A System For Coarse Grained Memory Protection In Tiny Embedded Processors

Ram Kumar, Akhilesh Singhania, Andrew Castner, Eddie Kohler, Mani Srivastava
University of California at Los Angeles
420 Westwood Plaza, Los Angeles, CA, USA
{ram,akhi,mbs}@ee.ucla.edu, {castner,kohler}@cs.ucla.edu

ABSTRACT

Many embedded systems contain resource constrained microcontrollers where applications, operating system components and device drivers reside within a single address space with no form of memory protection. Programming errors in one application can easily corrupt the state of the operating system and other applications on the microcontroller. In this paper we propose a system that provides memory protection in tiny embedded processors.[1]. Our system consists of a software run-time working with minimal low-cost architectural extensions to the processor core that prevents corruption of state by buggy applications. We restrict memory accesses and control flow of applications to *protection domains* within the address space. The software run-time consists of a *Memory map*: a flexible and efficient data structure that records ownership and layout information of the entire address space. Memory map checks are done for `store` instructions by hardware accelerators that significantly improve the performance of our system. We preserve control flow integrity by maintaining a *Safe stack* that stores return addresses in a protected memory region. Cross domain function calls are redirected through a software based jump table. Enhancements to the microcontroller `call` and `return` instructions use the jump table to track the current active domain. We have implemented our scheme on a VHDL model of ATMEGA103 microcontroller. Our evaluations show that embedded applications can enjoy the benefits of memory protection with minimal impact on performance and a modest increase in the area of the microcontroller.

Categories and Subject Descriptors: C.3 [Special - Purpose and Application-Based Systems]: Real-time and embedded systems

General Terms: Performance, Design, Reliability

Keywords: Memory Protection, Software Fault Isolation

[1]8, 16 and 32-bit microcontrollers with limited resources

This paper is based on research funded in part by UCLA CENS, NSF under award 0520006, U.S. ONR under award N000140610253, U.S. ARL and the U.K. MoD under Agreement W911NF-06-3-0002. Any opinions expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.
DAC'07, June 4 -8, 2007, San Diego, California, USA.

1. INTRODUCTION

Software complexity on tiny embedded processors is increasing. Software supports a diversity of peripheral devices, multiple distributed middleware services, dynamic code updates and concurrent applications in a resource constrained environment. Operating systems (TinyOS [10], SOS [5]), Virtual Machines (Maté, Sun SPOTS) and Wireless Applications (Zigbee [16]) are some of the complex software systems running on 8-bit microcontrollers. Implementing embedded software modules is a challenge, as programmers are forced to deal with severe resource constraints and concurrency issues. Furthermore, there is very limited debugging support on tiny embedded processors, leading to an abundance of programming errors. In particular, corruption of memory due to lack of protection from buggy applications is a very serious problem. The impact of these errors can be quite severe.

Architecture of tiny embedded processors is very simple. The entire memory is accessible to all software modules running on the processor via a single address space. Architecture features such as memory management units (MMU) and privileged-mode execution are common only in desktop/server class processors to isolate and protect data and code of one program from another. An MMU provides virtual memory in addition to protection and requires lot of memory for storing page tables for address translations. Embedded microcontroller designers face extreme pressure to minimize cost and area of a chip. Sometimes, even 32-bit ARM processor cores are not equipped with MMUs to minimize system cost and power [1]. Therefore, current MMU designs will continue to be absent from low-cost low-power micro-controllers.

Software-based Fault Isolation (SFI) (or "Sandboxing") is a class of techniques proposed by Wahbe et. all for memory protection within single address space in desktop microprocessors. In SFI, the address space of a process is partitioned into contiguous segments and each segment is allocated to a software module. Software modules are implemented with run-time checks that ensure that all memory accesses reside entirely within the segment allocated to it. These run-time checks are introduced through rewrite of a compiled binary. Our approach is motivated by SFI but it has fundamental differences due to the resource constraints of the tiny embedded processors. *First*, we do not partition the address space of the microcontroller, as the available memory on microcontrollers is severely limited. For example, the ATMEGA128 AVR has only 4 KB of on-chip RAM. In most systems [6] this is the total available memory. Static parti-

tioning would further limit the memory that is available to individual software components. Instead we rely on a *Memory Map*: a data structure that can efficiently record ownership and layout information of the entire address space. *Second*, we do not rewrite binary to introduce run-time checks. We instead enhance the implementation of `store, call` and `return` instructions in the microcontroller to perform run-time checks in hardware. This minimizes the performance overhead of performing run-time checks in software and also eliminates the binary rewrite step of SFI which can be quite error prone.

Our overall system can be viewed as a hardware/software co-design approach to memory protection. Low cost architecture extensions and a software run-time library work together to isolate different software components running on an embedded processor. Our system is entirely implementable in software albeit with a high performance overhead. Simple enhancements to the microcontroller core enable us to perform critical operations in hardware, thereby improving performance significantly.

1.1 System Overview

An overview of our system highlighting all its components is shown in Figure 1. The final firmware image is composed of multiple software modules that need to be protected from one another. A cross domain linking mechanism (described in Section 3) installs the software modules in separate protection domains. The cross domain calls are redirected through a software jump table. The redirection assists the processor in determining the identity of the currently active domain. The Memory Map (described in Section 2) tracks layout and ownership information for the protected address space. The firmware image interacting with the memory map and hardware enhanced run-time checkers is guaranteed to be memory safe. The cost and performance analysis of our system is done in Section 4. We present related work in Section 5 and concluding remarks in Section 6.

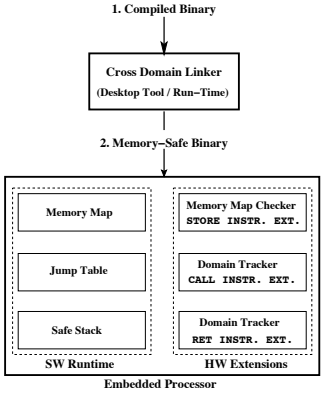


Figure 1: System Overview

2. MEMORY MAP

2.1 Protection Domains

Our *fault model* for memory protection is the corruption of state belonging to a module caused due to illegal write operations made by some another module. We create and enforce *Protection Domains* within data memory address space of the embedded processor. Protection domain refers to a fragmented but logically distinct portion of overall data memory address space (Figure 2). Every module stores its state in its own protection domain. No assumptions are made about layout of state within a domain. Modules are restricted from writing to memory outside their domain through run-time checks. There is one single trusted domain in the system that is allowed to access all memory.

Protection models based on domains do not address all possible memory corruption faults in the system. Modules can still corrupt their own state as it resides completely within a protection domain. This form of corruption, though undesirable, is less serious than corruption across domains. If we have an operating system we can load the kernel image in its separate domain. A stable kernel can always ensure a clean re-start of user modules when corruption is detected.

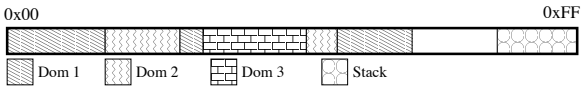


Figure 2: Protection Domains

2.2 Memory Map Data Structure

Creating and enforcing protection domains is a challenging task on resource constrained embedded platforms. Limited memory prohibits static contiguous partitioning of address space into multiple domains. Instead we partition the address space of the microcontroller into blocks of equal sizes. A *Block* is a small contiguous region of memory. Memory is allocated to domains as *segments*, which are simply sets of contiguous blocks. Allocation of segments to domains could be static (at compile time) or dynamic (through a memory heap). A domain could be allocated multiple segments that are scattered randomly across entire address space. *The Memory Map contains access permissions for every block of address space.* The Memory Map specifies two pieces of information. First, it contains ownership information (domain identity) for every block of memory. Second, it encodes information about memory layout such as start of a logical segment of allocation to programs. An example of actual encoded information and their meaning is specified in Table 1.

Code	Meaning
1111	Free or Start of Trusted Segment
1110	Later portion of Trusted Segment
xxx1	Start of Domain (0 - 6) Segment
xxx0	Later portion of Domain (0 - 6) Segment

Table 1: Encoded information in memory map table for multi-domain protection

2.3 Memory Map Checker

A memory map checker is required to validate memory accesses made by software components. It enforces the protection model that we described earlier; programs can write only into their domain. The memory map checker is implemented as a functional unit (MMC) that intercepts the signals generated by the CPU for writing into the data memory (Figure 3). If the write address is valid the MMC writes directly into the data memory.

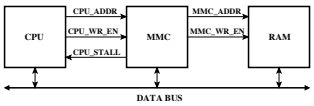
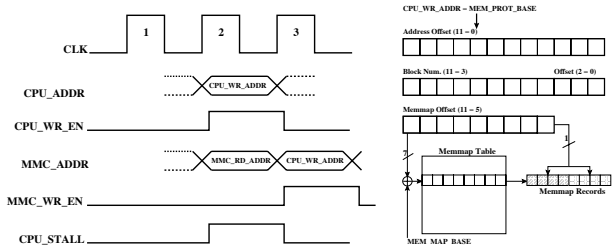


Figure 3: Memory Map Controller (MMC)



(a) Timing (b) Addr Translate

Figure 4: MMC Operations

The operations performed by MMC are three-fold. First, it stalls the processor execution and takes control of the address bus to memory. This occurs in the second cycle of the clock waveform shown in Figure 4(a). In the same clock cycle it performs an address translation operation to determine the address of the permissions in the memory map. Address translation is shown in Figure 4(b). Memory map permissions are also read in this cycle as the MMC unit has control over the address bus. Second, the MMC compares the ownership information to the identity of the current executing domain. Finally, if the check is successful, then the MMC issues a write enable signal to the data memory.

The subset of address space protected by the memory map is defined by the register pair `mem_prot_bottom` and `mem_prot_top`. The first step during translation is to determine the offset of the write address into the protected address space. This is done by subtracting the lower bound of protected memory address space from the issued write address. Assuming a block size of 8 bytes, the nine significant bits of the address offset represent the block number. Permissions are packed into a byte. If the encoded information is stored in four bits (assuming multi-domain protection), then each byte would contain information of two contiguous memory blocks. Therefore the last bit of the block number represents the byte offset of the permission. The remaining bits index into the Memory Map Table. The base pointer of the Memory Map Table is stored in a special register called `mem_map_base`. The address of the permissions byte is computed by adding the memory map index to the memory map base pointer.

The Memory Map data structure is configurable through a set of programmable registers shown in Table 2. The registers are accessible only by the run-time library loaded in the trusted domain. The `mem_map_config` register is used to configure the block size and the number of protection domains available in the system.

2.4 Memory Map Software Library

The software library manages all the memory available on the embedded processor. First, it ensures that the memory map accurately reflects current ownership and layout. In any real system, memory is constantly allocated, de-allocated or transferred from one module to another. The Memory Map

Register	Function
`mem_map_base`	Memory map base pointer
`mem_prot_bot`	Lower bound of protected address space
`mem_prot_top`	Upper bound of protected address space
`mem_map_config`	Configure block size and domains

Table 2: Memory Map Configuration Registers

should be immediately updated when any of these events occur. The library provides `malloc`, `free` and `change_own` calls that automatically update the Memory Map data structure. Second, it only permits the block owner to free or change its ownership. This condition is necessary as one module may (due to programming errors) free up memory that is being used by other module in the system. Also it prevents a module from accidentally hijacking memory that is owned by other modules. To enforce this condition, the software library reads the identity of the current active domain from the status register. Third, the software library sets up the memory map to be located in a protected region of memory. This prevents accidental corruption of the Memory Map data structure. It is the responsibility of the software library to ensure that a memory map of sufficient size is allocated in the system. Fourth, it initializes the MMC with the appropriate block size, number of protection domains and the range of protected address space.

3. CONTROL FLOW MANAGER

Programming errors can cause a module to corrupt its own state. Protection domains created and enforced by the memory map manager cannot prevent such internal memory corruption. Control flow within a system can be affected by internal memory corruption. For example, function pointers (commonly used to implement callbacks) are stored in RAM. Return addresses to function call-sites are stored in the stack. Corruption of these values can cause the processor to execute arbitrary code belonging to the trusted domain. The Control Flow Manager ensures that control can never flow out of a domain except via calls to functions exported by other domains and via returns to calls from other domains. Conversely, control flow can enter a domain only through an exported function or through the return site of a call that is made to a function exported by some other domain. In addition, the identity of the current domain (that is executing) also needs to be tracked. This information is required by the memory map checker to validate write accesses. Control flow integrity within a domain is preserved through the safe stack that stores return addresses.

3.1 Cross Domain Linking

A module loaded in a domain exports a set of functions that can be validly called by modules in other domains. A linker parses the set of functions exported by a domain and writes them to a *jump table* in flash memory. The jump table is similar in design to the processor interrupt vector table. Each entry in the jump table is an instruction to jump to a valid exported function. Each domain has its own jump table that contains all functions that it exports. Modules are not allowed to directly write to flash memory and therefore the jump table cannot be corrupted. Modules that subscribe to functions exported by a particular domain are re-directed through the jump table of that domain. This is illustrated in Figure 5. The jump table mechanism is independent of

the process used for dynamic linking i.e. exporting and subscribing to functions. Linking could be done statically or dynamically on the embedded processor [3].

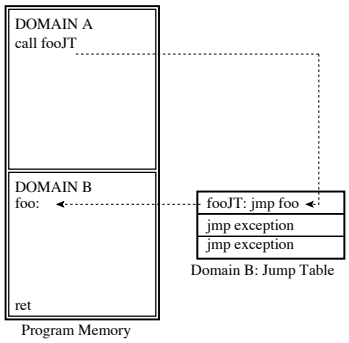


Figure 5: Cross Domain Linking

3.2 Domain Tracking

Domain tracking is performed in hardware by extending the implementation of `call` and `return` instructions. Each domain is allocated one complete page of flash memory to store its jump table. In the AVR architecture this imposes a limit of 128 functions that can be exported by every domain. This limit can be easily extended by allocating more space to the jump table. Empty entries in the jump table are filled with a jump instruction to an exception routine. Jump table pages of all domains are co-located and stored at fixed location in flash memory.

This organization simplifies the algorithm for verifying the target address of a call. A valid target address has to reside in the jump table. This is checked by a simple compare operation to the base address of the jump table. The check against the upper bound of jump table is deferred.

The identity of the called domain is also easily determined. Jump tables of all domains are organized linearly, starting from the domain 0 jump table located at the base address. The identifier of the target domain can be easily determined by first computing the address offset from the base address of the jump table and dividing it by the size of the jump table. If the target domain identifier exceeds the maximum number of domains in the system, then it indicates that the target address is greater than the upper bound of the jump table, and an exception is generated. Finally, a call is made into the jump table that is redirected to the actual entry point in the target domain.

The current domain identifier needs to be pushed to a stack, because cross domain calls can be chained: domain A calls domain B which in turn calls domain C. During cross domain return, the previous domain identifier is restored and the control is transferred back to the caller's domain. The cross domain state machine handles the push and pop operations transparently to the application programs.

3.3 Run-Time Stack Protection

Embedded micro-controllers have a single execution stack that is shared by the entire system. In most architectures, the stack is initialized at the end of address space and grows down towards the start of address space. The run-time stack is used for many purposes. First, it is used to record the return addresses of function calls. Second, it is used to set up data frames for storing local variables or function arguments that cannot be accommodated in registers. Third, it is also used to store arguments for variadic functions. Stack corruption is a serious problem. Our protection model prevents corruption of the stack belonging to one domain by any module belonging to a different domain. During a cross domain call the processor copies the current stack pointer into a `stack_bound` register. The previous stack bound is saved. The memory map checker compares the write address to the current stack bound and signals an exception if the address exceeds the stack bound. Therefore, modules belonging to a domain cannot corrupt the stack belonging to another domain.

3.4 Safe Stack

A module can call any local function within its domain. The return address of function calls are stored in stack and are protected from corruption from modules in other domains. However, a programming error can cause a module to corrupt its own stack. This cannot be prevented by protection domains. Therefore, we store all return addresses in a separate stack that resides in a different protection domain. We call this a safe stack. A safe stack can be setup only by the software in the trusted domain by writing to the `safe_stack_ptr`. The safe stack can be setup anywhere in data memory as long as it is protected from accidental writes and overflow. We usually setup Safe Stack at the end of all global data in the system and make it grow upwards. Run-Time stack and Safe Stack approach one another.

4. EVALUATION

In this section, we will analyze the protection benefits and overheads introduced by our methodology. We have implemented the hardware components of our design by making extensions to the AVR instruction set architecture. The VHDL model of the extended processor is synthesizable. We have instantiated the processor on Xilinx Vertex 2 Pro XC2VP30 FPGA. Our performance overheads are measured using Modelsim 6.0 simulator. The software library and applications were compiled using `avr-gcc` cross compiler.

4.1 Performance Overhead

We first present micro-benchmarks that measure CPU overhead introduced by the protection mechanism. Table 3 contains the overhead of run-time checks present in our mechanism. We compare our overhead with a completely software based approach to memory protection through binary rewrites proposed in [8] for the AVR architecture. The software based approach also introduces identical run-time checks except that they are implemented in assembly language without any modifications to the processor architecture. The results clearly indicate the superior performance of run-time checkers implemented in hardware.

Function Name	AVR Extension	AVR Binary Rewrite
Memmap Checker	1	65
Cross Domain Call	5	65
Cross Domain Ret	5	28
Save Ret Addr	0	38
Restore Ret Addr	0	38

Table 3: Overhead (CPU cycles) of Memory Protection Routines

The high overhead of software based memory map checker is mainly due to complex bit shift operations that are required to translate write addresses to memory map lookup. Cross domain call and return have an overhead of five clock cycles when implemented in hardware. The overhead occurs because the current domain identity, stack bound and return address have to be pushed to the safe stack before they can be updated with new values. The total information that needs to be pushed to the stack is five bytes and only one byte can be written every clock cycle. Similarly on the cross domain return, the five clock cycles are expended in restoring the values read from the safe stack. Saving and restoring return addresses to the safe stack does not introduce any added overhead. This is because the hardware unit for safe stack simply takes over the address bus when the processor is pushing the return address to the run-time stack. By stealing the address bus from the processor, the hardware unit is able to simply redirect the store of the return addresses to the safe stack.

Next we evaluate the software library. Performance overhead is also introduced by updates to memory map during allocation, free and transfer of memory within the system. Table 4 compares the overhead of memory allocation routines in the presence and absence of the protection mechanism. Relatively higher overhead of `change_own` and `free` calls is due additional checks that are introduced to prevent illegal ownership transfer or freeing of memory blocks by non-owners.

Function Name	Normal	Protected
malloc	343	610
free	138	425
change_own	55	365

Table 4: Overhead (CPU cycles) of memory allocation routines

4.2 Resource Utilization

We implemented our system on a AVR Atmega103 processor. This contains 4KB of RAM and 128KB of PROM. Resource utilization can be partitioned into sections: overhead of the software library and the overhead of the hardware checkers.

Code and data memory usage of the software library is shown in Table 5. Maximum memory map size is 256 bytes for multi-domain protection. This represents an overhead of 6.25%. However, by modifying data layout, portion of address space that requires memory map for protection can be reduced. For example, memory map can be configured only to protect the heap and safe-stack. By abutting these two data-structures, size of memory map required can be reduced to 140 bytes for multi-domain protection. For two domain protection, the overhead can be reduced to only 70 bytes (1.7%). The total code memory usage of the software library is only 3674 bytes (2.8%).

SW Component	FLASH (B)	RAM (B)
Dynamic Memory	1204	2054
Memory Map	422	256
Jump Table	2048	0

Table 5: FLASH and RAM overhead of software library

The hardware overhead of our mechanism is shown in Table 6. These results were computed by synthesizing our processor on Xilinx ISE 8.2i. Most of the additions to the core area are in the memory map decoder that maintains a barrel shifter to support arbitrary bit-shifts in a single clock cycles. We can eliminate this overhead if the processor is synthesized for a fixed block size and number of protection domains. The overall increase in the core area is about 32%. This represents a modest increase in the overall area of the chip as the core occupies only a small fraction of the overall area. Bulk of the chip area is occupied by SRAM and FLASH memories.

HW Component	Ext. Gate Count	Orig. Gate Count
AVR Core	22498	16419
Fetch Decoder	6783	6685
MMC	2284	N/A
Safe Stack	1749	N/A
Domain Tracker	541	N/A

Table 6: Gate count overhead of hardware extensions

5. RELATED WORK

Page-based virtual memory systems have become the dominant form of memory management in the modern general-purpose computer systems. While the process model of the virtual memory systems delivers protection for embedded applications, it also increases the overhead in memory consumption and processor performance. The memory consumption increases due to the need to store address translation tables. The processor performance is impacted because context switches have a high overhead; page tables have to be setup for the new context. To improve performance of virtual memory, architectural features such as Translation Lookaside Buffers (TLBs) and virtual-mapped caches are used that further increase the area, cost and complexity of the chip. For example, the addition of MMU and cache in an ARM7TDMI core [1] increases its area ten fold and its power consumption two fold. Therefore, current MMU designs will never be used in the low end price sensitive microcontrollers.

Memory protection units (MPU) provide hardware assisted protection in embedded processors such as ARM 940T [7] and Infineon TC1775 [13]. MPU can statically partition memory and set individual protection attributes for each partition. The partitions are contiguous segments within the address space defined by a pair of base and bounds registers. The protection model of MPU is not suited for the complex embedded software (such as operating systems) running on low-end microcontrollers. MPU defines only two protection domains viz. User-mode and Supervisor mode. This is sufficient for protecting the kernel from the applications but not the applications from one another. The static partitioning of address space into contiguous regions is infeasible for the low-end microcontrollers with very limited memory footprint. Further, the number of partitions is also limited. However, MPU has a lower memory footprint than UMPU because the partitioning information can be stored in registers instead of maintaining a memory map. MPU introduces no performance overhead while UMPU incurs a single clock cycle penalty for memory map accesses.

Mondrian Memory Protection (MMP) [15] inspects memory accesses at the instruction level from within the proces-

sor pipeline to provide word-level protection. It uses fairly complex and expensive hardware extensions to reduce overhead of monitoring all accesses. SafeMem [12] exploits existing ECC memory protection to guard memory regions and detect any illegal accesses through ECC violations. However, these techniques require significant resources to be performed on tiny embedded processors.

Hardware support for memory safe execution of embedded software was recently proposed in [2]. This technique uses CCured [11], a tool that generates type safe C programs through pointer inference techniques. Extensions to the instruction set architecture speed up the run-time bounds checking operations performed by CCured. Our techniques apply directly to machine instructions and are therefore agnostic to programming languages. Also, our hardware extensions do not modify the processor instruction set architecture. Hence, we can continue to use existing compilers. Custom modifications to compilers can become a source of new bugs.

Many software based approaches for memory protection have been proposed. Type-safe languages such as Virgil [14] can flag illegal accesses at compile or run-time. They provide fine-grained memory protection of individual objects. Type-safe languages do not interface with code written in non type-safe languages. However, most of the software developed for embedded systems is written in unsafe languages such as C (or even assembly for low-level drivers). Popular programming language NesC [4], contains minimal extensions to C (such as the `atomic` keyword) to prevent race-conditions that can cause memory corruption. ASVM [9] can also be used for providing memory protection. Software-based fault isolation for embedded processors has been proposed in [8]. All the software based approaches have a significantly higher overhead than custom hardware extensions.

6. CONCLUSION

In this paper, we have proposed a hardware software co-design approach for providing memory protection in tiny embedded processors. Though we have implemented the protection technology for the AVR microcontroller, our general approach is applicable to other RISC architectures such as TI MSP or ARM. Through a careful partitioning of the protection techniques, we have significantly improved performance by moving compute intensive operations into hardware. Our hardware is very flexible, it can accommodate various configuration parameters. The software library provides a standard programming interface. Moreover, our approach does not modify the instruction set architecture of the processor; hence we do not need to modify the cross compiler. These features ensure that our software library can be incorporated into existing projects with minimal modifications; a very practical benefit to the system developers. We are still exploring the design space of possible protection architectures. The resource utilization of our design can be further reduced by synthesizing hardware units that are pre-configured for a particular block size and number of protection domains. An interesting area of future work is to explore software techniques such as virtual machines or type-safe languages that can benefit from modest hardware extensions. Software reliability is an emerging concern in the domain of tiny embedded processors. Limited resources preclude the application of existing approaches used in desktop processors. We believe that hardware software co-design techniques are a promising avenue to explore for creating robust software for tiny embedded processors.

7. REFERENCES

[1] ARM7TDMI Technical Reference Manual. http://www.arm.com/pdfs/DDI0210C_7tdmi_r4p1_trm.pdf.

[2] D. Arora, A. Raghunathan, and N. K. Jha. Architectural support for safe software execution on embedded processors. In *CODES+ISSS '06: Proc. 4th International Conference on Hardware/Software Codesign and System Synthesis*, 2006.

[3] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt. Run-time dynamic linking for reprogramming wireless sensor networks. In *SenSys '06: Proc. 4th ACM Conference on Embedded Networked Sensor Systems*, 2006.

[4] D. Gay, P. Levis, R. von Behren, and M. Welsh. The nesC language: A holistic approach to networked embedded systems. In *PLDI '03: Proc. ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, 2003.

[5] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. SOS: A dynamic operating system for sensor networks. In *MobiSys '05: Proc. 3rd International Conference on Mobile Systems, Applications, and Services*, 2005.

[6] J. Hill and D. Culler. Mica: A wireless platform for deeply embedded networks. In *IEEE Micro.*, volume 22, pages 12–24, Nov/Dec 2002.

[7] A. Inc. *ARM 940T Technical Reference Manual.*

[8] R. Kumar, E. Kohler, and M. Srivastava. Harbor: Software based memory protection for sensor nodes. In *IPSN '07: Proc. 6th International Symposium on Information Processing in Sensor Networks*, 2007.

[9] P. Levis, D. Gay, and D. Culler. Active sensor networks. In *NSDI '05: Proc. 2nd Symposium on Networked Systems Design and Implementation*, 2005.

[10] P. Levis, D. Gay, V. Handziski, J. H. Hauer, B. Greenstein, M. Turon, J. Hui, K. Klues, C. Sharp, R. Szewczyk, J. Polastre, P. Buonadonna, L. Nachman, G. Tolle, D. Culler, and A. Wolisz. T2: A second generation OS for embedded sensor networks. Technical Report TKN-05-007, Telecommunication Networks Group, Technische Universität Berlin, 2005.

[11] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *POPL '02: Proc. 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002.

[12] F. Qin, S. Lu, and Y. Zhou. Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs. In *International Symposium on High-Performance Computer Architecture (HPCA)*, 2005.

[13] I. Technologies. *TC1775: 32-Bit Single Chip Microcontroller.*

[14] B. L. Titzer. Virgil: Objects on the head of a pin. In *OOPSLA '06: Proc. 21st ACM SIGPLAN Conference on Object-Oriented Systems, Languages, and Applications*, 2006.

[15] E. Witchel, J. Cates, and K. Asanović. Mondrian memory protection. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

[16] Zigbee Consortium. www.zigbee.com.