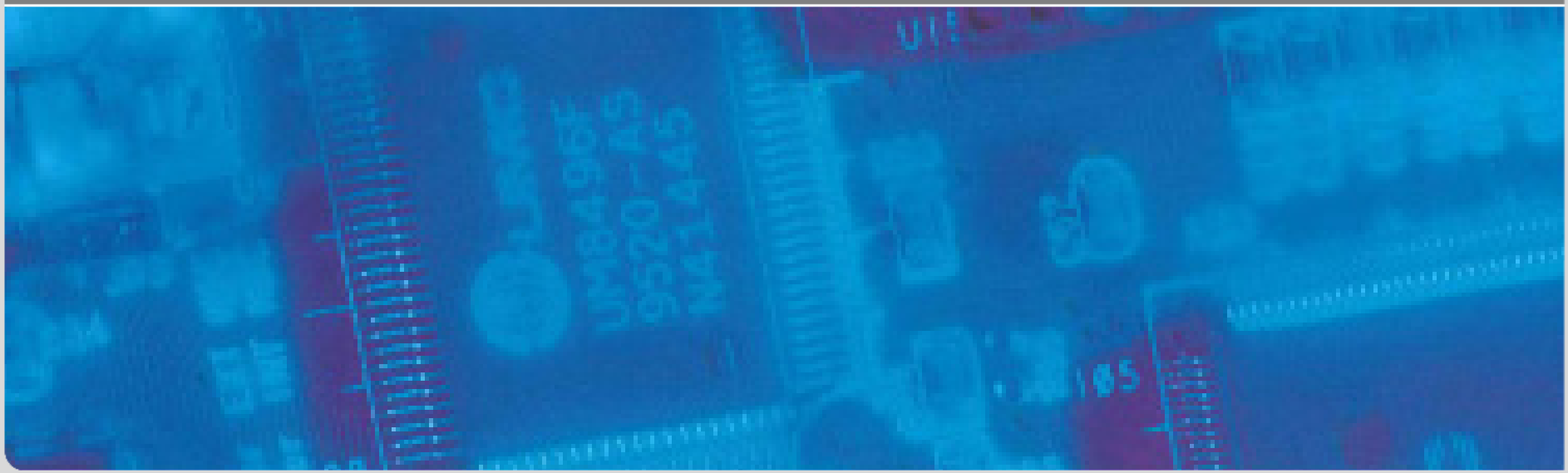
Karlsruhe Institute of Technology

Low Power Design

Volker Wenzel on behalf of Prof. Dr. Jörg Henkel
Summer Term 2016

CES – Chair for Embedded Systems



Lecture Slides

Overview Low Power Design Lecture

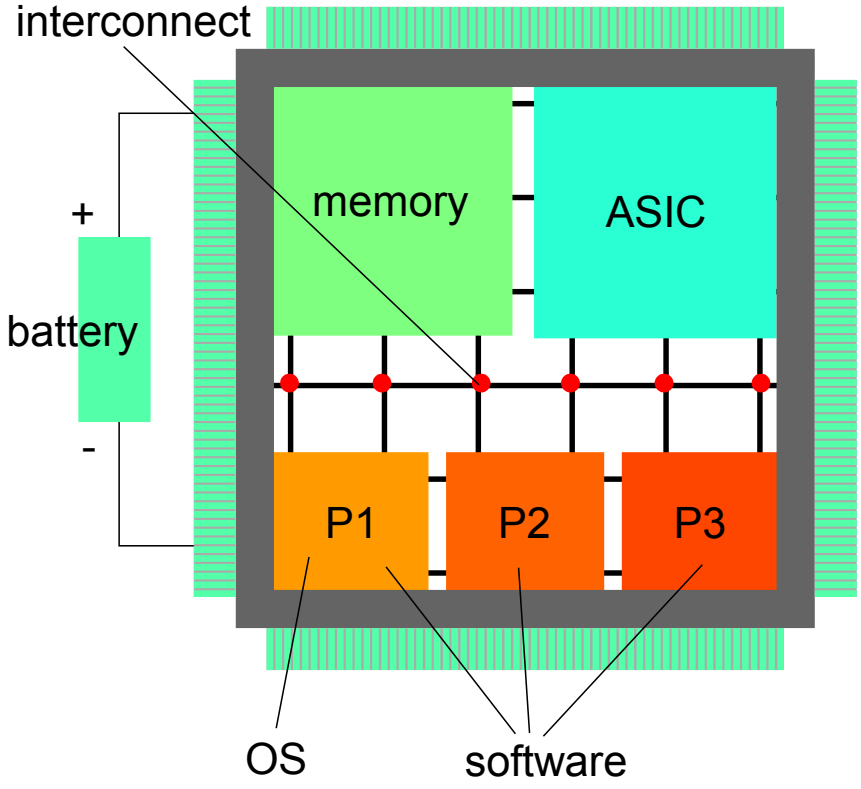
- Introduction and Energy/Power Sources (1)
- Energy/Power Sources(2): Solar Energy Harvesting
- Battery Modeling – Part 1
- Battery Modeling – Part 2
- Hardware power optimization and estimation – Part 1
- Hardware power optimization and estimation – Part 2
- Hardware power optimization and estimation – Part 3
- Low Power Software and Compiler
- Thermal Management – Part 1
- Thermal Management – Part 2
- Aging Mechanisms in integrated circuits
- Lab Meeting

Overview for today

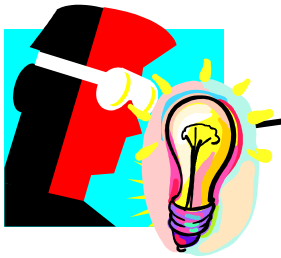
- Software power analysis/measurement
- Software power estimation models
- Optimizing software for low power through compilation phase
- Instruction scheduling
- Compiler-driven DVS
- Powertop Demo

Overview

- Levels of abstraction
  - system
  - RTL
  - gate
  - transistor
- Challenges
  - optimize (ie. minimize for low power)
  - design /co-design (synthesize, compile, …)
  - estimate and simulate



Low Power Software: Overview



Power efficient source code

Source

Target-independent optimizations

Code generation

Assembler/Linker

System Software: RTOS, device drivers, ...

Libraries

Target Memory Image

Low-power compilers:
-transformations
-code generations
-memory layout
-code compression

Target architecture model

Instruction-level power model

ISS, debugger

Co-simulator

HW

Low-power OS, middleware
-power management
-voltage/clock scheduling

(src.: A. Raghunathan, NEC)

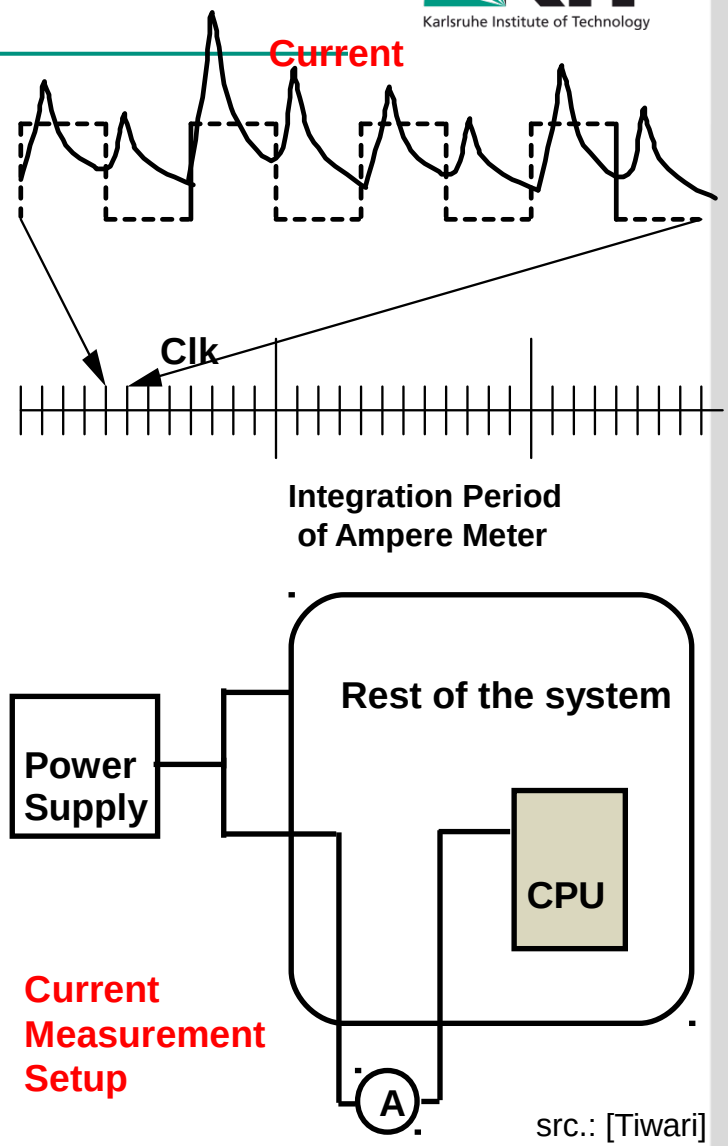
Instruction-level SW power modeling

- Energy consumed = f(Instruction sequence)
  - The model considers
    a) per-instruction costs
    b) circuit state overhead costs
    c) penalties for pipeline stalls and cache misses
- Program energy cost = $\sum_I (Base_I \cdot N_I) + \sum_{I,J} (Ovhd_{I,J} \cdot N_{I,J}) + N_{CM} \cdot Penalty_{CM} + N_{Stall} \cdot Penalty_{Stall}$
  - N_I number of times instruction I is executed
  - $Base_I$ Base energy cost of I (ignores stalls, cache misses)
  - $Ovhd_{I,J}$ Circuit state overhead when I,J are adjacent
  - $Penalty_{CM}$ Cache Miss Penalty
  - $Penalty_{Stall}$ Pipelines Stall Penalty
- Circuit state overhead: depends on processor architecture
  - constant value for 486DX2, Fujitsu SPARClite
  - table for Fujitsu DSP due to greater variation

src.: [Tiwari]
A. Raghunathan, NEC

Building instruction-level power models

- Characterize current drawn by CPU for given instruction sequence
- Simulation based methods
  - Simulate program execution on HW models of the CPU
- Physical measurement
  - Digital Ampere meter
  - Run programs in loops
  - Get stable visual reading
- Processors investigated: Intel 486DX, Fujitsu SPARClite, Fujitsu DSP



src.: [Tiwari]
A. Raghunathan, NEC

Estimation Example

	Program	Base Cost (mA)	Cycles
	main:		
B1	mov bp, sp	285.0	1
	sub sp, 4	309.0	1
	mov dx, 0	309.8	1
	mov word ptr -4[bp], 0	404.8	2
	L2:		
	mov si, word ptr -4[bp]	433.4	1
	add si, si	309.0	1
	add si, si	309.0	1
	mov bx, dx	285.0	1
	mov cx, word ptr _a[si]	433.4	1
	add bx, cx	309.0	1
	mov si, word ptr _b[si]	433.4	1
B2	add bx, si	309.0	1
	mov dx, bx	285.0	1
	mov di, word ptr -4[bp]	433.4	1
	inc di	297.0	1
	mov word ptr -4[bp], di	560.1	1
	cmp di, 4	313.1	1
	jl L2	405.7(356.9)	3(1)
	L1:		
	mov word ptr _sum, dx	521.7	1
B3	mov sp, bp	285.0	1
	jmp main	403.8	3

Block	Instances
B1	1
B2	4
B3	1
jl L2 (taken)	3
(not taken)	1

Base Cost $_{PROGRAM}$ =

Σ Base Cost$_{BLOCKi}$ * Instances$_{BLOCKi}$

Estimated base current =

Base Cost $_{PROGRAM}$ / 72 = 369.0mA

Final estimated current = 369.0 + 15.0

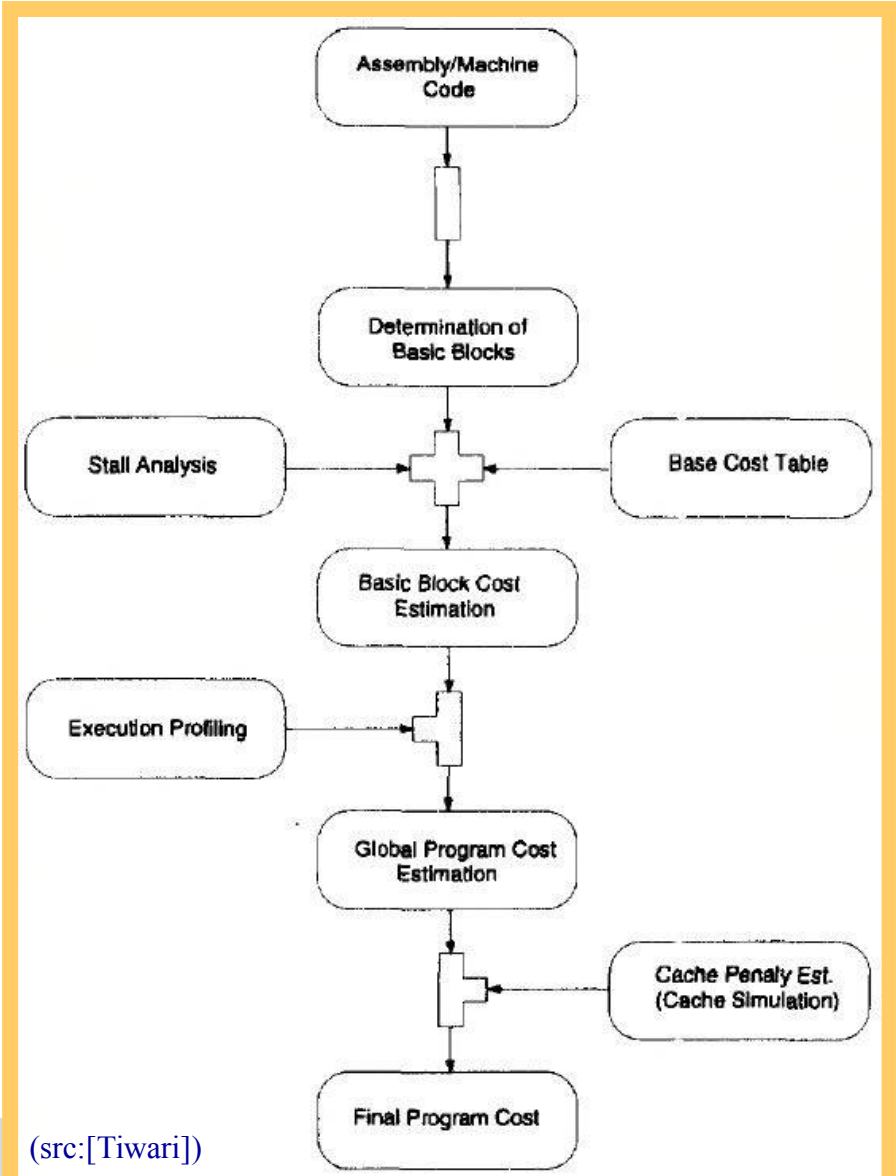
= *384.0mA*

Measured current = *385.0mA*

Similar experiments in 486DX2 and SPARClite accurate to within 3%

Estimation flow: summary

- Input: Assembly/Machine Code
- Output: Energy cost of final program



(src:[Tiwari])

Software power optimization: Example

- register optimizations
- Original code: lcc
- Optimized code: hand-generated
  - 9% current reduction
  - 24% running time reduction
  - 40.6% energy reduction
  - 33% for circle

Program	sort		circle	
Version	Original	Final	Original	Final
Current (mA)	525.7	486.6	530.2	514.8
Ex. Time (ms)	11.02	7.07	7.18	4.93
Energy (10-6J)	19.12	11.35	12.56	8.37
Saving		40.60%		33.40%

Compiler Generated Code

```
push   ebx
push   esi
push   edi
push   ebp
mov    ebp,esp
sub    esp,24
mov    edi,dword ptr 014H[ebp]

mov    esi,1
mov    ecx,esi
mov    esi,edi
sar    esi,cl
lea    esi,1[esi]
mov    dword ptr -20[ebp],esi

mov    dword ptr -8[ebp],edi
L3:
mov    edi,dword ptr -20[ebp]

cmp    edi,1
jle    L7
mov    edi,dword ptr -20[ebp]

sub    edi,1
mov    dword ptr -20[ebp],edi

lea    edi,[edi*4]
mov    esi,dword ptr 018H[ebp]

add    edi,esi
mov    edi,dword ptr [edi]
mov    dword ptr -12[ebp],edi

jmp    L8
L7:
mov    edi,dword ptr 018H[ebp]

mov    esi,dword ptr -8[ebp]
lea    esi,[esi*4]
add    esi,edi
mov    ebx,dword ptr [esi]
mov    dword ptr -12[ebp],ebx

mov    edi,dword ptr 4[edi]
mov    dword ptr [esi],edi
mov    edi,dword ptr -8[ebp]
sub    edi,1
mov    dword ptr -8[ebp],edi
```

Energy Efficient Code

```
push   ebp
mov    edi,dword ptr 08H[esp]

mov    esi,edi
sar    esi,1
inc    esi
mov    ebp,esi
mov    ecx,edi
L3:
cmp    ebp,1
jle    L7
dec    ebp
mov    esi,dword ptr 0cH[esp]
mov    edi,dword ptr[edi*4][esi]
mov    ebx,edi
jmp    L8
L7:
mov    edi,dword ptr 0cH[esp]

mov    esi,dword ptr 4[edi]
mov    ebx,dword ptr [ecx*4][edi]
mov    dword ptr [ecx*4][edi],esi

dec    ecx
cmp    ecx,1
jne    L8
mov    dword ptr 4[edi],ebx
jmp    L2
```

heapsort example

src.: [Tiwari]
A. Raghunathan, NEC

Overview

- Software power analysis/measurement
- Software power estimation models
- Optimizing software for low power through compilation phase
  - Instruction scheduling
  - Compiler-driven DVS

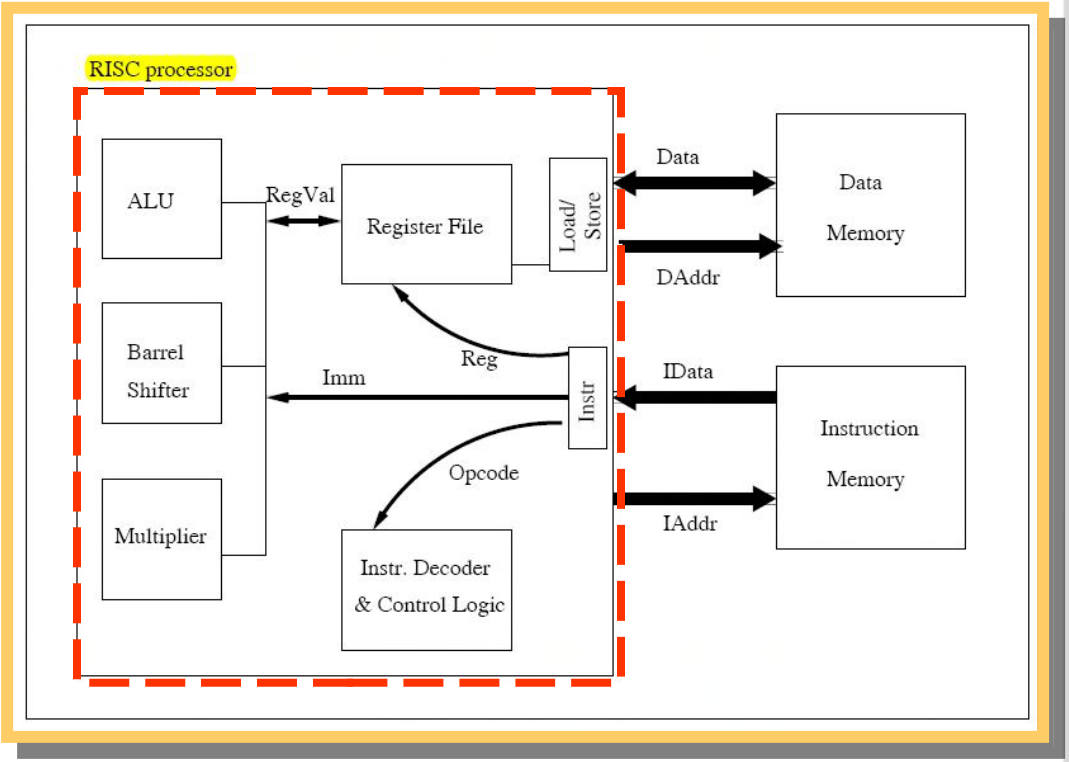
A detailed instruction-level power model

CPU-intern CPU-extern

$$E_{total} = E_{cpu\_instr} + E_{cpu\_data} + E_{mem\_instr} + E_{mem\_data}$$

(src:[Steinke])

- distinction between instruction dependency and data dependency
a) instruction-dependent cost inside the CPU
b) data-dependent cost inside the CPU
c) also considered but not discussed here: power extern to the CPU



A detailed instruction-level power model (cont'd)

- E_{CPU_instr}

$$E_{cpu\_instr} = \sum_{i=1}^{m} \Big(BaseCPU(Opcode_i) + \sum_{j=1}^{s} (\alpha_1 * w(Imm_{i,j}) + \beta_1 * h(Imm_{i-1,j}, Imm_{i,j})) + \sum_{k=1}^{t} (\alpha_2 * w(Reg_{i,k}) + \beta_2 * h(Reg_{i-1,k}, Reg_{i,k})) + \sum_{k=1}^{t} (\alpha_3 * w(RegVal_{i,k}) + \beta_3 * h(RegVal_{i-1,k}, RegVal_{i,k})) + \alpha_4 * w(IAddr_i) + \beta_4 * h(IAddr_{i-1}, IAddr_i) + FUChange(Instr_{i-1}, Instr_i) \Big)$$

Instruction-dependent costs inside the CPU depend on:
- the internal buses carrying the immediate value *Imm*
- the register numbers *Reg*, values kept within the registers *RegVal*
- and the instruction address *IAddr*.

(src:[Steinke])

A detailed instruction-level power model (cont'd)

- E_{CPU_data}

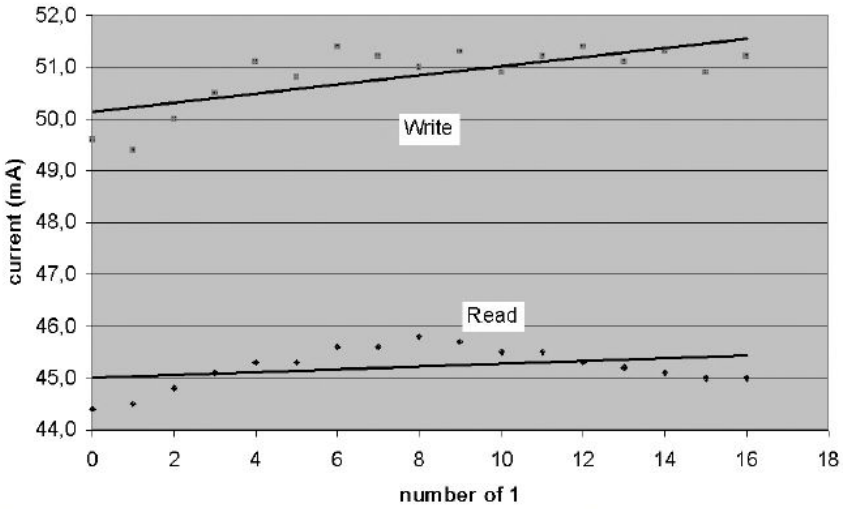
$$E_{cpu\_data} = \sum_{i=1}^{n} \Big(\alpha_5 * w(DAddr_i) + \beta_5 * h(DAddr_{i-1}, DAddr_i) + \alpha_{6,dir} * w(Data_i) + \beta_{6,dir} * h(Data_{i-1}, Data_i) \Big)$$

(src:[Steinke])

Data-dependent costs inside the CPU for *n* data accesses depend on the data address *DAddr*, the *Data* itself and on the direction *dir* (read/write)

A detailed instruction-level power model (cont'd)

- Results and parameters



parameter	energy (pJ) Read	Write	parameter	energy (pJ) Read	Write
α_4, α_5	n.a.	48.0	β_4, β_5	n.a.	219.9
α_6	11.0	26.4	β_6	-5.5	224.1
α_7, α_9	n.a.	-19.2	β_7, β_9	n.a.	138.9
α_8	-115.3	n.a.	β_8	57.7	n.a.
α_{10}	-115.3	-60.4	β_{10}	57.7	22.8

-parameters of ARM7TDMI energy model

(src:[Steinke])

CPU current depending on number of 1's on data bus

Overview

- Software power analysis/measurement
- Software power estimation models
- Optimizing software for low power through compilation phase
  - Instruction scheduling
  - Compiler-driven DVS

Low-power compilers

- Use instruction-level energy costs to guide code generation
- Minimize memory accesses
  - Utilize registers effectively
  - Reduce context saving
- Processor-specific optimizations
  - dual memory loads, instruction packing
- Optimize instruction scheduling to reduce activity in specific parts of the system
  - internal instruction-bus, processor-memory bus, instruction register and register decoder

(src.: A. Raghunathan, NEC)

Instruction scheduling for low power

- Traditional instruction scheduling strategies
  - Reordering instructions in order to
    - avoid pipeline stalls
    - improve resource (register file, etc.) usage
    - increase instruction level parallelism (ILP)
    - ...
  - main goal: increase performance
- Traditional steps for instruction scheduling
  1) partition program into regions or basic blocks
  2) build a control dependency graph (CDG) and data dependency graph
  3) schedule instructions within resource constraints

Instruction scheduling

- Traditional:

  $D(I_j I_{j+1})$ - number of pipeline stalls between instruction I_j und I_{j+1}

  Goal: minimize the number of all pipeline stalls PS within a basic block or region:

  $$PS = \sum D(I_j, I_{j+1}), j = 0, \ldots, n-1$$ (src: Despain)

- Idea:
  - Minimize switching those activities that depend on the sequence of instructions i.e. context sensitive switching
    - Examples: loading instructions in registers, using same or different operands, …
    - Switching activities may be measured by gate-level simulation running an instruction of a sequence of instructions on an RTL model of the processor architecture
    - Idea: use the profiled switching activities as a cost function for instruction scheduling through re-ordering
      - Assumption: there is leeway in data and control dependency graph that allows re-ordering
      - Re-ordering may cost performance => prefer those re-orderings that incur no or little penalty for performance

Instruction Scheduling
- “Cold Scheduling” (Despain et al.) -

$S(I_j, I_{j+1})$ -Switching activity (# of transistor switches) in the processor when instruction I_j+1 is executed right after I_j

$$BS = \sum S(I_j, I_{j+1}),\ j = 0, \ldots, n-1$$

-Switching activity in a basic block

$$\cos t = \frac{1}{k}(w_1 * BS_1 + \ldots + w_k * BS_k)$$

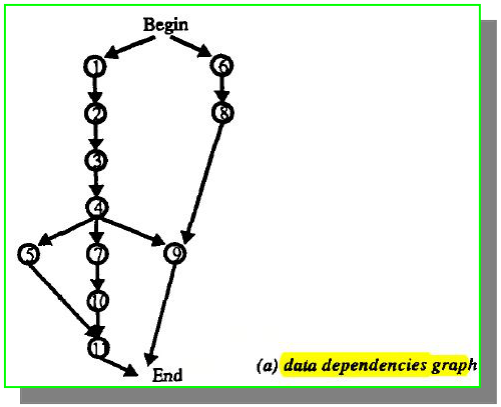
- cost function (k - # of basic blocks; w_i weight function takes into consideration dynamic execution frequency (profiling)

- Problem:
  - typically, instruction scheduling and register allocation are performed before assembly code using symbolic forms
  - It may be difficult to obtain bit switching information from symbolic representation
    - a) jump/branch targets may not be known before scheduling and register allocation
    - b) sizes of basic blocks may change during scheduling and register allocation
    - c) binary representation of indexes to symbol table may not be available

  ⇒ **Phase problem** of instruction scheduling and assembly
    - -If scheduling precedes assembly => may reduce potential of reducing bit switches
    - -If assembly precedes scheduling => flexibility of scheduling is limited

  => One solution: need to estimate binary representation of an instruction

Instruction Scheduling
- “Cold Scheduling” (cont’d) -



(src:Despain)

label(l(move/4,7)). $BS = 1$

1	mov(e,t0).	%D(1,2) = 0
2	umax(b,e,e).	%D(2,3) = 0
3	addi(e,4,e).	%D(3,4) = 0
4	pushd(t0/cp,e,2).	%D(4,6) = 0
6	ldi(-1,t1).	%D(6,7) = 0
7	std(r(2)/r(1),e+ -4).	%D(7,8) = 0
8	add28(r(0),t1,r(0)).	%D(8,10) = 0
10	mov(r(3),r(2)).	%D(10,9) = 0
9	st(r(0),e+ -6).	%D(9,5) = 1

Pipeline stalls = 1 **(b) Instruction sequence I**

label(l(move/4,7)). $BS = 1.05$

6	ldi(-1,t1).	%D(6,1) = 0
1	mov(e,t0).	%D(1,2) = 0
2	umax(b,e,e).	%D(2,3) = 0
3	addi(e,4,e).	%D(3,8) = 0
8	add28(r(0),t1,r(0)).	%D(8,4) = 0
4	pushd(t0/cp,e,2).	%D(4,7) = 1
7	std(r(2)/r(1),e+ -4).	%D(7,9) = 1
9	st(r(0),e+ -6).	%D(9,5) = 1
5	st(r(3),e+ -5).	%D(5,10) = 0
10	mov(r(3),r(2)).	%D(10,11) = 0

Pipeline stalls = 3 **(c) Instruction sequence II**

label(l(move/4,7)). $BS = 1.45$

1	mov(e,t0).	%D(1,2) = 0
2	umax(b,e,e).	%D(2,6) = 0
6	ldi(-1,t1).	%D(6,3) = 0
3	addi(e,4,e).	%D(3,4) = 0
4	pushd(t0/cp,e,2).	%D(4,5) = 1
5	st(r(3),e+ -5).	%D(5,8) = 0
8	add28(r(0),t1,r(0)).	%D(8,7) = 0
7	std(r(2)/r(1),e+ -4).	%D(7,10) = 0
10	mov(r(3),r(2)).	%D(10,11) = 0
11	ld(e-4, r3).	%D(11,9) = 0

Pipeline stalls = 1 **(d) Instruction sequence III**

shown:
- dependency graph and three schedules of a code sequences
- schedule info shows pipeline stalls
- schedule has been done without low power scheduling

- for each schedule, the normalized switching activity has been calculated
-Schedule I: is 1
-Schedule II is 1.05 (i.e. plus 5%)
-Schedule III is 1.45 (plus 45%)

Conclusion: no clear correlation between low power (i.e. low BS) and high performance (i.e. few pipeline stalls) => there is hope that often energy/power savings can be achieved without performance loss!

Instruction Scheduling
- “Cold Scheduling” (cont’d) -

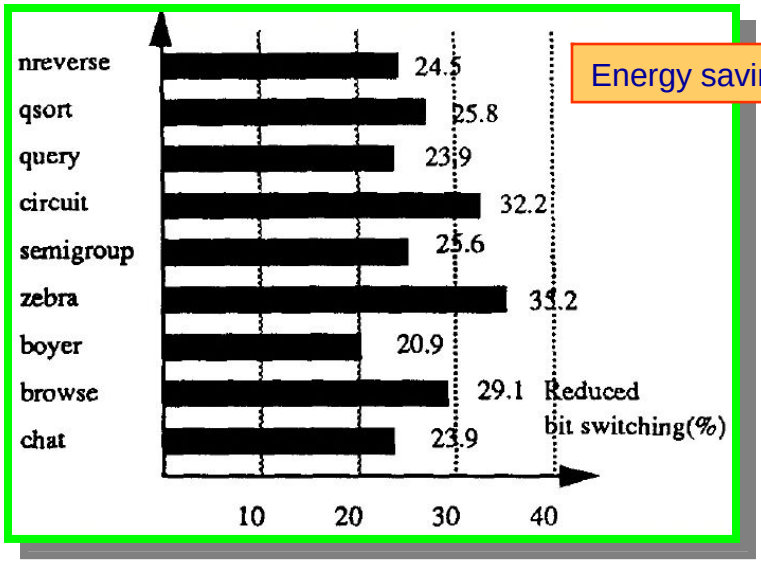
Cold Scheduling

INPUT: DAG representation and bit switching table
OUTPUT: A scheduled instruction stream

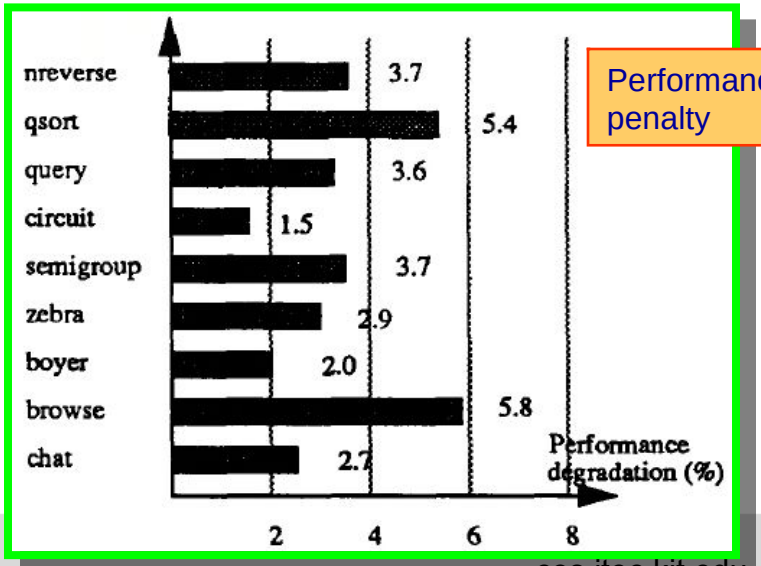
0. Set ready list RL to be { }
   Set the last scheduled instruction LSI = NOP
1. Remove ready instructions from DAG and
   add these ready instruction into RL.
2. For each instruction I in RL,
   find S(LSI,I).
3. Remove an instruction I with the smallest S(LSI,I) from RL.
   The removed instruction becomes the current LSI.
   Write out LSI.
4. IF there is any instruction yet to be scheduled,
   THEN go to step 1,
   ELSE return.

Figure 10 Cold Scheduling Algorithm

(src: Despain)



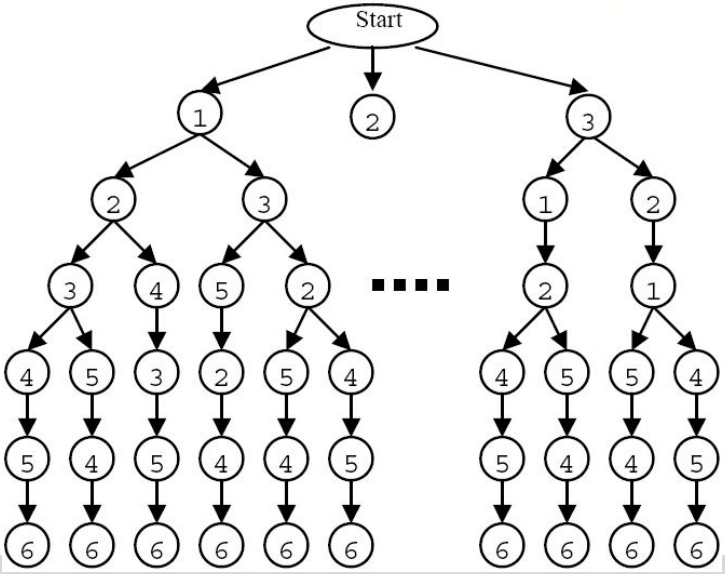
Energy savings

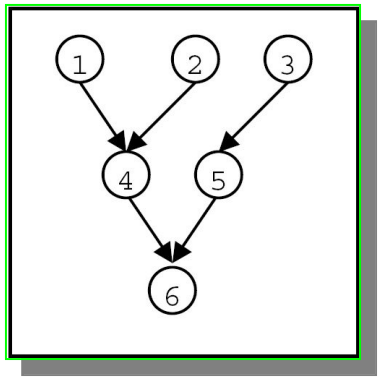


Performance penalty

Complexity and design space when considering instruction sequences

- Q: Assume n instructions. How many instruction sequences?
- A: (n-1)! / 2
- Ex: 11 instructions (a medium-sized basic block) => ~16Mio sequences
  Each sequences can potentially have a different power consumption
- In practice: it is less since there are precedence constraints



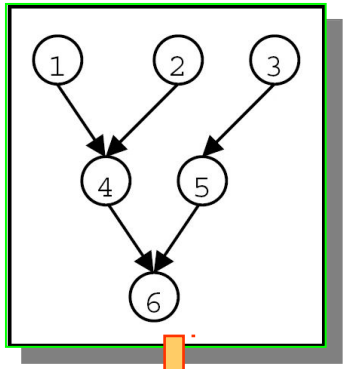


precedence constraints among instructions 1, 2, 3, 4, 5, 6

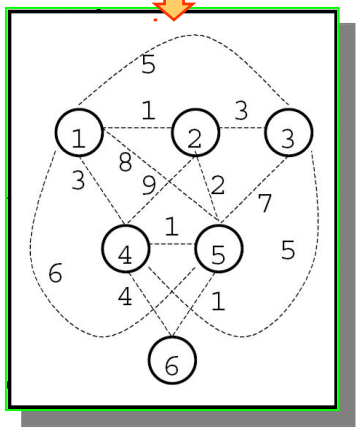
An efficient approach to the instruction scheduling problem

	subu	move	lw	sll	addu
subu	5	1	5	3	6
move	1	2	3	9	2
lw	5	3	2	8	7
sll	3	9	8	2	1
addu	8	2	7	1	1

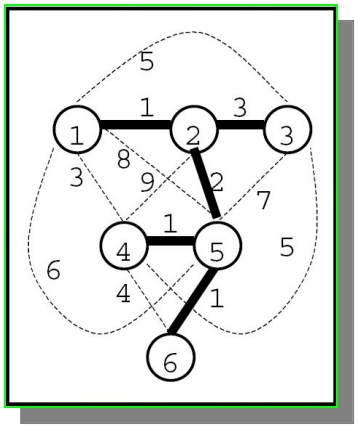
power dissipation table: When an instruction in leftmost column is followed by instruction in top row, then the given power consumption applies



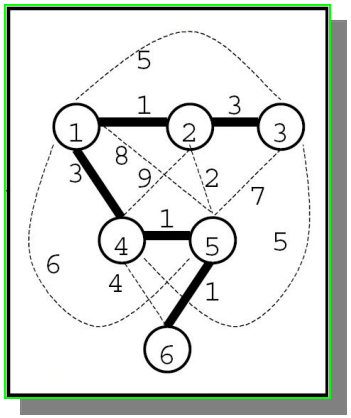
control dependency graph: Gives dependency constraints
Ex 1: before '4', '2','1' needs to execute;
Ex 2: '1', '2', '3' can be executed in any order



(src: chatterjee)



Minimum Spanning Tree (MST)

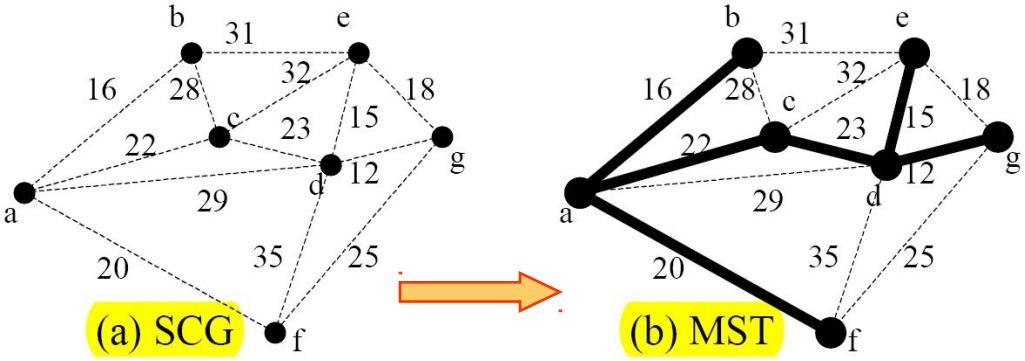


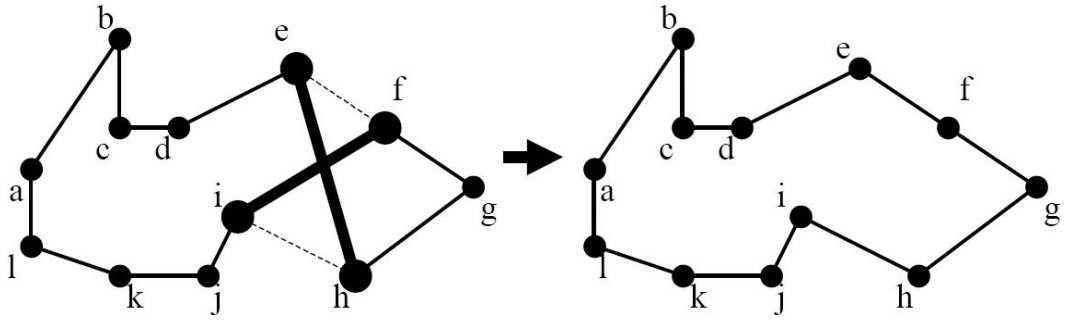
Using Simulated Annealing for finding Hamiltonian Path =>that is the energy efficient instruction sequence

Weighted strongly connected graph: Contains all edges of CDG plus: edges between any two nodes where precedence is not important (like 1<->2, 1<->3, 2<->3 etc.). This may be one or two edges subject to whether costs are different. Each weight gives power cost for repeated execution of the two connected instructions

An efficient approach to the instruction scheduling problem (cont'd)

- Problem can be identified as the TSP problem => cannot be solved in polynomial time. In fact, it is NP-hard => needs a heuristic like, for example, Simulated Annealing
- Some details on MST (Minimum Spanning Tree) and TSP with Simulated Annealing





(src: chatterjee)

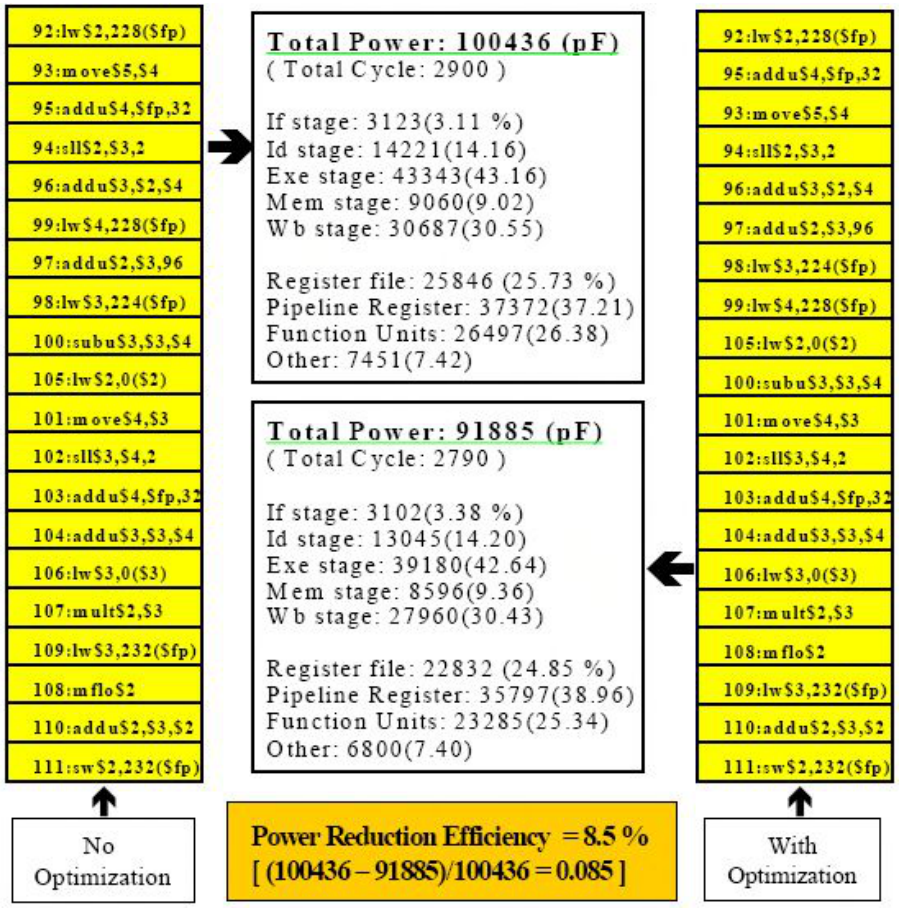
1) Computing the minimum cost spanning tree with ***Prim's Algorithm*** (greedy)
Ex: when starting with vertex a, edges are chosen in the order ab, af, ac, cd, dg, de

2) Computing MST is a constructive method to find an initial path. MST can be converted to a path using ***Christofide's Algorithm***, for example.

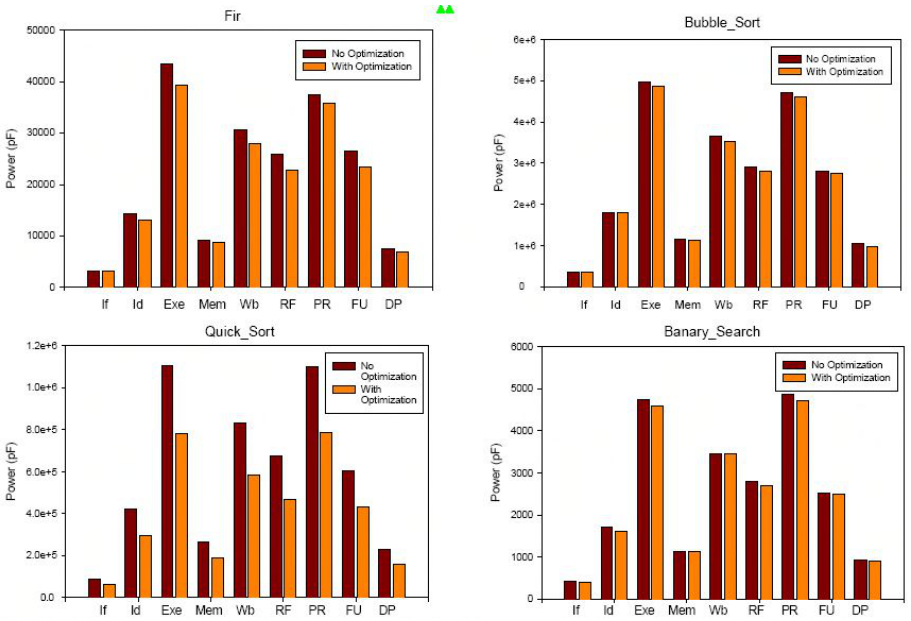
3) The initial path is improved using ***Simulated Annealing*** and 2-optimal mechanisms:
-non-adjacent pair of edges are selected and deleted => 2 paths
- recombine 2 path to 1 path (is unique) according to optimization algorithm (accept or reject the "move")

An efficient approach to the instruction scheduling problem (cont'd)

- Power saving results



(src: [chatterjee])



Power break down: Instruction Fetch, Id: Instruction Decode, Exe: Execution, Mem: Memory access, Wb: Wright back, RF: Register file, PR: Pipeline Register, FU: Functional Units, DP: Other Data Paths

Scheduling Example for one BB of FIR Source

Is optimizing SW for low power equal to increasing its performance?

- Traditionally 'yes' because the following optimizations lead to both power/energy reduction and performance increase
  - Common sub-expression elimination
  - Dead code elimination
  - Memory hierarchy optimizations: loop tiling, keeping data on-chip instead of off-chip, …
- **BUT:** there are fundamental differences in metrics/models used for low power/energy and performance
  - Ex 1: critical path is often used for performance constraints. When modifying the off-critical path in that case, performance is generally not affected; may instead lead to a decrease of the critical path;

    power/energy, on the other hand, are affected! Note any activity on or off the critical path will contribute to power/energy consumption.
  - Ex 2: moving loop invariant code:
    - Assumption: - code is loop invariant;
      - Case 1: on scalar machine performance optimization will move code out of the loop
      - Case 2: on a VLIW leaving code in may increase performance if there are empty slots. Then, critical path may be reduced. But: power goes up since ten times executed
  - Ex 3: speculative computation

```
for (i=0; i<10; i++) {
    a = b * 2;
    c[i] = d[i] + 2.0;
}
```

```
a = b * 2;
for (i=0; i<10; i++) {
    c[i] = d[i] + 2.0;
}
```

(src: [Kermer])

Overview

- Software power analysis/measurement
- Software power estimation models
- Optimizing software for low power through compilation phase
  - Instruction scheduling
  - Compiler-driven DVS

Some DVS basics

- DVS – Dynamic Voltage Scaling (the most effective method for power savings due to quadratic relationship between power and supply voltage
  - DVS comes at the cost of performance degradation. Idea: deploy DVS such that it does not incur a penalty
  - An effective DVS strategy:
    - determine intelligently when to adjust the voltage setting (i.e. find the best 'scaling points')
    - Where to adjust to i.e. which voltage setting to choose (i.e. 'scaling factors')
  - Overhead:
    - Switching to and from new voltage setting costs time and energy (=> may reduce or eliminate potential savings)
      - Hundreds of micro-seconds i.e. tens of thousands of instructions! (i.e. not even cache misses may be used to perform the transition)
  - Considered here: intra-task DVS: i.e. scaling points may be in the middle of the task execution (in contrast to inter-task DVS)
    - Sub-categories:
      - 1) interval-based DVS: fixed-length time intervals rely solely on state of the system and trace history. Scaling points determined online or offline
      - 2) checkpoint-based DVS: scaling points are determined offline, scaling factors are determined online. Scaling points are placed at selected branches to exploit the slacks due to run-time variations

Compiler-directed DVS

$$\min_{R,f} \; P_f \cdot T(R,f) + P_{f_{max}} \cdot T(P-R, f_{max}) + P_{trans} \cdot 2 \cdot N(R)$$

with

$$T(R,f) + T(P-R, f_{max}) + T_{trans} \cdot 2 \cdot N(R) \leq (1+r) \cdot T(P, f_{max})$$

(src: [Kremer03])

Compiler-directed DVS problem:
Given a program *P*, find a region *R* and a frequency *f* such that, if region *R* is executed at frequency *f* and the rest of the program *P* − *R* is executed at the peak frequency f_{max}, the total execution time plus the switching overhead T_{trans} 2 *N*(*R*) is increased no more than *r* percent of the original execution time *T*(*P*, f_{max}), while the total energy usage is minimized.

T(*R*, *f*) - total execution time of region *R* running at frequency *f*

N(*R*) - # of times region *R* is executed,

-> *specify input behavior (kept in table)*

P_f - power consumption of the system at frequency *f*

T_{trans}, P_{trans} - single switching overhead in terms of performance, power, respectively.

-> *modeling underlying machine*

r - specified by user

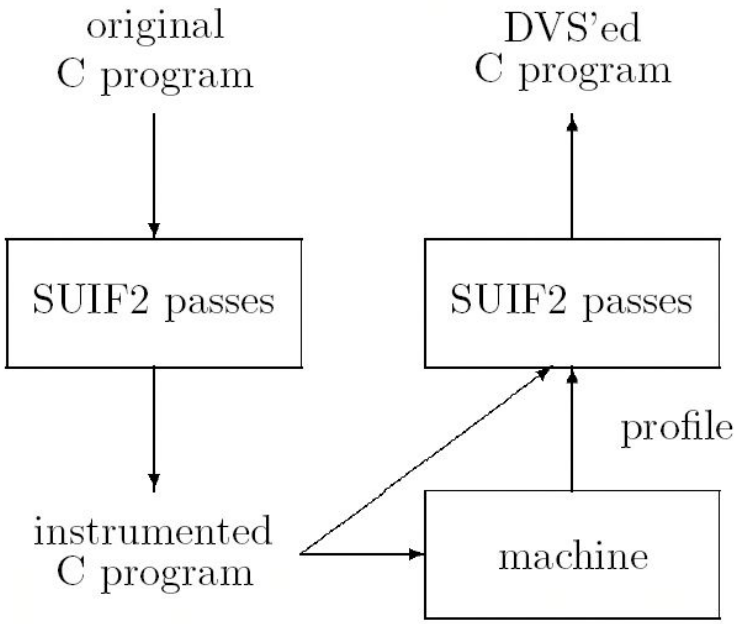
(src: [Kermer03])

Compiler-directed DVS (cont'd)

Additional constraint:
It needs to be made sure that region R is sufficiently large such that exec time is larger than a DVS call:

$$T(R, f_{max})/T(P, f_{max}) \geq \rho$$

ρ - compiler-directive



(src: [Kremer03])

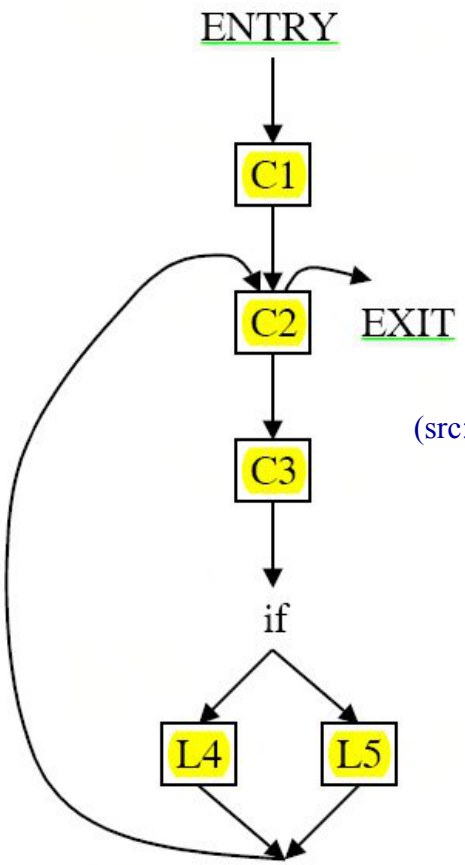
Steps of compiler-directed DVS:

1) instrumenting: the input program at selected program locations
2) profiling: the instrumented code is executed, filling a subset of entries in tables *T*(*R*, *f*) and *N*(*R*)
3) rest of table entries are derived (using call graphs etc.); based on inter-procedural analysis -> analysis is faster than profiling
4) the minimization problem is solved by enumerating all possible regions and frequencies.
5) the corresponding DVS system calls are inserted at the boundaries of the selected region.

Compiler-directed DVS (cont'd)

Example

- assumptions: - only two CPU frequencies f_max, f_min
  - C call sites
  - L loop nest
- First all $N(R_i)$, $T(R_i, f)$ are profiled for the basic regions
- Combined regions: one entry point, one exit point => all top level statements are executed same # of times
- example for combined regions: if(L4, L5), seq(C2, C3)
- not allowed: seq(C1, C2), seq(C3, L4), …
- C1, C3 are the only call points of function foo
  => $T(\texttt{C1}, f) = T(foo, f) \cdot 1/(1+10)$ (C3 accordingly)



Control flow between basic regions

(src: [Kremer03])

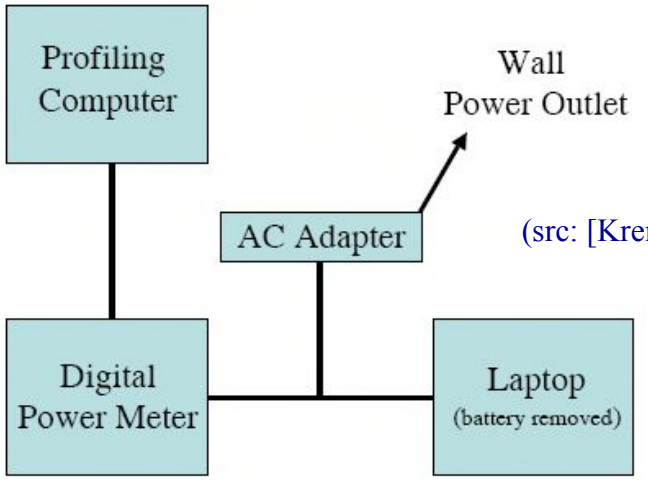
R	$N(R)$	$T(R,f)$	
		f_{max}	f_{min}
C1	1	0	0
C2	10	10	12
C3	10	0	0
L4	8	8	12
L5	2	2	4

Profiled data

Remarks:
The profile-driven approach gives results that are not portable but it captures properties that may not be captured using a compile-time prediction model

Compiler-directed DVS (cont'd)

- Results and experimental setup



(src: [Kremer03])

Experimental setup

parameter	value
$T(R,f)$	profiled
$N(R)$	profiled
P_f	$V_f^2 \cdot f$
T_{trans}	20 μs
P_{trans}	0 W
r	5%
ρ	20%

input parameter for DVS algorithm

	total compilation time	instru-mentation phase	profiling phase	selection phase
swim	34	7	8	19
tomcatv	173	4	158	11
hydro2d	340	44	173	123
su2cor	403	37	257	109
applu	284	83	13	188
apsi	1264	157	40	1067
mgrid	190	10	152	28
wave5	544	151	48	345
turb3d	1839	39	268	1532
fpppp	1628	82	11	1535

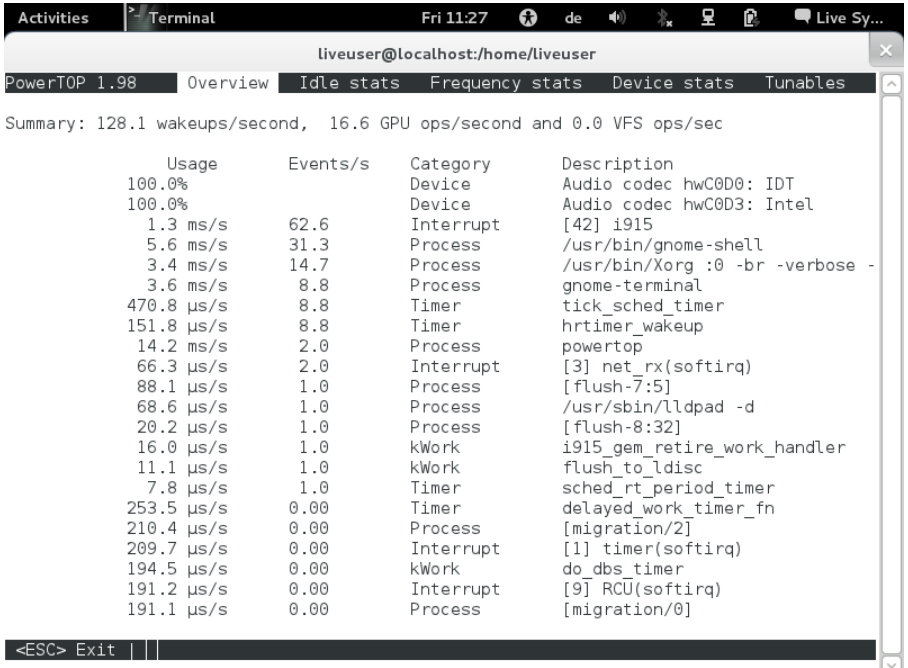
Compilation time [s]

- Results:
  - Power savings: 0%-28%
  - Performance penalty: 0%-4.7%

Conclusion

- Software power estimation is possible and necessary
  - It represents a high level of abstraction and therefore it is faster than estimating power consumption of the underlying hardware circuitry
- Compiler may include optimization for low power
  - Instruction scheduling
  - Intra-procedural DVS
- Optimizing for low power and high performance are two distinct tasks!

Powertop Demo



(src.:[Heise])

Powertop Demo

- open source power analysis tool
- diagnose issues related to
  - power consumption
  - power management
- interactive mode for experimentation
- „tuning“ possible from within powertop

(src.: [Powertop])

Sources

[Steinke] Stefan Steinke, Markus Knauer, Lars Wehmeyer, Peter Marwedel, “An Accurate and Fine Grain Instruction-Level Energy Model Supporting Software Optimizations”, PATMOS 2001.

[Kremer03] Chung-Hsing Hsu, U. Kremer, "The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction", ACM Conference on Programming Language Design and Implementation, pp. 38-48, 2003.

[A. Raghunathan, NEC] Tutorials on low power held at various CAD conferences.

[Despain] Ching-Long Su and Chi-Ying Tsui and Alvin M. Despain, "Low Power Architecture Design and Compilation Techniques for High-Performance Processors", In CompCon’94 Digest, pp.489-498, February 1994.

[Chatterjee] Kyu-won Choi, Abhijit Chatterjee, "Efficient Instruction-Level Optimization Methodology for Low-Power Embedded Systems", IEEE Proceedings of the 14th international symposium on Systems synthesis (ISSS '01), Efficient Instruction-Level Optimization Methodology for Low-Power Embedded Systems, pp. 147-152, 2001.

[Tiwari] V. Tiwari, S. Malik, and A. Wolfe, “Power analysis of embedded software: A first step toward software power minimization,” IEEE Trans.VLSI Syst., vol. 2, pp. 437–445, Dec. 1994.

[Powertop] Accardi, C and Yates, A “Powertop User's Guide”, https://01.org/sites/default/files/page/powertop_users_guide_201412.pdf

[Heise] http://www.heise.de/open/artikel/Powertop-2-0-Strom-sparen-unter-Linux-1167455.html