T-CREST

TACLe
Timing Analysis on Code-Level

`platin` – A Toolkit for Compiler and WCET-Analysis Integration

Stefan Hepp, Benedikt Huber, Daniel Prokesch

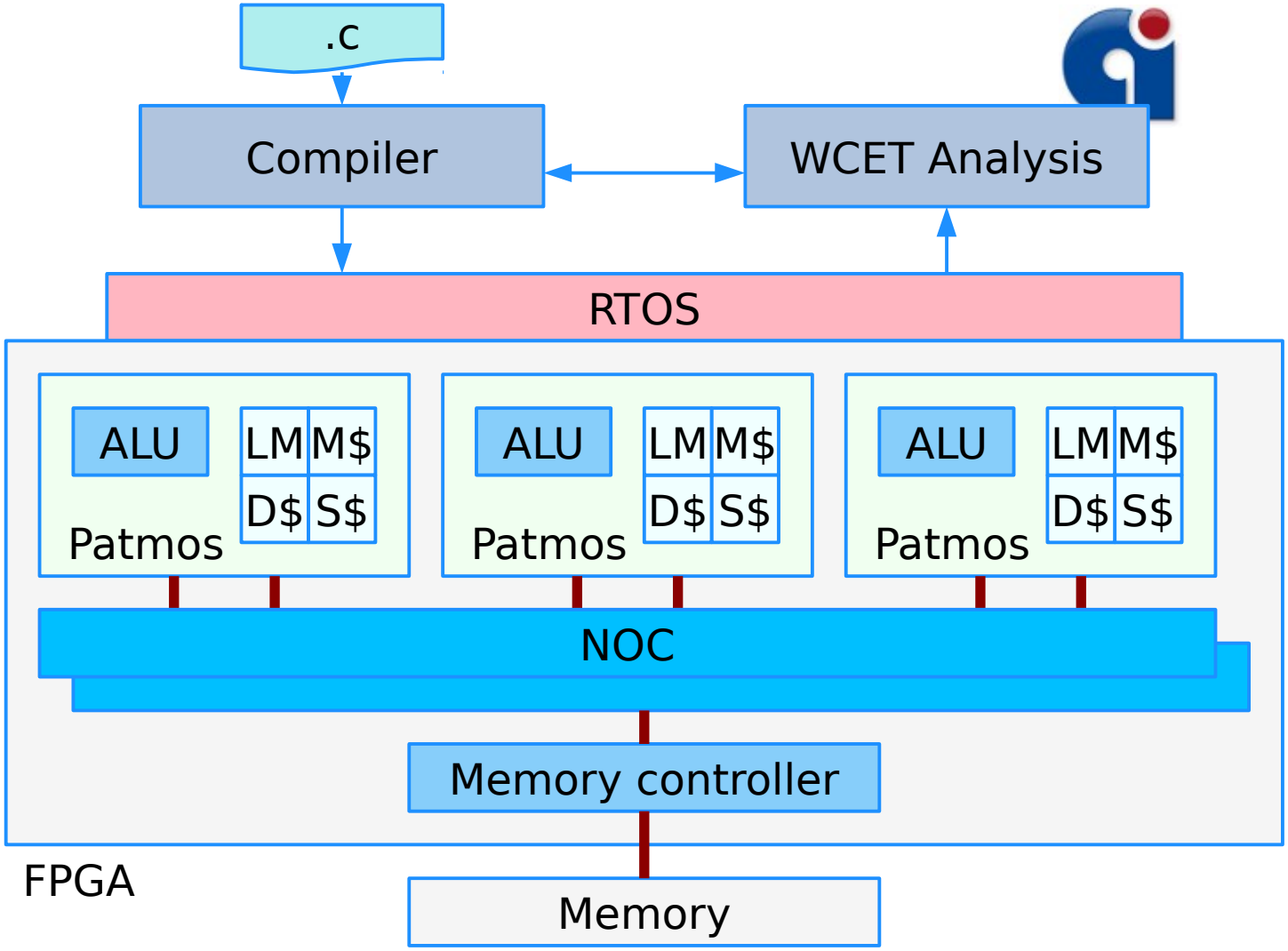
Vienna University of Technology

TU WIEN
TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

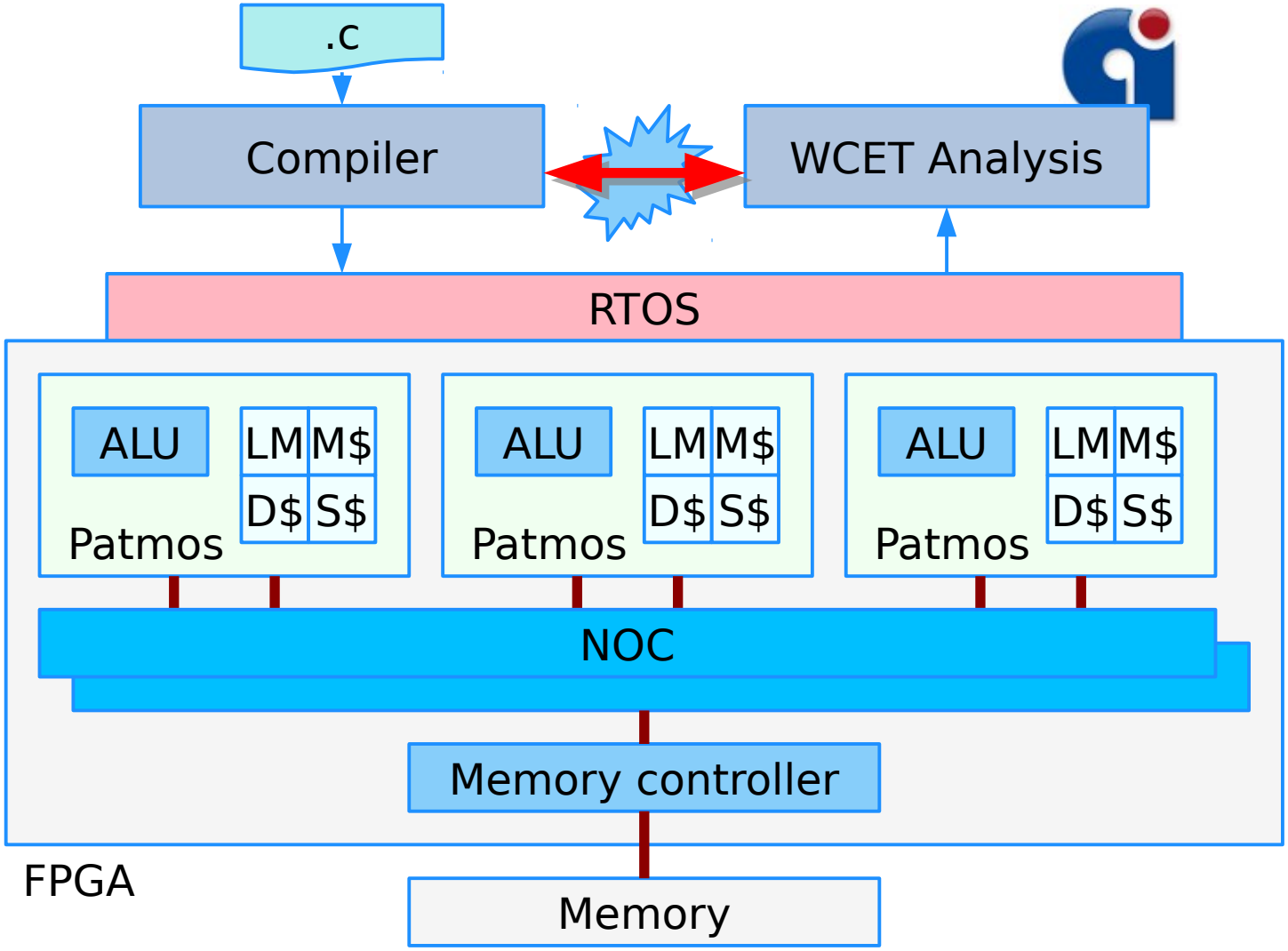
Outline

- The T-CREST Project
- The `platin` Toolkit
- The PML File Format
- The Future

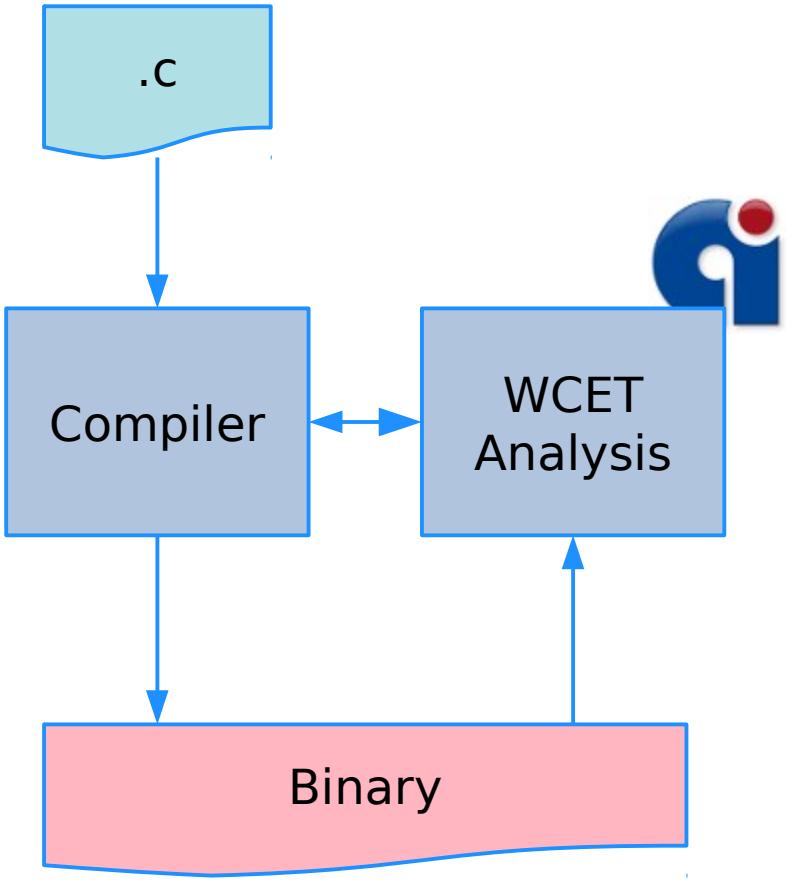
T-CREST: The Quest for a Time-Predictable Platform



T-CREST: The Quest for a Time-Predictable Platform

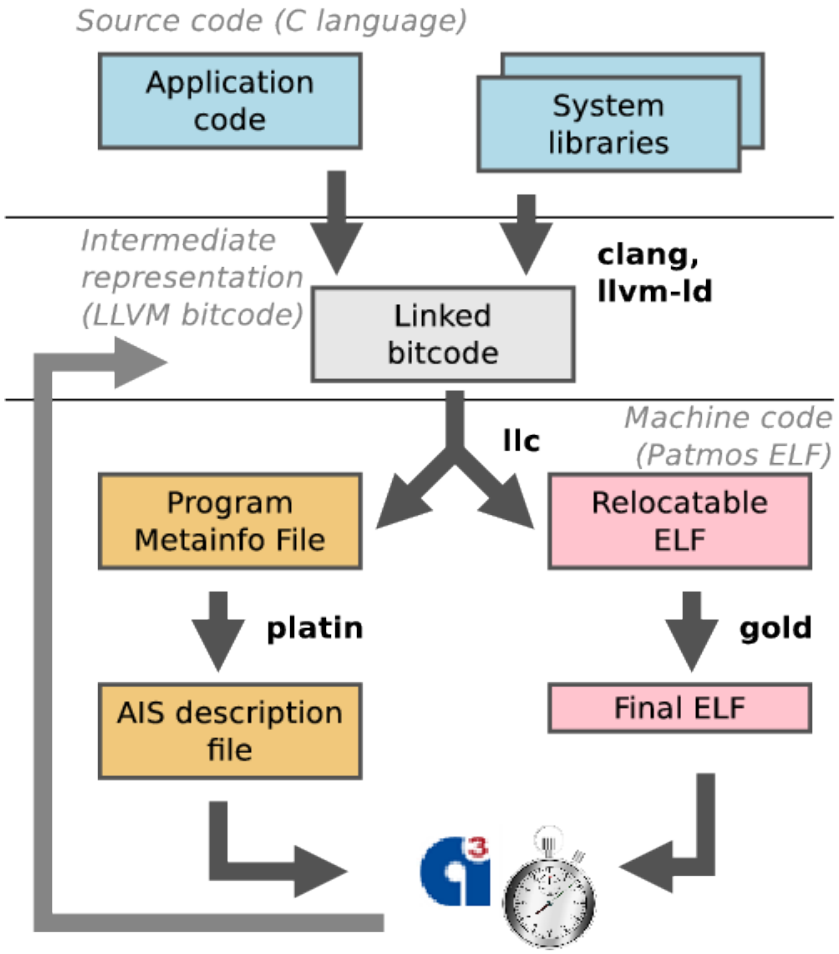


The Patmos Toolchain



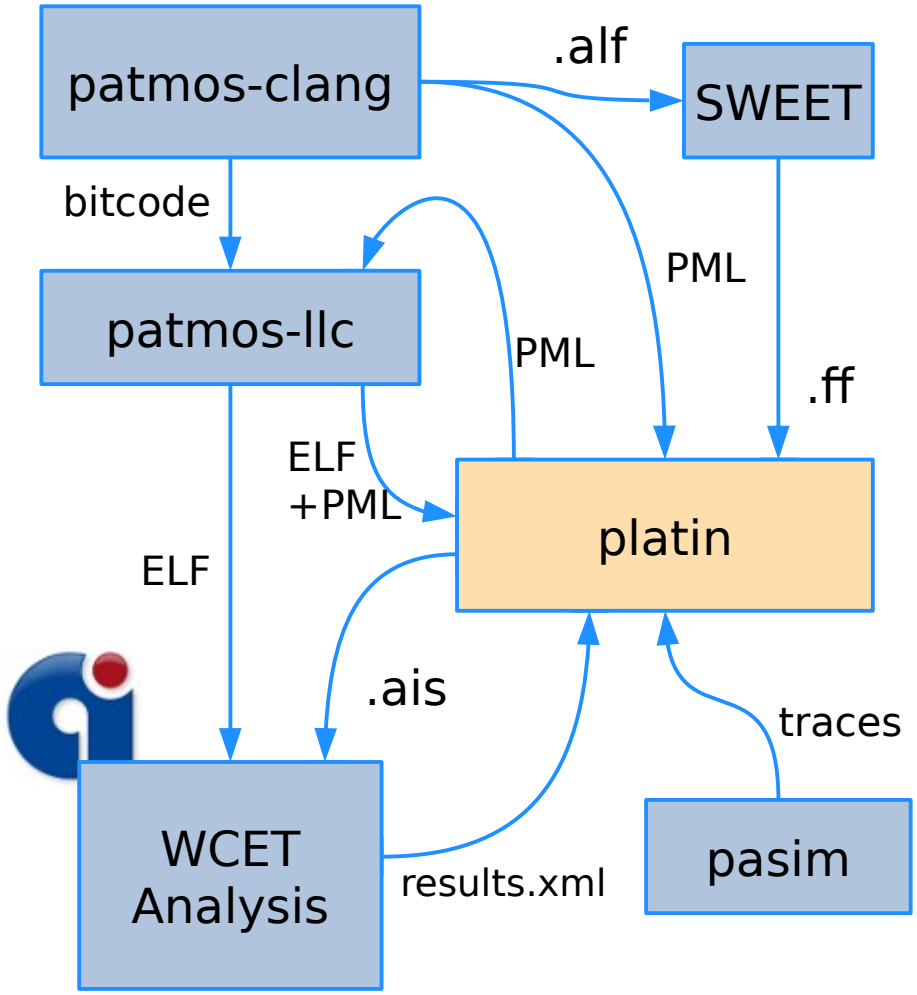
- Compile C code
- Optimize code (for Patmos)
- Support the WCET analysis
- Use WCET-analysis feedback

Compiler Overview



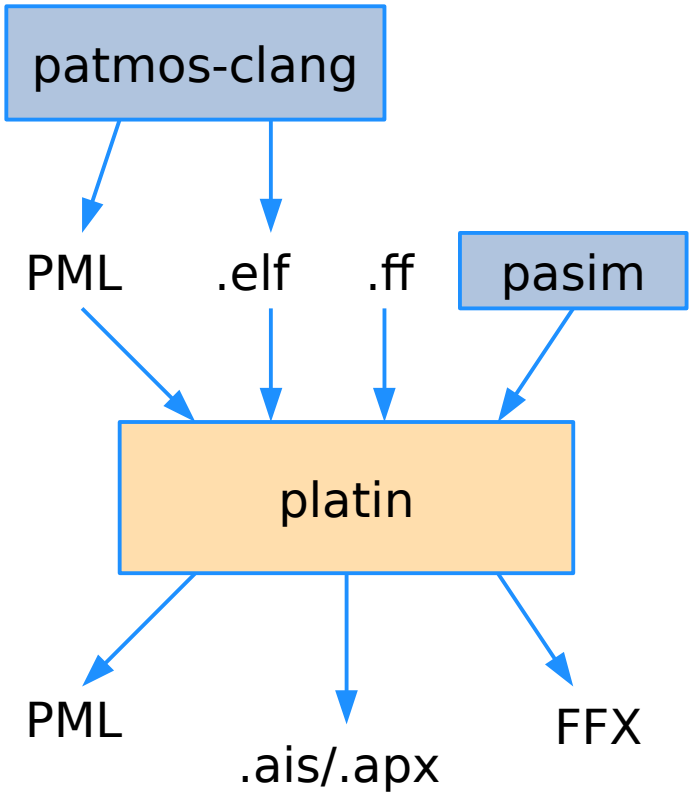
- Based on LLVM Compiler Framework
- C source files translated into LLVM bitcode (clang)
- Bitcode files (user, libraries) linked together to a single bitcode file
- Translated to machine code (relocatable ELF)
- Object code linker for final relocations
- Code generator exports information for WCET analysis

`platin` – The Portable LLVM Annotation and Timing Toolkit



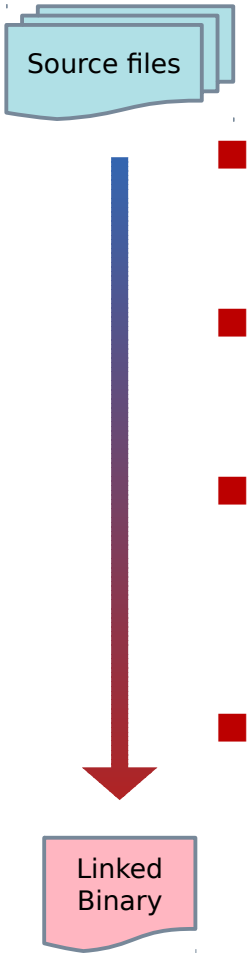
- Glue tool between compiler and analysis tools
- Import/export various file formats
- Transform information between various code representations
- Analyze programs, simulation traces, ..
- Central file format for meta-infos: PML

platin Tools



- `pml`: Merge, check, .. PML files
- `pml2ais`: Export to aiT's AIS format, create aiT project file
- `tool-config`: Get options for compiler, simulator, ...
- `extract-symbols`: Extract symbol addresses from ELF
- `sweet`: Run sweet analyzer
- `analyze-trace`: Generate flow facts from simulation trace
- `wca`: IPET-based WCET analysis
- `wcet`: Driver for WCET analysis

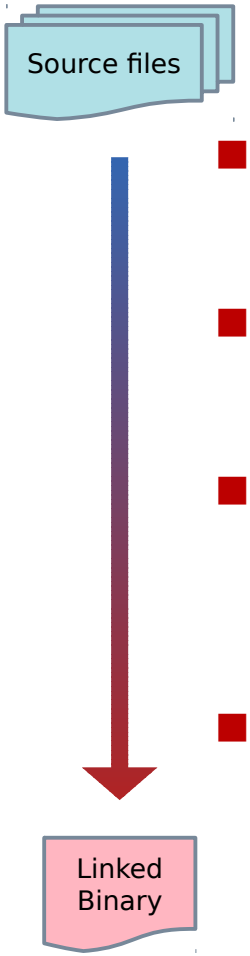
Preservation of Meta-information



- Source code → Bitcode
- High-level optimisations
- Bitcode → Machine code
  - Relations between basic blocks[1]
- Machine code → Linked binary
  - Symbols are preserved

[1] Benedikt Huber, Daniel Prokesch, and Peter Puschner. *Combined WCET analysis of bitcode and machine code using control-flow relation graphs*. In Proceedings of the 14th ACM SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems (LCTES '13). ACM, New York, NY, USA, 163-172.

Preservation of Meta-Information



- Source code → Bitcode
- High-level optimisations
- Bitcode → Machine code
  - Relations between basic blocks[1]
- Machine code → Linked binary
  - Symbols are preserved

PML file

- Configuration
- Structural Information
- Relation graphs[1]
- Flow facts
- Value facts
- Analysis results

[1] Benedikt Huber, Daniel Prokesch, and Peter Puschner. *Combined WCET analysis of bitcode and machine code using control-flow relation graphs*. In Proceedings of the 14th ACM SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems (LCTES '13). ACM, New York, NY, USA, 163-172.

Example: Jump-tables

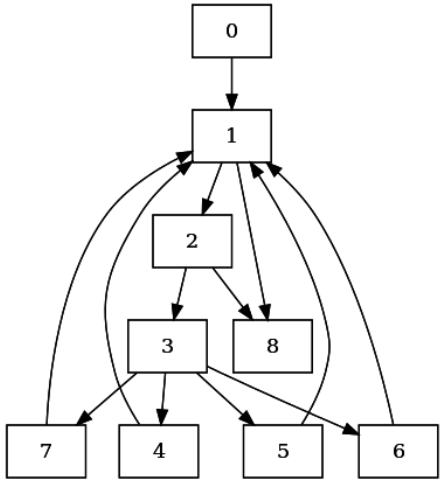
- Base64 encoding function

```
const char Base64[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
        "abcdefghijklmnopqrstuvwxyz0123456789+/";
const char Pad64 = '=';

int b64_pton(char const *src, char *target, size_t targsize)
{
  int tarindex=0, state=0;
  char *pos, ch;

  while ((ch = *src++) != '\0') {
    if (ch == Pad64) break;

    pos = strchr(Base64, ch);
    switch (state) {
      case 0:
        target[tarindex] = (pos - Base64) << 2;
        state = 1;
        break;
      case 1:
        target[tarindex]   |=  (pos - Base64) >> 4;
        target[tarindex+1]  = ((pos - Base64) & 0x0f) << 4 ;
        tarindex++;
        state = 2;
        break;
      case 2:
        target[tarindex]   |=  (pos - Base64) >> 2;
        target[tarindex+1]  = ((pos - Base64) & 0x03) << 6;
        tarindex++;
        state = 3;
        break;
      case 3:
        target[tarindex] |= (pos - Base64);
        tarindex++;
        state = 0;
        break;
      default:
        __builtin_unreachable();
    }
  }
  return (tarindex);
}
```



CFG for 2/b64_pton

```
.LBB2_3:
  16b0: 87 c2 10 0d 00 01 92 28   shadd2  $r1 = $r1, 102952
  16b8: 02 82 11 00               lwc     $r1 = [$r1]
  16bc: 00 40 00 00               nop
* 16c0: 07 00 10 01               br      $r1
  16c4: 00 40 00 00               nop
  16c8: 00 40 00 00               nop
```

Example: Jump-tables

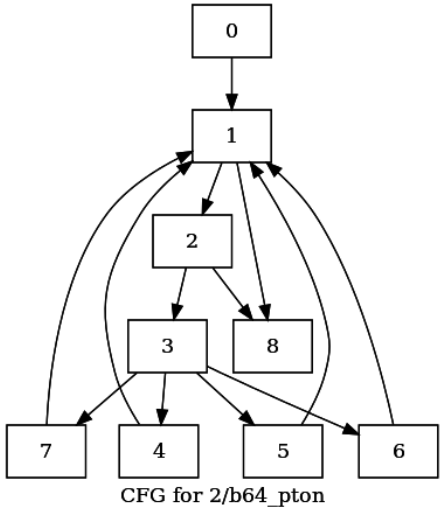
- Base64 encoding function

```
const char Base64[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
        "abcdefghijklmnopqrstuvwxyz0123456789+/";
const char Pad64 = '=';

int b64_pton(char const *src, char *target, size_t targsize)
{
  int tarindex=0, state=0;
  char *pos, ch;

  while ((ch = *src++) != '\0') {
    if (ch == Pad64) break;

    pos = strchr(Base64, ch);
    switch (state) {
      case 0:
        target[tarindex] = (pos - Base64) << 2;
        state = 1;
        break;
      case 1:
        target[tarindex]   |=  (pos - Base64) >> 4;
        target[tarindex+1]  = ((pos - Base64) & 0x0f) << 4 ;
        tarindex++;
        state = 2;
        break;
      case 2:
        target[tarindex]   |=  (pos - Base64) >> 2;
        target[tarindex+1]  = ((pos - Base64) & 0x03) << 6;
        tarindex++;
        state = 3;
        break;
      case 3:
        target[tarindex] |= (pos - Base64);
        tarindex++;
        state = 0;
        break;
      default:
        __builtin_unreachable();
    }
  }
  return (tarindex);
}
```



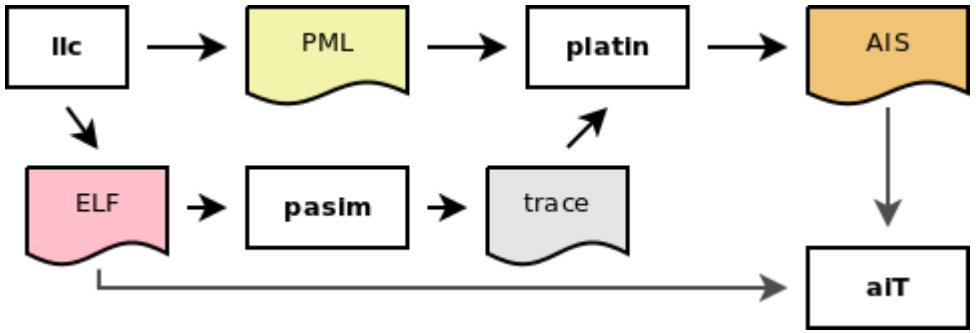
CFG for 2/b64_pton

```
.LBB2_3:
  16b0: 87 c2 10 0d 00 01 92 28   shadd2  $r1 = $r1, 102952
  16b8: 02 82 11 00               lwc     $r1 = [$r1]
  16bc: 00 40 00 00               nop
* 16c0: 07 00 10 01               br      $r1
  16c4: 00 40 00 00               nop
  16c8: 00 40 00 00               nop
```

```
instruction ".LBB2_3" + 16 bytes branches to
    ".LBB2_4", ".LBB2_5", ".LBB2_6", ".LBB2_7";
    # jumptable (source: llvm)
...
loop ".LBB2_1" max 95 ;
    # local loop header bound (source: trace)
```

Simulator Trace Analysis

- Aid during development and testing
- Extract flow information from simulation traces
  - Accompanying timing analysis at an early development stage
  - Give hints on missing static loop bounds
  - Find most relevant flow facts from trace facts
  - Incremental WCET analysis and refinement of flow facts
- Example:



Program Meta-Info Language

```
"machine-functions":
  type: seq
  desc: "list of machine-code functions"
  sequence:
    - &function
      type: map
      class: Function
      mapping:
        "name":
          type: scalar
          required: yes
          unique: yes
        "blocks":
          type: seq
          desc: "basic blocks of the function"
          sequence:
            - &block
              type: map
              desc: "basic block"
```

- YAML Based
- Central format used by `platin`
- Importer + Exporter for LLVM
- Contains all information in a single file format

- Description available as YAML schema at patmos-llvm/tools/platin/lib/core/pml.yml

PML: Platform Configuration

```
machine-configuration:
  memories:
    - name: "main"
      size: 67108864
      transfer-size: 8
      read-latency: 4
      read-transfer-time: 1
      write-latency: 4
      write-transfer-time: 1
  caches:
    - name: "data-cache"
      block-size: 32
      associativity: 4
      size: 2048
      policy: "lru"
      type: "set-associative"
  memory-areas:
    - name: "data"
      type: "data"
      memory: "main"
      cache: "data-cache"
      address-range:
        min: 0
        max: 0xFFFFFFFF
```

- Machine configuration
  - Latencies
  - Caches
  - Address ranges
- Analysis configuration
  - Program entries
  - Analysis entries
  - Custom tool configurations

PML: Structural Information

```
machine-functions:
- name: 3
  level: machinecode
  mapsto: main
  hash: 0
  blocks:
  - name: 0
    mapsto: entry
    predecessors: []
    successors:
    - 1
    - 2
    instructions:
    - index: 0
      opcode: SRESi
      size: 4
      stack-cache-argument: 8
      address: 131844
    - index: 1
      opcode: SUBi
      size: 4
```

```
bitcode-functions:
- name: main
  level: bitcode
  hash: 0
  blocks:
  - name: entry
    predecessors: []
    successors:
    - if.then
    - if.end
    instructions:
    - index: 0
    - index: 1
      opcode: r
    - index: 2
      memmode: store
```

- Functions, basic blocks, instructions
  - Branch targets, callees
  - Addresses
- Levels: bitcode, machine code
  - Structure and all analysis results are attached to a level
- Names: relate analysis results to structure within a level
- Labels: relate program points over different levels
- Relation graphs: relate CFGs to translate flow facts

PML: Flow Facts

```
flowfacts:
- scope:
    function: 3
  lhs:
  - factor: 1
    program-point:
      function: 3
      block: 1
  op: less-equal
  rhs: '0'
  level: machinecode
  origin: trace
- scope:
    function: 4
    loop: 8
  lhs:
  - factor: 1
    program-point:
      function: 4
      block: 8
  op: less-equal
  rhs: '5'
  level: machinecode
  origin: trace
```

- Linear Constraints
  - LHS: frequencies of program points
    - Function/block/instr./edge/context
  - RHS: constant or symbolic constant
    `(1 + ((-16 + %length) /u 16))`
- Scope, context
  - Scope: Function/loop
  - Context: Callsite/loop/iteration
- Various sources
  - LLVM Scalar evolution
  - Simulator traces
  - SWEET
- platin provides (internal) tools to simplify and transform flow facts

PML: Other Analysis Results

```
valuefacts:
- level: machinecode
  origin: llvm.mc
  values:
  - symbol: data
  program-point:
    function: 4
    block: 4
    instruction: 4
- level: machinecode
  origin: llvm.mc
  values:
  - symbol: data
  program-point:
    function: 4
    block: 6
    instruction: 8
```

```
timing:
- scope:
    function: 3
  cycles: 310
  level: machinecode
  origin: trace
- scope:
    function: 3
  cycles: 310
  level: machinecode
  origin: platin
  cache-cycles: 0
  profile:
  - reference:
      function: 3
      edgesource: 0
      edgetarget: 2
    cycles: 24
    wcet-frequency: 1
    wcet-contribution: 24
```

- Value facts
  - Accessed memory addresses
- Timing results
  - WCET (cycles)
  - BB exec times, BB frequencies
  - Criticalities
- Stack cache analysis
  - Spill and fill sizes

Outlook

- Source code flow annotations
  - Markers in source code
  - Flow facts as linear expressions over markers
  - Compiler transforms markers like regular instructions with side-effects
  - Order and number of 'execution' of markers stays intact on all paths
- Some changes to PML structure
  - Make it easier to add new analysis result types
- Back-annotation of analysis results to arbitrary compiler passes
- Integration of OTAWA into `platin`

Conclusion

- `platin` is our swiss army knife for compiler and analysis integration
  - Import/transform/export meta-infos
  - Invoke compiler and analysis tools
- Available at:

http://patmos.compute.dtu.dk/

http://github.com/t-crest/

Thanks! Questions?