Towards a Resilient Operating System for Wireless Sensor Networks

Hyoseung Kim Hojung Cha

Yonsei University, Korea

2006. 6. 1.

Hyoseung Kim
hskim@cs.yonsei.ac.kr

Motivation (1)

- **Problems: Application errors on sensor networks**

Diverse hardware devices

Cooperate with large number of nodes

Unattended operation for long-time

Kernel data corruption

Kernel code execution

Hardware control faults

Kernel fail & sensor node crash

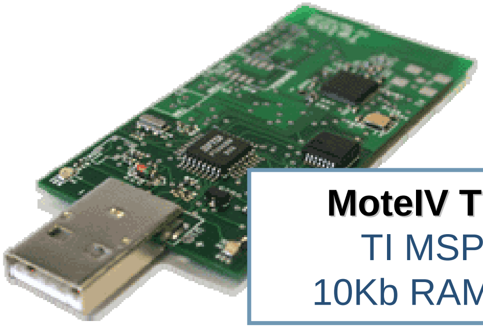
No way to recover from system failure

Crashed sensor node is useless in the field

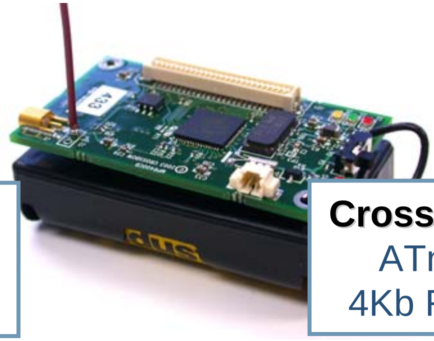
System safety like general purpose OS needed!

Motivation (2)

- **Popular sensor nodes: Telos / Mica**



MoteIV Telos Rev. B
TI MSP430 8Mhz
10Kb RAM, 60Kb ROM



CrossBow Mica2, MicaZ
ATmega128L 8Mhz
4Kb RAM, 128Kb ROM

  – No MMU, privileged mode, and exceptions

- **Current sensor network systems**

Sensor operating systems

TinyOS | SOS | MANTIS

Virtual machine

Maté

  – No system safety mechanism

Towards Resilient Sensor Networks

- **Ensure system safety at the operating system level**
  - **Applications** : we cannot write the code safely all the time
  - **Sensor node hardware** : does not easily detect application errors

- **Objectives**
  - Provide error-safe mechanism on resource-constrained sensor devices
  - Do not require any hardware supports
  - Do not require users to learn new programming language semantics

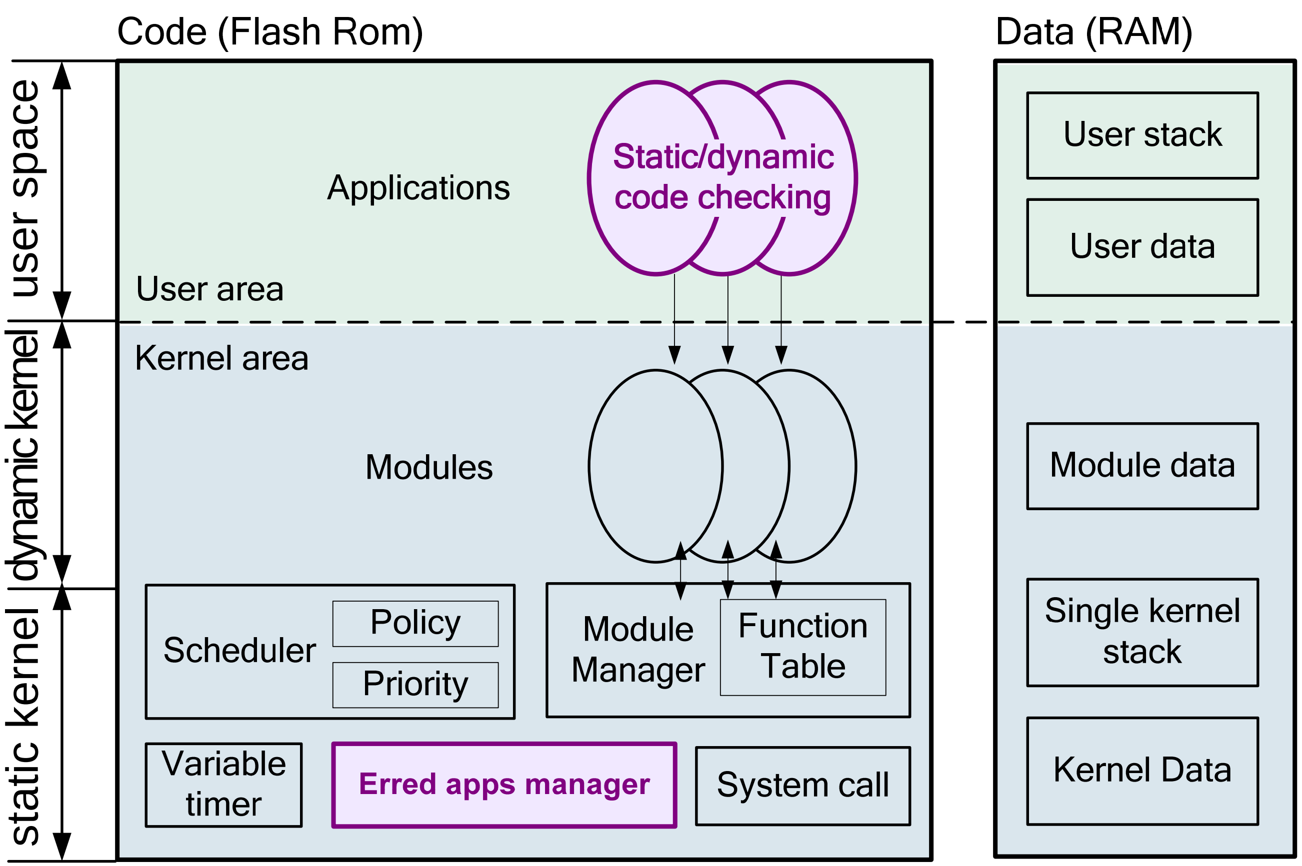
Previous Work

- **SFI (Software Fault Isolation)**
  - Requires MMU for segmentation and stack safety
  - Designed for fixed-length instruction architecture
- **Proof-carrying Code**
  - The automatic policy generator does not exist
- **Programming language approaches**
  - Cyclone, Control-C, Cuckoo
  - Users should be aware of the different usages of pointer/array
  - Requires hardware supports for stack safety

- T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang, **"Cyclone: A safe dialect of C,"** *Proc. of the 2002 USENIX Annual Technical Conference,* June 2002.
- D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. **"Memory safety without runtime checks or garbage collection,"** *Proc. of Languages, Compilers and Tools for Embedded Systems*, 2003.
- R. West, and G. T. Wong, **"Cuckoo: a Language for Implementing Memory and Thread-safe System Service"**, Proc. of International Conference on Programming Languages and Compilers, 2005.

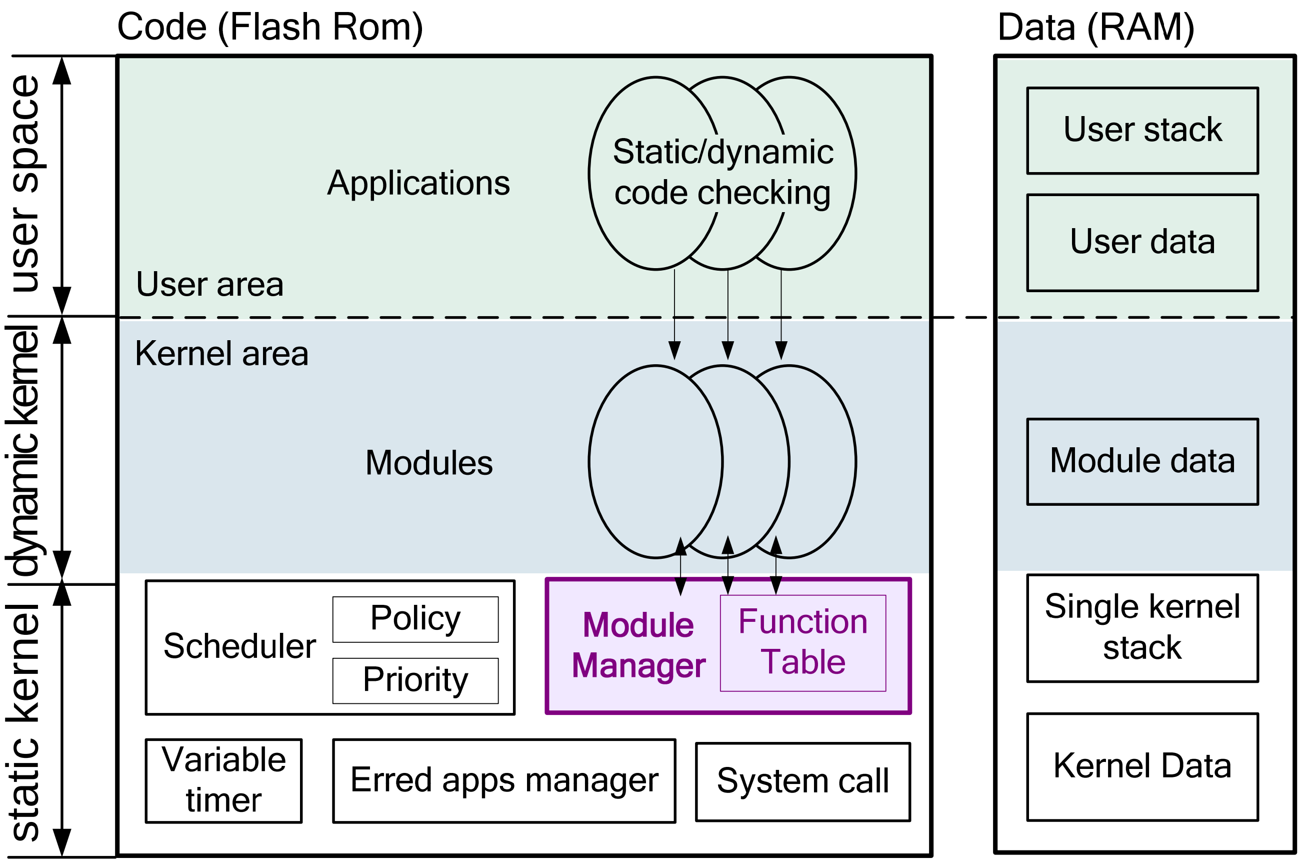
RETOS Architecture

Resilient, Expandable, and Threaded Operating System



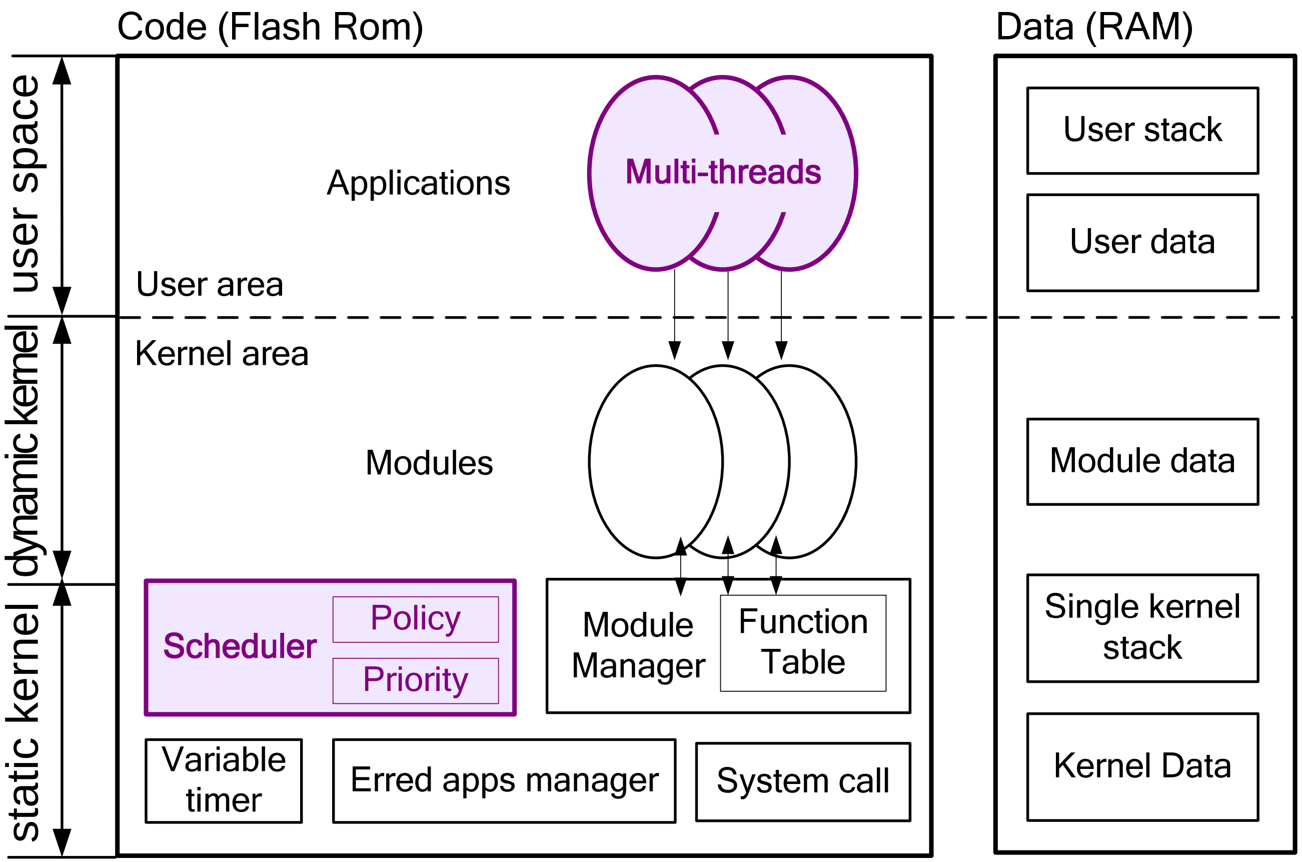
RETOS Architecture

Resilient, Expandable, and Threaded Operating System



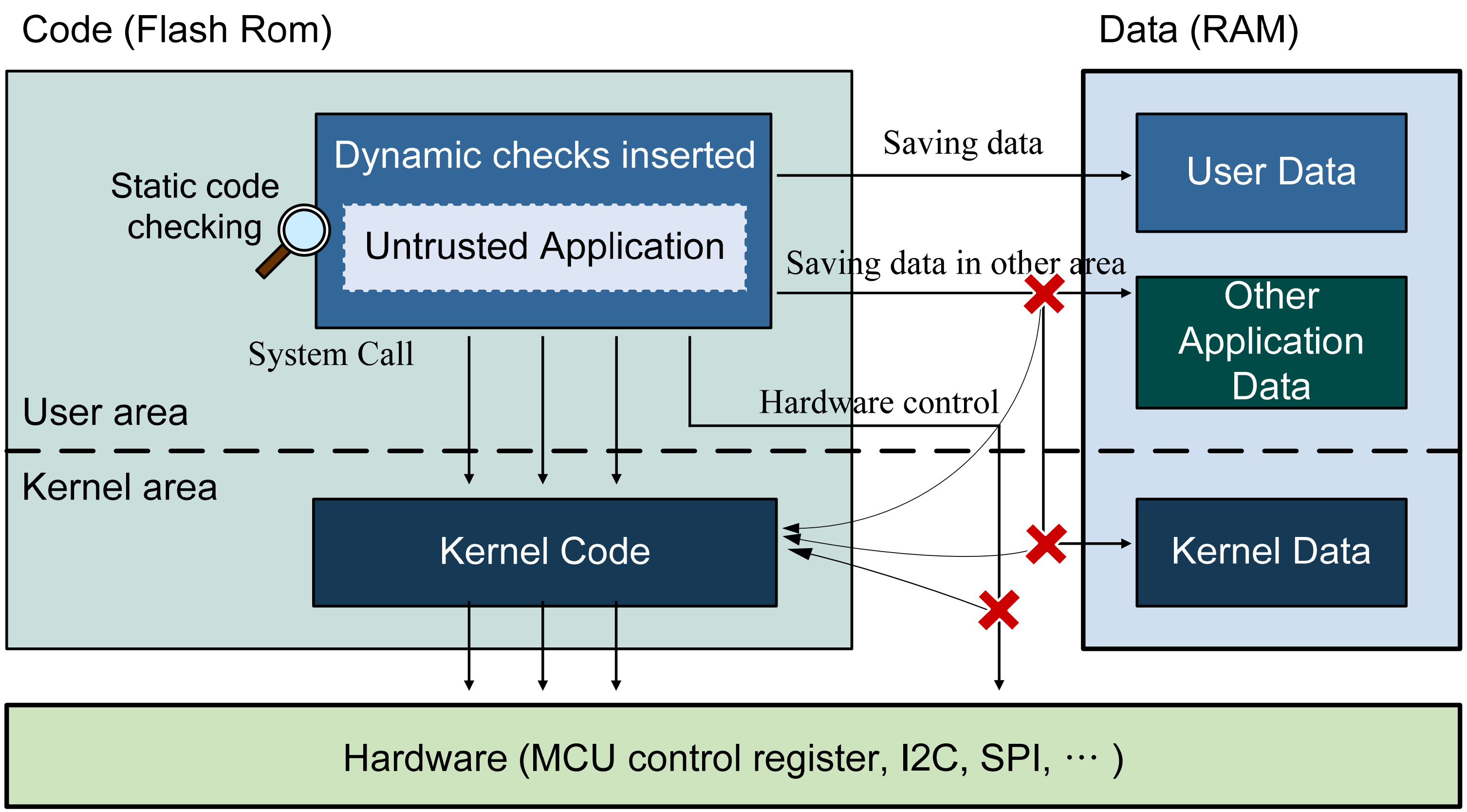
RETOS Architecture

Resilient, Expandable, and Threaded Operating System



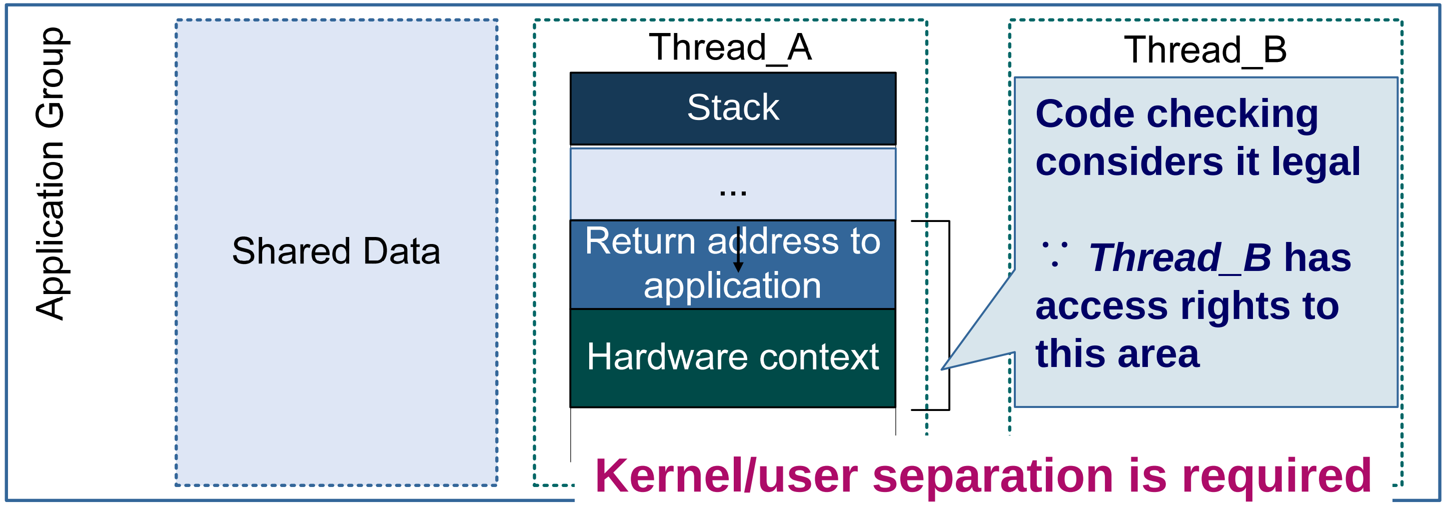
Ensuring System Safety

- **Principle**
  - Dual Mode operation & static/dynamic code checking



Dual Mode Operation

- **Why dual-mode is needed?**
  - Static/dynamic code checking evaluates if the application modifies data or issues code in its allocated data
  - Preemption would invoke problems
- **(ex) *Thread_A* is preempted by *Thread_B***



Static/Dynamic Code Checking

- Use data and code area within application itself
- Restrict direct hardware resource manipulation

Application Source Code → Assembly Code → Dynamic Checks Insertion → Binary Code → Static Code Checking → Trusted Application Code

```c
#include <io.h>
#include <syscall.h>
#include <pthread.h>

...

/* main thread */
int main ()
{
    while (1) {
        void (*func)(void);
        /* radio packet recv */
        radio_recv(PORT, &pkt);
        func = foo_func[pkt->id];

        /* do something*/

        func();
    }
}
```

Application Source Code

```
main:
/* prologue: frame size = 44 */
.L__FrameSize_main=0x2c
.L__FrameOffset_main=0x2e
        mov     #(__stack-44), r1
/* prologue end (size=2) */
.L23:
/* do something */
        call    &func
        mov     #llo(6), r15
        call    #_syscall0
        jmp     .L5
        br      #__stop_progExec__
/* epilogue: frame size=44 */
        add     #44, r1
        br      #__stop_progExec__
/* epilogue end (size=4) */
/* function main size 168 (162) */
.Lfe6:
        .size   main,.Lfe6-main
```

Dynamic Checks Insertion

Static Code Checking

```
mov     #1792,   r1
mov     #6,      r15
push    r15
mov     &0x0608,r15
cmp     12(r15),r1
jnc     $+8
cmp     14(r15),r1
jnc     $+6
call    #4488
pop     r15
call    #4494
mov     #1000,   r13
clr     r14
mov     #40,     r15
push    r15
mov     &0x0608,r15
cmp     12(r15),r1
jnc     $+8
cmp     14(r15),r1
jnc     $+6
call    #4488
```

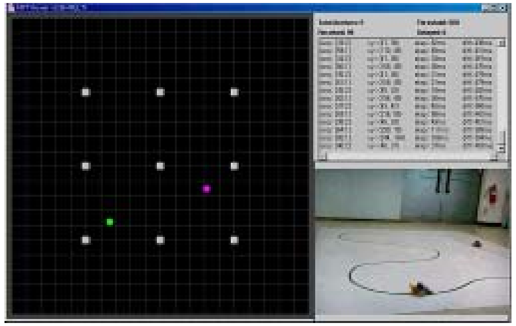
Trusted Application Code

Implementation

- **RETOS implemented on TI MSP430 F1611**
  - 8MHz core clock, 10Kb RAM, 48Kb Flash ROM
  - Current version: 0.92 (May 24 2006)

- **Peripheral supports**
  - 2.4Ghz RF Module (Chipcon 2420)
  - Ultrasound, Humidity, and Light sensors

- **Applications: MPT (Multi-Party Object Tracking)**



Functionality Test (1)

- **Classify safety domain into four parts:**
  - Stack / Data safety → Stack and data area
  - Code safety → Control flow
  - Hardware safety → Immediate hardware control

Hardware safety

Disable interrupt

```
DINT();
```

→ Detected by **Static check**

Flash ROM writing
(memory mapped registers)

```
FCTL1 = FWKEY+WRT;
```

→ Detected by **Static check**

Stack safety

General/Mutual recursive call

```
void foo() {
        foo();
}
```

```
void foo() {            void bar() {
        bar();                  foo();
}                       }
```

→ Detected by **Dynamic check**

Functionality Test (2)

Data safety

Directly addressed pointers

```
int *tmp = 0x400;
*tmp = 1;
```

→ Detected by **Static check**

Illegal array indexing

```
/* array in heap area */
int array[10];
…
for(i = 10; i > 0; i--) {
        array[i-100] = i;
}
```

→ Detected by **Dynamic check**

Code safety

Directly addressed function call

```
void (*func)(void) = 0x1000;
func();
```

→ Detected by **Static check**

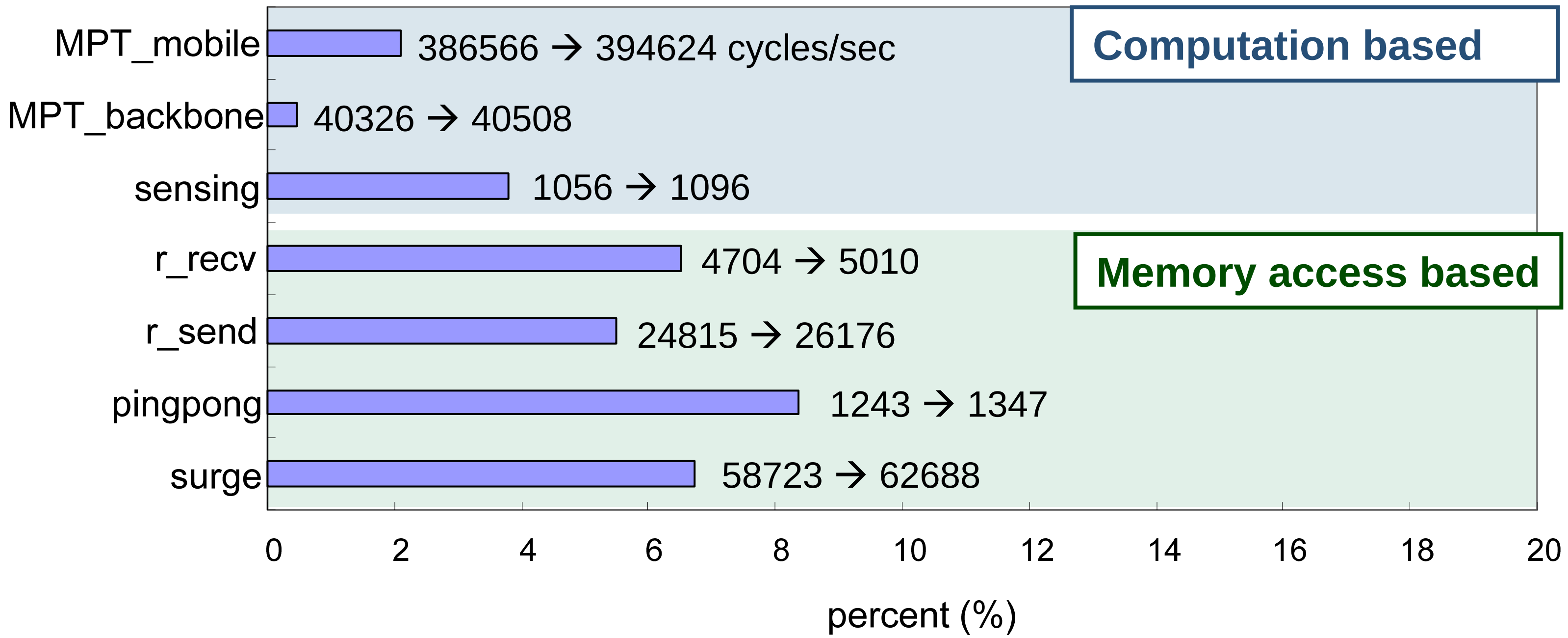
Corrupted return address (Buffer overflow)

```
void func() {
        int array[5], i;
        for(i = 0; i < 10; i++)
                array[i] = 0;
}
```

→ Detected by **Dynamic check**

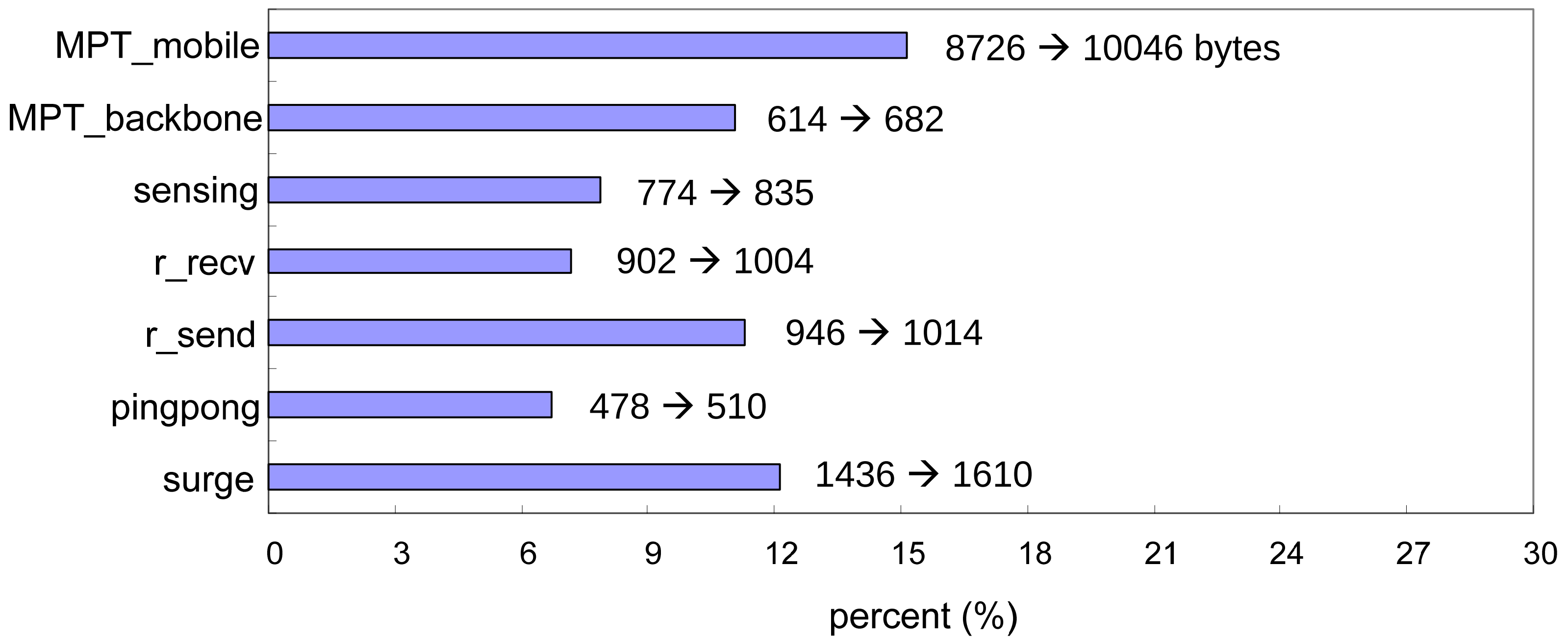
Overhead Analysis (1)

- **Execution time running in user mode**
  - Programs, which require more memory access than complex arithmetic calculations, shows larger overhead



Overhead Analysis (2)

- **Application code size comparison**
  - Code size increased, independent of application type
  - Applications are inherently small in size, separated from kernel



Overhead Analysis (3)

- **Dual mode operation**

	Single mode (cycles)	Dual mode (cycles)	**Overhead (cycles)**
system call (led toggle)	264	302	**38**
system call (radio packet send)	352	384	**32**
timer interrupt (invoked in kernel)	728	728	**0**
timer interrupt (invoked in user)	728	760	**32**

- **Summary**
  - Computational overhead & larger code size are inevitable to provide system safety in sensor node platform
  - However, they are considered as a fair trade-off compared to a system failure

Conclusions

- **Safety mechanism for wireless sensor networks**
  - Separate errant application from the kernel, “Kernel never die”
  - Automatically recover from serious errors
  - Useful in the real, large-scale sensor networks

- **Current status**
  - RETOS is being developed in our research group
  - Being ported to other processors (AVR)
  - “Application-Centric Networking Framework for Wireless Sensor Networks”
    - *Mobiquitous 2006,* San Jose, July 17-21

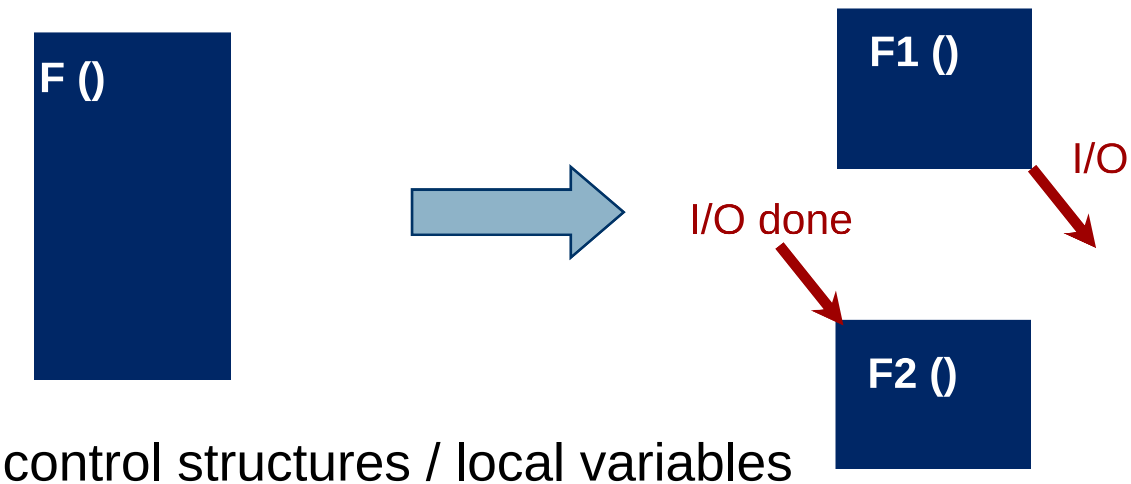
Questions

WatchdogTimer

- **Not easy to recognize and handle problems such as:**
  - Memory access beyond the application area
  - Immediate hardware control
- **Watchdog timer simply makes the system be restarted**
  - One errant application can stop all other applications
  - Disturbance of long-term operations of sensor applications

Multithread vs. Event-driven

- **Concurrency for computation based applications**
  – **Run-to-completion** vs. **preemptive time-sliced thread**
- **Poor software structure on event-driven model**
  – One conceptual function split into multiple functions



  – Loss of control structures / local variables
- **Simplified synchronization methods on thread model**
  – Like “atomic” operation in nesC

Limitations

- **Library codes**
  - We assume the libraries always operate safely
  - In real situation: If a user passes an invalid address to *memcpy()*?
  - Make wrapper functions that checks address parameters
- **Divide by zero**
  - Hardware multiplier/divider equipped processor
    : instruction level checking
  - Emulated : library level checking
- **Intentionally skips the code checking sequences**
  - User authentication on code updating would be required

Sensor OS Comparison

	System Safety (kernel-user separation)	Light-weight dynamic reprogramming	Programming Model	Programming Language
TinyOS	**No**	**No**	**Event-driven**	**nesC**
SOS	**No**	**Yes**	**Event-driven**	**Standard C**
MANTIS	**No**	**No**	**Preemptive Multithreading**	**Standard C**
RETOS	**Yes**	**Yes**	**Preemptive Multithreading**	**Standard C**