of meta data and a 48 bit pointer
- The pointer tag contains information about of the stored node
- Hardware support allows pointer tagging without overhead

Faster case expressions with pointer tagging

- If the pointer tag has known constructor information
  the jump the right case alternative can be made early
- Load node data and the code for case alternative concurrently

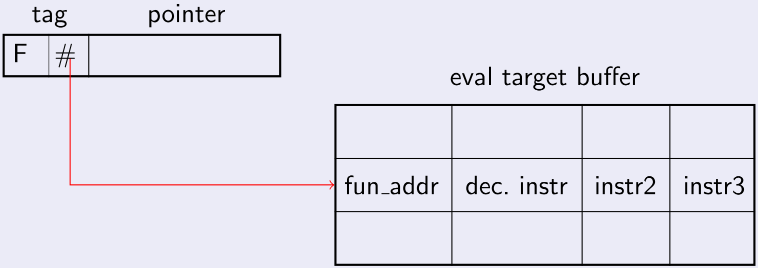
Storing some nodes only in the reference

- Constructors of enumeration like data types
- Dynamicly storing small boxed Ints in the reference

Reducing delay on evaluation of thunks

Eval target buffer for function references

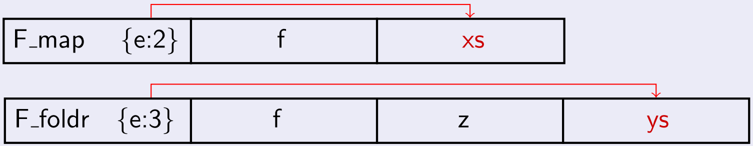
- While waiting on function node load use pointer tag
- Prediction of which function will do the evaluation
- Put a small hash of the function address into the pointer tag
- Eval target buffer contains a few (predecoded) instructions
- Start executing function right after load on correct prediction



Starting loads without waiting for the instruction

Most functions start with fetching one of the arguments

- But have to wait for the instruction with the load in the function
- Solution is storing argument index of next fetch in header
- Can start load while waiting for instructions of this function



Can use similar trick for case alternatives

Add a few bits for next fetch in case jump table:

```
foldl f z xs = case xs of
  Nil       {e:_} -> z
  Cons y ys {e:2} -> foldl f (f z y) ys
```

Starting known loads early

Prefetching know loads

- Compiler often know which references will be loaded
- Could insert prefetch instructions but can be too aggressive
- Problem is that function calls could take a long time

Using the 'application' part of instructions

- Queue on stack the next load after a eval/function call
- Now can start loads early when they are needed soon
- And does not increase number of instructions

Example for the function addInt x y is:

$\mathtt{C}_{\mathtt{Int}}\ a \leftarrow \mathtt{Call}\ (\mathtt{Eval}\ x)\ (\mathtt{ThenFetch}\ y)$
$\mathtt{C}_{\mathtt{Int}}\ b \leftarrow \mathtt{Call}\ (\mathtt{EvalFetched}\ x)\ ()$
$c \leftarrow \mathtt{PrimOp}\ +\ a\ b$

Open problems

Are these trick enough to keep the pipeline filled?

- Unfornately need finished hardware design to tell you
- Can the instruction cache deliver enough instructions?
- Might need to use multithreading to keep the core busy

Dealing with conditional branches

- A branch disrupts most pipelining stall avoiding tricks
- Might fetch and decode both branches at the same time
- Internal state too complex for using speculative execution
- Convert some branches in conditional or select instructions
- Combining nested if expressions into case like construct
- Eager execution of functions with branches outside the critical path

Conclusions

Can avoid most pipeline stalls with a lot of effort

- Combining common control flow in high level instruction set
- Design memory system for keeping useful data local
- Pointer tagging for avoiding dependencies on loads
- Can start many loads early by extra annotations
- Stalls still happen but something useful is done at same time

Future work

- Implement all these idea in cycle accurate simulation
- Produce synthesizable hardware from it (for FPGA?)
- Run bigger and complex programs to find the next bottleneck
- Compiler optimizations to reduce amount of control flow
- Multithreaded core to deal with external memory latencies?
- In the long term built multicore variant of the architecture

Thank you

Questions?

Thank you

Questions?

Advertisement:

Master assignment available on compiler optimizations for PilGRIM

How can specialized hardware support improve performance of lazy functional languages?

Major improvements:

- Reading/writing of whole nodes from/to the heap at once
- Using the stack without load and store instructions
- Parallel movements of data between all the stacks
- Keeping more data local by reference counting

Minor improvements:

- Hardware supported call/return instructions
- Extensive pointer tagging scheme without any overhead
- Hardware support for evaluation/updating/application
- Heap and stack checks are done in hardware
- Cache behaviour tuned for functional programs

Comparing the PilGRIM with the Reduceron

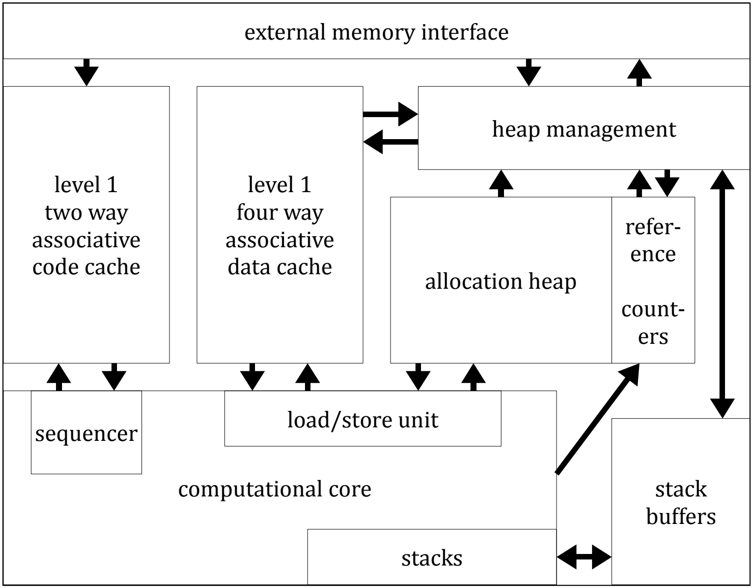
PilGRIM:

- deeply pipelined design
- silicon targeted
- high clockspeed is as important as parallelism
- GRIN derived instruction set
- trying to benefit from GHC's optimizations
- data types/case expr. hardware supported

Reduceron:

- single cycle reduction step
- designed for a FPGA
- focus on exploiting memory parallelism
- template instantiation
- focus on dynamic runtime optimizations
- data types/case expr. encoded in function

Memory hierarchy and reference counters


external memory interface
heap management
level 1
two way
associative
code cache
level 1
four way
associative
data cache
allocation heap
refer-
ence
count-
ers
sequencer
load/store unit
computational core
stacks
stack
buffers