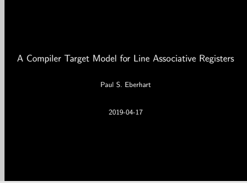
A Compiler Target Model for Line Associative Registers

Paul S. Eberhart

2019-04-17

A Compiler Target Model for LARs



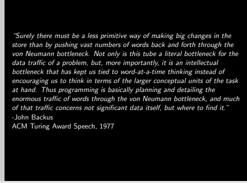
A Compiler Target Model for Line Associative Registers

Paul S. Eberhart

2019-04-17

"Surely there must be a less primitive way of making big changes in the store than by pushing vast numbers of words back and forth through the von Neumann bottleneck. Not only is this tube a literal bottleneck for the data traffic of a problem, but, more importantly, it is an intellectual bottleneck that has kept us tied to word-at-a-time thinking instead of encouraging us to think in terms of the larger conceptual units of the task at hand. Thus programming is basically planning and detailing the enormous traffic of words through the von Neumann bottleneck, and much of that traffic concerns not significant data itself, but where to find it."
-John Backus
ACM Turing Award Speech, 1977

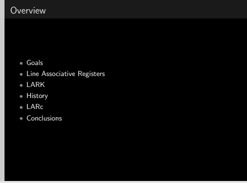
A Compiler Target Model for LARs


"Surely there must be a less primitive way of making big changes in the store than by pushing vast numbers of words back and forth through the von Neumann bottleneck. Not only is this tube a literal bottleneck for the data traffic of a problem, but, more importantly, it is an intellectual bottleneck that has kept us tied to word-at-a-time thinking instead of encouraging us to think in terms of the larger conceptual units of the task at hand. Thus programming is basically planning and detailing the enormous traffic of words through the von Neumann bottleneck, and much of that traffic concerns not significant data itself, but where to find it."
-John Backus
ACM Turing Award Speech, 1977

- This quote is my favorite way to introduce LARs
- Backus was talking about programming languages
- It's also a problem for architectures
- lots of people have taken shots at it, here's ours

Overview

- Goals
- Line Associative Registers
- LARK
- History
- LARc
- Conclusions


Overview

- Goals
- Line Associative Registers
- LARK
- History
- LARc
- Conclusions

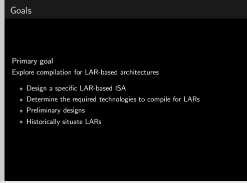
-

Goals

Primary goal

Explore compilation for LAR-based architectures

- Design a specific LAR-based ISA
- Determine the required technologies to compile for LARs
- Preliminary designs
- Historically situate LARs



Goals

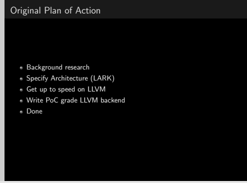
Primary goal
Explore compilation for LAR-based architectures

- Design a specific LAR-based ISA
- Determine the required technologies to compile for LARs
- Preliminary designs
- Historically situate LARs

- Early plan was "Port LLVM" - that went out the window
- Specific ISA is LARK
- History is a side-effect

Original Plan of Action

- Background research
- Specify Architecture (LARK)
- Get up to speed on LLVM
- Write PoC grade LLVM backend
- Done

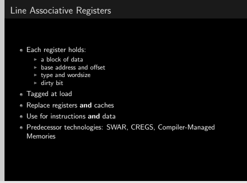

Original Plan of Action

- Background research
- Specify Architecture (LARK)
- Get up to speed on LLVM
- Write PoC grade LLVM backend
- Done

- This was the plan in early 2010, it is now mid-2019, obv. things didn't work out
- It turned out the initial assumptions were completely wrong
- The first 18mo were spent learning that, the rest of the decade on figuring out what it means
- Only recently come around to the things learned being anything but discouraging

Line Associative Registers

- Each register holds:
  - a block of data
  - base address and offset
  - type and wordsize
  - dirty bit
- Tagged at load
- Replace registers **and** caches
- Use for instructions **and** data
- Predecessor technologies: SWAR, CREGS, Compiler-Managed Memories

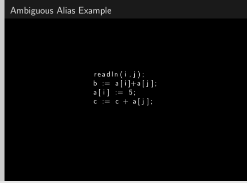

Line Associative Registers

- Each register holds:
  - a block of data
  - base address and offset
  - type and wordsize
  - dirty bit
- Tagged at load
- Replace registers **and** caches
- Use for instructions **and** data
- Predecessor technologies: SWAR, CREGS, Compiler-Managed Memories

- Underway since early 2000s, (Krishna Melarkode MS 2004)
- Several projects to implement pieces since (Lim, Ponalla, Sparks, Clark)
- SWAR: Vectors (Mid 90s)
- CREGS: Address Tags for Ambiguous Aliases (Deitz/Chi 89)
- Compiler-Manged: Explicit fetch

Ambiguous Alias Example

```
readln(i,j);
b := a[i]+a[j];
a[i] := 5;
c := c + a[j];
```



Ambiguous Alias Example

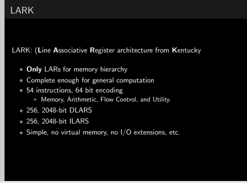
readln(i,j);
b := a[i]+a[j];
a[i] := 5;
c := c + a[j];

- No way to statically determine if a[i] and a[j] are pointed at the same thing
- Requires repeated flushes: 2 round trips
- Unknown time. L1 Cache? TLB miss?
- a[i] and a[j] in each register is two loads, always.
- a[j] would require a flush on conventional because a[i] might be pointed at it!

LARK

LARK: (**L**ine **A**ssociative **R**egister architecture from **K**entucky

- **Only** LARs for memory hierarchy
- Complete enough for general computation
- 54 instructions, 64 bit encoding
  - Memory, Arithmetic, Flow Control, and Utility.
- 256, 2048-bit DLARS
- 256, 2048-bit ILARS
- Simple, no virtual memory, no I/O extensions, etc.


LARK

LARK: (**L**ine **A**ssociative **R**egister architecture from **K**entucky

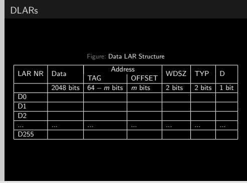
- **Only** LARs for memory hierarchy
- Complete enough for general computation
- 54 instructions, 64 bit encoding
  - Memory, Arithmetic, Flow Control, and Utility.
- 256, 2048-bit DLARS
- 256, 2048-bit ILARS
- Simple, no virtual memory, no I/O extensions, etc.

- also "On a Lark" because it's meant to be a strawman design
- about 192KB of high-speed memory, reasonable in a modern context
- 4 instruction formats
- Talk about the calling conventions prelim?

DLARs

Figure: Data LAR Structure

LAR NR	Data	Address		WDSZ	TYP	D
		TAG	OFFSET			
	2048 bits	$64 - m$ bits	m bits	2 bits	2 bits	1 bit
D0						
D1						
D2						
...	...	...	...	...	...	...
D255						


DLARs
Figure: Data LAR Structure

- low bits of address are offsets in the line
- DLARs are like fast named windows into main memory
- DLAR0 is reserved for constant 0 (which will be 0 in all representations)

Tag Encodings

Figure: Word Size Encodings

Value	Object Size
00	8
01	16
10	32
11	64

Figure: Type Encodings

Value	Type
00	Reserved
01	Unsigned Integer
10	Signed Integer (2's compliment)
11	Float (IEEE754-ish)

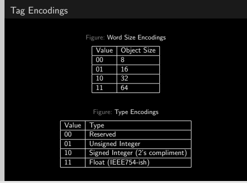

Tag Encodings

Figure: Word Size Encodings

Value	Object Size
00	8
01	16
10	32
11	64

Figure: Type Encodings

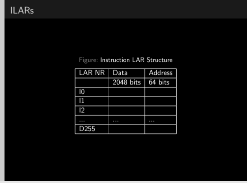
Value	Type
00	Reserved
01	Unsigned Integer
10	Signed Integer (2's compliment)
11	Float (IEEE754-ish)

- 8bit 754ish $(1 + 4 + 3)$ - 1sign, 4 exponent, 3 mantissa.
- all 1 $exp + 0mantissa = NaN$, all $1exp + non0mantissa = \infty$, $0exp = 0$

ILARs

Figure: Instruction LAR Structure

LAR NR	Data	Address
	2048 bits	64 bits
I0		
I1		
I2		
...	...	...
D255		



- No need for dirty bits or type tags.
- Instructions execute only out of ILARs PC points to ILAR,Offset
- Could compress in memory, LARK doesn't for simplicity

Memory Instructions

```
LOAD - 0b0100 + typ + wdsz
STORE - 0b0101 + typ + wdsz
```

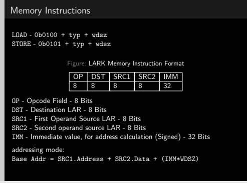
Figure: LARK Memory Instruction Format

OP	DST	SRC1	SRC2	IMM
8	8	8	8	32

OP - Opcode Field - 8 Bits
DST - Destination LAR - 8 Bits
SRC1 - First Operand Source LAR - 8 Bits
SRC2 - Second operand source LAR - 8 Bits
IMM - Immediate value, for address calculation (Signed) - 32 Bits

addressing mode:

```
Base Addr = SRC1.Address + SRC2.Data + (IMM*WDSZ)
```


Memory Instructions

LOAD - 0b0100 + typ + wdsz
STORE - 0b0101 + typ + wdsz

Figure: LARK Memory Instruction Format

OP	DST	SRC1	SRC2	IMM
8	8	8	8	32

OP - Opcode Field - 8 Bits
DST - Destination LAR - 8 Bits
SRC1 - First Operand Source LAR - 8 Bits
SRC2 - Second operand source LAR - 8 Bits
IMM - Immediate value, for address calculation (Signed) - 32 Bits

addressing mode:
Base Addr = SRC1.Address + SRC2.Data + (IMM*WDSZ)

- Load loads and sets tags
- Store only sets tags
- A sane me would have called STORE ATTR or something
- Alignment? On LAR-width bounds only
- SRC1 Address field used as base address for load
- SRC2 Data field used for effective address calculation
- Round-Robin writeback, just stall on load to dirty.
- Sparks' LOON version used queues for data load, instruction fetch, and data writeback with a FSM to govern.

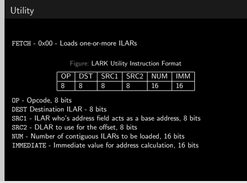
Utility

`FETCH` - 0x00 - Loads one-or-more ILARs

Figure: LARK Utility Instruction Format

OP	DST	SRC1	SRC2	NUM	IMM
8	8	8	8	16	16

`OP` - Opcode, 8 bits
`DEST` Destination ILAR - 8 bits
`SRC1` - ILAR who's address field acts as a base address, 8 bits
`SRC2` - DLAR to use for the offset, 8 bits
`NUM` - Number of contiguous ILARs to be loaded, 16 bits
`IMMEDIATE` - Immediate value for address calculation, 16 bits


Utility

FETCH - 0x00 - Loads one-or-more ILARs

Figure: LARK Utility Instruction Format

OP	DST	SRC1	SRC2	NUM	IMM
8	8	8	8	16	16

OP - Opcode, 8 bits
DEST Destination ILAR - 8 bits
SRC1 - ILAR who's address field acts as a base address, 8 bits
SRC2 - DLAR to use for the offset, 8 bits
NUM - Number of contiguous ILARs to be loaded, 16 bits
IMMEDIATE - Immediate value for address calculation, 16 bits

- The only Utility instruction
- Same addressing mode as LOAD/STORE, but with an ILAR as base
- Generated by Assembler during packing

Arithmetic

Table: LARK Arithmetic Instruction Behaviors

Instruction	Function
ADD	DST=S1+S2
SUB	DST=S1-S2
MUL	DST=S1*S2
DIV	DST=S1/S2
MOD	DST=S1%S2
AND	DST=S1&S2
OR	DST=S1|S2
XOR	DST=S1^S2
NEG	DST=~S1
SLL	DST=S1<<IMM
SRA	DST=S1>>IMM (always sign extend result)
SRL	DST=S1>>IMM
SLT	DST=S1>S2

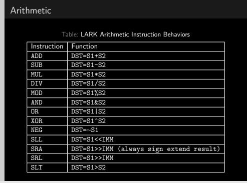

Arithmetic

Table: LARK Arithmetic Instruction Behaviors

Instruction	Function
ADD	DST=S1+S2
SUB	DST=S1-S2
MUL	DST=S1*S2
DIV	DST=S1/S2
MOD	DST=S1%S2
AND	DST=S1&S2
OR	DST=S1|S2
XOR	DST=S1^S2
NEG	DST=~S1
SLL	DST=S1<<IMM
SRA	DST=S1>>IMM (always sign extend result)
SRL	DST=S1>>IMM
SLT	DST=S1>S2

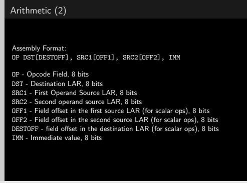
- Reasonable assortment, maps easily to useful languages
- Shifts are sketchy on some types, just doing them naively on bits

Arithmetic (2)

Assembly Format:

```
OP DST[DESTOFF], SRC1[OFF1], SRC2[OFF2], IMM
```

`OP` - Opcode Field, 8 bits
`DST` - Destination LAR, 8 bits
`SRC1` - First Operand Source LAR, 8 bits
`SRC2` - Second operand source LAR, 8 bits
`OFF1` - Field offset in the first source LAR (for scalar ops), 8 bits
`OFF2` - Field offset in the second source LAR (for scalar ops), 8 bits
`DESTOFF` - field offset in the destination LAR (for scalar ops), 8 bits
`IMM` - Immediate value, 8 bits


Arithmetic (2)

Assembly Format:
OP DST[DESTOFF], SRC1[OFF1], SRC2[OFF2], IMM

OP - Opcode Field, 8 bits
DST - Destination LAR, 8 bits
SRC1 - First Operand Source LAR, 8 bits
SRC2 - Second operand source LAR, 8 bits
OFF1 - Field offset in the first source LAR (for scalar ops), 8 bits
OFF2 - Field offset in the second source LAR (for scalar ops), 8 bits
DESTOFF - field offset in the destination LAR (for scalar ops), 8 bits
IMM - Immediate value, 8 bits

- offsets and the immediate are optional in asm, and assumed zero if not defined.

Arithmetic (3)

Table: LARK Arithmetic Instruction Encodings

Mnemonic	Encoding (Bin)	Encoding (Hex)
ADDS/ADDV	0b10?00000	0x80/0xA0
SUBS/SUBV	0b10?00001	0x81/0xA1
MULS/MULV	0b10?00010	0x82/0xA2
DIVS/DIVV	0b10?00011	0x83/0xA3
MODS/MODV	0b10?00100	0x84/0xA4
ANDS/ANDV	0b10?00101	0x85/0xA5
ORS/ORV	0b10?00110	0x86/0xA6
XORS/XORV	0b10?00111	0x87/0xA7
NOTS/NOTV	0b10?01000	0x88/0xA8
SLLS/SLLV	0b10?01001	0x89/0xA9
SRAS/SRAV	0b10?01010	0x8A/0xAA
SRLS/SRLV	0b10?01011	0x8B/0xAB
SLTS/SLTV	0b10?01100	0x8C/0xAC

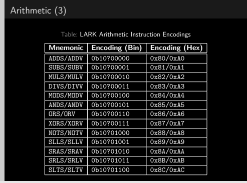

Arithmetic (3)

Table: LARK Arithmetic Instruction Encodings

Mnemonic	Encoding (Bin)	Encoding (Hex)
ADDS/ADDV	0b10?00000	0x80/0xA0
SUBS/SUBV	0b10?00001	0x81/0xA1
MULS/MULV	0b10?00010	0x82/0xA2
DIVS/DIVV	0b10?00011	0x83/0xA3
MODS/MODV	0b10?00100	0x84/0xA4
ANDS/ANDV	0b10?00101	0x85/0xA5
ORS/ORV	0b10?00110	0x86/0xA6
XORS/XORV	0b10?00111	0x87/0xA7
NOTS/NOTV	0b10?01000	0x88/0xA8
SLLS/SLLV	0b10?01001	0x89/0xA9
SRAS/SRAV	0b10?01010	0x8A/0xAA
SRLS/SRLV	0b10?01011	0x8B/0xAB
SLTS/SLTV	0b10?01100	0x8C/0xAC

- One bit set to switch from scalar operand to LAR-At-A-Time vector ops

Flow Control

Table: LARK Flow Control Instructions

Mnemonic	Encoding (Bin)	Encoding (Hex)
SEL	0b11000000	0xC0
CALL	0b11000001	0xC1
RETURN	0b11000010	0xC2

`OP` - Opcode Field, 8 bits
`COND` - Condition LAR, 8 bits
`CONDOFF` - Offset into condition LAR, 8 bits
`TGT1` - Target ILAR for nonzero condition, 8 bits
`OFF1` - Offset into nonzero target ILAR, 8 bits
`TGT2` - Target ILAR for zero condition, 8 bits
`OFF2` - Offsets into zero target ILAR, 8 bits
`PAD` - Pad bits, 8 bits

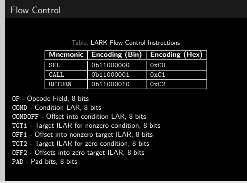

Flow Control

Table: LARK Flow Control Instructions

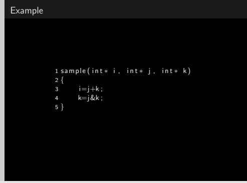
Mnemonic	Encoding (Bin)	Encoding (Hex)
SEL	0b11000000	0xC0
CALL	0b11000001	0xC1
RETURN	0b11000010	0xC2

OP - Opcode Field, 8 bits
COND - Condition LAR, 8 bits
CONDOFF - Offset into condition LAR, 8 bits
TGT1 - Target ILAR for nonzero condition, 8 bits
OFF1 - Offset into nonzero target ILAR, 8 bits
TGT2 - Target ILAR for zero condition, 8 bits
OFF2 - Offsets into zero target ILAR, 8 bits
PAD - Pad bits, 8 bits

- Select with SLT and XOR gets common general flow control constructs
- Stack is preliminarily packed into LARs, but could be skewed to multiple offsets by type
- WHY 8'b OFFSETS, 2048/64=32, only need 5b. What the hell 2010-me? Was that to maintain byte alignment?
- CALL and RETURN are unconditional to the first target, right?
- Discuss calling behavior? Just the basic "two options, pack arguments into a DLAR vs. DLAR-per-Type?"

Example

```
sample(int* i, int* j, int* k)
{
	i=j+k;
	k=j&k;
}
```

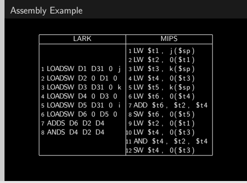

Example

```
sample(int* i, int* j, int* k)
{
	i=j+k;
	k=j&k;
}
```

●

Assembly Example

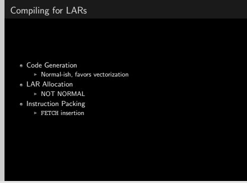
LARK	MIPS
	LW $t1 , j($sp)
	LW $t2 , 0($t1)
LOADSW D1 D31 0 j	LW $t3 , k($sp)
LOADSW D2 0 D1 0	LW $t4 , 0($t3)
LOADSW D3 D31 0 k	LW $t5 , k($sp)
LOADSW D4 0 D3 0	LW $t6 , 0($t4)
LOADSW D5 D31 0 i	ADD $t6 , $t2 , $t4
LOADSW D6 0 D5 0	SW $t6 , 0($t5)
ADDS D6 D2 D4	LW $t2 , 0($t1)
ANDS D4 D2 D4	LW $t4 , 0($t3)
	AND $t4 , $t2 , $t4
	SW $t4 , 0($t3)



- if the values are close together, the memory accesses would be a single LAR load
- we don't have to statically know if they fit in a LAR, processor will leverage the alias automatically
- Memory: 0-5 to 9
- Reads: 0-3 to 9
- Writes: 0-2 to 2
- Count: 9
- We could do a whole LAR-wide vector for free

Compiling for LARs

- Code Generation
  - Normal-ish, favors vectorization
- LAR Allocation
  - NOT NORMAL
- Instruction Packing
  - `FETCH` insertion

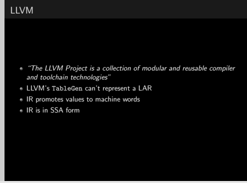

Compiling for LARs

- Code Generation
  - Normal-ish, favors vectorization
- LAR Allocation
  - NOT NORMAL
- Instruction Packing
  - `FETCH` insertion

- And continuation tails for ILARs

LLVM

- *“The LLVM Project is a collection of modular and reusable compiler and toolchain technologies”*
- LLVM’s `TableGen` can’t represent a LAR
- IR promotes values to machine words
- IR is in SSA form


LLVM

- *“The LLVM Project is a collection of modular and reusable compiler and toolchain technologies”*
- LLVM’s `TableGen` can’t represent a LAR
- IR promotes values to machine words
- IR is in SSA form

- LLVM used to be ”Low Level Virtual Machine” but they de-acronymed years ago
- TableGen is their architecture description tool, used for instructions and registers
- IR promotes all things to machine word.
- Didn’t realize SSA was undesirable at the time
- No SSA form, PREFER reuse of locations

After LLVM Fell Through

- Specified LARc
- Wrote ANTLR grammar
- Researched ancestor technologies for inspiration
- A diversion with LARKem

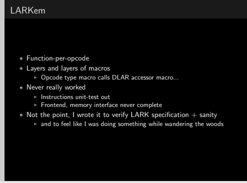
After LLVM Fell Through

- Specified LARc
- Wrote ANTLR grammar
- Researched ancestor technologies for inspiration
- A diversion with LARKem

- ANother Tool for Language Recognition
- Used antlr 2 out of familarity and C support

LARKem

- Function-per-opcode
- Layers and layers of macros
  - Opcode type macro calls DLAR accessor macro...
- Never really worked
  - Instructions unit-test out
  - Frontend, memory interface never complete
- Not the point, I wrote it to verify LARK specification + sanity
  - and to feel like I was doing something while wandering the woods


LARKem

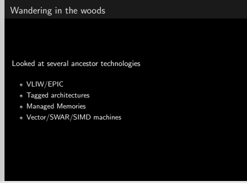
- Function-per-opcode
- Layers and layers of macros
  - Opcode type macro calls DLAR accessor macro...
- Never really worked
  - Instructions unit-test out
  - Frontend, memory interface never complete
- Not the point, I wrote it to verify LARK specification + sanity
  - and to feel like I was doing something while wandering the woods

- Also to feel like I was accomplishing something while wandering in the woods

Wandering in the woods

Looked at several ancestor technologies

- VLIW/EPIC
- Tagged architectures
- Managed Memories
- Vector/SWAR/SIMD machines



Wandering in the woods

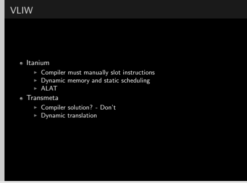
Looked at several ancestor technologies

- VLIW/EPIC
- Tagged architectures
- Managed Memories
- Vector/SWAR/SIMD machines

- also scratchpads? We do have that non-uniform addressed memory thing.
- also decoupling efforts like the CSPI MAP 200?
- Largely "efforts to cheat the VonNeumann Bottleneck"

VLIW

- Itanium
  - Compiler must manually slot instructions
  - Dynamic memory and static scheduling
  - ALAT
- Transmeta
  - Compiler solution? - Don't
  - Dynamic translation

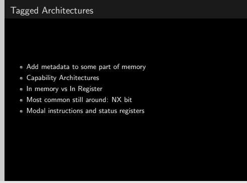

VLIW

- Itanium
  - Compiler must manually slot instructions
  - Dynamic memory and static scheduling
  - ALAT
- Transmeta
  - Compiler solution? - Don't
  - Dynamic translation

- Itanium announced 1989, delivered 2001. Bad times.
- Abandonment of (DEC/COMPAQ) Alpha, (SGI) MIPS, (HP) PA-RISC to pile on
- Itanic.
- ALAT = Advanced Load Address Table, most akin to a LAR in production
- ld.a instructions to preload into associative memory, with address tags, test with ld.c
- There never was a native compiler for Transmeta, just CMT (Code Morphing Software)
- Their internal arch didn't even have a MMU
- Eventually everyone did this by fancy reassembly "microcoding" onto multi-issue pipelines to schedule around memory
- First lesson: Dynamic Memory + Static Scheduling = Fail
- Upon reflection: We can KEEP THINGS STATIC

Tagged Architectures

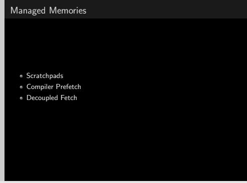
- Add metadata to some part of memory
- Capability Architectures
- In memory vs In Register
- Most common still around: NX bit
- Modal instructions and status registers


Tagged Architectures
- Add metadata to some part of memory
- Capability Architectures
- In memory vs In Register
- Most common still around: NX bit
- Modal instructions and status registers

- Large penalty for extra memory traffic
- Lisp Machines: MIT, 1981 used 2+ extra bits per machine word for type tags
- iAPX 432:Intel, 1981 Used 128b object descriptors 32b access descriptors
- Another epic fail by Intel
- Co-designed with languages
- Set IEEE 754 Rounding mode
- RISC V vector extension uses configuration registers for vector shaping

Managed Memories

- Scratchpads
- Compiler Prefetch
- Decoupled Fetch

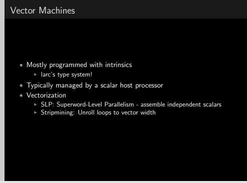

Managed Memories

- Scratchpads
- Compiler Prefetch
- Decoupled Fetch

- Scratchpads are an extra address space: non-portable with different size
- Ex: ClearSpeed, Cell Broadband Engine, CUDA Cores "Shared Memory"
- Compiler-Guided prefetch: Stomping via. Associativity, optional, often ignored
- Explicit prefetch, locking lines, etc.
- Decoupled Fetch: CSPI MAP200; Access (memory) Processor feeding queues to + execute processor

Vector Machines

- Mostly programmed with intrinsics
  - larc's type system!
- Typically managed by a scalar host processor
- Vectorization
  - SLP: Superword-Level Parallelism - assemble independent scalars
  - Stripmining: Unroll loops to vector width

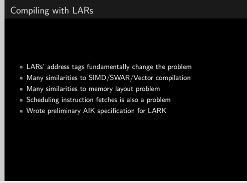

Vector Machines

- Mostly programmed with intrinsics
  - larc's type system!
- Typically managed by a scalar host processor
- Vectorization
  - SLP: Superword-Level Parallelism - assemble independent scalars
  - Stripmining: Unroll loops to vector width

- (st)ardent Titan used MIPS, GPUs use PCs, etc.
- SIMD as the current dominant species
- This was actually useful
- Common elements: Greedy algorithms

Compiling with LARs

- LARs' address tags fundamentally change the problem
- Many similarities to SIMD/SWAR/Vector compilation
- Many similarities to memory layout problem
- Scheduling instruction fetches is also a problem
- Wrote preliminary AIK specification for LARK

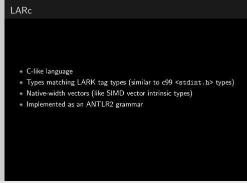

Compiling with LARs

- LARs' address tags fundamentally change the problem
- Many similarities to SIMD/SWAR/Vector compilation
- Many similarities to memory layout problem
- Scheduling instruction fetches is also a problem
- Wrote preliminary AIK specification for LARK

- Instruction fetches is a problem for the assembler
- "Assembler Interpreter from Kentucky"
- AIK isn't fancy enough to do this on its own, but it's a start
- No constant pooling, doesn't have facility for insertion, etc.
- Does know about multiple segments.

LARc

- C-like language
- Types matching LARK tag types (similar to c99 `<stdint.h>` types)
- Native-width vectors (like SIMD vector intrinsic types)
- Implemented as an ANTLR2 grammar



LARc

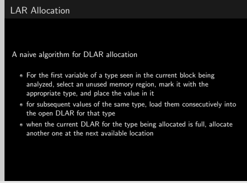
- C-like language
- Types matching LARK tag types (similar to c99 `<stdint.h>` types)
- Native-width vectors (like SIMD vector intrinsic types)
- Implemented as an ANTLR2 grammar

- Generating an AST so I could start playing with walkers to generate
- ex. int32t, uintvec16x128

LAR Allocation

A naive algorithm for DLAR allocation

- For the first variable of a type seen in the current block being analyzed, select an unused memory region, mark it with the appropriate type, and place the value in it
- for subsequent values of the same type, load them consecutively into the open DLAR for that type
- when the current DLAR for the type being allocated is full, allocate another one at the next available location


LAR Allocation

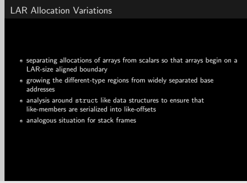
A naive algorithm for DLAR allocation

- For the first variable of a type seen in the current block being analyzed, select an unused memory region, mark it with the appropriate type, and place the value in it
- for subsequent values of the same type, load them consecutively into the open DLAR for that type
- when the current DLAR for the type being allocated is full, allocate another one at the next available location

- Basically, segment per type
- Some similarities to Linear Scan allocators (orig. 1999 Poletto/Sarkar) - spent a lot of time trying to map their algorithms to my problem
- No NP-hard problems like graph coloring

LAR Allocation Variations

- separating allocations of arrays from scalars so that arrays begin on a LAR-size aligned boundary
- growing the different-type regions from widely separated base addresses
- analysis around `struct` like data structures to ensure that like-members are serialized into like-offsets
- analogous situation for stack frames


LAR Allocation Variations

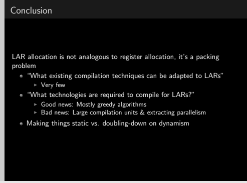
- separating allocations of arrays from scalars so that arrays begin on a LAR-size aligned boundary
- growing the different-type regions from widely separated base addresses
- analysis around `struct` like data structures to ensure that like-members are serialized into like-offsets
- analogous situation for stack frames

- All these things are to encourage natural packing/extract parallelism.
- LARs associative copy for same base address, so mixed-type data is OK (just use 2 names)

Conclusion

LAR allocation is not analogous to register allocation, it's a packing problem

- "What existing compilation techniques can be adapted to LARs"
  - Very few
- "What technologies are required to compile for LARs?"
  - Good news: Mostly greedy algorithms
  - Bad news: Large compilation units & extracting parallelism
- Making things static vs. doubling-down on dynamism



Conclusion

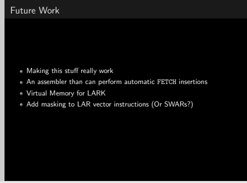
LAR allocation is not analogous to register allocation, it's a packing problem

- "What existing compilation techniques can be adapted to LARs"
  - Very few
- "What technologies are required to compile for LARs?"
  - Good news: Mostly greedy algorithms
  - Bad news: Large compilation units & extracting parallelism
- Making things static vs. doubling-down on dynamism

- Speculation attacks and timing attacks aren't possible on static machines.
- Dynamic means lots of active circuitry consuming power.

Future Work

- Making this stuff really work
- An assembler than can perform automatic `FETCH` insertions
- Virtual Memory for LARK
- Add masking to LAR vector instructions (Or SWARs?)


Future Work
- Making this stuff really work
- An assembler than can perform automatic FETCH insertions
- Virtual Memory for LARK
- Add masking to LAR vector instructions (Or SWARs?)

- Virtual Memory: Physical tag v.s. Virtual tag: v-tag potential aliasing, p-tags would have to have TLB lookups every time a tag is touched
- I can't build a fancy parallelizing, analyzing compiler solo (For an MS or at all, millions of man-hours), so stuck by disjoint from early tools.