Part 4: Diablo Backends

- Object formats
- Retargetability
- Linkers
- Architectures

Overal design for retargetability

Object file backend

Linker map & linker script

Code
Data
Meta

Code
Data
Meta

Code
Data
Code
Data
Code
Data

Architecture Backend

Data
Data
Data

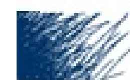
Prog

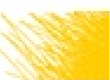
EXE Backend

Layout script & Layout code

Object file backends

- Goal: dismantle object files and present them in Diablo's internal graph representation:
  - put raw data from sections into t_section objects
  - extract the section flags from the object file headers
- Mostly architecture-independent
- Exception: architecture-specific relocation types
  - depends by and large on how addresses are produced
  - specified in ABI documents
- Modules: object_backends/<file format>/



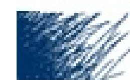
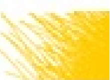


Example

- ElfReadCommon32()
  - reads an ELF file
  - extracts headers, symbol table, string table, ...
- ElfRead<arch>SameEndian()
  - calls ElfReadCommon32()
  - reads <arch>-specific sections: relocation info, special sections, specific flags, ...

Existing object file backends

- ELF
  - actively maintained
- ECOFF (Alpha Tru64Unix)
  - will be revived in near future
- COFF
  - Texas Instruments variant will arrive soon

Overal design for retargetability

Object file backend

Linker map & linker script

Code

Data

Meta

Code

Data

Meta

Code

Data

Code

Data

Code

Data

Architecture Backend

Data

Data

Data

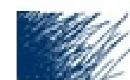
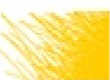
Prog

EXE Backend

Layout script & Layout code

EXE backends

- GOAL: pack Diablo sections in platform-specific executable format
- Practical: the inverse of the object file backend
- Example:
  - ElfWriteCommonSameEndian()
  - ElfWrite<arch>SameEndian()
- Modules: object_backends/<file format>/

Overal design for retargetability

Object file backend

Linker map & linker script

Code
Data
Meta

Code
Data
Meta

Code
Data
Code
Data
Code
Data

Architecture Backend

Data
Data
Data

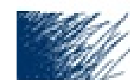
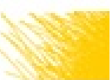
Prog

EXE Backend

Layout script & Layout code

Linker backends

- GOAL: mimick how native linker links
- 3 parts
  - Linker map file parser
    - Needs to be implemented in C and plugged-in
  - Linker script
    - Specifies symbols and sections created by the linker
    - Specifies the values of linker-generated symbols
    - Diablo has its own script language
  - Layout script
    - Specifies the placement of sections in final program

Example: GNU ld linker script

```
.eh_frame_hdr : { *(.eh_frame_hdr) }

. = ALIGN (0x1000) - ((0x1000 - .) & (0x1000 - 1));

PROVIDE (__preinit_array_start = .);
.preinit_array     : { *(.preinit_array) }
PROVIDE (__preinit_array_end = .);
PROVIDE (__init_array_start = .);
.init_array     : { *(.init_array) }
PROVIDE (__init_array_end = .);
```

1) Combine object file sections into program sections

Example: GNU ld linker script

```
.eh_frame_hdr : { *(.eh_frame_hdr) }

. = ALIGN (0x1000) - ((0x1000 - .) & (0x1000 - 1));

PROVIDE (__preinit_array_start = .);
.preinit_array     : { *(.preinit_array) }
PROVIDE (__preinit_array_end = .);
PROVIDE (__init_array_start = .);
.init_array     : { *(.init_array) }
PROVIDE (__init_array_end = .);
```

1) Combine object file sections into program sections
2) Add symbols to combined sections

Example: GNU ld linker script

```
.eh_frame_hdr : { *(.eh_frame_hdr) }

. = ALIGN (0x1000) - ((0x1000 - .) & (0x1000 - 1));

PROVIDE (__preinit_array_start = .);
.preinit_array     : { *(.preinit_array) }
PROVIDE (__preinit_array_end = .);
PROVIDE (__init_array_start = .);
.init_array     : { *(.init_array) }
PROVIDE (__init_array_end = .);
```

1) Combine object file sections into program sections
2) Add symbols to combined sections
3) Add alignment between sections

Example: GNU ld linker script

```
.eh_frame_hdr : { *(.eh_frame_hdr) }

. = ALIGN (0x1000) - ((0x1000 - .) & (0x1000 - 1));

PROVIDE (__preinit_array_start = .);
.preinit_array     : { *(.preinit_array) }
PROVIDE (__preinit_array_end = .);
PROVIDE (__init_array_start = .);
.init_array     : { *(.init_array) }
PROVIDE (__init_array_end = .);
```

1) To which sections do these symbols refer?

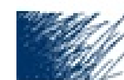
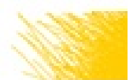
Example: GNU ld linker script

```
.eh_frame_hdr : { *(.eh_frame_hdr) }

. = ALIGN (0x1000) - ((0x1000 - .) & (0x1000 - 1));

PROVIDE (__preinit_array_start = .);
.preinit_array      : { *(.preinit_array) }
PROVIDE (__preinit_array_end = .);
PROVIDE (__init_array_start = .);
.init_array      : { *(.init_array) }
PROVIDE (__init_array_end = .);
```

1) To which sections do these symbols refer?
2) Do the combined sections respect our assumption that a section is only reacable via a symbol/relocation?

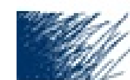
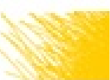



Example: GNU ld linker script

```
.eh_frame_hdr : { *(.eh_frame_hdr) }

. = ALIGN (0x1000) - ((0x1000 - .) & (0x1000 - 1));

PROVIDE (__preinit_array_start = .);
.preinit_array     : { *(.preinit_array) }
PROVIDE (__preinit_array_end = .);
PROVIDE (__init_array_start = .);
.init_array     : { *(.init_array) }
PROVIDE (__init_array_end = .);
```

More expressive Diablo linker script

```
InitVector {
   action  { VECTORIZE(".init_array") }
   trigger { SUBSECTION_EXISTS("*",".init_array") }
}

InitStart {
 action  { ADD_SYMBOL("__init_array_start") }
 trigger { UNDEFINED_SYMBOL("__init_array_start") &&
           SUBSECTION_EXISTS("Linker",
                             "VECTOR___.init_array") }
 symbol  { START_OF_SUBSECTION("Linker",
                              "VECTOR___.init_array") }
```

1) Combine object file sections into program sections, specifiying VECTOR property of combined sections.

More expressive Diablo linker script

```
InitVector {
   action  { VECTORIZE(".init_array") }
   trigger { SUBSECTION_EXISTS("*",".init_array") }
}

InitStart {
 action  { ADD_SYMBOL("__init_array_start") }
 trigger { UNDEFINED_SYMBOL("__init_array_start") &&
           SUBSECTION_EXISTS("Linker",
                             "VECTOR___.init_array") }
 symbol  { START_OF_SUBSECTION("Linker",
                               "VECTOR___.init_array") }
```

1) Combine object file sections into program sections, specifiying VECTOR property of combined sections.
2) More expressive addition of symbol.

Diablo Layout script

```
.eh_frame_hdr : { *(.eh_frame_hdr) }

. = ALIGN (0x1000) - ((0x1000 - .) & (0x1000 - 1));

PROVIDE (__preinit_array_start = .);
.preinit_array     : { *(.preinit_array) }
PROVIDE (__preinit_array_end = .);
PROVIDE (__init_array_start = .);
.init_array     : { *(.init_array) }
PROVIDE (__init_array_end = .);
```

Is based on GNU ld linker scripts with unnecessary things removed.

Overal design for retargetability

Object file backend

Linker map & linker script

Code
Data
Meta

Code
Data
Meta

Code
Data
Code
Data
Code
Data

Architecture Backend

Data
Data
Data

Prog

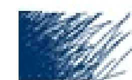
EXE Backend

Layout script & Layout code

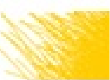
Architecture Backend: 4 steps

Link	Assembly	Flowgraph	Transform
Read & link			
	Disassemble		
		Create ICFG	
			Transform CFG
		Layout ICFG	
	Assemble		
Write			

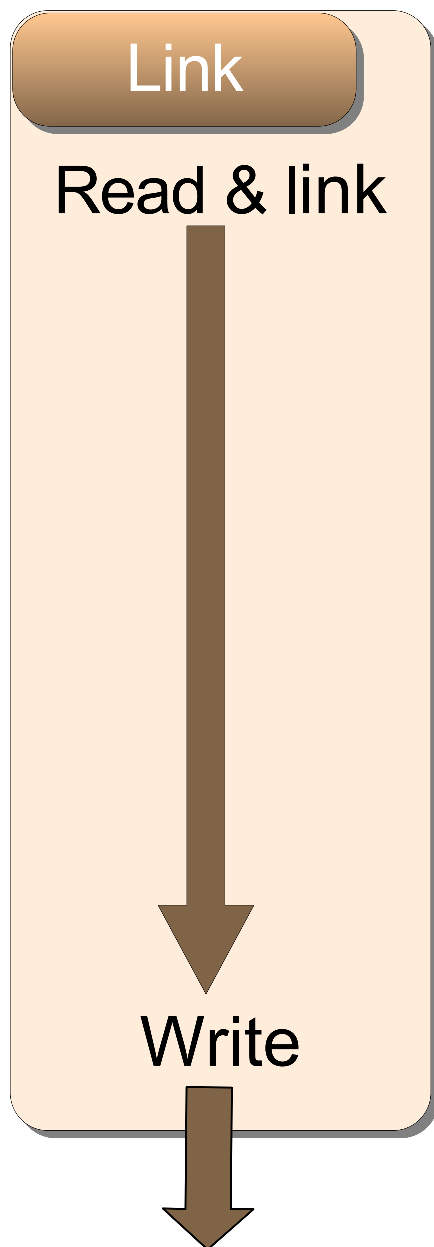
test & debug



UNIVERSITEIT GENT

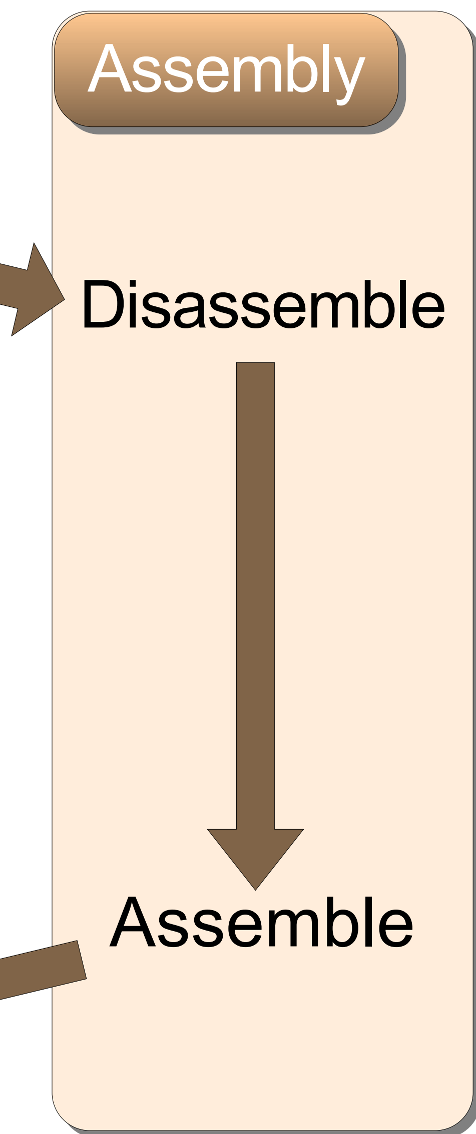


Step 1: linking



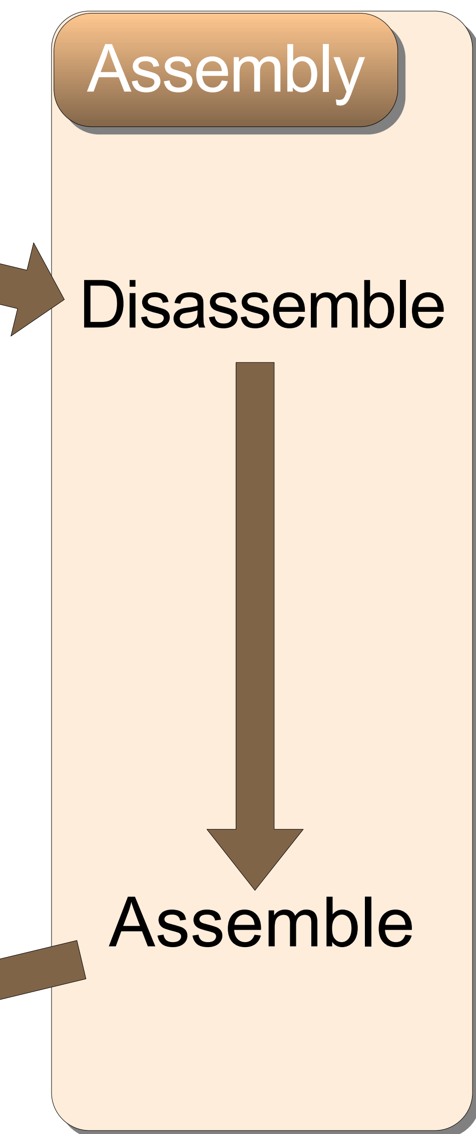
- You implement
  - Object format, EXE and linker backend
  - Relocations for your architecture
- Diablo
  - Links the program
    - Using the relocations
    - Using the linker script
  - Checks against original program
  - Linked program is written
    - Using the layout script and exe backend

Step 2: Assembling & Disassembling



- You implement
  - Instruction assembler/disassembler callback functions
  - t_architecture_description
    - min/max instruction_size (CISC)
    - #ins per bundle (IA64)
    - 32 or 64 bit,
    - # registers,
    - string array of register names,
    - pointers to callback functions,
    - instruction print function
  - Module: architecture_backends/<arch>/diablo<arch>_*.[ch]

Step 2: Assembling & Disassembling



- Diablo
  - Disassembles and assembles
  - Can generate a listing of your program
    - Using Diablo's iterators
    - and output facilities with the @I format specifier

Step 3: ICFG construction and layout

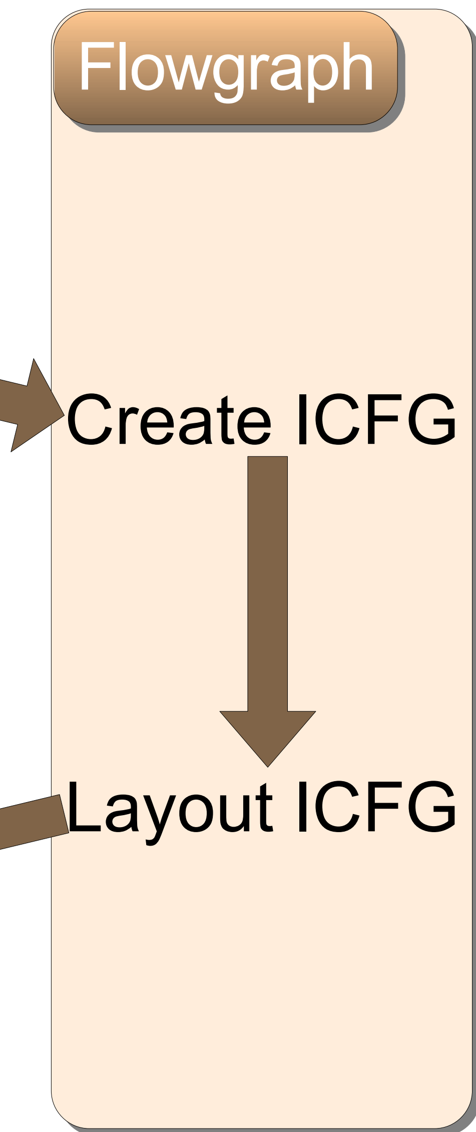
Flowgraph

Create ICFG

Layout ICFG

- You implement
  - basic block leader detection
    - control flow + relocations
  - add edges between blocks
  - further graph refinements
    - switch/pattern detection
  - Layout algorithm that linearizes the bbl's
  - Module: architecture_backends/<arch>/diablo<arch>_*.[ch]

Step 3: ICFG construction and layout



- Diablo
  - Constructs the ICFG (DCFRAG already done)
  - ICFG can be inspected with LANCET
  - Performs a code & data layout again
    - Linearization of the graphs

- Mind you:

  THIS IS THE HARDEST PART IN CREATING NEW ARCHITECTURE BACKENDS

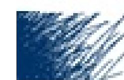
UNIVERSITEIT GENT

Step 4: Analyses and Transformations

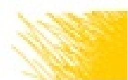
Transform

Transform ICFG

- You implement
  - t_architecture_description:
    - calling conventions information
    - compiler built-in functions
  - INS_REGS_USE, INS_REGS_DEF for register liveness analysis
  - (limited) instruction emulator for constant propagation
  - your own analysis/optimizations
- Diablo
  - Rewrites your binary programs



UNIVERSITEIT GENT



Address Producers

OBSERVATIONS

- Addresses and relocations are propagatad during constant propagation
  - For identifying addresses from which constant data can be loaded.
  - For detecting constant offsets between two pointers

- Otherwise, addresses have no real meaning in the AWPCFG
- What to do with instructions who's only purpose is to generate an address?

Address Producers

OBSERVATIONS

- On some (x86) architectures, one instruction suffices.
- Others (ARM) require a sequence of instructions
- Most efficient one depends on
  - Address to be produced
  - Position in the program if PC-relative addressing is possible

Address Producers

CONCLUSION

- Keeping real instructions that produce addresses in the ICFG makes no sense

SOLUTION

- Introduce a virtual instruction of type ADDRESS_PRODUCER in ICFG.
- Replace this by real instructions during code layout
- Similar with CONST_PRODUCERS