# *t-kernel* – Reliable OS support for WSN

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

*Best paper award.*

# Wireless Sensor Networks

- A wireless network
  - Spatially distributed autonomous devices
  - With attached sensors
  - to cooperatively monitor physical or environmental conditions (e.g. temperature)
- Initially motivated by military applications, but many civilian apps today
  - Environmental and species monitoring, agriculture, production and delivery, healthcare, etc.

# Motivation

- Wireless sensor networks (WSNs)
  - Although using resource constrained nodes
    - Low-power microcontrollers
    - Small memory
    - Power constraints
  - Complex application requirements

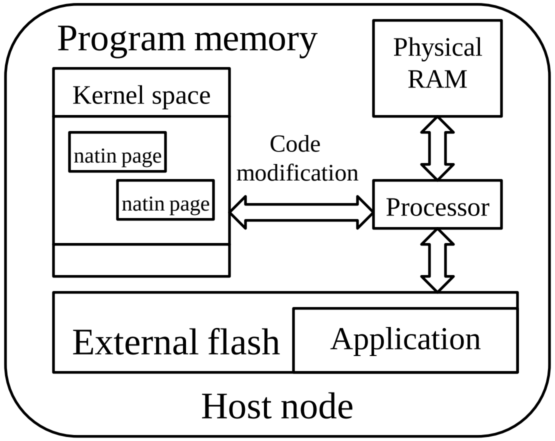
- OS support is very limited; applications (developers) could benefits from
  - OS protection
  - Virtual memory
  - Preemptive scheduling
- But microcontrollers don’t have HW support for this
  - E.g. privileged execution, virtual address translation, memory protection
- *How can we efficiently provide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

# Context – Complex apps requirements

- VM - VigilNet – large-scale surveillance
  - 30 middleware services & 40K SLC
  - In only 4KB RAM – note remotely enough!
  - 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 rebooting
    - To reduce too frequent restarts – long time w/o OS control

# Overview

- Wide variety of microcontrolers, minimum assumptions
  – It's reprogrammable, it allows writing something into memory & executing it
  – It has some external nonvolatile storage
  – It has some RAM available (4KB)
- Application
  – Binary program in sensor node's instruction set
  – Resident in flash memory




- When control reaches a new code page
  – Load-time code modification – *naturalization*
  – Done on demand, one page at a time
  – Output – a cooperative program supporting OS protection, VM & preemptive scheduling

# Naturalization and control

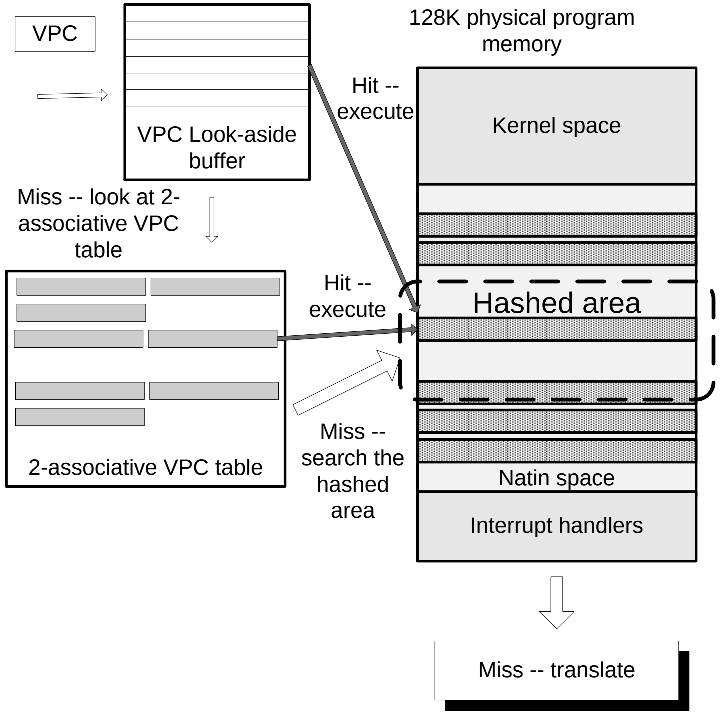
- CPU control – the OS can get the CPU to execute
  - Traditionally guaranteed by privilege support & clock interrupts
  - But in many microcontrollers the app can disable interrupts
- t-kernel
  - Modify program to ensure the naturalized version yields CPU to the kernel frequently
  - Which instructions? All branching instructions
- How to jump
  - Save registers, save destination & go to homeGate (welcomeHome)
  - welcomeHome (routine in the dispatcher) retrieves destination, seeks for a natin page (or create one) & transfer control to it
  - Transferring control flow to entry point – go to natin page & go through cascading branch chain to entry point

# Naturalization and control

- Just like that – too slow!
- A few fixes
  - Bridge transition directly link branch source & destination
  - Town transitions – first time make it into a bridge transition
  - Backward branching, less frequent than forward branching (6-8 instructions before any branching, 26-36 instructions before a backward one)
    - Count them – one of every 256 backward branches calls the kernel's sanity check routine
  - The rest goes almost unmodified

# Three-level look up for a VPC

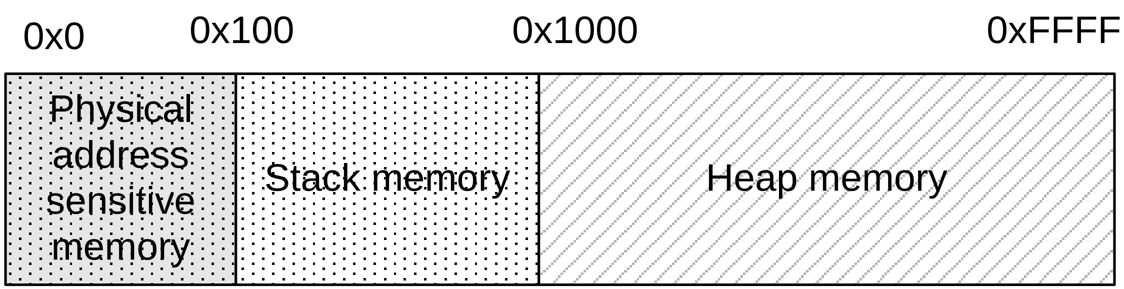
- Topology of naturalized program != application program
  - Code modification is done page-by-page
  - Code density changes after code modification
- No linear relationship between VPCs and HPCs
  - Need to check all entry points to decide
- Three level lookup
  - (1) VPC look-aside buffer (fast)
  - (2) Two-associative VPC table
  - (3) Brute-force search on the natin pages (slow but reliable)
    - Each VPC is hashed to a number of natin pages; each natin page cascading branch tests all entry points



# Differentiated Virtual Memory

- t-kernel provides virtual memory > physical memory
- Virtual/physical memory address translation, boundary check and memory swapping handle by natins
- To efficiently support large virtual address space without virtual memory hardware
  - Three types of memory with different attributes
    - Physical address sensitive memory (PASM)
    - Stack memory
    - Heap memory

Example of a virtual data memory configuration



# Differentiated Virtual Memory

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

# Kernel/Application Interface

- Interface: system calls, event triggering and interrupt handling
- System calls
  – A set of special VPC as system call entry points
- Notification of service completed – event trigger
  – Kernel generates a software interrupts that is handle by the application
- Same mechanism to handle hardware interrupts

# Implementation

Implemented and tested in several platforms

One example

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

MICA2

| |
|---|
| Kernel space<br>0x16200-0x1FFFF |
| Natin space<br>0x200-0x161FF |
| Interrupt handlers<br>0x0-0x1FF |

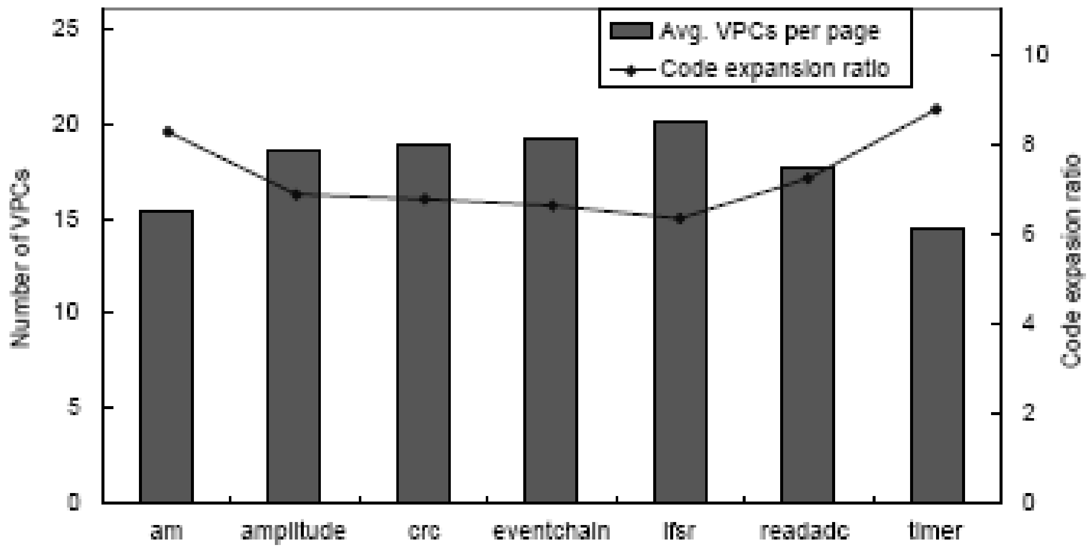
128K Physical program memory (28KB for kernel)

# Overhead of naturalization

- Kernel transition time
  - ~20 cycles for backward branches taken, rare
    - Avg. number (over?) with amortized cost of sanity check routine
  - 5 cycles for the most common forward branch taken
- 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 erase/write to flash – 25.73ms (i.o. I/O latency dominated)

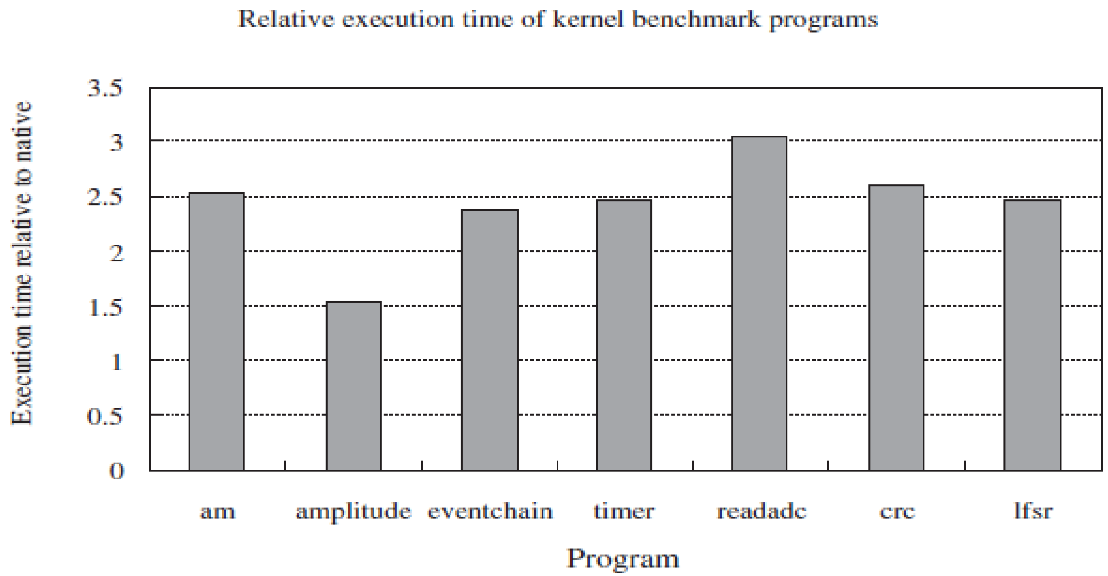
microbenchmarks

# Overhead from the app's perspective




- Naturalization expands the code size because of branch regulating, DVM and cascading branch chain
- Large variance in kernel overhead from naturalization
  - 22 to 51 natin page writes or 590 to 1380ms of naturalization time per 1KB of application code

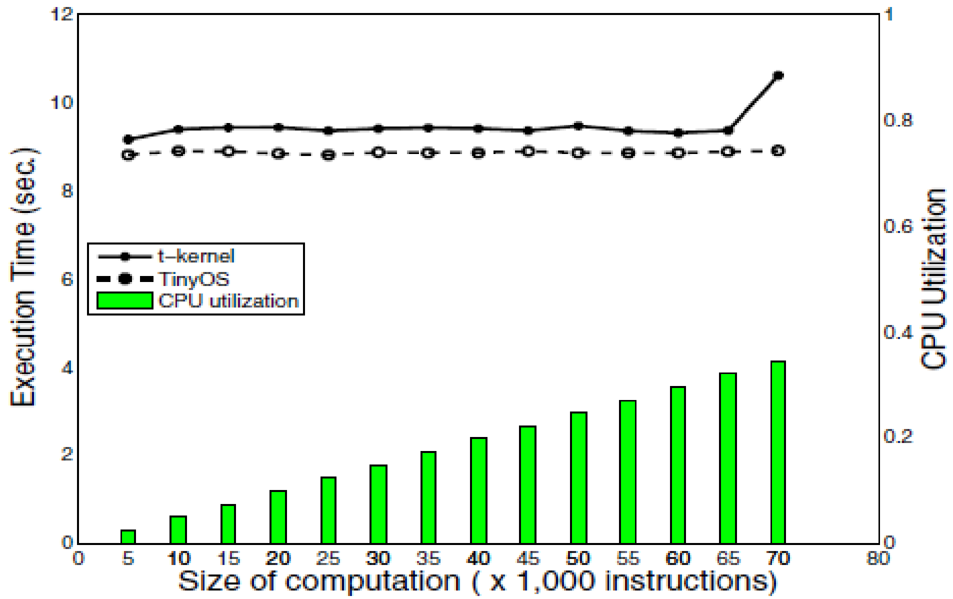
# Overhead from the app’s perspective’




- 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
  – >92% CPU time in iddle mode for the survey apps

# Overhead from the app's perspective

- PeriodicTask
  - Wake-up/poll-sensors/communicate
    - Common WSN model
  - Varying the amount of computation in each task
  - Keep in mind the CPU idle ratio of TinyOS apps
    - μ - CPU utilization (0.34 ~ 3x higher than usual)




μ = 0.02

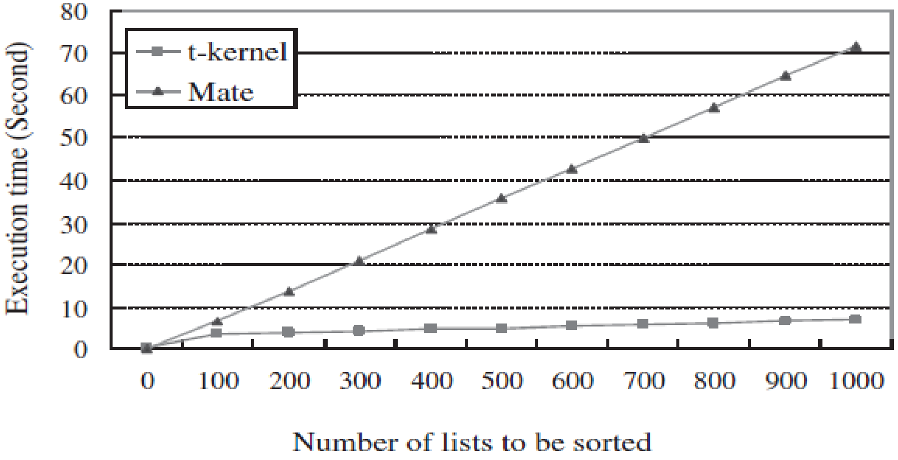
μ = 0.34

# The power issue

- Power consumption on sensor nodes depends on
  - Percentage and average sleep mode current
    - Low-power modes where nodes wait to be woken up
  - Percentages and average of idle & active modes (duty cycle)
  - With t-kernel – energy consumed by flash I/O & avg #swaps
- t-kernel trades energy for higher abstraction, but upgrading hardware could do the same
  - If app has mem. access with  low-locality, DVM thrashes, energy consumptions goes up
- Still,
  - Most apps seem to have good locality
  - Flash I/O should get cheaper, in terms of power consumption
  - Bigger RAM leaks more power

# Comparison to VM approach

- Comparing with Maté, a Virtual Mach 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

- Supporting useful OS abstractions without hw support – *Ontogeny recapitulates phylogeny**
  - Higher abstraction maybe well worth the price
  - Target – low energy budget, low CPU utilization, but high application requirements
- Make the common case fast
  - Use uncommon branches for control
- Overhead of naturalization killed some apps with timing assumptions
  - Working on RT support (e.g. pre-naturalization)
- Thrashing can kill you
- And if the power issue were to go away …

Computer-chip fabrication techniques to make tiny gas-turbine engine (Epstein, MIT).

The development of an embryo repeats the evolution of the species (* Ernst Haeckel)

# *Did you think this was interesting?*

- Let's keep the conversation in Advanced Operating Systems

What others have to say (Rating: 5.8/6)

"Discussions by the instructor, always probed areas that weren't originally explored and proved to be extremely useful in stimulating my mind./ This class is engaging, fun, and a great learning experience./ This is a great class for gaining exposure to various types of computer systems. Fabian is a great, fun professor./ A great introductions to current Systems research. Reviewing a conference paper for each class really does improve your technical reading and critiquing skills."