

Efficient Type and Memory Safety for Tiny Embedded Systems

John Regehr Nathan Coopriider Will Archer Eric Eide

University of Utah
School of Computing

A conflict in embedded systems

- Tiny embedded systems coded in C
- These are used in important and even safety critical systems
- Developers are extremely reluctant to change languages
- Developers are extremely reluctant to introduce time, space, and memory overhead

Our contribution

- Type and memory safety with little overhead
 - Lots of engineering required
 - Must be fast if people are going to use it
 - Low or no costs
 - 2% average decrease in duty cycle
 - 8% average increase in code size
 - 6% average increase in data size
- Safety with minimal programmer impact
 - Safe TinyOS
 - works on unmodified legacy C code

Software and hardware platform

- TinyOS
 - libraries of code for sensor networks
 - idioms are conducive to static analysis
 - static memory allocation model
 - written in nesC
 - unsafe dialect of C that compiles to C
- Mica2 from Crossbow
 - ATmega128 8-bit processor
 - 4 KB RAM, 128 KB flash



Safety for embedded systems

- Type and memory safety
 - early detection of bugs
 - observability and predictable consequences of run-time faults
- CCured is the starting point
 - Safe C dialect developed at Berkeley
 - Translates a C program into a safe C program by inserting safety checks
- Example of null check:

```
__CHECK_NULL((void *)next_task, 0xAA);  
((*next_task))();
```

Initial drawbacks of CCured

- Overhead
 - All these checks slow down the program and use up memory
- CCured library
 - Does not fit on motes if unmodified
 - First estimation at fitting onto Mica2
 - over 1KB of RAM
 - over 3KB of ROM

Addressing the drawbacks

- One-time, manual changes to CCured library
 - remove OS and x86 dependencies
 - drop garbage collector
- Refactor hardware accesses
- Protect non-atomic accesses to fat-pointers
- Compress the error messages
- Optimize

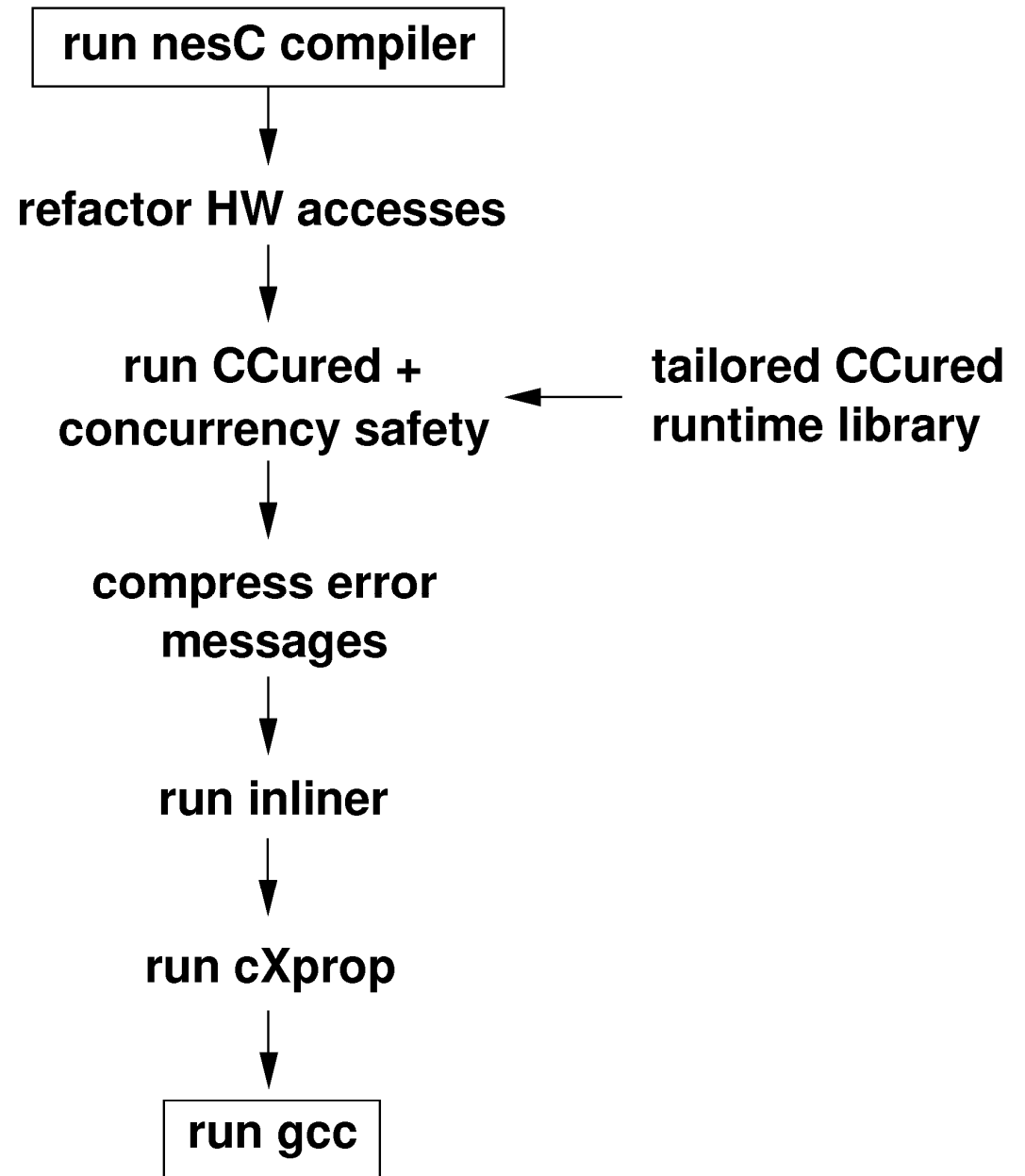
Optimize with an inliner

- Inlining can reduce size
- Inlining introduces some context sensitivity
- Parameterize inlining decisions
 - sweet spot in between
 - maximally reduce size AFTER optimization

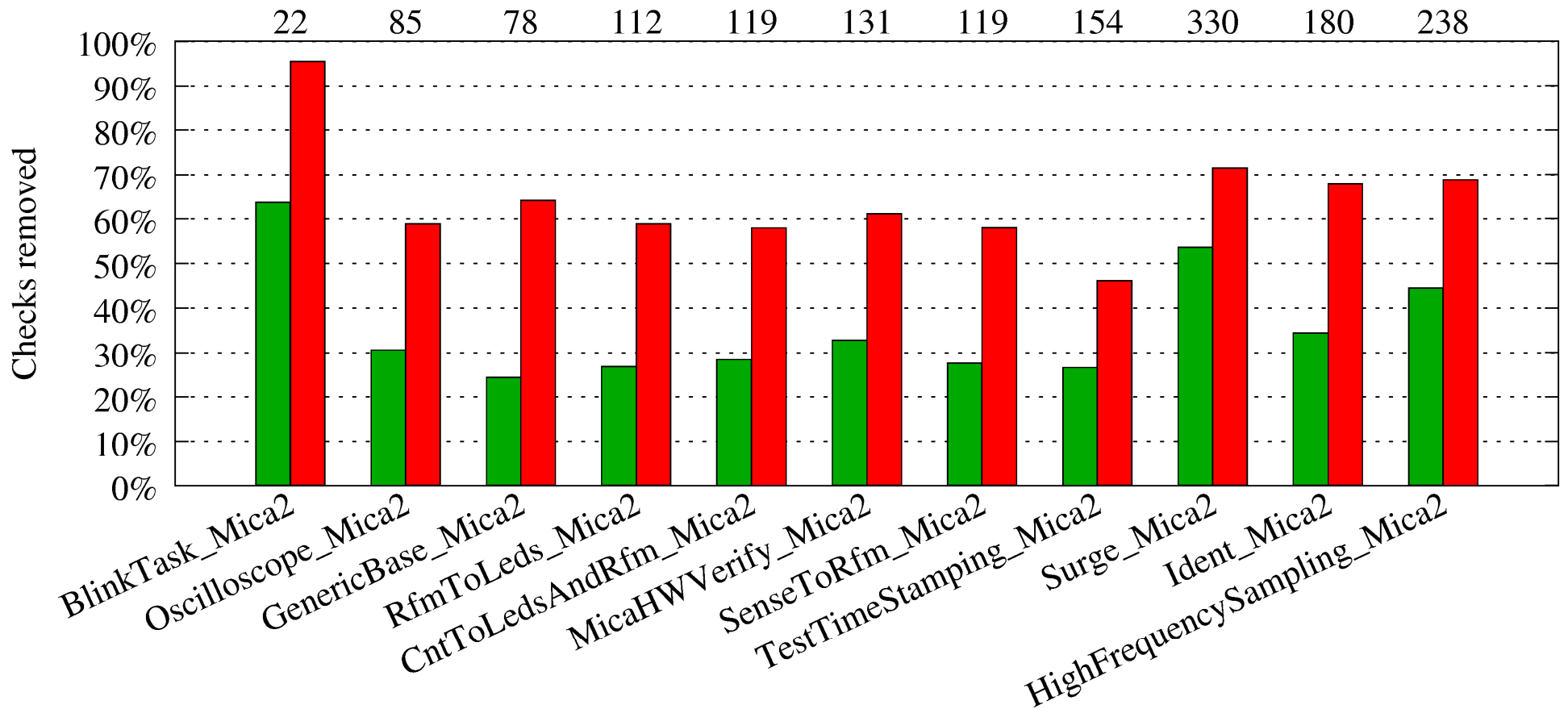
Optimize with cXprop

- Interprocedural dataflow analysis
- Analyzes concurrent programs
 - Removal of nested atomic sections
- Simultaneous pointer analysis
- Aggressive dead code elimination
 - whole program

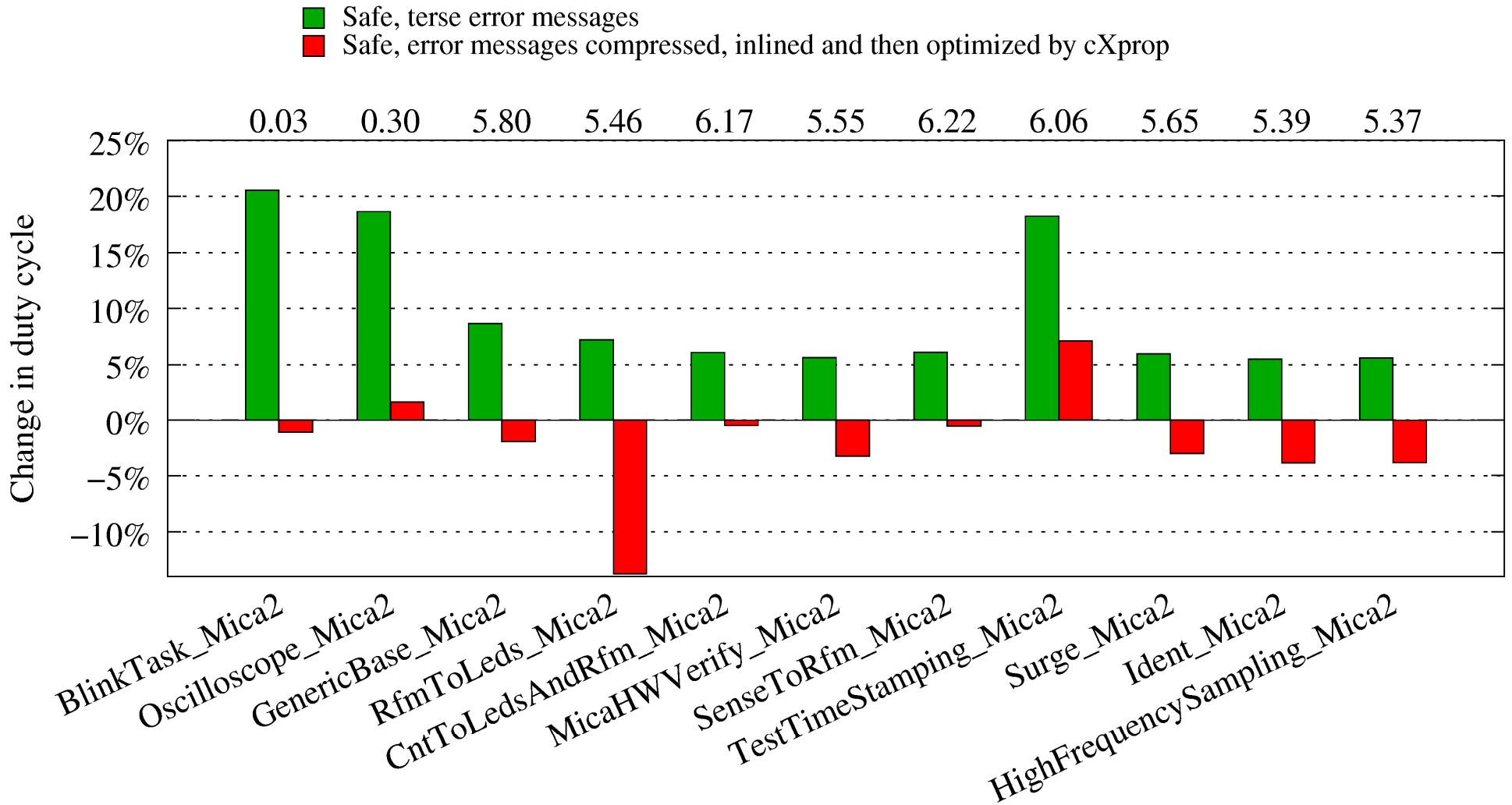
Safe TinyOS Toolchain



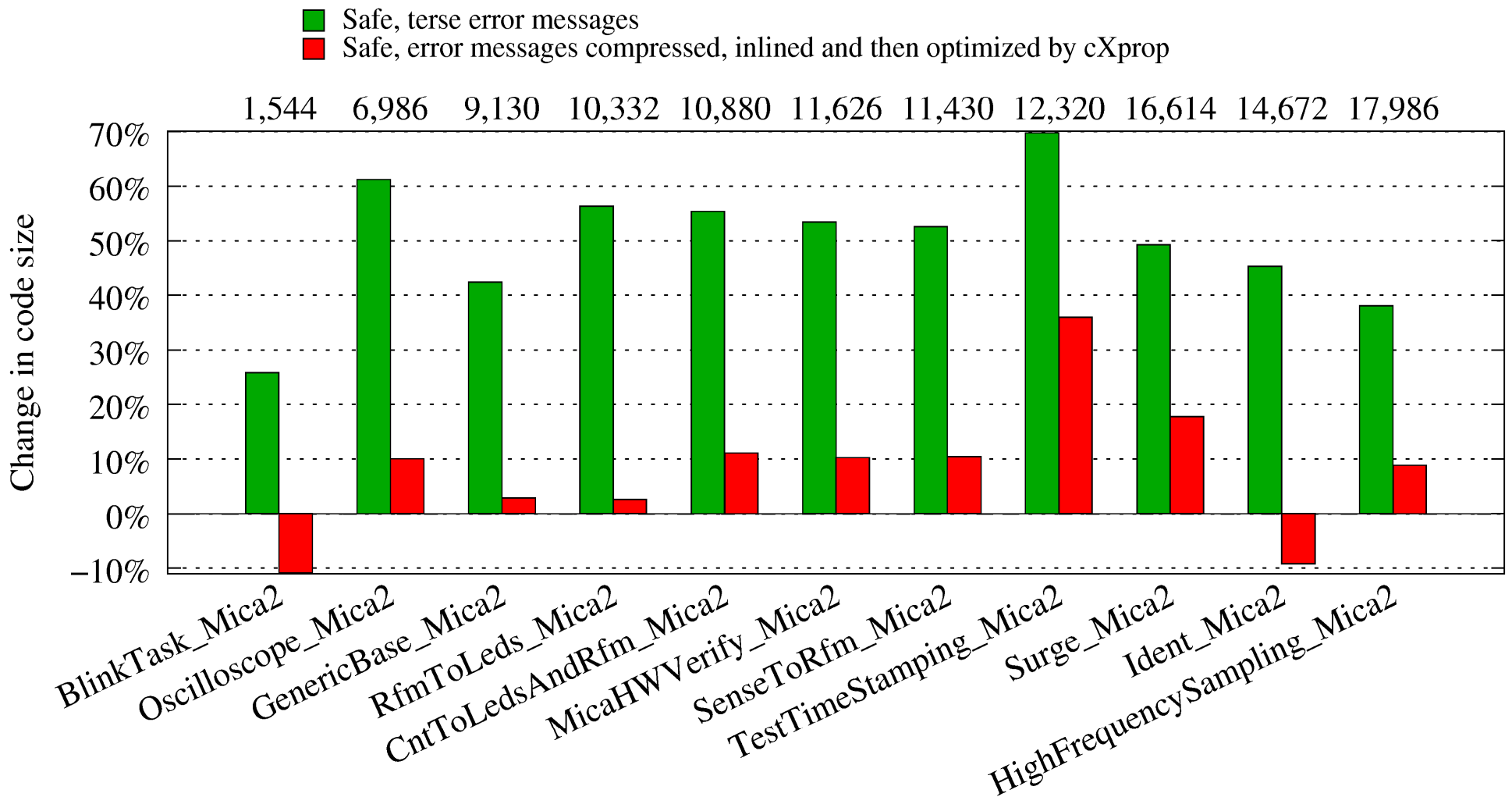
■ CCured optimizer + gcc
■ CCured optimizer + inlining + cXprop + gcc



Average 64% of checks removed in Safe TinyOS



Average 2% decrease in duty cycle via Safe TinyOS

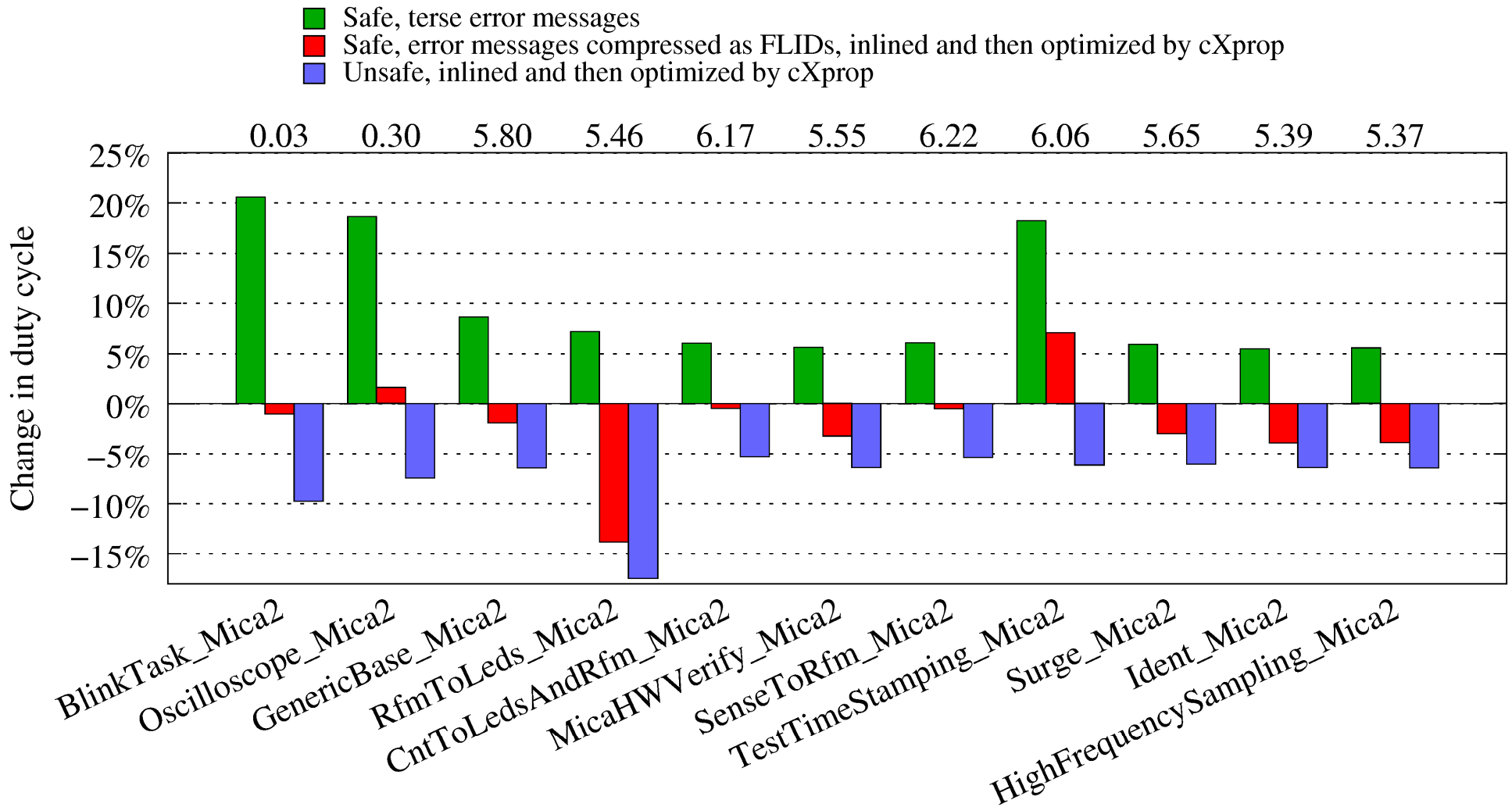


Average 8% increase in code size via Safe TinyOS

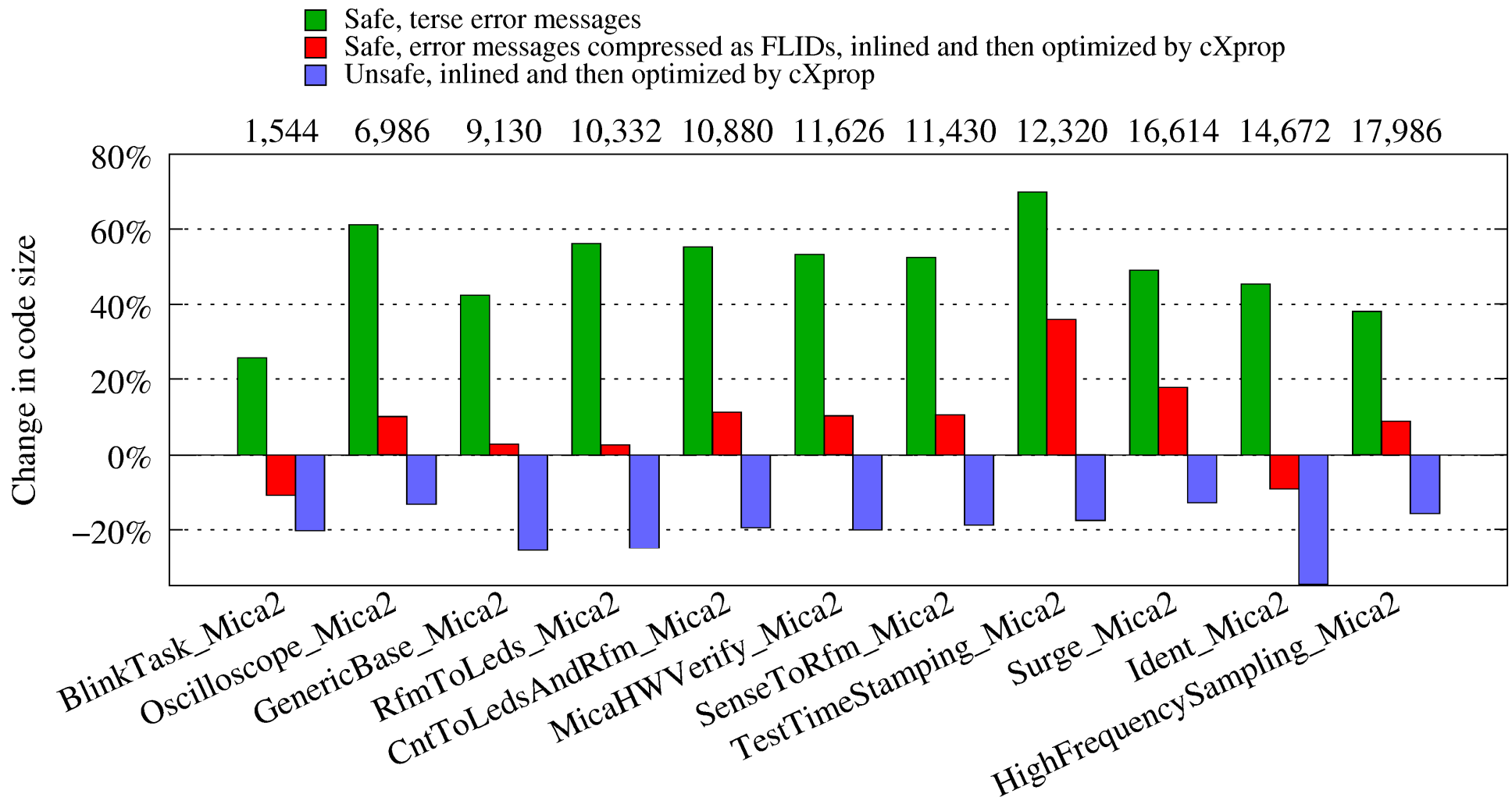
Conclusion

- Type and memory safety can be practical for tiny embedded systems
 - Low or no run-time cost compared to original unsafe applications
 - Can fit easily into existing programming practice
 - Legacy code is cured with no programmer effort

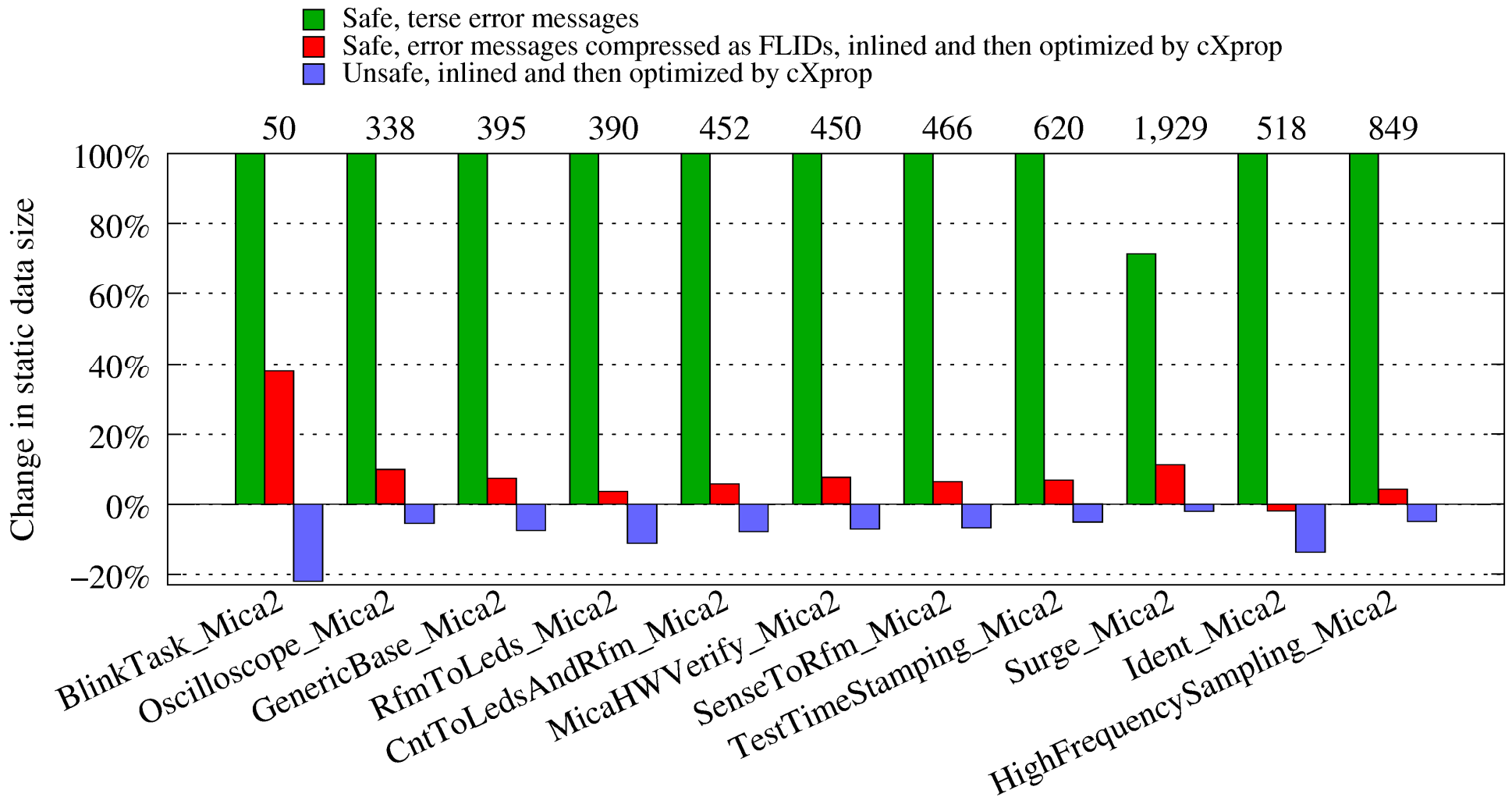
Questions?



Average 2% decrease in duty cycle via Safe TinyOS



Average 8% increase in code size via Safe TinyOS



Average 6% increase in data size via Safe TinyOS