A generic approach to the definition of low-level components for multi-architecture binary analysis

Cédric Valensi

PhD advisor: William Jalby

University of Versailles Saint-Quentin-en-Yvelines, France
Exascale Computing Research Center, France
LRC ITACA, France

July 2nd, 2014

LRC IT@CA

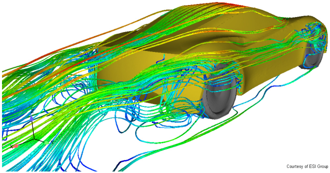
High Performance Computing

Supercomputers

- Front-line of the computing capacity
- Multiprocessor systems
- Current top speed 33 Petaflop/s

Applications

- Physical simulations
- Natural resources exploration
- Molecular modeling
- Weather forecasts



Performance analysis for optimisation

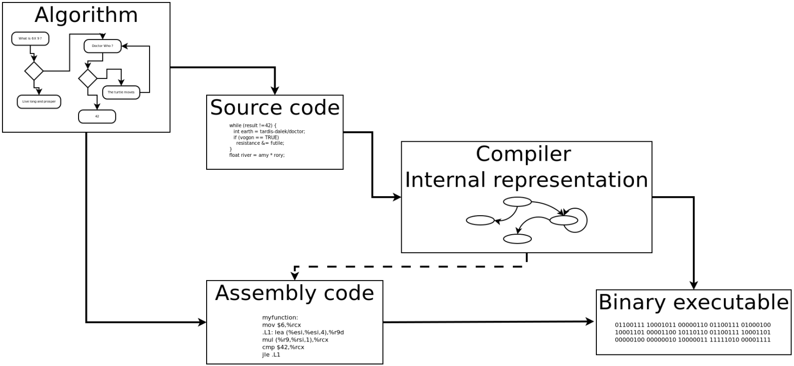
Optimising performance of HPC applications

- Optimise use of processors in terms of speed and power
- Pinpoint bottlenecks
- Estimate gain from improvements

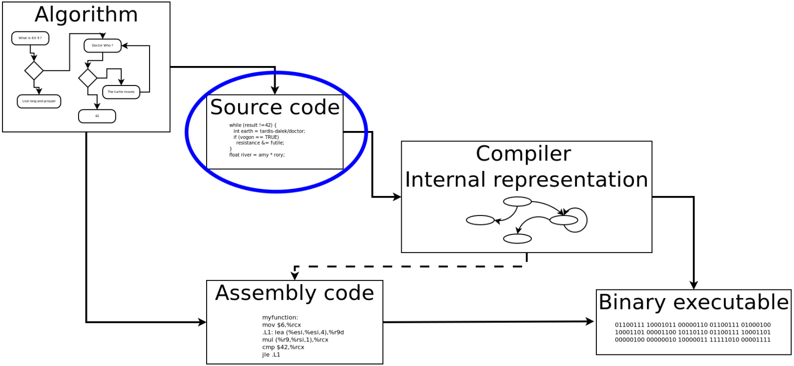
Performance analysis

- Static or dynamic
- Instrumentation
- Possible in all steps of the design process

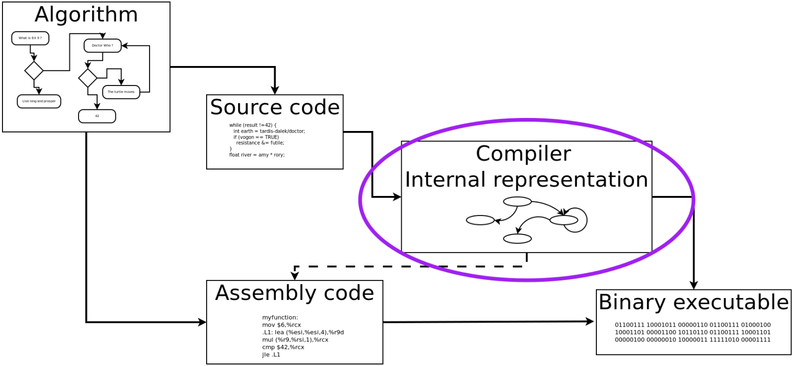
Steps of an application design process



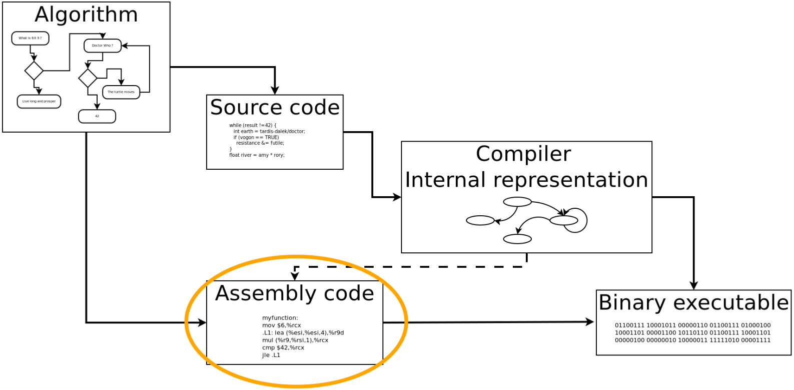
Steps of an application design process



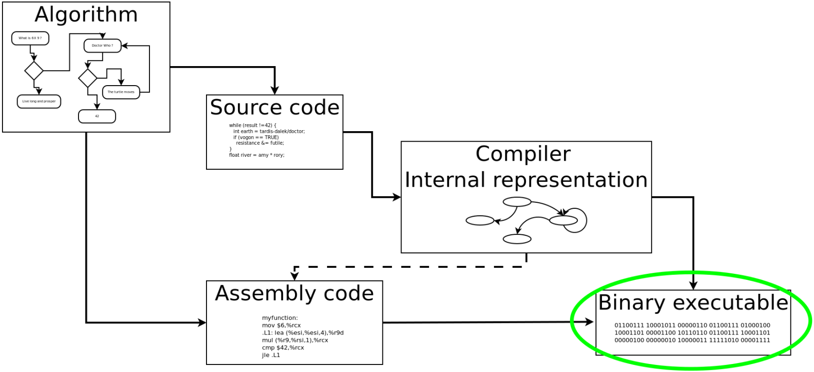
Steps of an application design process



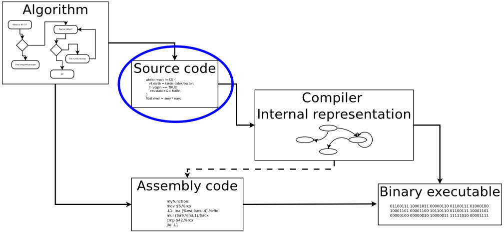
Steps of an application design process



Steps of an application design process



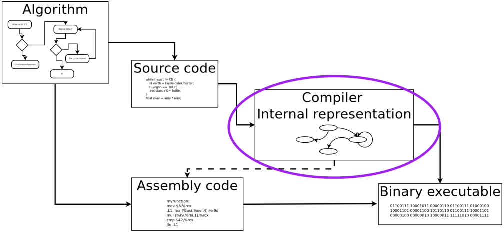
Performance analysis levels



Source code

- Knowledge of source language
- Requires access to source files
- Compilation may perform complex transformations
- Instrumenting at the source level may modify these transformations

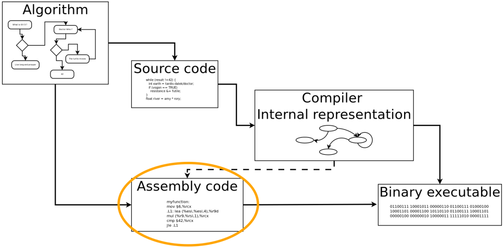
Performance analysis levels



Compiler Internal Representation

- More accurate
- Requires access to compiler internals
- Requires intrusion into compilation process
- Ineffective for code written in assembly

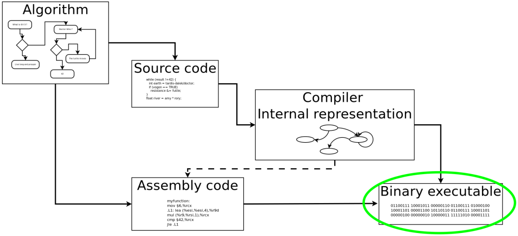
Performance analysis levels



Assembly analysis

- Closer to the actual executable
- Not available by default
- Requires intrusion into compilation process

Performance analysis levels



Binary analysis

- “What you see is what you run”
- Allows to retrieve additional information
- More complex

Challenges of binary analysis

Dependent on the architecture

- Multiple architectures may be used by a single application
- Binary architectures evolve frequently

Static Analysis

- Requires disassembly of binary code

Instrumentation

- Requires static or dynamic patching
- Extensive changes can be needed

Contribution

Low level binary encoder and decoder

- Able to support multiple architectures
- Minimised implementation workload

Usage in analysis context

- Customisable behaviour
- Unified output format
- Acceptable performance
- Static analysis and instrumentation

Outline

Objectives

Generic encoder and decoder

- Multi-architecture support
- Customisable output and behaviour
- Reduced implementation workload

Challenges

- Complex binary coding rules
- Coding rules and assembly vary significantly between architectures
- Avoid hard coding

Example: Encoding of an Intel 64 instruction

0x66 49 89 4C 90 20 <=> mov %r9, 0x20(%eax,%edx,4)

Example: Encoding of an Intel 64 instruction

0x66 49 89 4C 90 20 <=> mov %r9, 0x20(%eax,%edx,4)

01100111 01001100 10001001 01001100 10010000 00100000

mov %r9,0x20 (%eax,%edx,4)

Example: Encoding of an Intel 64 instruction

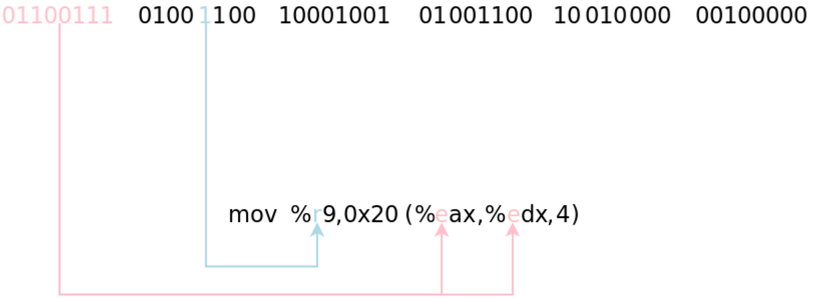
0x66 49 89 4C 90 20 <=> mov %r9, 0x20(%eax,%edx,4)

01100111 01001100 10001001 01001100 10010000 00100000

mov %r9,0x20 (%eax,%edx,4)

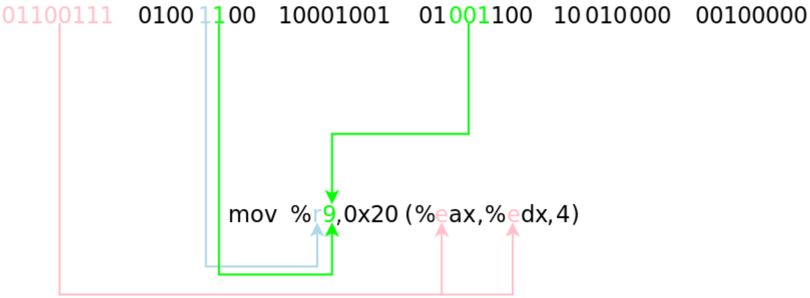
Example: Encoding of an Intel 64 instruction

0x66 49 89 4C 90 20 <=> mov %r9, 0x20(%eax,%edx,4)



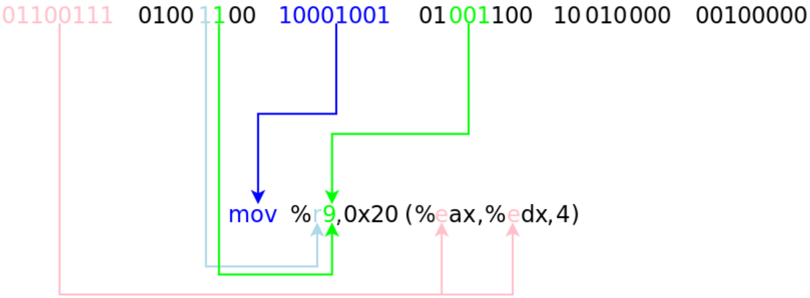
Example: Encoding of an Intel 64 instruction

0x66 49 89 4C 90 20 <=> mov %r9, 0x20(%eax,%edx,4)



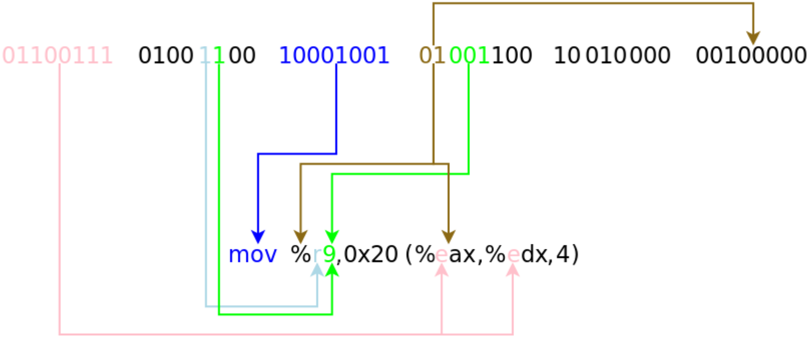
Example: Encoding of an Intel 64 instruction

0x66 49 89 4C 90 20 <=> mov %r9, 0x20(%eax,%edx,4)



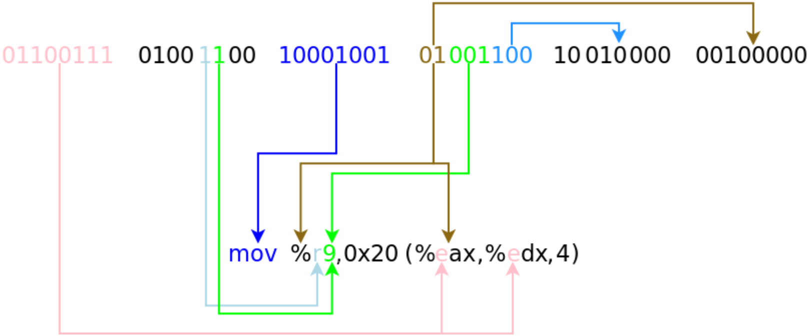
Example: Encoding of an Intel 64 instruction

0x66 49 89 4C 90 20 <=> mov %r9, 0x20(%eax,%edx,4)

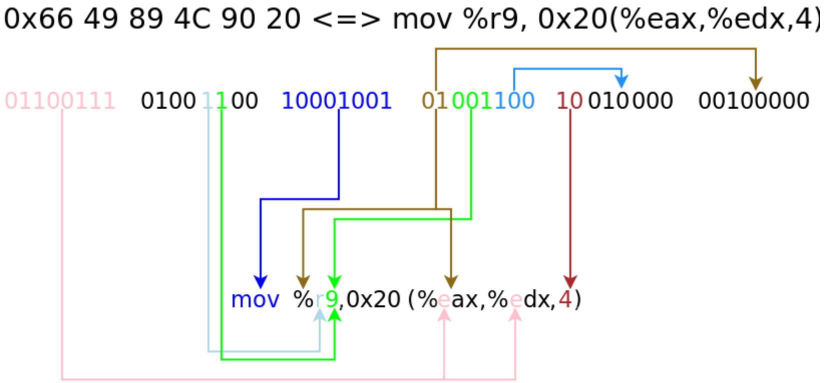


Example: Encoding of an Intel 64 instruction

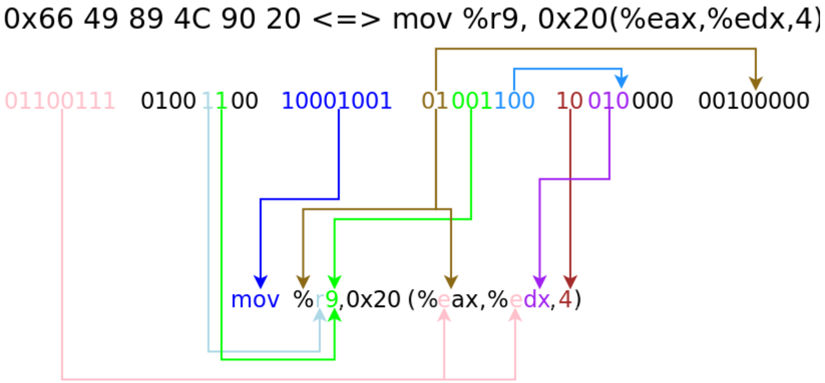
0x66 49 89 4C 90 20 <=> mov %r9, 0x20(%eax,%edx,4)



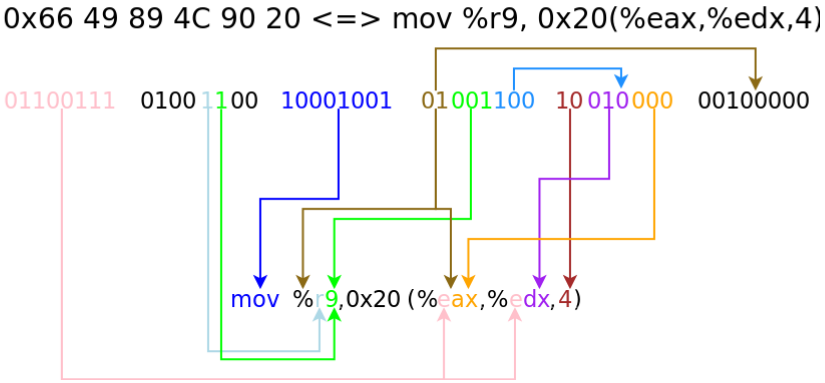
Example: Encoding of an Intel 64 instruction



Example: Encoding of an Intel 64 instruction

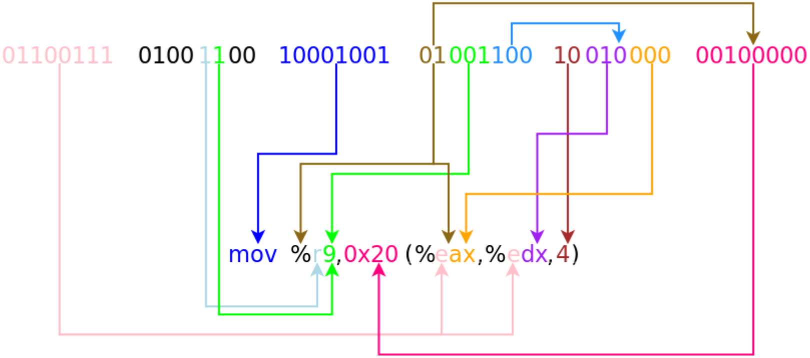


Example: Encoding of an Intel 64 instruction



Example: Encoding of an Intel 64 instruction

0x66 49 89 4C 90 20 <=> mov %r9, 0x20(%eax,%edx,4)



Example: Encoding of an ARM instruction

0x15 2D 40 05 <=> strne r4, [sp, #-5]!

Example: Encoding of an ARM instruction

0x15 2D 40 05 <=> strne r4, [sp, #-5]!

00010101 00101101 01000000 00000101

strne r4 ,[sp ,#-5]!

Example: Encoding of an ARM instruction

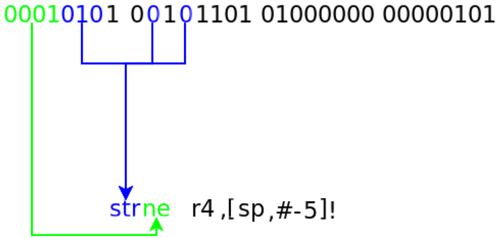
0x15 2D 40 05 <=> strne r4, [sp, #-5]!

00010101 00101101 01000000 00000101

strne r4 ,[sp,#-5]!

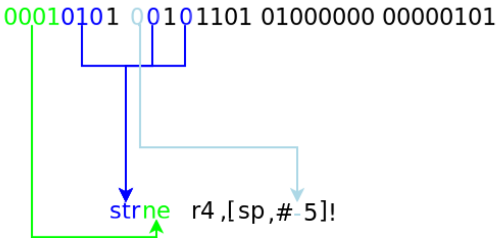
Example: Encoding of an ARM instruction

0x15 2D 40 05 <=> strne r4, [sp, #-5]!

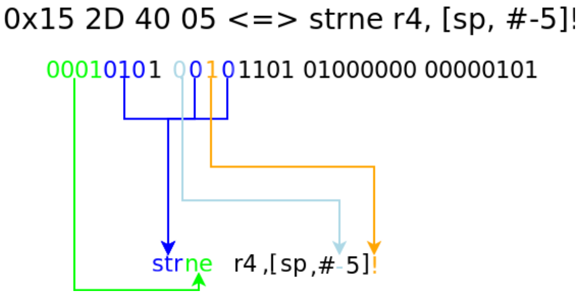


Example: Encoding of an ARM instruction

0x15 2D 40 05 <=> strne r4, [sp, #-5]!

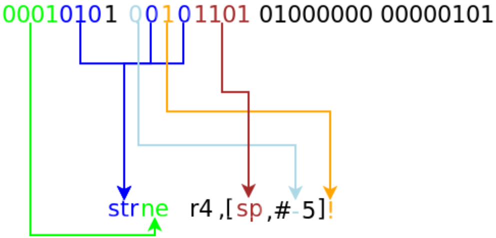


Example: Encoding of an ARM instruction



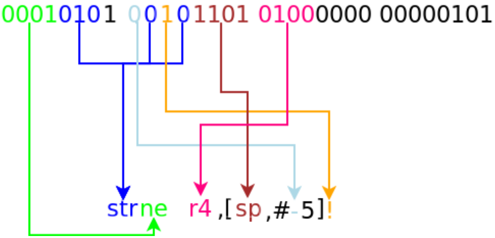
Example: Encoding of an ARM instruction

0x15 2D 40 05 <=> strne r4, [sp, #-5]!

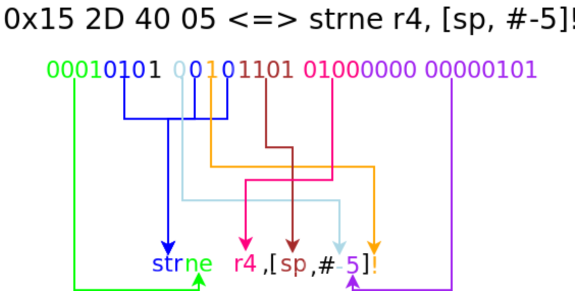


Example: Encoding of an ARM instruction

0x15 2D 40 05 <=> strne r4, [sp, #-5]!



Example: Encoding of an ARM instruction



Requirements

Ensuring agnosticism with regard to architecture

- Unified representation of an architecture encoding rules
- Decorrelation of decoding from post parsing actions
- Same representation to generate encoder and decoder

Remaining close to the documentation format

- Handling exclusions and restricted cases
- Possibility of fields with no fixed value

Using a context-free grammar formalism

Advantages

- Allows to decorrelate the encoding rules from the actions performed
- Decoder implemented as the corresponding parser
- Multiple possible uses for the decoder
- Encoder built from the same grammar

Challenges

- Grammars usually operate at the character level
- Using a bit by bit parsing would be inefficient
- Lookahead challenged by instructions of variable sizes

Standard notions

Context free grammars

- Symbols associated to list of productions
- A production contains terminal and nonterminal symbols
- Terminal symbols have no production
- Semantic actions associated to productions

LR parsers

- Processing left to right
- Bottom-up matching
- Implemented as finite state automata
- Shift and reduction states

Our algorithm for parser generation

New principles

- Bits can have a fixed or unfixed value
- Terminals are defined as groups of bits
- A state represents the matching of bits anywhere in the production
- Transitions over terminals can include bits ahead of the parsing step
- Shift/reduce states are authorised

Parser execution

- Processing left to right
- Terminals containing less unfixed bits are tested first

Example: Context free grammar

```
%token <2> d
Start:
A 00
|B 01
;
A:
C
|0111
;
B:
0111
|d 11
;
C:
00 d
|0000
;
```

Example: Context free grammar

```
%token <2> d
Start:
A 00
|B 01
;
A:
C
|0111
;
B:
0111
|d 11
;
C:
00 d
|0000
;
```

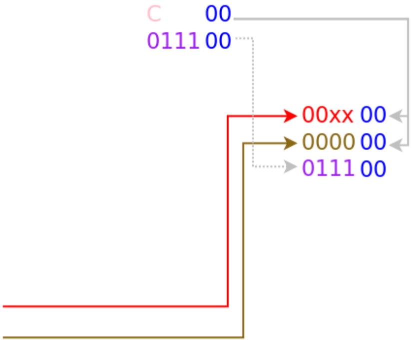
Example: Context free grammar

```
%token <2> d
Start:
A 00
|B 01
;
A:
C
|0111
;
B:
0111
|d 11
;
C:
00 d
|0000
;
```

```
C    00
0111 00
```

Example: Context free grammar

%token <2> d
Start:
A 00
|B 01
;
A:
C
|0111
;
B:
0111
|d 11
;
C:
00 d
|0000
;



Example: Context free grammar

```
%token <2> d
Start:
A 00
|B 01
;
A:
C
|0111
;
B:
0111
|d 11
;
C:
00 d
|0000
;
```

00xx 00
0000 00
0111 00

Example: Context free grammar

```
%token <2> d
Start:
A 00
|B 01
;
A:
C
|0111
;
B:
0111
|d 11
;
C:
00 d
|0000
;
```

00xx 00
0000 00
0111 00
0111 01
xx11 01

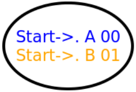
Example: Context free grammar

```
%token <2> d
Start:
A 00
|B 01
;
A:
C
|0111
;
B:
0111
|d 11
;
C:
00 d
|0000
;
```

```
00xx 00
0000 00
0111 00
0111 01
xx11 01
```

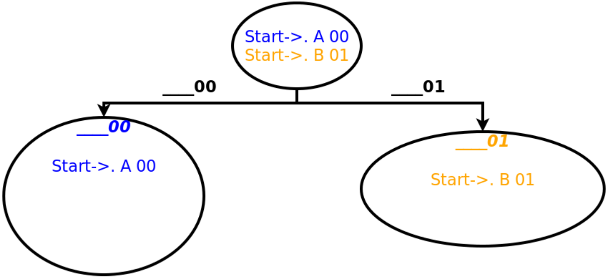
Example: Parser Generation from grammar

```
%token <2> d
Start:
A 00
| B 01
;
A:
C
| 0111
;
B:
0111
| d 11
;
C:
00 d
| 0000
;
```



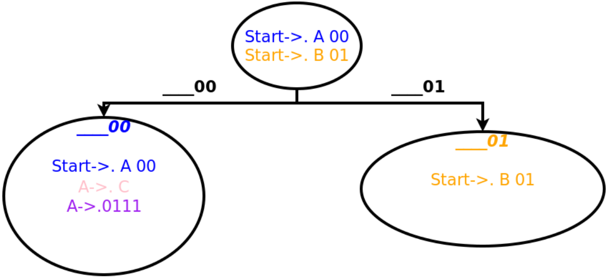
Example: Parser Generation from grammar

```
%token <2> d
Start:
A 00
| B 01
;
A:
C
| 0111
;
B:
0111
| d 11
;
C:
00 d
| 0000
;
```



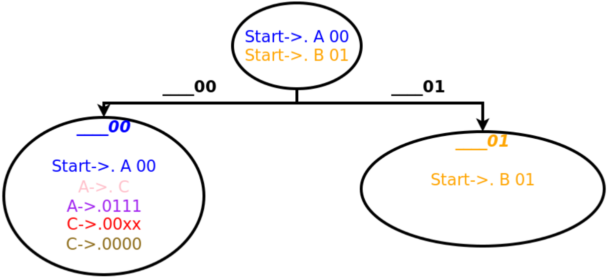
Example: Parser Generation from grammar

%token <2> d
Start:
A 00
| B 01
;
A:
C
| 0111
;
B:
0111
| d 11
;
C:
00 d
| 0000
;



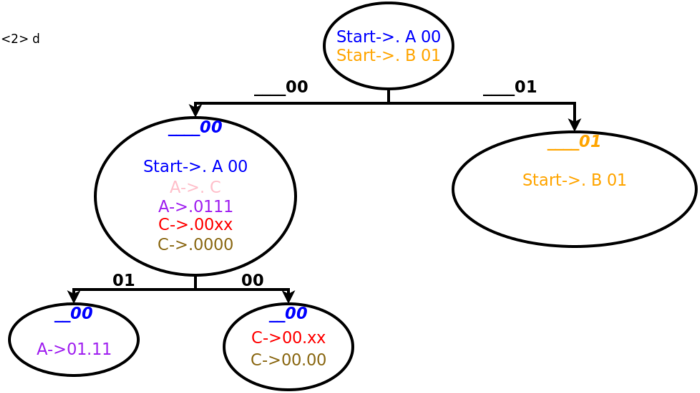
Example: Parser Generation from grammar

```
%token <2> d
Start:
A 00
| B 01
;
A:
C
| 0111
;
B:
0111
| d 11
;
C:
00 d
| 0000
;
```



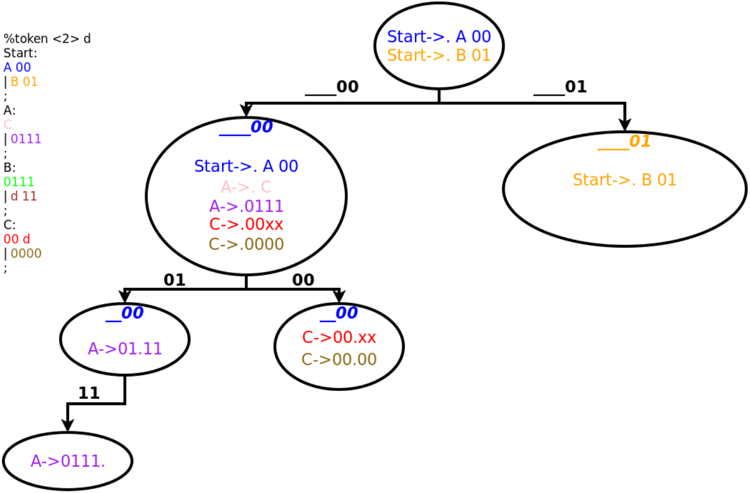
Example: Parser Generation from grammar

```
%token <2> d
Start:
A 00
| B 01
;
A:
C
| 0111
;
B:
0111
| d 11
;
C:
00 d
| 0000
;
```



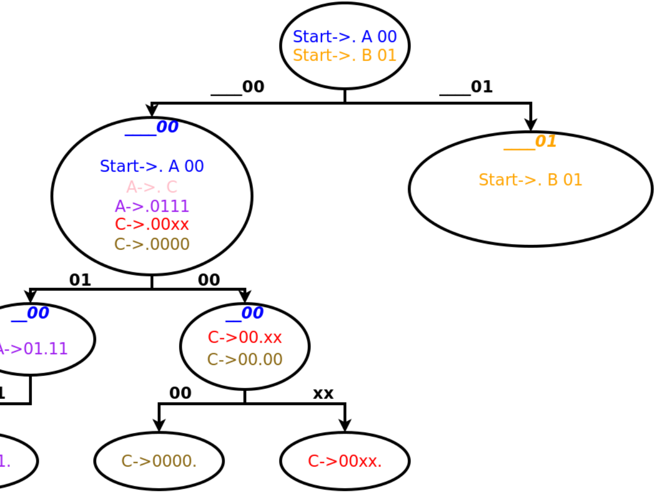
Example: Parser Generation from grammar

```
%token <2> d
Start:
A 00
| B 01
;
A:
C
| 0111
;
B:
0111
| d 11
;
C:
00 d
| 0000
;
```



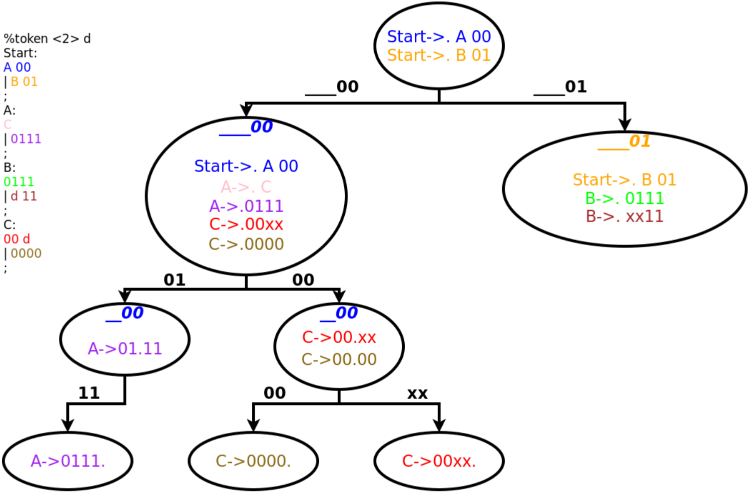
Example: Parser Generation from grammar

```
%token <2> d
Start:
A 00
| B 01
;
A:
C
| 0111
;
B:
0111
| d 11
;
C:
00 d
| 0000
;
```



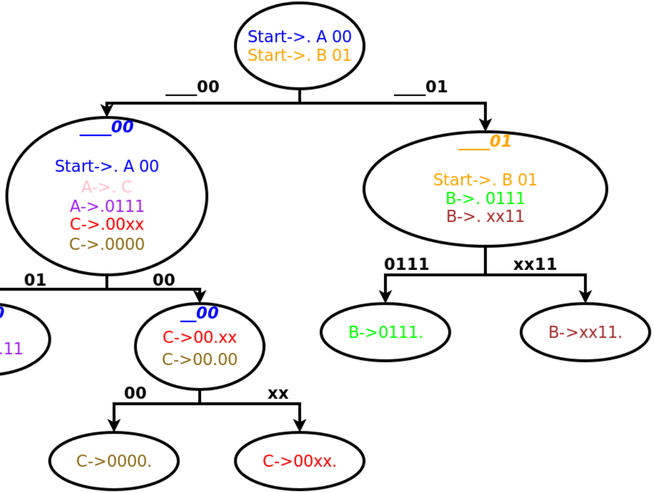
Example: Parser Generation from grammar

```
%token <2> d
Start:
A 00
| B 01
;
A:
C
| 0111
;
B:
0111
| d 11
;
C:
00 d
| 0000
;
```



Example: Parser Generation from grammar

%token <2> d
Start:
A 00
| B 01
;
A:
C
| 0111
;
B:
0111
| d 11
;
C:
00 d
| 0000
;



Encoder generation

Building an encoder from the same grammar file

- Semantic actions are redefined as matching functions
- Input tentatively matched over all productions of nonterminals
- Shortest productions are matched first
- Nonterminals in a matching production are recursively matched
- Resulting encoder algorithm corresponds to a top-down parser

Example: Encoder algorithm

```
%token <2> d
Start:
A 00   #[ S_ACT1($1) ]#
| B 01 #[ S_ACT2($1) ]# ;
A:
C       #[ A_ACT1($1) ]#
| 0111 #[ A_ACT2()    ]#;
B:
0111  #[ B_ACT1()     ]#
| d 11 #[ B_ACT2($1) ]# ;
C:
00 d    #[ C_ACT1($1) ]#
| 0000 #[ C_ACT2() ]#    ;
```

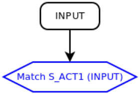
Example: Encoder algorithm

```
%token <2> d
Start:
A 00   #[ S_ACT1($1) ]#
| B 01 #[ S_ACT2($1) ]# ;
A:
C        #[ A_ACT1($1) ]#
| 0111 #[ A_ACT2()    ]#;
B:
0111  #[ B_ACT1()    ]#
| d 11 #[ B_ACT2($1) ]# ;
C:
00 d    #[ C_ACT1($1) ]#
| 0000 #[ C_ACT2() ]#    ;
```

INPUT

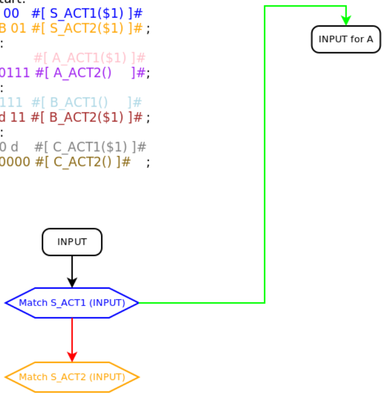
Example: Encoder algorithm

```
%token <2> d
Start:
A 00   #[ S_ACT1($1) ]#
| B 01 #[ S_ACT2($1) ]# ;
A:
C       #[ A_ACT1($1) ]#
| 0111 #[ A_ACT2()    ]#;
B:
0111  #[ B_ACT1()     ]#
| d 11 #[ B_ACT2($1) ]# ;
C:
00 d    #[ C_ACT1($1) ]#
| 0000 #[ C_ACT2() ]#    ;
```



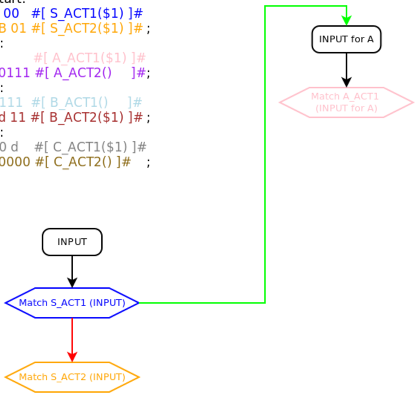
Example: Encoder algorithm

```
%token <2> d
Start:
A 00   #[ S_ACT1($1) ]#
| B 01 #[ S_ACT2($1) ]# ;
A:
C      #[ A_ACT1($1) ]#
| 0111 #[ A_ACT2()    ]#;
B:
0111  #[ B_ACT1()    ]#
| d 11 #[ B_ACT2($1) ]# ;
C:
00 d   #[ C_ACT1($1) ]#
| 0000 #[ C_ACT2() ]#   ;
```



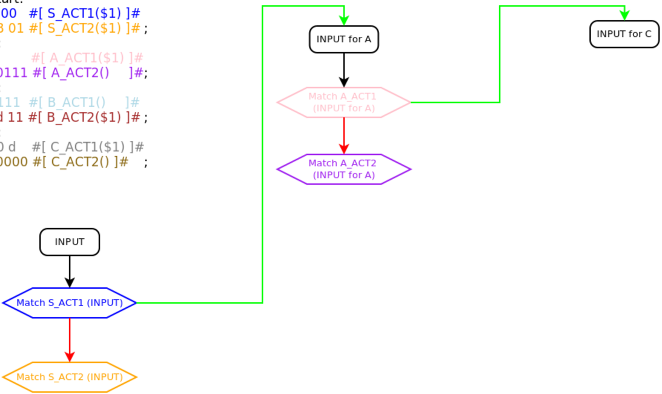
Example: Encoder algorithm

```
%token <2> d
Start:
A 00   #[ S_ACT1($1) ]#
| B 01 #[ S_ACT2($1) ]# ;
A:
C       #[ A_ACT1($1) ]#
| 0111 #[ A_ACT2()    ]#;
B:
0111  #[ B_ACT1()    ]#
| d 11 #[ B_ACT2($1) ]# ;
C:
00 d    #[ C_ACT1($1) ]#
| 0000 #[ C_ACT2() ]#   ;
```



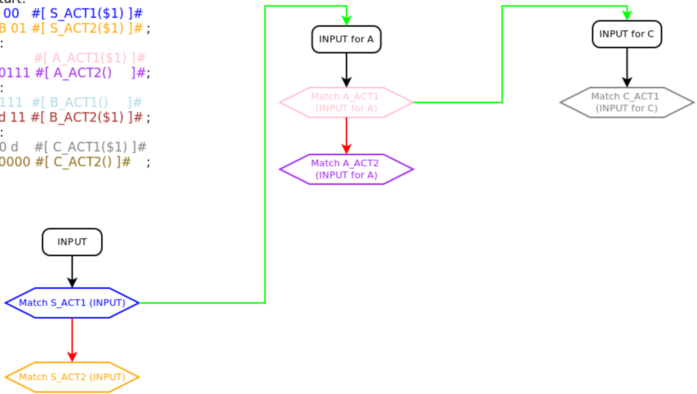
Example: Encoder algorithm

```
%token <2> d
Start:
A 00   #[ S_ACT1($1) ]#
| B 01 #[ S_ACT2($1) ]# ;
A:
C      #[ A_ACT1($1) ]#
| 0111 #[ A_ACT2()    ]#;
B:
0111  #[ B_ACT1()    ]#
| d 11 #[ B_ACT2($1) ]# ;
C:
00 d   #[ C_ACT1($1) ]#
| 0000 #[ C_ACT2() ]#    ;
```



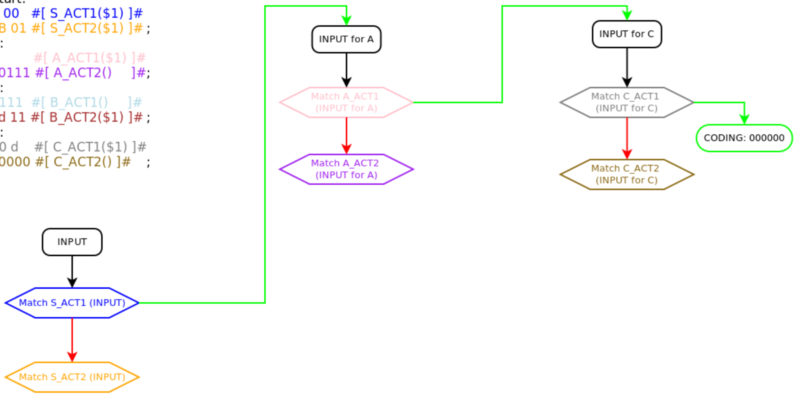
Example: Encoder algorithm

```
%token <2> d
Start:
A 00   #[ S_ACT1($1) ]#
| B 01 #[ S_ACT2($1) ]# ;
A:
C       #[ A_ACT1($1) ]#
| 0111 #[ A_ACT2()    ]#;
B:
0111  #[ B_ACT1()     ]#
| d 11 #[ B_ACT2($1) ]# ;
C:
00 d    #[ C_ACT1($1) ]#
| 0000 #[ C_ACT2() ]#    ;
```



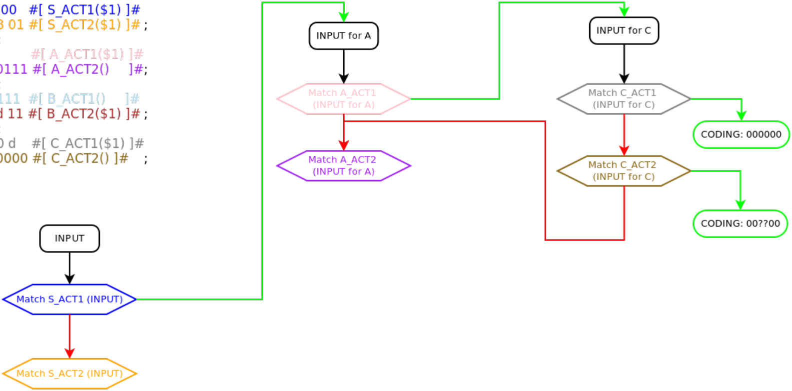
Example: Encoder algorithm

```
%token <2> d
Start:
A 00   #[ S_ACT1($1) ]#
| B 01 #[ S_ACT2($1) ]# ;
A:
C       #[ A_ACT1($1) ]#
| 0111 #[ A_ACT2()    ]#;
B:
0111  #[ B_ACT1()     ]#
| d 11 #[ B_ACT2($1) ]# ;
C:
00 d    #[ C_ACT1($1) ]#
| 0000 #[ C_ACT2() ]#    ;
```



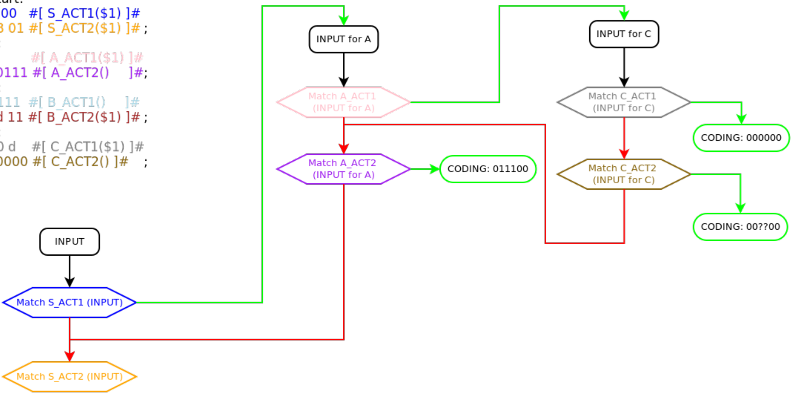
Example: Encoder algorithm

```
%token <2> d
Start:
A 00   #[ S_ACT1($1) ]#
| B 01 #[ S_ACT2($1) ]# ;
A:
C      #[ A_ACT1($1) ]#
| 0111 #[ A_ACT2()    ]#;
B:
0111  #[ B_ACT1()    ]#
| d 11 #[ B_ACT2($1) ]# ;
C:
00 d   #[ C_ACT1($1) ]#
| 0000 #[ C_ACT2() ]#   ;
```



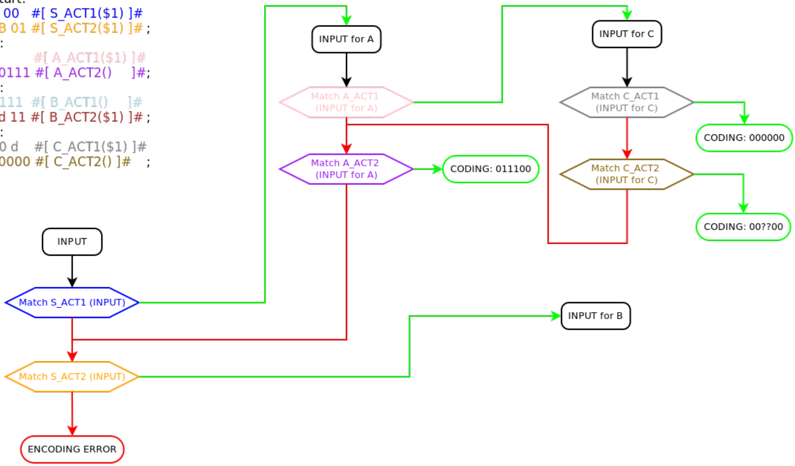
Example: Encoder algorithm

```
%token <2> d
Start:
A 00   #[ S_ACT1($1) ]#
| B 01 #[ S_ACT2($1) ]# ;
A:
C      #[ A_ACT1($1) ]#
| 0111 #[ A_ACT2()    ]#;
B:
0111  #[ B_ACT1()     ]#
| d 11 #[ B_ACT2($1) ]# ;
C:
00 d   #[ C_ACT1($1) ]#
| 0000 #[ C_ACT2() ]#    ;
```



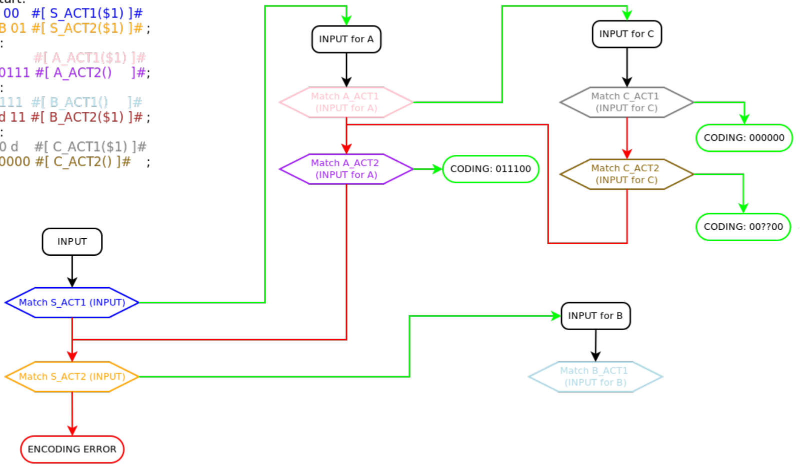
Example: Encoder algorithm

```
%token <2> d
Start:
A 00   #[ S_ACT1($1) ]#
| B 01 #[ S_ACT2($1) ]# ;
A:
C       #[ A_ACT1($1) ]#
| 0111 #[ A_ACT2()    ]#;
B:
0111  #[ B_ACT1()     ]#
| d 11 #[ B_ACT2($1) ]# ;
C:
00 d    #[ C_ACT1($1) ]#
| 0000 #[ C_ACT2() ]#    ;
```



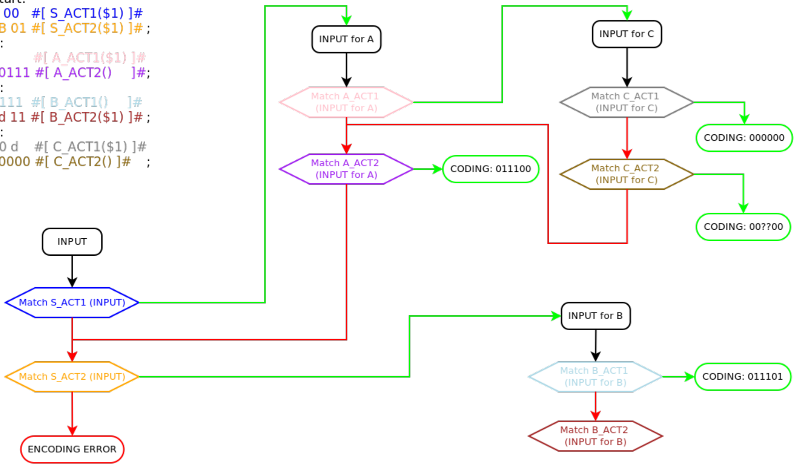
Example: Encoder algorithm

```
%token <2> d
Start:
A 00   #[ S_ACT1($1) ]#
| B 01 #[ S_ACT2($1) ]# ;
A:
C       #[ A_ACT1($1) ]#
| 0111 #[ A_ACT2()    ]#;
B:
0111  #[ B_ACT1()     ]#
| d 11 #[ B_ACT2($1) ]# ;
C:
00 d    #[ C_ACT1($1) ]#
| 0000 #[ C_ACT2() ]#    ;
```



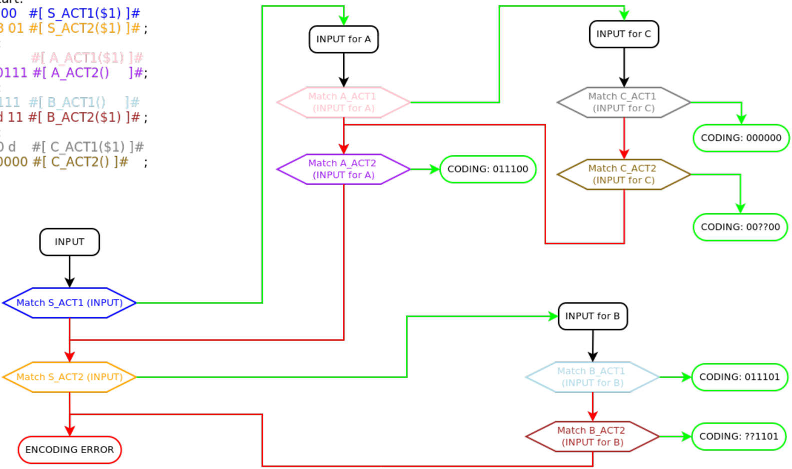
Example: Encoder algorithm

```
%token <2> d
Start:
A 00   #[ S_ACT1($1) ]#
| B 01 #[ S_ACT2($1) ]# ;
A:
C      #[ A_ACT1($1) ]#
| 0111 #[ A_ACT2()    ]#;
B:
0111  #[ B_ACT1()     ]#
| d 11 #[ B_ACT2($1) ]# ;
C:
00 d   #[ C_ACT1($1) ]#
| 0000 #[ C_ACT2() ]#    ;
```



Example: Encoder algorithm

```
%token <2> d
Start:
A 00   #[ S_ACT1($1) ]#
| B 01 #[ S_ACT2($1) ]# ;
A:
C       #[ A_ACT1($1) ]#
| 0111 #[ A_ACT2()     ]#;
B:
0111  #[ B_ACT1()      ]#
| d 11 #[ B_ACT2($1) ]# ;
C:
00 d    #[ C_ACT1($1) ]#
| 0000 #[ C_ACT2() ]#    ;
```



Validation

MINJAG

- Uses a context-free grammar describing the architecture
- Grammar generated from architecture documentation through simple transformations
- Generates the code for decoder and encoder from the same grammar
- Functional tool used in a production context
- Tested over Intel 64, Intel Xeon Phi coprocessor and ARM

Characteristics of implemented architectures

Architecture	Intel 64	Intel Xeon Phi	ARM
Lines in instruction list	2,398	1,194	1,512
Lines in grammar	6,082	3,082	1,491
Reduction states	5,950	2,406	1,625
Shift states	4,019	1,468	2,916
Shift/reduce states	2	2	6
Total states	9,971	3,876	4,547

Challenges of disassembly

Binary code is not intended to be read

- No constraints on the code as long as the program can be executed
- No separation between instructions
- Instructions may be of varying sizes

Specific examples

- Interleaved foreign bytes
- Overlapping instructions
- Obfuscated code or binary format
- Self rewriting code

Example: Interleaved foreign bytes

Correct disassembly up to that point

Binary code	Corresponding assembly instructions
67 4C 89 4C 90 20	mov %r9,0x20(%eax,%edx,4)
EB 04	jmp <+4 bytes>
80	*Alignment byte, never executed*
4C 89 C8	mov %r9,%rax
F2 0F 10 EE	movsd %xmm6, %xmm5
F2 0F 10 F4	movsd %xmm4, %xmm6

Mistaking the alignment byte for the beginning of the next instruction

Realignement of the parser on a valid boundary

Disassembly

67 4C 89 4C 90 20	mov %r9,0x20(%eax,%edx,4)	Instructions correctly disassembled
EB 04	jmp <+4 bytes>	
80 4C 89 C8 F2	or $0xf2,-0x38(%rcx,%rcx,4)	Erroneous instructions
0F 10 EE	movups %xmm6, %xmm5	
F2 0F 10 F4	movsd %xmm4, %xmm6	Instructions correctly disassembled

Disassembly algorithms

Linear sweep

- Decoding one instruction after another
- Errors when encountering interleaved foreign bytes
- Vulnerability to obfuscation methods
- Faster disassembly

Recursive traversal

- Decoding following the actual execution of the program
- Resists to some obfuscation techniques
- Finding the destination of a branch can be difficult
- Slower disassembly

Linear Sweep vs Recursive Traversal

Linear sweep

01100111010011001000100101001100
1001000000100000 1110101100000011
010010011000100111001110 10010000
11110010000011110001000011101110

Recursive traversal

01100111010011001000100101001100
1001000000100000 1110101100000011
010010011000100111001110 10010000
11110010000011110001000011101110

Linear Sweep vs Recursive Traversal

Linear sweep

01100111010011001000100101001100
1001000000100000 1110101100000011
010010011000100111001110 10010000
11110010000011110001000011101110

00: mov %r9,0x20(%eax,%edx,4)

Recursive traversal

01100111010011001000100101001100
1001000000100000 1110101100000011
010010011000100111001110 10010000
11110010000011110001000011101110

00: mov %r9,0x20(%eax,%edx,4)

Linear Sweep vs Recursive Traversal

Linear sweep	Recursive traversal
01100111010011001000100101001100 1001000000100000 1110101100000011 010010011000100111001110 10010000 11110010000011110001000011101110	01100111010011001000100101001100 1001000000100000 1110101100000011 010010011000100111001110 10010000 11110010000011110001000011101110
00: mov %r9,0x20(%eax,%edx,4) 06: jmp <0C> #+4 bytes	00: mov %r9,0x20(%eax,%edx,4) 06: jmp <0C> #+4 bytes

Linear Sweep vs Recursive Traversal

Linear sweep

```
01100111010011001000100101001100
1001000000100000 1110101100000011
01001001100010011100111 0 10010000
11110010000011110001000011101110
```

00: mov %r9,0x20(%eax,%edx,4)
06: jmp <0C> #+4 bytes
08: mov %rcx,%r14

Recursive traversal

```
01100111010011001000100101001100
1001000000100000 1110101100000011
010010011000100111001110 10010000
11110010000011110001000011101110
```

00: mov %r9,0x20(%eax,%edx,4)
06: jmp <0C> #+4 bytes

0C: movsd %xmm6,%xmm5

Our constraints

Disassembler intended to be used by analysis tools

- Retrieve all possible available information from the file
- Architecture independent output format
- Possibility to add customisable additional information
- Acceptable performance in terms of speed and accuracy

Our disassembling algorithm

General execution

- Linear sweep parsing
- Extraction of executable code from binary format
- Retrieval of labels and debug information if present

Additional processing

- Resolving destination of direct branches
- Associating labels and debug information to instructions
- Post parsing actions to fill additional information
- Detection of unreachable instructions
- Identification of dubious disassembled data

Implementation: the MADRAS disassembler

Multi Architecture Disassembler, Rewriter and ASsembler

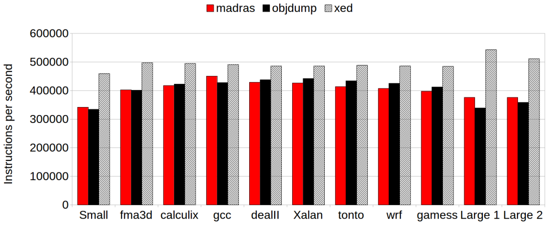
- Relies on MINJAG for source code of decoder
- Processes binaries using the ELF format used by Unix and Linux
- Disassembler available for Intel 64, Xeon Phi coprocessor and ARM
- Base component of the MAQAO framework

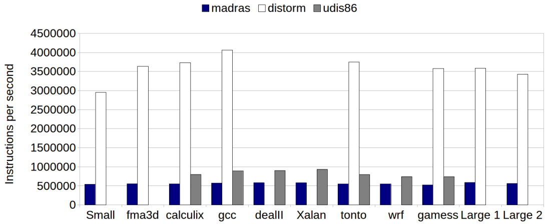
Performance tests

Protocol

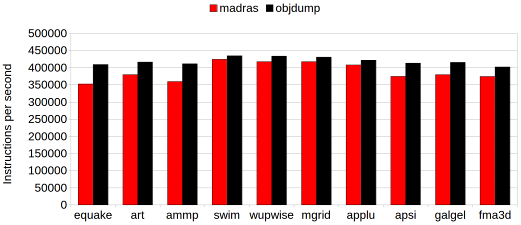
- Comparison between MADRAS and hard coded disassemblers
- Disassembling SPEC benchmarks and test files
  - Size of executable code varying between 1 and 23 MBytes
  - Executables compiled for Intel 64 and Xeon Phi coprocessor
- Speed measured as disassembled instructions per second

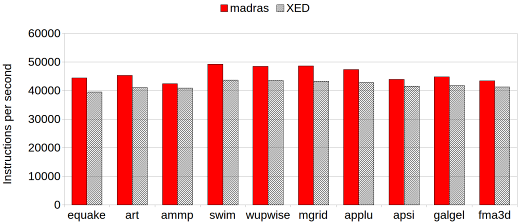
Disassembler performance on Intel 64 files



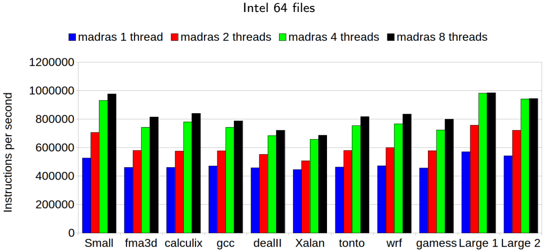


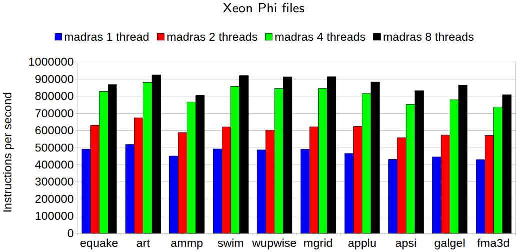
Disassembler performance on Xeon Phi files



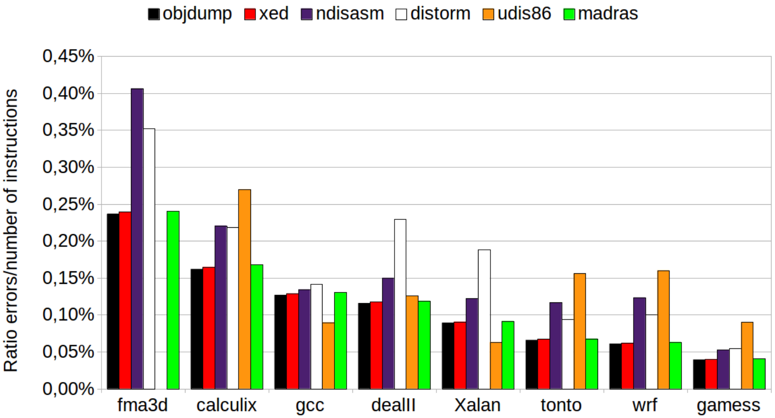


Parallel disassembler performance





Disassembler accuracy



Instrumentation

Retrieving information during execution

- Monitoring memory usage
- Value profiling

Dynamic: Performed during execution

- Monitoring code execution using a supervising thread
- Invoking functions under specified conditions
- Modifying the image loaded in memory

Static: Modifying the executable file

- Probe insertion
- Instructions modification

Binary rewriting

Static instrumentation

- No recompilation needed
- No overhead from instrumentation process
- No additional requirements for execution

Binary rewriting allows other modifications to the program

- Deleting or adding instructions to test their overall impact
- Modifying variables defined in the file

Challenges of binary rewriting

Patched file must remain valid

- Preservation of the structure of the binary file
- Preservation of the control flow
- Preservation of data environment

Executables are not intended to be modified

- All references are fixed
- No relocation tables
- Addresses can appear as immediate operands

Example of patching pitfalls

00:	48 8B 04 25 0E 00 00 00	mov $0x14, %rax
08:	FF E0	jmp *%rax
0A:	83 45 F8 01	add $1, -8(%rbp)
0E:	83 7D F8 01	cmp $1, -8(%rbp)
12:	7E F6	jle 0A
14:	B8 00 00 00 00	mov $0, %eax

Example of patching pitfalls

```
E8 xx xx xx xx    callq  <myfunc>

00:  48 8B 04 25 0E 00 00 00    mov  $0x14, %rax
08:  FF E0                      jmp  *%rax
0A:  83 45 F8 01                add  $1, -8(%rbp)
0E:  83 7D F8 01                cmp $1, -8(%rbp)
12:  7E F6                      jle  0A
14:  B8 00 00 00 00             mov  $0, %eax
```

Example of patching pitfalls

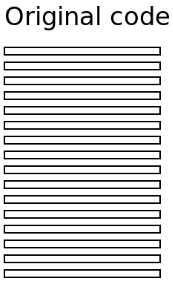
```
00:  48 8B 04 25 0E 00 00 00   mov  $0x14, %rax
08:  FF E0                     jmp  *%rax
0A:  83 45 F8 01               add  $1, -8(%rbp)
0E:  E8 xx xx xx xx            callq  <myfunc>
13:  83 7D F8 01               cmpl $1, -8(%rbp)
17:  7E F6                     jle  0A
19:  B8 00 00 00 00            mov  $0, %eax
```

Binary rewriting algorithm

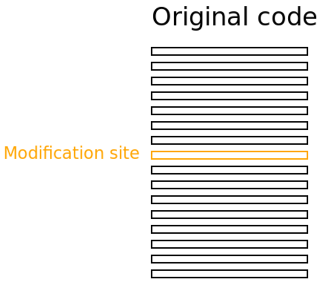
Block relocation

- The code to be modified is moved in a new section in the executable
- Code moved at the basic block level
- Use of trampolines if the patching site is too small

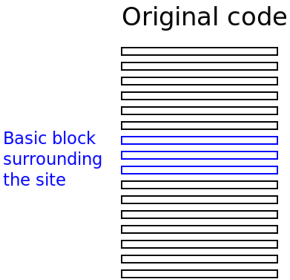
Code relocation



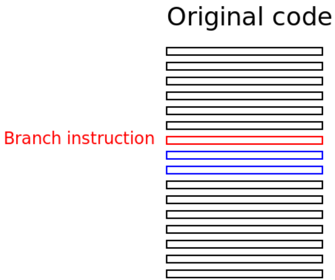
Code relocation



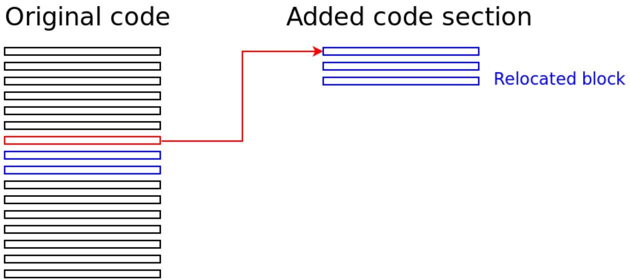
Code relocation



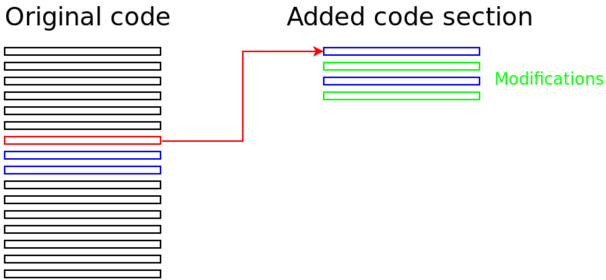
Code relocation



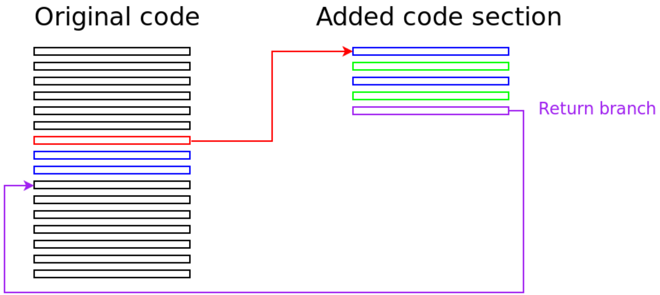
Code relocation



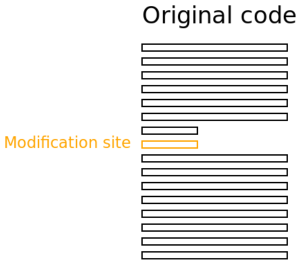
Code relocation



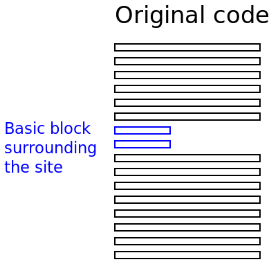
Code relocation



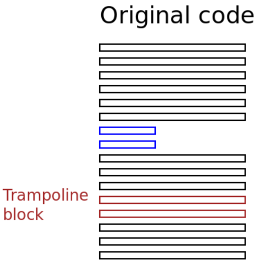
Trampolines



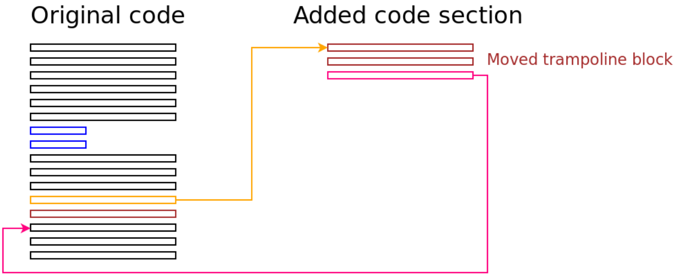
Trampolines



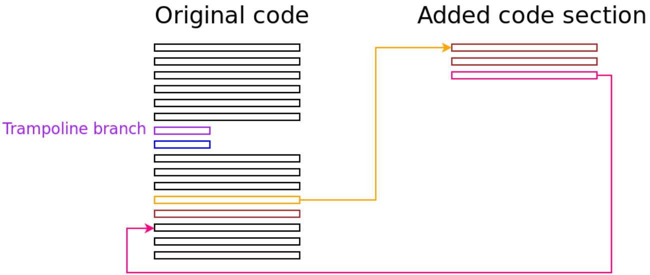
Trampolines



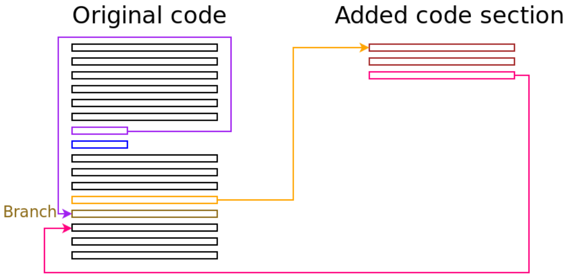
Trampolines



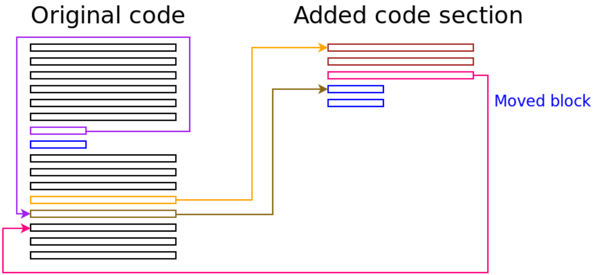
Trampolines



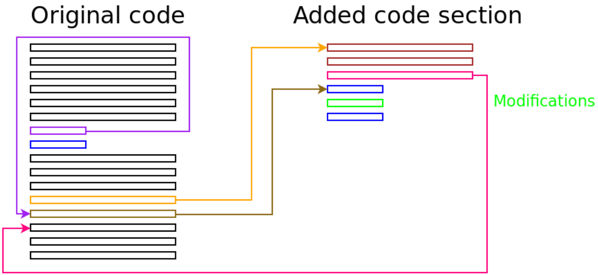
Trampolines



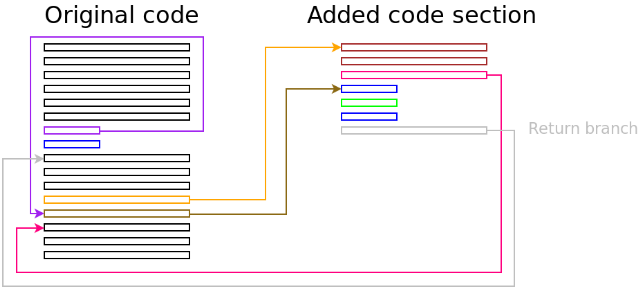
Trampolines



Trampolines



Trampolines



Implementation: the MADRAS patcher

Multi Architecture Disassembler, **Rewriter and ASsembler**

- Relies on MINJAG for source code of assembler
- Processes binaries under the ELF format used by Unix and Linux
- Available for Intel 64 and Xeon Phi coprocessor

A production tool

- C API
- Back end of the MAQAO Instrumentation Language (MIL)
- Used by the DECAN module

Patcher features

Code insertion

- Insertion of calls to functions from external or static libraries
- Insertion of lists of assembly instructions

Conditions

- Possibility to set conditions on the execution of an inserted code
- Possibility to specify code to execute if such a condition is not met

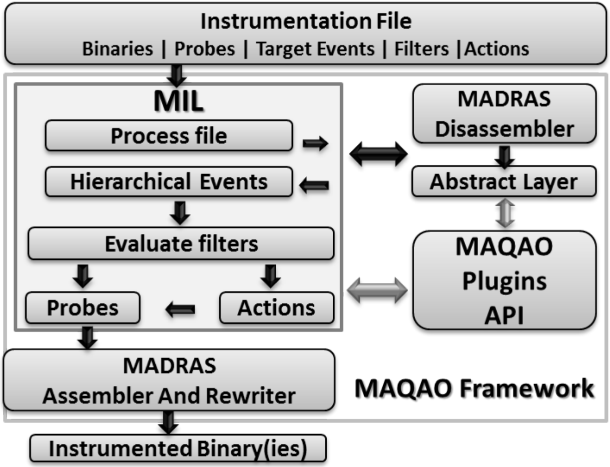
Other features

- Modification or deletion of instructions
- Insertion of global variables usable by inserted code

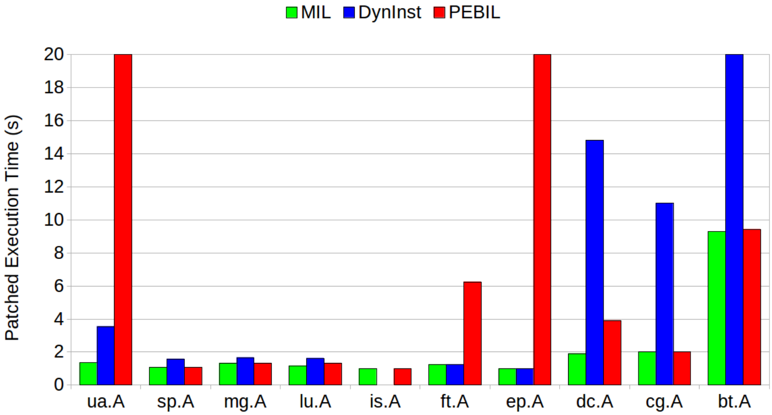
Example: Using MADRAS API to insert a function call

```
void insert(char* in,char* lib,char* fct,uint addr,char* out) {
  //Disassemble the file and inits the modifications
  elfdis_t* madras = madras_disass_file(in);
  madras_modifs_init(madras, STACK_SHIFT, 512);
  //Adds a function call at the given address
  insert_t* ifct = madras_fctcall_new(madras, fct, lib, addr, 0);
  //Adds the given address as an immediate parameter
  madras_fctcall_addparam_imm(madras, ifct, addr, 0);
  //Commit changes
  madras_modifs_commit(madras,out);
  //Terminates the madras structure
  madras_terminate(madras);
}
```

Interface with the MAQAO Instrumentation Language



Performance of code patched by MIL



Contributions

Generic representation of binary encoding rules

- Unified format
- Use of the same grammar for encoder and decoder generation
- Validated for the Intel and ARM architectures
- Implemented as the functional tool MINJAG

Disassembly

- Easier updates of architecture specific code
- Performance comparable to existing hard coded tools
- Customisable output

Contributions

Patching

- Fine granularity offering wide range of options
- Patched code has similar or better performance than existing tools

MADRAS

- Functional tool
- Standalone implementation of the whole disassembly and instrumentation chain
- Handling of multiple architectures from a single executable
- Integral component of the MAQAO framework
- Used by the DECAN module

Future work

General

- Implement additional architectures
- Support additional binary file formats

Generic encoder and decoder

- Generic meta language for representing instruction lists
- Extensions allowing to specialise generated parser

Future work

Disassembler

- Improve accuracy through use of recursive traversal
- Detection of switch tables
- Improve speed
- Parallel disassembly
- Application to domains outside performance analysis

Patcher

- Improve safety of patching
- Update of indirect branch destinations

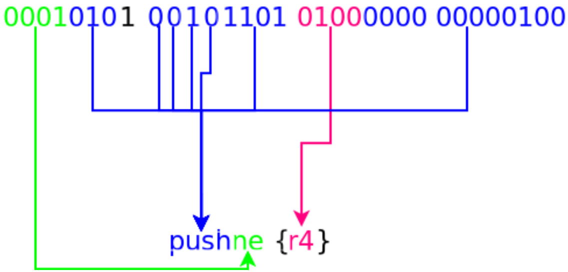
Q/A

Thank you for your attention!

Additional slides

Example: Encoding of an ARM instruction

0x15 2D 40 04 <=> pushne {r4}



Example of grammar for binary definition

```
%token <3,b> reg
%%
Start: template ;
template: Legacy3 Insn #[ FULLINSN_L3PREFIX($1,$2) ]#
        | Insn #[ FULLINSN($1) ]# ;
MemModRM: 00 reg RMSIB_00 #[ OPRS_REG_MEM($1,$2) ]#
        | 01 reg RMSIB_01 #[ OPRS_REG_MEM($1,$2) ]#
        | 10 reg RMSIB_10 #[ OPRS_REG_MEM($1,$2) ]# ;
RegModRM: 11 reg RMSIB_11 #[ OPRS_REG_REG($1,$2) ]# ;
Insn: 00010000 RegModRM #[ INSN(ADC,
        REG(GEN8b,R,$1),REG(GEN8b,RW,$1)) ]#
    | REX 00010000 MemModRM #[ INSN(ADC,
        REG(GEN8b,R,$1,$2),MEM(MEM8b,RW,$1,$2)) ]# ;
```

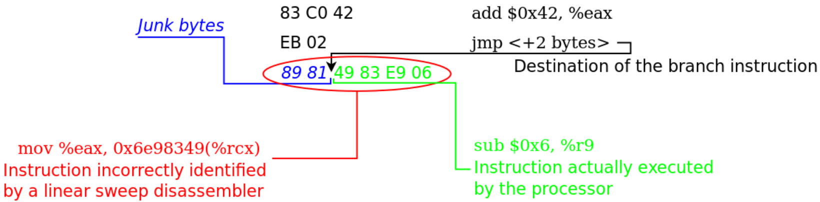
Overlapping instructions

Destination of the branch instruction

```
F3 AB          rep stos
48 FF C1       inc %rcx
48 83 F9 7F    cmp $127,%rcx
75 F6          jne <-10 bytes>
```

The first iteration of the loop will execute instruction rep stos
Later iterations will skip the F3 (rep) prefix and execute only the stos instruction

Obfuscated code



Performance tests

Disassemblers

- `objdump`
- `XED`
- `udis86`
- `distorm`
- `ndisasm`

Disassembly modes

- Print only mode for comparison against `objdump` and `XED`
- Without parsing of the binary file against `udis86` and `distorm`

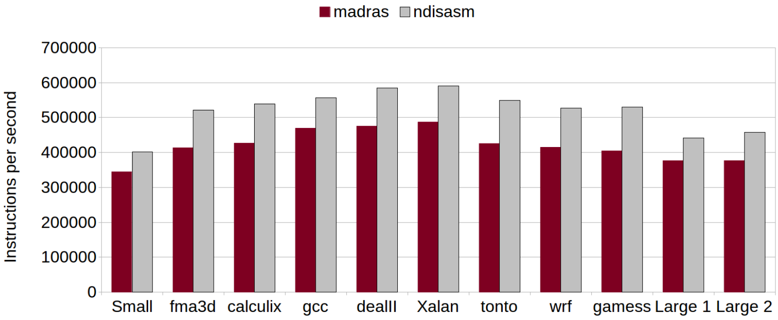
Intel 64 files used for the disassembler performance tests

File	File size (MByte)	Code size (MByte)	Description
Small	0,96	0,96	Test file
fma3d	3,78	1,75	SPEC2001
calculix	5	2,31	SPEC2006
gcc	9,02	2,56	SPEC2006
dealII	60,94	2,83	SPEC2006
Xalan	130,64	3,46	SPEC2006
tonto	33,27	5,81	SPEC2006
wrf	19,52	6,83	SPEC2006
gamess	18,2	10,55	SPEC2006
Large 1	11,95	11,94	Test file
Large 2	23,22	23,22	Test file

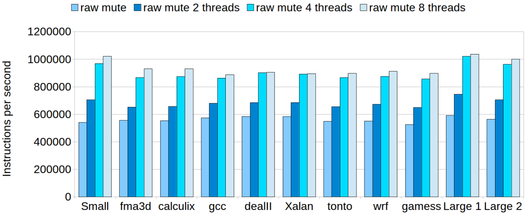
Xeon Phi files used for the disassembler performance tests

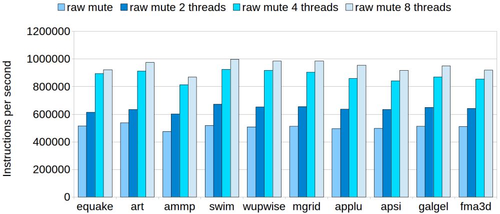
File	File size (Mb)	Code size (Mb)	Description
equake	0,12	0,05	SPEC2001
art	0,21	0,12	SPEC2001
ammp	0,84	0,44	SPEC2001
swim	0,96	0,57	SPEC2001
wupwise	0,96	0,66	SPEC2001
mgrid	0,95	0,68	SPEC2001
applu	1,03	0,71	SPEC2001
apsi	2,61	1,72	SPEC2001
galgel	2,84	2,08	SPEC2001
fma3d	4,62	2,35	SPEC2001

Disassembler performance on Intel 64 files

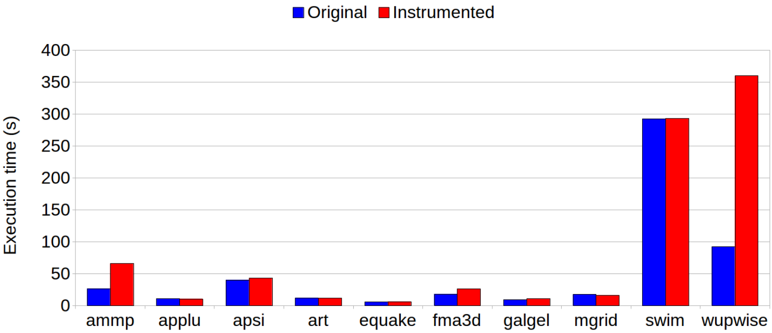


Parallel disassembler performance

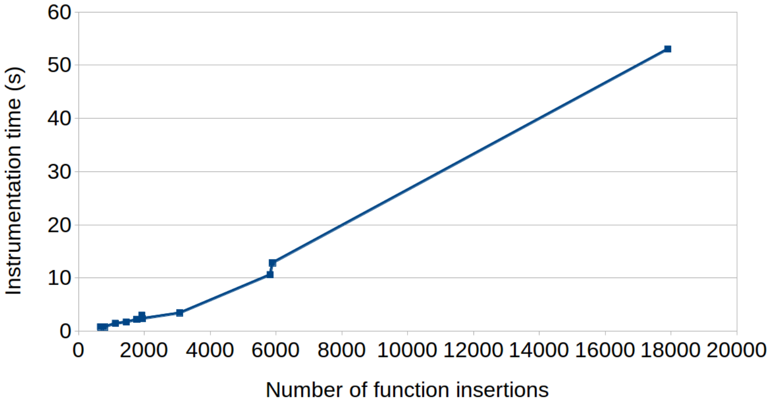




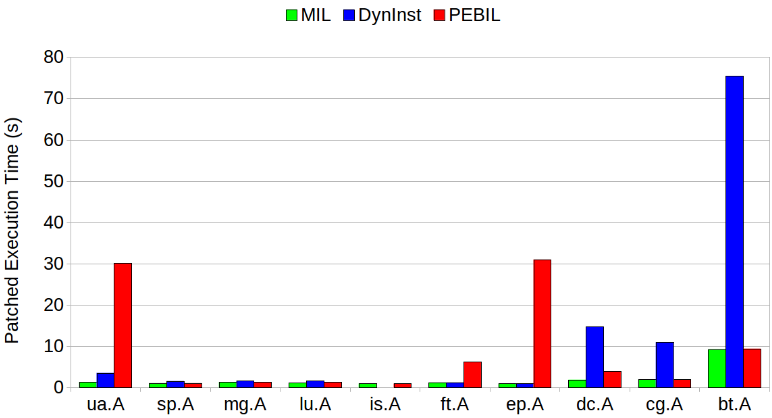
Performance of patched code



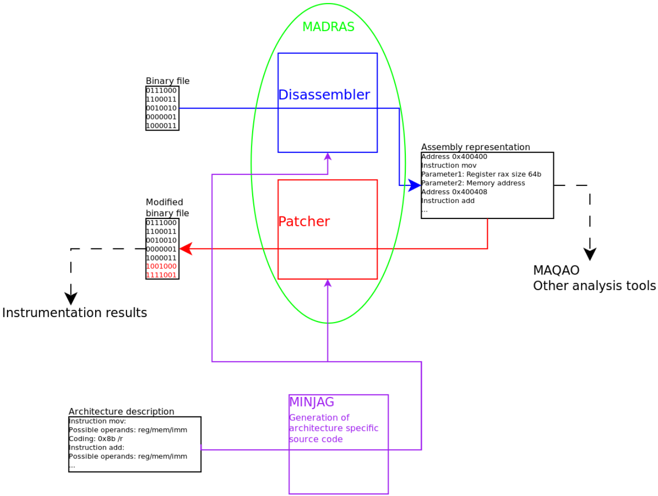
Performance of instrumentation

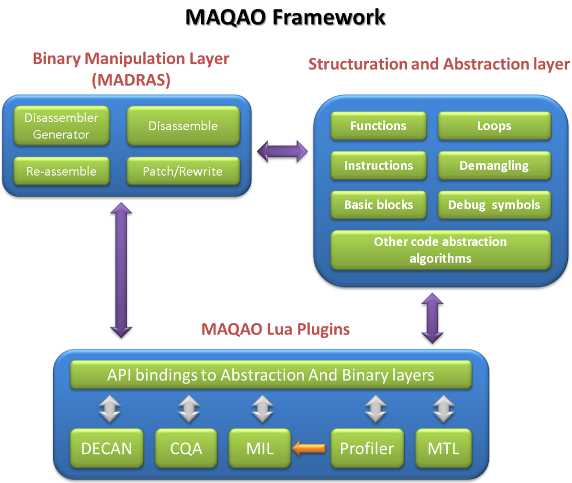


Performance of patched code



MADRAS overall architecture




MAQAO Framework
Binary Manipulation Layer (MADRAS)
Disassembler Generator
Disassemble
Re-assemble
Patch/Rewrite
Structuration and Abstraction layer
Functions
Loops
Instructions
Demangling
Basic blocks
Debug symbols
Other code abstraction algorithms
MAQAO Lua Plugins
API bindings to Abstraction And Binary layers
DECAN
CQA
MIL
Profiler
MTL