Timing Analysis

Dr. Florian Martin
AbsInt

AbsInt.com
AIT OVERVIEW
The Timing Problem

aiT WCET Analyzer results:
• safe
• precise (close to exact WCET)
• computed automatically
• valid for all inputs and any execution context

HUGE overestimation by naive WCET methods
aiT WCET Analyzer
A Solution to the Timing Problem

- Combines global program analysis by abstract interpretation for cache, pipeline, and value analysis with integer linear programming for path analysis in a single intuitive GUI.
Input/Output

Specifications (*.ais)

clock 10200 kHz;
loop "_codebook" + 1 loop exactly 16 end;
recursion "_fac" max 6;
SNIPPET "printf" IS NOT ANALYZED AND TAKES MAX 333 CYCLES;
flow "U_MOD" + 0xAC bytes / "U_MOD" + 0xC4 bytes is max 4;
area from 0x20 to 0x497 is read-only.

Application Code

void Task (void){
    variable++; function();
    if (next) do this;
    terminate();
}

Compiler
Linker

Executable (*.elf / *.out)

Entry Point

Worst Case Execution Time
Visualization, Documentation
**Tool: exec2crl**

**Reader:** disassembles the input files (ELF, COFF, assembly, ...) into instructions

**Writer:** writes different output formats (CRL, visualization, ...)

**Decoder:** decodes the machine operations of executables and assembly files
CFG Reconstruction

- Indirect Jumps
  - Case/Switch statements as compiled by the C-compiler are automatically recognized
  - For hand-written assembly code annotations might be necessary
    \[
    \text{INSTRUCTION ProgramPoint BRANCHES TO Target}_1, \ldots, \text{Target}_n
    \]

- Indirect Calls
  - Can often be recognized automatically if a static array of function pointers is used
  - For other cases
    \[
    \text{INSTRUCTION ProgramPoint CALLS Target}_1, \ldots, \text{Target}_n
    \]
Value Analysis

• **Goal:** calculate lower and upper bounds for the values occurring in the program (addresses, registers, local and global variables)

• **Method:**
  interval analysis by Abstract Interpretation

• **Usage:**
  • cache analysis
  • detection of infeasible paths
Loop Bound Analysis

• aiT includes a loop bound analysis based on interval analysis and pattern matching that is able to recognize the iteration count of many „simple“ FOR loops automatically.

• Other loops need to be annotated
  • Example:
    ```
    loop "_prime" + 1 loop end max 10;
    ```
Cache

• The cache is a fast memory on chip.
• When the CPU wants to read/write at memory address a, it sends a request for a to the bus.
• If the block m containing a is in the cache (hit), the request for a is served in the next cycle.
• If m is not in the cache (miss), it is transferred from main memory to the cache replacing some block in the cache. The request for a is served asap while the transfer still continues.
Pipelines

Ideal Case: 1 Instruction per Cycle
Pipeline Analysis

• Goal: calculate all possible pipeline states at a program point
• Method: perform a cycle wise evolution of the pipeline, determining all possible successor pipeline states
• Implemented from a formal model of the pipeline, its stages and communication between them
• Generated from a PAG specification
• Results in WCET for basic blocks
Pipeline Analysis

• Abstract state is a set of concrete pipeline states
• I.e. we try to obtain a superset of the collecting semantics
• Sets are "small",
  since pipeline is not too history sensitive
• Joins in CFG are set union
• Allows to include (static) branch prediction
The ColdFire Pipeline

- **Fetch Pipeline** of 4 stages
  - Instruction Address Generation (IAG)
  - Instruction Fetch Cycle 1 (IC1)
  - Instruction Fetch Cycle 2 (IC2)
  - Instruction Early Decode (IED)
- **Instruction Buffer** (IB) for 8 instructions
- **Execution Pipeline** of 2 stages
  - Decoding and register operand fetching (1 cycle)
  - Memory access and execution (1 – many cycles)
Pipeline Model
Path Analysis
by Integer Linear Programming (ILP)

• Execution time of a program =

\[ \sum_{\text{Basic\_Block } b} \text{Execution\_Time}(b) \times \text{Execution\_Count}(b) \]

• ILP solver maximizes this function to get the WCET

• Program structure described by linear constraints
  • automatically created from CFG structure
  • user provided loop/recursion bounds
  • arbitrary additional linear constraints
to exclude infeasible paths
Example

if a then
  b
elseif c then
  d
else
  e
endif

Value of objective function: 19

\[
\text{max: } 4x_a + 10x_b + 3x_c + 2x_d + 6x_e + 5x_f
\]

where

\[
\begin{align*}
  x_a &= x_b + x_c \\
  x_c &= x_d + x_e \\
  x_f &= x_b + x_d + x_e \\
  x_a &= 1
\end{align*}
\]
VALUE ANALYSIS
aiT ValueAnalysis

- Purpose
- Implementation using the PAG Framework
  - Analysis Domain and Fixpoint Iteration
  - Widening and Narrowing
  - Improvements for higher precision
- Details on Interval Domain
Purpose

• Tries to find ranges of addresses for all (data-) accesses to memory.
• Tries to reduce the control flow graph by identifying unreachable code parts.
• Knowledge about the contents of registers and memory can be used to detect loop bounds.
Purpose

- **aiT**: following analyses like cache and pipeline analysis depend on the results of the ValueAnalysis.

- Quality of ValueAnalysis has a massive impact on cache and pipeline analysis:
  - Reduce the number of concurrent analysis states.
  - Allow a higher precision.
Implementation using PAG Framework
Implementation

• The ValueAnalysis computes for each program point \( P \) a function which maps all processor registers and memory cells to an interval representing its current value:

\[
\begin{align*}
\text{F}_\text{register}: & \text{ register } \rightarrow \text{ interval} \\
\text{F}_\text{memory}: & \text{ address } \times \text{ access width } \rightarrow \text{ interval}
\end{align*}
\]
Implementation

Program Points

load r4, [r3+]

F_register: r3 = 0x1234
F_memory: 0x1234:4 = 0

Computed values for Program Points

F_register: r3 = 0x1238, r4 = 0
F_memory: 0x1234:4 = 0
Implementation

• Semantics for all assembly instructions of the target architecture are modelled in an abstract form to operate on the interval domain, e.g:
  • Addition: $[3..5] + [1..5] = [4..10]$
  • Multiplication: $[3..5] * [2..3] = [6..15]$
Implementation

• aIT ValueAnalysis **iterates** on the Control Flow Graph and simulates the program’s execution using the abstract instruction semantics until a **fixpoint** is reached.

• Initial information is „**don’t know“** for all program points. Users can specify the initial information for the program’s starting point.
mov r3, #0x1234
load r4, [r3+]
store [r3], r3

F_register: r3 = ?, r4 = ?
F_memory: 0x1234:4 = 0

F_register: r3 = 0x1234, r4 = ?
F_memory: 0x1234:4 = 0

F_register: r3 = 0x1238, r4 = 0
F_memory: 0x1234:4 = 0
0x1238:4 = 0x1238
Implementation: Widening

• To speed up the fixpoint iteration we use a widening function on the interval domain.

• The $F_{\text{Register}}$ and $F_{\text{Memory}}$ functions are widened element by element.
Implementation: Widening

\[
\text{widening}((l_o,u_o), (l_n,u_n)) = \\
\begin{cases}
\text{bot, }_n &\Rightarrow \text{bot} \\
_,\text{bot} &\Rightarrow \text{bot} \\
1_l,1_u &\Rightarrow \text{if } 1_u < 1_l \text{ then } \text{bot} \text{ else } 1_l \\
\end{cases}
\begin{cases}
\text{top, }_n &\Rightarrow \text{top} \\
_,\text{top} &\Rightarrow \text{top} \\
u_1,u_2 &\Rightarrow \text{if } u_2 > u_1 \text{ then } \text{top} \text{ else } u_1 \\
\end{cases}
\]
add r3, r3, #1

F_register: r3 = 1

F_register: r3 = 2
F_register: r3 = [2..T]

add r3, r3, #1

Widening ([2..2], [2..3]) = [2..T]

F_register: r3 = 1
F_register:  r3 = [?

...  

add r3, r3, #1 

...  

F_register:  r3 = 1

Widening ([2..2], [?]) = [?]
Implementation: Widening

- To reduce the loss of precision caused by widening, we postpone widening a little bit:

\[
\text{widening'}(\text{old, new}) = \begin{cases} 
\text{new} & \text{if } \frac{\text{width(new)}}{\text{width(old)}} \geq n\% \\
\text{widening}(\text{old, new}) & \text{else}
\end{cases}
\]
Implementation: Widening

• Idea: the original widening function terminates a cycle by jumping to the unknown interval, if it is enlarged during one iteration of a cycle.

• Since not all cycles are unbounded we hope that the fixpoint is reached before the changes are too small to be postponed.
F_register: r3 = [2]

add r3, r3, #1

Widening ([2..2], [2..3]) = [2..3]

F_register: r3 = 1
Implementation: Widening

- Alternatively: enlarge interval bounds to the next power of two:

\[
\text{Widening } ([2,2], [2,n]) = [2,m] \\
\text{where } m = 2^x \geq n
\]
Implementation: Narrowing

• Narrowing is used as a counterpart to widening and makes the analysis results more precise.

• Similar as in widening we don’t accept too small changes to increase analysis performance.
Implementation: Narrowing

Narrowing ((l_o, u_o) as old, (l_n, u_n) as new) =
   case l_o,u_o of
       (bot, top) => old;
       (bot, _)   => if l_n = bot then old else new endif;
       (_, top)   => if u_n = top then old else new endif;
       _          => if width(new)/width(old) <= n% then new
                   else old
                   endif
   endcase;
Interval Domain
Interval Domain

• Intervals consist of a lower and upper bound:

\[ \text{value} = \text{flat (snum)} \times \text{flat (snum)} \]
Interval Domain

• Valid intervals:
  • [-3, 4] [0, 0] [1, T]
  • [⊥, T] (top, unknown interval)
  • [T, ⊥] (bot, neutral element for union, widening)

• Illegal intervals:
  • [4, -3] [T, 1] [0, ⊥]
Interval Domain

- Using the described interval domain makes some operations unprecise whenever the input operands are not exactly known:
  - Bitoperations (logical and, or, xor, …)
  - Division and modulo operations
  - SIMD operations / MACC units
Interval Domain: SIMD

• Legacy architectures often use splitted registers (x86: register AX with subparts AL and AH)

• Newer architectures (e.g. TriCore) also use similar schemes in MACC Units or for integer division (division and modulo result in lower/upper half of target register)
AX = AH * 256 + AL

256 = 1 0000 0000
257 = 1 0000 0001
258 = 1 0000 0010
... 
511 = 1 1111 1111
mov [0x1234], AH
mov [0x1236], AL

F_register: AH = [1..4]
            AL = [0]

F_register: AH = [1..4]
            AL = [0]
F_memory:   0x1234:2 = [1..4]
            0x1236:2 = [0]

F_register: AH = [1..4]
            AL = [0]
F_memory:   0x1234:2 = [1..4]
            0x1236:2 = [0]

mov BX, [0x1234]

F_register: AH = [1..4]
            AL = [0]
F_memory:   BX = [256..1024]
            0x1234:2 = [1..4]
            0x1236:2 = [0]
Improving Precision
Improvements: Calling Convention

• Writing to memory at unknown addresses invalidates all knowledge about memory contents (RAM)

• Unknown writes may happen e.g. in loops where the loop counter becomes unknown if the loop is not completely unrolled

```c
for (i=0; i < 100; i++)
    array [i] = 0;
```
Improvements: Calling Convention

• Usually compilers implement a calling convention, where a fixed set of registers is stored/restored on the stack when a function call happens.

• Inexact write accesses to memory therefore may also invalidate the saved registers on the stack => loss of precision.
Improvements: Calling Convention

• The PAG framework introduces a local-edge and some artificial nodes in the Control Flow Graph to propagate invariants through function calls.
call $FUNCTION

CALL

safe registers
call local

RETURN

... return

... return

$FUNCTION

all registers

unsafe registers
Stackpointer

- Stackpointer is one of the most important registers, since all variables in the local stackframe are referenced through SP.
- Usually the stackpointer is not modified by normal function calls and is hence a safe register according to calling conventions.
void main (void) {
    int i, j, k;
    sub SP, 12 ; 12 bytes variables
    ... 
    i = 0;
    mov [SP+8], #0 ; i accessed via SP
    ... 
    return;
    ... 
    # store safe registers 
    # restore safe registers
}

main:
External Knowledge
External Knowledge

• aiT ValueAnalysis improves its quality by using information about:
  • Contents of memory areas that remain constant during program execution:
    • ROM
    • Dataportions inside the program code
  • Refinement of register values to known ranges, e.g. refining a loop counter register to all possible values.
External Knowledge

• Information provided by the user (AIS annotations):
  • Contents of registers and memory cells at certain program points.
  • Annotation of memory accesses to avoid destruction of information through unknown write accesses to memory.
Infeasible Paths
Infeasible Paths

Analysis results can be greatly improved by:

• Finding infeasible paths
• Splitting information at conditional branches
Infeasible Paths

- A path $p_1 \rightarrow p_2 \rightarrow \ldots \rightarrow p_n$ in the Control Flow Graph is infeasible if the program execution does never follow this path for all possible inputs.

- ValueAnalysis can detect infeasible path which are caused by infeasible branches, i.e. paths that start at branches which are always true or false.
Finding infeasible paths

• ValueAnalysis analyzes the behavior of compare and conditional branch instructions to find infeasible path:

```assembly
mov r3, 0x42
...  
cmp r3, 0x10  # compare 0x42 with 0x10
blt target  # branch is always infeasible
...  # since 0x42 is larger than 0x10
```
Finding infeasible paths

- Infeasible paths for computed branches can be detected by comparing the abstract information about the computed address (usually a register) with the possible successors in the control flow graph (address of instruction in control flow).
- Mismatching addresses indicate infeasible paths.
Switchtable

switch (i) {  
  case 0:  
    ...  
    break;  
  case 1:  
    ...  
    break;  
  case 2:  
    ...  
  }

# compute address of target label in r3  
  b r3  # branch to target label

L0:  
...  
  # first case, address X1

L1:  
...  
  # second case, address X2

L2:  
...  
  # third case, address X3

end:

Mark branches as infeasible if r3 != X1, X2 or X3
Splitting Information

- At conditional branches information can be split, i.e. for each outgoing edge remove that part of information that would lead to an infeasible path.
Splitting Information

mov r3, <unknown value>

cmp r3, #0x10

blt $target

F_register: r3 = ?

F_register: r3 = ?

F_register: r3 = [0x10...]

F_register: r3 = [..0x9]
Observations

• The quality of the ValueAnalysis depends on:
  • The target architecture: PowerPC and other load/store architectures are very easy to analyze, 16bit legacy architectures like C16x are harder (splitted registers, many addressing modes, ...).
Observations

• C-Compilers use a very small set of instructions and processor features to compute memory addresses. Usually implementing add/sub/mul instructions together with the address computation of load/stores is sufficient for first prototypes.

• Address computations are very „local“, i.e. most information used to compute a memory address can be derived from instructions near the load/store.
Summary

aiT ValueAnalysis:

• Interval based domain for registers and memory cells.
• Abstract Semantics for all processor instructions.
• Fixpoint iteration on Control Flow Graph.
• Refinement through context based analysis approach and detection of infeasible paths/splitting of information.
• User can improve analysis through annotations.
CACHE ANALYSIS
Example: Direct Mapped I-Cache

CPU

Program Counter:
1038

Instruction:
ble 1024

I-Cache

1032: ble 1024
1028: mul …

Main memory

1024: add …
1028: mul …
1032: ble 1024

Cache Hit: ~ 1 Cycle

Cache Miss: ~ +1 to +100 Cycles
Cache Memories

• Improve access times of fast microprocessors to slow memories

• Store recently referenced memory blocks (principle hope)

• Hit time $\sim 1$cycle
  Miss penalty $\sim 1$-$100$ cycles
Principle of Locality

• Caches work on the basis of the principle of locality of program behavior.

• Programs tend to reuse data and instructions they have used recently.

• Rule of thumb: Programs spend 90% of their execution time in only 10% of the code.
Principle of Locality

There are the following key aspects:

- **Temporal locality** -- Recently accessed instructions and data are likely to be accessed in the near future.

- **Spatial locality** -- Instructions and data whose addresses are near one another tend to be referenced close together in time.

- **Sequentiality** -- This is a special case of spatial locality. Given an execution of a reference to an address \( s \), it is likely that a reference to address \( s+1 \) is made in the near future.
Why Are Caches Used?

• Performance
  • Smaller physical sizes of cache memories with respect to the size of main memories allow for faster access, as signals have to be propagated over shorter distances.

• Costs
  • Fast memory
    • Tends to be expensive
    • Has high power consumption
    • Is physically larger than slower memory
Direct-mapped Cache
Direct-mapped Cache

Advantages
- simple
- cheap

Disadvantages
- bad performance by unlucky code layout
Fully Associative Cache

CPU

Address:

Block address Byte in line

fully associative cache of A elements with LRU, FIFO, or random replacement strategy

1
Block addr. Tag Rep Data block

2
Block addr. Tag Rep Data block

...A

Compare block address
If not equal, fetch block from memory

Main Memory

Byte select & align

Data Out
Fully Associative Cache

Advantages
• good performance

Disadvantages
• complicated
• expensive
Set Associative Cache

CPU

Address:

Address
prefix
Set
number
Byte in
line

1
2
... A

Addr. prefix | Tag | Rep | Data block | Addr. prefix | Tag | Rep | Data block
---|---|---|---|---|---|---|---

Set: Fully associative subcache of A elements with LRU, FIFO, rand. replacement strategy

Compare address prefix
If not equal, fetch block from memory

Byte select & align

Data Out

Main Memory
Cache Memories

key parameter:

• Size,

• line length,

• level of associativity

• replacement strategy
Cache Analysis

• Must Analysis:
  For each program point and calling context, find out which blocks *are* in the cache

• May Analysis:
  For each program point and calling context, find out which blocks *may be* in the cache
Example:
Fully Associative Cache (2 Elements)
# Result of the Cache Analyses

## Categorization of memory references

<table>
<thead>
<tr>
<th>Category</th>
<th>Abb.</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>always hit</td>
<td>ah</td>
<td>The memory reference will always result in a cache hit.</td>
</tr>
<tr>
<td>always miss</td>
<td>am</td>
<td>The memory reference will always result in a cache miss.</td>
</tr>
<tr>
<td>not classified</td>
<td>nc</td>
<td>The memory reference could neither be classified as ah nor am.</td>
</tr>
</tbody>
</table>
Abstract Semantics: Transfer

concrete

\[
\begin{array}{ccc}
  z & s & y \\
  y & z & x \\
  x & y & t \\
  t & x & s \\
\end{array}
\]

abstract

\[
\begin{array}{ccc}
  \{x\} & \{s\} & \{x\} \\
  \{\} & \{t\} & \{\} \\
  \{s,t\} & \{y\} & \{y\} \\
\end{array}
\]

Age

"young"

"old"
Abstract Semantics: Join

**Join (must)**

```
{a}  
{ }  
{c, f}  
{d}  

{c}  
{e}  
{a}  
{d}  
```

“intersection + maximal age”

Interpretation: memory block a is definitively in the (concrete) cache

=> always hit

Question: How many references will a memory block surely survive in the cache?
Abstract Semantics: Join

Join (may)

Interpretation: memory block \( s \) not in the abstract cache \( \Rightarrow s \) will definitively not be in the (concrete) cache

\( \Rightarrow \) always miss

"union + minimal age"

Question: How many references will a memory block maximally survive in the cache?
Cache Analysis: Results