Worst-case Execution Time Analysis and Optimizing the Worst-case

Sudipta Chattopadhyay
(Guest lecture: Advanced Compiler Construction course)

Real-time Applications

- Real-time applications have temporal constraints
  - Hard real-time systems
  - Soft real-time systems
- System-level analysis for hard real-time systems
  - Application contains many tasks/programs
  - Check whether application meets deadline
- Task-level analysis
  - Worst-case execution time (WCET) of a task over all inputs
  - Analysis to obtain WCET

Obtaining WCET

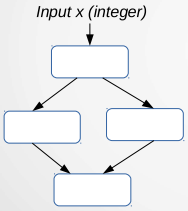
- How do we calculate WCET of a program?
  - Exhaustively enumerating all inputs?
  - Consider an image processing application

Obtaining WCET

- How do we calculate WCET of a program?
  - Exhaustively enumerating all inputs?
    - *Until the end of the world :-)*
    - Consider an image processing application
  - Systematically generating inputs?
    - Can we give hard guarantees?

Obtaining WCET

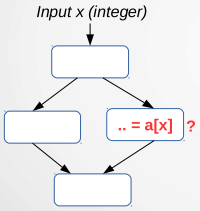
- But wait a minute.....
  - But do we need all inputs?



Why don't we just test 2 paths and take the maximum of them?

Obtaining WCET

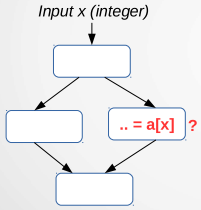
- But wait a minute.....
  - But do we need all inputs?



Why don't we just test 2 paths and take the maximum of them?

Obtaining WCET

- But wait a minute.....
  - But do we need all inputs?

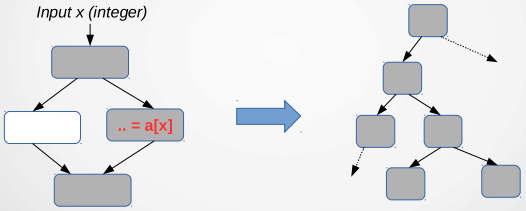


Why don't we just test 2 paths and take the maximum of them?

Different inputs may access different memory blocks, which might incur different latency to fetch from memory subsystem

Obtaining WCET

- Besides.....
  - What you see is not what you execute (WYSINWYX, Balakrishnan et al., ACM TOPLAS 2010)

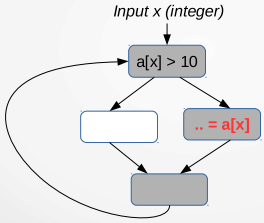


Source code

Many possible executions of the binary code

Obtaining WCET

- And.....
  - We still have exponential blow-up

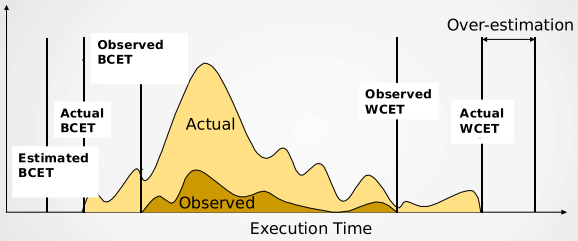


Source code with exponential number of paths

Obtaining WCET

- Static analysis
  - Does not require inputs
  - Estimates an upper bound on execution time without executing a program.....
    - *Sounds too good to be true?*
  - How expensive?
  - How accurate?
  - What are the limitations?

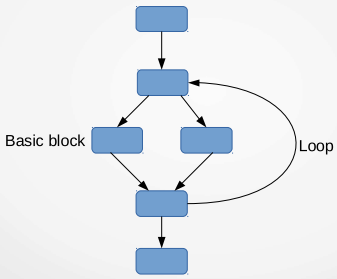
WCET



WCET = Worst-case Execution Time
BCET = Best-case Execution Time

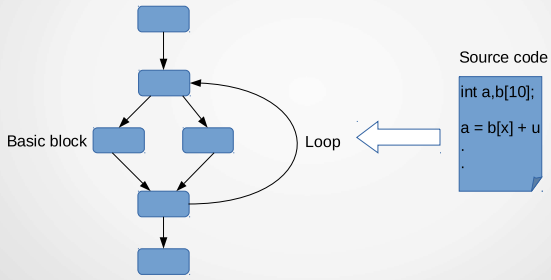
WCET calculation

- Control flow graph (CFG) representation of a program



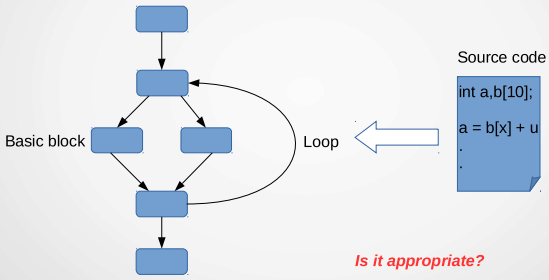
WCET calculation

- Control flow graph (CFG) representation of a program



WCET calculation

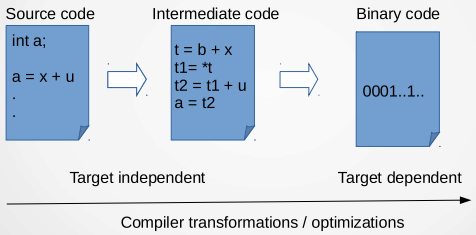
- Control flow graph (CFG) representation of a program



Is it appropriate?

WCET calculation

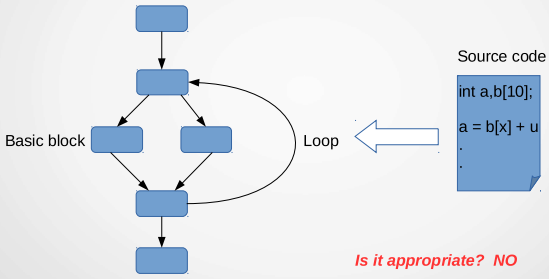
- Control flow graph (CFG) representation of a program



Timing is target dependent. Any doubt?

WCET calculation

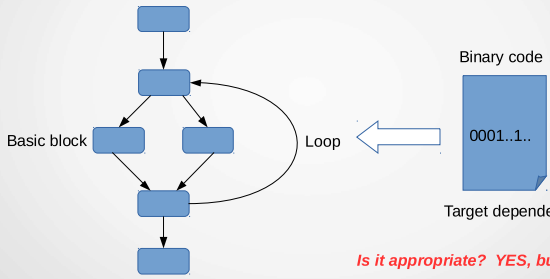
- Control flow graph (CFG) representation of a program



Is it appropriate? NO

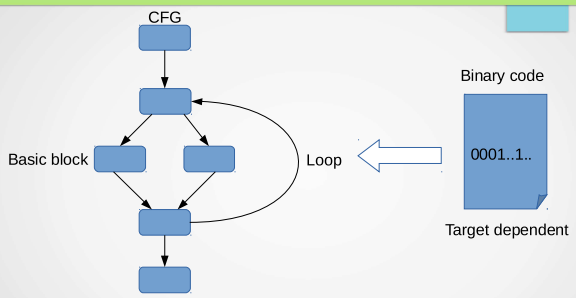
WCET calculation

- Control flow graph (CFG) representation of a program



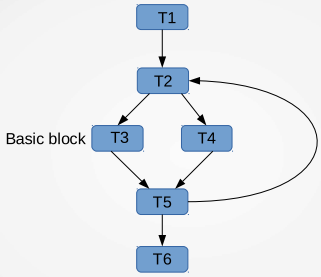
Is it appropriate? YES, but.....

Typical WCET Derivation



- *Derive WCET of basic blocks (Micro-architectural modeling)*
- *Use WCET of basic blocks to derive the longest path in the CFG (calculation)*

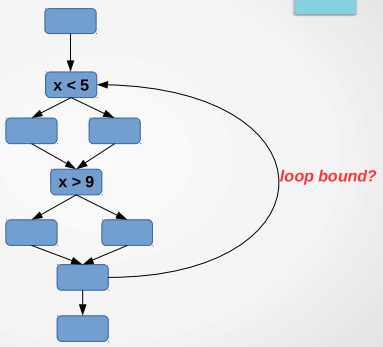
WCET Calculation



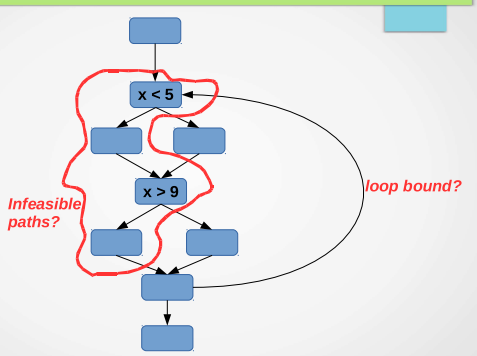
Find the path that maximizes the sum of Ti

Hmmmm, but isn't this undecidable?

WCET Calculation – not so fast



WCET Calculation – not so fast

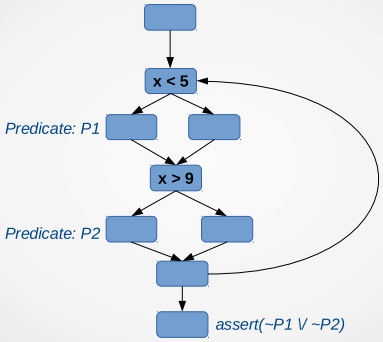


- Longest path with all loops having bounded length
- How do we eliminate infeasible paths?

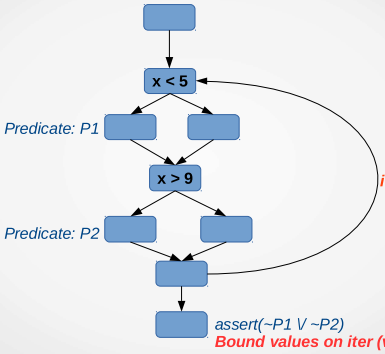
Loop bounds and infeasible paths

- Finding loop bounds
  - Static analysis
  - Simulations (might be unsafe)
  - Explicitly provided by programmer
- Finding infeasible paths
  - Model checking, symbolic execution
  - Explicitly provided by programmer
- Integrating loop bounds and infeasible path information into WCET calculation

Loop bounds and infeasible paths



Loop bounds and infeasible paths

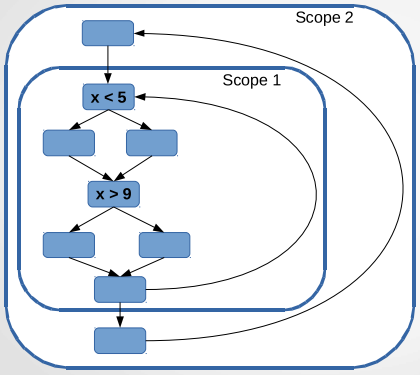


Bound values on iter (value analysis)

Flow facts (Ermedahl et al., RTSS 2000)

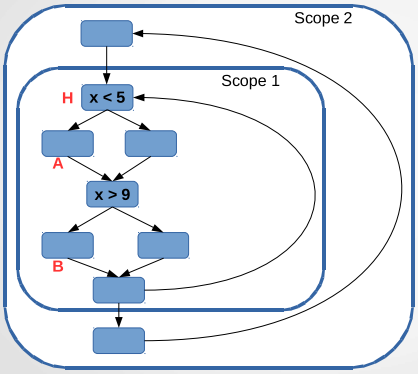
- A generic way to specify program flow information
  - Loop bounds and infeasible paths
- Flow fact structure
- Derivation of flow facts
- Integration of flow facts into WCET calculation

Flow fact structure



Scope-based flow facts

Flow fact structure



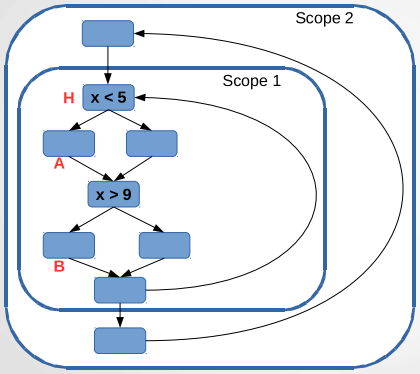
[]: total

<>: for each iteration

Loop bound:

Scope 1: []: XH <= 100

Flow fact structure



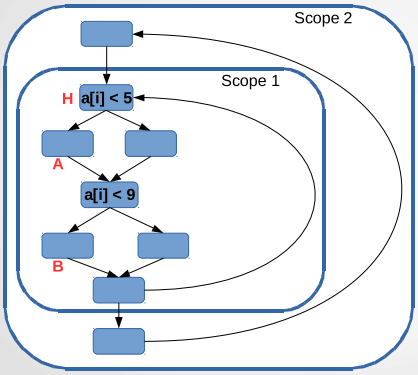
[]: total

<>: for each iteration

Infeasible path:

Scope 1: <>: XA + XB = 1
Scope 2: <>: XA + XB = 1

Flow fact structure



[]: total

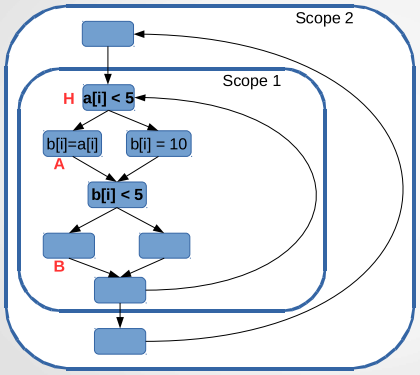
<>: for each iteration

Infeasible path:

Scope 1: <>: XA <= XB

Execution of A implies Execution of B.

Flow fact structure



[]: total

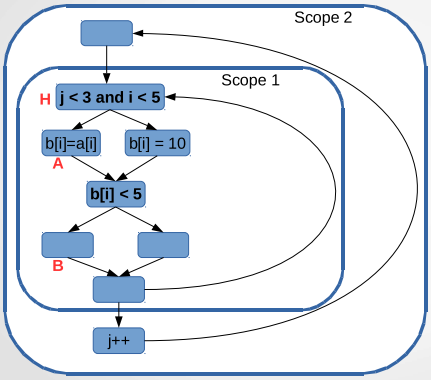
<>: for each iteration

Infeasible path:

Scope 1: <>: XA = XB

A and B are executed in the same iteration.

Flow fact structure



[]: total

<>: for each iteration

Infeasible path (range):

Scope 1: <1..2,1..4>: XA = 1

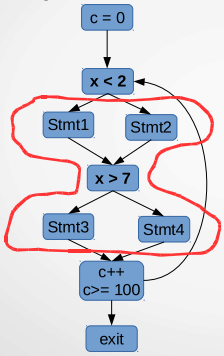
A must execute for first 4 iterations of Scope 1 for first two iterations of Scope 2.

IPET based WCET calculation

- Implicit path enumeration via integer linear programming (ILP)
- Current state-of-the-art in most WCET analyzers
- Still an NP-complete problem, however, efficient solvers exist for ILP, such as CPLEX

IPET based WCET calculation

- Modeling control flows



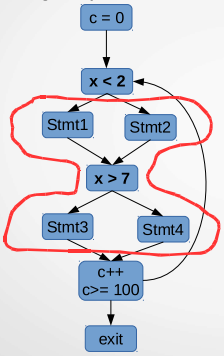
E(Stmt1,x>7) + E(Stmt2,x>7)
= N(x>7)
= E(x>7, Stmt3) + E(x>7, Stmt4)

E(v1,v2) = Number of times the control flow edge (v1,v2) is executed

N(v) = Number of times the control flow node v is executed

IPET based WCET calculation

- Modeling loop bounds



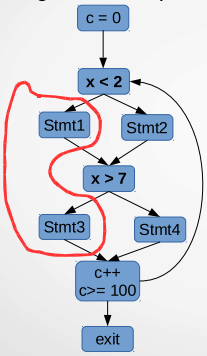
- Recall flow facts

- Flow facts for loop bounds can be modeled as ILP constraint

- N(x < 2) <= 100

IPET based WCET calculation

- Modeling infeasible path



- Recall flow facts

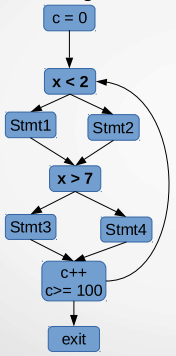
- Stmt1 and Stmt3 can never be executed in the same iteration of the loop

- Flow fact:
<>: x(stmt1) + x(stmt3) <= 1

- ILP constraint
N(Stmt1) + N(Stmt2) <= 100

IPET based WCET calculation

- Overall modeling



Objective:
Maximize
T(c=0)*N(c=0) + T(x<2)*N(x<2)
+ T(Stmt1)*N(Stmt1)
+ T(Stmt2)*N(Stmt2)
+ T(Stmt3)*N(Stmt3)
+ T(Stmt4)*N(Stmt4)
+ T(x>7)*N(x>7)
+ T(c++;c>=100)*N(c++;c>=100)

Subject to:
N(x < 2) <= 100 [loop bound]

N(Stmt1) + N(Stmt2) <= 100 [infeasible path]

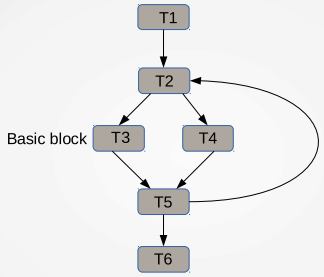
E(Stmt1,x>7) + E(Stmt2,x>7) = N(x>7)
= E(x>7, Stmt3) + E(x>7, Stmt4) [control flow]

Similar control flow constraints….

IPET based WCET calculation

- Advantages and disadvantages
  - Accuracy
    - *Good as long as the constraints are modeled appropriately*
  - How difficult is it to integrate flow facts?
    - *Most of the flow facts could be integrated as ILP constraints*
    - *However, approximated sometimes*
  - Scalability
    - *NP complete, however efficient solvers exist*

WCET Calculation

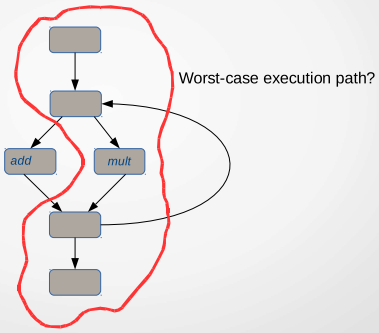


- Find the path that maximizes the sum of Ti (So far)

- Getting the value of Ti (now)

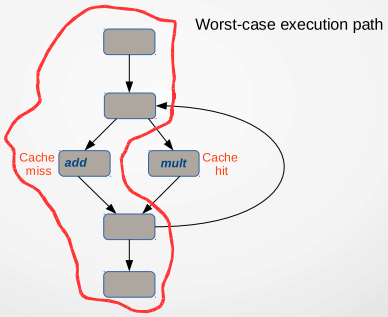
Why?

- Why do not we annotate maximum execution time of each instruction



Why?

- Because....



Micro-architectural Modeling

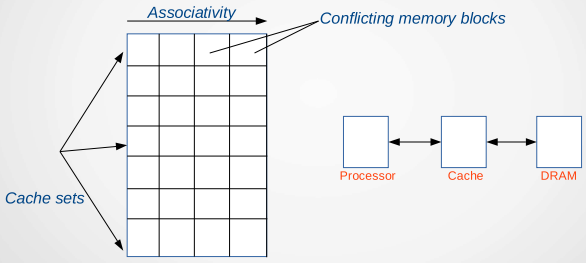
- Need timing models of underlying micro-architecture
  - Caches
  - Pipeline
  - Branch predictors
  - ?

Modeling caches

- Why model caches?
  - Caches play a significant role in improving performance
  - Caches in modern processors (ARM, X86) are several magnitudes faster than main memory (DRAM)
  - Not modeling caches does not mean caches don't perform at runtime
    - Imprecision in the analysis

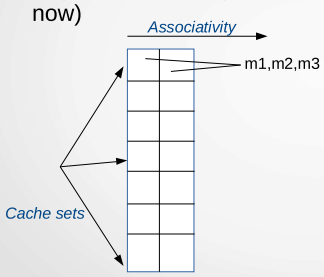
Modeling Instruction Caches

- Consider cache as a matrix



Instruction Cache Modeling

- Cache sets are modeled independently
- Depends on the replacement policy (we consider LRU for now)



Cache states:

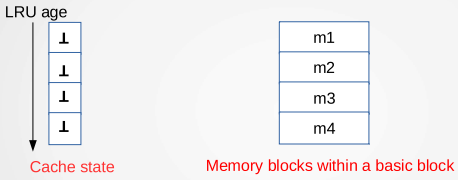
X,X
X,m1
X,m2
X,m3
m1,m2
m2,m1
m2,m3
m3,m2
m1,m3
m3,m1

- No need to enumerate cache states

Instruction Cache Modeling with AI

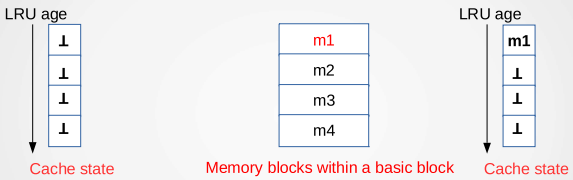
- Abstract interpretation (AI)
  - A static analysis framework (already in lectures)
  - Abstract and concrete states of programs
  - We shall talk about cache states within a program
- Predicts cache hits and cache misses
  - AH – always a cache hit (called *must analysis*)
  - AM – always a cache miss (called *may analysis*)
  - NC – unknown (remember it is a static analysis)

Instruction Cache Modeling with AI



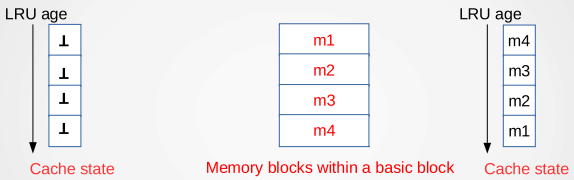
We need to formulate how the cache states are transformed by accessing each memory block (transfer operation in AI)

Instruction Cache Modeling with AI



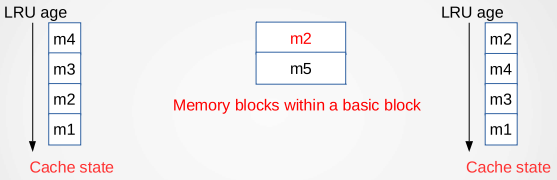
We need to formulate how the cache states are transformed by accessing each memory block (transfer operation in AI)

Instruction Cache Modeling with AI



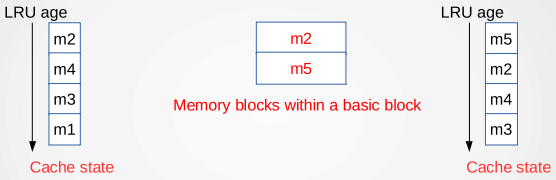
We need to formulate how the cache states are transformed by accessing each memory block (transfer operation in AI)

Instruction Cache Modeling with AI



We need to formulate how the cache states are transformed by accessing each memory block (transfer operation in AI)

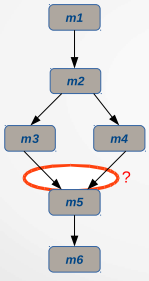
Instruction Cache Modeling with AI



We need to formulate how the cache states are transformed by accessing each memory block (transfer operation in AI)

Instruction Cache Modeling with AI

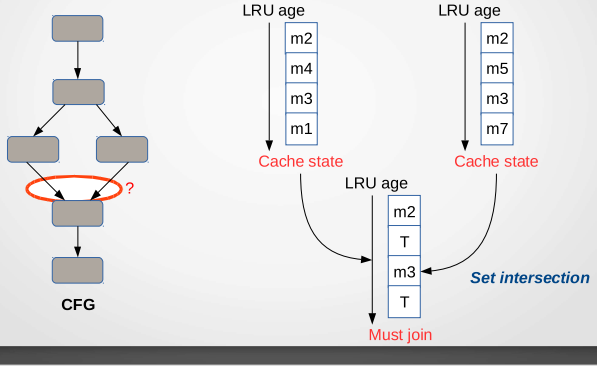
- How to handle control flow merge points?



Depends on the type of analysis
- *Must analysis* (to predict definite cache hits)
- *May analysis* (to predict definite cache misses)

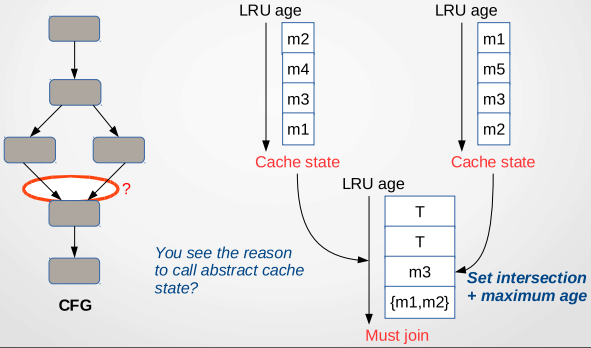
Must Analysis

- How to handle control flow merge points?



Must Analysis

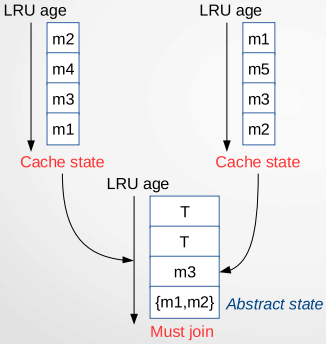
- How to handle control flow merge points?



You see the reason to call abstract cache state?

Must Analysis

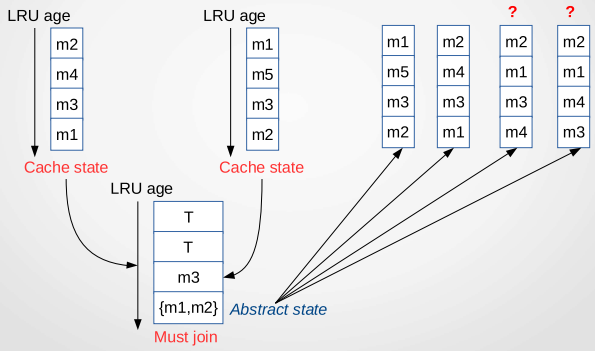
- Abstract interpretation



What are the concrete states?

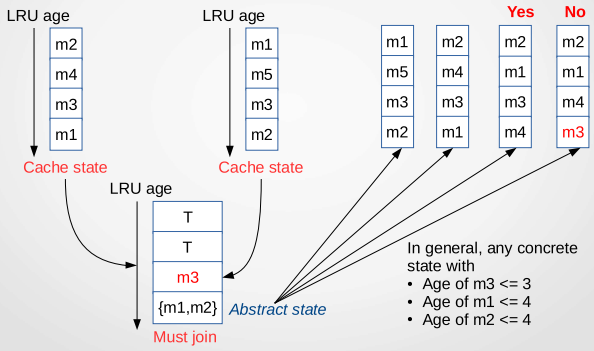
Must Analysis

- Abstract interpretation



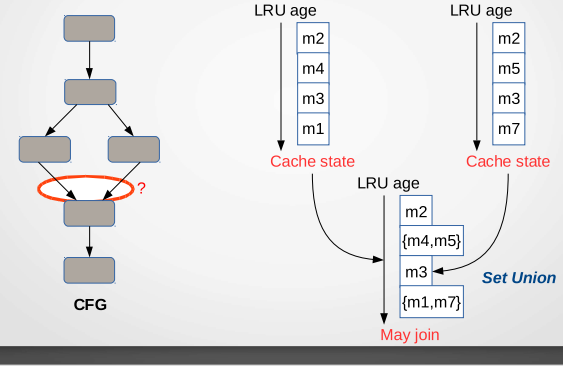
Must Analysis

- Abstract interpretation



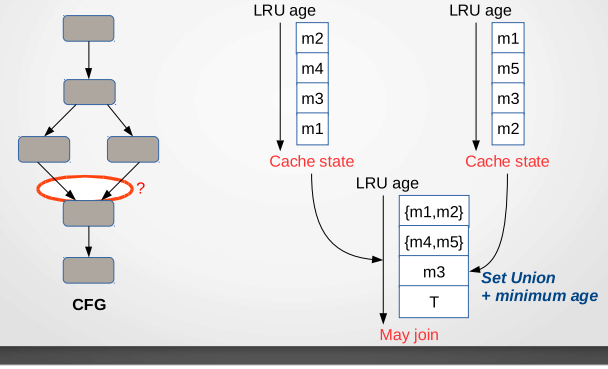
May Analysis (Predicting cache misses)

- How to handle control flow merge points?



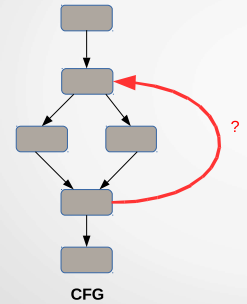
Must Analysis

- How to handle control flow merge points?



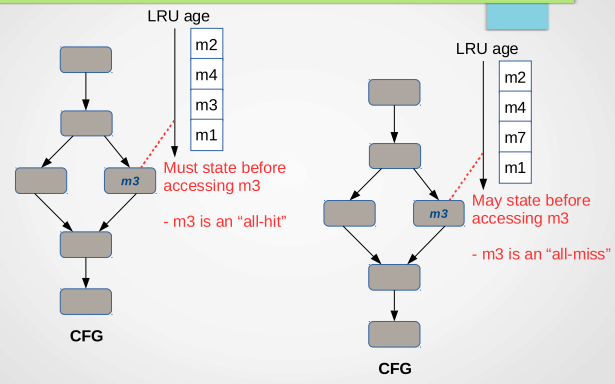
Instruction Cache Modeling with AI

- How to deal with loops?



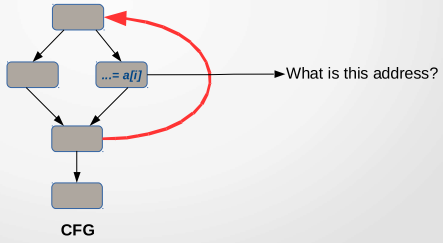
- Loops are special type of merge points
- Solutions are repeated
- Fixed point is obtained on the cache states over all points
- *Not loop unrolling*

Categorizing hits and misses



Data Cache Modeling

- What is the main difference with instruction cache modeling?
  - Instruction address is program counter value
  - Data address need to be tracked



Data Cache Modeling

- Difficulties
  - Needs separate analysis to know the set of addresses tracked – value analysis
  - Value analysis at binary code is difficult
    - Absence of variable information
    - Register values can be faithfully tracked
    - But need a model of memory, to capture the value

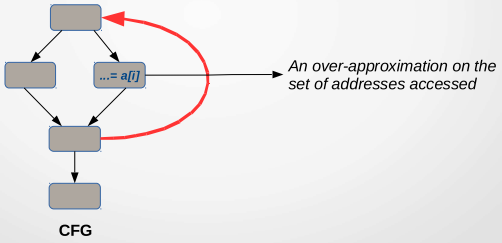
Data cache modeling

- r1 ← r2+r3
- r5 ← Load (r1 + 100)
- r4 ← r5 + 6
- r7 ← Load (r4 + 1)

  //requires value at memory location r1+100

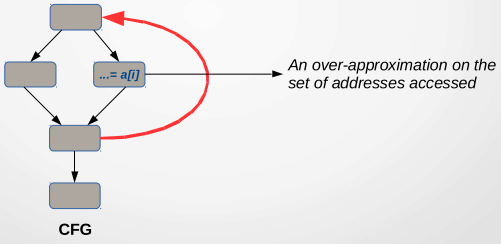
Data Cache Modeling

- Difficulties
  - Value analysis may compute an over-approximation on the set of addresses accessed by each load/store



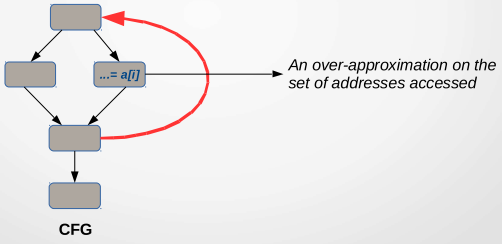
Data Cache Modeling

- Difficulties
  - Not all addresses are accessed in all iterations
  - Instruction addresses do not change based on loop iterations



Data Cache Modeling

- Difficulties
  - Modeling write-policies
    - Write through
    - Write-back



Data Cache Modeling (Summary)

- Accurate data cache modeling is a very difficult problem
  - Most results so far are highly pessimistic
    - Difficulty due to address analysis at binary level
    - Difficulty in distinguishing memory blocks at different loop iterations
    - Difficulty in modeling write-back policies
    - Data caches with non-LRU replacement policy is almost unexplored
  - A different solution is to avoid data cache and use software controlled scratchpad memory (next lecture)

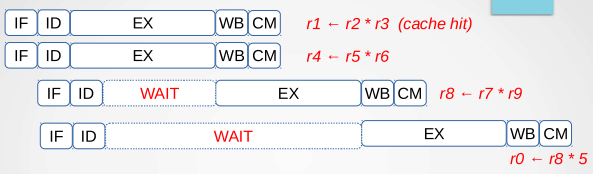
Pipeline delay computation

- Instruction cache
  - Accessed at “IF stage” of the pipeline
- Instruction All-hit (AH)
  - *T(IF) = T(IF) + [1,1]*
- Instruction All-miss (AM)
  - *T(IF) = T(IF) + [100,100] //100 cycle miss penalty*
- Instruction unclassified (NC)
  - ***T(IF) = T(IF) + [100,100]?***

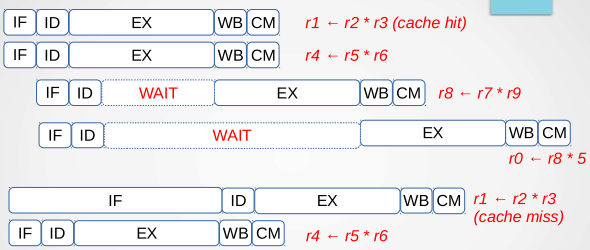
Integration of Cache and Pipeline Model

- Instruction cache
  - Accessed at “IF stage” of the pipeline
- Instruction All-hit (AH)
  - *T(IF) = T(IF) + [1,1]*
- Instruction All-miss (AM)
  - *T(IF) = T(IF) + [100,100] //100 cycle miss penalty*
- Instruction unclassified (NC)
  - ***T(IF) = T(IF) + [100,100]? NO***

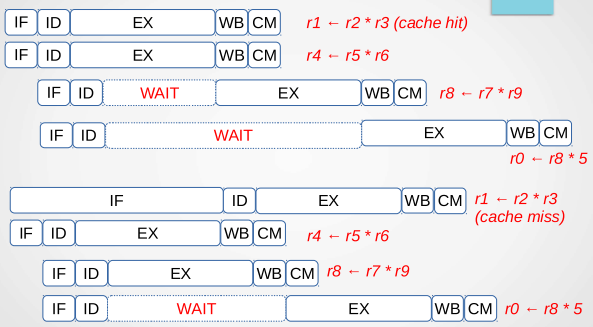
Timing Anomaly



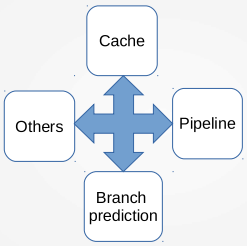
Timing Anomaly



Timing Anomaly

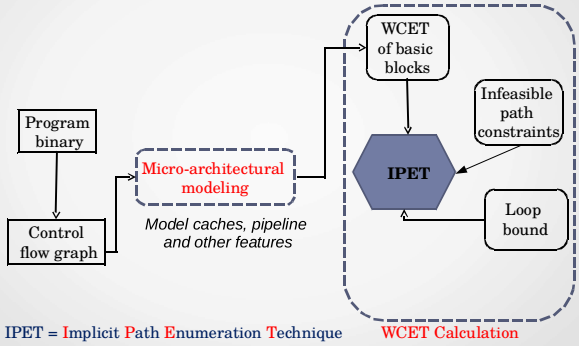


Dependency among analyses



- *Analyses of different micro-architectural entities are not independent*
- *Compositional analyses*
  - *At cost of accuracy*

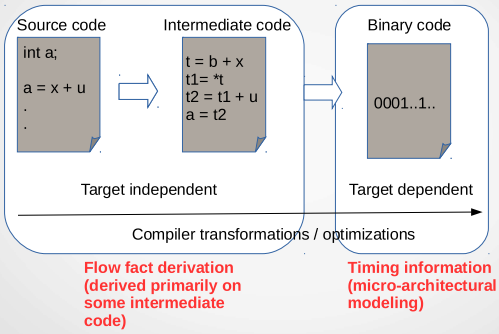
WCET Analysis Framework



Current Trend of Research

- A fairly active research area
  - WCET analysis in multi-core era
    - Resource sharing
  - WCET analysis for multi-threaded programs
    - Synchronizations
    - Coherence
  - Modeling timing delays for multi-tasking system
    - Effect of preemption
    - Effect due to the presence of supervisory software (operating systems)
  - Systematic input generation to obtain near worst-case performance
    - Soft real-time applications
    - Eliminates the need to statically model hardware
  - .....

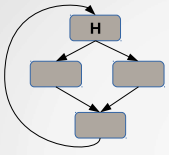
Recall flow facts



Flow facts in Compilation

- Flow facts are easier to derive on source/intermediate code
  - Variable information is present
- Programmers are more likely to specify constraints on source code
  - Annotating object code is cumbersome
- Compiler transformations may destroy flow facts.....

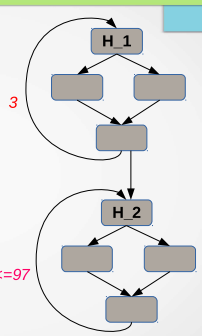
Flow Facts



Scope 1 (source code)

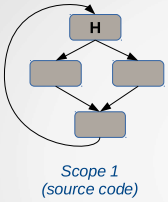
[]: total
<>: for each iteration

Loop bound:
Scope 1: []: XH <= 100



(object code: loop is unrolled 3 times)

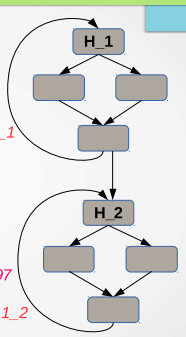
Flow Facts



Scope 1
(source code)

[]: total
<>: for each iteration

Loop bound:
Scope 1: []: XH <= 100



3
Scope 1_1

<=97
Scope 1_2

Scope 1_1: []: XH_1 = 3
Scope 1_2: []: XH_2 <= 97

(object code: loop is unrolled 3 times)

Other Code Annotations

- Some code annotations could be provided to capture the range of values
  - e.g. *Variable "c" takes values between 1 and 10*

```
func (c) {
    … = a[c] //access address a+[1,10]
}
```

Value annotations may greatly simplify data cache modeling!!!

  - However, error-prone and labor intensive to perform such annotations in object code

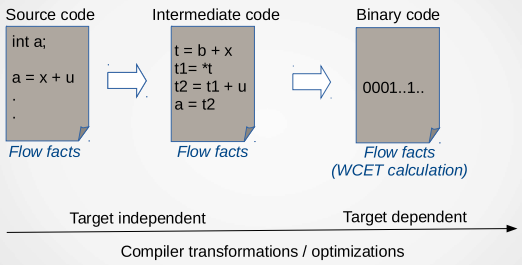
Code Annotations for WCET Analysis

- Flow facts
  - Loop bounds
  - Infeasible paths
- Range of values for variables
  - Helps in data cache modeling
  - Finding infeasible paths
- Addresses of indirect jumps
  - Difficult to obtain at binary level
  - Used for CFG reconstruction
  - Examples: function pointers

Compilation and Code Annotations

- Why a compiler can help?
  - Compiler knows different levels of code representations
    - Source code
    - Intermediate code
    - Object code
  - Compiler knows which transformations are applied
  - A compiler can translate annotations after each transformation
  - In practice, most code annotations can be translated by a compiler

Flow Fact Translation

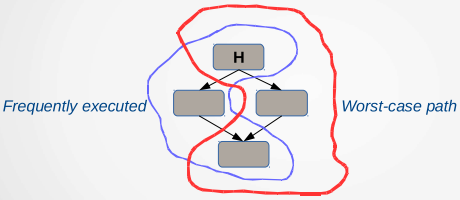


Optimizing the worst-case

- WCET analysis can benefit from a compiler
  - Translating code annotations
- A compiler can optimize the worst-case performance of a program
  - Improves the WCET bound
  - Traditional compiler optimizations aim to optimize average-case performance

Worst-case Optimizations

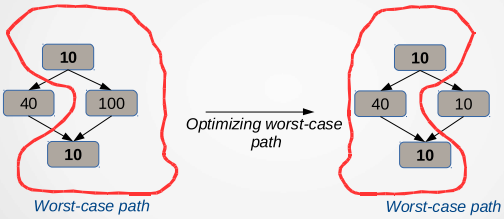
- Fundamental differences with other optimizations



Optimizing a frequently executed path may not improve the worst-case

Worst-case Optimizations

- Fundamental differences with other optimizations



Worst-case path may change, commonly known as worst-case path switching

Worst-case Optimizations

- Worst-case optimizations aim to improve WCET bound
- Common optimizations
  - Scratchpad allocation
  - Cache locking
  - Procedure positioning
  - Procedure cloning

Worst-case optimizations

- Fundamental differences with other optimizations
  - Sometimes, does not really improve performance
  - However, makes the WCET analysis simpler and the analyzer more accurate

Worst-case Optimizations

```
f(x) {
    for(i = 0; i < x; i++) {
        //do some work
    }
}
main() {
    f(10);
    f(100);
}
```

- The same function "f" is called for two different values 10 and 100
- The parameter is used as a loop bound
- ***How would be the flow fact for the loop in function "f"?***

Worst-case Optimizations

```
f(x) {
    for(i = 0; i < x; i++) {    /* Scope: []: XH <= 100 */
        //do some work
    }
}
main() {
    f(10);
    f(100);
}
```

- The same function "f" is called for two different values 10 and 100
- The parameter is used as a loop bound
- ***How would be the flow fact for the loop in function "f"?***
- ***WCET analysis will assume that both invocations of function "f" may execute the loop at most 100 times***
  - ***A pessimistic assumption***
  - ***Compiler may transform the code for better annotations***

Worst-case Optimizations

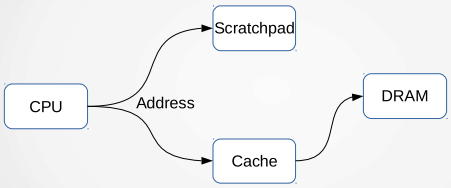
```
f_10() {
    for(i = 0; i < 10; i++) {    /* Scope: []: XH_10 <= 10 */
        //do some work
    }
}
f_100() {
    for(i = 0; i < 100; i++) {    /* Scope: []: XH_100 <= 100 */
        //do some work
    }
}

main() {
    f_10();
    f_100();
}
```

- *Function "f" is clones twice*
- Commonly known as Procedure Cloning Optimization

Scratchpad Memory

- What is it?
  - A software controlled memory (typically, SRAM)
  - Visible to the application programmer
  - Mapped in the same virtual address space of the processor
  - Many names
    - Local memory in Cell processor
    - Shared memory in GPGPU

Scratchpad Memory



Typically, from the address, it can be determined whether the memory block is fetched from scratchpad

Why Scratchpad?

- Software controlled
  - Every memory access is predictable
  - Improves accuracy of WCET analysis
- A promising approach to replace data cache in embedded systems
  - Practically eliminates the need of complex value/address analysis
- Energy efficient compared to caches
  - Does not require tag comparison logic. *WHY?*

Why not scratchpad?

- Programmers may not use scratchpad efficiently
  - Puts burden on the programmer
  - Error-prone if used manually
- Requires compiler support
  - To allocate appropriate blocks in scratchpad
  - Average case performance optimization
  - Worst case performance optimization

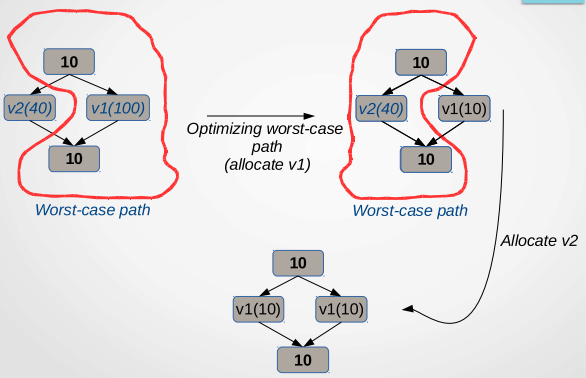
Scratchpad Allocation

- For average case optimization
  - Typically done via profiling
  - From profiling
    - frequency of memory accesses
    - the expected gain to allocate each memory block in scratchpad
  - An optimization problem
    - Minimize execution time
    - With constraints on scratchpad size

Scratchpad Allocation

- Worst-case optimization
  - Complex, requires input from WCET analysis
  - The worst-case path can be obtained from a WCET analyzer
  - Not very useful to perform on a single worst-case path
    - Worst-case path switching

Scratchpad Allocation



Scratchpad Allocation

- A greedy strategy
  - Perform WCET analysis
  - Obtain worst-case path
  - Find the variable “v” facing maximum memory latency
  - Allocate “v” into scratchpad
  - Repeat WCET analysis and repeat the process
- Advantages
  - Simple and efficient implementation possible
  - Works good in practice

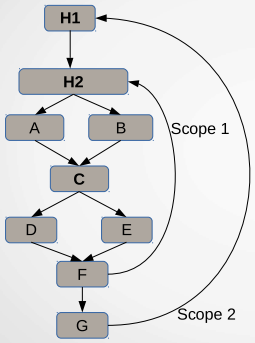
Scratchpad Allocation

- Greedy strategy
  - Not optimal, may not obtain the best WCET
  - Fits well with path-based WCET calculation
    - How about flow facts?
    - If flow facts are not considered, the optimization may concentrate on an infeasible path
      - Optimization effort is lost

Scratchpad Allocation

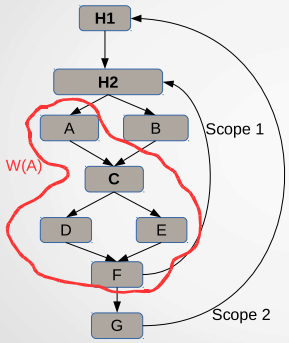
- WCET calculation based on IPET
  - State-of-the-art
  - An ILP-based optimization problem which can be integrated with IPET formulation

Scratchpad Allocation



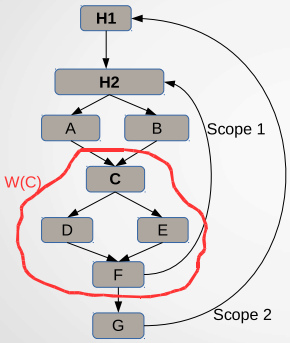
- Consider one scope (loop nest) at a time
- Consider first Scope 1
- Use the results of Scope 1 to formulate the cost for Scope 2

Scratchpad Allocation



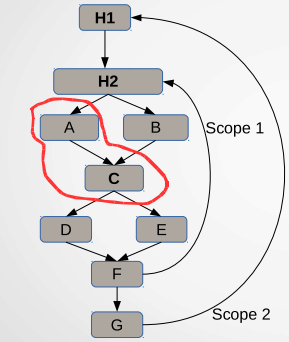
- Consider one scope (loop nest) at a time
- Consider first Scope 1
- Use the results of Scope 1 to formulate the cost for Scope 2
- *W(A) = worst-case cost for the acyclic path starting at A and ending at F (after allocation)*

Scratchpad Allocation



- Consider one scope (loop nest) at a time
- Consider first Scope 1
- Use the results of Scope 1 to formulate the cost for Scope 2
- *W(A) = worst-case cost for the acyclic path starting at A and ending at F (after allocation)*
- *W(C) = worst-case cost for the acyclic path starting at C and ending at F (after allocation)*

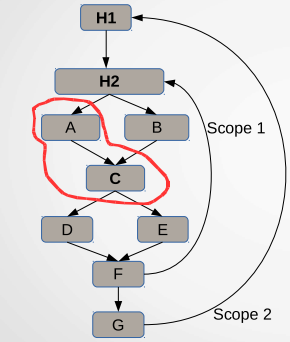
Scratchpad Allocation



- *W(A) = worst-case cost for the acyclic path starting at A and ending at F (after allocation)*
- *W(C) = worst-case cost for the acyclic path starting at C and ending at F (after allocation)*
- ILP constraints to relate W(A) and W(C)
- *X(A)=1, if A is allocated to scratchpad*
- *X(A)=0, otherwise*

Similarly, for each variable, X(C), X(D) etc.

Scratchpad Allocation



- *W(A) = worst-case cost for the acyclic path starting at A and ending at F (after allocation)*
- *W(C) = worst-case cost for the acyclic path starting at C and ending at F (after allocation)*
- ILP constraints to relate W(A) and W(C)
- *X(A)=1, if A is allocated to scratchpad*
- *X(A)=0, otherwise*

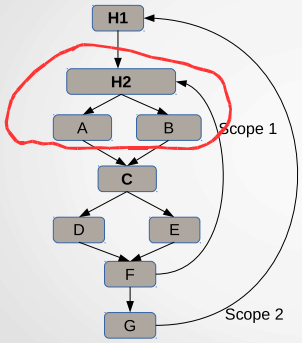
Similarly, for each variable, X(C), X(D) etc.

ILP constraint:

*W(A) >= W(C) + (T(A) – X(A) * Gain(A))*

Obtained from micro-architectural modeling

Scratchpad Allocation

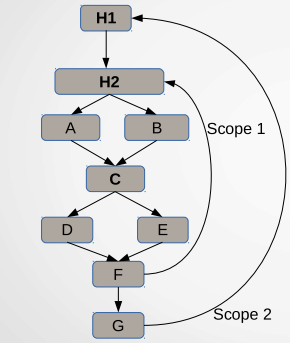


ILP constraint:

*W(A) >= W(C) + (T(A) – X(A) * Gain(A))*
*W(H2) >= W(A) + (T(H2) – X(H2) * Gain(H2))*
*W(H2) >= W(B) + (T(H2) – X(H2) * Gain(H2))*

And so on.....

Scratchpad Allocation



ILP constraint:

*W(A) >= W(C) + (T(A) – X(A) * Gain(A))*
*W(H2) >= W(A) + (T(H2) – X(H2) * Gain(H2))*
*W(H2) >= W(B) + (T(H2) – X(H2) * Gain(H2))*

And so on.....

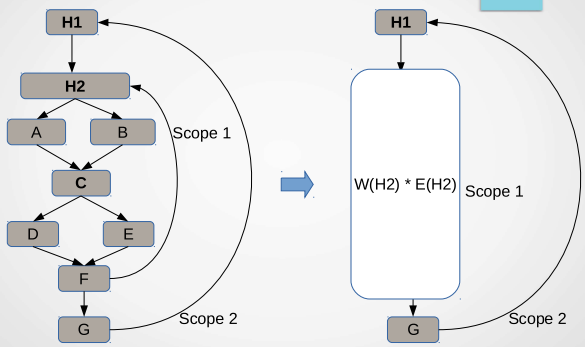
Cost of Scope 1:

*W(H2) * E(H2)*

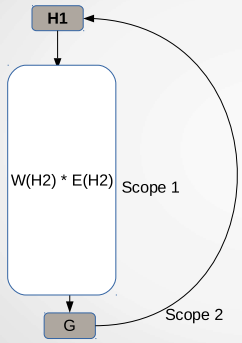
E(H2) obtained from flow facts such as

Scope1: []: XH2 <= 100

Scratchpad Allocation



Scratchpad Allocation



W(H1) = worst-case cost for the acyclic path starting at H1 and ending at G (after allocation)

- X(H1)=1, if H1 is allocated to scratchpad
- X(H1)=0, otherwise

ILP constraint:

*W(H1) >= W(Scope1) + (T(H1) – X(H1) * Gain(H1))*
*= W(H2) * E(H2) + (T(H1) – X(H1) * Gain(H1))*

Final objective:

*Minimize W(H1) * E(H1)*

Scope 2: [] : XH1 <= N

Scratchpad Allocation

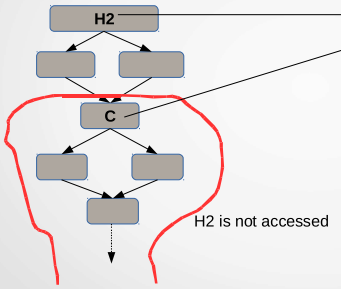
- Greedy allocation
  - Fast, but not optimal
  - Works well in practice
  - Relatively difficult to integrate flow facts
- ILP based model
  - Optimal
  - Integrates well with IPET based WCET calculation
  - Easy to integrate flow facts
  - Relatively slow, compiler optimizations need to be fast

Scratchpad Allocation

- We have discussed static allocations
  - Allocations decisions are fixed at compile time
  - Allocation decisions do not change at runtime
- Most programs have phases
  - Spatial and temporal locality
  - Automatically handled by caches
- Dynamic scratchpad allocation
  - Allocations may change at runtime

Scratchpad Allocation

- The discussed allocation does not consider overlay



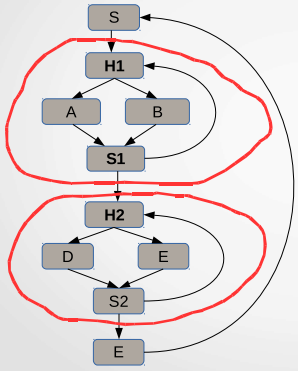
- H2 and C can share scratchpad space

- Similar to register allocation in compiler

- Variable live-range can be used to improve static scratchpad allocation

Scratchpad Allocation

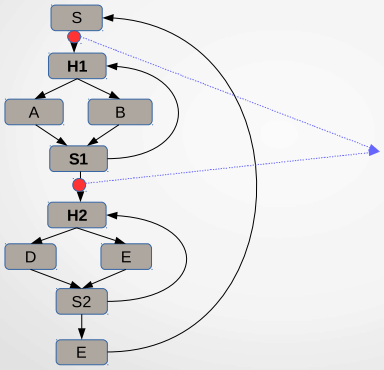
- Program phases



- Scratchpads are usually small
- Assume that the scratchpad can hold variables of only one inner loop
- Static allocation will not be very useful as one of the loop will face large memory latencies
- How about caches?

Scratchpad Allocation

- Program phases



Change allocation decision

Dynamic allocation

- Allocate H1,A,B,S1
- Flush Scratchpad
- Allocate H2,D,E,S2
- Flush Scratchpad

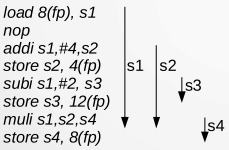
Require more analysis to decide appropriate points where allocations will be switched

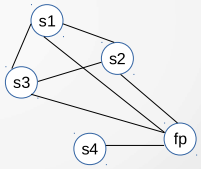
Register Allocation

- One of the most important compiler optimizations
  - Extensively studied in the past
  - Important, potentially in every execution platforms
  - Register accesses substantially reduce overall memory access penalty
- Limited registers
  - Not all variables can be allocated in registers
  - Which variables should be allocated registers?

Register Allocation

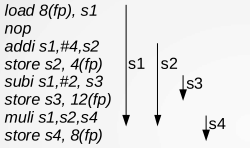
- Classic algorithms
  - Based on graph coloring
    - Chaitin et al.
    - Briggs et al.

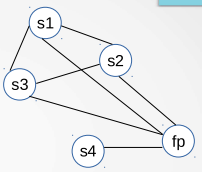




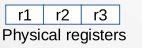
Interference Graph with virtual registers

Register Allocation





Interference Graph with virtual registers



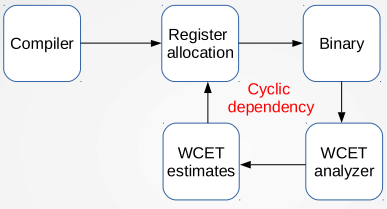
Physical registers

- The interference graph cannot be colored with 3 colors
- Some node has to be *spilled (written to memory)*
- How to spill?

Register Allocation

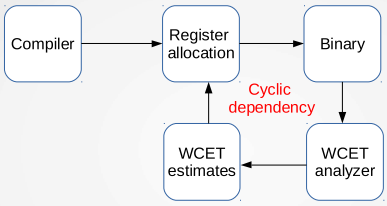
- Spill Decision lacks appropriate timing model
  - Current register allocation strategies may substantially affect the worst-case performance
- WCET analysis is performed on executable
  - Code with virtual registers is not executable
  - And, WCET estimates decide the virtual to physical register mapping
  - Cyclic dependency
- Heiko Falk, *WCET-aware Register Allocation based on Graph Coloring*, DAC 2009

Register Allocation



- Take a pessimistic approach
- Initially assume all virtual registers are kept in memory (traditional register allocators are usually optimistic)
- During register allocations, virtual registers are moved to physical registers
- Intermediate code is always executable, therefore WCET analysis can be performed

Register Allocation

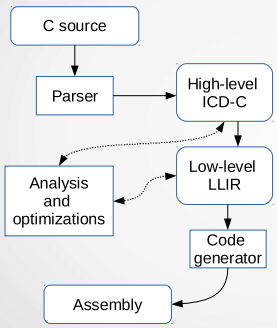


- Find the worst-case execution path (WCEP)
- Find basic block “b” leading to the highest execution of spill code along the WCEP
- “b” is chosen to be allocated in registers
- Sort all virtual registers in “b” by their number of occurrences
- Potential spill is the register with least occurrences
- *Run WCET analysis again with the modified code*

WCC Compiler

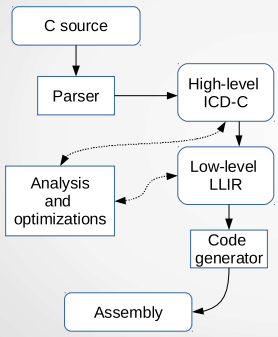
- WCET-aware C Compiler
  - Developed and maintained at TU, Dortmund
  - Extends traditional compiler passes to integrate a WCET analyzer
  - Specification of memory hierarchies to optimize memory performance
    - Scratchpad allocation
    - Cache locking
  - WCET-aware code generation
    - Preserving flow facts
    - Back annotations (timing information obtained from binary level mapped to intermediate level)
  - WCET-aware compiler optimizations

WCC Compiler

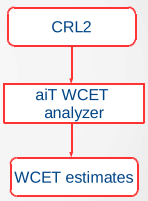


The flow of a traditional compiler

WCC Compiler



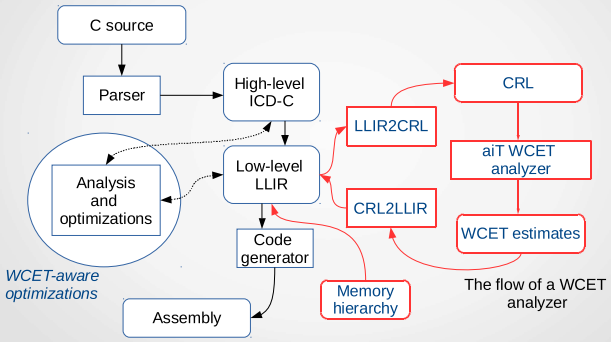
The flow of a traditional compiler



The flow of a WCET analyzer

WCC Compiler

[http://ls12-www.cs.tu-dortmund.de/research/activities/wcc]

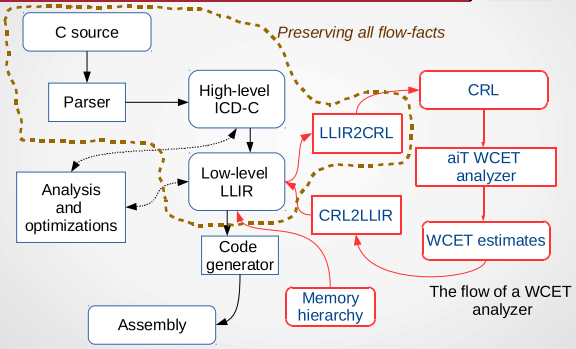


The flow of a traditional compiler

The flow of a WCET analyzer

WCC Compiler

[http://ls12-www.cs.tu-dortmund.de/research/activities/wcc]



The flow of a traditional compiler

The flow of a WCET analyzer

WCET Analyzers

- Commercial
  - aiT
    - http://www.absint.com/ait/ (WCET analyzer based on static analysis)
  - RapiTime
    - http://www.rapitasystems.com/products/rapitime (measurement-based WCET analysis tool)
- Academic research
  - SWEET (Mälardalen University)
    - http://www.mrtc.mdh.se/projects/wcet/sweet/index.html
  - Chronos (National University of Singapore)
    - http://www.comp.nus.edu.sg/~rpembed/chronos/
  - Ottawa (IRIT, University of Toulouse, France)
    - http://www.irit.fr/recherches/ARCHI/MARCH/OTAWA/doku.php?id=start

Fun Stuff

- Using the same abstract domain of LRU instruction cache analysis, can you design an abstract interpretation based framework for FIFO caches? In particular, try to formulate the update (after visiting each memory block) and join operations (at control flow merge point) for must analysis and reason about their correctness.

Lecture Materials

- Jakob Engblom and Andreas Ermedahl, *Modeling Complex Flows for Worst-Case Execution Time Analysis*. RTSS 2000 (Flow fact language)
- Henrik Theiling, Christian Ferdinand, Reinhard Wilhelm: *Fast and Precise WCET Prediction by Separated Cache and Path Analyses*. Real-Time Systems 18(2/3) (Cache modeling)
- Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, Ting Chen: *WCET Centric Data Allocation to Scratchpad Memory*. RTSS 2005 (Scratchpad allocation for WCET optimizations)
- http://ls12-www.cs.tu-dortmund.de/daes/en/forschung/wcet-aware-compilation.html *(WCET aware compilation)*
- http://ls12-www.cs.tu-dortmund.de/daes/en/forschung/wcet-aware-compilation/wcet-aware-optimizations.html *(WCET oriented compiler optimizations)*

Questions

Thank you