Lecture 24: Optimization II

Seyed-Hosein Attarzadeh-Niaki

Based on slides by Peter Marwedel

Review

- Task level concurrency management
- High-level optimizations
  - Floating-point to fixed-point conversion
  - Simple loop transformations
    - Loop permutation
    - Loop fusion, loop fission
    - Loop unrolling
  - Loop tiling/blocking
  - Loop splitting
  - Array folding

Outline

- Compilers for embedded systems
  - Energy-aware compilation
  - Memory-architecture aware compilation
  - Reconciling compilers and timing analysis

Compilers for embedded systems: Why are compilers an issue?

- Many reports about low efficiency of standard compilers
  - Special features of embedded processors have to be exploited.
  - High levels of optimization more important than compilation speed.
  - Compilers can help to reduce the energy consumption.
  - Compilers could help to meet real-time constraints.
- Less legacy problems than for PCs.
  - There is a large variety of instruction sets.
  - **Design space exploration** for optimized processors makes sense

Energy-Aware Compilation

- Based on energy/power models
  - Measured values on real system associated with instructions
  - Based on datasheets
  - Pipeline analysis
  - Energy consumption of caches
  - Architecture-level estimation

Energy-aware compilation (1): Optimization for low-energy the same as for high performance?

No !

- High-performance if available memory bandwidth fully used; low-energy consumption if memories are at stand-by mode
- Reduced energy if more values are kept in registers

```
LDR r3, [r2, #0]
ADD r3,r0,r3
MOV r0,#28
LDR r0, [r2, r0]
ADD r0,r3,r0
ADD r2,r2,#4
ADD r1,r1,#1
CMP r1,#100
BLT LL3
```

2096 cycles
19.92 µJ

```
int a[1000];
c = a;
for (i = 1; i < 100; i++) {
 b += *c;
 b += *(c+7);
 c += 1;
}
```

2231 cycles
16.47 µJ

```
ADD r3,r0,r2
MOV r0,#28
MOV r2,r12
MOV r12,r11
MOV r11,r10
MOV r10,r9
MOV r9,r8
MOV r8,r1
LDR r1, [r4, r0]
ADD r0,r3,r1
ADD r4,r4,#4
ADD r5,r5,#1
CMP r5,#100
BLT LL3
```

Energy-aware compilation (2)

- Operator strength reduction: e.g. replace * by + and <<
- Minimize the bitwidth of loads and stores
- Standard compiler optimizations with energy as a cost function

E.g.: Register pipelining:

```
for i:= 0 to 10 do
    C:= 2 * a[i] + a[i-1];
```

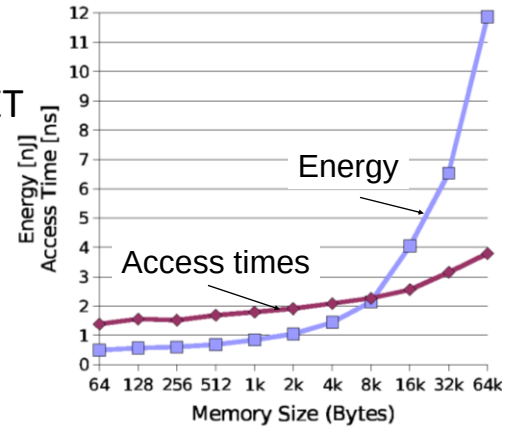
```
R2:=a[0];
for i:= 1 to 10 do
 begin
    R1:= a[i];
    C:= 2 * R1 + R2;
    R2 := R1;
 end;
```


Energy-aware compilation (3)

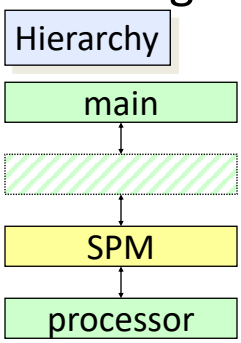
- Energy-aware *scheduling*: the order of the instructions can be changes as long as the meaning does not change.
  Goal: reduction of the number of signal transitions
  - Popular (can be done as a post-pass optimization with no change to the compiler).
- Energy-aware *instruction selection*: among valid instruction sequences, select those minimizing energy consumption
- Exploitation of the *memory hierarchy*: huge difference between the energy consumption of small and large memories
  - Best energy saving method -> use scratchpad memories

Three key problems for future memory systems

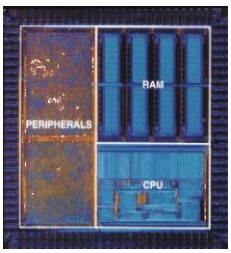
- (Average) Speed
- Energy/Power
- Predictability/WCET



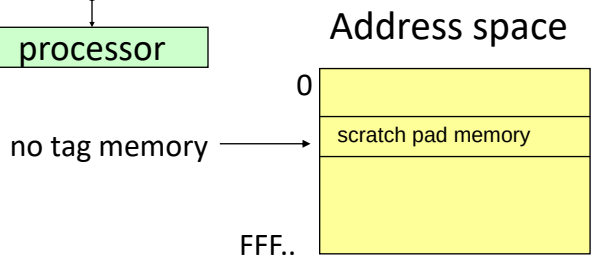
Hierarchical memories using scratch pad memories (SPM)



Example

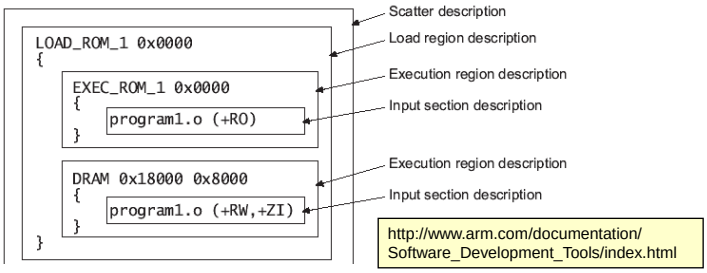


ARM7TDMI cores, well-known for low power consumption



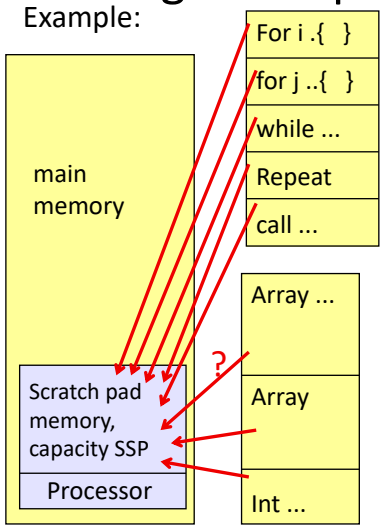
Very limited support in ARMcc-based tool flows

1. Use pragma in C-source to allocate to specific section:
   For example:
```
#pragma arm section rwdata="foo", rodata="bar"
int x2 = 5; // in foo (data part of region)
int const z2[3] = {1,2,3}; // in bar
```
2. Input scatter loading file to linker for allocating section to specific address range



Migration of data & instructions, global optimization model

Example:



Which memory object (array, loop, etc.) to be stored in SPM?

Non-overlaying ("Static") allocation:

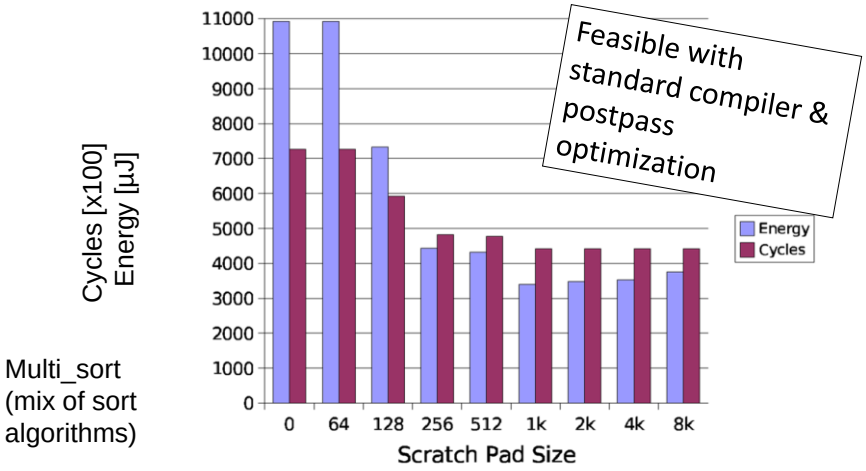
Gain g_k and size s_k for each object k.
Maximize gain $G = \Sigma g_k$, respecting size of SPM $SSP \geq \Sigma s_k$.

Solution: knapsack algorithm.

Overlaying ("dynamic") allocation:

Moving objects back and forth

Reduction in energy and average run-time



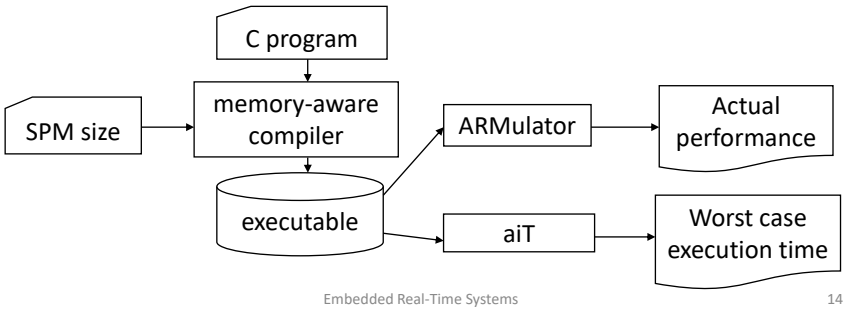
Multi_sort (mix of sort algorithms)

Numbers will change with technology, algorithms remain unchanged.

Scratch-pad memory based predictability

Pre run-time scheduling *is often the only practical means of providing predictability in a complex system.* [Xu, Parnas]

- Time-triggered, statically scheduled operating systems
- Let's do the same for the memory system
  - Are SPMs really more timing predictable?
  - Analysis using the aiT timing analyzer



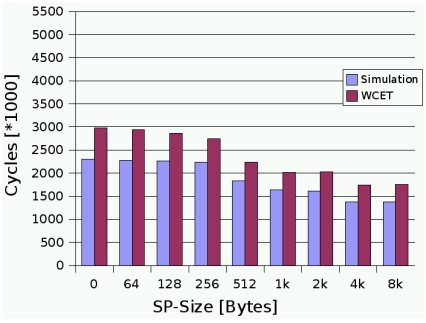
Experiment: Architectures considered

ARM7TDMI with 3 different memory architectures

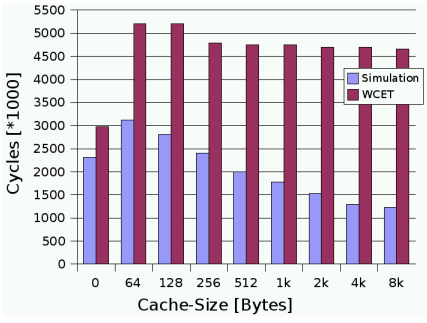
1. **Main memory**
   LDR-cycles: (CPU,IF,DF)=(3,2,2)
   STR-cycles: (2,2,2)
   * = (1,2,0)
2. **Main memory + unified cache**
   LDR-cycles: (CPU,IF,DF)=(3,12,6)
   STR-cycles: (2,12,3)
   * = (1,12,0)
3. **Main memory + scratch pad**
   LDR-cycles: (CPU,IF,DF)=(3,0,2)
   STR-cycles: (2,0,0)
   * = (1,0,0)

Results for G.721

Using Scratchpad



Using Unified Cache



References:

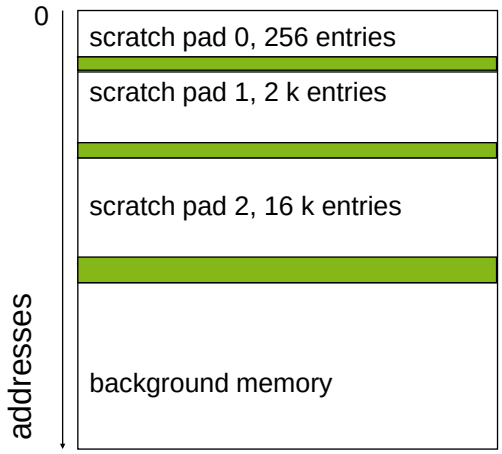
- Wehmeyer, Marwedel: Influence of Onchip Scratchpad Memories on WCET: 4th Intl Workshop on worst-case execution time (WCET) analysis, Catania, Sicily, Italy, June 29, 2004
- Second paper on SP/Cache and WCET at DATE, March 2005

Multiple scratch pads

Small is beautiful:

One small SPM is beautiful (☺).

May be, several smaller SPMs are even more beautiful?

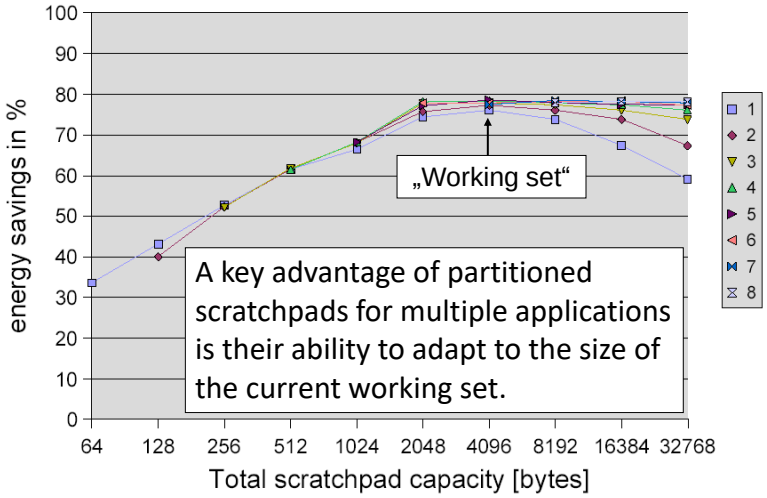


Considered partitions

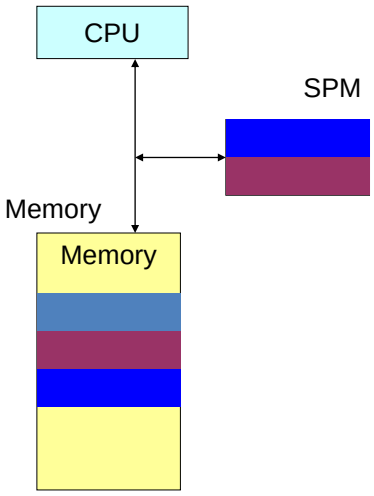
Example of considered memory partitions for a total capacity of 4096 bytes

# of partitions	number of partitions of size:						
	4k	2k	1k	512	256	128	64
7	0	1	1	1	1	1	2
6	0	1	1	1	1	2	0
5	0	1	1	1	2	0	0
4	0	1	1	2	0	0	0
3	0	1	2	0	0	0	0
2	0	2	0	0	0	0	0
1	1	0	0	0	0	0	0

Results for parts of GSM coder/ decoder



Overlaying/dynamic replacement within scratch pad

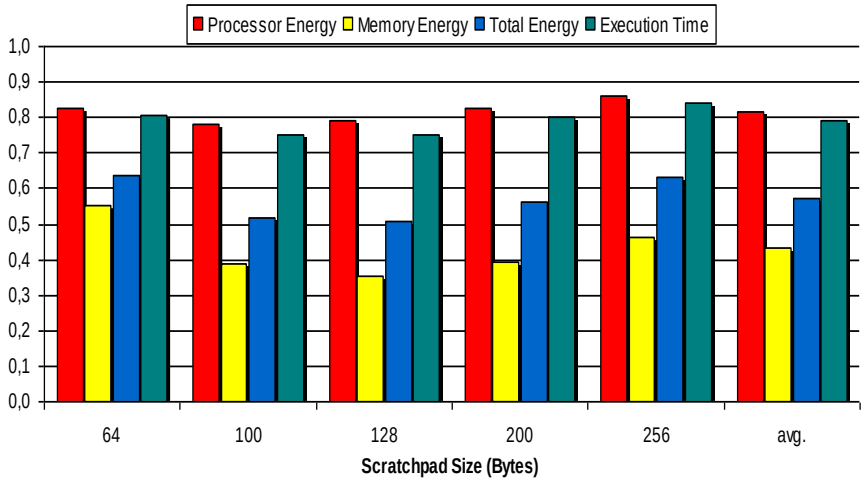


- Effectively results in a kind of compiler-controlled segmentation/paging for SPM
- Address assignment within SPM required (paging or segmentation-like)

Reference: Verma, Marwedel: Dynamic Overlay of Scratchpad Memory for Energy Minimization, ISSS 2004

Overlaying/dynamic replacement within SPM

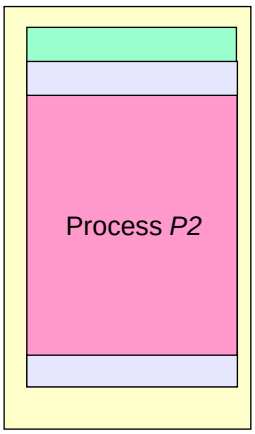
Edge detection relative to non-overlaying/static allocation



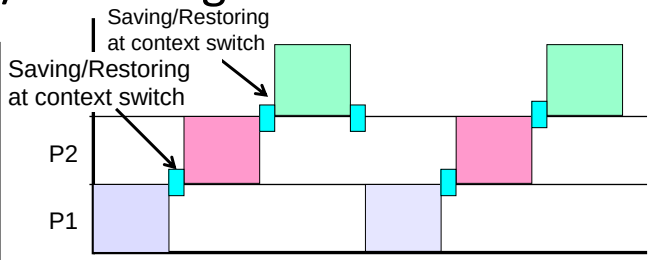
Approaches for Overlaying Allocation

- Tiling of large arrays
  - To use parts of large data
- Multiple hierarchy levels
  - Automatically find appropriate allocation to multiple hierarchy levels
- Region-based memory object migration
- Selection of the memory objects to be copied based on a global ILP model

Multiple threads/processes: 1) Saving/Restoring Context Switch

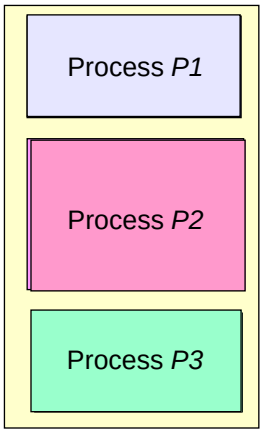


Scratchpad

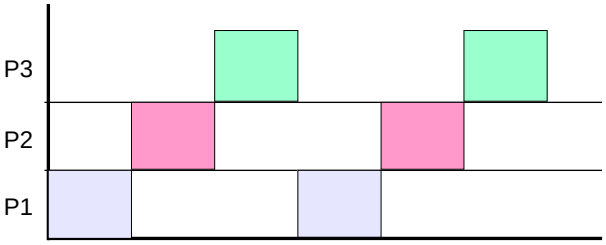


- **Saving Context Switch (Saving)**
  - Utilizes SPM as a common region shared all processes
  - Contents of processes are copied on/off the SPM at context switch
  - Good for small scratchpads

Multiple threads/processes: 2) Non-Saving Context Switch

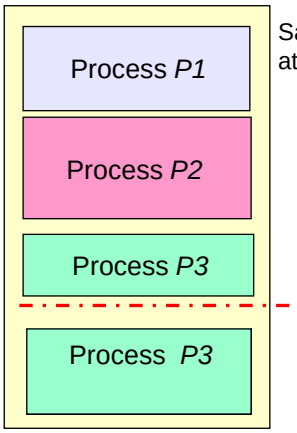


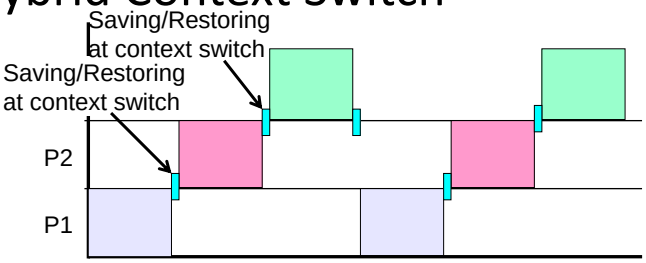
Scratchpad



- **Non-Saving Context Switch**
  - Partitions SPM into disjoint regions
  - Each process is assigned a SPM region
  - Copies contents during initialization
  - Good for large scratchpads

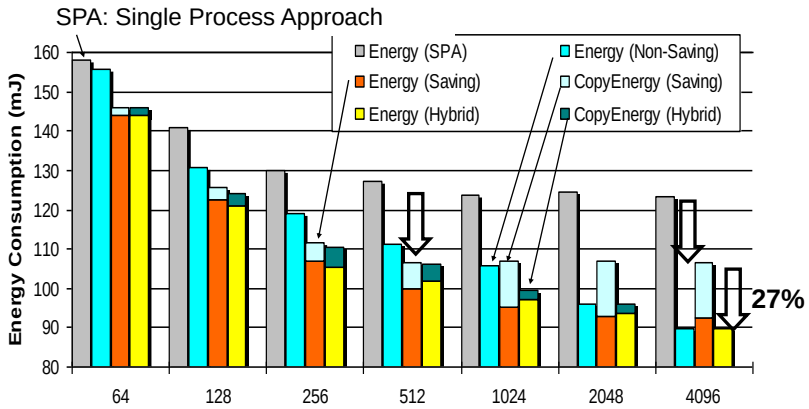
Multiple threads/processes: 3) Hybrid Context Switch





- **Hybrid Context Switch (Hybrid)**
  - Disjoint + Shared SPM regions
  - Good for all scratchpads
  - Analysis is similar to Non-Saving Approach
  - Runtime: $O(nM^3)$

Multi-process Scratchpad Allocation: Results



- For small SPMs (64B-512B) Saving is better
- For large SPMs (1kB- 4kB) Non-Saving is better
- Hybrid is the best for all SPM sizes.
- Energy reduction @ 4kB SPM is 27% for Hybrid approach

edge detection, adpcm, g721, mpeg

Current Trial-and-Error Based Development

1. Specification of CPS/ES system
2. Generation of Code (ANSI-C or similar)
3. Compilation of Code
4. Execution and/or simulation of code, using a (e.g. random) set of input data
5. Measurement-based computation of "estimated worst case execution time" ($WCET_{meas}$)
6. Adding safety margin m on top of $WCET_{meas}$: $WCET_{hypo} := (1 + m)* WCET_{meas}$
7. If "$WCET_{hypo}$" > deadline: change some detail, go back to 1 or 2.

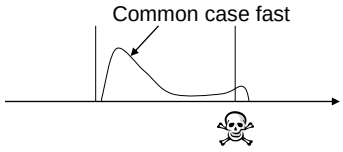
Problems with this Approach

Dependability
- Computed "$WCET_{hypo}$" not a safe approximation
  - ➢ Time constraint may be violated

Design time
- How to find necessary changes?
- How many iterations until successful?

"Make the common case fast" a wrong approach for RT-systems
- Computer architecture and compiler techniques focus on average speed
- Circuit designers know it's wrong
- Compiler designers (typically) don't

"Optimizing" compilers unaware of cost functions other than code size



Challenges for $WCET_{EST}$-Minimization

Worst-Case Execution Path (WCEP)

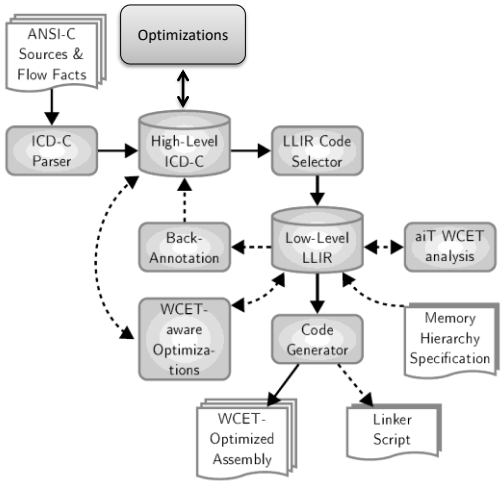
- $WCET_{EST}$ of a program = Length of longest execution path (WCEP) in that program
- $WCET_{EST}$-Minimization: Reduction of the longest path
- Other optimizations do not result in a reduction of $WCET_{EST}$

☞ Optimizations need to know the WCEP

Integration of WCET estimation and compilation

- Computing $WCET_{EST}$ after code generation is too late.
- Why not consider $WCET_{EST}$ as an objective function already in the compiler?

☞ Integration of aiT and compiler



WCET-oriented optimizations

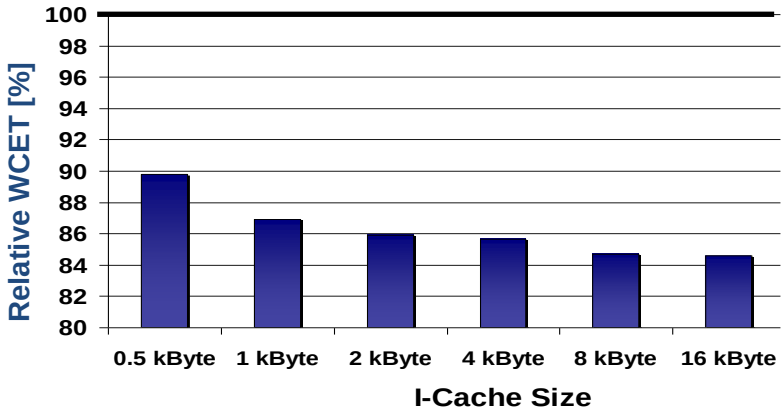
- Extended loop analysis (CGO 09)
- Instruction cache locking (CODES/ISSS 07, CGO 12)
- Cache partitioning (WCET-Workshop 09)
- Procedure cloning (WCET-WS 07, CODES 07, SCOPES 08)
- Procedure/code positioning (ECRTS 08, CASES 11 (2x))
- Function inlining (SMART 09, SMART 10)
- Loop unswitching/invariant paths (SCOPES 09)
- Loop unrolling (ECRTS 09)
- Register allocation (DAC 09, ECRTS 11))
- Scratchpad optimization (DAC 09)
- Extension towards multi-objective optimization (RTSS 08)
- Superblock-based optimizations (ICESS 10)
- Surveys (Springer 10, Springer 12)

Loop Unrolling as an Example

- Unrolling replaces the original loop with several instances of the loop body
- Positive Effects
  - Reduced overhead for loop control
  - Enables instruction level parallelism (ILP)
  - Offers potential for following optimizations
- Unroll *early* in optimization chain
- Negative Effects
  - Aggressive unrolling leads to I-cache overflows
  - Additional spill code instructions
  - Control code may cancel positive effects

Consequences of transformation hardly known

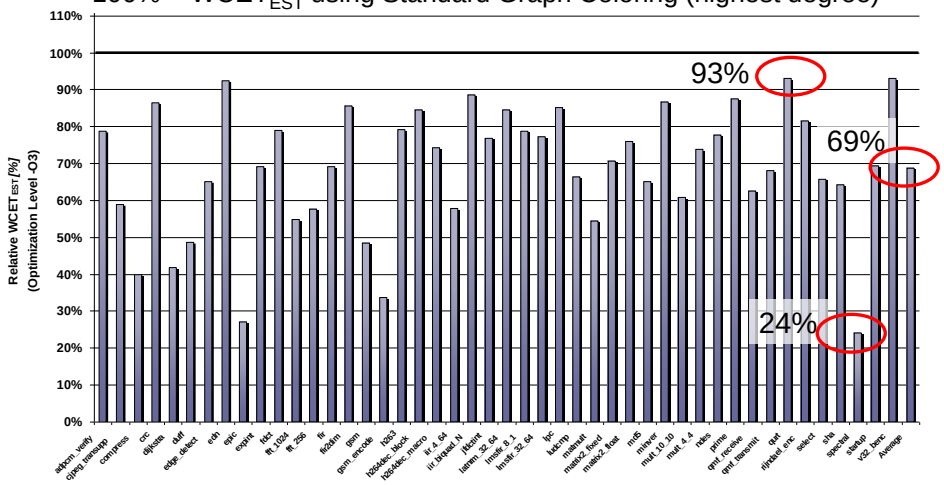
Results for unrolling: $WCET_{EST}$



100%: Avg. $WCET_{EST}$ for all benchmarks with –O3 & no unrolling
$WCET_{EST}$ reduction between 10.2% and 15.4%
$WCET_{EST}$-driven unrolling outperforms standard unrolling by 13.7%

Results: Register Allocation

100% = $WCET_{EST}$ using Standard Graph Coloring (highest degree)

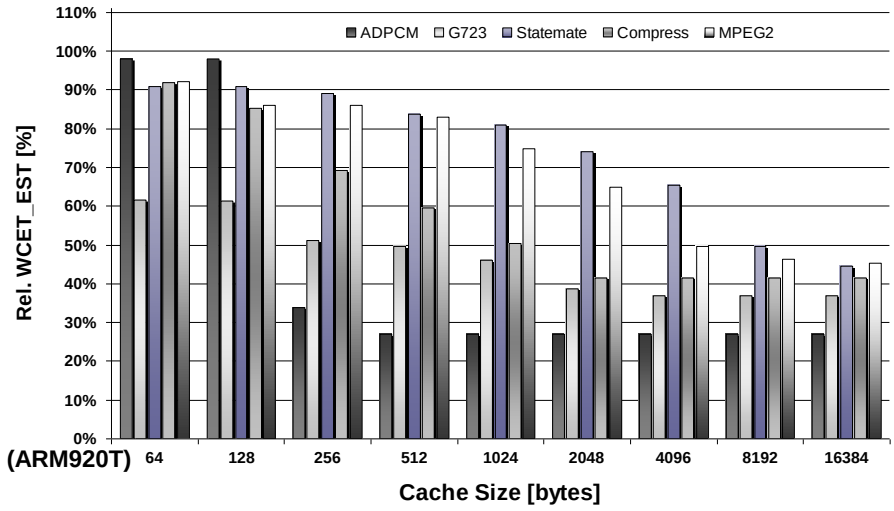


[H. Falk]

Improving predictability for caches

- Loop caches
- Mapping code to less used part(s) of the index space
- Cache locking/freezing
- Changing the memory allocation for code or data
- Mapping pieces of software to specific ways.
  Methods:
  - Way prediction in hardware
  - Generating appropriate way in software
  - Allocation of certain parts of the address space to a specific way
  - Including way-identifiers in virtual to real-address translation

Relative $WCET_{EST}$ with I-*Cache Locking*
5 Benchmarks/ARM920T/Postpass-Opt



[S. Plazar et al.]