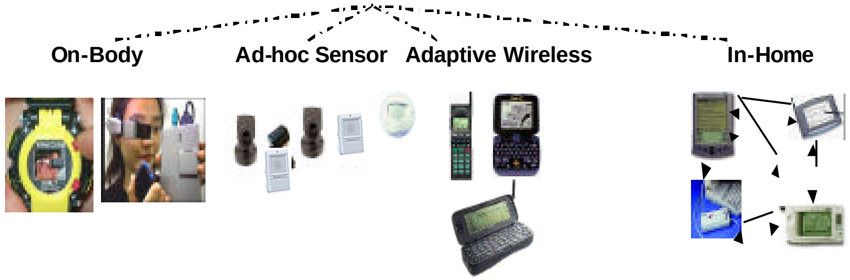
System-Level Power Optimization

Design Techniques & CAD Tools

Luca Benini
DEIS Universita' di Bologna, Italy
lbenini@deis.unibo.it

The vision: Ambient Inteligence

Devices as Appliances



On-Body

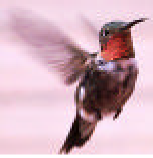
Ad-hoc Sensor

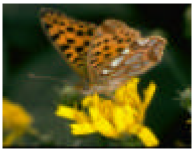
Adaptive Wireless

In-Home

- ***Energy-efficient communcations is the cornerstone of ambient intelligence***
- ***Requires highly efficient hardware & software***
  - RF circuits
  - Baseband protocol processing & appl.

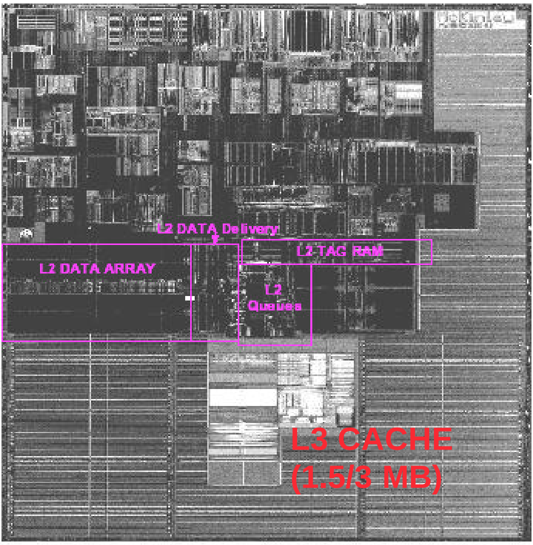
The AMI processing Bestiary





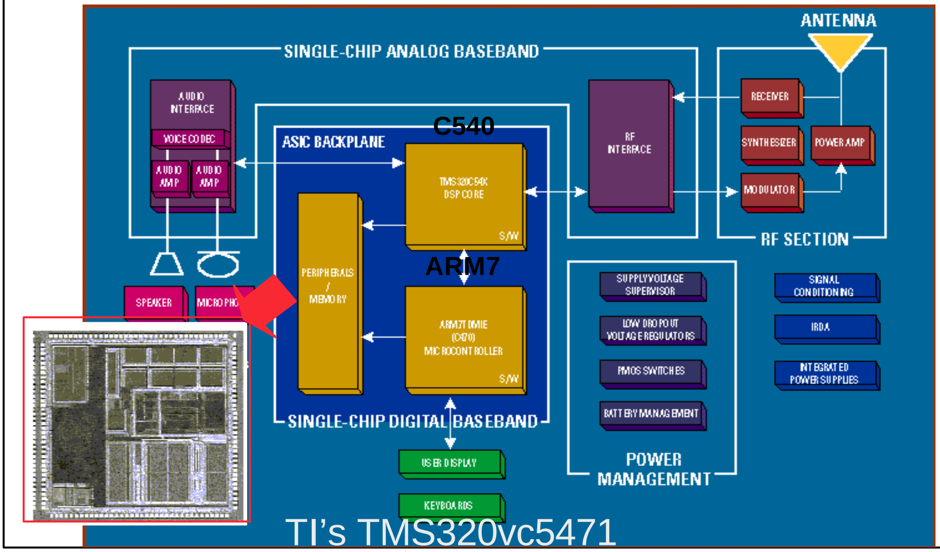
- The work-horse
  - Powers the fixed base network machines
  - Power W Performance GB/s
- The hummingbird
  - Powers the wireless base network interfaces
  - Power mW Performance MB/s
- The butterfly
  - The sensor network hardware
  - Power µW Performance KB/s

Itanium® 2 Processor



- Released at 733MHz and 800MHz, now 1GHz
- Three level caching system
- 25 million transistors in the CPU and 300 million in the cache (0.18µm)
- 421mm^2 die size
- The CPU running at full load draws ~130 Watts
- The clock signals and logic total to approx 84% of the total power usage.
- Leakage power: approx. 2%.
- Power delivery: V_{dd}=1.5V, P=130W, $P=V_{dd}I$ (!!)

Handset architecture

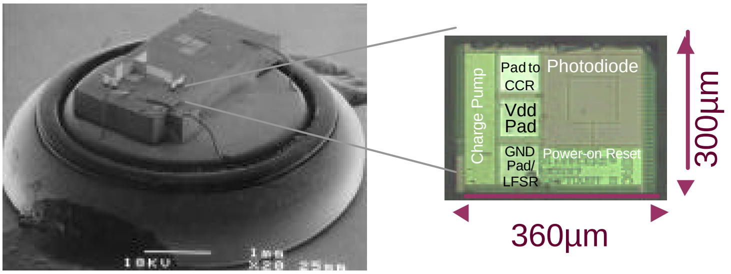


C540

ARM7

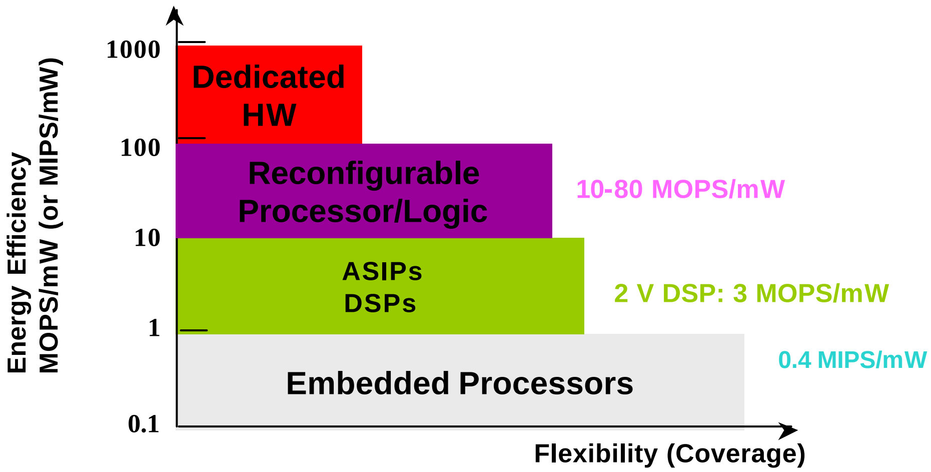
TI’s TMS320vc5471

Berkeley’s Daft Dust Device



- 63 mm^3
- Circuits: 0.25 µm CMOS
  - digital circuits underneath ground pad
  - metal shields to prevent photogenerated carriers
- CCR: Cronos MUMPS
- ☹ Optical wireless connection (line of sight)

The Energy-Flexibility Tradeoff



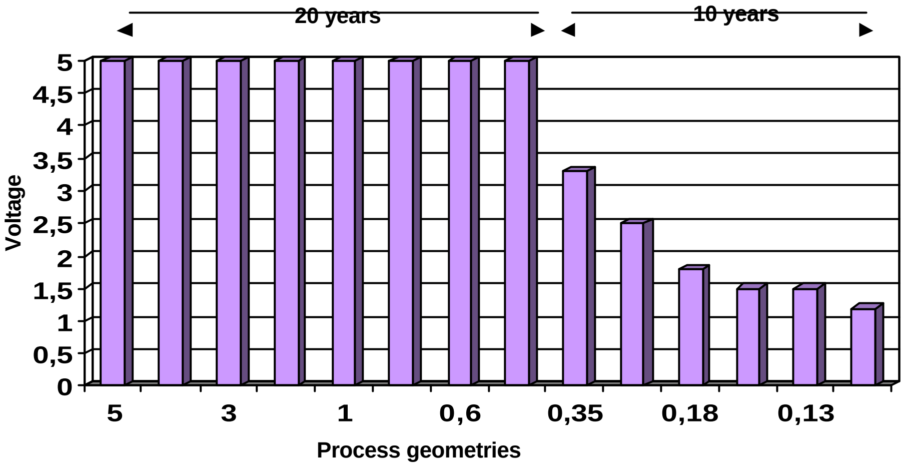
Why designing low-power circuits/systems?

- Practical reasons
  - – Extend battery-lifetime of high-throughput portable applications.
- Financial reasons:
  - – Reducing costs of: Packaging, PCB, Heat-sinks, Ventilation.
  - – Reducing ownership cost
- Technological reasons:
  - – Producing high-density chips:
  - – Interconnect design issues:
  - – Power delivery and distribution.
  - – Reliability issues:

Deep Sub-Micron Technologies

- Smaller geometries:
  - Higher device densities.
  - Higher clock frequencies.
- Consequence:
  - Greater power consumption in spite of lower supply voltages:
    - Technology scales faster than supply voltage.

Voltage Trends



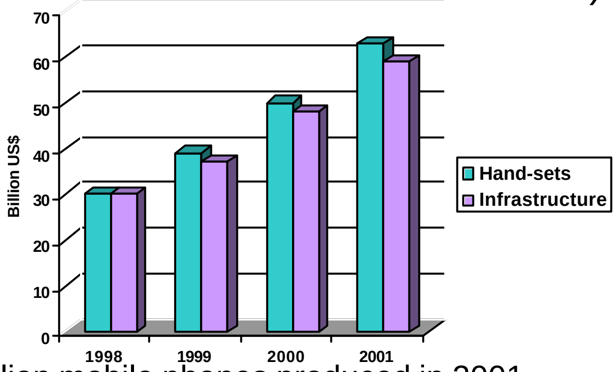
Power Trends

Example: Alpha processor

	1992	1994	1998	1999	2001
Process (m)	0.75	0.5	0.35	0.25	0.18
Clock speed (MHz)	200	300	667	750	1000
Transistors (millions)	1.68	9.3	15.2	15.2	100
Voltage	3.3	3.3	2.3	2.1	1.5
Power (W)	30	50	72	90	100

Mobile Electronics (I)

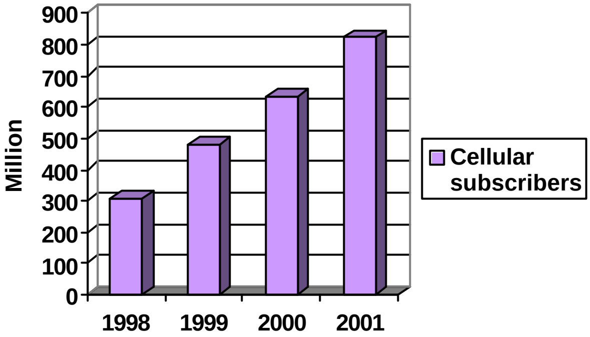
- Wireless communication (appliances and infrastructure)



- 600 million mobile phones produced in 2001.

Mobile Electronics (II)

- Cellular network subscribers



  - 1.9 billion subscribers predicted for year 2004.

Battery Technology

- Battery maximum power and capacity increase by 10-15% per year.
- Chip power requirements increase much faster: 35-40%. [source: 1999 SIA Technology Roadmap]
- Consequence:
  - Larger gap between battery technology enhancements and chip power demand.

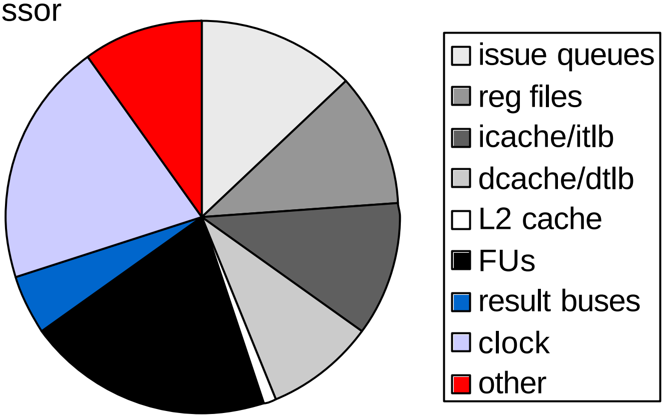
Not Only Mobile

- 20% of electrical energy consumed in Amsterdam is used for telecom.
- In the US, Internet is responsible for 9% of the electrical energy consumed nation-wide.
  - This grows to 13% if all computer applications are considered.
- The transfer of 2 MBytes of data through the net consumes the energy of 1 pound of coal.

[source: 2000 CO_2 conference, Amsterdam, NL]

Where Does the Power Go?

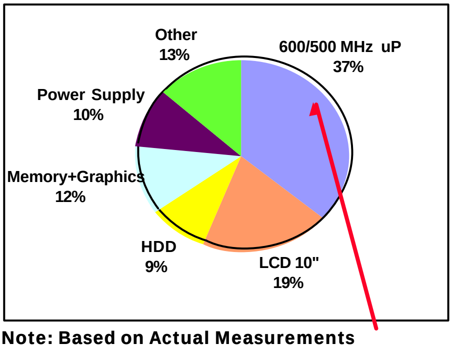
- Power profile (dynamic power) of a 4-way superscalar microprocessor



- Bottom line: power needs to be reduced across-the-board

Need to consider CPU & System Power

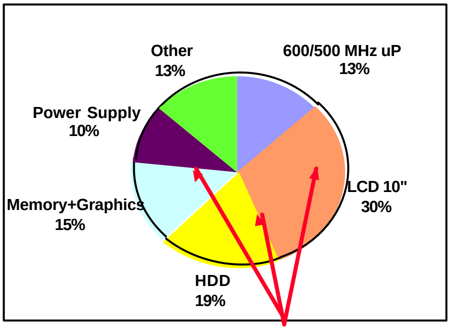
Mobile PC
Thermal Design (TDP) System Power



Note: Based on Actual Measurements

CPU Dominates Thermal Design Power

Mobile PC
Average System Power



Multiple Platform Components Comprise Average Power

[Courtesy: N. Dutt; Source: V. Tiwari]

Design for Low Power (I)

- CMOS technology dominates in modern ICs.
- Power consumption of a CMOS gate:

$$P = P_{SW} + P_{SC} + P_{Lk}$$

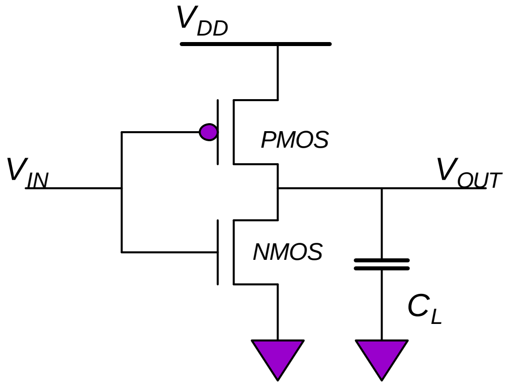
where:

- – P_{SW} = Switching (or dynamic) power.
- – P_{SC} = Short-circuit power.
- – P_{Lk} = Leakage (or stand-by) power.

- So far, switching power minimization has been the primary objective.
- In deep sub-micron, low-voltage processes, leakage power becomes critical.

Design for Low Power (II)

- CMOS inverter:



$V_{IN} = 0$ => $V_{OUT} = 1$

$V_{IN} = 1$ => $V_{OUT} = 0$

Design for Low Power (III)

- Switching power of a CMOS gate:

$$P_{SW} = 0.5\, V_{DD}^2\, f_{CK}\, C_L\, E_{SW}$$

where:

– V_{DD} = Supply voltage.
– f_{CK} = Clock frequency.
– C_L = Output load capacitance.
– E_{SW} = Switching activity factor.

- Design for low switching power:

Minimization of V_{DD}, f_{CK}, C_L and E_{SW}.

Design for Low Power (III)

- V_{DD} and/or f_{CK} scaling:
  - Very effective.
  - Big impact on performance.
- Switched capacitance optimization (i.e., $C_L \times E_{SW}$):
  - Applicable at all levels of design abstraction.
  - Many techniques proposed.

Objectives

- Describe design techniques and tools for system-level design
- Address system-level modeling, design and power management issues
- Purposely neglect:
  - Chip-level design issues
    - physical and logic design
  - Distributed systems (e.g. wireless networks)

Electronic systems

- A system is a combination of:
  - Hardware:
    - – Computation units
    - – Storage units
    - – Communication units
    - – Peripherals
  - Software :
    - – Application and system software
- Energy is required by all hardware units
- Software organization affects how hardware consumes energy

Electronic system design

- Conceptualization and modeling:
  - From idea to model
- Design:
  - HW: computation, storage and communication
  - SW: application and system software
- Run-time management:
  - Run-time system management and control of all units including peripherals

Examples

- Modeling:
  - Choice of algorithm
  - Application-specific hardware vs. programmable hardware (software) implementation
  - Word-width and precision
- Design:
  - Structural trade-off
    - Resource sharing and logic restructuring
  - Exploit multiple/variable supplies
- Management:
  - Operating system
  - Dynamic power management

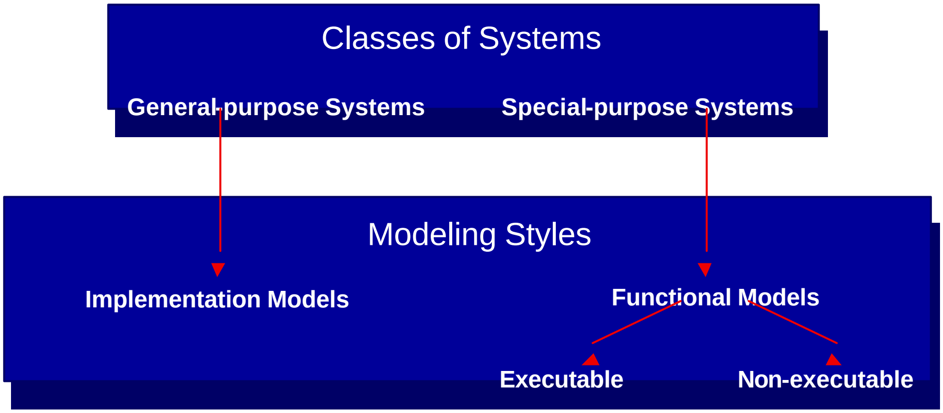
Outline

- Introduction
- System conceptualization and modeling
  - Modeling and design
  - Modeling for energy-efficient design
- System design
- System management
- Conclusions

System models

- Modeling is an abstraction:
  - Represent important features and hide unnecessary details
- Functional models:
  - Capture functionality and requirements
  - Executable models:
    - Support hw and/or sw compilation and simulation
- Implementation models:
  - Describe target realization

Taxonomy



Classes of Systems

General-purpose Systems

Special-purpose Systems

Modeling Styles

Implementation Models

Functional Models

Executable

Non-executable

Modeling Power at the System Level

- The abstraction challenge
  - Model complex behavior
  - …at a reasonnable computational effort
  - With “acceptable” accuracy
- A spectrum of approaches depending on the amount of functional information taken into account

The spreadsheet model

- General-purpose systems
  - Backward compatibility
  - Component-based
- Spreadsheet-based analysis
  - Basic budgeting
  - Simple “what if” analyses
  - No learning curve

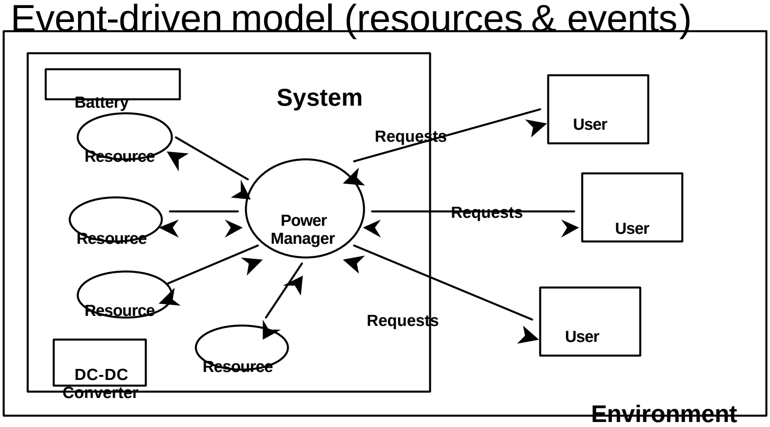
Example: spreadsheet analysis

PDA	#Comp	Vdd	Iidle	Ion	%on	%idle	I(mA)
Proc	1	3.3	0.5	50	0.7	0.3	36.15
DRAM	1	3.3	0.1	12	0.7	0.3	8.43
FLASH	5	3.3	0.0	9	0.7	0.3	31.5
IR	1	3.3	0.0	64	0.05	0.95	3.2
RTC	1	3.3	0.0	0.1	1	0	0.1
DC-DC	1	-	0.1	5.5	0.99	0.01	5.44

TOT 83.82

Power State Machines: System Model

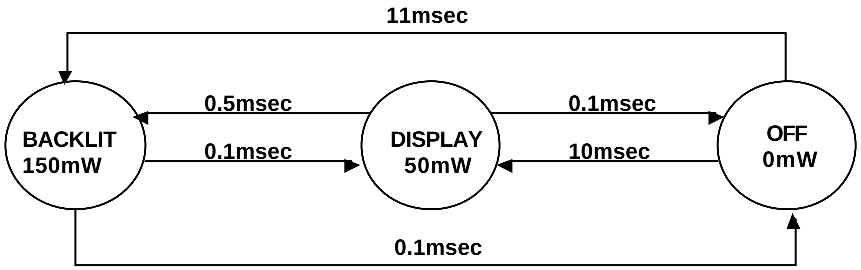
- Event-driven model (resources & events)



- Key feature:
  No overhead for long inactivity (no events).

Power State Machines: Resource Model

- Example of PFSM: LCD display unit.



- Key features:
  - Power associated with states
  - Transitions have a cost

Power State Machines: Additional Components

- Workload:
  - User/Environment:
    Non-deterministic FSM
    (models the non-deterministic nature of the requests).
- Power supply sub-system
  - Battery
  - DC-DC converter

Functional Power Models

- Objective:
  Estimate the power dissipated by a specific fragment of code
- Needs to track instruction execution
- Must be fast (millions of instructions)
  - RTL or Gate-level are not fast enough
- Needs to model processor & memory system

Software Power Estimation: Instruction-Level

ILPA [TMWL96]

- Empirical method for characterizing single (or very short sequences of) instructions.
- Key issues:
  - Evaluation of power dissipation for single instructions.
  - Choice of representative instructions for characterization.
- Advantage: Roughly architecture-independent.

Instruction-Level Power Characterization

- Direct measurement of the currents drawn from the power supply while executing the instructions.
- HDL simulation:
  - The instructions are simulated on a processor model in some HDL.
  - The processor is plugged into a tester machine and simulation traces are applied. The current is measured by the tester.
- Use simulation of a gate-level description of the processor.

Instruction-Level Models

- A power cost is assigned to each instruction.
- Two components of the cost:
  - Static component, called ``base-cost'': It is the individual instruction cost without a notion of ``state''.
  - Dynamic component, called ``circuit state effects'': It accounts for the previous processor state.
- Dynamic cost accounts for events depending on sequences of events (e.g., cache misses, pipeline stalls).

Extracting the model

- The base cost is computed as follows:
  - An infinite loop containing a total of *N* copies of the target instruction *I* is executed.
  - The average current is measured as described earlier.
  - The power cost is obtained from the values of the current, the supply voltage and the cycle/instruction.
- *N* should not be too small to amortize the loop overhead.

Computing program execution cost

- Due to the averaging process, the costs for $I_1 \rightarrow I_2$ and $I_2 \rightarrow I_1$ cannot be distinguished.
- The cost of a program can be summarized as follows:

$$Cost(Program) = S_i\,(B_i \cdot N_i) + S_{ij}\,(O_{ij} \cdot N_{ij}) + S_k\, E_k$$

where:

- B_i : Base cost of instruction *i*.
- N_i : # of occurrences of instruction *i.*
- O_{ij} *:* Dynamic cost of sequence →*j*.
- N_{ij} : # of occurrences of sequence →*j*.
- E_k: Other effects, obtained from program profiling.

Instruction-Level power model: example

- Example of power cost values (expressed in *pJ*):

Instruction Name	Base Cost	Circuit State Effects			
		LOAD	DLOAD	ADD	MULT
LOAD	1.98	0.13	0.15	1.19	0.92
DLOAD	2.37		0.17	1.19	0.92
ADD	0.99			0.26	0.53
MULT	1.19				0.66

- Example of computation:

Program (initial state is ADD)	Evaluation	
	Base Cost	Circuit State
DLOAD A←*x*, B ←*y*	2.37	1.19
LOAD C←*z*	1.98	0.15
ADD A←C, B	0.99	1.19
Total	3.34	2.53

Total value = $5.87pJ/(3 \cdot 25ns) = 78.26mW$ ($T_c = 25ns$)

Micro-architectural Power Model

- The processor is viewed as an interconnection of macro blocks
  - E.g. Execution units, register file, etc.
- Power models are built for the macros
  - E.g. Analytical, look-up tables, etc.
- <u>Advantage</u>: allows micro-architecture expl.
- <u>Disadvantage</u>: no black-box for COTS proc.

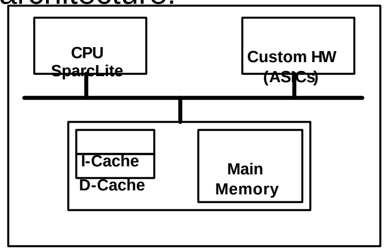
Integrating functional and power models

- Estimating together HW and SW power consumption is more effective than considering the two contributions separately.
- This is because the power consumption of a task mapped onto software is not independent of the implementation of the remaining tasks.
- Two approaches:
  - Non-interacting (trace-based) HW/SW estimation.
  - Concurrent HW/SW estimation.

Non-Interacting HW/SW Power Estimation

Avalanche [LH98]

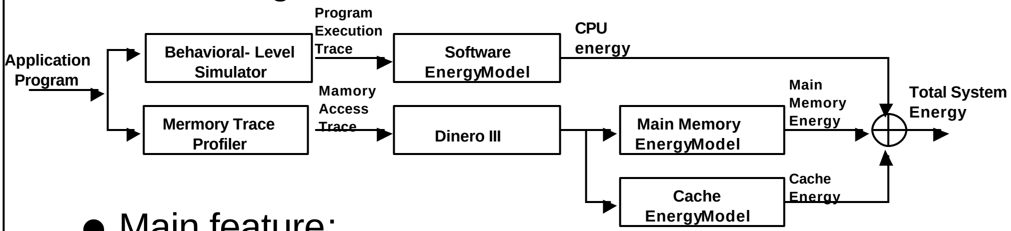
- Target system architecture:



- Power estimation of custom HW done separately (constant power in the model).
- Focus on power dissipation of SW and memory hierarchy.

Trace-based Estimator Architecture

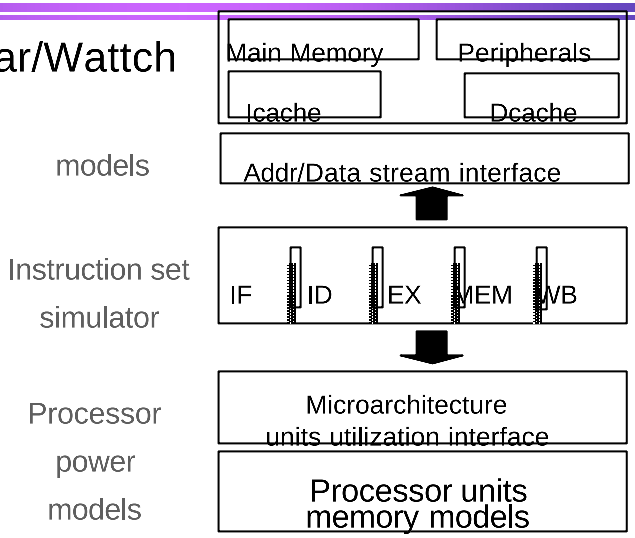
- Block diagram:



- Main feature:
  Exploitation of detailed software, memory, and cache energy models.
- Main limitation:
  No interaction between SW and HW during the estimation.

Concurrent HW/SW Power estimation

E.g.: Simplescalar/Wattch



Outline

- Introduction
- System conceptualization and modeling
  - Modeling and design
  - Energy efficient design from
    - **Executable functional models**
    - Non-executable functional models
- System design
- System management
- Conclusions

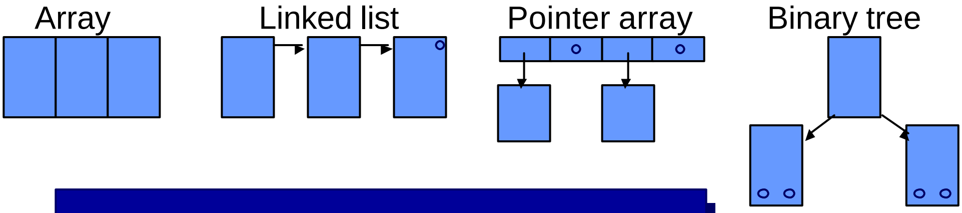
Algorithm selection

- Inputs
  - A target macro-architecture
  - Abstract functional/executable spec.
  - Constraints
  - Library of algorithms
- Objective

Select the most energy-efficient algorithm that satisfies constraints

Example: set data types [Wuytack96]

- Select abstract data type (data struct & algorithm)
- Minimize memory power
- Application: ATM segment protocol processor
- Library with 32 complex abstract data types



×1000 ΔP between best and worst

Issues in algorithm selection

- Applicable only to general-purpose primitives with many alternative implementation
- Pre-characterization on target architecture
- Limited search space exploration

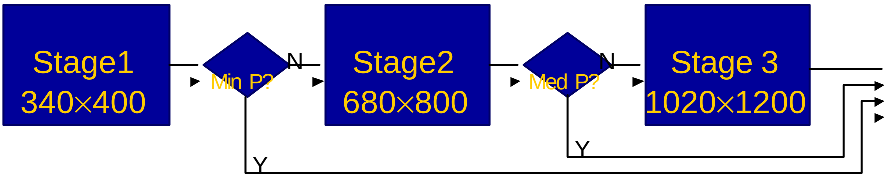
Power Conscious Algorithm Design

- Change the semantic of computation
- Hard to automate
- Very effective

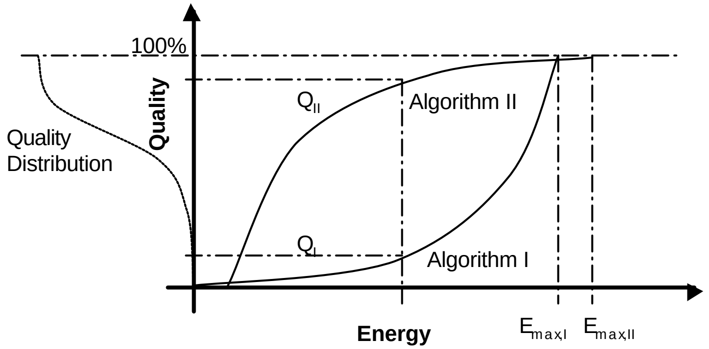
Approximate processing

Introducing well-controlled errors can be advantageous for power

- Reduced data width (coarse discretization)
- Layered algorithms (successive approximations)
- Lossy communication



Energy Scalable Algorithms



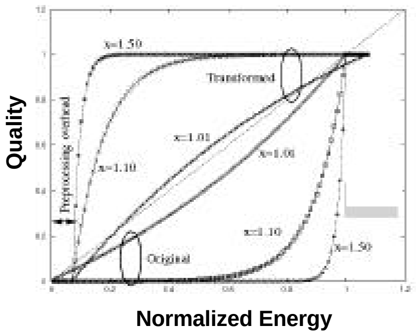
- Maximize quality for given energy availability
- Energy Quality (*E*-*Q*) graph maximally concave

Series Expansion

$$y = f(x) = 1 + k_1 x + k_2 x^2 + \cdots + k_N x^N$$

Original	Scalable

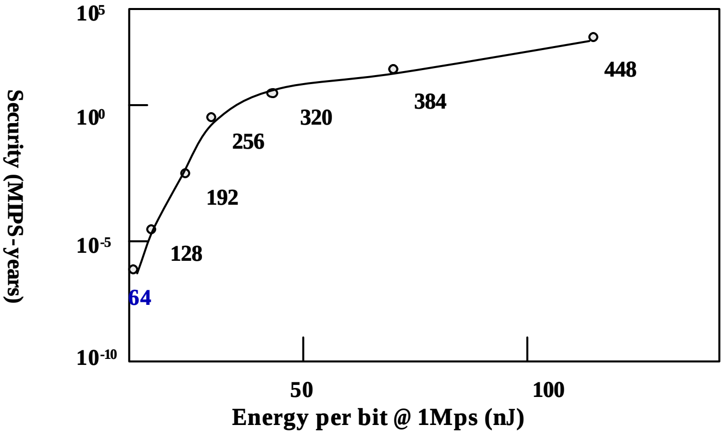
`xpowi = 0.0;`
`y = 1.0;`
`for(i=1; i<N; i++) {`
` xpowi *= x;`
` y += xpowi*k[i];`
`}`
	`if(x > 1.0) {`
	` xpowi = pow(x,N);`
	` y = k[N]*xpowi+1;`
	` for(i=N-1; i>0; i--) {`
	`  xpowi /= x;`
	`  y += xpowi*k[i]; }`
	`}`
	`else { //original algo }`



- Incremental refinement
- Most-significant-first approach

Encryption

- Scalable encryption [Chandrakasan 98]



P=X.Y mod Q

$w = \log_2 Q$

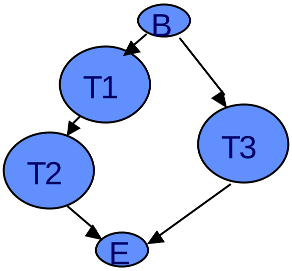
Quality scales with w

Outline

- Introduction
- System conceptualization and modeling
  - Modeling and design
  - Energy efficient design from
    - Executable functional models
    - **Non-executable functional models**
    - Implementation models
- System design
- System management
- Conclusions

Task graph

- Nodes are tasks
- Edges are dependencies
- Periodic execution is implicitly assumed



Task graph techniques

- Problem formulation
  - Input
    - Task graph
    - Set of available processing elements (PEs)
    - Power, performance, cost metrics
    - Performance & cost constraints
  - Output
    - Power-optimized implementation (constrained)
- [Dave99], [Kirovski97]

Processing elements

- Several classes of PEs
  - General-purpose processors (e.g. RISC core)
  - Digital signal processors (e.g. VLIW core)
  - Programmable logic (e.g. LUT-based FPGA)
  - Specialized processors (e.g. custom DCT core)
- Trade off flexibility vs. efficiency
  - Specialized is faster and power-efficient
  - General-purpose is flexible and inexpensive

Cost metrics

- Multi-objective problems
  - Constrained optimization
  - Explore design trade-offs
- Example: performance and power

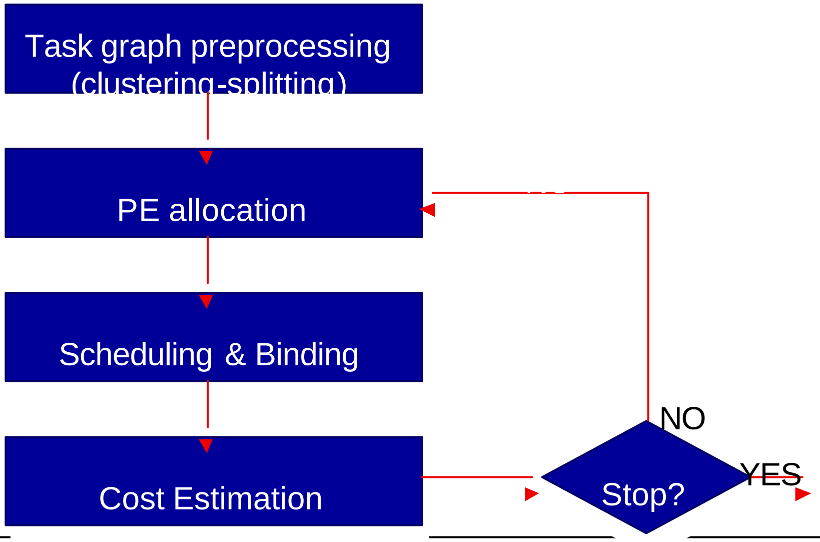
	#Clock Cycles			Energy (µJ)		
	PE1	PE 2	PE 3	PE 1	PE 2	PE 3
	Energy per Cycle			1.5	.2	.05
T1	100	250	900	150	50	45
T2	110	260	900	165	52	45

T_{CLK} = 10ns
@ Vdd = 3.3V

Constrained optimization

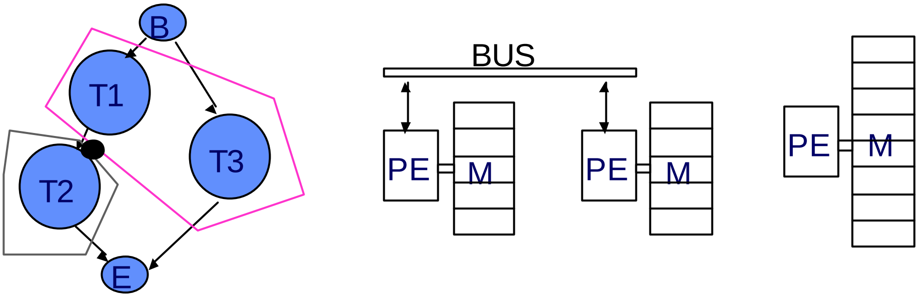
- Design space
  - Who does what and when (binding & scheduling)
  - Supply voltage of the various PEs:
    - $T_{CLK} = K V_{dd}/(V_{dd} - V_t)^2$
- Design target
  - Minimize power
  - Performance constraint (e.g. $T_{iteration}$=21µsec)

Basic search algorithm



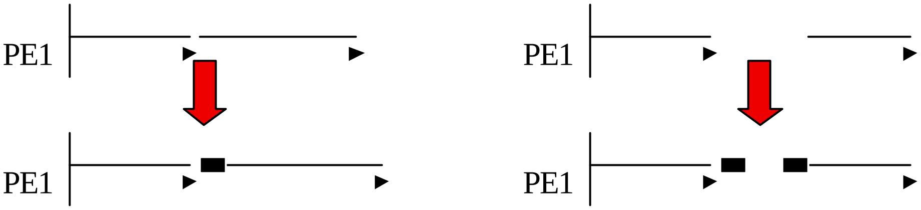
Refined Power Metrics I

- Communication power
- Memory power



Refined Power Metrics II

- Multi-tasking overhead
- Power management overhead



Limitations of Task-Level Abstraction

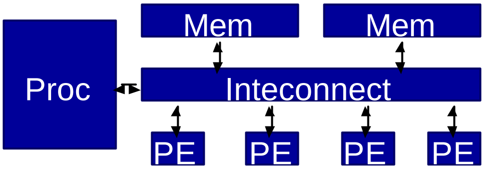
- Task-level description does not express complete functional information
  - Tasks must be defined a priori
  - Functional transformations are impossible
- Power metric characterization
  - Exaustive (every PE for every task)
  - Inaccurate (misses inter-task effects)

Outline

- Introduction
- System conceptualization and modeling
- System design
  - Computation
  - Memory
  - Communication
  - Software
- System Management
- Conclusions

System design

- Input
  - The output of the conceptualization phase
    - A macro-architectural template
    - A hardware-software partition
    - Component by component constraints
- Output
  - Complete hardware design



Mem

Mem

Proc

Inteconnect

PE

PE

PE

PE

Design process

- Specify computation, storage, template components, and software
  - Synergic process
- Fundamental tradeoff: general-purpose vs. application-specific
  - Flexibility has a cost in terms of power

Processing element design

- Application specific processing unit
  - Minimum flexibility, minimum power
- Application-specific processor
  - Tailored processor template
- Core processor
  - Maximum flexibility, maximum power

Application-specific computational units

- Designed at the circuit/logic/RT level
  - Outside the scope of this tutorial
- Synthesized from high-level executable specification (behavioral synthesis)
  - Supply voltage reduction
  - Switching frequency reduction
  - Load capacitance reduction
  - Minimization of switching activity

Power-driven voltage scaling

From *faster* to *power efficient* by scaling down voltage supply

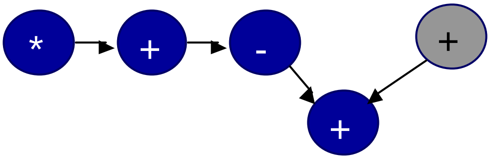
- Traditional speed-enhancing transformations can be exploited for low-power design
  - Pipelining
  - Parallelization
  - Loop unrolling
  - Re-timing

Issues

- Performance-enhancing transformations do not always pay off
  - Region of diminishing returns (e.g. speculation)
- Voltage supply is driven low by technological reasons
  - Reduced headroom

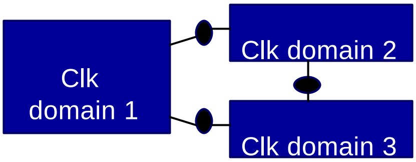
Advanced voltage scaling

- Multiple voltages
  - Slow down non-critical path with lower voltage supply
  - Two or more power grids
  - High-efficiency voltage converters



Clock frequency reduction

- $f_{clk}\downarrow$ does not decrease energy
  - …but it may increase battery life: $C=K/I^{\alpha}$
- Multi-frequency clocks
  - GALS [Hemani99]
  - Low-frequency distribution [Chung95]



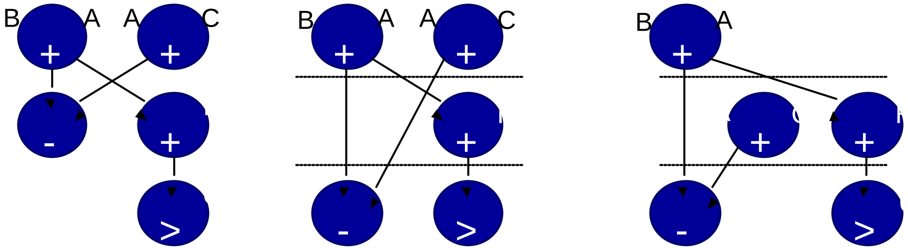
Reducing load capacitance

- Reduce wiring capacitance
  - Reduce local loads
  - Reduce global interconnect

Global interconnect can be reduced by improving spatial locality: trade off communication for computation

Reduce switching activity

- Improve correlation between consecutive input to functional macros
- Reduced glitching
- All basic HLS steps have been modified
  - A synergic approach lead best results



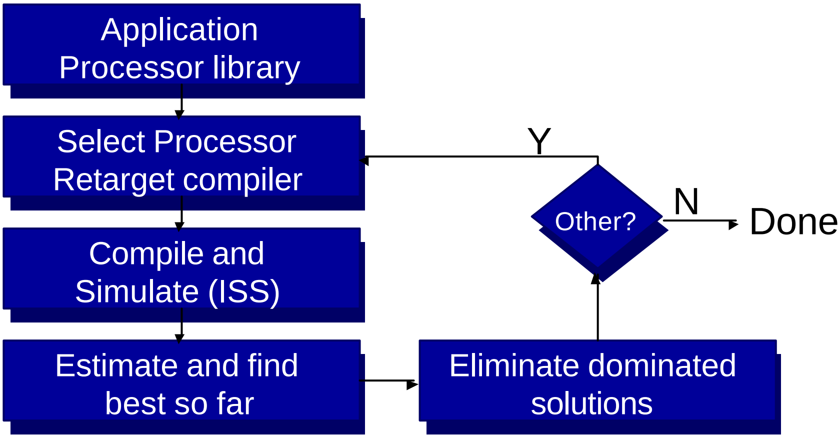
Issues

- High-level estimation
  - Accuracy is still limited
  - Dependency on input patterns
- Design flow
  - HLS is not yet mainstream technology
  - Compete with RTL techniques

Application-specific processors

- Parameterized processors tailored to a specific application
  - Optimally exploit parallelism
  - Eliminate unneeded features
- Applied to different architectures
  - Single-issue cores ⇒ instruction subsetting
  - Superscalar cores ⇒ # and type of FUs
  - VLIW cores ⇒ FUs and compiler

Example: Application-specific VLIW optimization

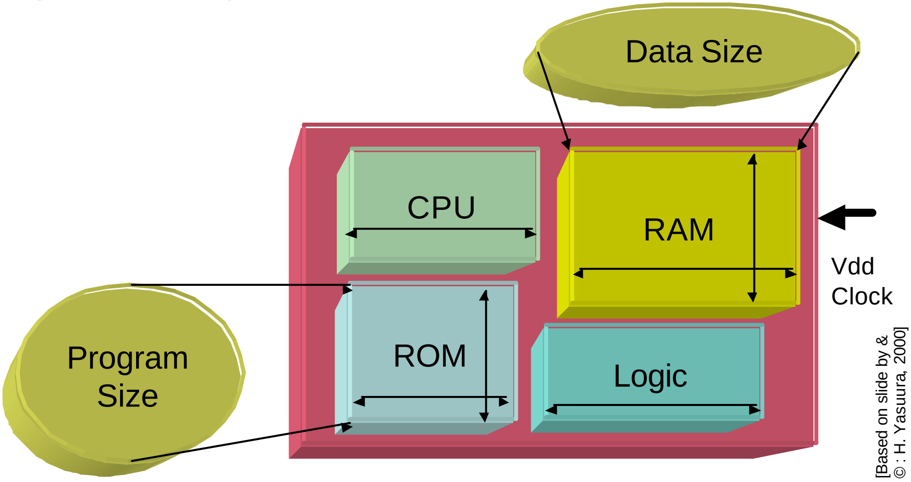


Issues

- Exploration techniques
  - Limited search space
  - Accuracy of cost metrics
- Back-end
  - Synthesis of ASIPS
  - Competitive with highly-optimized cores?
- Preliminary research results

Exploiting processor reconfigurability

Example: reconfigurable bit width



[Based on slide by & © : H. Yasuura, 2000]

Bung DLX

- Standard 32-bit design of the DLX RISC Architecture

# of general registers:	32
# of instructions:	72
the datapath width	32 bits
the instruction length	≤ 32 bits
VHDL Description	~ 7,000 lines
Synthesized circuit	23,282 gates

- ASIPs defined through design modification table containing:

The datapath width	≤ up to 64
The data memory space	≤ 2^{32} words
The instruction length	≤ 32 bits
The instruction memory space	≤ 2^{32} words
The number of general registers	≤ 32
The number of instructions	≤ 72

[Based on slide by & © : H. Yasuura, 2000]

Key software elements: Valen-C and a Retargetable Compiler

- Valen-C
  - Programmers can specify the effective bit width for each variable: e.g.: int20 x, y, z
  - The semantics of the program is independent from processor architecture.
- Retargetable compiler
  - Processor Definition + Valen-C Program
  - ➪ Assembly code for the processor

[Data are available at
http://kasuga.csce.kyushu-u.ac.jp/~codesign/Valen-C/index-j.html
– Source code, documentation on Valen-C compiler]

[Based on slide by &
© : H. Yasuura, 2000]

How does compilation with Valen-C work?

Valen-C code

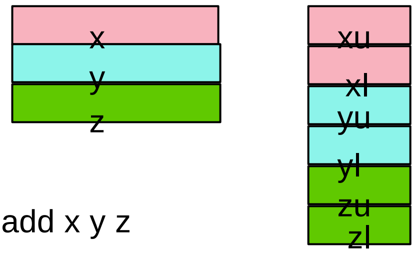
Int20 x, y, z;

....

z = x + y;

20-bit Processor

10-bit Processor



add x y z

add xl yl zl
addc xu yu zu

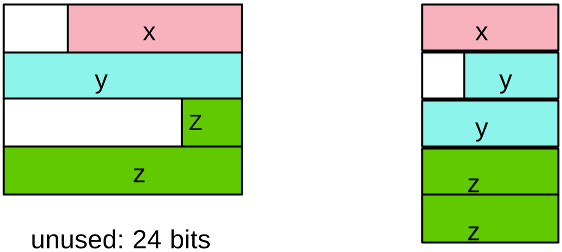
[Based on slide by &
© : H. Yasuura, 2000]

More complicated cases

Valen-C Program
int12 x;
int20 y;
int24 z;

20-bit processor

12-bit processor



unused: 24 bits
total: 80 bits

unused: 4 bits
total: 60 bits

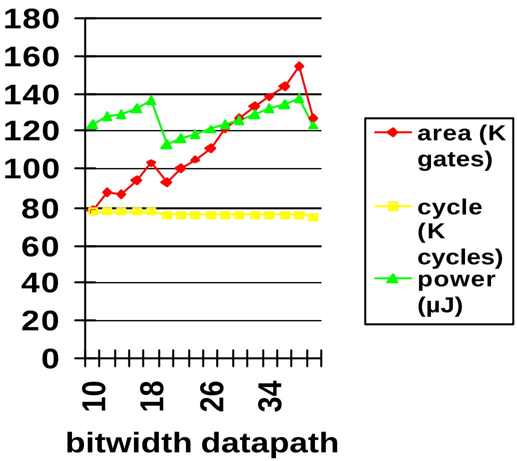
□ unused bits

[Based on slide by &
© : H. Yasuura, 2000]

Application: Decimal 12 bit Calculator

- Valen-C (400 lines)

The length of Variables	# of Variables
4	257
8	257
14	3
39	258

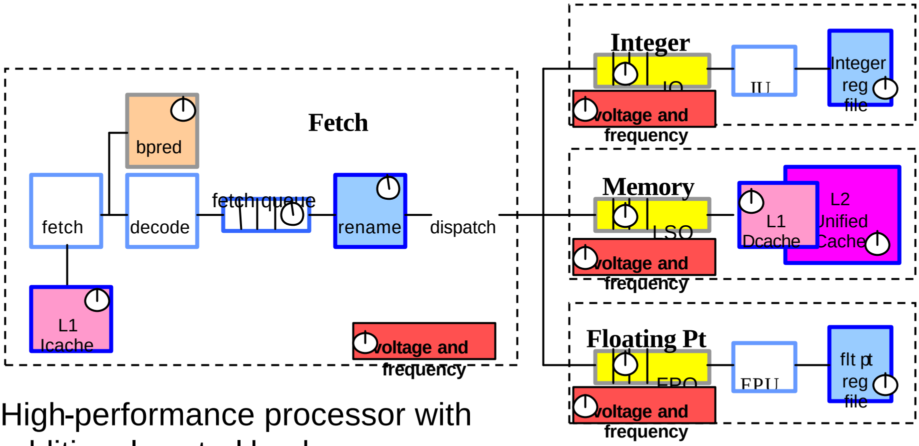


[Based on slide by &
© : H. Yasuura, 2000]

Dynamically Tunable Microprocessors

- Dynamically tailor the hardware to meet program needs *on-the-fly* while the program runs
  - Fine grain level: dynamically resize caches, TLBs, issue queues, register files, etc.
  - Exploits logic shutdown (clock-gatinc etc.)
- Two parts:
  - Dynamically configurable hardware (“knobs”)
  - Feedback and control mechanisms (“tuning”)

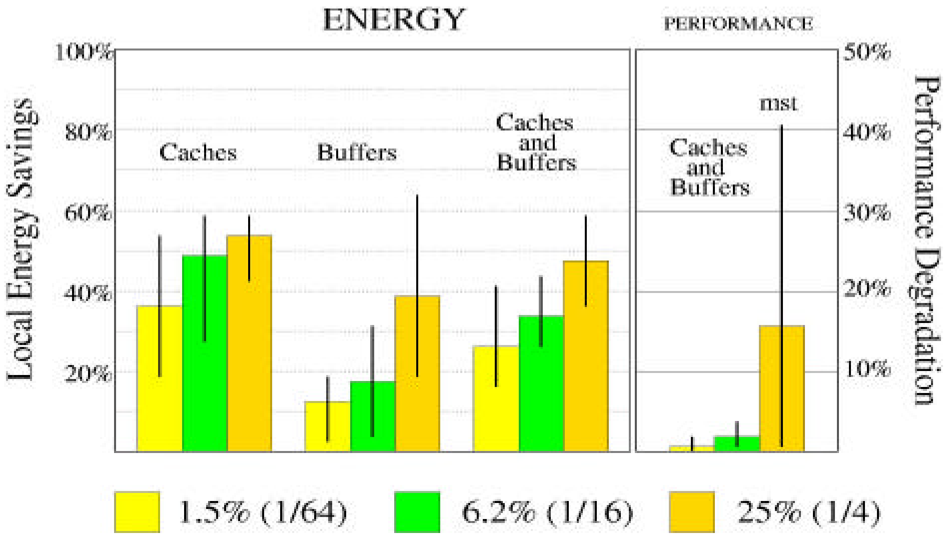
Dynamically Configurable Architecture



High-performance processor with additional control knobs

[Albornesi 02]

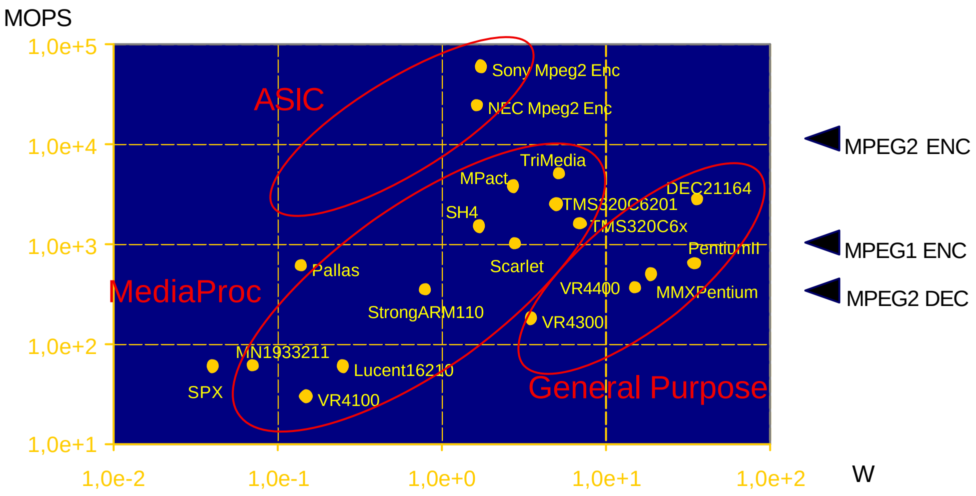
Energy Savings and Performance Cost



Low power core processors

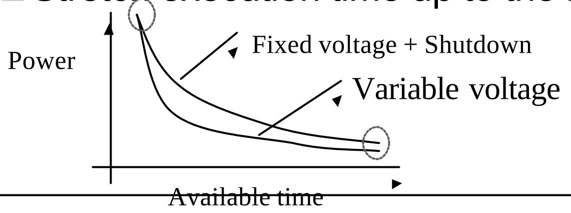
- Details are outside the tutorial’s scope
  - [Gonzalez96,Burd96]
- Key ideas
  - Low voltage
  - Reduce wasted switching
  - Specialized modes of operations/instructions
  - Variable voltage supply

Core design space for Multimedia [Nishitani99]

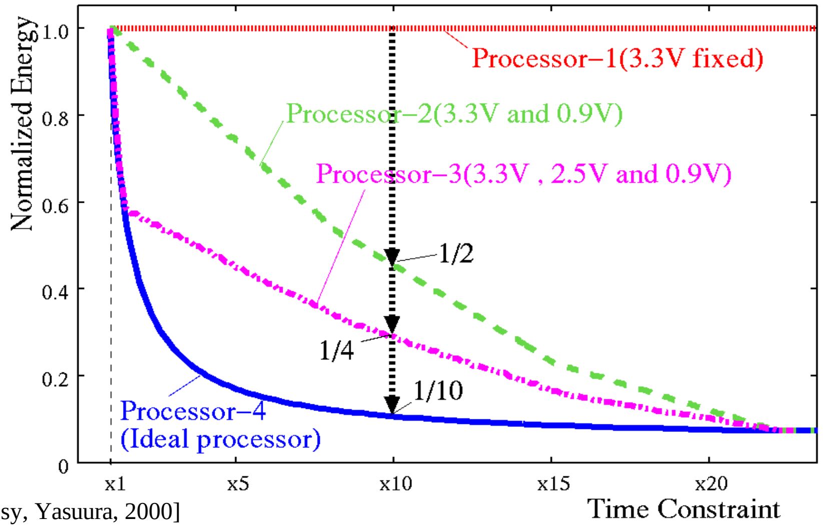


Exploiting Variable Supply

- Supply voltage can be dynamically changed during system operation
  - Cubic power savings
  - Circuit slowdown
- Just-in-time computation
  - Stretch execution time up to the max tolerable



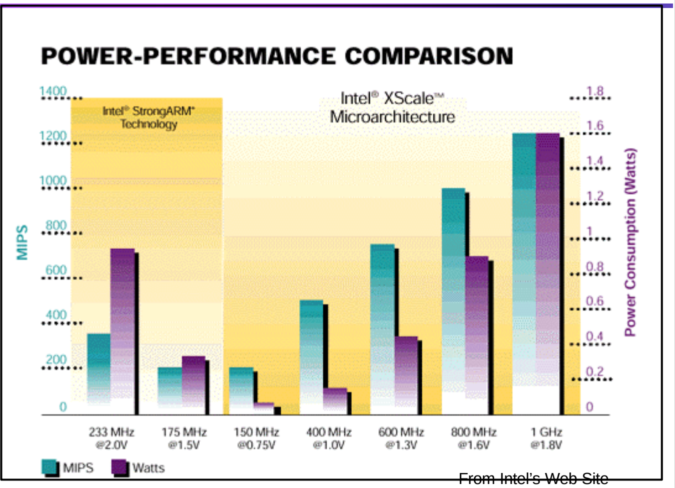
Voltage scaling example



[Courtesy, Yasuura, 2000]

Variable-voltage processor example: INTEL Xscale

- Discrete VS
  - 3 to 4 voltages
  - More frequencies
- Transition penalties
  - ≈ milliseconds
  - Dominated by supply voltage transient
- System support
  - Voltage supply circuitry
  - Interface circuits (!!)
- Voltage ranges
  - Decrease with tech.

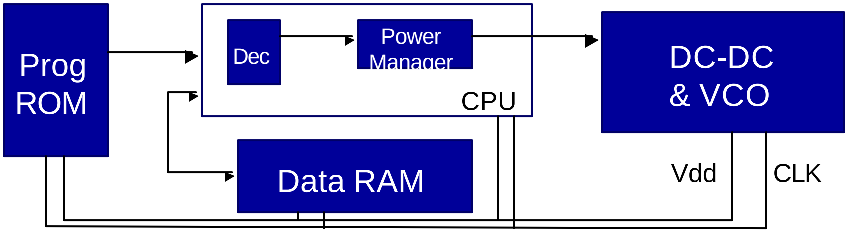


From Intel's Web Site

[INTEL01]

Variable-supply Architectures

- High-efficiency adjustable DC-DC converter
- Adjustable synchronization
  - Variable-frequency clock generator [Chandrakasan96]
  - Self-timed circuits [Nielsen94]
- Example: Power-pro architecture [Ishiara98], Crusoe embedded processor [Transmeta00]



Issues

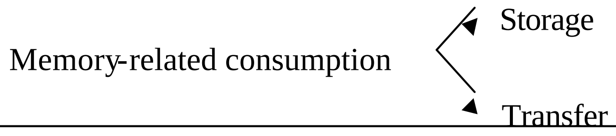
- Optimization still not proven on real-life architectures
- Overhead in supporting variable voltage
  - Adjustable DC-DC
  - Adjustable clock
  - Interfaces
- Reliability concerns

Outline

- Introduction
- System conceptualization and modeling
- System design
  - Computation
  - Memory
  - Communication
  - Software
- System Management
- Conclusions

Memory Optimization

- Custom data processors
  - Computation is less critical than data storage (for data-dominated applications)
- General-purpose processors
  - A significant fraction of system power is consumed by memories



Memory-related consumption

Storage

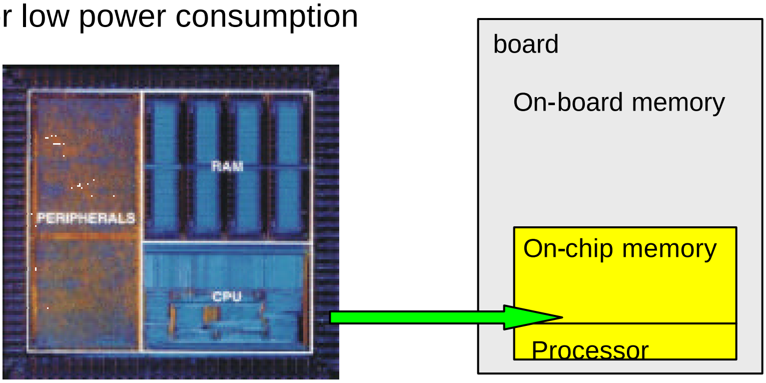
Transfer

Off-chip vs. on-chip memories

Larger & off-chip memories need more energy than smaller & on-chip memories; Example:

ARM7TDMI cores, well-known for low power consumption

ARM Atmel Evaluation Board



board

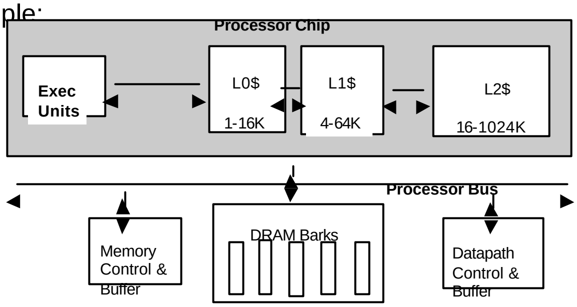
On-board memory

On-chip memory

Processor

Minimization of Memory Access Power

- Basic concept: “Close” vs. “far” memory accesses:
  - Close: Faster, less energy consuming, smaller block sizes.
  - Far: Slower, more energy consuming, larger block sizes.
  - Example:



Processor Chip

Exec Units

L0$ 1-16K

L1$ 4-64K

L2$ 16-1024K

Processor Bus

Memory Control & Buffer

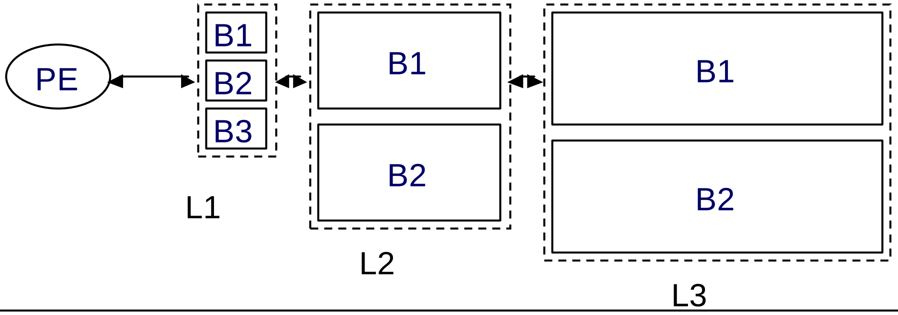
DRAM Barks

Datapath Control & Buffer

E_{L0}= 1.5nJ E_{L1}= 3nJ E_{L2}= 7nJ E_{RAM}= 127nJ

Memory Power Optimization

- Key idea: exploit locality
  - Hierarchical memory
  - Partitioned memory
- Optimize software for power-efficient memory architectures



PE

B1
B2
B3

L1

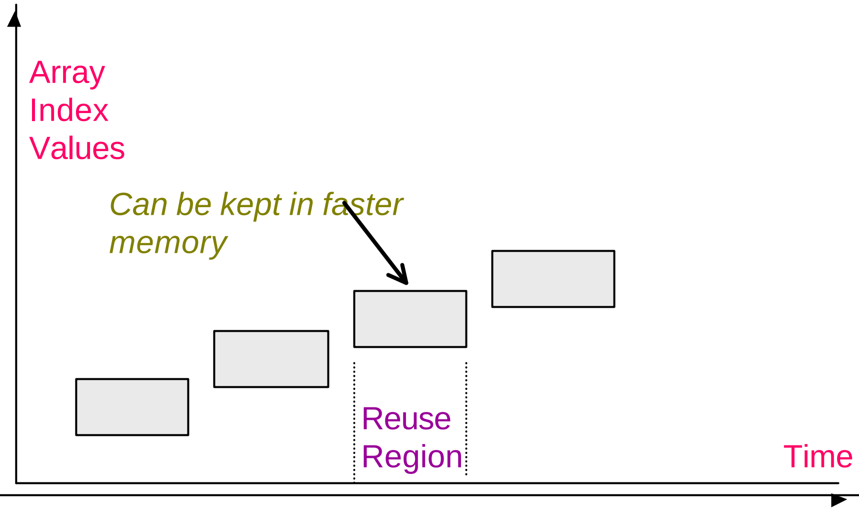
B1
B2

L2

B1
B2

L3

Exploiting Temporal Locality



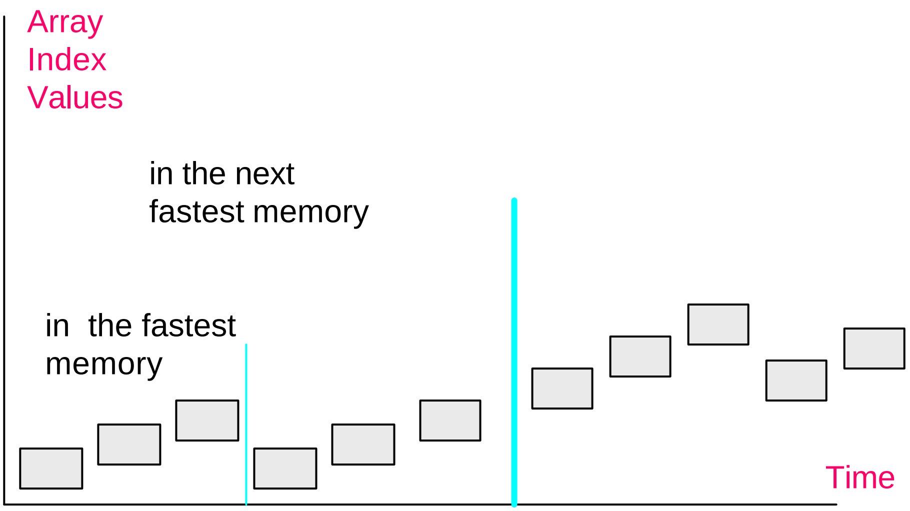
Array
Index
Values

Can be kept in faster memory

Reuse
Region

Time

Exploiting Temporal Locality (Multiple Levels)



Array Index Values

in the next fastest memory

in the fastest memory

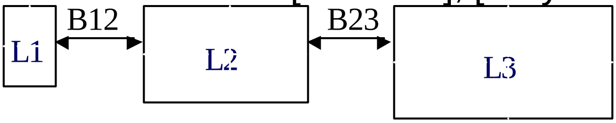
Time

Optimization approaches

- Fixed memory access patterns
  - Optimize memory architecture
- Fixed memory architecture
  - Optimize memory access patterns
- Concurrently optimize memory architecture and accesses

Optimize Memory Architecture

- Data replication to localize accesses
  - Implicit: multi-level caches [Su95], [Bahar98]
  - Explicit: buffers [Bajwa97], [Wuytack98]
- Partitioning to minimize cost per access
  - Multi-bank caches [Ko98]
  - Partitioned memories [Tellez97], [Wuytack98]



$$P_{mem} = P_{L1}\cdot hit_{L1}+P_{B12}(1-hit_{L1})+P_{L2}\cdot hit_{L1}+(P_{B23}+P_{L3})\cdot(1-hit_{L1}-hit_{L2})$$

Optimize Memory Accesses

- Sequentialize memory accesses
  - Reduce address bus transitions [Catthoor94], [Su95]
  - Exploit multiple small memories [Panda96]
- Localize program execution
  - Fit frequently executed code into a small instruction buffer (or cache) [Panwar95], [Bellas98]
- Reduce storage requirements [Gebotys96], [Catthoor]

Optimize Memory Architecture and Access Patterns

- Two phase-process
  - Specification (program) transformations
    - Reduce memory requirements
    - Improve regularity of accesses
  - Build optimized memory architecture
- Highest potential
  - How to automate program transformations

Memory Hierarchy Optimization

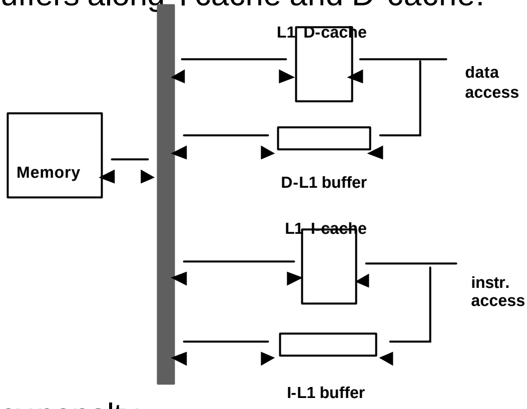
- Idea: Enforce locality in the cache and memory sub-systems.
- Solutions:
  - Data replication.
  - Alternatives to caches (e.g., scratch-pad buffers).
  - Cache/Memory partitioning.

Implicit Data Replication

- Usage of a *filter* cache:
  - Introduce an extra L0 cache.
  - Much smaller (e.g., 256 byte).
  - Latency penalty due to L0 misses compensated by low-energy hits in L0 cache.
  - Energy/delay product is reduced.

Explicit Data Replication

- Exploit buffers along I-cache and D-cache:



- No latency penalty.

Explicit Data Replication (Cont.)

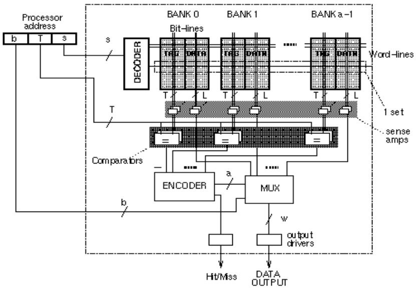
- Use of buffers as victim cache:
  - Accessed on a main cache miss.
  - Hit:
    - Datum is promoted to main cache (and returned to CPU).
    - The replaced line in the cache is moved to the victim cache.
  - Miss:
    - L2 cache is accessed.
    - The incoming datum is put in the main cache.
    - The replaced line in the cache is moved to the victim cache.

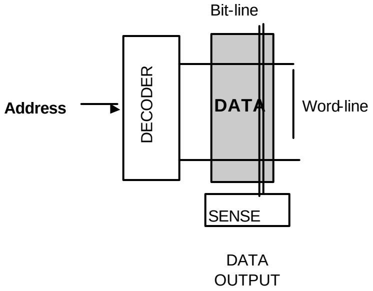
Explicit Data Replication (Cont.)

- Use of *speculative* buffers:
  - Every cache access is marked with a "*confidence* level", obtained by examining processor state (i.e., current branch prediction state).
  - The main cache contains *misses* with high confidence level.
  - The speculative buffer contains *misses* with low confidence level.

Replace Caches with Scratch-Pad Buffers

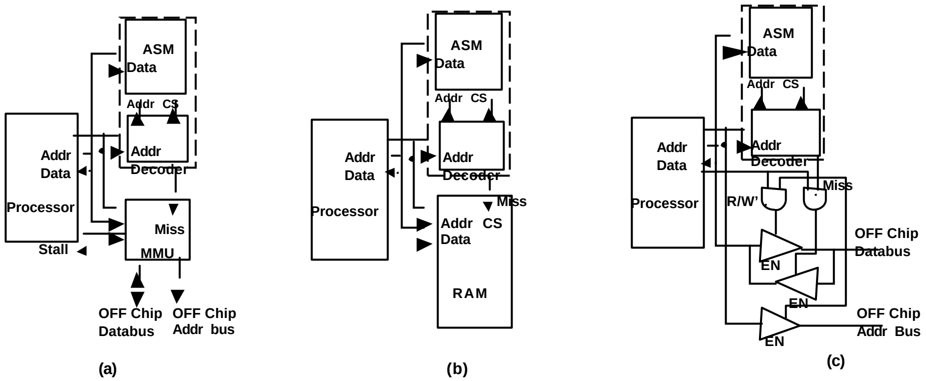
- Viable solution for embedded systems, where memory access profiles may be available.
- Trade-off cache flexibility for lower access cost.





Replace Caches with Scratch-Pad Buffers (Cont.)

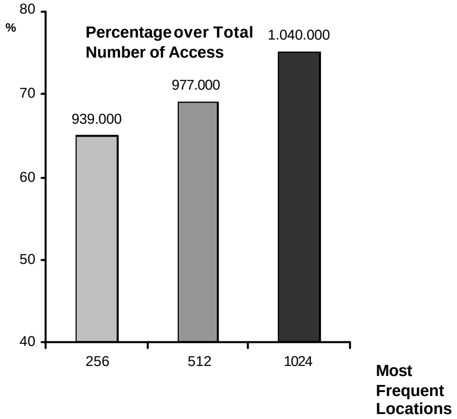
(a) ASM used as a cache.
(b) ASM used as a buffer for on-chip memory (no latency penalty).
(c) ASM used as a buffer for off-chip memory (no latency penalty).



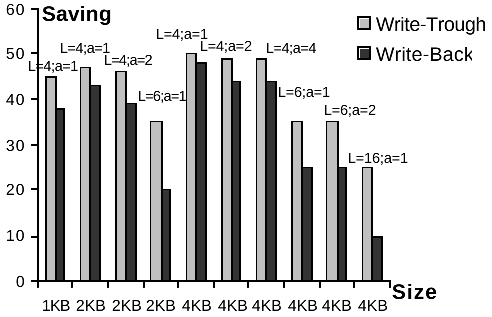
(a) (b) (c)

Replace Caches with Scratch-Pad Buffers (Cont.)

- Results for MP3 decoder:



Profiling Results



Energy Savings

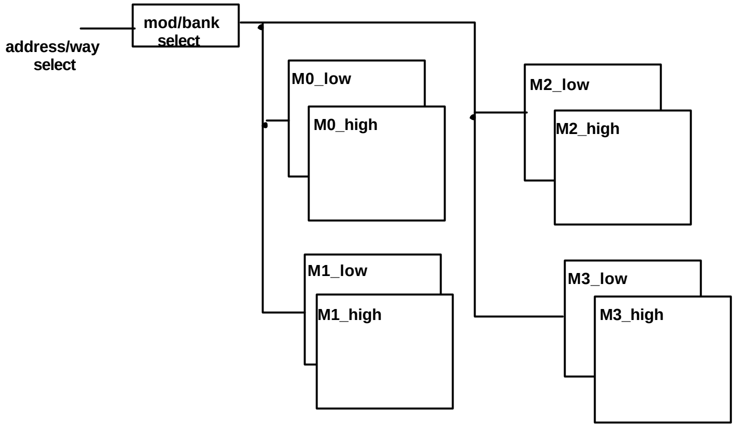
Cache/Memory Partitioning

Multi-bank caches:

- Use independently-addressable banks.
- Two-dimensional partitioning: M modules with B banks each.
- Power savings achieved through exploitation of reduced capacitance of smaller memories.
- Ad-hoc, low-power bank selection circuitry is used.

Cache/Memory Partitioning (Cont.)

- Example of multi-bank caches (M=4, B=2):



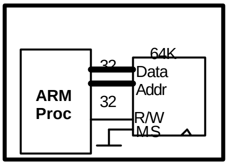
Cache/Memory Partitioning (Cont.)

Partitioned memories:

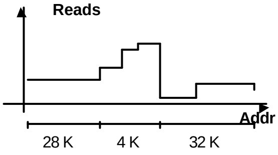
- Memory hierarchy with independently-addressable memory banks.
- Exploit sleep-mode features to shut down individual banks.
- Design memory partition so as to maximize the *sleep-time*.
- Typical memory traces are used to drive the partitioning process.

Cache/Memory Partitioning (Cont.)

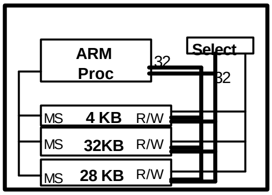
- In the case of embedded systems, the dynamic memory access profile may be available.
- Idea:
  Map most frequent addresses onto small partitions close to the processor.
- Example:



TRADITIONAL ARCHITECTURE



Dynamic Access Profile



Power Optimized Architecture

Cache/Memory Partitioning (Cont.)

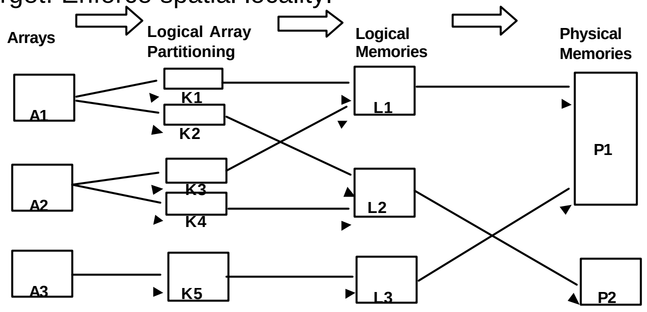
- Assumptions:
  - Energy per access monotonically increases with memory size.
- Target: Automatic memory partitioning.
- Need of:
  - Cost metrics.
  - Optimization algorithm.
- The energy savings obtained by partitioning must compensate the overhead of adding banks (longer wires, bank selection logic).
- Link to physical design is key for overhead characterization.

Memory Access Pattern Optimization

- Address sequentialization:
  - Exploitation of multiple (smaller) memories.
  - Low-transition bus encoding can also be viewed as a tool for making addresses sequential (e.g., Gray-code address generation).
- Localization of execution:
  - Ad-hoc memory (or cache) for storing frequently executed code [BHPS98].

Exploiting Multiple Memories

- Mapping of arrays onto multiple physical memories:
  - Logical memory partitions are allocated according to some optimal array organization (e.g., tile-based vs. row-major).
  - Target: Enforce spatial locality.



Arrays

Logical Array Partitioning

Logical Memories

Physical Memories

A1

A2

A3

K1

K2

K3

K4

K5

L1

L2

L3

P1

P2

Code Density Optimization

- Basic idea:
  Minimize program memory occupation so as to reduce the bandwidth of processor-memory communication.
- Approaches:
  - Custom instruction sets.
  - Object code compression.
- Privilege memory traffic reduction (i.e., dynamic code size) over static code size reduction:
  - Sometimes static code size may even increase.

Custom Instruction Sets

- Viable solution for general-purpose processors.
- Example: ARM Thumb code.
  - Interleaving of regular (32 bit) and Thumb (16 bit) instructions.
  - Requires modifications to the basic processor architecture.
  - Requires specific compilers and software development kits.

Object Code Compression

- Viable solution for embedded processors.
- Idea:

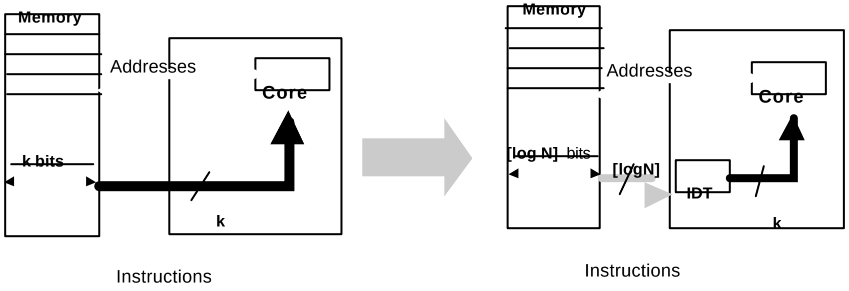
Exploit the small subset of instructions used by firmware code running on embedded processors.

- Approaches:
  - Full code compression.
  - Selective code compression.

Full Code Compression

Replace all instructions with binary patterns of minimum width.

- ([$\log_2$ N] , where N is the number of instructions).
- Architecture:



IDT= Instr. Decompr. Table

Full Code Compression (Cont.)

- Advantages:
  - Availability of ad-hoc source-code compilers is not required (replace original instructions with compressed ones with script).
  - Architectural modifications to the processor are not required (key feature for users of IP cores and µC).
- Limitations:
  - Very often the number of distinct instructions, *N*, used by a program is not small. This implies:
    - Size of IDT may be very large.
    - Original and compressed instruction widths may be comparable.
    - $[\log_2 N]$ may not be a multiple of 8.

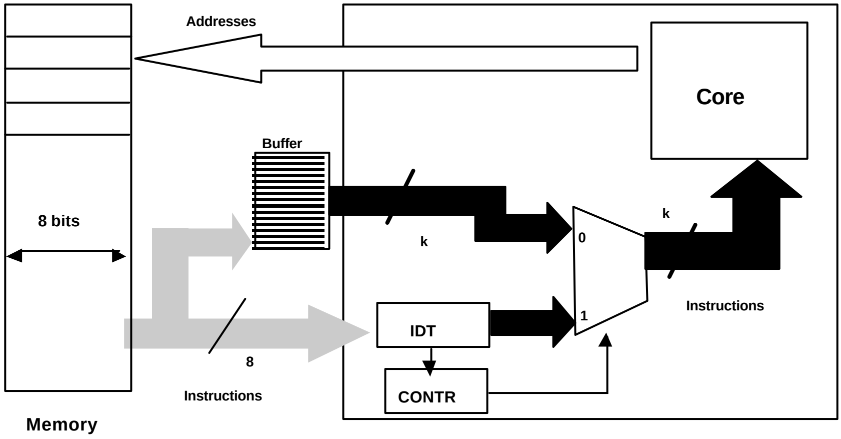
Selective Code Compression

Very often program traces are covered by a small subset of instructions.

- Consider for compression only such subset.
- Candidates: Instructions that maximize program coverage.
- IDT sf fixed (256 words).
- Program is a mix of compressed and uncompressed instructions.

Selective Code Compression (Cont.)

- Architecture:



Selective Code Compression (Cont.)

- Assumptions:
  - Byte-addressable memory.
  - Memory banks (8-bit wide) can be independently disabled

    (on a cycle-by-cycle basis).
  - A reserved special word: The *mark*

    (used to signal compressed/uncompressed instruction).
- Different use of the mark is possible.

Selective Code Compression (Cont.)

- Various architectures available.
- Example:

0		L1*		
1		L2*		
2		L3*		
3		L4		
4		L5*		
5		L6**		
6		L7		
7		L8*		

0	M	S3	S2	S1
1	L4-4	L4-3	L4-2	L4-1
2	▨	▨	M	S5
3	L6-4	L6-3	L6-2	L6-1
4	▨	▨	▨	M
5	L7-4	L7-3	L7-2	L7-1
6	▨	▨	▨	S8

0	S1
1	S2
2	S3
3	M
4	L4 - 1
5	L4 - 2
6	L4 - 3
7	L4 - 4
8	S5
9	S6
10	M
11	L7 - 1
12	L7 - 2
13	L7 - 3
14	L7 - 4
15	S8

0	▨	▨	▨	S1
1	▨	▨	▨	S2
2	▨	▨	▨	S3
3	▨	▨	▨	M
4	L4-4	L4-3	L4-2	L4-1
5	▨	▨	▨	S5
6	▨	▨	▨	S6
7	▨	▨	▨	M
8	L7-4	L7-3	L7-2	L7-1
9	▨	▨	▨	S8

0	M	S3	S2	S1
1	L4-4	L4-3	L4-2	L4-1
2	▨	S6	M2	S5
3	▨	▨	▨	M1
4	L7-4	L7-3	L7-2	L7-1
5	▨	▨	▨	S8

Selective Code Compression (Cont.)

- Advantages:
  - Size of IDT is fixed *a priori* and limited (we picked *N* = 256).
  - Instruction fetching/decompression logic has reduced complexity.
- Drawback:
  - Requires a controller to handle instruction fetching

    (the program stored in memory is a mix of compressed (many) and uncompressed (few) instructions).
- Average power savings on execution of standard programs around 45%.

Data Density Optimization

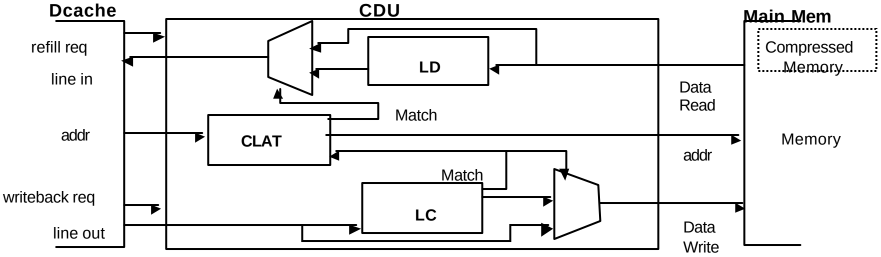
- Same principle as code density optimization.
- Existing approaches based on data compression:
  - Target is memory traffic reduction (dynamic size of the data-set).
  - More complex than code compression, because both compression and decompression are required.
  - Hardware compression/decompression unit (CDU) needed.
    - – Speed vs. power design trade-off.

On-The-Fly Data Compression

- CDU placed between D-Cache and main memory. Data are uncompressed in the D-Cache, possibly compressed in memory:
  - Compression is performed on cache write-backs.
  - Decompression is performed on cache refills.
  - Compression and decompression are performed one cache line at a time.
- A small portion of the main memory is dedicated to store compressed data.

On-The-Fly Data Compression (Cont.)

- Architecture:



- LC: Line compressor (CAM); LD: Line decompressor (RAM); CLAT: Compressed line address table (CAM).
- A cache line is compressed only if it fits a slot in the compressed memory.

On-The-Fly Data Compression (Cont.)

- Profile-driven approach:
  LC and LD are filled once and for all with data profiling information.
  - Memory traffic reductions around 42%.
  - Off-line data profiling needed; applicable to embedded systems.
- Adaptive approach:
  Requires two LC CAMs and two LD RAMs; while the first pair CAM-RAM is in use, the second pair is updated with current data statistics. When “mature”, the two pairs are swapped.
  - Memory traffic reductions around 30%.
  - No data profiling needed; applicable to general-purpose systems.

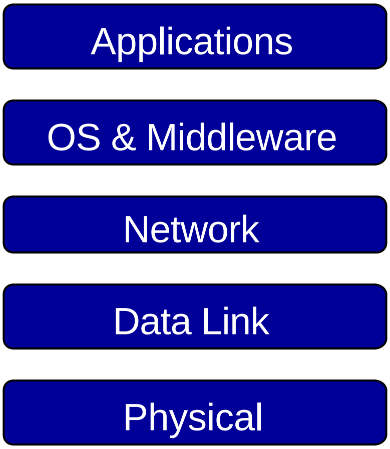
Outline

- Introduction
- System conceptualization and modeling
- System design
  - Computation
  - Memory
  - Communication
  - Software
- System Management
- Conclusions

Design of communication units

- Trends:
  - Faster computation blocks, larger chips
  - Communication speed is critical
  - Energy cost of communication is significant
- Multifaceted design approach:
  - On chip, networks, wireless, ...
  - Protocol stack

Protocol stack
a simplified view



Applications

OS & Middleware

Network

Data Link

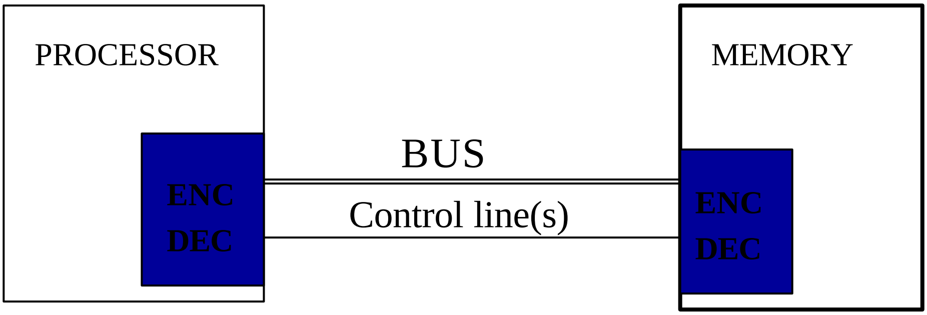
Physical

- Data Link
  - Error control through coding and channel management
  - Shared busses
- Physical layer
  - Signaling
  - Modulation

Data encoding

- Theoretical results:
  - Bounds on transition activity reduction:
    - The higher the entropy rate of the source is, the lower is the gain achievable by coding
- Practical applications:
  - Processor-memory (and other) busses
    - Data busses, address busses
- Transition activity reduction does not guarantee energy savings

Bus encoding



Bus encoding

- Data buses:
  - Random white noise model
- Address busses:
  - Some spatio-temporal correlations
- Embedded software:
  - Addresses and data can be analyzed *a priori* to determine encoding

Bus-Invert coding for data busses

- Add redundant line INV to bus
- When INV=0
  - Data is equal to remaining bus lines
- When INV = 1
  - Data is complement of remaining bus lines
- Performance:
  - Peak: at most n/2 bus lines switch
  - Average: Code is optimal. No other code with 1-bit redundancy can do better

Bus-Invert coding for data busses

- Average switching reduction is bus-width dependent:
  - Ex: 3.27 for an 8-bit bus
- Average switching per line decreases as busses get wider
  - Use partitioned codes
  - No longer optimal (among redundant codes)
- Implementation issues:
  - Difference (XOR) of two data samples and majority vote

Bus-Inver code comparisons

lines	mode	A- trans	A-trans/ line	A-power
2	-	1	0.5	100%
2	BI	0.75	0.375	75%
8	-	4	0.5	100%
8	1 BI	3.27	0.409	81.8%
8	4 BI	3	0.375	75%
16	-	8	0.5	100%
16	1 BI	6.83	0.427	85.4%

Extensions and generalizations

- Transition signaling:
  - Assert logic **TRUE / FALSE** by signal transitions
- Use several redundant lines
  - Limited-weight codes [Stan, Burelson]
  - Coding is space and time
  - Modulation techniques

Encoding instruction addresses

- Most instruction addresses are consecutive
  - Use Gray code [Su, Tsui, Despain]
- Word-oriented machines:
  - Increments by 4 (32bit) or by 8 (64bit).
  - Modify Gray code to switch 1 bit per increment [Metha, Owens, Irwin]
  - Gray code adder for jumps
    - Harder to partition
    - Convert to Gray code after update

Working zone encoding (WZE)

- Conjecture:
  - Software programs favor working zones of their address space
- WZE:
  - Transmit WZ identifier and offset in WZ
  - 1-hot encoding for offsets
- Applicability:
  - No caches: data/instruction/shared address busses
  - With caches: data/instruction-only busses

T0 Code

- Add redundant line INC to bus
- When INC = 0
  - Address is equal to remaining bus lines
- When INC = 1
  - Transmitter freezes other bus lines
  - Receiver increments previously transmitted address by a parameter called stride
- Asymptotically zero transitions for sequences
  - Better than Gray code

Mixed bus encoding techniques

- T0_BI:
  - Use two redundant lines: INC and INV
  - Good for shared address/data busses
- Dual encoding:
  - Good for time-multiplexed address busses
  - Use redundant line SEL :
    - SEL=1 denotes addresses
    - SEL is already present in the bus interface
  - Dual T0:
    - Use T0 code when SEL is asserted.
  - Dual T0_BI:
    - Use T0 when SEL is asserted; otherwise use BI

Address bus encoding using statistical analysis

- Statistical analysis of bus traces
  - Spatio-temporal correlation of word K-tuples
  - Often limited to first / second order statistics (K=1,2)
- Encode words according to correlation
- Use transition signaling
- Spatio-temporal correlation computation:
  - On-line adaptive
  - Off-line for embedded software

Address bus encoding for embedded software

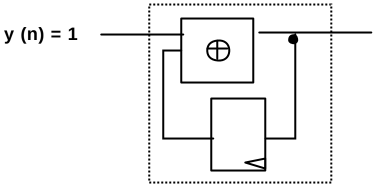
- Off-line statistical analysis of bus traces
  - Compute bit 2nd order correlation from known stream:
    - Correlate bit $\mathbf{i_t}$ with bit $\mathbf{j_{t+1}}$
    - Use correlation measure to group bits into fields
  - Apply graph clustering algorithm
  - Cluster correspond to mutual high spatio-temporal correlation
- Re-encode bus lines in each cluster
  - Group bus lines into clusters (with locally high correlation)
  - Encode signals within each cluster to reduce bus switching

Information-Theoretic Code

- Idea:

  Exploit the concept of *correlator* (widely used in information theory) to simplify the encoding problem.
- New problem formulation:

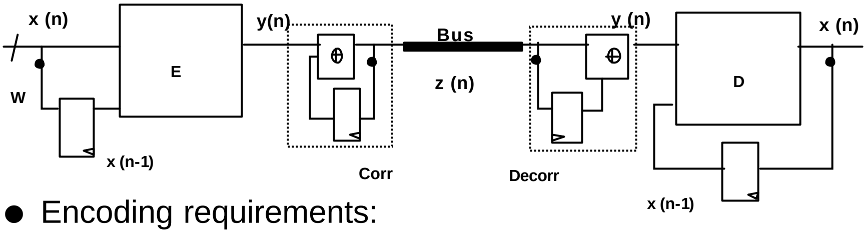
  Minimize word transition probabilities, that is, minimize the number of 1's being transmitted.



Correlator:
Maps ones
to transitions

Information-Theoretic Code (Cont.)

- Generic encoder-decoder (codec) architecture:



- Encoding requirements:

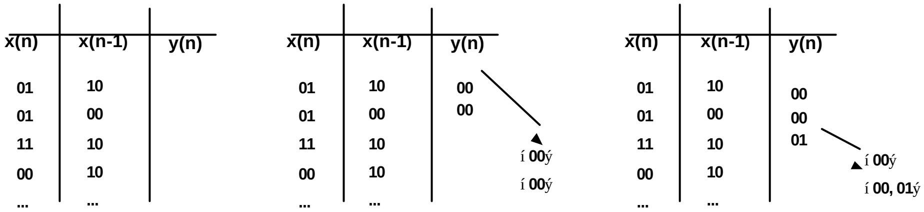
  E should minimize the average number of 1's while guaranteeing unique decodability of *y*(*n*).
- Symmetric operations occur in the decoding phase.

Information-Theoretic Code (Cont.)

- Encoding algorithm:
  - Sort the pairs of input data words according to their probabilities.
  - Starting from the most probable pair:
    - Assign *minimum-one* codes.
    - Update *decodability* constraints.
  - Extract *E* and *D*.
- The probability of input data words is required up-front. Therefore, this approach is applicable in embedded systems.

Information-Theoretic Code (Cont.)

- Example (bus width *W*=2):



x(n)	x(n-1)	y(n)
01	10	
01	00	
11	10	
00	10	
...	...	

x(n)	x(n-1)	y(n)
01	10	00
01	00	00
11	10	
00	10	
...	...	

x(n)	x(n-1)	y(n)
01	10	00
01	00	00
11	10	01
00	10	
...	...	

- The algorithm provides optimal results, but it is impractical in both size and time.
- Approximate solutions are required:
  - Clustered Encoding.
  - Discretized Encoding.

Bus encoding: summary

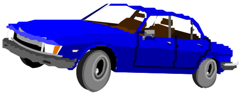
- Bus encoding is very useful to reduce switching of high-capacitance busses
- Some techniques require synthesis of dedicated encoder/decoder circuitry
- Power consumption of such circuits must be weighted against power savings on busses
- Techniques differ for address and data busses

Outline

- Introduction
- System conceptualization and modeling
- System design
  - Computation
  - Memory
  - Communication
  - Software
- System Management
- Conclusions

Views on embedded software

... it is now common knowledge that more than 70% of the development cost for complex systems such as automotive electronics and communication systems are due to software development [A. Sangiovanni-Vincentelli, 1999]



For many products in the area of consumer electronics the amount of code is **doubling every two years** [Fritz Vaandrager in: Rozenberg, Vaandrager (eds.): Lectures on Embedded Systems, LNCS, Vol. 1494, 1998]

Optimization for low-energy always the same as optimization 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**

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

←

```
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

→

```
ADD r3,r0,r2
MOV r0,#28
MOV r2,r12
MOV r12,r11
MOV r11,rr10
MOV r0,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
```

2231 cycles
16.47 µJ

Impact of software

- For a given a hardware platform, the energy to realize a function depends on software
  - Operating system
  - Different algorithms to embody a function (e.g., sorting)
  - Different coding styles
  - Application software compilation

Outline

- Introduction
- System conceptualization and modeling
- System Design
  - Software design
    - Compilation techniques (memory hierarchy)
    - High-level transformations
    - Dynamic power management
- Conclusions

Reducing Memory Area

```
For I = 1, N
  …  = C[I]
For I = 1, N
  B[I] = A[I]

For I = 1, N
  …  = C[I]
For I = 1, N
  C[I] = A[I]
```

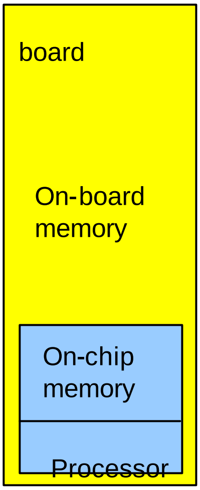
Reusing the same memory space

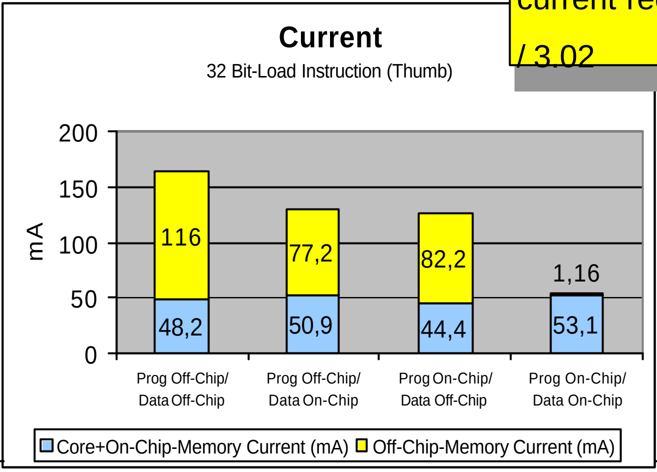
- Can reduce capacity misses
- Can lead to smaller memory in embedded design

last use

On-chip vs. off-chip current

Example: Atmel ARM-Evaluation board

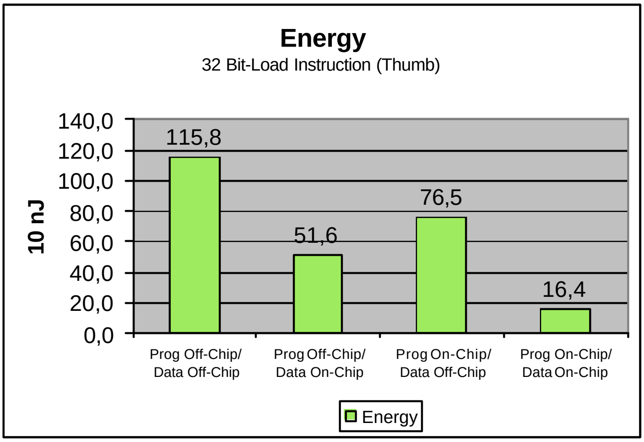




current reduction: / 3.02

On-chip vs. off-chip energy

Example: Atmel ARM-Evaluation board



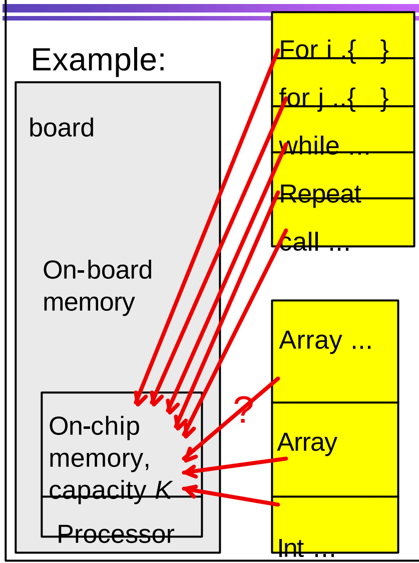
Off-chip access takes more cycles
☞ savings (86%) are larger than for the current.

energy reduction:

/ 7.06

Exploitation of on-chip memory

Example:



board

On-board memory

On-chip memory, capacity *K*

Processor

For i .{ }

for j ..{ }

while ...

Repeat

call ...

Array ...

Array

Int ...

?

Which segment (array, loop, etc.) to be stored in on-chip memory?

Gain g_i and size s_i for each segment *i*.

Maximise gain $G = \Sigma g_i$, respecting constraint $K \geq \Sigma\, s_i$.

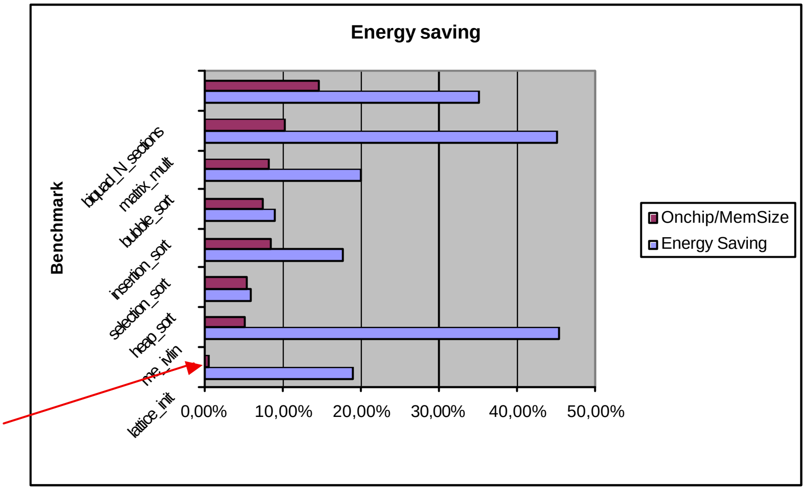
Static memory allocation:

Solution: knapsack algorithm.

Dynamic reloading:

Paging theory.

Results for knapsack algorithm

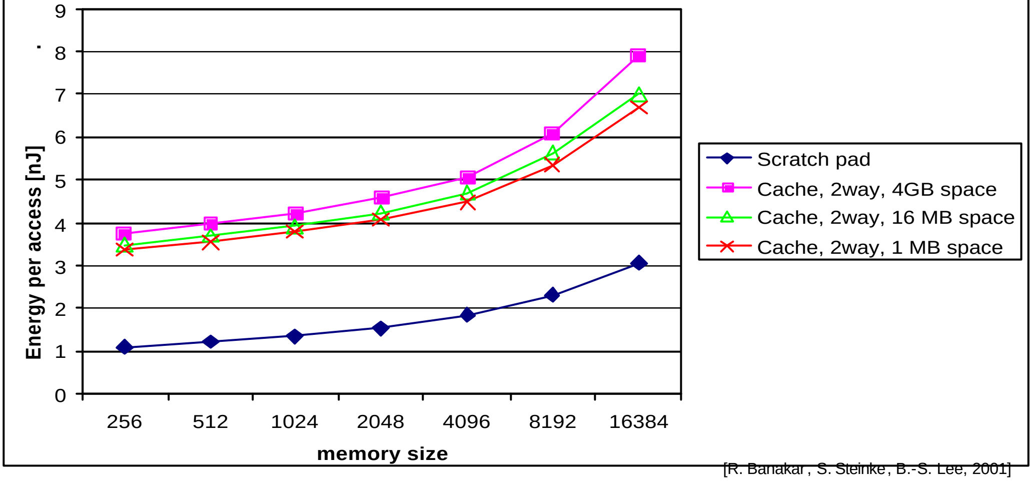


0.5%

[Steinke et al., 2002]

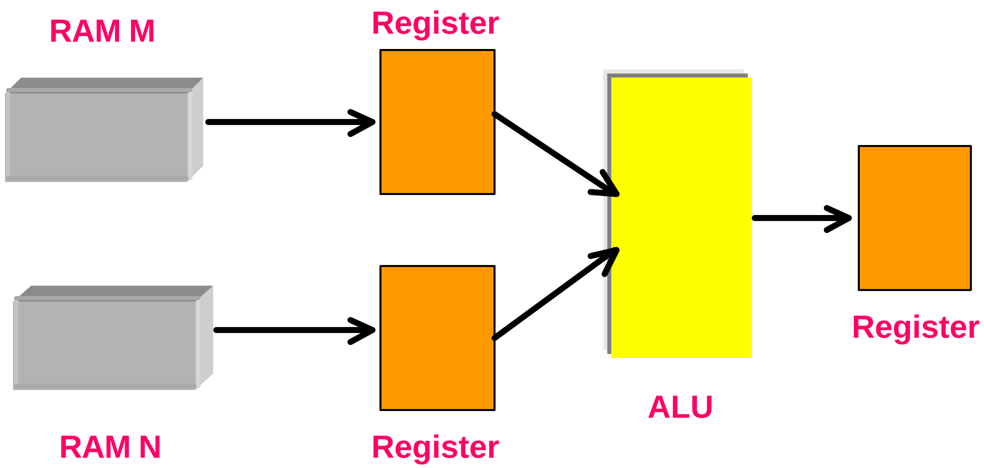
Why not just use a cache ?

Energy consumption in tags, comparators and muxes is significant.



[R. Banakar , S. Steinke, B.-S. Lee, 2001]

III. Dual Memory Loads (Architecture)



RAM M

Register

RAM N

Register

ALU

Register

Dual Memory Loads (Example)

(X*Y)+ Z	Case 1	Case 2	Case 3
M N	X,Y,Z	X,Y Z	X,Z Y
	LD B,X LD C,Y LD A,Z; MUL B,C ADD A,B	DLD B,X;A,Z LD C,Y MUL B,C ADD A,B	DLD B,X;C,Y LD A,Z; MUL B,C ADD A,B
Energy	10.57pJ	9.32pJ	8.85pJ

Register Optimizations

- Jui-Ming Chang, Massoud Pedram, Register Allocation and Binding for Low Power, Univ. of Southern California, ACM 1995
  - technique for minimizing the switching activity of a set of registers shared by different data values
    - assumes known probability density function of the primary input random variables or sufficiently large number of input vectors
  - power consumption of well designed register sets depends mainly on the total switching activity of the registers
  - power model based on switching activity
  - 22.5% power reduction

Register Optimizations

- Software Energy Optimization
  [Tiwari J. VLSI Signal Proc. Aug '96]
  - Reduce Memory Accesses, Make better use of Registers
    - Data for i486
    - register access = 300 mA/cycle
    - memory read (cache hit) = 430 mA/cycle
    - memory write (write-through cache) = 530 mA/cycle
  - can be achieved by e.g. saving the least amount of context during function calls (compiler policies)
  - better utilization of registers
    - optimal register allocation of temporaries
    - global register allocation for the most used variables
      - use register operands as opposed to memory operands

Register pipelining: key idea and results

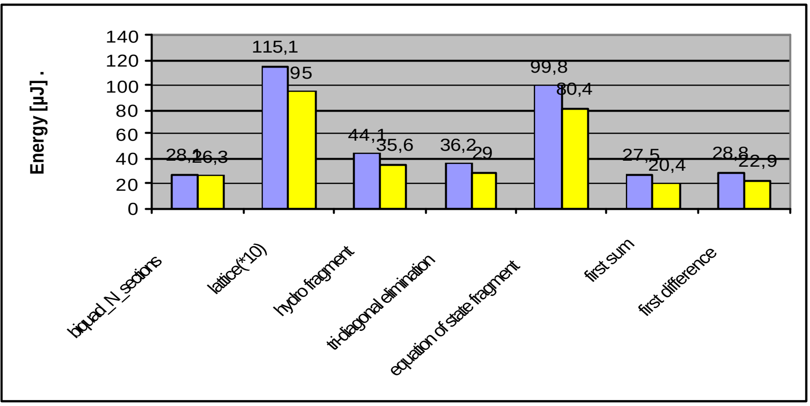
Key idea:

```
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;
```

Results:



[Steinke et al., 2001]

III. Register Allocation

- Objective: to reduce memory system energy dissipation, proposed by Catherine H. at University of Waterloo
- Power model
  - Energy dissipation of register file and memory system

$$E_{msystem}=\sum_{v\in M}(E^{m}_{w(v)}+E^{m}_{r(v)})+\sum_{v\in R}(E^{r}_{w(v)}+E^{r}_{r(v)})$$

  - Assume constant energy for memory read/write
  - Consider switch activity for register file read/write

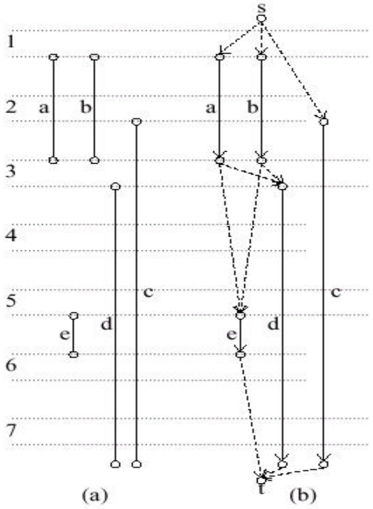
$$E_{msystem}=\sum_{v\in M}(E^{m}_{w(v)}+E^{m}_{r(v)})+\sum_{v1,v2\in R}H(v1,v2)E^{r}_{rw}$$

Register Allocation (cont'd)

- Map to a minimum cost network flow problem
  - Solid arc represents the life time of a variable
  - Dashed arc represents the sharing of one register (memory location) by two variables
  - Capacity: 1 for each arc
  - Cost functions
    - $e_{w(v)\rightarrow r(v)} = 0$
    - $e_{r(v1)\rightarrow w(v2)} = -(E_{w(v2)}^{m} + E_{r(v1)}^{m}) + E_{w(v2)}^{r} + E_{r(v1)}^{r}$
      $= -(E_{w(v2)}^{m} + E_{r(v1)}^{m}) + H(v1, v2)E_{rw}^{r}$
  - Amount of flow: F = number of registers
  - Objective: find a flow of at most F, while minimizing

$$\sum_{r(v1)\rightarrow w(v2)} e_{r(v1)\rightarrow w(v2)} x_{r(v1)\rightarrow w(v2)}$$

- Results: 28-60% energy reduction for memory system



Outline

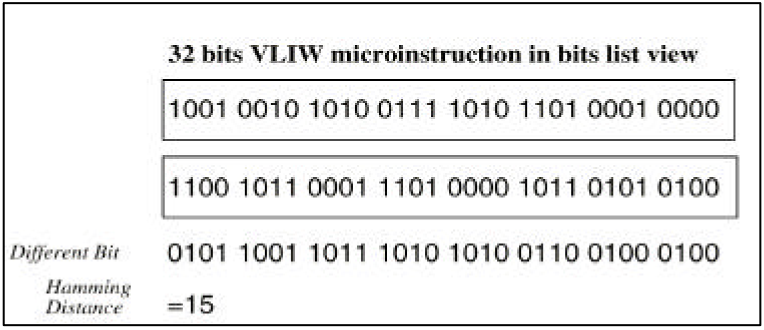
- Introduction
- System conceptualization and modeling
- System Design
  - Software design
    - Compilation techniques (memory hierarchy)
    - High-level transformations
    - Dynamic power management
- Conclusions

I. Instruction Scheduling and Reordering

- Power depends on switching activity, units accessed
- Power-driven scheduling
  - scheduling to reduce pipeline stalls
  - selecting a minimum-power instruction mix for an application
  - reducing switching on address/data lines
    - instruction reordering
      (pairs of instructions have different power consumptions)
    - operand swapping
  - Reorder Instructions to reduce switching effects
    - Not much impact on large general purpose CPUs
    - Useful in DSPs - (~15% benefit) **[Lee et. al. TVLSI, Dec '96]**
  - low-power instruction sets
  - shut down unused units

"Cold" Instruction Scheduling

- Two adjacent instructions have smaller hamming distance → fewer instruction bus lines recharge from 0 to 1 (1 to 0)



32 bits VLIW microinstruction in bits list view

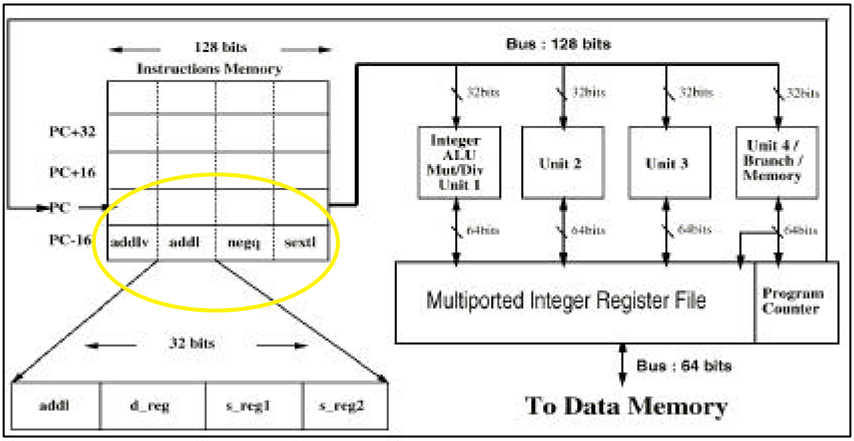
1001 0010 1010 0111 1010 1101 0001 0000

1100 1011 0001 1101 0000 1011 0101 0100

Different Bit 0101 1001 1011 1010 1010 0110 0100 0100

Hamming Distance =15

Machine Architecture



VLIW Experimental Testbed

Instruction Scheduling Policies

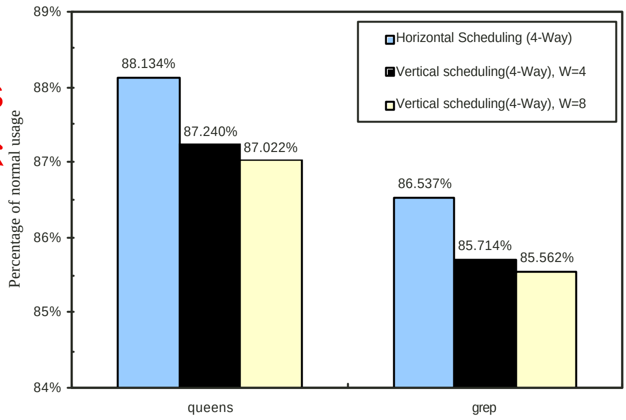
- Software **re-arranging** optimization (helper) **without** performance **penalty**
- “*List Scheduling*” with critical path information
- A side constraint on standard performance-oriented scheduler
  - General problem: side effects!!

Solutions

- Horizontal Scheduling
  - Permute micro-instructions within a given VLIW instruction
- Vertical Scheduling
  - Reorder VLIW instructions’ sequence in a basic block
- Possible Component Activity Solution
  - Extension to Pipeline States

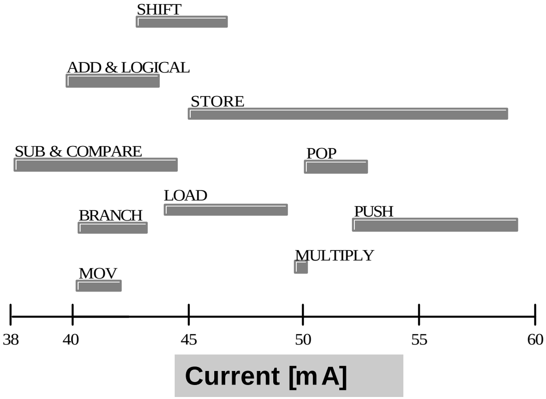
Experiment Results

Only on IBUS



IV. Using energy consumption as a cost function in instruction selection

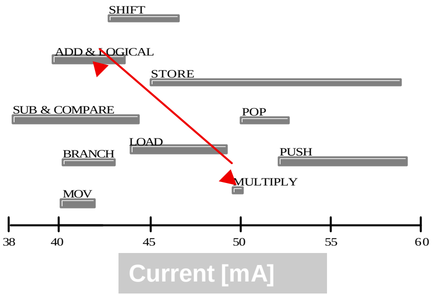
Current for different instructions (ARM core):



Only ~ 10% improvement by using these values in instruction selection

Avoid power hungry multiplies

- Replace multiplies by additions/shifts



Outline

- Introduction
- Power modeling for software optimization
- Compilation techniques
- High-level transformations
- Dynamic power management
- Synergistic techniques
- Software optimization for wireless applications
- Conclusions

Manual Optimization Methodology

- Motivation
  - Many source code optimizations are hard to automate
  - Provide ***guidelines*** for code developers
- Layered approach
  - Enables designers to focus first on abstract view & then perform optimizations narrower in scope
  - Optimization problem is partitioned - enables parallelism
- Three levels of optimization
  - Algorithmic
  - Data
  - Instruction
- ***Prerequisite***: system level power estimator and energy profiler

[Source: Benini]

Algorithmic Optimization

- Identify computationally intensive kernels
- Consider alternative algorithms for those kernels
- Evaluate and implement the most promising algorithms

- MP3 audio example:
  - focused on two most computationally intensive kernels: sub-band synthesis and DCT algorithms
    - – e.g. replacing standard DCT algorithm with Chen DCT, reducing multiply count by 28%

[Source: Benini]

Data Optimization

Goal:

Change representation of data to match the target architecture

- MP3 audio example:
  - signal processing algorithms often use floating point data
  - CPUs usually are much more efficient with integer computation
    - – e.g. StrongARM emulates floating point in software
  - → implement a fixed-precision library
  - only slight changes to the code
  - implement independently form algorithmic optimizations
  - resulted in large energy savings and performance increase

[Source: Benini]

Instruction Optimization

- Exploit characteristics of the target architecture
- Examples of instruction optimizations
  - Integer division and modulo operation
  - Conditional Execution
  - Boolean Expressions
  - Switch Statement versus Table Lookup
  - Register Allocation
  - Variable Types
  - Function Design
  - Inline assembly

Integer division and modulo operation

```
int div16s (int a)
{
    return a / 16;
}
```

```
uint div16u (uint a)
{
    return a / 16;
}
```

- Unsigned modulo 2 shift is 14.7% more energy efficient as it does not require sign extension
- Condition is 51.39% more energy efficient as compared to the modulo operation

```
uint counter1 (uint count)
{
  return (++count % 60);
}
```

```
uint counter2 (uint count)
{
  if (++count >= 60)
       count = 0;
  return (count);
}
```

Conditional Execution

- all ARM instructions are conditional
- conditional execution reduces the number of branches
- code sequences with function calls are not conditionalized
- grouped relational expressions below are 1.25% more energy efficient than the ungrouped ones due to conditionalization

```
int g(int a, int b, int c, int d)
{
   if (a > 0 && b > 0 && c < 0 && d < 0)
        return a + b + c + d;
   return -1;
}
```

Variable Types

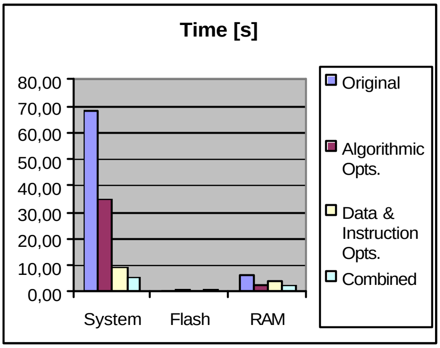
- StrongARM default “**`int`**” variable type is 18.2% more energy efficient than “**`char`**” or “**`short`**”
- sign or zero extending is needed for shorter variable types

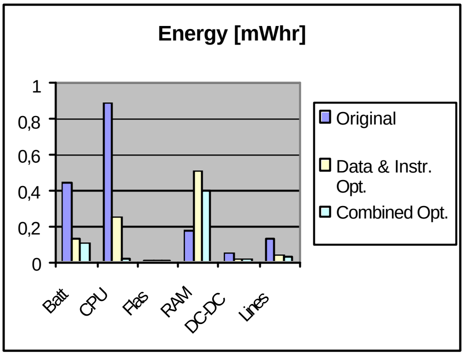
```
char charinc (char a)
{
   return a + 1;
}
```

```
int wordinc (int a)
{
  return a + 1;
}
```

Experimental Results for Software Optimization (MP3 decoder)

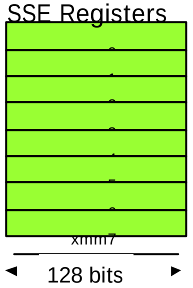
- Overall :10 times faster, 5 times less energy consumption
- Profiler provides results by functions and for each HW component
- Increase in energy consumption and decrease of performance in FLASH due to increase in the code size with algorithmic change

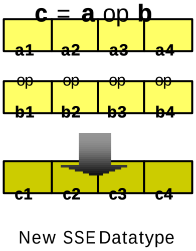




[Source: Benini]

Using Special-Purpose Instructions





Pentium III Benchmarks

Program	Time (ms)		Normalized		Power Savings (%)
	Normal	SIMD	V_{dd}	f	
dot	0.0022	0.0009	0.41	0.60	85.3
fir	0.3700	0.1700	0.46	0.63	81.6
exp	0.0480	0.0260	0.54	0.69	74.4
lms	1.2800	1.0900	0.85	0.89	32.2
fft	5.8000	1.7000	0.29	0.52	92.0
Average Power Savings (%)					73.1

```
class SSEVec {
public:
  float *vec;
  int size;
public:
// Constructors
  SSEVec();
  SSEVec(int size);
// Overloaded operators
  SSEVec& operator+(SSEVec v);
  SSEVec& operator-(SSEVec v);
  SSEVec& operator*(SSEVec v);
  SSEVec& operator/(SSEVec v);
  float operator[](SSEVec v);
}
```

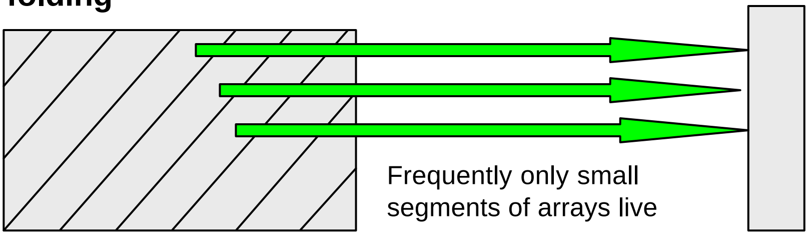
C++ Vector Class

```
SSEVec& operator+(SSEVec v) {
SSEVec *sv = new SSEVec(size);
__m128 m1, m2, m3;
for(int i=0; i<size; i+=4) {
m1 = _mm_load_ps(v.vec+i);
m2 = _mm_load_ps(vec+i);
m3 = _mm_sum_ps(m1,m2);
_mm_store_ps(*sv.vec+i,m3); }
return *sv;
}
```

- MATLAB style code
- Fixed throughput results in substantial power savings

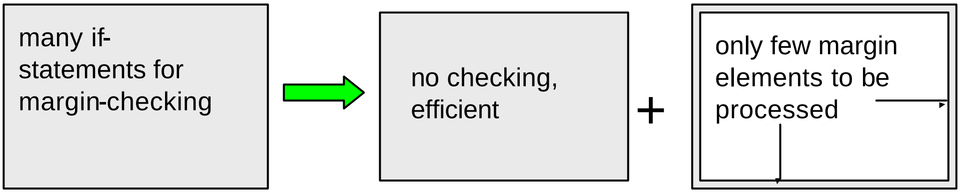
Loop transformations (1)

Array folding



Frequently only small segments of arrays live

Separation of margin handling



many if-statements for margin-checking

no checking, efficient

+

only few margin elements to be processed

Loop transformations (2)

Loop permutation:

```
for (j=0; j<=n; j++)
  for (k=0; k<=m; k++)
    p[j][k] = ....
```

➡

```
for (k=0; k<=m; k++)
  for (j=0; j<=n; j++)
    p[j][k] = ....
```

Next reference to array element adjacent in the cache.

Loop unrolling:

```
for (j=0; j<=n; j++)
  p[j] = ... ;
```

➡

```
for (j=0; j<=n; j+=2)
 { p[j] = ....;
   p[j+1] = ....}
```

Improves utilization of pipeline;
simplifies keeping more values in registers

Loop transformations (3)

Loop tiling:

```
for (j=0; j<=n; j++)
  for (k=0; k<=m; k++)
    p[j][k] = ....
```

➡

```
for (j1=0; j1<=n; j1+=t)
  for (k1=0; k1<=m; k1+=t)
    for (j2=j1; j2<=j1+t-1; j2++)
      for (k2=k1; k2<=k1+t-1; k2++)
        p[j2][k2] = ....
```

Loop adjusted to size of cache lines

Loop fusion/fission

```
for (j=0; j<=n; j++)
  p[j] = ... ;
for (j=0; j<=n; j++)
  p[j]= p[j] + ... ;
```

fusion ➡

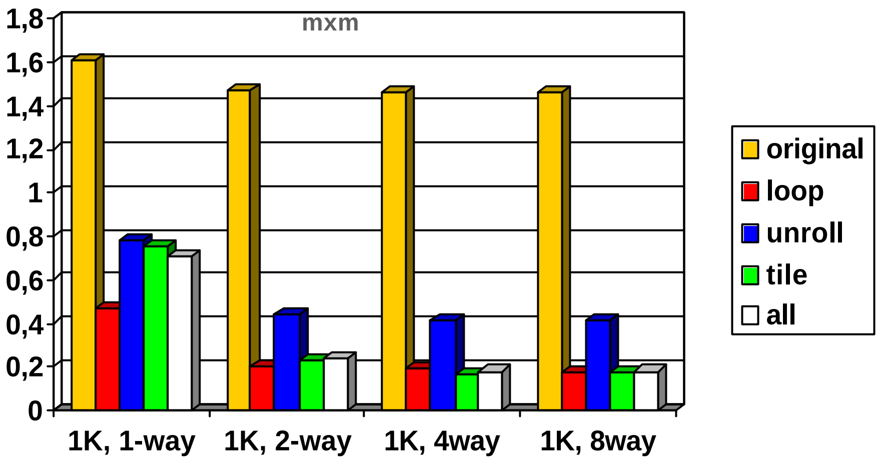
⬅ fission

```
for (j=0; j<=n; j++)
 { p[j] = ....;
   p[j] = p[j] + ....}
```

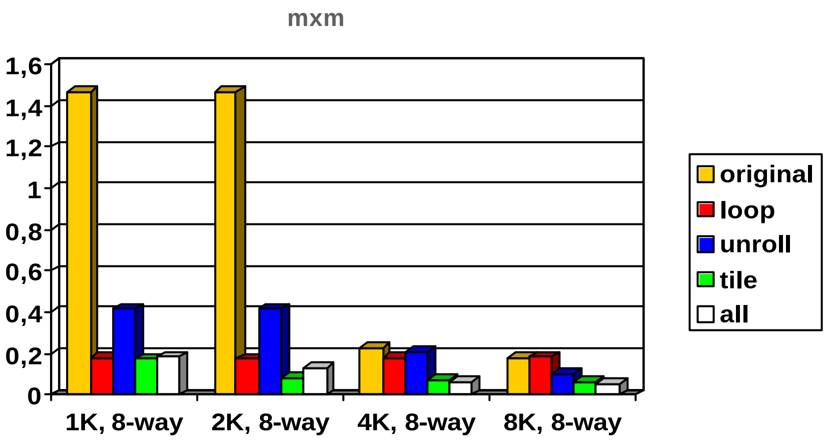
Exploits small HW loops

Improves caching and use of registers

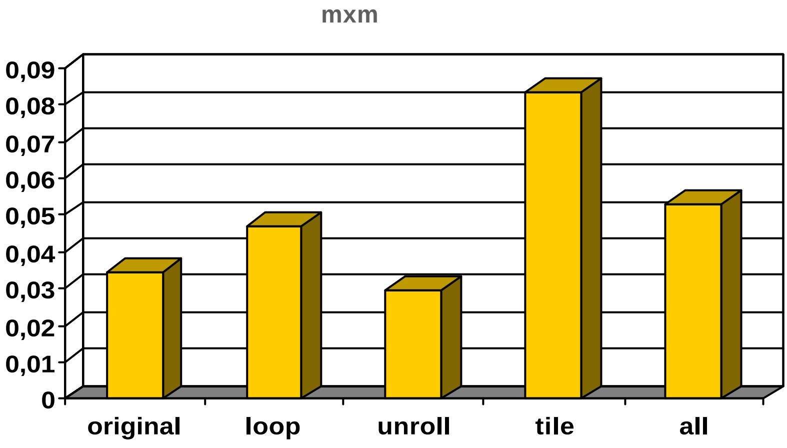
Memory Energy (J)



Memory Energy (J)



Datapath Energy (J)



Improving Locality: Data Transformations

- Linear layout transformations
  - Dimension re-indexing
  - Diagonal (skewed) memory layouts
- Blocked memory layouts

Data transformations might be useful where loop transformations fail, but they have their own problems (e.g. aliasing)

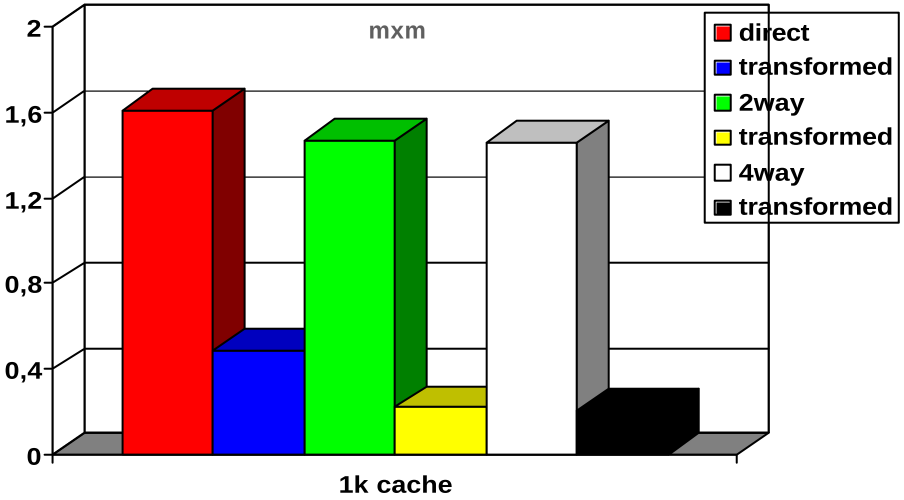
Example: Dimension Reindexing

```
For I = 1, N
   For J = 1, N
      A[I][J] = B[J][I]
```

```
For I = 1, N
   For J = 1, N
      A[I][J] = B'[I][J]
```

- Imitates the effect of a different layout
- Should be applied with a global view
- Less negative impact on datapath energy

Data Transformation Effects



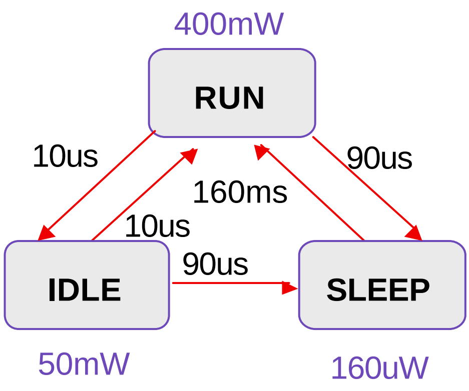
Outline

- Introduction
- System conceptualization and modeling
- System Design
  - Software design
    - Compilation techniques (memory hierarchy)
    - High-level transformations
    - Dynamic power management
- Conclusions

Hardware support

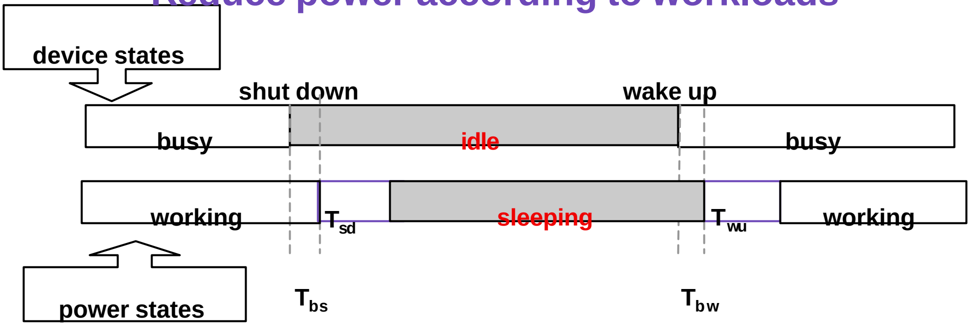
Example: STRONGARM SA1100

- RUN: operational
- IDLE: a sw routine may stop the CPU when not in use, while monitoring interrupts
- SLEEP: Shutdown of on-chip activity



The opportunity

Reduce power according to workloads



T_{sd}: shutdown delay

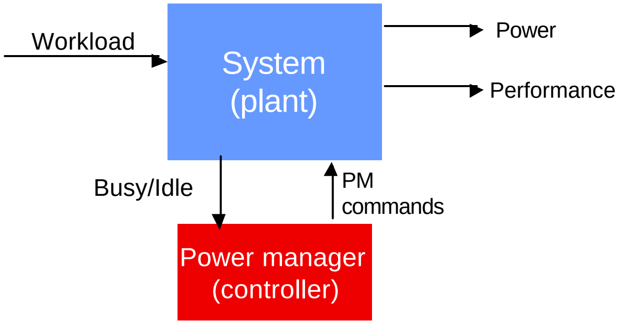
T_{wu}: wakeup delay

T_{bs}: time before shutdown

T_{bw}: time before wakeup

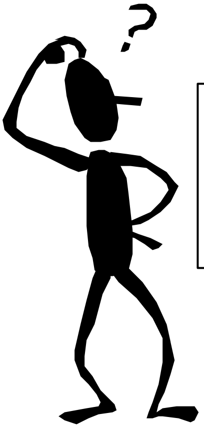
Shutdown only during long idle time

A control system abstract model



- System responds to input (workload) with a performance level and a power consumption
- Controller samples B/I and issues PM commands
- Objective: *minimize power* for a *desired performance*

The challenge



- *Is an idle period long enough for shutdown (T_{be})?*

Predicting the future!

Approaches to workload prediction

- Timeout : [Karlin94, Douglis95, Li94, Krishnan99]
  - – Shutdown the system when timeout expires
- Predictive : [Chung99, Golding95, Hwang00, Srivastava96]
  - – Shutdown the system if prediction is longer than T_{be}
- Stochastic : [Benini99, Qiu99, Simunic01]
  - – Model the system stochastically (Markov chain)
  - – Policy optimization with constraints
    - • Trade off between energy saving and performance
  - – Non-deterministic decision
  - – Discrete time model / Continuous time model
  - – Superior to predictive and timeout

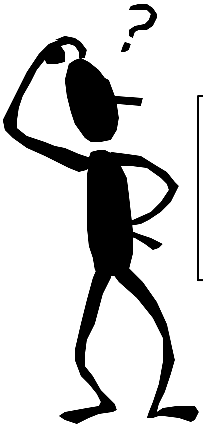
Performance of predictors

Algorithm	P	N_{sd}	N_{wd}
off-line	0.33	250	0
Semi- Markov	0.40	326	76
Sliding Window	0.43	191	28
Device-Specific Timeout	0.44	323	64
Learning Tree	0.46	437	217
Exponential Average	0.50	623	427
always on	0.95	-	-

P : average power **N_{sd}: number of shutdowns**

N_{wd} : wrong shutdowns (actually waste energy)

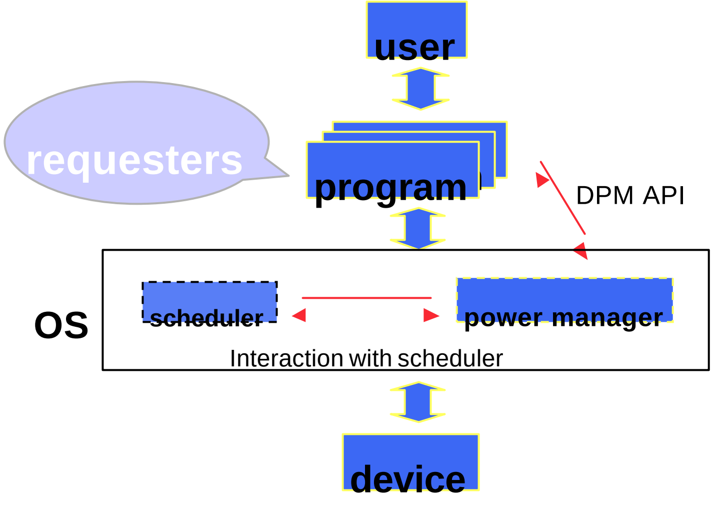
Can I do better than that?



- *Improve workload information*

Application-aware DPM!

Application aware DPM



user

requesters

program

DPM API

OS

scheduler

power manager

Interaction with scheduler

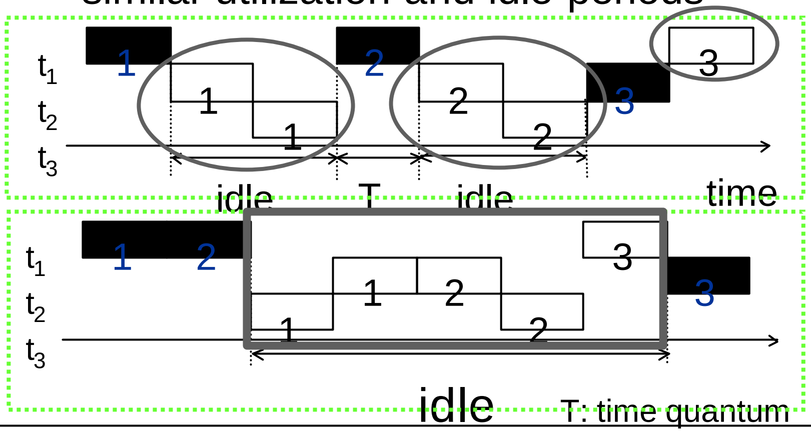
device

Interaction with processes

- Concurrent processes
  - Created, executed, and terminated
  - Have different device utilization
  - Generate requests only when running (occupy CPU)
- Power manager is notified when processes change state
- Processes ask for “service levels” to the PM

Interaction with Task Scheduling

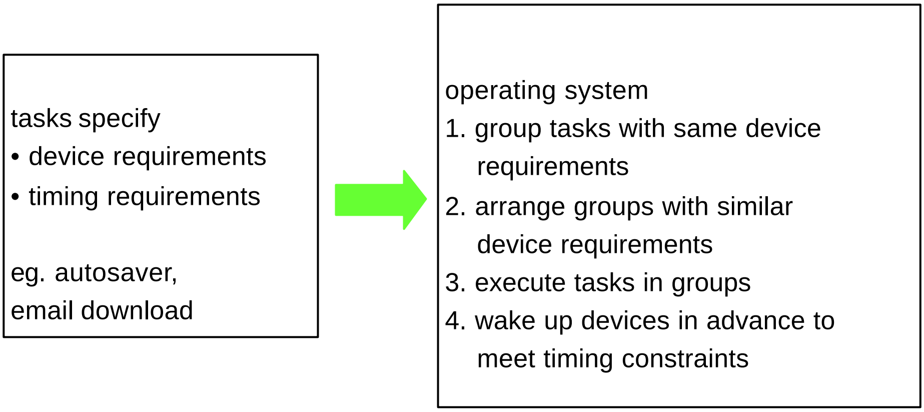
Rearrange task execution to cluster similar utilization and idle periods



Shutdown-friendly scheduling

- Cluster processes with similar utilization patterns
  - Localize resource usage in time
- Tradeoff against latency
  - A process may be delayed
- Exploit application-knowledge
  - Processes can specify their latency requirements via API calls
  - Safe assumptions on legacy processes: can specify laxity of timing constraints

Low-Power Scheduling in Practice



tasks specify
• device requirements
• timing requirements

eg. autosaver,
email download

operating system
1. group tasks with same device requirements
2. arrange groups with similar device requirements
3. execute tasks in groups
4. wake up devices in advance to meet timing constraints

Power and Overhead Reduction

Timing constraints	Power	Power State Changes
1000	67%	57 %
500	69 %	61%
100	80 %	95 %

100%: scheduling without considering power management

Task scheduling reduces power *and* overhead.

Process awareness improvements

better

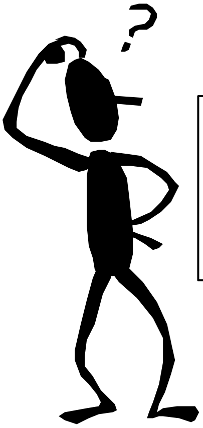
↑ ↓ ↓

Algorithm	T_s (ksec)	N_d	P_a
PA-DPM*	23.6	181	1.00
[ICCAD 97]	17.9	325	1.35
Timeout (2min)	12.3	64	1.56

T_s: time during sleeping state
N_d: number of shutdowns
P_a: average power (normalized to row 1)

[Yung01]

Can I do better than that?



- *Application-level DPM*

Shaping the workload!

Example: Communication Power

NICs powered by portables reduce **battery life**

In general

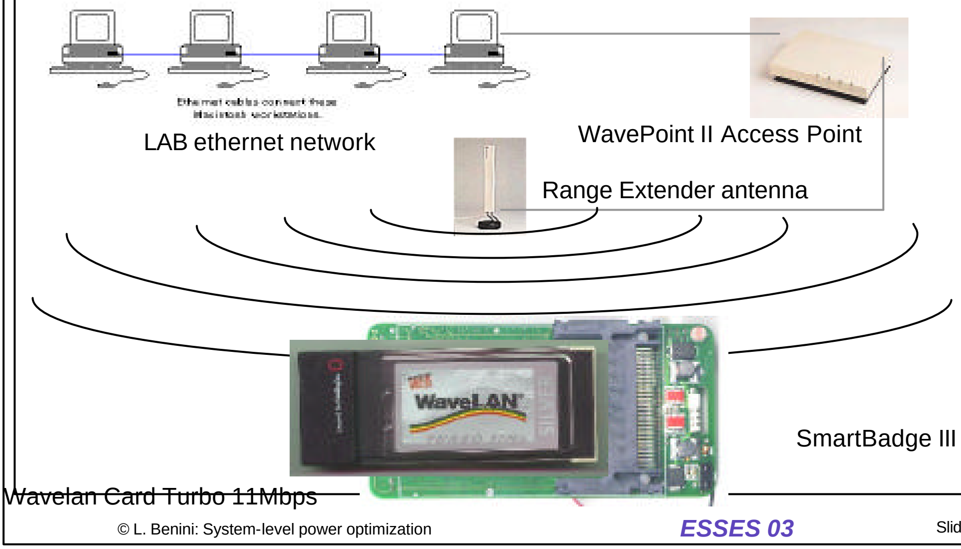
- Higher bit rates → Higher power consumption
- 90% of power is drawn by listening to the radio channel!!

Proper use of PHY layer services by MAC is critical

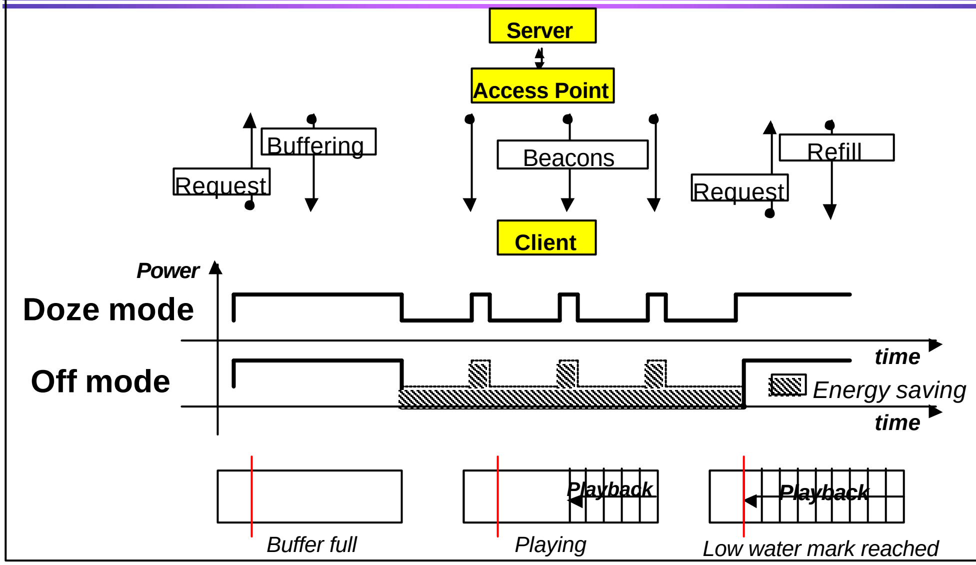
NIC power states

- Transmit mode
- Receive mode
- Doze mode
- Off-mode
  - NIC completely turned off
  - Use only when streaming multimedia is somehow requested by the client
  - Power overhead for frequent switches
  - Must come with proper buffering strategies

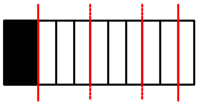
Networked Streaming multimedia


Ethernet cables connect these Macintosh workstations.
LAB ethernet network
WavePoint II Access Point
Range Extender antenna
WaveLAN
SmartBadge III
Wavelan Card Turbo 11Mbps

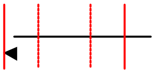
Off mode power savings



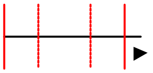
Buffering strategies



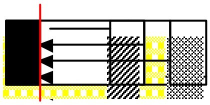
Where shall I put the LWM?



- Higher error probability
- Exploits NIC off-state
- ~~Minimum value exists to allow data acquisition~~



- lower error probability
- Incurs NIC off-state overhead
- Maximum value: Buffer_length – 1 block

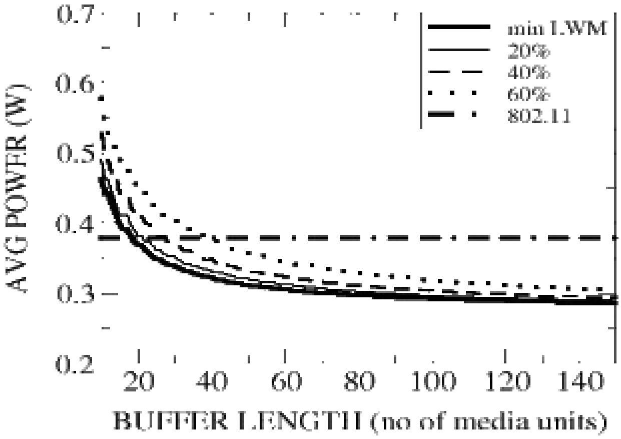


How long should the buffer be?

- It dipends on the memory availability
- The longer the buffer length, the more the benefits of NIC off-state

BUFFERING STRATEGIES SHOULD BE POWER AWARE

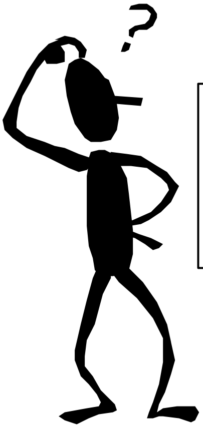
Results



[Bertozzi02]

- ✓ Low length buffers incur off mode power overhead
- ✓ For high length buffers, good power saving

What if the system is not idle?

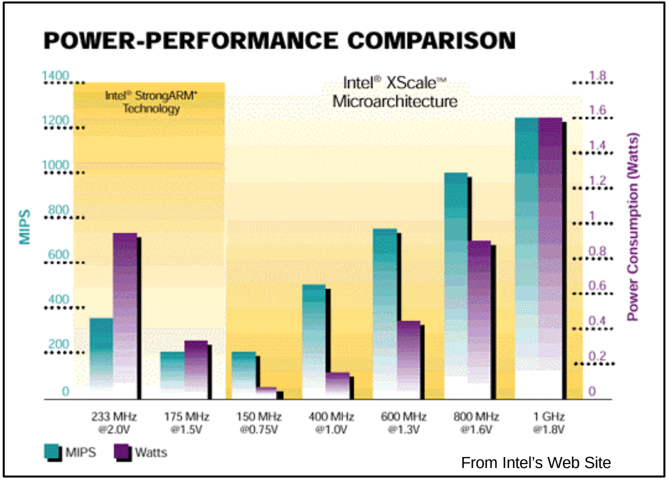


- *Exploiting underutilization*

Dynamic Voltage Scaling!

Variable-voltage processor example: INTEL Xscale

- Discrete VS
  - 3 to 4 voltages
  - More frequencies
- Transition penalties
  - ≈ milliseconds
  - Dominated by supply voltage transient
- System support
  - Voltage supply circuitry
  - Interface circuits (!!)
- Voltage ranges
  - Decrease with tech.



From Intel's Web Site

[INTEL01]

Variable Frequency

Energy as a function of frequency

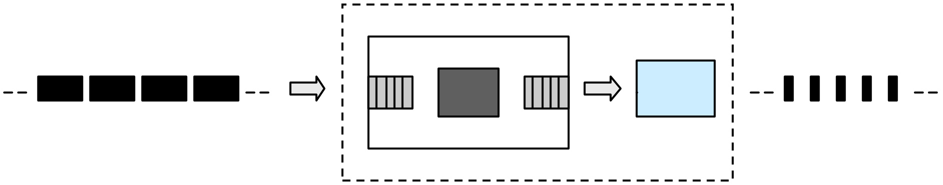
- Energy consumption: $E_{frame} = V_{DD}^{2} \cdot C_{eff} \cdot f \cdot T_{frame}$
- *T* is given by: $T_{frame} = N_{frame} \cdot t = (N_{useful} + N_{idle}) \cdot t$
- Hence the energy equation can be written as:

$$E_{frame} = V_{DD}^{2} \cdot C_{eff} \cdot f \cdot (N_{useful} + N_{idle}(f))$$

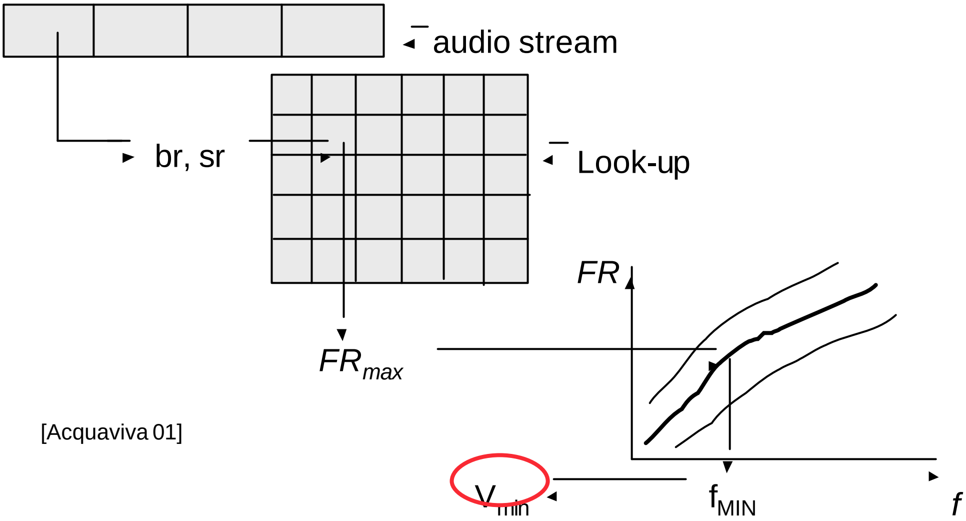
- Energy savings
  - Reduces costs of *memory latency*
  - Reduces costs of *I/O synchronization*
- Discrete frequency range
  - Adaptation mismatch

Streaming real-time single application example

- An ***MPEG*** stream is composed of frames
  - The decoder produces audio samples by processing block of frames.
- SW and HW buffering allows synchronization among input rate, output rate and processing time
- Each block must be elaborated in a fixed time, during this time the CPU does not access input or output buffers
- Output data are sent to the audio ***CODEC*** by the ***DMA***



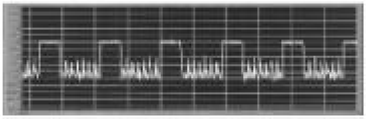
Single task: Frequency setting



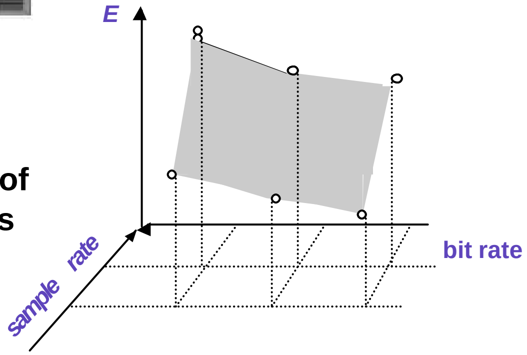
[Acquaviva 01]

Experimental Results

- Current waveform – no policy applied

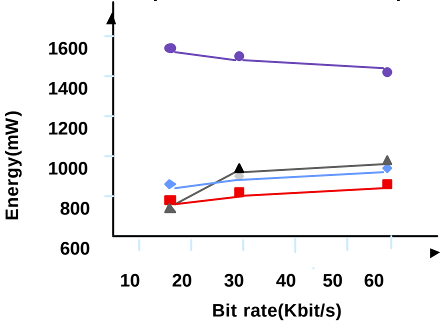


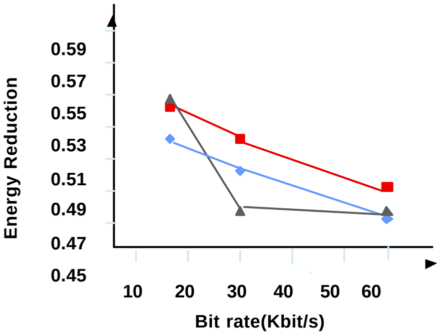
- **Energy as a function of stream characteristics**



Experimental Results (II)

- Comparison between policies





$$\text{Energy Reduction} = \frac{E_{\max} - E_{opt}}{E_{\max}}$$

●—● **without policies**

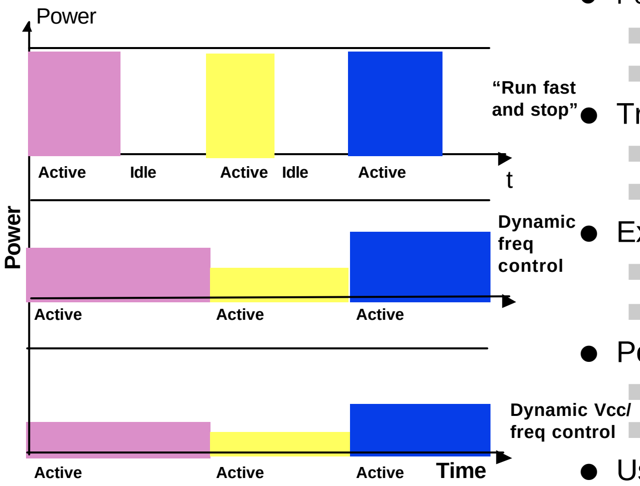
■—■ **mixed policy**

◆—◆ **shutdown**

▲—▲ **variable frequency**

Sample rate 16KHz

Multiple tasks: voltage scheduling



- Performance constraints
  - Static/dynamic workloads
  - Hard/soft deadlines
- Transition costs
  - Number of states
  - State transition overhead
- Execution time estimation
  - WCET
  - Stochastic
- Policy granularity
  - Inter-task
  - Intra-task
- Using sleep states

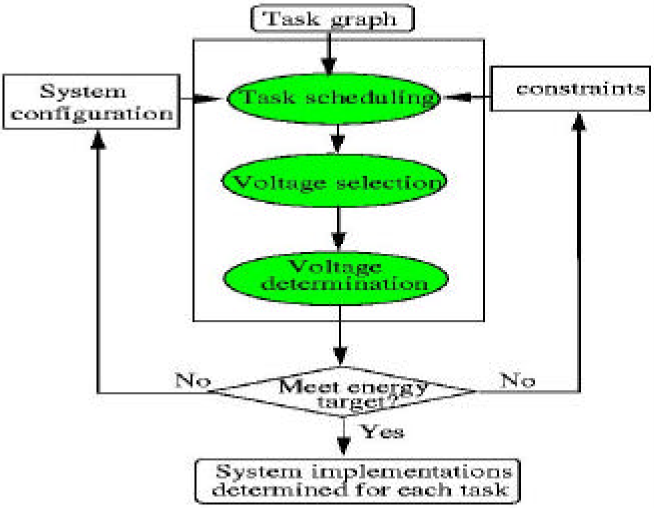
Conservative vs. aggressive DVS approaches

- Conservative: hard real-time guarantees
  - Basic idea
    - – Use conservative estimates (WCET)
    - – Perform scheduling with RT guarantees
    - – Stretch execution when running faster than WCET
  - Slack recovery
- Aggressive: soft real-time constraints
  - Basic idea
    - – Monitor system usage @ run time
    - – Predict future usage based on past history
    - – Set speed (and voltage) based on prediction
  - Slack prediction

Conservative DVS formulations

- Task type: Dependent tasks (task graphs)
- Task characteristics: tasks can have different energy profiles, deadlines, release times
- Number of processors: single or multiple
- Voltage type: Continuous or discrete
- Voltage resolution: Different cycles of the same task can have different voltages

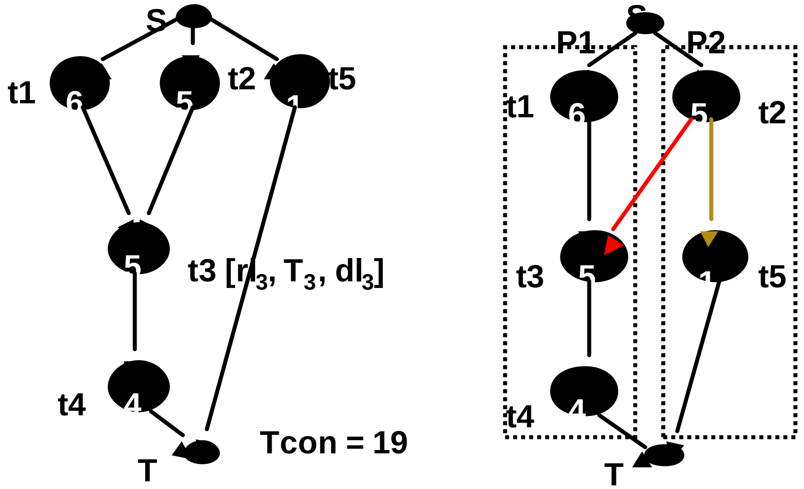
Overall Flow



Approach

- An Integer Programming (IP) formulation of the voltage selection problem
  - Number of variables and constraints linear to number of tasks
  - Polynomial time solvable for continuous voltage
  - Efficient approximation for discrete voltage
- Earliest Deadline First (EDF) scheduling for a single processor and a priority-based list scheduling for multiple processors
  - Polynomial time algorithm aimed at providing more energy saving

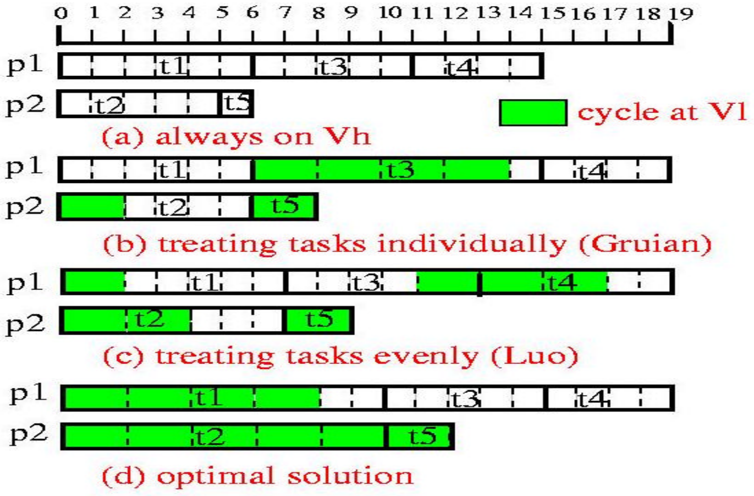
Scheduling



Before Scheduling

After Scheduling

Scheduling and Voltage Setting (2 Vdds)



(a) always on Vh

(b) treating tasks individually (Gruian)

(c) treating tasks evenly (Luo)

(d) optimal solution

Energy Consumption of the Four Implementations

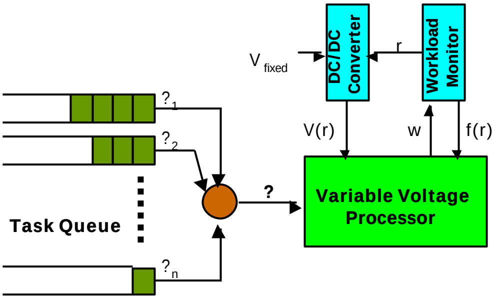
Processor Data (normalized)

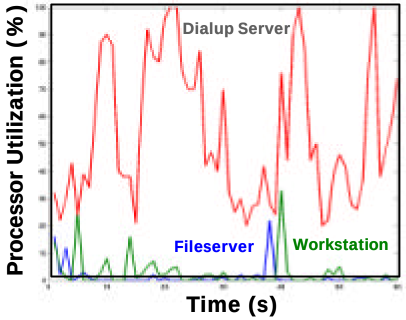
	V	CT_V	ET_V
V_h	2	1	4
V_1	1	2	1

Energy consumption of different implementations

	N_{Vh}	N_{Vl}	d	E	Sav
(a)	21	0	15	84	0%
(b)	15	6	19	66	21
(c)	14	7	19	63	25
(d)	11	10	19	54	36

Aggressive DVS: Workload Dependant Processing





- How to predict workload, w?
- How frequently should the processing rate, f(r), be updated

Prediction Strategy

Predicted Workload

$$w_p[n+1] = \sum_{k=0}^{N-1} h_n[k]w[n-k]$$

Previous Workloads

Moving Average Workload (MAW)

$$h_n[k] = \frac{1}{N} \forall n, k$$

- Simplest
- Performance degradation with fast loads

Exp. Weighted Average (EWA)

$$h_n[k] = a^{-k}$$

- Lower significance of older data
- Event prediction context [Hwang97]

Expected Workload State (EWS)

$$w[n+1] = \mathrm{E}\{w[n+1]\} = \sum_{j=0}^{L} w_j p_{ij}$$

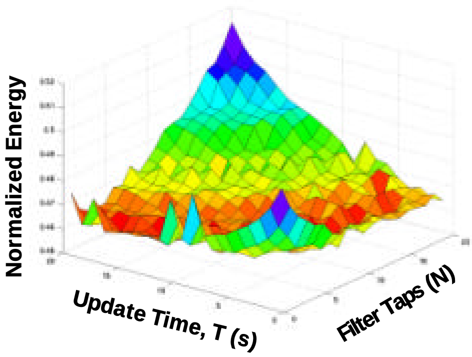
- Probabilistic formulation
- Transition matrix updated every slot

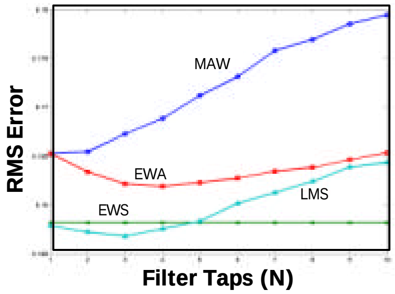
Least Mean Square (LMS)

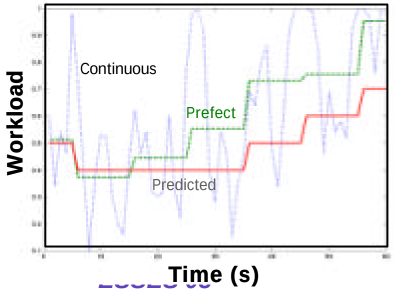
$$h_{n+1}[k] = h_n[k] + m w_e[n] w[n-k]$$

- Adaptive filter, self-adjusting
- Convergence issues

Prediction Performance





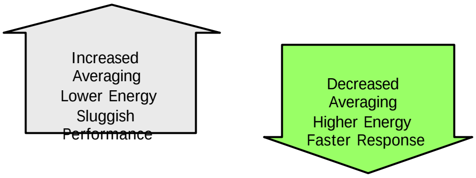


- N = 3 taps and T = 5 s is a good choice

Energy Performance Tradeoff

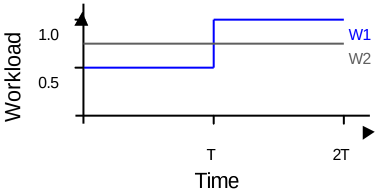
- Averaging is energy efficient

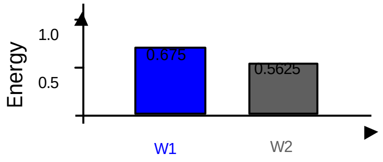
$$\frac{r_1^2+r_2^2}{2} \ge \left(\frac{r_1+r_2}{2}\right)^2 \rightarrow \overline{E(r)} \ge E(\bar{r})$$



Increased Averaging Lower Energy Sluggish Performance

Decreased Averaging Higher Energy Faster Response





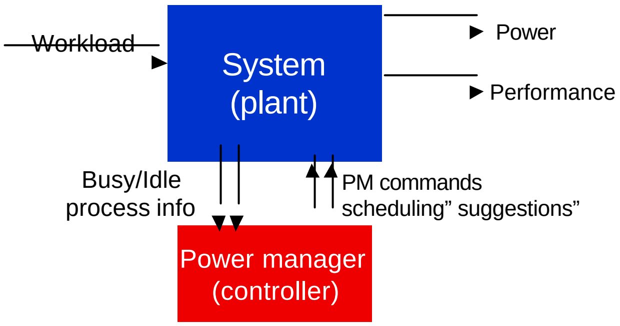
- Update time T depends on
  - Maximum allowed performance hit
  - DC/DC converter and frequency change overheads

Energy Savings

Trace	Filter	Energy Savings Ratio (ESR)			ESR Comparison		F $_{avg}$ (%)	F $_{max}$ (%)
		Max	Perfect	Actual	Max / Perfect	Perfect / Actual		
Dialup Server	MAW	2.9	2.4	2.2	1.2	1.10	10.6	34.8
	EWS			2.1		1.11	10.8	36.3
	EWA			2.2		1.09	10.6	35.4
	LMS			2.3		1.03	14.7	43.1
File Server	MAW	76.7	23.5	16.7	3.3	1.41	12.6	42.8
	EWS			15.7		1.50	7.4	33.8
	EWA			16.7		1.41	9.2	37.4
	LMS			19.6		1.20	14.1	47.7
User Work-Station	MAW	445.9	275.2	52.7	1.6	5.22	3.6	35.3
	EWS			59.5		4.63	3.8	35.1
	EWA			52.1		5.28	3.7	35.6
	LMS			53.0		5.19	3.9	36.0

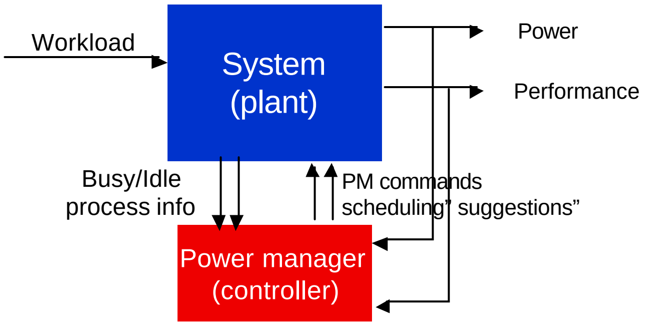
[Sinha, VLSI 01]

A control system abstract model


Workload
System (plant)
Power
Performance
Busy/Idle process info
PM commands scheduling" suggestions"
Power manager (controller)

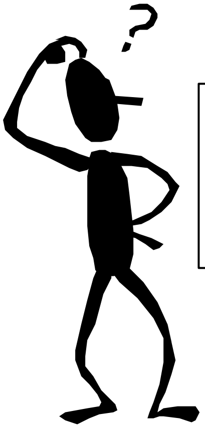
- Better observation of the system
- More control "knobs"
- Objective: *increase controllability & observability*

What about closed-loop control?



- Stabilizes the system
- Reduces sensitivity to "modeling noise"
- Challenge: *high quality power/performance sampling*

Many degrees of freedom...



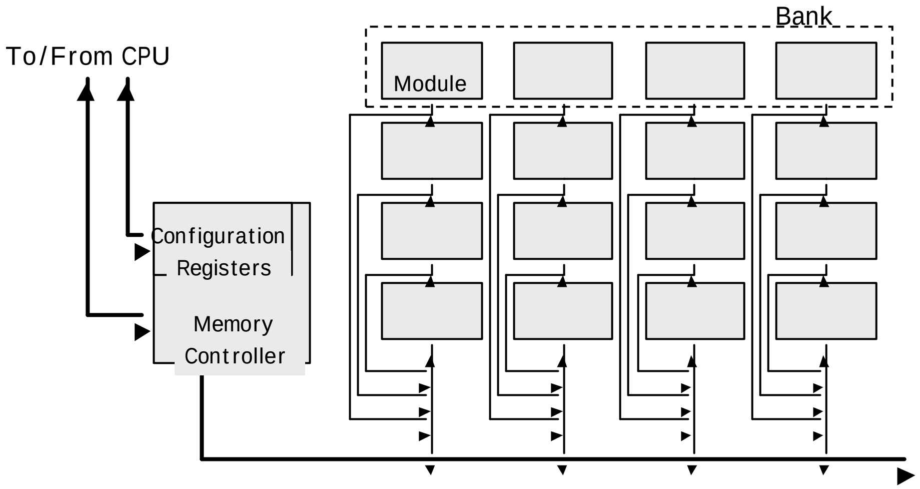
- *Putting it all together*

The energy-efficient OS

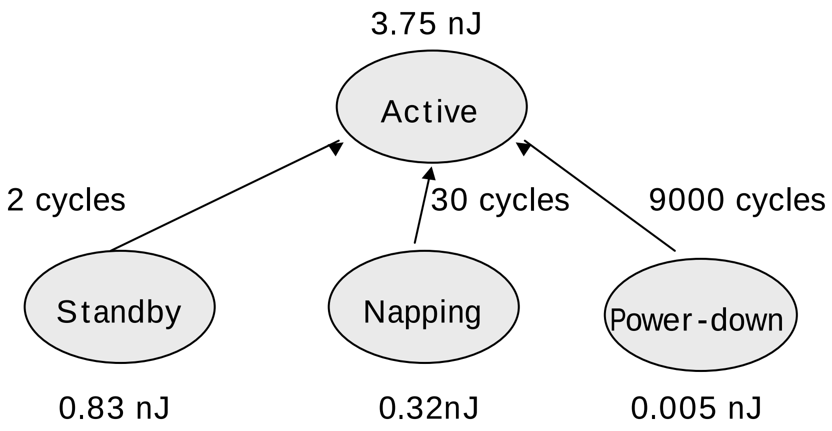
Energy efficient OS: features

- Controls multiple heterogeneous devices
  - Multiple sleep states, multiple active states
- Manages performance constraints
  - Minimizes latency (for event handling)
  - Satisfies throughput bounds & deadlines
  - Handles hard and soft constraints
- Interacts with applications & other OS services
  - Supports DPM APIs
  - LP scheduler, LP memory manager
- Various practical research solutions: HPL-Stanford-UNIBO, UCI, Berkeley, Delft
- Closed-loop control is yet to be explored

DRAM Memory Architecture

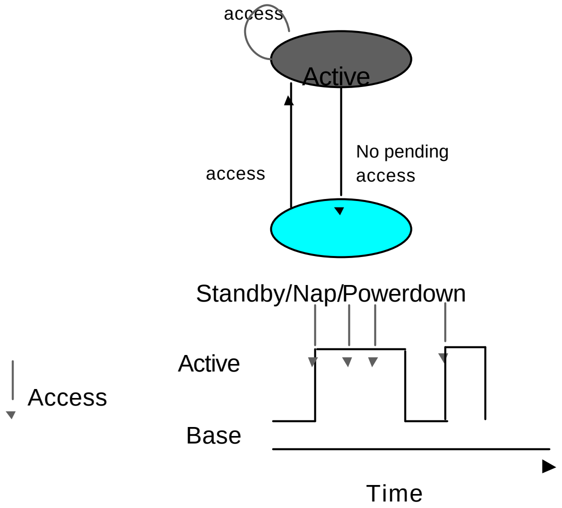


Memory management:
Memory Operating Modes



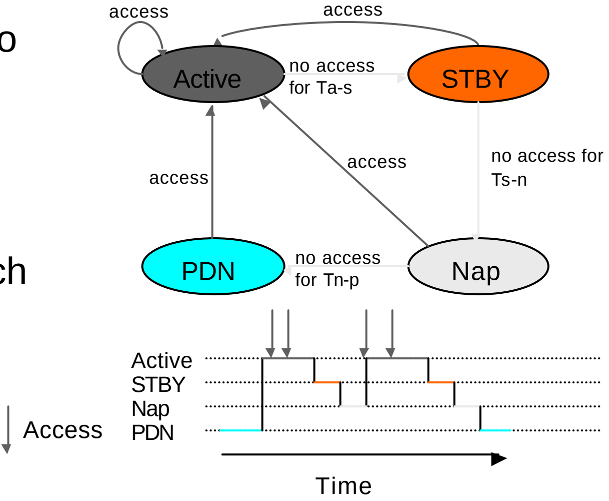
Dual-state (Static) HW Power State Policies

- All chips in one base state
- Individual chip Active while pending requests
- Return to base power state if no pending access

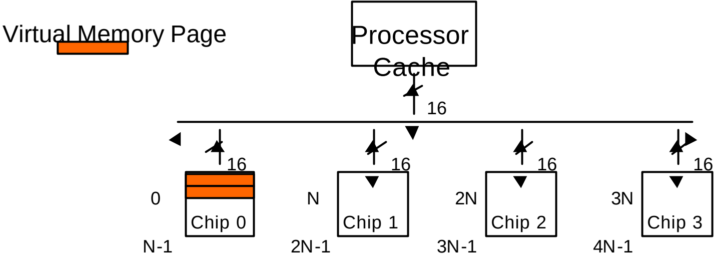


Quad-state (Dynamic) HW Policies

- Downgrade state if no access for threshold time
- Independent transitions based on access pattern to each chip
- Competitive Analysis
  - rent-to-buy
  - Active to nap 100's of ns
  - Nap to PDN 10,000 ns



Page Allocation and PADRAM



- Physical address determines which chip is accessed
- Assume non-interleaved memory
  - Addresses 0 to N-1 to chip 0, N to 2N-1 to chip 1, etc.
- Entire virtual memory page in one chip
- Virtual memory page allocation influences chip-level locality

Page Allocation Polices

- Random Allocation
  - Pages spread across chips
- Sequential First-Touch Allocation
  - Consolidate pages into minimal number of chips
  - One shot
- Frequency-based Allocation
  - First-touch not always best
  - Allow movement after first-touch

Conclusions

- System-level power minimization requires hardware & software interactions
- Architectures provide increased degree of control on power vs. performance
- Software must exploit it
- It is important to understand both!

The end

Thank you very much for attending this class !