CS4272: HW SW Codesign

Compiler controlled Memories

Dr. Abhik Roychoudhury
School of Computing
National University of Singapore

Reading

- WCET Centric Data Allocation to Scratch-pad Memory
  - Suhendra, Mitra, Roychoudhury, Chen, IEEE Real-time Systems Symposium (RTSS) 2005.
- Discussion in pages 180-181 of textbook.

The context

General Purpose processor architectures have the following memory hierarchies

L1 cache

L2 cache

Main memory

--- Difficult to estimate cache behavior as we saw during WCET analysis

--- Can we tackle the problem at a system level as well ?

--- Develop an on-chip memory whose contents will be fixed during program execution

---- Conceptually a locked cache, called scratch-pad memory

--- Compiler controlled memories !

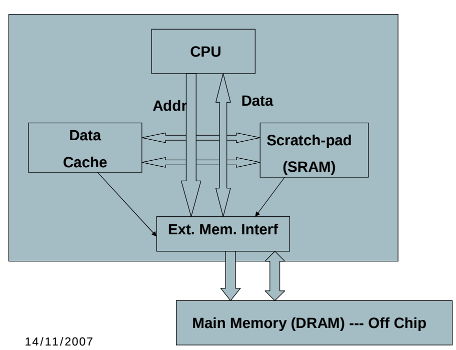
Cache

- Processor-memory performance gap
  → memory optimizations
- Cache:
  - hardware-managed
  - access determined during runtime
  - *unpredictable timing*
    → problematic for hard real-time systems
- Scratchpad
  - On chip memory, more predictable than cache

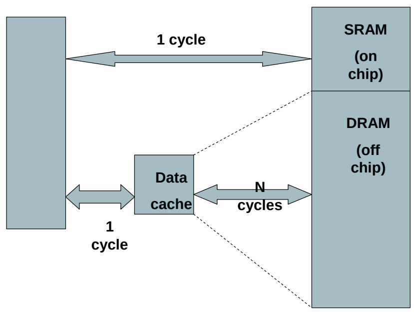
Scratch-pad

- *Scratchpad Memory*
  - software-managed
  - access: pre-defined address range
    → latencies completely *predictable*
  - lower die area, energy consumption *(Banakar et al., 2002)*
  - job shifted to compiler
- An alternative to or on top of cache

Architecture



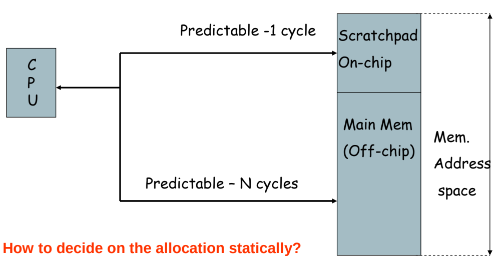
Address Space



Allocation Strategy

- Allocate scalars to scratch-pad
- For arrays
  - Find out which arrays overlap in life-time, and
  - Result in cache conflicts (if mapped to cache)
  - Map one of them to scratch-pad
- The above strategy has a caveat
  - Which array results in cache conflict requires a program trace
  - Not based on static program analysis
    - This problem is hard even when we rule out the data cache (quite common in Real-time systems)

Simplified Address Space



How to decide on the allocation statically?

ACET based Allocation

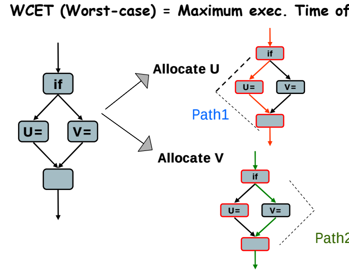
- Profile the program for selected inputs
  - Build up memory access profile
- Allocate heavily accessed variables
  - Optimal solution via 0-1 Knapsack
- If inputs are representative
  - Aims to reduce Average-Case Execution Time
- In real-time systems WCET is a key metric

Knapsack formulation

- Possible for ACET based allocation
  - items: $v_1 \ldots v_{|allvars|}$
  - capacity: scratchpad space
  - weight of each variable $u \in \{v_1 \ldots v_{|allvars|}\} = area_u$
  - gain of each variable $u \in \{v_1 \ldots v_{|allvars|}\}$
    - Constant for a given path
    - Not constant for WCET based allocation.

ACET vs WCET based

WCET (Worst-case) = Maximum exec. Time of a pgm for all possible inputs



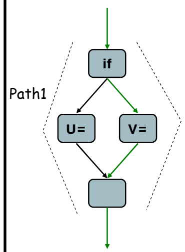
In this simple example assume

- *U and V have same size*
- *Only one of them can be allocated.*
- *ACET savings depends on the **execution counts** of Path1 and Path2*
- *WCET savings depend on which path is **longer**.*

Difficulty in WCET-based allocation

Allocate V

	Path1	Path2
Before	90	100
After	90	80

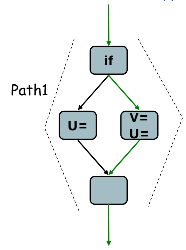


Contribution of V to WCET path = 20
Reduction in WCET by allocating V = 10

Path1 is now the WCET path which has different var. access frequencies from Path2

Sub-optimal (local) allocation

V,U both appear in WCET path. Suppose #V > # U



Path2

WCET path based allocation = {V}

	Path1	Path2
Before	90	100
After	90	80

Optimal allocation = {U}

	Path1	Path2
Before	90	100
After	75	85

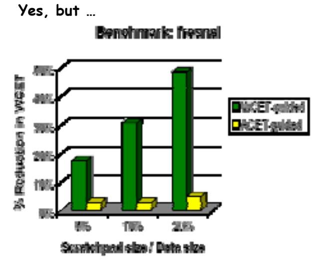
Summary of bad news

- Optimal WCET based allocation
  - Must not be profile-guided (WCET path).
  - Cannot take WCET contribution of variables as constant.
    - Rules out Knapsack like solutions.
  - Should ideally be a global optimization procedure aware of infeasible paths.
    - Paths in CFG not executed on any program input.

Why not take ACET-based?

ACET based allocation should also reduce WCET, after all.

Yes, but …



→46% additional WCET reduction

→Varies in other benchmarks, but ~20%

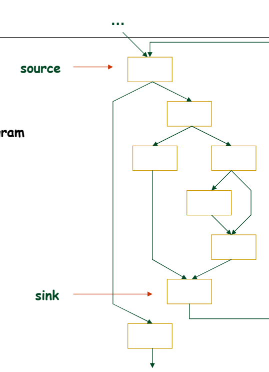
Allocation methods

- Integer Linear Programming (ILP)
  - Cannot cater for infeasible paths.
- Branch and Bound (BnB)
  - High Complexity.
- Greedy Heuristic
  - Diff. from greedy allocation using WCET path.
- Experiments
  - WCET reduction, Running time.

Control Flow Graph (CFG) of a program

Control Flow Graph (CFG) of a loop

Directed Acyclic Graph (DAG)



ILP Formulation

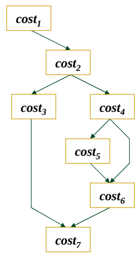
allvars : set of all variables in the program

For each *v* in *allvars*:

$area_v$: memory area occupied by variable *v*

$gain_v$: gain in single var. access by allocating *v*

S_v : 1 if *v* is allocated, 0 otherwise



$$\sum_{v \in allvars} S_v * area_v \leq scratchpad\_size$$

$$\forall v \in allvars \quad S_v \geq 0, \; S_v \leq 1$$

Minimize W1 ∗ loopbound

$$W_1 = W_2 + cost_1 - \sum_{v \in vars(1)} S_v * gain_v * n_{v,1}$$

$$W_2 \geq W_3 + cost_2 - \sum_{v \in vars(2)} S_v * gain_v * n_{v,2}$$

$$W_2 \geq W_4 + cost_2 - \sum_{v \in vars(2)} S_v * gain_v * n_{v,2}$$

$$W_3 = W_7 + cost_3 - \sum_{v \in vars(3)} S_v * gain_v * n_{v,3}$$

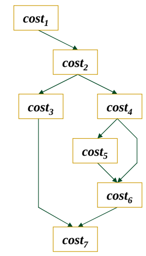
$$W_4 \geq W_5 + cost_4 - \sum_{v \in vars(4)} S_v * gain_v * n_{v,4}$$

$$W_4 \geq W_6 + cost_4 - \sum_{v \in vars(4)} S_v * gain_v * n_{v,4}$$

$$W_5 = W_6 + cost_5 - \sum_{v \in vars(5)} S_v * gain_v * n_{v,5}$$

$$W_6 = W_7 + cost_6 - \sum_{v \in vars(6)} S_v * gain_v * n_{v,6}$$

$$W_7 = cost_7 - \sum_{v \in vars(7)} S_v * gain_v * n_{v,7}$$

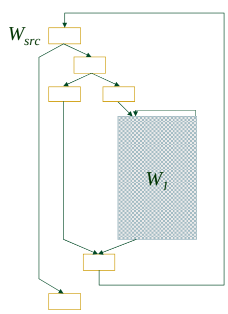


$$\sum_{v \in allvars} S_v * area_v \leq scratchpad\_size$$

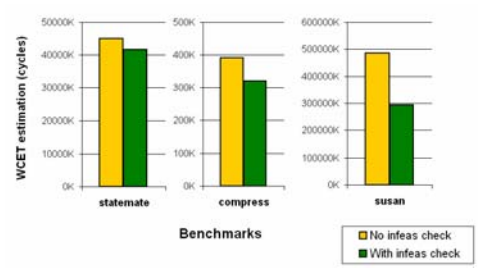
$$\forall v \in allvars \quad S_v \geq 0, \; S_v \leq 1$$

ILP Formulation

Extension to whole program



WCET est. with/without infeasibility checking



Allocation methods

- Integer Linear Programming (ILP)
  - Cannot cater for infeasible paths.
- Branch and Bound (BnB)
  - Rule out easy solutions --- ACET based alloc
  - High Complexity of BnB
  - Greedy Heuristic
    - Diff. from greedy allocation using WCET path.
  - Infeasible path detection
- Experiments
  - WCET reduction, Running time.

Knapsack?

- Possible for ACET based allocation
  - items: $v_1 \ldots v_{|allvars|}$
  - capacity: scratchpad space
  - weight of each variable u ∈ $\{v_1 \ldots v_{|allvars|}\}$= $area_u$
  - gain of each variable u ∈ $\{v_1 \ldots v_{|allvars|}\}$
    - Constant for a given path
    - Not constant for WCET based allocation.

Re-cap on Knapsack problem

- Given n objects and a knapsack
  - Capacity of knapsack W
  - Object i has weight w_i and value v_i
  - Fill up the knapsack so as to maximize value
- Perfect fit for our allocation problem if the gain by allocating a variable to scratchpad memory is a constant
  - Holds for ACET based allocation
  - Not true for WCET based allocation
- Knapsack problem can be easily solved by dynamic programming.

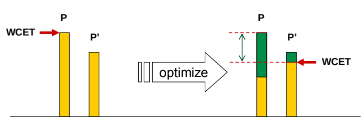
Dynamic Programming

- Array V[0..n,0..W]
- V[i,j] = maximum value if we are restricted to objects 1..i, and the weight limit is j
- Define
  - $V[i,j] = \max(V[i-1,j], V[i-1,j-w_i] + v_i)$ for i > 0
  - $V[0,j] = 0$
- Final Answer V[n,W]

- This solution is useful only for ACET based allocation.

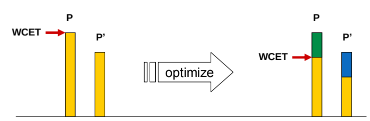
For WCET-based

- Knapsack?
- gain in WCET reduction due to a variable v
  - *not a constant, depends on*
    - current WCET path
    - Diff in exec. Times of WCET path and other paths
    - # Occurrence of v in WCET path and other paths

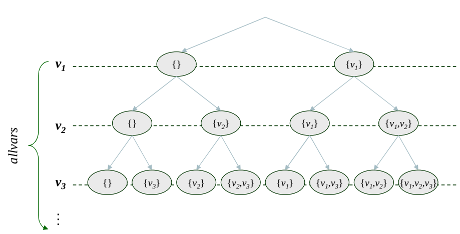


For WCET-based

- Knapsack?
- gain in WCET reduction due to a variable:
  - *not a constant*
  - *not cumulative*
    - $gain_{\{v,v'\}} \leq gain_v + gain_{v'}$

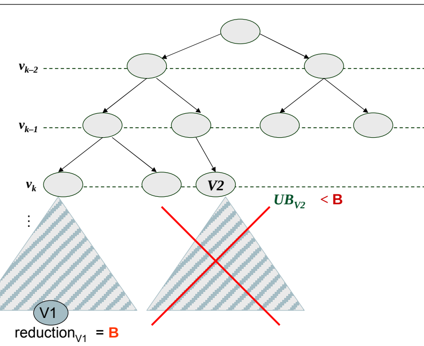


Search tree



But, cannot search it exhaustively.

Branch-and-bound



How to estimate the upper bound?

- For partial allocation V at level k,
  - $V \subseteq \{v_1,\ldots, v_k\}$
  - UB_V = (WCET reduction due to V) + max_k
    - max_k computed as knapsack problem:
      - items: $v_{k+1}\ldots v_{|allvars|}$
      - capacity: scratchpad space after allocating V
      - weight of each variable u $\in \{v_{k+1}\ldots v_{|allvars|}\}$ = $area_v$
      - gain of each variable u $\in \{v_{k+1}\ldots v_{|allvars|}\}$ = *maximum contribution of v towards exec. time of any path*
        - Can be obtained by bottom-up pass of syntax tree.

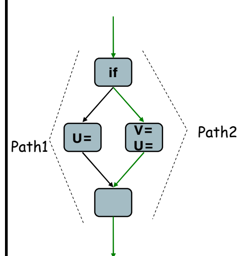
Efficiency Issues

- Search exponential in #vars. even after pruning.
  - Efficient heuristics with near-optimal allocation.
  - Do not only consider the WCET path.
- WCET est. for a given scratchpad allocation
  - Invoked several times by Branch-and-bound
  - Needs to consider Infeasible path information
    - Infeasible pairs of assgn/branch in a loop iteration
  - Must run very fast [we do not discuss it here]
    - Find heaviest iteration in a loop considering this info.
    - Avoid backtracking without path enumeration

Greedy Heuristic

- Initially:
  - No variable allocated
- Iteratively:
  - Identify current WCET path P (considering current allocation)
  - Allocate variable with max. gain in P that can fit into remaining space in scratchpad
- Stop when:
  - scratchpad is filled, or
  - no more variable can be allocated from current WCET path

Notes on sub-optimality



Becomes close to WCET path based allocation if

→Scratchpad is very small

→One of U, V can be allocated

→Less overlap among vars. appearing in diff. paths

→V does not appear in Path1

Unlikely, so backtracking based heuristics did not work any better!

Allocation methods

- Integer Linear Programming (ILP)
  - Cannot cater for infeasible paths.
- Optimal with infeasible path info.
  - Branch-and-Bound
    - High Complexity
  - Greedy Heuristic
    - Diff. from greedy allocation using WCET path.
  - Infeasible path detection

Other Works

- Scratchpad allocation for data memory
  - WCET-based
  - Easily extensible to
    - Variables with disjoint lifetimes
    - Code memory
- Extensions to Multi-processor SoCs
  - Allocate to reduce bus contention among processors.
- ACET-based allocation can be modified to get scratch-pad allocation to reduce energy for example --- simple change
- Any such allocation must be combined with WCET analysis & infeasible path detection