Memory Hierarchy Optimizations with Compilers/Software

Jaejin Lee

Advanced Compiler Research Laboratory
School of Computer Science and Engineering
Seoul National University
jlee@cse.snu.ac.kr
http://aces.snu.ac.kr/~jlee

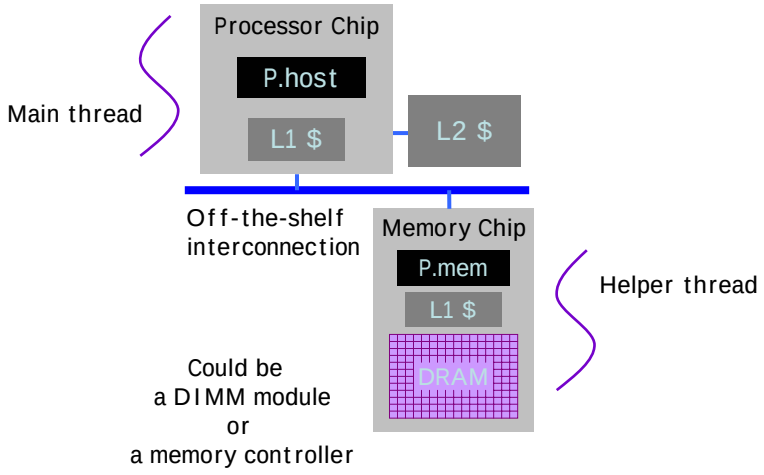
Outline

- Heterogeneous Multithreading (Helper Threading)
  - Intelligent Memory
  - Coexecution
  - Prefetching
- Compiler-Assisted Demand Paging
- Wrap-up

Memory Wall Problem

- The performance gap of processors and memory
  - Microprocessor performance has been improving at a rate of 60% per year.
  - The access time to DRAM has been improving at a rate of less than 10% per year.
- The performance of applications is dominated by memory.
- Thousands of papers.

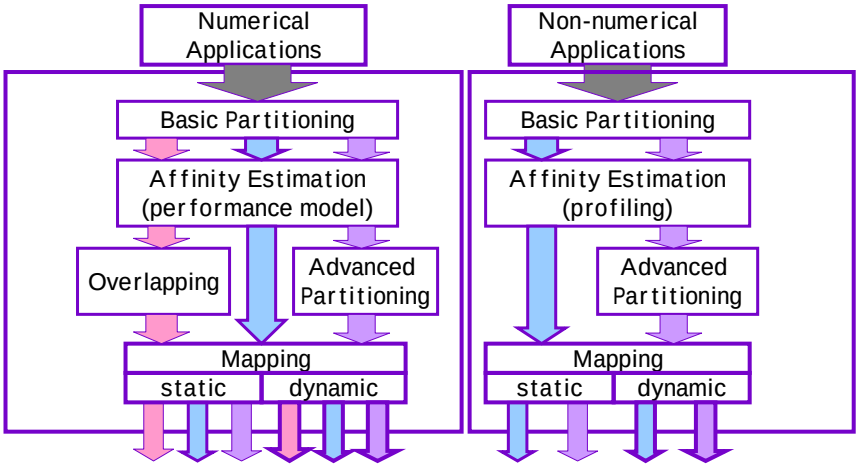
The Intelligent Memory Architecture



Co-execution

- Using a compiler,
  - Partition code into compute-/memory-intensive sections (so called modules).
    - Performance prediction
  - The memory-intensive sections are wrapped into a helper thread.
  - Statically/dynamically map the sections to the best processor.

Overview of the Co-execution Algorithm



Static Mapping

- Performance model (numerical apps)
  - Execution time = T_{comp} + $T_{memstall}$
  - Stack distance model for the number of misses

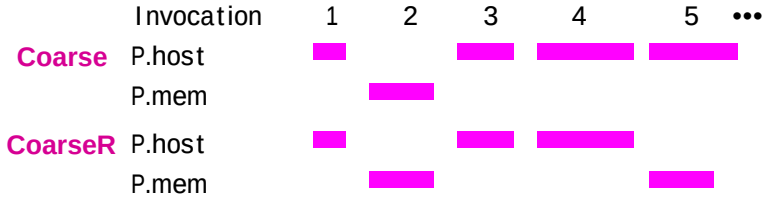
$$T_{comp} = \max\left(\frac{T_{int}}{N_{int}}, \frac{T_{fp}}{N_{fp}}, \frac{T_{ldst}}{N_{ldst}}\right) + T_{other}$$

$$T_{memstall} = \sum_{i \in caches} miss_i \bullet penalty_i$$

- Profiling (non-numerical apps)
  - Gather execution time and the number of invocations for all modules and subroutines.

Dynamic Mapping

- Decision runs at runtime to determine affinity
- Coarse and CoarseR
  - Decision runs are module invocations



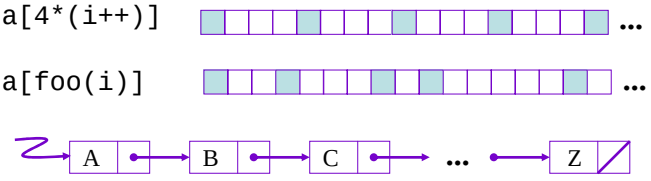
Overall Speedups for Co-execution

- Our co-execution algorithm delivers speedups that are comparable to the ideal speedup.

Apps.	P.host(alone) /AdvCoarseR	P.host(alone) /OverDyn	Amdahl's 2 P.hosts	2-processor SGI
Swim	1.67	2.71	2.00	1.85
Tomcatv	1.17	1.60	1.67	1.44
LU	1.26	1.22	1.04	0.99
TFFT2	1.42	1.22	1.91	0.80
Mgrid	1.05	1.55	1.94	1.47
Average	1.31	**1.66**	**1.71**	1.31
Bzip2	1.37	-	1.01	0.99
Mcf	1.37	-	1.01	1.00
Go	0.97	-	1.01	0.57
M88ksim	1.01	-	1.03	1.00
Average	**1.18**	-	**1.02**	0.89

Correlation Prefetching in Software

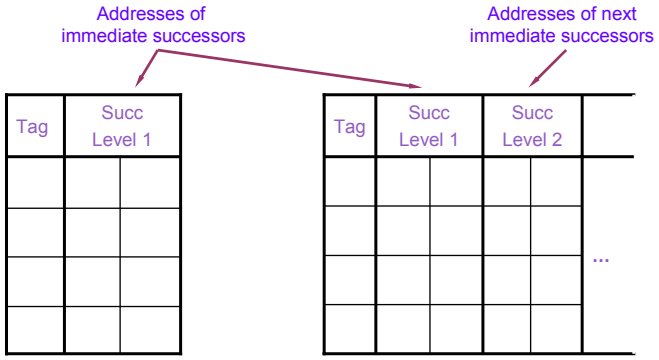
- New correlation prefetching in software using the memory thread.
- Records sequences of miss addresses in a correlation table.
- When the head of a sequence is seen, prefetch the rest.



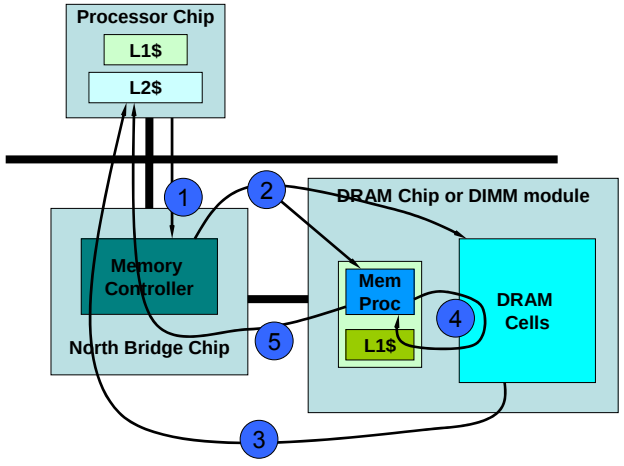
Correlation Table

Basic Organization
(Joseph & Grunwald)

Advanced Organization

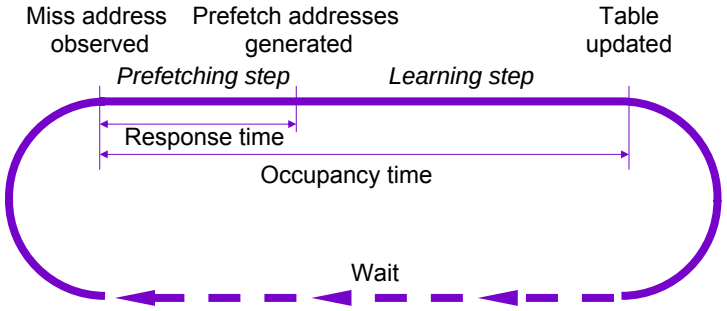


Our Scheme

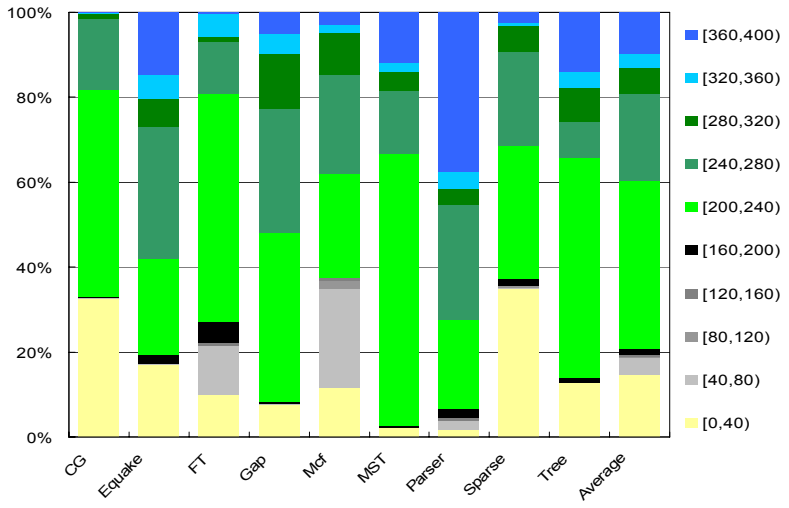


The Mechanism of the Memory (Helper) Thread

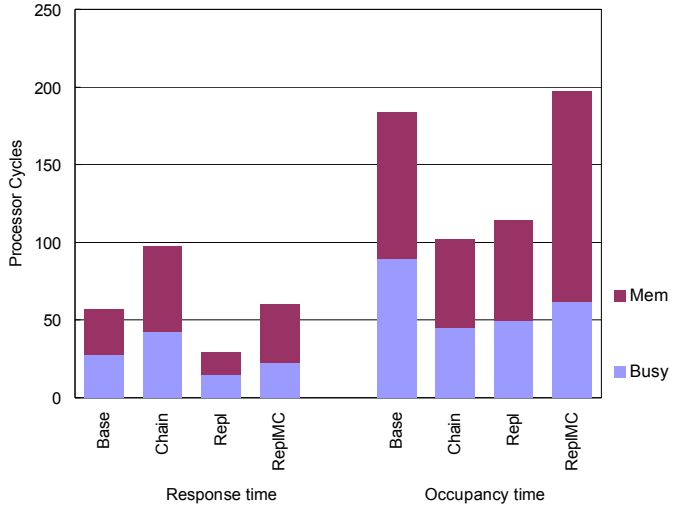
- Requirements:
  - Low response time
  - Occupancy time < miss distance



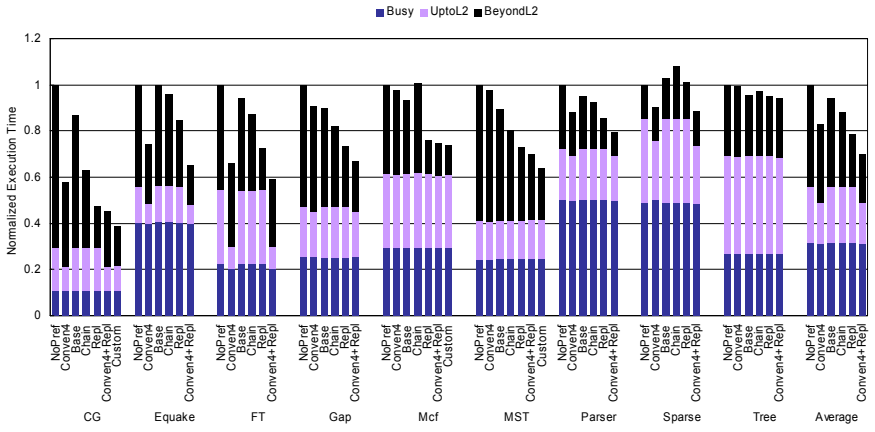
Miss Distance



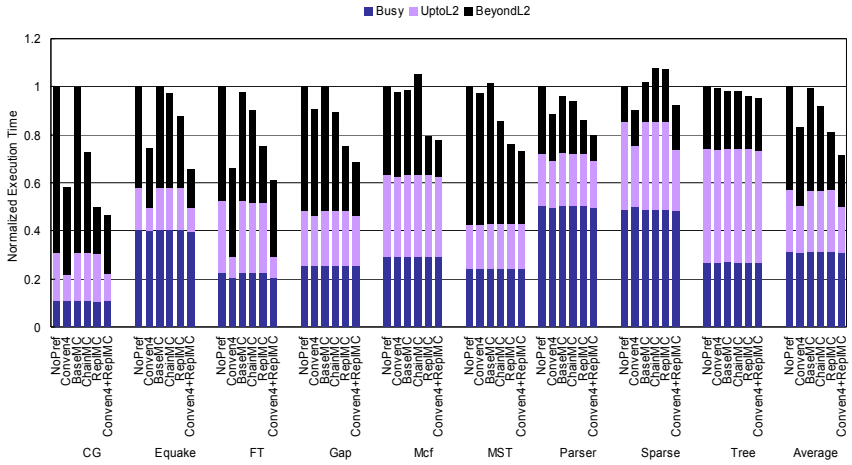
Response and Occupancy Time



Execution Time in DRAM



Execution Time in MC



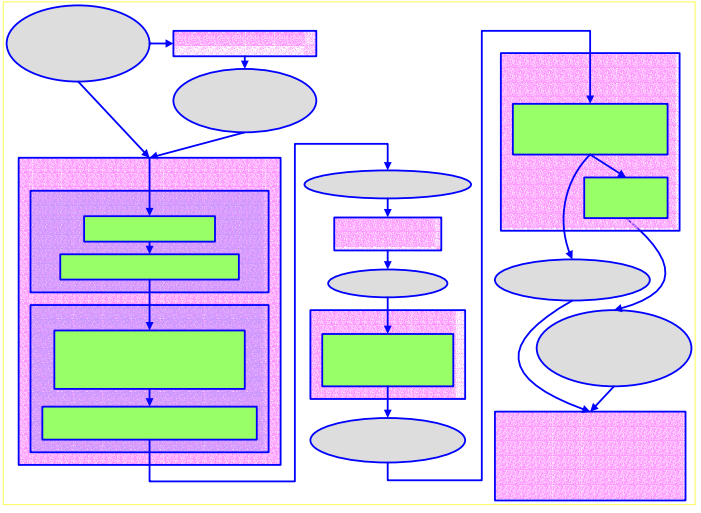
Active Prefetching

- The helper thread runs the skeleton of the original code
  - Address computation
  - Prefetch instructions
- More accurate prefetches
- The helper thread should be faster than the original code
- Synchronization overhead

Outline

- Heterogeneous Multithreading (Helper Threading)
  - Intelligent Memory
  - Coexecution
  - Prefetching
- Compiler-Assisted Demand Paging
  - Motivation
  - Framework
  - Example
  - Performance Results
- Wrap-up

Seoul **N**ational university **A**dvanced **C**ompiler tool **K**it (***SNACK***)



SNACK Components

- SNACK-cc: a C compiler for embedded systems
- SNACK-c2c: C-to-C translator
- SNACK-asm: assembler
- SNACK-link: linker
- SNACK-pop: post-pass optimizer
- SNACK-jvm: embedded Java VM (planned)

Goals

- High performance
- Small code size
- Low power/energy

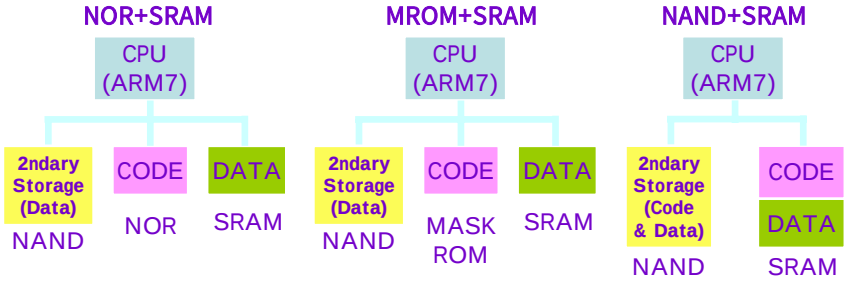
Using *SNACK*,

- Memory hierarchy optimization
  - Scratchpad memory optimization
  - Demand paging
  - Cache design
- Code generation for ARM and DSPs
- Embedded Java VM research
- Java memory model

Compiler-Assisted Demand Paging

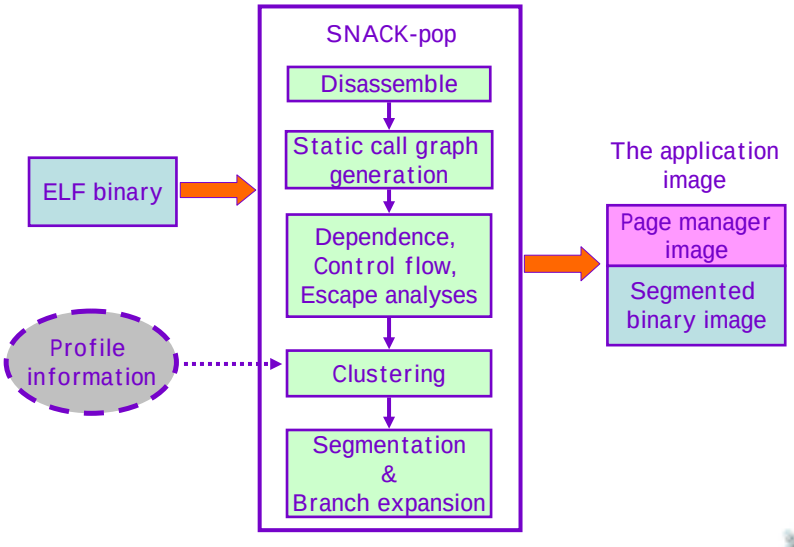
- Motivation
  - The size of code is getting bigger → the cost of (code) memory is getting higher → energy consumption is getting bigger
- How to reduce *code-memory size* with comparable performance and energy consumption for low-end embedded systems?
  - Demand paging, dynamic loading
  - No virtual memory

Memory Architecture



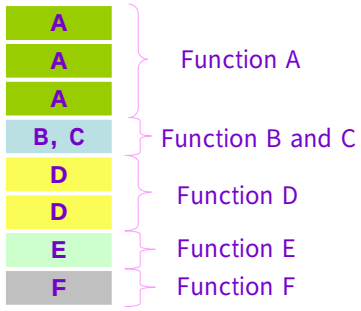
	NOR+SRAM	MROM+SRAM	NAND+SRAM
Performance	**Poor**	**Poor**	**Good**
Code size (for the same cost)	**Big**	**Big**	**Small (big?)**
Upgrade	**Easy**	**Difficult**	**Easy**

Framework



Segmented Paging

- A segment consists of at least one page.
- At least one function per page or segment.
- Page size = the block size of the NAND flash memory



Page Manager

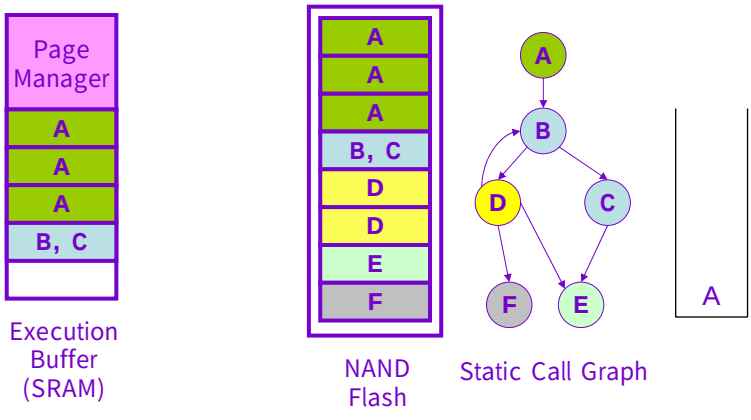
- Contains a page table, a buffer page table and a segment table.
  - Generated by the post-pass optimizer
- When a page hit occurs, simply branches to the page on the buffer.
- Otherwise, load the corresponding segment in to the execution buffer, and then branches to the page.

Call/Return Expansion

- Basically, a branch to another segment are expended to the calls to the page manager
- Many patterns

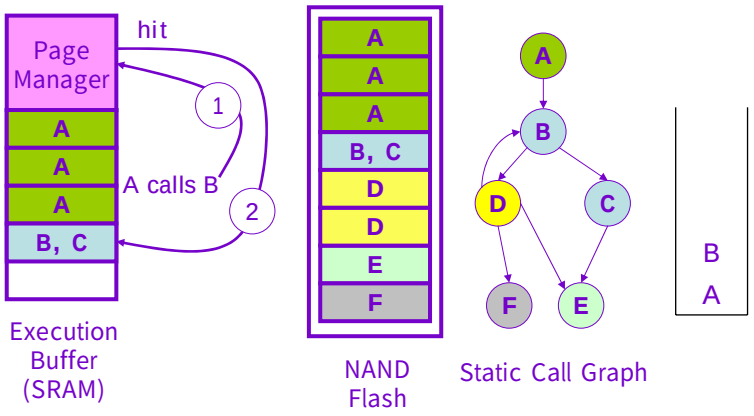
```
foo:                          foo:
...                           ...
    BL    bar      ⇒              BL  T1
...                           ...
    pc ← lr                       B   page_manager
                              T1:
                                  r0 ← bar's absolute address
                                  B   page_manager
```


Example (1)



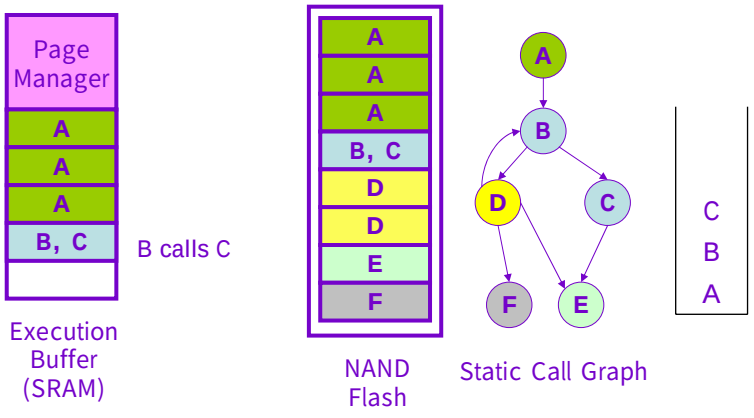
Calling sequence: A

Example (2)



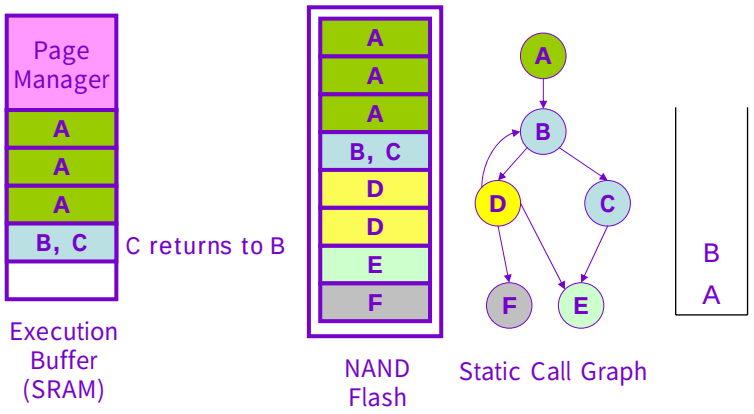
Calling sequence: A B

Example (3)



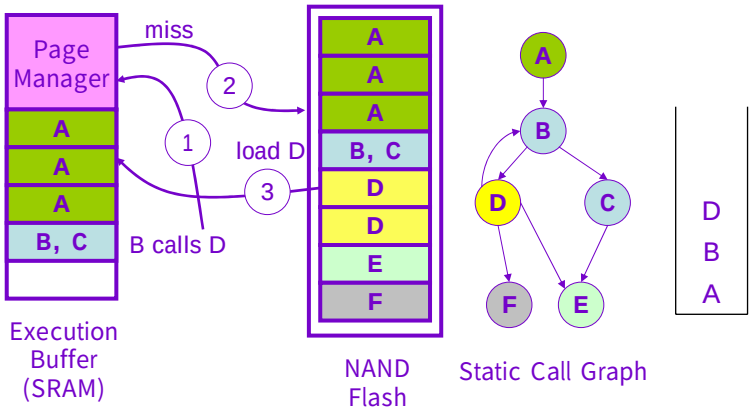
Calling sequence: A B C

Example (4)



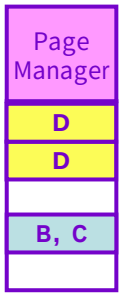
Calling sequence: A B C

Example (5)

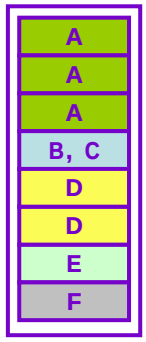


Calling sequence: A B C D

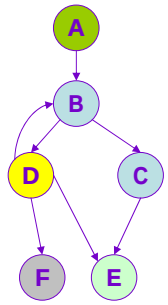
Example (6)



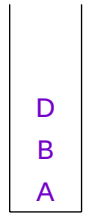
Execution Buffer (SRAM)



NAND Flash



Static Call Graph



Calling sequence: A B C D

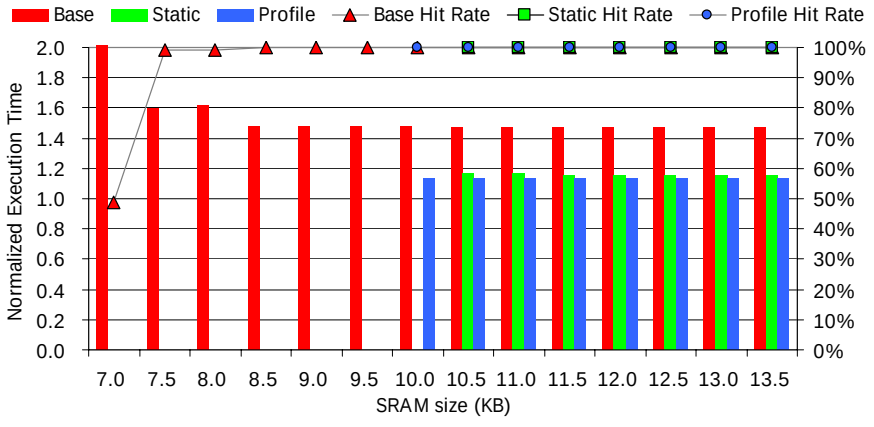
Strategies

- *Base*: segmented paging with round-robin replacement
- *Static*: clustering with the static call graph
  - Nodes with multiple parents pinned in the resident segment.
  - Parents and children together
- *Profile*: pinning some segments in the SRAM using profile data (dynamic call graph)

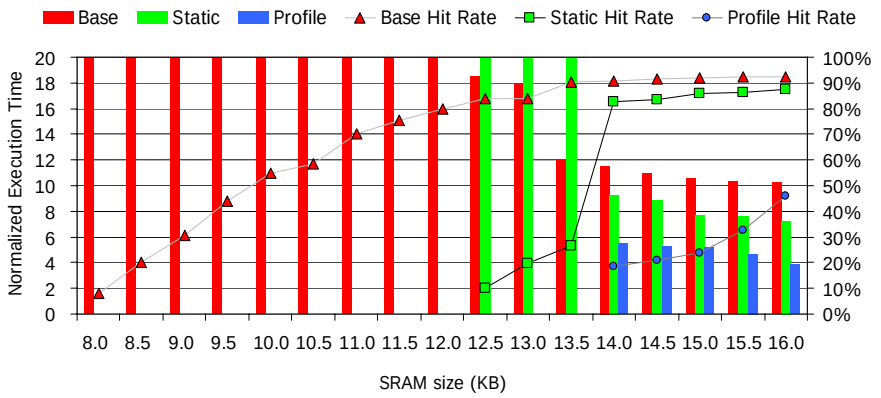
Simulation Environment

- ARMulator, ARM ADS 1.2
  - ARM7, cycle accurate, no caches
  - 20Mhz
  - NAND flash read latency is fully incorporated in the simulation.
    - NAND cell → page register: 10us
    - Reading 256 word (16bit) from page register: 256 x 50ns=12.8 us
- Benchmark Programs
  - From MI bench and Media Bench
    - Combine (qsort, dijkstra, adpcm, sha, bitcount), FFT, Epic, Unepic, MP3

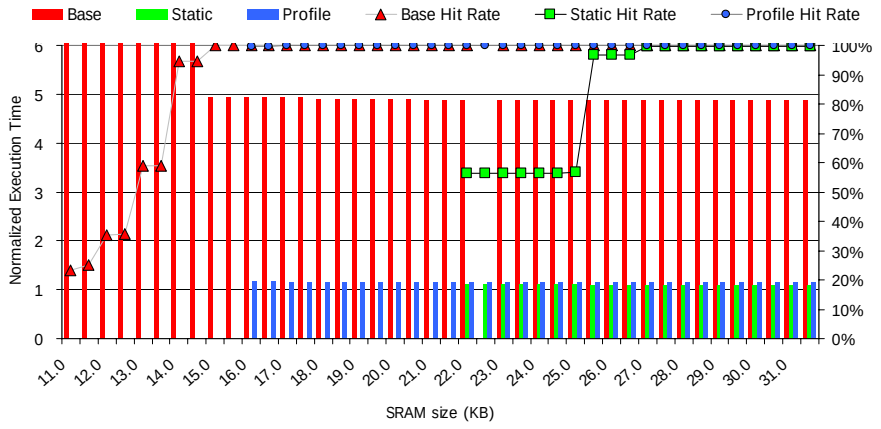
Combine (14KB)



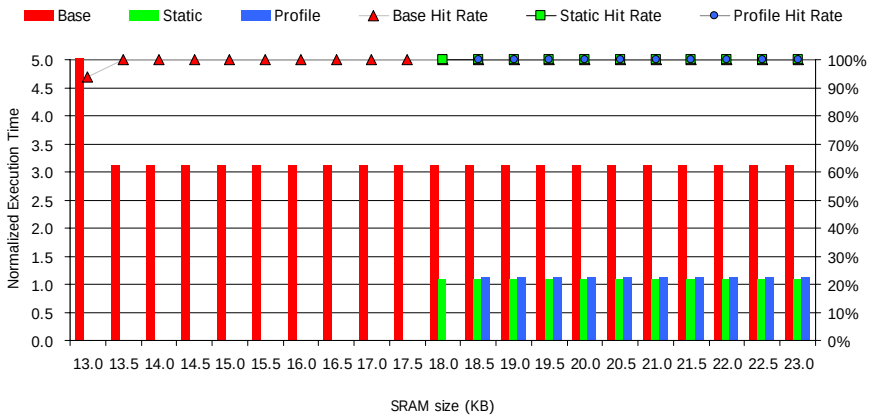
FFT (16KB)



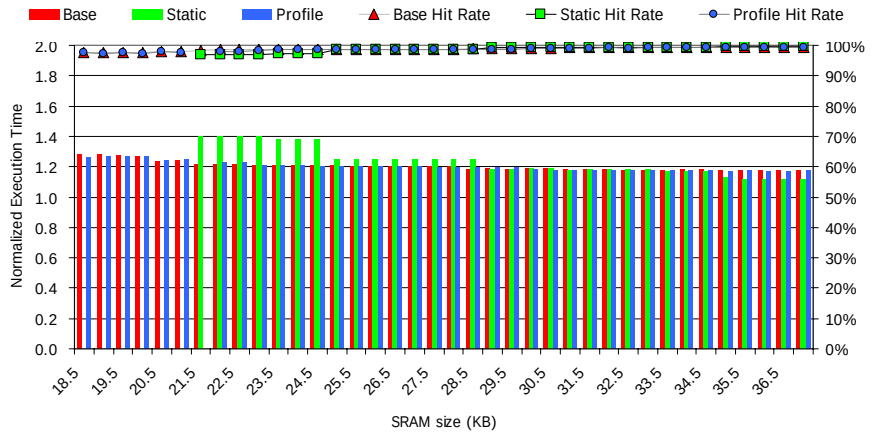
EPIC (32KB)



UNEPIC (23KB)



MP3 (37KB)



Summary

- 33% of SRAM reduction on average with less than 20% performance degradation and 8% more energy consumption.
- The designer can select the SRAM size depending on their cost, energy, and real-time requirements.

Application	Original Code Size	SRAM size with paging	Normalized Energy Consumption
Combine	14KB	10KB (71%)	1.16
FFT	16KB	16KB (100%)	1.00
Epic	32KB	16KB (50%)	1.08
Unepic	23KB	18KB (78%)	1.02
MP3	37KB	22KB (59%)	1.14
Average	24.4KB	16.4KB (67%)	1.08

Challenges

- How to reduce the number of page manager calls (branching overhead)?
- How to reduce the number of page/segment misses?
- Aim at embedded systems with MMU and RTOS support
- Speed differentiated (a.k.a. scratchpad memory) on-chip memory optimization
- Data paging

References

- "Automatically Mapping Code in an Intelligent Memory Architecture", Jaejin Lee, Yan Solihin, and Josep Torrellas. *HPCA 2001*
- "Using a User-Level Memory Thread for Correlation Prefetching", Yan Solihin, Jaejin Lee, and Josep Torrellas. *ISCA 2002*
- “Compiler-Assisted Demand Paging for Embedded Systems with Flash Memory”, Chanik Park, Junghee Lim, Kiwon Kwon, Jaejin Lee, and Sang Lyul Min, *EMSoft 2004*