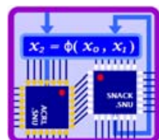
Virtual Memory Environments for Instruction Scratchpad Memory Management

Jaejin Lee
Advanced Compiler Research Laboratory
School of Computer Science and Engineering
Seoul National University
http://aces.snu.ac.kr

Other contributors:
Bernhard Egger, Chihun Kim, Choonki Chang, Yoonsung Nam, Sang Lyul Min, and Heonshik Shin

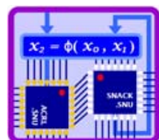
Dagstuhl Seminar 07101, Germany, March 4 – 9, 2007



This Talk is based on,

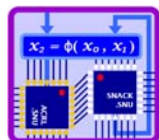
- Bernhard Egger, Chihun Kim, Choonki Jang, Yoonsung Nam, Jaejin Lee, and Sang Lyul Min. A Dynamic Code Placement Technique for Scratchpad Memory using Postpass Optimization. *In Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems* (CASES'06), Seoul, Korea, October 2006

- Bernhard Egger, Jaejin Lee, and Heonshik Shin. Scratchpad Memory Management for Portable Systems with a Memory Management Unit. *In Proceedings of the International Conference on Embedded Software* (EMSoft'06), Seoul, Korea, October 2006.



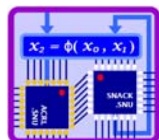
Motivation and Goal

- Scratchpad memory (SPM) in embedded systems
  - Smaller energy consumption per access than caches
  - Less die area required than caches
  - 1-cycle access
  - Managed by software (explicit addressing)
- To reduce energy consumption
  - But, keep (at least) the same performance



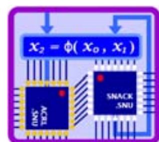
Our Approach

- Based on ***demand paging*** and ***post-pass*** optimization
- Two mechanisms
  - Software only
    - For low-end embedded systems with no MMU
  - With MMU support
    - For contemporary portable devices with an MMU
    - New horizontally-partitioned memory subsystem

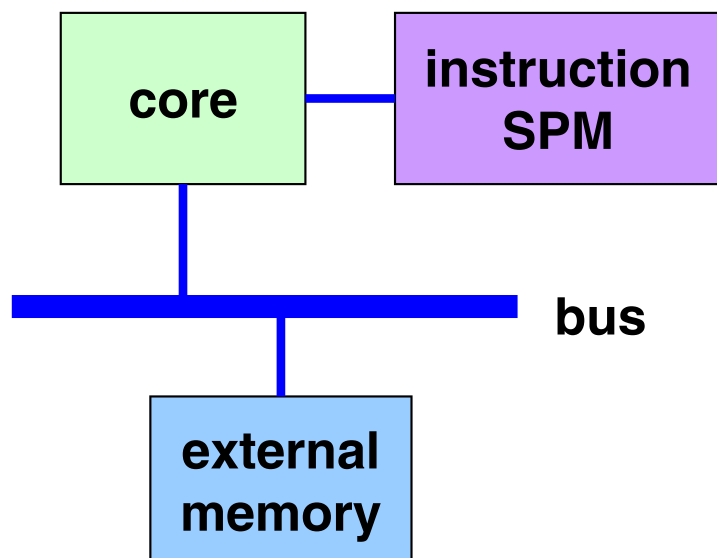


Outline

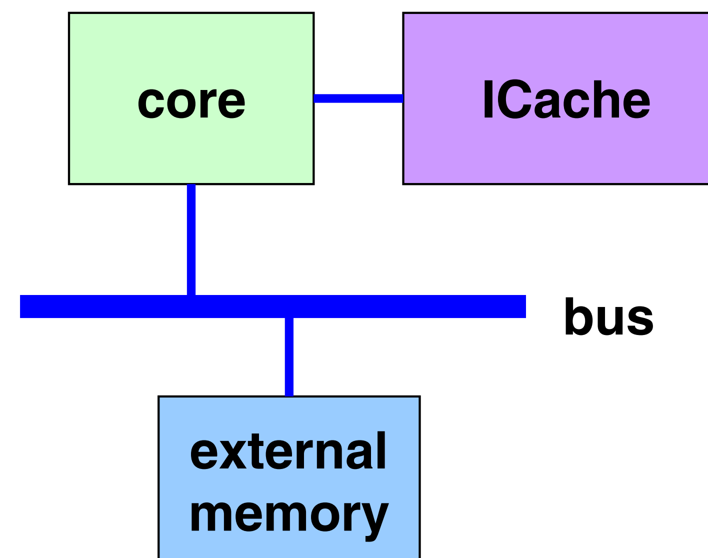
- Motivation
- ***Software-only virtual memory***
- Virtual memory with MMU support
- Conclusions



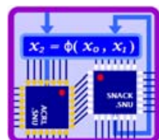
Target Architecture



target architecture

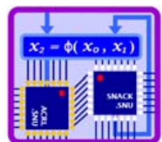


reference architecture

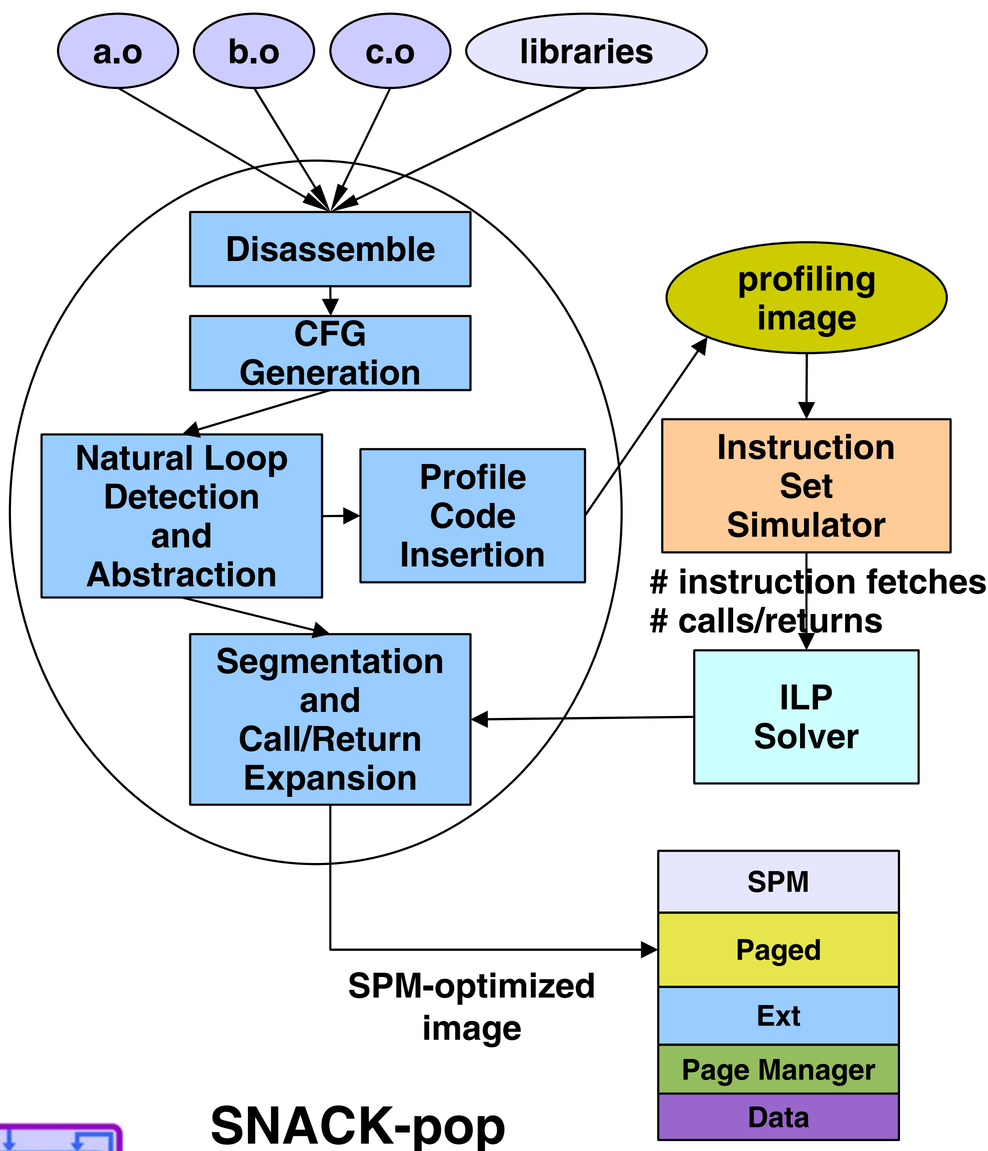


Related Work

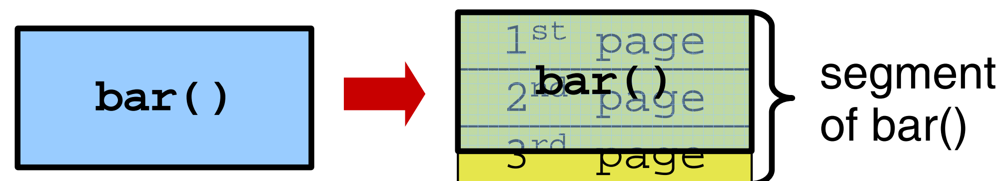
- Static allocation
  - Verma et al. (DATE'04): Cache-Aware Scratchpad Allocation Algorithm
- Dynamic allocation
  - Steinke et al. (ISSS'02): Reducing Energy Consumption by Dynamic Copying of Instructions onto Onchip Memory
- Udayakumaran et al.(CASES'03), Poletti et al. (DAC'04), Angiolini et al (CASES'04)

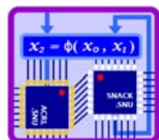


Post-pass Optimization (Seoul National university Advanced Compiler tool Kit)

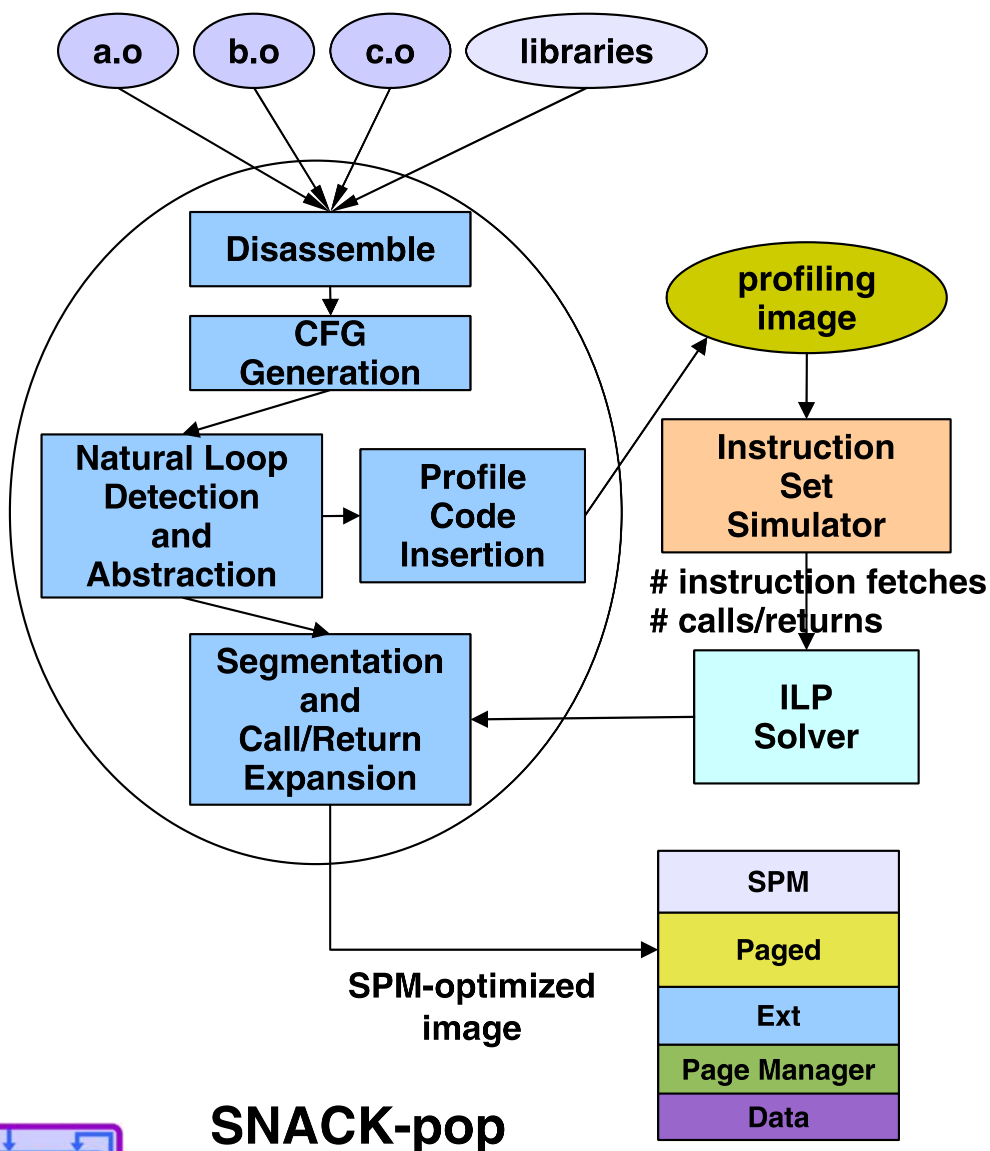


- Break down functions into three classes: ***SPM***, ***Ext***, and ***Paged***
  - Based on profiling and ILP
- Natural loop extraction
  - Extracted loops are transformed into independent functions
- Each paged function is transformed into a segment
  - Basic loading units are segments





Post-pass Optimization (contd.)



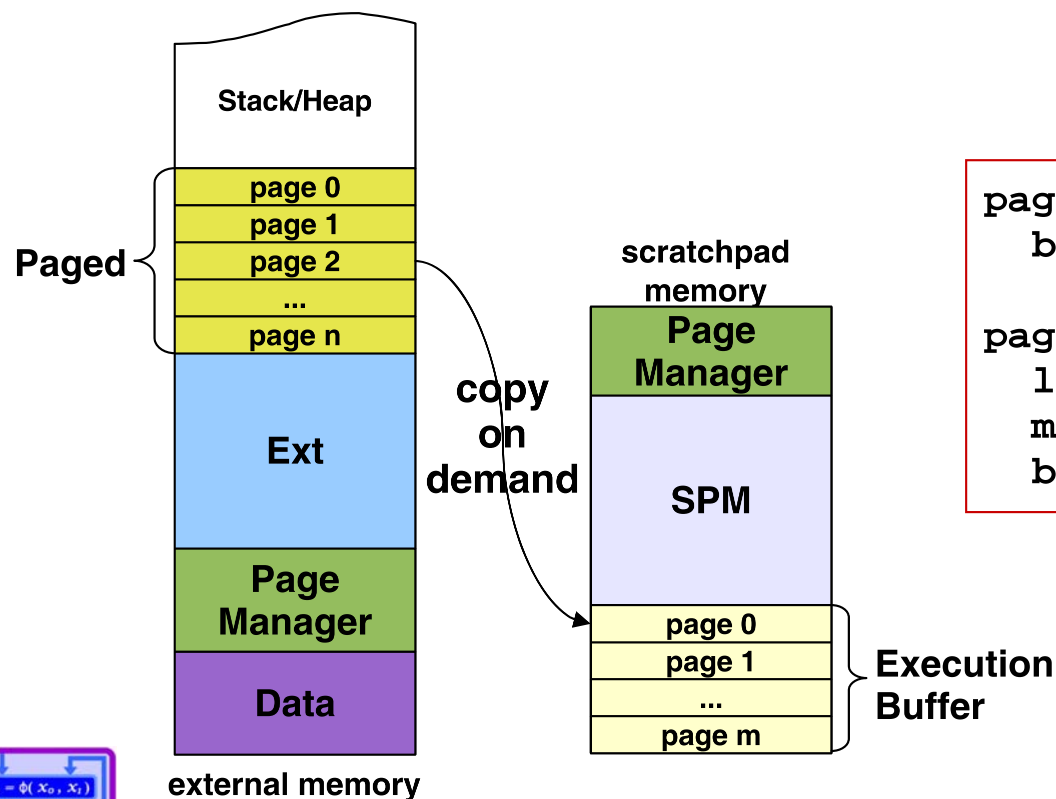
- Calls/Returns to paged functions must be intercepted by the page manager

```
bl    bar
  ↓
bl    pagetable[bar]
```

```
mov   pc, lr
  ↓
push lr
b     pagemanager_return
```

Dynamic Code Placement: Page Manager

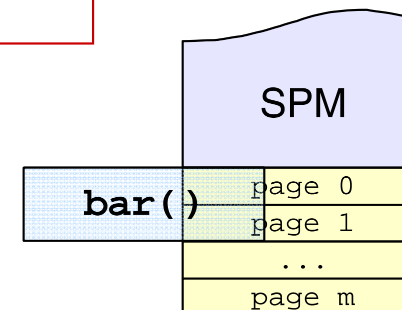
- Loads paged functions on-demand
- Evicts functions if necessary
- Modifies the contents of the paged function table
- Performs function lookups for control transfers to targets that cannot be resolved at compile-time
  - function pointers and returns

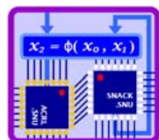



```
bl    pagetable[bar]
```

```
pagetable[bar]
  b page_manager

page_manager
  load bar
  modify pagetable[bar]
  b ex_buffer[bar]
```

```
pagetable[bar]
  b ex_buffer[bar]
```





Integer Linear Programming Formulation: 0-1 Knapsack Problem

- Given
  - A_i: the number of instruction fetches and read-only data word accesses located in a function f_i
  - S_i: the size of a function f_i in bytes
  - N: the number of functions in the application
  - S_{spm}: the SPM size in bytes
  - E_{spm}: the energy consumed to fetch an instruction (or a word) from the SPM
  - E_{ext}: the energy consumed to fetch an instruction (or a word) from the external memory
- Binary integer variables

$$I_{spm}(i) = \begin{cases} 1 & \text{if } f_i \text{ is placed in the SPM} \\ 0 & \text{otherwise} \end{cases} \qquad I_{ext}(i) = \begin{cases} 1 & \text{if } f_i \text{ is placed in the external memory} \\ 0 & \text{otherwise} \end{cases}$$

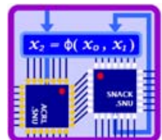
- Minimize

$$\sum_{i=1,N} (I_{spm}(i) \cdot A_i \cdot E_{spm} + I_{ext}(i) \cdot A_i \cdot E_{ext})$$

- Constraints

$$I_{spm}(i) + I_{ext}(i) = 1 \text{ for all } 0 < i \leq N$$

$$\sum_{i=1}^{N} I_{spm}(i) \cdot S_i \leq S_{spm}$$



Integer Linear Programming Formulation: Demand Paging

- Additional definitions
  - P: page size
  - I_{buffer}: the size of the execution buffer (a new general integer variable)
  - C_i: the number of calls to f_i
  - R_i: the number of returns to f_i
  - E_c: the energy consumed by extra instructions generated by the call expansion
  - E_r: the energy consumed by extra instructions generated by the return expansion
- Additional binary integer variable

$$I_{paged}(i) = \begin{cases} 1 & \text{if } f_i \text{ is in Paged and } S_i \leq I_{buffer} \times P \\ 0 & \text{otherwise} \end{cases}$$

- Minimize

$$\sum_{i=1,N} (I_{spm}(i) \cdot A_i \cdot E_{spm} + I_{ext}(i) \cdot A_i \cdot E_{ext} + I_{paged}(i) \cdot [A_i \cdot E_{spm} + Penalty_i])$$

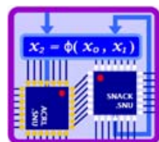
$$Penalty_i = (C_i + R_i)(E_{spm} + E_{ext})(\lceil S_i/P \rceil (P/4)) + C_i E_c + R_i E_r$$

- Additional constraints

$$0 \leq I_{buffer} \cdot P \leq S_{spm}$$

$$I_{spm}(i) + I_{ext}(i) + I_{paged}(i) = 1 \text{ for all } 0 < i \leq N$$

$$\sum_{i=1,N} I_{spm}(i) \cdot S_i \leq S_{spm} - I_{buffer} \cdot P$$



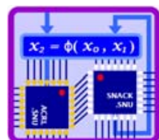
Evaluation Environment

- SNACK-armsim
  - Cycle-accurate ARM9 architecture simulator
- Benchmark Programs
  - synthetic (qsort, dijkstra, adpcm, sha, bitcount), epic/unepic, mpeg4 encoder/decoder, and FFT
- Energy Model
  - Systems with SPM

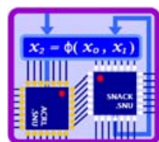
$$E = E_{core} + E_{spm} + E_{ext}$$
$$E_{core} = t \cdot P_{core}$$
$$E_{spm} = N_{spm} \cdot E_{sram}$$
$$\begin{aligned} E_{ext} = \; & t \cdot P_{static} \\ & + N_{ext_read_non-burst} \cdot E_{sdram_read_non-burst} \\ & + N_{ext_write_non-burst} \cdot E_{sdram_write_non-burst} \\ & + N_{ext_read_burst} \cdot E_{sdram_read_burst} \\ & + N_{ext_write_burst} \cdot E_{sdram_write_burst} \end{aligned}$$

  - With an instruction cache

$$\begin{aligned} E_{icache} = \; & N_{icache_hit} \cdot E_{icache} \\ & + N_{icache_miss} \cdot (E_{icache} + L_{icache} \cdot E_{sram}) \end{aligned}$$



Energy Consumption and Execution Time



Comparison to ICache

Core | SDRAM Standby | SDRAM Data | SDRAM Inst. | ICache | SPM

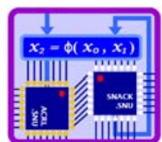
Normalized Energy Consumption

120%
100%
80%
60%
40%
20%
0%

2KB | Knap | PageLP | 2KB | Knap | PageLP | 4K | Knap | PageLP | 4K | Knap | PageLP | 8K | Knap | PageLP | 8K | Knap | PageLP

ICache | SPM 6K | ICache | SPM 6K | ICache | SPM 8K | ICache | SPM 8K | ICache | SPM 12K | ICache | SPM 12K

synthetic | fft | epic | unepic | mpeg4enc | mpeg4dec



Number of Memory Accesses

SDRAM data | SDRAM inst. | ICache | SPM data | SPM inst.

Accesses

100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%

SDRAM

ICache: 512B, 1KB, 2KB, 4KB, 8KB

SPM 4K: Knap, KnapLP, PageLP

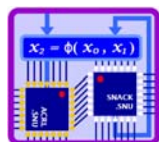
SPM 6K: Knap, KnapLP, PageLP

SPM 8K: Knap, KnapLP, PageLP

SPM 12K: Knap, KnapLP, PageLP

SPM 16K: Knap, KnapLP, PageLP

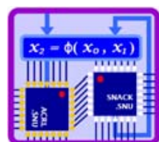
mpeg4enc



Software-Only Virtual Memory Summary

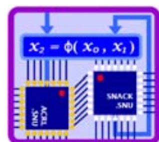
- The ICache size is about 20% of the executed code size
- Instruction cache vs. SPM with comparable die area
  - energy consumption reduced by 21.6%
  - execution time reduced by 20.2%

Application	Executed code size	ICache size	Comparable SPM size	Exec. Time to ICache (%)	Exec. Time to Knap (%)	Total Energy to ICache (%)	Total Energy to Knap (%)	Number of SPM pinned library/user functions	Number of paged library/user functions
synthetic	12KB	2KB	6KB	64.7	97.5	62.6	97.3	11/35	0/3
fft	14KB	2KB	6KB	70.1	78.2	75.3	77.7	14/6	0/0
epic	21KB	4KB	8KB	90.5	88.3	85.1	87.3	14/15	0/5
unepic	20KB	4KB	8KB	87.6	38.2	83.4	36.8	15/41	0/6
mpeg4enc	49KB	8KB	12KB	74.9	95.9	73.7	95.7	8/45	0/15
mpeg4dec	43KB	8KB	12KB	96.1	85.2	94.8	84.7	15/16	0/12
Average (geometric mean)				79.8	77.1	78.4	76.3		

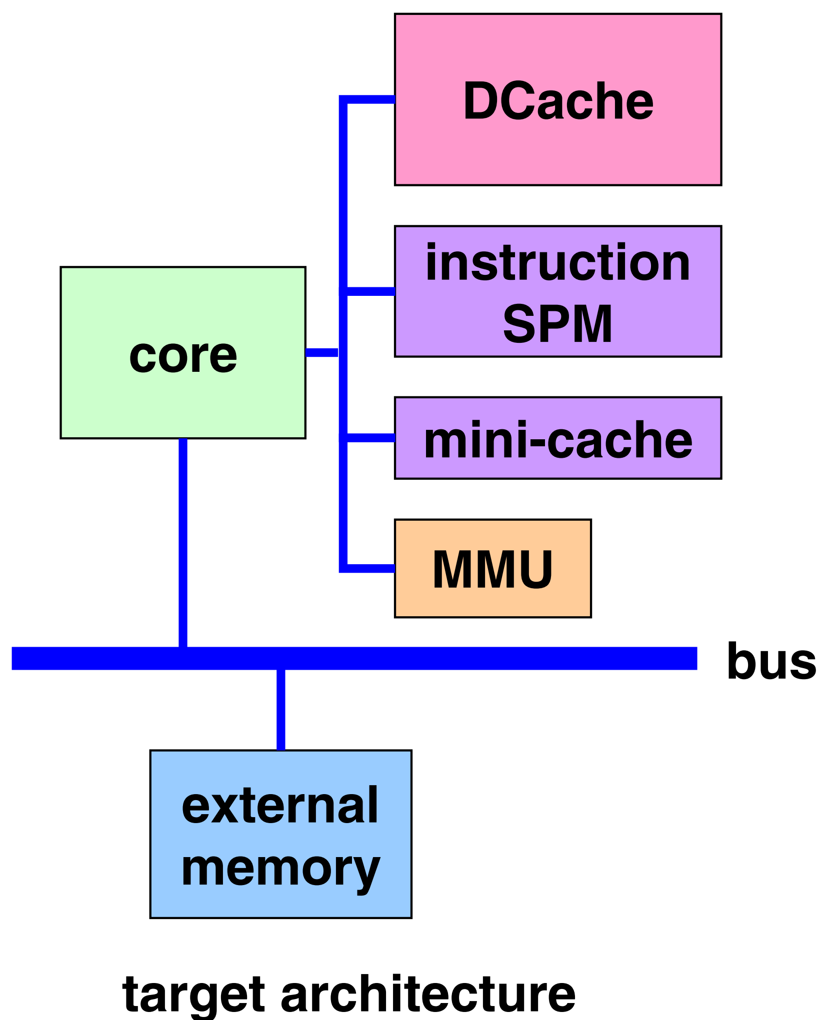


Outline

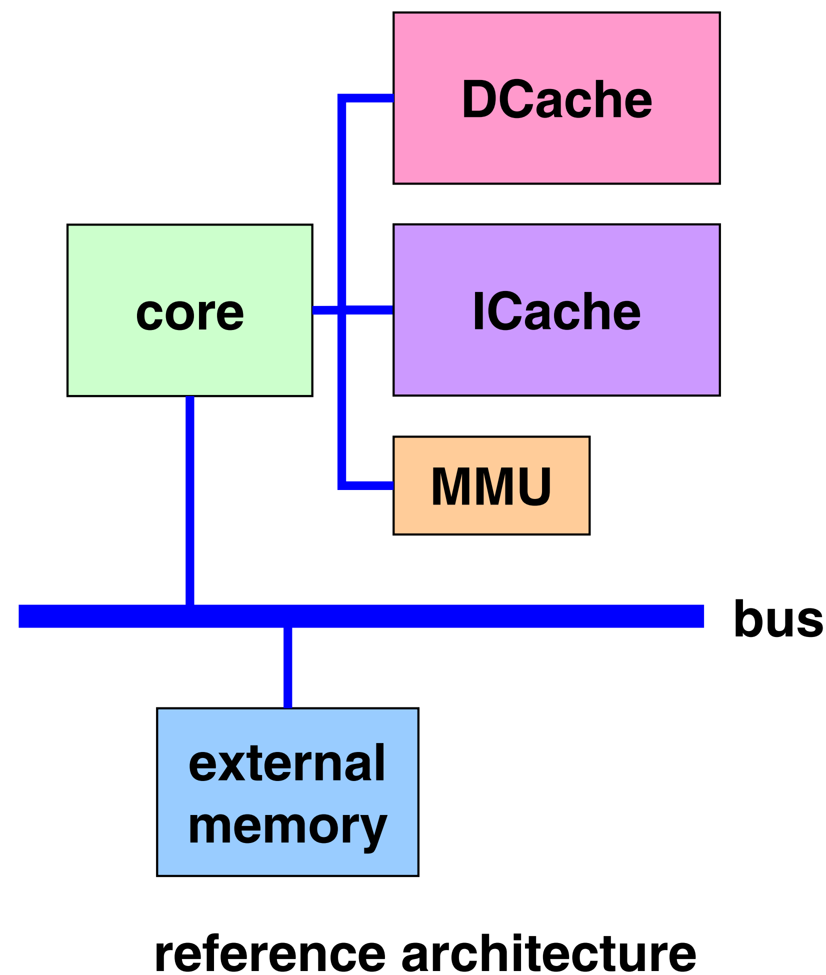
- Motivation
- Software-only virtual memory
- ***Virtual memory with MMU support***
- Conclusions



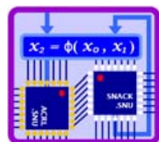
Target Architecture



target architecture

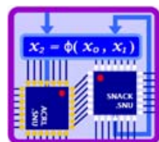


reference architecture



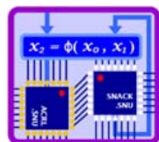
Related Work

- Existing Research
  - SPM size must be known at compile time
  - one (or a fixed set of) process(es)
  - no virtual memory
- Ngyuen et al. (CASES'05)
  - Memory allocation for embedded systems with a compile-time unknown scratch-pad size
  - static approach
  - decision made when the application is loaded
- Shrivastava et al. (CASES'05)
  - Compilation techniques for energy reduction in horizontally partitioned cache architectures
  - XScale: big main data cache + 2KB minicache
  - allocate data objects to one of the caches to save energy



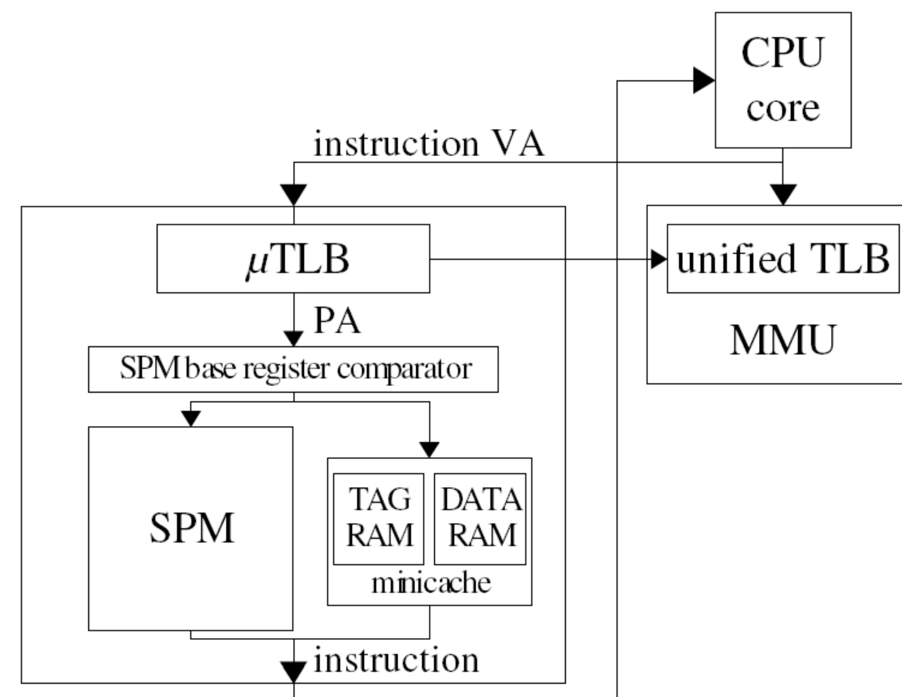
SPM Management using an MMU

- Hardware
  - The on-chip instruction cache is replaced by a physically-addressed SPM plus a small instruction minicache
- At compile-time
  - Code is classified based on a trace analysis
    - ***uncached***, ***cached***, **and** ***paged*** regions
  - Paged code is clustered into pages based on temporal locality
- At run-time
  - (Un)cached code regions are mapped (un)cacheable
  - Paged code regions are not mapped
  - The SPM manager intercepts MMU page fault exceptions, loads the requested page into the SPM, modifies the corresponding page table entry, and resumes control to the application

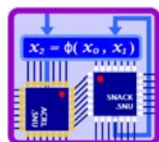


Proposed Memory Architecture

- Existing architectures
  - Do not guarantee 1-cycle access of SPM, or
  - Consumes more energy by accessing the SPM and cache simultaneously
- Horizontally-partitioned on-chip memory subsystem
  - μTLB
  - Scratchpad memory
  - Direct-mapped minicache
  - up to ~500MHz, 1-cycle access



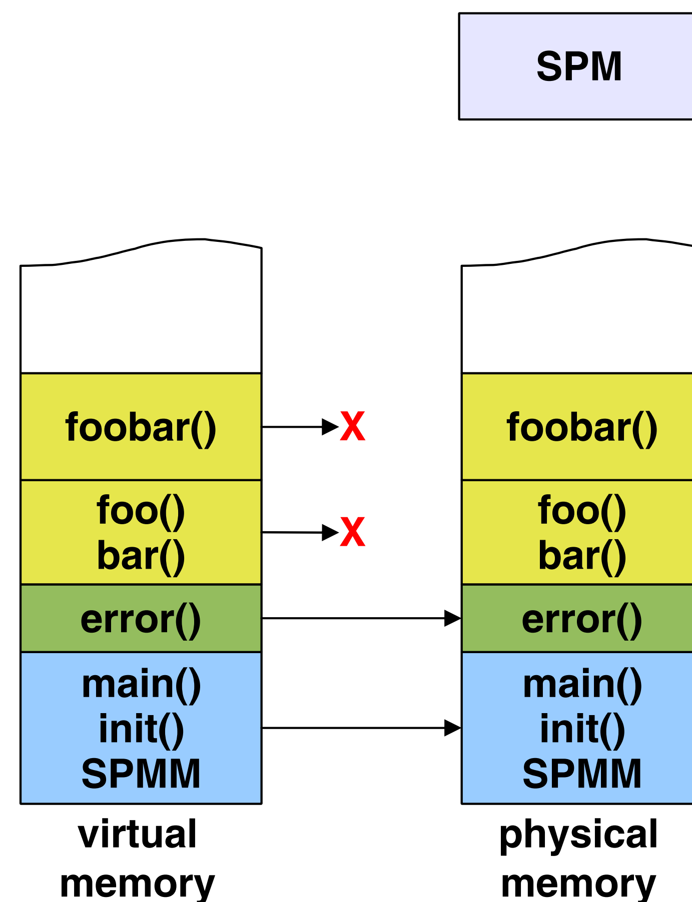
16-entry μTLB	1.14ns
16KB SPM	0.81ns
1K direct-mapped cache	0.81ns
Total latency (μTLB -> SPM or cache)	1.95ns

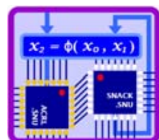


Scratchpad Memory Management

```
spmm_loader() {
    load SPM-optimized binary;
    setup page tables;
    start process;
}
```

- Runtime SPM Manager (SPMM)
  - The loader
    - detects SPM-optimized binaries
    - sets up the page tables accordingly

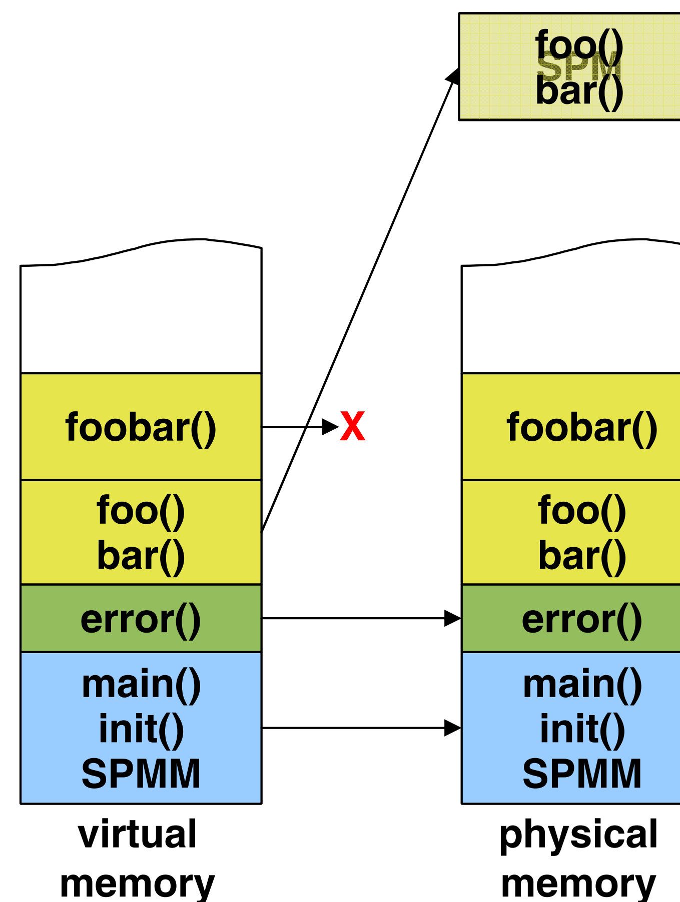


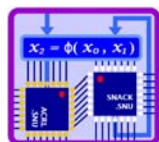


Scratchpad Memory Management

```
int main(void) {
       init();
PC→   return foobar(bar());
}

int bar() { ... }

int spmm_pgflt(uint adr) {
  ...
  copy page to SPM;
  modify page table entry;
  resume aborted inst;
}
```

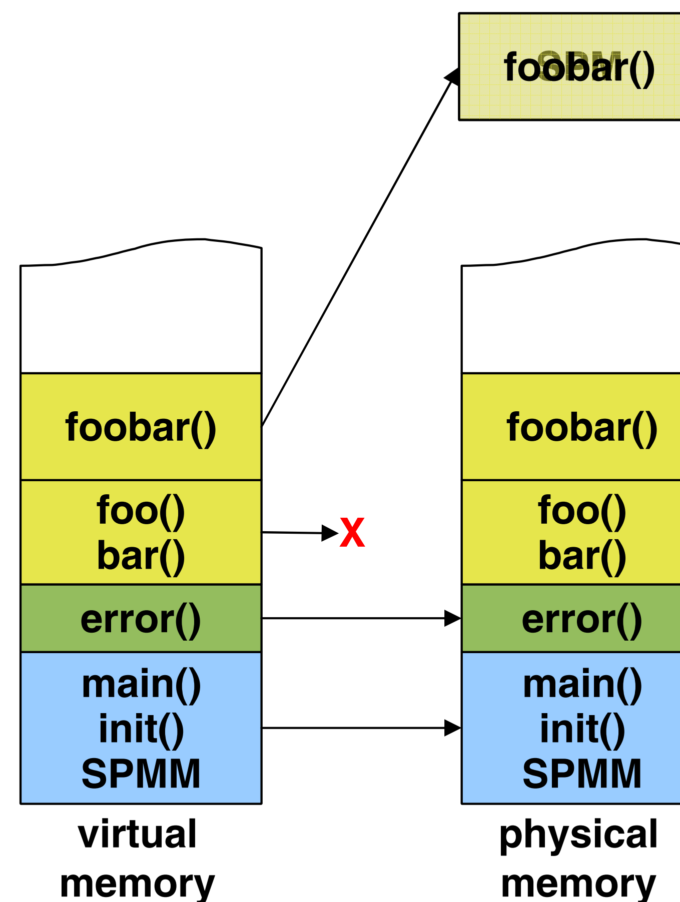
- The SPMM handler
  - copies code pages to the SPM on demand
  - intercepts page fault exceptions generated by the MMU

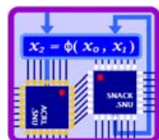




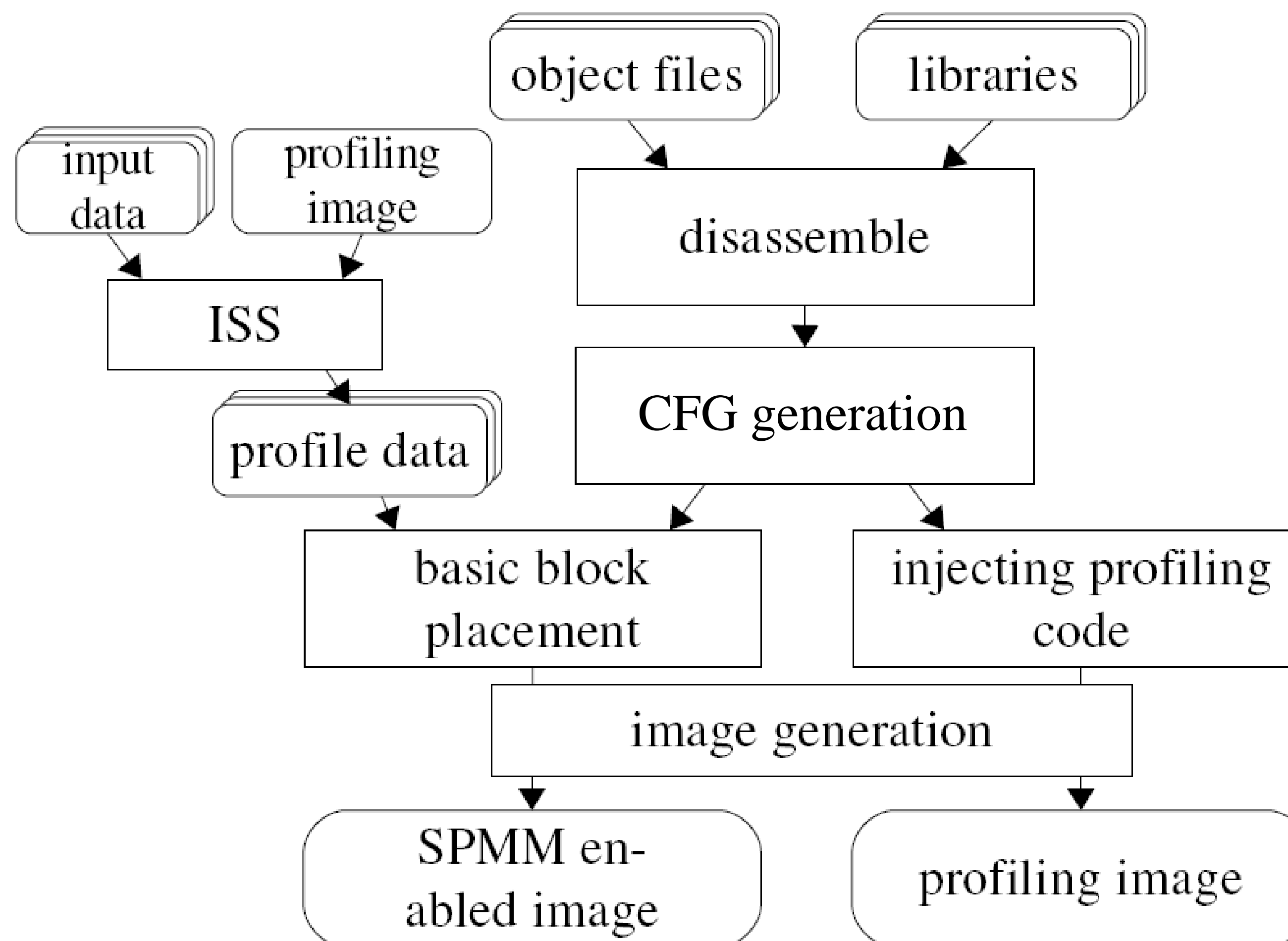
Scratchpad Memory Management

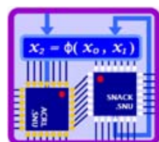
```
    int main(void) {
      init();
PC→   return foobar(bar());
    }

    int foobar(int n) { ... }

    int spmm_pgflt(uint adr) {
      disable PTE of loaded page;
      copy page to SPM;
      modify page table entry;
      resume aborted inst;
    }
```





SNACK-pop: Post-pass Optimizer





SNACK-pop: Code Classification

- For each basic block b_i, the code region is determined by

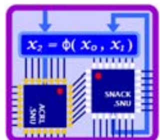
$$Loc_i = \begin{cases} uncached & \text{if the code is executed less than once} \\ cached & \text{if } E_{cached}(b_i) < E_{paged}(b_i) \\ paged & \text{otherwise} \end{cases}$$

with

$$\begin{aligned} E_{paged}(b_i) &= A_i E_{spm} + M S_i (E_{ext} + E_{spm}) \\ E_{cached}(b_i) &= A_i (E_{cache} + m_{cache} E_{miss}) \end{aligned}$$

where

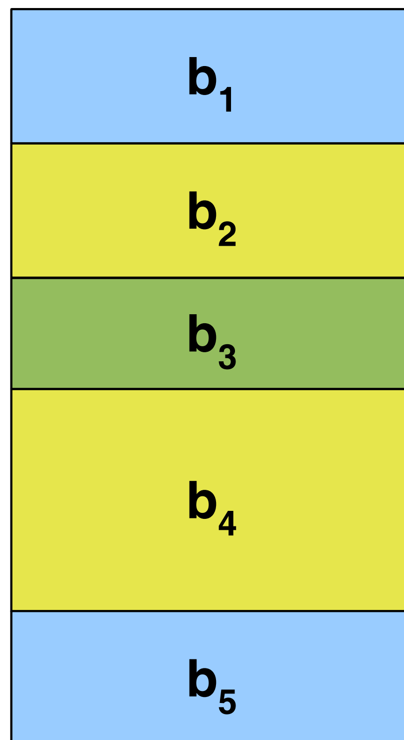
- A_i: number of instructions fetched
- S_i: size of block i
- M: average number of page misses
- m_{cache}: cache miss ratio
- $E_{spm/ext/cache/miss}$: SPM/external memory/cache access/cache miss energy



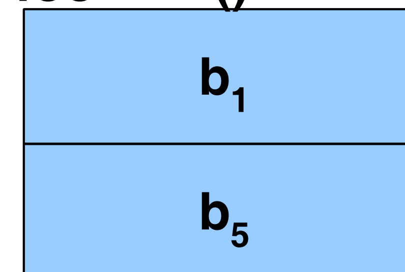
SNACK-pop: Dividing Functions

paged
uncached
cached

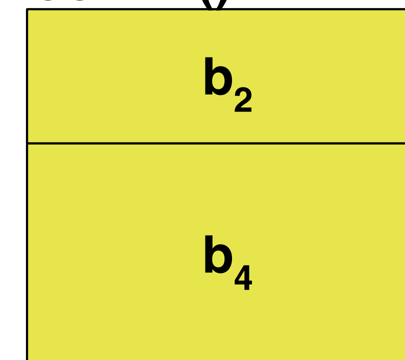
foo()



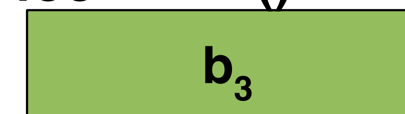
$foo^{cached}()$

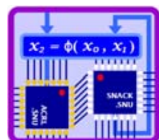


$foo^{paged}()$



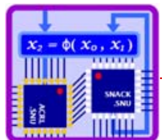
$foo^{uncached}()$





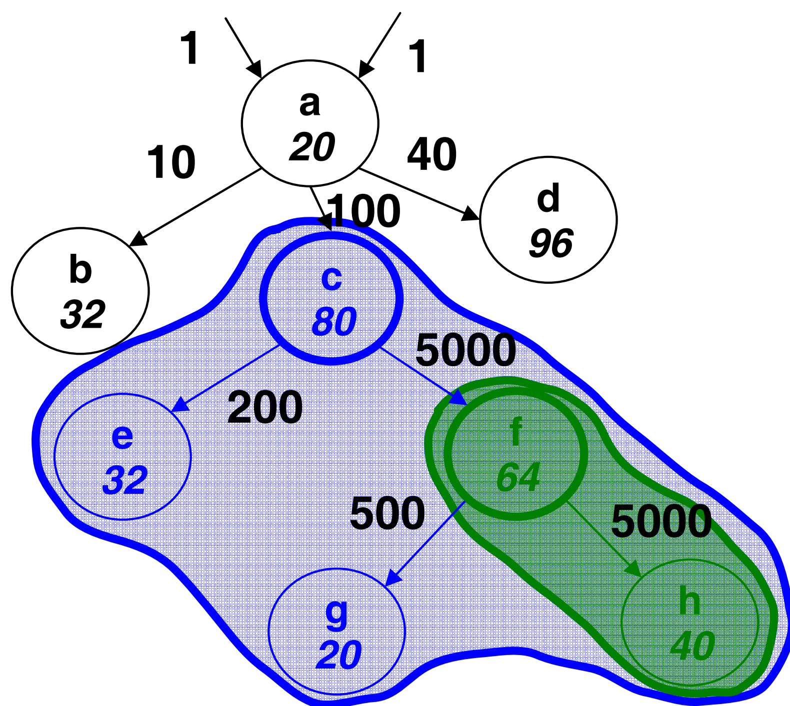
SNACK-pop: Code Arrangement

- To achieve maximum performance,
  - Paged code must be arranged in a particular way to allocate as few pages as possible
    - cluster temporally local code together into as few pages as possible
    - optimal solution: harder than Knapsack

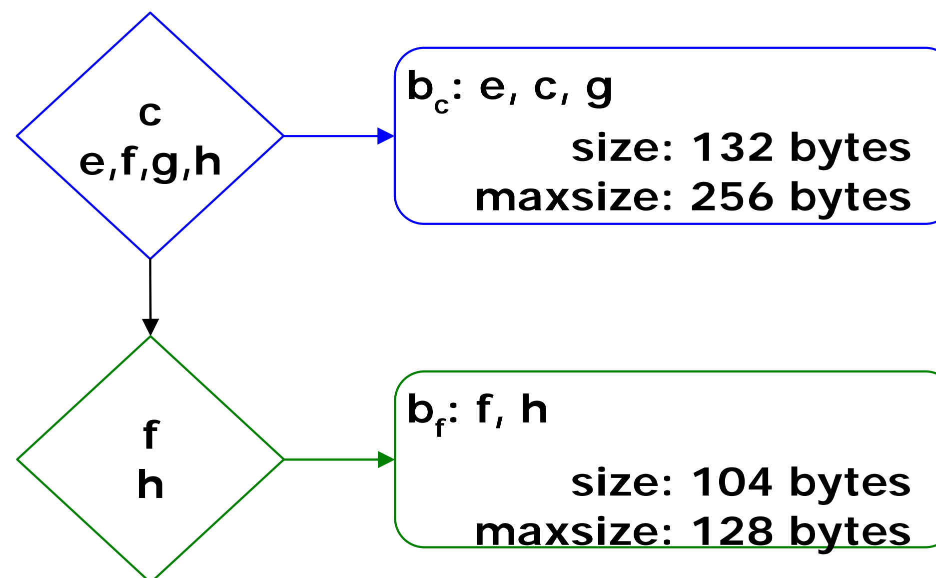


SNACK-pop: Code Arrangement (contd.)

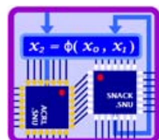
- Step 1: Loop detection in the trace level
- Step 2: Compute the loop control graph and allocate functions



dynamic call graph

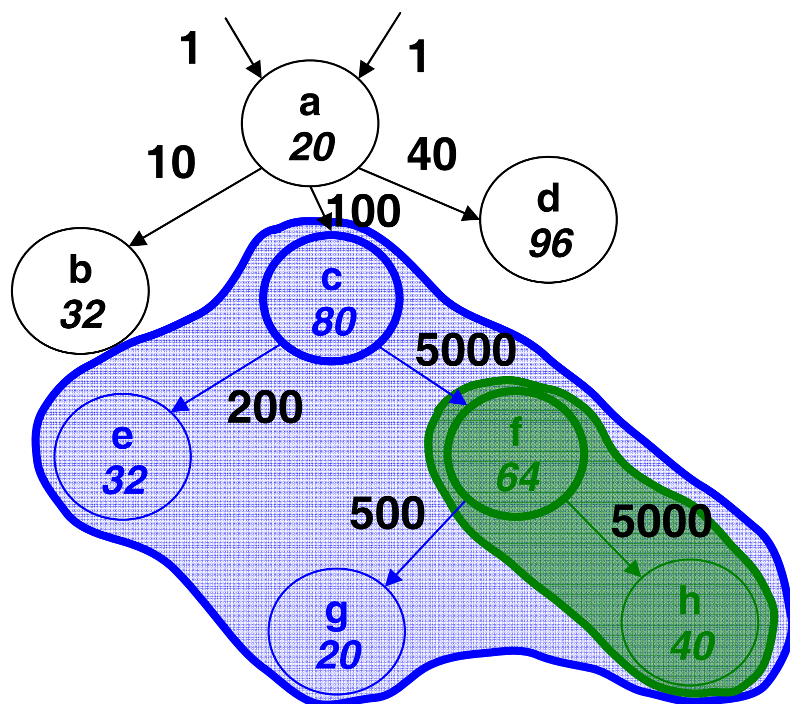


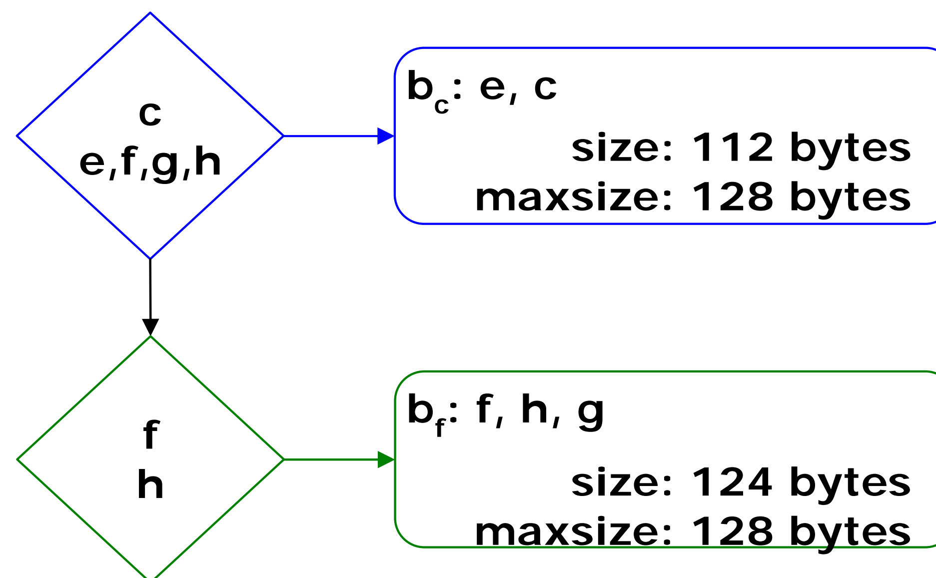
pagesize = 128

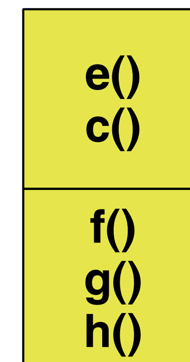


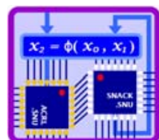
SNACK-pop: Code Arrangement (contd.)

- Step 3: Pack loop bins
  - push functions from outer loops into inner loops



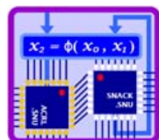






Evaluation Environment

- SNACK-armsim
  - Cycle-accurate ARM9 core architecture simulator with horizontally-partitioned memory architecture as proposed
- Benchmark Programs
  - combine (qsort, dijkstra, adpcm, sha, bitcount), FFT, epic/unepic, mp3 decoder, mpeg4 encoder/decoder



Energy Model

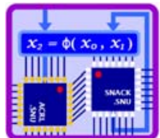
$$T_{total} = \frac{core\ clocks}{core\ frequency}$$

$$E_{total} = E_{SPM} + E_{icache} + E_{dcache} + E_{ext_static} + E_{ext_dynamic}$$

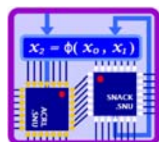
$$E_{cache} = e_{cache}(hit + \frac{linesize}{4} \cdot miss)$$

$$E_{SPM} = e_{SPM}(read + write)$$

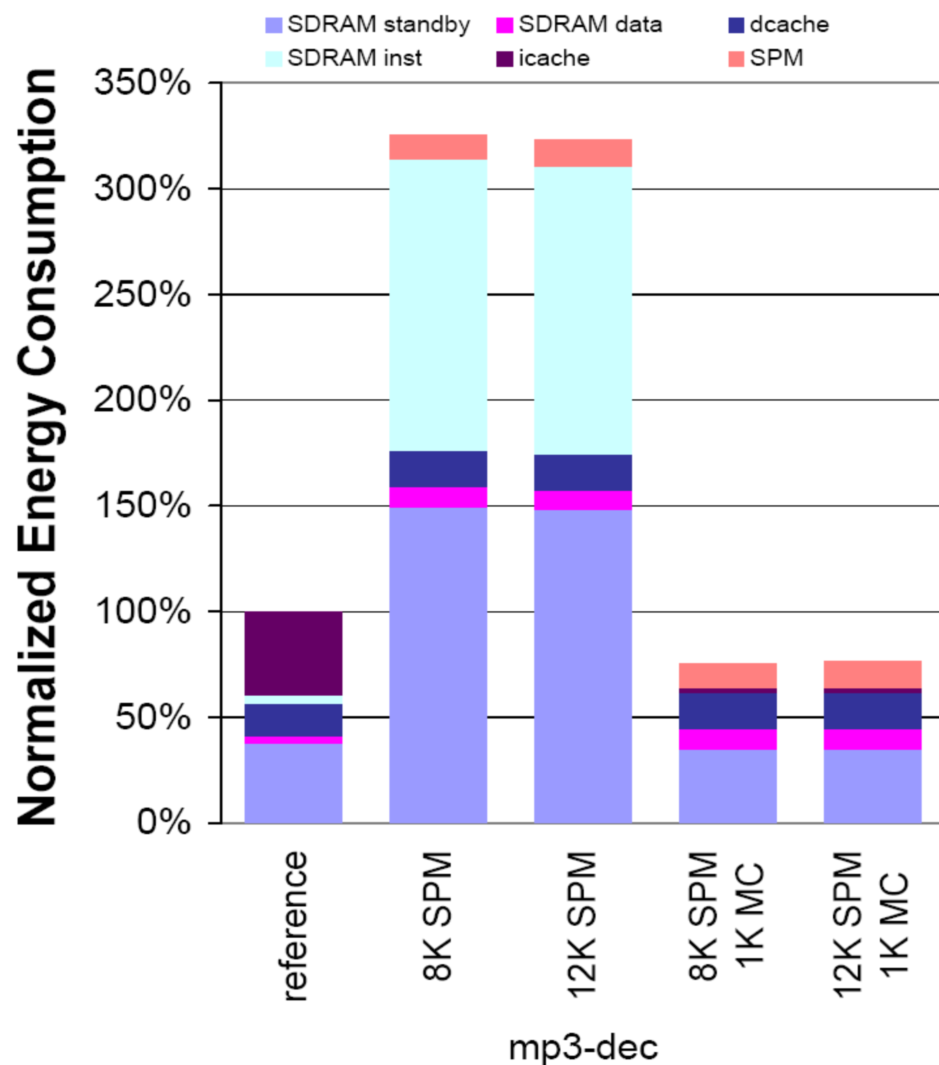
$$E_{ext_static} = P_{standby} \cdot T_{total}$$



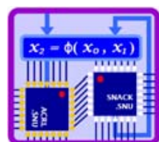
Performance and Energy Consumption



Effect of the Minicache

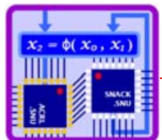


	no minicache		minicache	
	time	energy	time	energy
combine	27.94	16.25	1.21	0.92
fft	10.10	7.68	3.92	3.35
epic	29.00	18.25	8.67	6.52
unepic	22.26	14.57	6.15	4.62
mp3-dec	18.15	12.16	4.15	3.04
mpeg4-dec	16.51	11.55	5.00	3.89
mpeg4-enc	15.71	11.07	2.28	1.85
	18.89	12.64	3.82	2.97



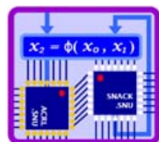
Virtual Memory with MMU Support
Summary

- Comparing SPM+minicache vs. ICache (20%-30% of the executed code size) with comparable die area requirements, we achieve
  - 12% improvement in runtime performance
  - 33% reduction in energy consumption



Conclusion

- Based on post-pass optimizations and demand paging
  - Software-only virtual memory
    - Fully automated dynamic code placement technique with natural loop extraction
    - At run-time, function calls/returns to paged functions are intercepted by the page manager
  - Virtual memory with MMU support
    - Horizontally-partitioned memory architecture composed of a big SPM and a small minicache
    - On-demand paging using the MMU's page fault exception
- Can replace the instruction cache
- Effective to reduce energy consumption without loosing performance



Future Work

- Virtual memory environments for data side
- Virtual memory environments for multitasking
- Predictable scratchpad memory management