t-kernel – Reliable OS support for WSN

L. Gu and J. Stankovic, appearing in 4th Proc. of the 4th ACM Conference on Embedded Networked Sensor Systems, Oct. 2006.

Best paper award.

Motivation

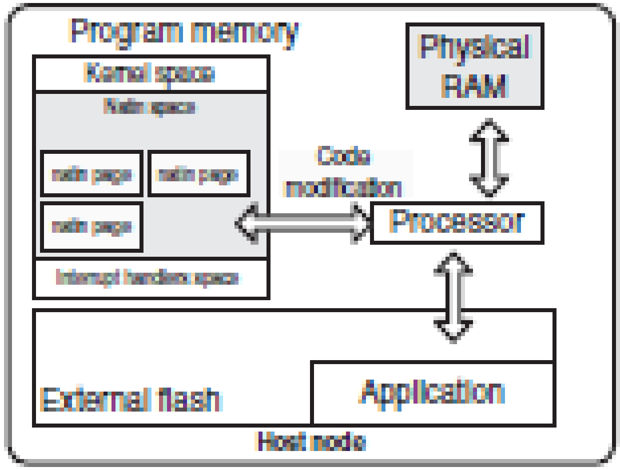
- Wireless sensor networks (WSNs) with
  - Resource constrained embedded microcontrollers
  - Complex application requirements
- OS support is very limited; applications (developers) could benefits from
  - OS protection
  - Virtual memory
  - Preemptive scheduling
- But microcontrollers don’t have HW support for this
  - E.g. privileged execution, virtual address translation, memory protection
- *How can we efficiently provide such support w/o hardware help?*

Context – Complex apps requirements

- VM - VigilNet – large-scale surveillance
  - 30 middleware services & 40K SLC
  - Using overlay in absence of VM is not really an answer
    - Application specific, inefficient, labor intensive, error-prone
- OS Control - Extreme scaling
  - To ensure the OS gets the CPU back, grenade timer or periodic reboot
    - Coarse control granularity
    - Applications must adapt to this
    - Long time w/o OS control to reduce too frequent restarts

Approach - Naturalization

- Minimum assumptions (REM)
  - Reprogrammable – you can write something into mem. & execute it
  - External nonvolatile storage
  - Some RAM available (4KB)
- Load-time code modification – *naturalization*
  - Done on demand, one page at a time
  - Output – a cooperative program supporting OS protection, VM & preemptive scheduling
- Paging
  - Storage management
- Dispatcher
  - Controls execution

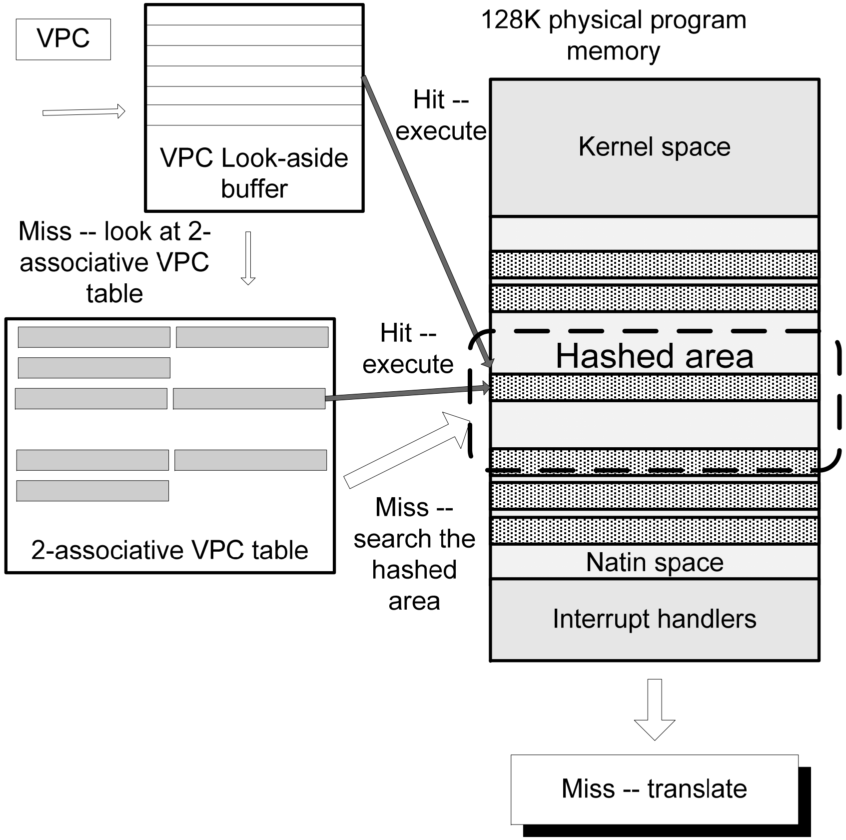


Naturalization and control

- CPU control – the OS can get the CPU to execute
  - Traditionally supported by privilege support & clock interrupts
- t-kernel – modify all branching instructions
  - Save registers, save destination and go to homeGate (welcomeHome)
  - welcomeHome – routine in the dispatcher; retrieve destination, seeks for a natin page (or create one) and transfer control to it
  - Transferring control flow to entry point – go to natin page and go through cascading branch chain
    - Just like that – too slow!
  - For branching instructions that are application-kernel transitions
    - One of every 256 backward branches calls the kernel's santy check routing
    - The rest goes almost unmodified

Three-level look up for a VPC

- Each VPM is hashed to a number of natin pages; need to check all entry points to decide
- 1. VPC look-aside buffer (fast)
- 2. Two-associative VPC table
- 3. Brute-force search on the natin pages (slow)



Differentiated VM – three memory areas

- Physical address sensitive memory (PASM)
  - Virtual/physical addresses are the same
  - The fastest access
- Stack memory
  - Virtual/physical addresses directly mapped
  - Fast access with boundary checks
- Heap memory
  - May involve a transition to kernel
  - The slowest, sometimes involves swapping
  - For kernel data integrity – the kernel has its own heap
- A challenge with flash
  - After 10k writes, a flash page cannot longer be used
  - If swap-outs evenly distributed to all pages, maximum lifetime

Implementation

Hardware paramenters	Data RAM External flash Program mem	4KB 512KB 128KB
OS Parameters	Virtual mem. Data frame Look-aside buffer 2-associative VPC System stack I/O Buffer	64KB 64 frames 64 entries 256 entries 1KB 516 bytes
Implementation details	Code size (source) Code (binary)	10 KLSC 29KB

MICA2

Kernel space
0x16200-0x1FFFF

Natin space
0x200-0x161FF

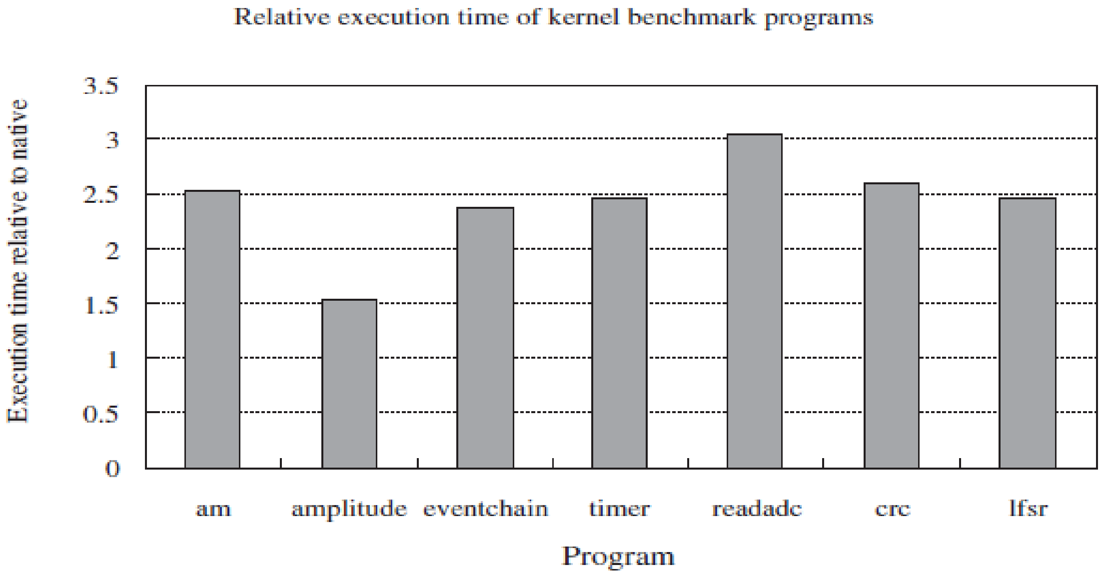
Interrupt handlers
0x0-0x1FF

128K Physical
program memory

Overhead of naturalization

- Kernel transition time
  - ~20 cycles for backward branches, rare
  - 4 cycles for the most common forward branch
- Kernel transition
  - Saves/restore registers / checks the stack pointers / Increments system counters
  - May need to
    - Look for destination address / Trigger naturalization of a new page / Re-link naturalized page
- Overhead of VM
  - Slowest stack access: 16 cycles
  - Heap access w/o swapping: 15 cycles
  - Heap access w/ swapping: 25.8ms (180,857 cycles)
  - .. But swap out time – 25.73ms (near hardware's limit)

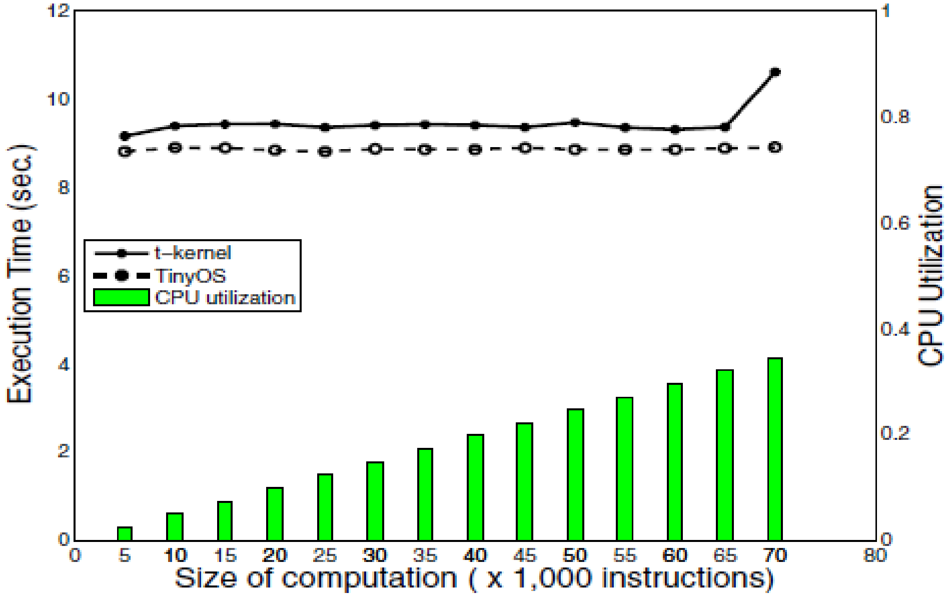
Overhead from the app's perspective



- Performance differs noticeably among applications
  - Different branch density
  - Different frequency of heap access
- For CPU-bound tasks – relative execution time 1.5-3
- But most WSN apps have low CPU utilization

Overhead from the app’s perspective

- PeriodicTask
  - Wake-up/poll-sensors/communicate
  - Varying the amount of computation in each task
  - Keep in mind the CPU idle ratio of TinyOS apps
    - μ - CPU utilization

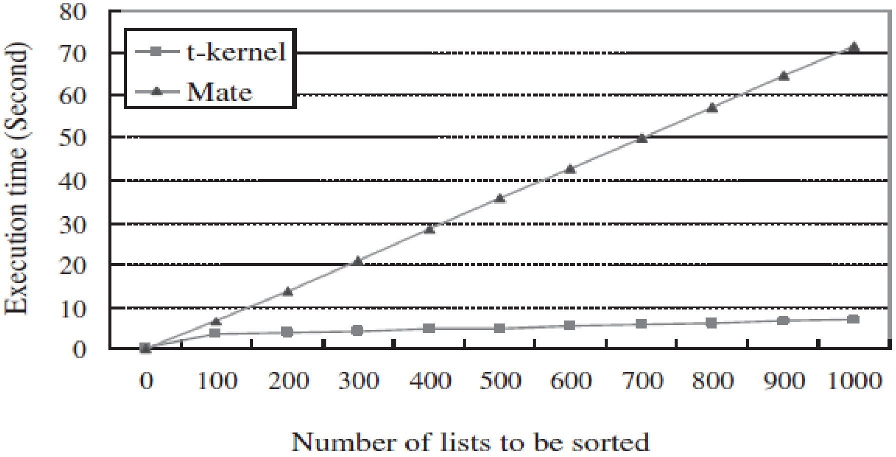


μ = 0.02

μ = 0.34

Comparison to VM approach

- Comparing with Maté, a VM for TinyOS
  - A stack based virtual architecture
  - Comparison with an insertion-sorting program
  - Initial cost of t-kernel comes from naturalization
    - After 100 grows slowly; naturalization has a one-time overhead
  - In contrast, bytecode translation has to be done every time
    - And sophisticated optimizations for VMs cannot save you here
- Of course, you could build Maté/TinyOS on top of t-kernel



Conclusions & Future Work

- Aiming at REM
  - Low energy budget, low CPU utilization, but high application requirements
- Make the common case fast
  - Use uncommon branches for control
  - Optimize memory mapping based on this
- What if power where not an issue?
- The overhead of naturalization killed some applications with timing assumptions built in
- Trashing will kill you – learn about typical locality and working set

Computer-chip fabrication techniques to make a gas-turbine engine that fits in the palm of a hand (Epstein, MIT).