t-kernel: Providing Reliable OS Support to Wireless Sensor Networks

Lin Gu
Department of Computer Science
University of Virginia
lingu@cs.virginia.edu

John A. Stankovic
Department of Computer Science
University of Virginia
stankovic@cs.virginia.edu

Abstract

The development of a reliable large-scale wireless sensor network (WSN) is very difficult because of resource constraints, energy budget, and demanding application requirements. Three OS features—OS protection, virtual memory, and preemptive scheduling—can significantly improve the reliability of WSN systems and facilitate developing complex WSN software. However, due to the lack of hardware support for privileged execution and address translation, it is impossible to implement these features with traditional OS design techniques. To solve this problem, we design a new OS kernel, the *t-kernel*, to perform extensive code modification at load time. The modified code and the OS work in a collaborative way supporting the aforementioned features. Having implemented the *t-kernel* on MICA2 motes, we evaluate its performance by measuring the overhead and execution speed. We analyze the CPU utilization of sensor network applications, and verify that, though CPU-bound tasks execute 1.5–3 times as long as in native mode, application performance under typical workloads does not noticeably degrade. The *t-kernel* significantly enhances developers' ability to design reliable and sophisticated sensor networks, and includes several new design techniques, such as efficient binary translation on highly constrained sensor nodes, differentiated virtual memory without repeatedly writable swapping devices, and the protection of the OS from application errors without privileged execution hardware.

Categories and Subject Descriptors

D.4.7 [**Operating systems**]: Organization and Design

General Terms

Design, Reliability, Performance

Keywords

Wireless Sensor networks, OS Protection, Virtual Memory, Binary Translation, Low-Power Systems

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SenSys'06, November 1–3, 2006, Boulder, Colorado, USA.
Copyright 2006 ACM 1-59593-343-3/06/0011 ...$5.00

1 Introduction

Many wireless sensor networks (WSNs) are based on mote-class devices, e.g., the Berkeley motes [21]. These devices include low-power microcontrollers and very small amounts of data memory (3KB–10KB RAM). Energy efficiency and cost are very important factors so we will continue to see highly constrained devices for many WSNs—likely with increasingly small form factors. While some WSNs will indeed migrate to more expensive and capable devices, this migration does not diminish the need for inexpensive devices. For example, if costs were not such an important factor then we would not see that today more than 98% of all processors are embedded microcontrollers.

Even though each device has limited CPU, memory and energy, systems being built with large numbers of these devices, often running in an unattended manner, can be quite sophisticated. Currently, the OS' support to application development and run-time execution is very limited. OS protection (protect the OS from application errors), virtual memory, and preemptive scheduling are three features that can significantly improve WSN systems' reliability and lower the development cost. But it is difficult to provide these features on sensor nodes. The reason is that many microcontrollers do not provide related hardware support, e.g., privileged execution and virtual address translation.

We design a new OS kernel, the *t-kernel*, to support the aforementioned features using a load-time modification approach. The kernel modifies the necessary native instructions when it loads the application's instructions and dispatches them for execution. Without assuming advanced hardware support or a trusted compiler, the modified program guarantees OS control against faulty application code, performs preemption at 16 priority levels, and supports a 64KB virtual memory space over 4KB physical memory. With the inherent advantages that these mechanisms impart, the *t-kernel* raises the level of system abstraction visible to application programmers.

Promoting the reliability of real-world WSN systems, the *t-kernel* is a novel solution in a particular design context. It has multiple contributions to the research on system software design for WSNs, including efficient binary translation with very small memory, efficient software based virtual memory, and OS protection without memory protection or privileged execution hardware.

The paper is organized as follows. Section 2 analyzes the requirements and examines why stronger OS support is

necessary for WSNs. Section 3 states the assumptions and briefly describes the *t-kernel*'s approach. Then, Section 4 details the design of the *t-kernel*, followed by Section 5 which introduces the implementation on a widely used WSN platform. Section 6 evaluates the performance of the *t-kernel*. After Section 7 discusses related work, Section 8 lists the limitations of the *t-kernel* and briefly discusses future work.

2 Motivation

To generalize major application requirements and illustrate the need for advanced OS features, we study two real-world scenarios and illustrate the difficulties researchers encounter when developing high-quality WSN systems.

Scenario 1: OS control

The Extreme Scaling [10] project studies large-scale WSNs of more than 1,000 nodes. Such a network can be deployed by aircraft in an inaccessible area and operate in an unattended manner. For maintenance, it is essential to guarantee that the nodes always respond to wireless control requests, such as reprogramming commands. However, such a guarantee turns out to be difficult to accomplish because the OS and application share the same memory space and have the same privilege. A faulty application can grab the CPU and prevent the OS from processing any control requests.

In this scenario, we see an interference between the OS kernel and application code. As a solution, the Extreme Scaling project employs a sanity operation by hardware—a grenade timer automatically restarts the sensor node periodically [10]. After restarting, the boot loader listens to the wireless channel before executing application code. Periodic restarts ensure that the administrators can reprogram the network, but complicate application design because the application must expect and handle such periodic rebooting. Moreover, it has a fairly coarse control granularity. Not wanting a node to restart too frequently, we tend to set a longer grenade timer interval. This means that, in a relatively long time between two restarts, the OS control is not guaranteed.

Scenario 2: memory shortage

VigilNet [18] is a large-scale surveillance network performing target tracking and classification. For adaptive operations in realistic environments, it has more than 30 middleware services (e.g., fault-tolerant routing, power management, and signal processing) and consists about 40,000 lines of code. However, the sensor nodes in VigilNet have only 4KB RAM, which by no means can support what the application needs.

In this scenario, the application requires more resources than the hardware platform provides. A possible solution is for the designers to re-use the data memory—to analyze when portions of memory are not used any more and assign them to functions currently running. However, such a programmer-controlled memory overlay scheme is a deprecated method because it is application specific, inefficient, and error-prone. Some recent sensor nodes have 8KB–10KB RAM, but memory scarcity is still limiting the development of sophisticated WSN software.

The two scenarios discussed above illustrate difficulties programmers encounter while developing large-scale real-world WSN applications. In PCs and servers, these problems do not exist because the hardware is much more advanced. For example, the PC hardware supports privileged instructions. Hence, the OS can control the computer periodically by exclusively handling clock interrupts.

If the OS supports OS protection and virtual memory, the difficulties illustrated in these two scenarios are neatly solved. In addition, preemptive scheduling can provide a vehicle to significantly improve the quality of signal processing and communication, which are important operations in WSNs. These observations motivate us to design the *t-kernel* to support these three features, and the former two, OS protection and virtual memory, are the focus of this paper.

Traditionally, the three OS features mentioned above rely on advanced hardware support—privileged instructions, virtual address translation, and memory protection. Such hardware support is absent on many microcontrollers used for WSNs. In the past few years, some high-end sensor nodes have scaled up with the technology progress to employ better processors or co-processors. Examples include the BTnode (ATmega128L, 240KB external SRAM), the Stargate (XScale, 64M RAM), and the Intel Mote (ARM or XScale, 32M RAM) [3, 30]. However, many low-end sensor nodes still stay with simple hardware without advanced architectural features [10, 12, 26]. This is also true for some recent microcontrollers designed specifically for sensor networks.

One important reason for the lack of advanced hardware features is sensor nodes' very low power consumption. While a Pentium 4 processor can easily consume 90W power, and a CrossBow Stargate core consumes about 1W [28], some low-end sensor nodes have a power budget of only 0.06W. When energy efficiency is emphasized, the system designer prefers to use simple hardware and barely enough memory. Even when high-end sensor nodes are used, it is not unusual that the designers incorporate low-end sensors to form a hybrid network [22].Besides energy efficiency, the requirements of small form factor and low cost also contribute to the modest hardware configuration of very-low-power sensor nodes [27, 22]. SRAM often occupies a large area and uses a significant transistor count in a System-on-Chip (SoC) processor. For example, the SRAM in a high-end embedded processor (Intel XScale) uses 60% of the area and 90% of the transistor count [6]. In a low-end microcontroller, SRAM can account for 76% of the transistor count [8]. Currently, the form factor has not been an as severe constraint as energy. However, if the RAM size keeps increasing, it may become a serious concern very soon.

In general, with energy, form factor, and cost being emphasized considerations, it is undesirable, if not impossible, to upgrade the hardware configuration. Hence, the technology development does not diminish the class of low-end sensor nodes featuring simple hardware and very low power consumption. This class of platforms include many widely-used sensor nodes, including RENE, MICA, MICA2, ExScal, MICAz, and so on.

While the hardware is modest, the application requirements are demanding. In a realistic environment, a sensor network with energy-and-cost-efficient sensor nodes needs to be a distributed, fault-tolerant, and adaptive network performing a wide range of services, including topology control, routing, aggregation, network management, power management, security, and maintenance [18, 13, 10]. Inevitably, there is a wide gap between the application complexity and

the hardware simplicity. This is the intrinsic reason for the difficulties illustrated in the two scenarios discussed before.

3 Assumptions

Compared to PC industry, microcontrollers used in WSNs and embedded systems have a much wider variety. To ensure wide platform support and facilitate future porting, we choose the following assumptions on the hardware.

- Assumption R: Reprogrammable—The system allows writing data into some memory space and executing it.
- Assumption E: External nonvolatile storage—Low-power, nonvolatile, and relatively large-capacity external storage is available, and it supports fast and repeated read operations (read-friendly).
- Assumption M: Memory—A certain amount of RAM is available. Hence, we are excluding embedded processors with only registers. To facilitate efficient indexing and swapping, we recommend no less than 4KB physical memory be available.

From now on, we call computer systems that meet these three assumptions **REM computers**. They represent a wide range of systems in the area of WSNs and, generally, embedded systems. Also, in the rest of the paper, we use flash as a representative of nonvolatile storage, but the discussions apply to general read-friendly nonvolatile storage devices.

4 Design

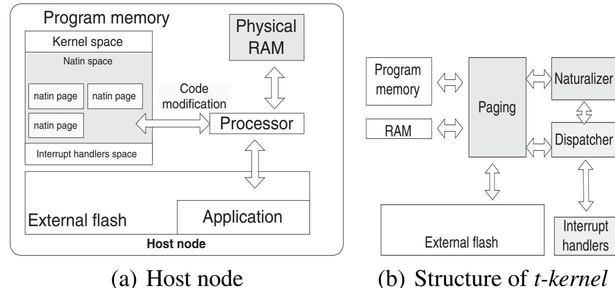
This section introduces the design of the *t-kernel*. First, we give a high-level description of the *t-kernel* and define terminology in Section 4.1. Then Section 4.2 and 4.3 discuss the CPU control and virtual memory in detail. Finally, Section 4.4 discusses the interface between the kernel and application.

4.1 Overview

There are three sources of challenges in designing *t-kernel* for a REM computer: stringent resource constraints, possibly write-unfriendly (limited erasure/write cycles) external nonvolatile storage (e.g., flash), and lack of hardware features (e.g., privileged execution). With these challenges, it is impossible to use traditional OS design techniques to implement OS protection, virtual memory, and preemptions on REM computers. A new design is needed.

Figure 1(a) shows the hardware and software components in a WSN platform, henceforth called a host node. The *application* is a binary program in the sensor node's instruction set, and resides in the external flash. After initializing its own working environment, the *t-kernel* loads and modifies the instructions in the application, and dispatches the modified instructions for execution.

We roughly define a small block of consecutive instructions in the application as a code page. When the control flow reaches a new code page, the *t-kernel* reads that page from the flash and modifies some of its instructions to assure they run on the host node in a collaborative manner. We call such a process **naturalization**, and the modified code *naturalized instructions*, or **natins**. Naturalization is one type of binary translation. We use this term to denote the instruction-level code modification the *t-kernel* performs on a small-memory platform to significantly enhance system abstraction. During naturalization, only a fraction of instructions in the application code are changed to different types and numbers of



(a) Host node (b) Structure of *t-kernel*

Figure 1. The host node and *t-kernel*

instructions. Most of the natins are the same as their corresponding instructions in the application code. Natins form a new, collaborative "naturalized program", which guarantees not to compromise the host node, and, hence, is trusted and executed without restrictions.

Blocks of natins are organized into a *natin page* of a fixed size (256 bytes on MICA2). Natin pages reside in the natin space in the program memory. When one natin branches to or calls an address in the application, the *t-kernel* helps transfer the control flow. An address in the application is called a **VPC** (virtual program counter), which is defined by a compiler or a programmer. Meanwhile, we call an address in the naturalized program an **HPC** (host program counter). If there is no natin page for the destination VPC, the *t-kernel* reads the corresponding code page and naturalizes it, then transfers control flow to the HPC (in the new natin page) corresponding to the destination VPC. This procedure implies that naturalization is an incremental, on-the-fly process. Usually, a natin page is naturalized only once. An exception is the "bridging" process, to be discussed in Section 4.2.2, which re-writes some branch destination addresses.

The *t-kernel* module shown in Figure 1(a) can be further dissected into several components, as illustrated in Figure 1(b). The *dispatcher* controls the execution of the natins and some *t-kernel* routines (such as sanity check routines). When new application code is reached, it invokes the *naturalizer* to perform code modification. The *paging* module interacts with various memory and storage devices on the host node. It handles swapping in and out among the RAM, the program memory, and the external flash.

4.2 Naturalization and CPU control

CPU control and OS data integrity are two aspects of OS protection. The former means the OS kernel must be able to take hold of the CPU to execute, and the latter means that the kernel must execute with valid data. In this section, we introduce the naturalization process and explain how it guarantees OS control. The kernel data protection is introduced in Section 4.3 with virtual memory.

4.2.1 Kernel/application transitions

Traditionally, the CPU control is guaranteed by privilege support and clock interrupts. However, many microcontrollers used by sensor nodes do not have privilege support. The application can disable interrupts and occupy the CPU for an arbitrarily long time.

The *t-kernel*'s approach is to modify the application program so that the naturalized program yields CPU to the kernel frequently. The dispatcher keeps track of the current VPC and dispatches the corresponding natin page. The VPC be-

gins with the starting address of the application. When a natin page is executed, it guarantees a return to the dispatcher with information about the next VPC.

To guarantee that the execution of a natin page returns to the dispatcher, the naturalizer modifies all branching instructions, including branch instructions, calls, returns, and skip instructions. As an example, on the Berkeley MICA2 platform, an unconditional branch, "jmp DEST", is modified as follows.

```
push r31;       save r31
push r29;       save r29
in r29, 0x3f;   acquire CPU's state flags
push r29;       save CPU's state flags
push r30;       save r30
push r28;       save r28
ldi r31, DEST3; load bits 24-31 of DEST
ldi r30, DEST2; load bits 16-23 of DEST
ldi r29, DEST1; load bits 8-15 of DEST
ldi r28, DEST0; load bits 0-7 of DEST
jmp homeGate;   jump to homeGate
```

Here DEST0 - DEST3 are byte0 - byte3 of the destination VPC, and homeGate points to the *welcomeHome* routine in the dispatcher. The *welcomeHome* routine retrieves the destination VPC (DEST in this example) from r28–r31, seeks for a natin page that services this destination VPC, creates a new natin page if no suitable one exists, and transfers control flow to the start address (HPC) of the natins corresponding to the destination VPC.

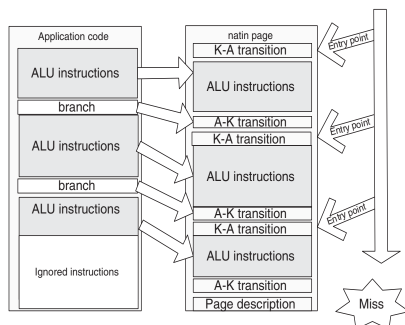


Figure 2. Application code page and natin page

When the kernel has prepared the natin page servicing a destination VPC, the dispatcher needs to transfer the control flow to the **entry point**—the HPC in the natin page that corresponds to the destination VPC. A straightforward solution similar to traditional context switching is to restore register r28–r31 and machine state flags from the stack, and then point HPC to the entry point. This method is used in some binary translation systems on high-power computers. However, three concerns—RAM size, performance, and code density—lead us to adopt a different approach.

First, the RAM size is small. Hence, we cannot put all the information about the destination addresses in RAM. Instead, we store it on the natin pages themselves.

Second, in a natin page, there can be multiple entry points corresponding to multiple VPCs (see Figure 2). To improve performance, we do not want to look up the accurate position of an entry point from the natin page, because it takes a few dozen instructions. Instead, we prefer to directly jump to the natin page. Hence, we prefix each entry point in a natin page with "cascading branch" logic—it checks if this entry point services the destination VPC, and continues to execute if it does; otherwise, it jumps to the next entry point. Therefore, the seek for a matching entry point is conducted by a **cascading branch chain** on a natin page. The cascading branches have to use at least one register, thus we cannot restore registers in the kernel, but have to do it in the natin page, after the cascading branch chain matches an entry point.

Third, to increase code density, we prefer not to put all the register-restoring instructions in the natin page.

Considering all the above concerns, the *t-kernel* adopts the following procedure for returning to application code:

```
1. r29 = destination VPC's last 6 bits (by kernel)
2. restore r28 (by kernel)
3. Register indirect jump to the natin (by kernel, using r30--r31)
4. Restore r30 (by natins)
5. Execute the cascading branch chain (by natins)
6. Restore the status register, r29, and r31 (by natins)
```

When jumping to the natin page, the *t-kernel* makes sure there is no ambiguity on the high bits of the destination VPC. Hence, a match on the low bits in r29 ensures a hit. At step 5, the cascading branch chain stops when there is a match (hit). If the cascading branch chain does not match any entry point, it jumps back to kernel.

Illustrated in this design, multiple constraints interact and make the *t-kernel* adopt a protocol different than traditional kernel traps/returns to jump back and forth between the kernel and the application. The invocation of kernel services and the returning to the application logic involve a variable sequence of instructions distributed in the kernel and natin pages collaborating to perform a search on the cascading branch chain. To be clear, we call such a process a **kernel transition**, and, specifically, a transition from application to kernel or from kernel to application an **A-K transition** or a **K-A transition**, respectively (Figure 2).

4.2.2 Branch regulating

After modifying branching instructions to A-K transitions, the kernel is guaranteed to get hold of the CPU very frequently. However, the transitions involve overhead and can significantly reduce the computation speed because branch instructions are common. In one test, we observe that unoptimized A-K transitions slow down a program by 30 times. To promote performance, the *t-kernel* performs a bridging process to directly link the branch source and destination. The A-K transitions and the bridging process form a technique called **branch regulating** in the *t-kernel*. With branch regulating, the execution speed of branch instructions is accelerated, but no loops in the application could take hold of the CPU infinitely. Hence, the CPU control by the OS is guaranteed.

The *t-kernel* classifies the transitions into several types and handle them differently. One type of A-K transitions, called "town transitions", is designed to speed up branches. The first time a town transition occurs for a specific VPC, the dispatcher invokes the naturalizer to perform the bridging operation: the naturalizer modifies the A-K transition at the source natin page to be a direct jump to the corresponding HPC. Bridging is similar to "fragment linking" and "translation chaining" in other binary translation systems. However, in the *t-kernel*, the bridging operation handles backward taken branches (branches with the destination VPCs no larger than their own VPCs) differently than forward branches (branches with larger destination VPCs than their own VPCs). Backward taken branches are modified to

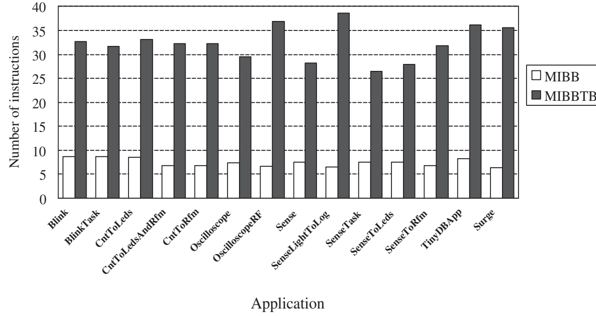


Figure 3. Branch density of WSN applications

increment an 8-bit system counter, and, if the counter reaches zero, call a special sanity check routine in the kernel before jumping to the destination HPC. Hence, for every 256 backward taken branches, one calls the kernel's sanity check routine, and yields the CPU to the kernel.

Among branching instructions, most of the branches (e.g., conditional branches and relative jumps) are modified to be town transitions. Their execution speed is close to that in native instructions when the branches are forward, and slower when they are backward taken branches. In typical WSN programs, we expect branches to be frequent, but backward taken branches are much less frequent. We analyze dynamic traces collected by Avrora [33] for 14 commonly seen TinyOS applications, and show in Figure 3 the mean number of instructions before a branch instruction is executed. These applications range from simple *Blink* to more complex *SenseToRfm* and *Surge* with sensing and radio communication. Two metrics, MIBB (Mean Instructions Before Branch) and MIBBTB (Mean Instructions Before Backward Taken Branch), are shown in the figure. The MIBB for the applications vary from 6 to 8 instructions, indicating that branch instructions are frequent. However, backward taken branches are much less frequent. The profiling shows that the applications execute 26–36 instructions before a backward taken branch. With such a low frequency, the overhead of branch regulating is kept low, and we accelerate the overall execution speed of branches by making the common case fast.

4.2.3 HPC/VPC look-up

When the dispatcher sees a new branch, or when the branch target's VPC is decided at run time (e.g., register indirect jumps), the dispatcher needs to find the HPC that corresponds to the destination VPC. Because the naturalizer performs code modification page by page following the execution order, the topology of the naturalized program becomes different than the original application program. The emphasis on execution speed prohibits the *t-kernel* from reorganizing the natin pages, and the resource constraints prohibit reserving space for application code that has not been executed. The code density is also changed after code modification. For these reasons, there is no linear relationship between the VPCs and HPCs. An efficient VPC-to-HPC look-up algorithm is a critical part in the kernel design.

The *t-kernel* performs the look-up at three levels, illustrated in Figure 4. The slowest, but most reliable level, is at the program memory itself. Each VPC is hashed to a number of natin pages, and each natin page's cascading branch chain tests all the entry points in the page. Hence, the *t-kernel*

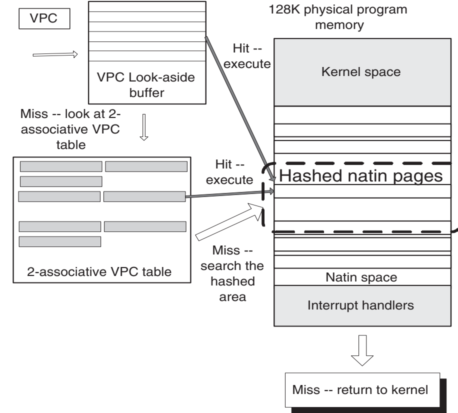


Figure 4. Three-level loop up for a VPC

looks up the natin page that services the VPC by looking at all the possible natin pages and checking whether any of them services the VPC. At the middle level, the kernel maintains a 2-associative VPC table, indexed by 8 bits in the VPC address, in RAM. Using only two bytes, each entry contains a tag for matching the high VPC bits, a frequency field, and a natin page index number. As an associative table, it offers a moderate speed and low miss rate. The fastest level is a VPC lookaside buffer. It is a direct-mapped buffer of 4-byte entries indexed by several bits of the VPC (6 bits in the current implementation). Direct-mapped, the lookaside buffer is fast, but has a higher miss rate than the 2-associative VPC table.

The sizes of the 2-associative VPC table and the lookaside buffer are configurable. Section 5 gives the numbers used in the implementation on MICA2.

4.3 Differentiated virtual memory

The *t-kernel* provides a virtual memory space much larger than the physical data memory to the application. The virtual-physical memory address translation, boundary check, and memory swapping are handled by natins without virtual memory and exception hardware.

4.3.1 Memory areas

It is a challenge to support fast virtual memory accesses without hardware support. Most virtual memory systems provide a flat memory space, and virtual addresses in the space are translated by the virtual memory hardware. REM computers, however, do not assume such hardware. Alternatively, if we use software interpretation to translate the addresses at run time, the execution speed of memory accesses becomes very slow.

To efficiently support a large virtual address space, the *t-kernel* defines three types of memory areas with different attributes, and differentiates memory accesses to these areas to make the common case fast.

- Heap memory: swappable and relocatable.
- Stack memory: relocatable but not swappable. For efficiency, this memory area is physically contiguous.
- Physical address sensitive memory (PASM): not swappable and not relocatable.

Consequently, we call the virtual memory in the *t-kernel* differentiated virtual memory (DVM). The specification of these memory areas in the naturalized virtual memory space

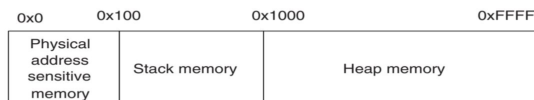


Figure 5. Example of virtual data memory configuration

is configurable for specific systems. Figure 5 shows an example of memory configuration of a 64KB virtual memory space.

4.3.2 Differentiated of memory accesses

The DVM treats memory accesses to different memory areas differently. Most of the memory accesses are accesses to local variables, temporary variables, parameters, and register saves and restores. All these happen in the stack memory area. Hence, the *t-kernel* does not swap data in the stack memory area out, and optimizes stack memory accesses to make them execute very fast. Some instructions (e.g., push, pop) execute at native speed. Accesses to the PASM area is also at native speed since the virtual and physical addresses are the same.

Accesses to the heap memory area can be quite slow. The *t-kernel* maintains a buffer of data frames in RAM. Each of the data frame corresponds to a page of minimum page size (16 Bytes) in the heap memory area, as well as a control block. The control block contains a tag identifying its starting address in the virtual memory, a frequency field, and some flags. When the *t-kernel* handles a heap access, it searches the data frames beginning from the data frame where the last heap memory access was performed, and swaps in data frames from the external flash if necessary. An optimization is included to map the heap area statically to the data frame area for small-heap programs. It is worth mentioning that the DVM uses no page or segment tables in RAM, thus minimizing memory overhead.

4.3.3 Kernel data integrity

As mentioned in Section 4.2, the integrity of kernel data is an important aspect of OS protection. Maintaining kernel data integrity involves a number of trade-offs and design choices. We briefly describe some key components here.

The *t-kernel* shares a common stack with the application, and maintains its state information in a kernel heap. Because the stack is shared, the *t-kernel* does not trust any data stored in the stack before the kernel transition happens. Instead, when a kernel transition happens, it establishes a new stack on top of the current stack for its execution. The kernel heap, on the contrary, is a memory area dedicated to kernel, and the naturalizer does not map any virtual memory address into this area. Hence, logically, both the kernel stack and kernel heap are isolated from the application memory space. Measures are taken to prevent accidental application errors from writing the kernel heap. For example, the naturalizer makes sure the growth of the stack cannot invade the kernel heap, and that the application cannot misuse the return addresses in the stack to jump outside the naturalized program.

4.3.4 Swapping

Swapping is a grand challenge because the limited erasure/write cycles of the external flash, and the scarcity of the physical memory. Hence, its design principles and swapping policy are different than those for traditional hard disks. Specifically, the design space of traditionally virtual memory systems has two major dimensions: space and speed, known as the slogan “as large as the hard disk, and as fast as RAM”. The *t-kernel* must consider another dimension: longevity—the same number of swap-outs, if directed to different pages, can either succeed or destroy the flash. As a result, the paging module in the *t-kernel* optimizes to reduce the number of swap-out’s, resulting in an unbalanced number of effective swap-in’s and swap-out’s. This policy slightly increases the swap-out overhead, but significantly extends the lifetime of the flash. Our technical report contains details of the swapping system in the *t-kernel* [16]. We highlight here that, different from flash file systems and flash drivers with wear-leveling, this design optimizes for swapping activities, and uses only 32 bytes in RAM.

4.4 Kernel/application interface

While an application can run on a host node without any knowledge about the kernel, there are circumstances where the application desires to interact more closely with the kernel. For example, the application need to call the kernel’s scheduler to perform preemptive priority scheduling. To facilitate such an interaction, the TinyOS uses function calls to make the OS and application communicate with each other. However, the *t-kernel* has separate kernel and application spaces, and the function calls cannot cross the boundary between them.

The *t-kernel* provides a communication interface between the kernel and the application to facilitate such interaction. This interface comprises system calls, event triggering, and interrupt handling. System calls are made by the application to the kernel to invoke system services. The *t-kernel* designates a set of special virtual program addresses as system call entry points, and the naturalizer modifies calls to these addresses to transitions into the kernel. After performing a system service, the kernel notifies the application via an event triggering interface implemented as a software interrupt. The kernel generates the software interrupt, and the application handles it, thus connecting the kernel and application logic. The *t-kernel* implements a shared memory mechanism, to support the data exchange between the kernel and application.

Some of the lowest level events are hardware interrupts. Hence, the same event triggering mechanism is used to handle interrupts, except that the event handler is not triggered by the software interrupt, but a hardware interrupt. The event handler for an interrupt can be at a low level, directly manipulating the hardware, or at a higher level, leaving part of the logic to be performed by the kernel. This design gives the application developers maximum flexibility for interrupt handling.

We use radio communication to illustrate the interaction of the kernel and the application. The kernel and the application collaborate to conduct radio communication. Such collaboration can be at a very low or relatively high level. At the very low level, the *t-kernel* provides a byte-level interface which allows the application to give an outgoing byte to and receive an incoming byte from the kernel. When a radio interrupt (an SPI interrupt on MICA2) occurs, the kernel only reads and writes the radio transceiver’s data register, and triggers the byte-level event (SPI interrupt handler on MICA2) in the application for the new incoming byte. The application conducts the major tasks, including byte pro-

cessing, transceiver controlling, framing, packet parsing, and traffic regulating, in its event (interrupt) handler. At the byte level, the application handles interrupts at about 2.5KHz on MICA2 motes. At this interrupt rate, the packet sending and receiving works reliably, though the radio throughput is slightly lower than that in native mode.

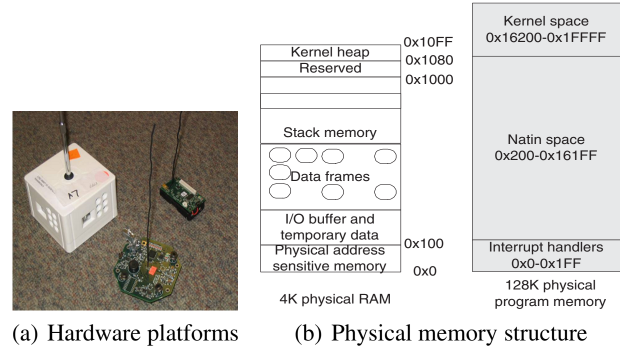
Note that the kernel itself supports radio communication in order to respond to wireless control requests. Hence, much of the byte-level processing code in the application is duplicate to the kernel's code. As an alternative, the application can elect to collaborate with the kernel at a higher level, using the frame-level service provided by the kernel. The application issues a system call to request the kernel to send a packet frame. The kernel transmits the packet, then triggers the application's packet level event handler. At this level, the application handles interrupts at the same frequency as the packet sending rate. The performance is comparable to that in native mode (refer to Section 6.4).

It is worth noting that the load-time code modification implies that the execution of the application has a warm-up stage. When new code pages are first loaded and executed, the naturalization process performs flash I/O to fetch instructions and write natin pages, resulting in a slow execution speed. This naturalization overhead is also involved when one type of interrupts fire for the first time. As a result, the handling of the first few interrupts can be very slow. For time-sensitive interrupts, such as radio interrupts, this warm-up delay means that the first few interrupts may not be handled in time. However, as more interrupts of the same type occur, and the interrupt handler becomes naturalized, the load-time overhead is not involved any more, and interrupt handling becomes very fast. As work in progress, support for real-time tasks has been partially implemented in the *t-kernel*, including the pre-naturalization of functions, and mechanisms to pin data frames and natin pages in the data and program memory. In the future, time-sensitive interrupts can use the real-time support to eliminate warm-up delay.

Though preemptive scheduling is not a focus of this paper, it is worth mentioning that scheduling is an important system service provided by the kernel, and has significant impact on the quality of signal processing and communication services. Branch regulating and DVM make the naturalized program use more CPU cycles than the original application, and therefore increase the CPU utilization. However, the increased CPU utilization does not necessarily affect the time accuracy of sampling and communication when preemptive priority scheduling is used. Without preemptive scheduling, long computation tasks would affect the time-sensitive tasks even when the overall CPU utilization is low.

5 Implementation

We have implemented the *t-kernel* for the ATmega128L microcontroller, which is broadly used in many WSN platforms [3, 10, 21]. We have tested *t-kernel* on MICA2 family motes, including MICA2, XSMv1, and ExScal motes, as shown in Figure 6(a). Table 1 lists specifications of the hardware and system parameters for the *t-kernel* on MICA2. We intend to release the *t-kernel* to the WSN research community, and are currently improving it for TinyOS compatibility. Both the kernel and the test data, including the kernel benchmark programs (refer to Section 6), will be made available.



(a) Hardware platforms (b) Physical memory structure

Figure 6. The *t-kernel* implementation

Hardware parameters	Data RAM	4KB
	External flash	512KB
	Program memory	128KB
OS parameters	Virtual memory	64KB
	Data frame	64 frames
	Lookaside buffer	64 entries
	2-associative VPC table	256 entries
	System stack	1KB
	I/O buffer	516 bytes
Source code	Total	15,312 lines
Binary	Total	28,864 bytes
	Naturalizer	14,000 bytes
	Dispatcher	5,020 bytes
	Paging module	6,172 bytes

Table 1. System characteristics

The implementation on MICA2 motes supports 64KB of virtual data memory (16 times the physical memory). The configuration of the virtual memory space is shown in Figure 5, and the organization of the physical memory is shown in Figure 6(b). Empirical study leads us to choose bits 2-7 of the VPC to index VPC lookaside buffer, and bit13-bit6 to index the 2-associative VPC table.

As listed in Table 1, the kernel code occupies 28KB of the 128KB program memory. The *t-kernel* reserves 1KB RAM for stack space. According to our study on one of the largest WSN applications [18], the stack usage seldom reaches 512 bytes. The *t-kernel* reserves 516 bytes for system I/O buffer, mainly used for flash I/O.

6 Performance evaluation

In this section, we present the performance results of *t-kernel* on the Berkeley MICA2 platform. We first measure the overhead of the kernel, including kernel transitions and the DVM. Then we evaluate the relative execution time of a set of kernel benchmark programs, and assess the slowdown of computational tasks. We verify our estimation by examining the execution speed of a TinyOS application under typical and stressed workloads. After that, we study the performance of radio communication, and analyze energy efficiency. We compare the *t-kernel*'s performance with an interpretation based virtual machine. Finally, we verify OS protection against a number of application errors.

When conducting the evaluation tests, we take measures to avoid variance introduced by different hardware and different environments. The former affects the calculation of CPU cycles, and the comparison between the *t-kernel* and the native mode; the latter may affect the quality of wireless communication. In this section, all execution times (absolute

Type of branch	Cost (cycle)
Forward branch, taken	5.0
Forward branch, not taken	4.0
Backward branch, taken	21.2
Backward branch, not taken	4.0
Relative jump, forward	3.0
Relative jump, backward	19.2

Table 2. Overhead of branches

Name	Numberof accesses	Native (sec.)	Cycle	*t-kernel* (sec.)	Cycle
mem.stack	8388608	5.63	2	17.24	7
mem.heap	8388608	N/A	N/A	28.14	16
mem.swap	1024	N/A	N/A	1.93	180857
mem.none	8388608	3.39	N/A	8.98	N/A

Table 3. Memory access performance

or relative) specified are averages of three or more runs. The deviation of the execution times in these runs are less than 1% of the execution time unless otherwise specified.

6.1 Overhead of kernel transitions

The kernel transitions and DVM are major sources of overhead impacting the execution speed of an application. We discuss the overhead of kernel transitions in this section. The overhead of DVM is discussed in Section 6.2.

We use several measurement programs written in assembly to study the overhead of kernel transitions. The results are reported in Table 2. The forward branches execute a fixed sequence of of natins. Hence, the overhead in cycles is deterministic. The overhead of backward taken branches varies (refer to Section 4.2.2), and is an average number taking into consideration the amortized cost of the sanity check routine. As the table shows, the backward taken branches have a relatively high overhead. However, forward branches execute for only 3–5 CPU cycles. Hence, we enhance the overall performance of branches by making the common case (forward branches) fast.

6.2 Overhead of naturalized virtual memory

First, we use a group of assembly programs to measure the speed of native and naturalized memory accesses. Specifically, *iter.mem* accesses memory in the stack area using register indirect addressing with offset, which is the slowest stack area memory access in the *t-kernel*. *iter.heap* accesses the heap memory area without incurring swapping. *iter.swap* sweeps 16KB of the virtual memory space (from 0x2000 to 0x6000) with steps of 16 bytes to ensure there is a page fault for each memory access and a swap-out for some of them. To isolate the execution time of non-memory instructions and quantitatively assess the execution time of various memory accesses, we also measure the execution time of a program *iter.nomem* which has the same program structure, but no memory access instructions. Table 3 summarizes the evaluation results.

We notice that the heap access time has a large variance. When there is swapping, a heap access can take 180,857 cycles (25.80ms). According to our test, it takes the *t-kernel* 25.73ms, on average, to perform an erase/write operation on the external flash. Hence, the swap-out time is dominated by the I/O latency. This is a similar situation to traditional virtual memory where a disk I/O operation costs hundreds of thousands of CPU cycles. In the meantime, memory accesses to the stack memory area are very fast. Again, we optimize the overall performance by making the common case (stack access) fast.

Name	Function	Application
am	Active messaging	Network protocol
amplitude	Signal processing	Sensing
eventchain	Event dispatch	All TinyOS apps.
timer	Timer event dispatch	Periodic tasks
readadc	Read analog sensor	Sensing applications
crc	CRC calculation	Network protocol
lfsr	Random number	Various applications

Table 4. Kernel benchmark programs

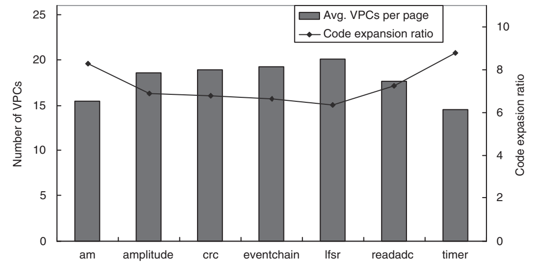


Figure 7. Code expansion of kernel benchmark programs

6.3 Assess application-level performance

The measurement programs can accurately measure the overhead, but they do not represent typical instruction mix in WSN applications. To the authors' best knowledge, benchmark research for WSNs is still in an early stage [20]. Without an existing standard benchmark set, we compile a group of kernel benchmark programs representing typical activities in a WSN, as listed in Table 4. All these programs' core part are extracted from existing WSN applications that have been deployed and tested. For comparison, all these programs are written in TinyOS' programming language – nesC [14], and are compiled with the same compiler settings [1]. We evaluate the kernel benchmark programs on code expansion ratio, naturalization cost, and relative execution time (the ratio of the execution time on the *t-kernel* to that in native mode).

Naturalization expands the code size because of branch regulating, DVM, and the cascading branch chain. Figure 7 shows the average number of VPCs per natin page, and the corresponding code expansion ratio. Both the code expansion and the variance of the expansion ratios are a known phenomena in binary translation systems. For example, the IBM DAISY system expands code size 1–21 times [11]. But the *t-kernel* performs a more complex task of providing an enhanced system abstraction.

In the naturalization process, the kernel involves overhead for reading and parsing application instructions, writing natin pages, and patching destination HPCs when performing bridging operations. Dominating these naturalization overhead is time for the flash I/O, which is directly related to the number of natin page writes. Interestingly, we find that naturalization cost for the kernel benchmark programs varies from 22 to 51 natin page writes per kilobyte of application code. This approximately translates to 590ms–1380ms of naturalization time for one kilobyte of applica-

[1] The build process only changes several linker parameters to relocate the ".data" and ".bss" memory sections.

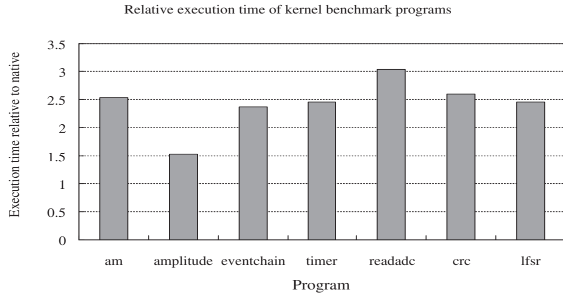


Figure 8. Performance of kernel benchmark programs

tion code. Naturalization is a complex process, and we are still in the process of investigating the cause of such a large variance.

Figure 8 compares the relative execution time of the kernel benchmark programs. The execution time on the *t-kernel* includes the time for naturalization. Different programs' performance differ noticeably, due to the different branch density and frequency of heap accesses in the programs. The relative execution times range from to 1.53 (*amplitude*) to 3.03 (*readadc*). An interesting observation is a fairly high relative execution time (2.6) for *crc* even though it has no heap accesses. It is due to a combination of "sbrs" (skip if bit in a register is set) and "rjmp" (relative jump) instructions close to the end of a natin page, which makes efficient handling impossible. Showing an example of application code that is inefficient for the *t-kernel*, it also suggests an opportunity for code optimization if we have a *t-kernel* compiler.

Overall, we expect CPU-bound computation to have a relative execution time of 1.5–3 on *t-kernel*, corresponding to a slowdown of 0.5–2 times. This is a noticeable instruction-level overhead in both execution time and energy consumption. However, most of current sensor network applications are not CPU-bound. Hence, the users of an application do not necessarily observe a slow-down at the application level. Energy-wise, upgrading hardware to provide the same functionality as the *t-kernel* also increases the power consumption. Depending on the hardware technology and the embedded application, the *t-kernel* may prove to be more energy efficient in some areas in the design space.

With energy efficiency studied in Section 6.5, we focus on execution speed in this section. To determine whether the computation slowdown is acceptable, we examine the CPU utilization, denoted as μ, of WSN applications. For this purpose, we cannot use the kernel benchmark programs because, as computational tasks, they are designed to have $\mu = 1$. We use Avrora to profile the same applications as those in Figure 3. The result is shown in Figure 9. We find that, without exception, the applications spend more than 92% of CPU time in idle mode. This result is not surprising— many WSN applications are intrinsically I/O bound. Obviously, with $\mu < 0.1$, using 1.5–3 times CPU cycles does not noticeably degrade the applications' performance. The execution on the *t-kernel* should not show any slow down except for naturalization cost.

Generally, for mote-class sensor nodes running typical applications, memory and energy, not CPU cycles, are scarce resources and often the bottleneck of system performance.

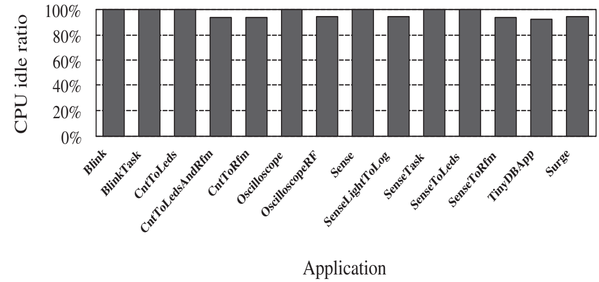


Figure 9. CPU idle ratio of TinyOS applications

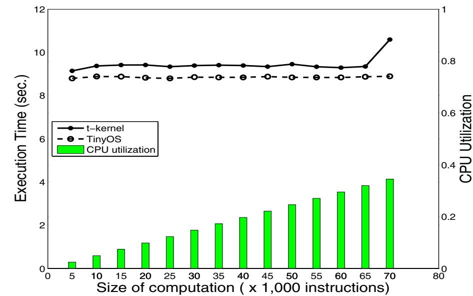


Figure 10. Execution Time and CPU utilization under different workloads

Consequently, a mechanism that moderately slows down the speed, but provides better service, is useful. Based on this observation and the performance data, we believe *t-kernel*'s overhead is acceptable.

To verify this claim, we test *t-kernel* with a sensor network application, PeriodicTask, which posts and executes computation tasks when timer events occur. Most of the sensor network applications periodically wake up, poll sensors, and communicate with other nodes (transmit or listen). Hence, the PeriodicTask application captures the typical work mode of WSNs. Note that existing TinyOS applications, like those listed in Figure 9, have very small μ. Trivially, these applications run as smooth on the *t-kernel* as on TinyOS. To probe the limit of *t-kernel*, we vary the amount of computation in each task to examine typical and stressed workloads.

In this experiment, we set the timer to fire every 30ms, and configure the computation tasks to contain 5,000 to 70,000 instructions, corresponding to $\mu = 0.02, 0.04, \ldots, 0.34$. Figure 10 shows the execution time of 300 computation tasks of different computation sizes on TinyOS and on the *t-kernel*. For all the computation sizes, the deviation of execution time is less than 5%.

When $\mu < 0.3$, we observe that the execution time on the *t-kernel* is slightly longer because of naturalization cost. Except for that, the execution of PeriodicTask on the *t-kernel* does not show noticeable difference from that on TinyOS. When $\mu > 0.3$, the PeriodicTask still runs well functionally, but experiences a degraded performance in execution time.

This experiment reveals characteristics of the *t-kernel*'s region of operation. With a relatively large μ (e.g., $\mu > 0.3$), the overhead of the *t-kernel* is exhibited at the application level, and users observe operations of a WSN application slow down. The extreme case ($\mu = 1$) is shown in the eval-

uation of the kernel benchmark programs, which run 1.5–3 times as long as in native mode. Whether or not the application still works depends on the application's property. Applications requiring functional correctness (e.g., computing a large prime) may tolerate the slow down. Applications with time requirements fail, or experience performance degradation. Generally, we do not consider the area with $\mu > 0.3$ as a recommended region of operation for the *t-kernel*.

On the other hand, *t-kernel* is suitable for applications with a relatively light workload (e.g., $\mu < 0.1$). In this region, applications run on the *t-kernel* without experiencing noticeable performance degradation at the application level. As shown in 9, light workload is the common case for WSN applications. Hence, we believe the region of operation of the *t-kernel* covers a wide range of WSN applications.

Note that the CPU utilization is not the only factor deciding whether an application works on the *t-kernel*. The variance of the CPU utilization may stress a "stage" of the computation, and, when the slowdown in that stage breaks the application's timing assumptions, the application may fail. Section 8 discusses timing assumptions in further detail.

The benefits of the *t-kernel*, on the other hand, is obvious. Now the WSN developers can take advantage of the powerful features—reliable OS protection, expanded virtual memory space, and preemptive task scheduling—on very-low-power sensor nodes. In the experience of developing and deploying WSN systems in the past few years, we observe that these features are so desirable that the benefits by far outweigh the cost in most situations.

6.4 Radio communication

In this section, we evaluate the performance of radio communication on the *t-kernel* by studying one-hop communication, which is the foundation of networking, and involves a number of components in the system. We compare the link-layer performance of the *t-kernel* to that of TinyOS. For this purpose, we write a communication application in nesC to broadcast wireless packets at certain intervals. On the *t-kernel*, the application invokes the frame-level service to transmit packets. We use a "sender" mote to send packets, and a base station to receive packets. In order to test the throughput at high sending rates, the application disables the listening and backoff mechanisms at MAC layer. The packet reception ratio is calculated against all packets supposed to be sent.

We use three settings with varying intervals—50ms, 500ms, and 5,000ms—to represent different traffic loads as measured by packet rate. For each setting, we study the average packet reception ratio of three or more runs. The deviation of the packet reception ratio for each setting is less than 5%. The shortest interval is 50ms, representing a scenario of high packet rate (20pkt/sec). With this setting, both the *t-kernel* and TinyOS deliver most of the packets successfully—99.8% in the native mode and 93.9% with the *t-kernel*. Based on our observation, at least one packet is lost due to the warm-up delay, when the code to handle radio interrupts and the packet framing is being naturalized and experiences a warm-up delay. But we have not quantitatively analyzed how many packets are lost in the warm-up stage. The intervals 500ms and 5000ms represent moderate packet rates—2pkt/sec and 0.2pkt/sec, respectively. With these settings, both the *t-kernel* and TinyOS deliver more than 95% of the packets.

The performance data shows that the *t-kernel* performs nearly as efficiently as native code (TinyOS) at a moderate packet rate. Based on our observation of existing WSN applications, a moderate packet rate is a common case in WSNs.

6.5 Energy efficiency

On current very-low-power sensor nodes, the *t-kernel* transforms the simple hardware into a platform that supports OS protection, virtual memory, and preemptive scheduling. An interesting question is whether this technology will still be useful when the hardware technology improves in the future. In Section 2, we listed energy budget, form factor, and cost as limiting factors against upgrading hardware of very-low-power computers. In this section, we study the energy efficiency in further detail.

To enhance the system abstraction, we can either use a software approach, such as the *t-kernel*, or upgrade the hardware (e.g., increase the RAM size). Both approaches increase the power consumption compared to the basic hardware configuration. However, they reflect different trade-offs and are more efficient than the other approach in different areas of the design space. We introduce the major variables in the energy consumption of a WSN system, then study several design points in the space to show where the *t-kernel* performs better than the alternative approach with enhanced hardware and where it does not. Following that, we list some observations which make us believe that the *t-kernel* will prove a flexible and scalable solution for some application systems in the future.

The power consumption of a sensor node, denoted as P, depends on a number of variables. To save energy, the sensor nodes in many WSN systems enter a low-power sleep mode, until they are awakened by timer interrupts or external events [15, 18, 19]. When awakened, the sensor node may be computing (active mode) or waiting (idle mode). The idle mode typically has lower power consumption than the active mode. The sleep mode leakage power is the ultimate limit on the lifetime of a sensor node. We denote the average active-mode current as I_{active}, the idle-mode current as I_{idle}, and the average sleep-mode current as I_{sleep}. Typically, I_{active} and I_{idle} are much larger than I_{sleep}. We call the fraction of the time spent in non-sleep modes the duty cycle, denoted as f_{duty}, and call the ratio between the time in active mode and the time in non-sleep mode the CPU utilization, denoted as $f_{utilization}$. When a system runs the *t-kernel*, the system performs extra flash writes. We denote the average energy consumed by one flash page I/O to be E_f. Flash I/O only happens in non-sleeping mode. We denote the average number of flash page writes (swap-outs) per second in non-sleep mode as N_f. Assuming the battery voltage is v, we approximate the average power consumption with the following formula

$$P = v(f_{duty} \cdot (f_{utilization} \cdot I_{active} + (1 - f_{utilization}) \cdot I_{idle}) + (1 - f_{duty}) \cdot I_{sleep}) + N_f \cdot E_f$$

This approximation does not consider some factors of secondary importance. Both the active-mode and sleep-mode currents have variations depending on the activities (e.g., whether the radio is on or off), ambient temperature, and the

battery status. We assume that such variations are captured by the averaging process. For example, if the radio transmission increases the active current by 10mA for 10ms in every second, the average active-mode current increases by 0.1mA. The energy for flash I/O depends on the erasure block size and the writing method. The variable E_f represents the average energy cost.

With the formula for the average power consumption, we examine several points in the design space. The system energy consumption depends on the hardware technology, the fabrication process, the power management scheme, and the workload. The numbers used below represent some reasonable design points, but a different platform can certainly have different numbers. Many of the numbers are based on experimental results on MICA2 and ExScal sensor nodes and the VigilNet application. A surveillance network similar to the VigilNet may constitute a design point where $v = 3V$, $I_{active} = 15mA$, $I_{idle} = 10mA$, $I_{sleep} = 100\mu A$, $f_{duty} = 0.01$, $f_{utilization} = 0.05$, $N_f = 0$, and $E_f = 1.2mJ$. The average power consumption is 0.6mW.

We can enhance the complexity of the microcontroller to support privileged instructions and expand the RAM size to 64KB, in order to provide the enhanced system abstraction. The additional hardware logic, particularly, the expanded RAM component, slightly increases I_{active} and I_{idle}, and significantly increases I_{sleep}. As another design point, suppose $I_{active} = 16$mA, $I_{idle} = 11$mA, and $I_{sleep} = 120\mu A$. The average power consumption becomes 0.69mW, 14.8% higher than the MICA2 with 4K RAM.

In contrast, the *t-kernel* provides the enhanced system abstraction without increasing I_{active},I_{idle}, or I_{sleep}. However, it increases the CPU utilization, and introduces additional flash I/O. Suppose that the application runs 3 times as long on the *t-kernel* as in native mode, and $N_f = 1$. We have $P = 0.63mW$, which consumes 4.5% more energy than the basic MICA2, but 9% less than the upgraded hardware with 64K RAM. Hence, at this design point, the *t-kernel* is more energy efficient.

On the other hand, the *t-kernel* can be less energy efficient than enhanced hardware in some areas in the design space. For example, when the application's memory accesses have low locality, the DVM thrashes, and the application slows down further. Thrashing dramatically increases the energy consumption. Suppose there are 10 swap-outs per second when the sensor node is in non-sleep mode. The average power consumption becomes 0.74mW, which is 6.6% higher than the 64KB-RAM sensor node.

Generally, upgrading hardware increases the all-time power consumption of a sensor node, while the *t-kernel* increases the active mode power consumption, but keeps the idle and sleep mode power consumption at the same level as the basic hardware. Several observations, listed below, make us believe the *t-kernel*'s approach will prove to be a flexible and scalable solution for some applications.

- Lowering the duty cycle is an important technique to extend system lifetime. Many systems seek to let sensor nodes spend more and more time in sleep mode.
- With the same technology, the leakage power of SRAM increases with its size. Unless the system can turn off part of the memory structure and discard the data stored in it, the leakage current consumes energy throughout the lifetime of a sensor node.
- The activities in many WSN systems are highly repetitive after the initialization stage. We use Avrora [33] to collect dynamic traces of the Blink, Sense, and Surge applications for 2.2 billion cycles. They represent WSN programs with increasing complexity. We analyze the unique addresses (VPCs) accessed per 100 million cycles. As shown in Figure 11, the sizes of the unique VPCs decrease after the initial stage, and stay fairly stable after that. The repetitive activities performed by a stable set of code result in a stable working set and should keep the swapping rate low.
- Flash I/O is an energy-hungry operation on MICA2 motes. However, if we examine other available products and the projected future development, the flash I/O is very likely to become a relatively "lightweight" operation in energy consumption on a sensor node, compared to communication [29]. Computation is also "lightweight" in energy consumption.

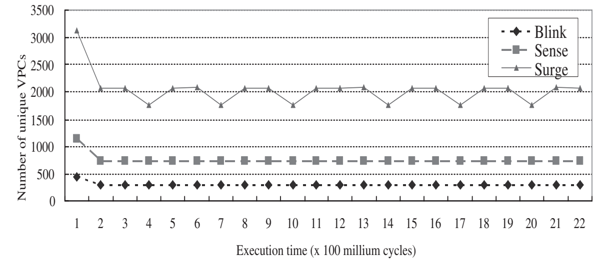


Figure 11. Number of unique VPCs executed in three applications

Based on these observations, we believe that the *t-kernel* will continue to be a preferred approach for some applications. By keeping the hardware simple, the *t-kernel* gives future system designers the flexibility of applying low-power hardware designs to WSN systems. By limiting the overhead to be within the active-mode time, it improves the scalability of energy-efficiency as the systems lower the duty cycle and extend the system lifetime. By taking advantage of the repetitive operations in WSN applications, it keeps the software overhead within a reasonable range. In the future, an optimizing compiler for the *t-kernel* can further reduce the overhead, and expand the area in the design space where the *t-kernel* outperforms the upgraded hardware in energy efficiency.

Note that manual memory management can be more efficient than virtual memory in both speed and energy consumption, when the physical memory can hold the working set, and the memory access pattern is known. For example, the VigilNet system uses a memory overlay scheme to reuse memory areas, taking advantage of the knowledge on the state transitions of the application. However, manual memory management has a high programming cost, coarse granularity, and strong application dependence. Automating the memory swapping used in manual overlay schemes, virtual memory is a more general solution.

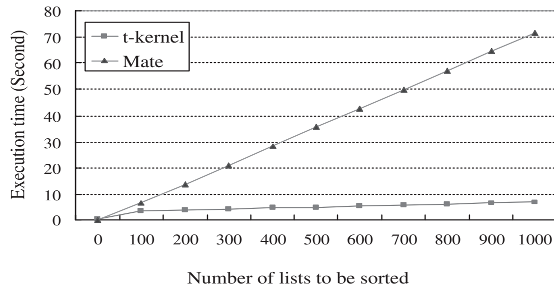


Figure 12. Execution time of insertion sort

6.6 Comparison to the virtual machine approach

Virtual machines can also provide enhanced abstraction not directly supported by hardware. Using an interpretation based virtual machine is a much simpler way to implement OS protection and virtual memory. The reason that the *t-kernel* does not elect to use an interpretation based approach is for extensibility and performance.

Maté is a virtual machine on TinyOS, and interprets a stack based bytecode language [25]. Implemented on several mote platforms including MICA2, Maté is possible to be extended to support "functionality equivalent of the benefits a virtual memory system brings" Hence, it is instructive to compare it with the *t-kernel*. It is worth noting that this comparison is limited and is meant to stress some point to illustrate the difference between code modification and interpretation. In fact, Maté can be re-constructed to be an application-specific VM by incorporating custom functions [25]. The functions can implement complex algorithms inside the VM and execute them at native speed, then Maté would significantly outperform the *t-kernel*, just like what we have shown with the kernel benchmark programs running in native mode. However, the custom functions essentially introduce application code into the OS, and make the system reliability dependent on the quality of those functions. Because this is a situation the *t-kernel* aims to avoid, we compare the *t-kernel* to the basic Maté.

We study Maté's performance with an insertion-sorting program, which is a common choice for sorting small-data sets, and is used in WSNs (e.g., VigilNet). Figure 12 plots the performance of insertion sort in the *t-kernel* and in basic Maté. Note that, for a customized Maté with the sorting operation implemented as an opcode, Maté would run close to native speed and be several times faster than the *t-kernel*. Hence, this comparison uses Maté in a suboptimal mode. The purpose is to analyze the pros and cons of the modification-based and interpretation based approaches.

The data shows that, for 0 list, Maté runs faster than the *t-kernel*. With 100 lists, the *t-kernel* has a moderate increase in execution time when the number of lists increases from 0 to 100. It is caused by the load-time naturalization overhead when new application code is used. After that, the execution time increases very slowly. On the other hand, Maté shows a consistently larger increase of execution time. When the number of lists increases, the initial naturalization cost is amortized by the repeated use of natin pages, and the modification-based approach becomes one order of magnitude faster than the virtual machine approach.

The *t-kernel* supports the hardware's instruction set, and code modification has high load-time overhead for naturalization, but very low run-time overhead. Hence, it is convenient to implement new hardware drivers, algorithms, and network protocols with the microcontroller's instruction set, and execute them on the *t-kernel*. In contrast, Maté supports a bytecode language, and interprets instructions at run time. Due to limited resource, it is hard to perform sophisticated virtual machine optimization on MICA2 motes. Hence, it has almost no load-time overhead, but a fair amount of run-time overhead. Meanwhile, high-level bytecode instructions, such as sending a packet, is very efficient in Maté because they can usually be mapped to a few operations each of which corresponds to a function in native instructions. In fact, the *t-kernel* and virtual machines work at different levels and can work together to benefit from each other's strengths.

6.7 Resistance to application faults

Without OS protection, when a serious program fault engenders an application error, a sensor node may crash, loosely defined as a situation when the sensor node is not able to respond to system commands any more. The *t-kernel* is resistant to application faults. This means that the *t-kernel* always control the sensor node despite application errors, i.e., application errors do not compromise the OS. On the other hand, the errors may compromise the application itself, and make the application exhibit wrong behavior.

We have tested a number of "wild" programs with left recursion, pointer errors, bad return addresses, bad branch destination, invalid instructions, bad array index, and infinite loop. With memory errors (e.g., left recursion, bad array index), the sensor node may crash when critical data is overwritten. However, the *t-kernel* handles the wrong memory addresses as if they were valid memory accesses. For control errors (e.g., bad return address, infinite loop) the kernel checks the control flow to make sure it directs to an entry point in a natin page, and the natin pages invoke kernel services frequently (branch regulating). Hence, when application error happens, the OS' operation is not affected and critical system services, such as wireless reprogramming, are still operative if they are implemented in kernel space.

7 Related work

The work on the *t-kernel* is related to, but distinct from, many research areas including embedded OS, virtual machines, network program distribution, binary translation, and software fault isolation.

Besides TinyOS [21], a number of OS's have been developed for WSNs and networked low-power systems. MANTIS and Contiki are two recent projects providing multi-thread support on MICA2 motes [4, 9]. Not supporting virtual memory, both systems consider RAM a highly limited resource, and neither guarantees OS protection. Recently, Han et al. developed SOS, an OS that supports dynamically loaded modules [17]. The current implementation of SOS does not support OS protection.

Labrosse et al. designed μC/OS and it has evolved for years and reached the maturity of an industry-quality real-time multitasking embedded OS [24]. In the smallest configuration, it can be reduced to 2KB code and 200 bytes data, not including the stack space. However, when components providing advanced functionalities are included, the

code and data sizes increase significantly. In a typical configuration allowing 10 tasks, the data size becomes 3K, not including the stack. Hence, ATmega128 class microprocessors cannot actually leverage all the benefits provided by such an RTOS after "porting". This is a common problem with module based "configurable" OS's—a decent set of features are only practically available when the platform has enough resources to accommodate all the corresponding modules. WSN applications require rich features, but have scarce resources, therefore rendering such solutions less useful.

We have studied the virtual machine approach with Maté (Bombilla) [25] in Section 6.6. Other virtual machines include MagnetOS [2], whose "single system image" unifies the whole network as one distributed machine, and SensorWare [5], which disseminates "script" code over the network. Kwon et al. developed a mobile agent platform called ActorNet [23]. ActorNet provides a virtual execution environment including virtual memory. As explained in Section 6.6, these virtual machines are different than the *t-kernel*. For example, the ActorNet uses a Scheme-like language, and the virtual memory does not distinguish memory areas.

Reijers et al.[32] designed a scheme for code distribution on the EYES platform. Unlike virtual machines, Reijers' network-distributed code is not interpreted, but "patched" into the program memory. The patching re-constructs a new application program, but does not provide enhanced system abstraction for the new program.

Binary translation systems have been used for a number of purposes [1, 7]. The code modification process in the *t-kernel* is one special type of binary translation that performs page-based, single-instruction modification to provide an enhanced system abstraction. However, due to the very different design context, most of the algorithms in traditional binary translation systems cannot be used in the *t-kernel*.

Providing a "safe" execution environment, the approach closest to the *t-kernel* is sandboxing in software fault isolation [31, 34]. They both modify part of untrusted code at program load time. However, these two methods are significantly distinct in many ways. For example, sandboxing's major target is to regulate memory accesses. In contrast, the *t-kernel* has a much more ambitious goal of providing enhanced abstraction that hardware does not support. Sandboxing relies on virtual memory hardware, but does not provide functionalities beyond the hardware's abstraction, while the *t-kernel* establishes a virtual memory mechanism on hardware platforms that do not support it.

8 Limitations and future work

The *t-kernel* is designed to target applications where the energy budget is tight, the CPU utilization is low, and the application requirements are relatively high. It is not suitable for all WSN systems. In this section, we list some limitations of the *t-kernel*, and discuss future work.

One important reason for the constraints on REM computers is their very low power consumption. We notice that the development of the energy technology is, in general, much slower than the computing technology. However, if technological breakthroughs make it possible to supply a large amount of energy with a small form-factor battery set, much of the resource constraints can be removed, and much of the *t-kernel*'s overhead becomes unnecessary. A similar argument is true when the energy scavenging technology can supply sufficient power to the sensor nodes.

Development of low power memory, including SRAM, DRAM, FRAM (Ferroelectric RAM), and flash, is another direction of technology progress that can change the tradeoffs in the very-low-power design context. Though we have not seen DRAM being widely used in mote-class devices, low-power SRAM chips have been on the market. If technology breakthroughs make large RAM structures much cheaper, smaller, and lower in power consumption, the benefit of the *t-kernel* may become less significant. On the other hand, with the development of FRAM and low-power flash, the cost of *t-kernel* should also keep decreasing. Moreover, the emergence of MRAM (Magnetic RAM), a nonvolatile, low-power, and high-speed RAM, may replace both the SRAM and flash on current sensor nodes. Overall, the memory technology certainly will have an impact on the design context of the *t-kernel*, and will change the tradeoffs and the balance in the design.

The code modification process and virtual memory swapping introduce unpredictable latencies to the instruction execution time. Even when the CPU utilization is low, this may be a problem when the application programmers make implicit timing assumptions. For example, if the programmer writes a loop to wait 1 microsecond for the hardware, the loop may execute for 2.5 microseconds on the *t-kernel*. We have tested a number of TinyOS applications on the *t-kernel*. The complexity of the applications range from the simplest "Blink" application to multi-hop flood routing applications. The routing application utilizes multiple, nested interrupts (SPI, ADC, clock, etc.). We found some applications (e.g., CntToRfm) failed when we set the timer to fire at a relative high rate (e.g., 50Hz). The reason is that some low-level modules have implicit timing assumptions. After modifying such low-level modules, all these applications work correctly. As mentioned in Section 4.4, the *t-kernel* design has preparation for real-time support. Combined with a real-time task specification mechanism to be developed in the future, the kernel real-time support will eliminate the problem of implicit timing assumptions.

Similar to traditional virtual memory systems, thrashing significantly reduces system performance. Besides execution time, energy efficiency is a concern in WSN systems. In Section 6.5, we have discussed one example in which the swapping makes *t-kernel* consume more energy than the upgraded hardware approach. Generally, when a program's working set is in memory, there are very few swaps. An inspection on the application code indicates that the working set of Blink, Sense, and Surge corresponds to 21%—40% of the program. If the working set cannot reside in the physical memory, the designer must be very prudent to use the *t-kernel* because thrashing may happen. Similarly, if the frequently accessed data items cannot be held by the data frames in RAM, the *t-kernel* must perform frequent swapping, and the energy efficiency may be worse than the upgraded hardware approach. Because virtual memory is still a new concept in WSN systems, we cannot find and study "typical" programs using virtual memory and analyze the locality and the working set for data. This is one limitation of this work. As a future work, we plan to adapt the VigilNet application to use virtual memory, and study its behavior [18].

Other directions of future work include real-time specifications and an optimizing compiler for the *t-kernel*. As mentioned before, they can further enhance the *t-kernel*'s performance.

9 Acknowledgments

This work is supported in part by NSF grant CCR-0098269 and CCR-0325197, the MURI award N00014-01-1-0576 from ONR, and the DARPA IXO offices under the NEST project (grant number F336615-01-C-1905). Special thanks to Sang H. Son, John Heidemann, Yingmin Li, and Ronghua Zhang for their help in this work.

10 References

[1] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.

[2] R. Barr, J. Bicket, D. Dantas, B. Du, T. Kim, B. Zhou, and E. Sirer. On the need for system-level support for ad hoc and sensor networks. In *ACM Operating Systems Review*, volume 36, pages 1–5, Apr. 2002.

[3] J. Beutel, O. Kasten, F. Mattern, K. R02mer, F. Siegemund, and L. Thiele. Prototyping wireless sensor networks with BTnodes. In *Proc. of 1st European Workshop on Wireless Sensor Networks (EWSN 2004)*, pages 323–338, Berlin, Germany, Jan. 2004.

[4] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han. MANTIS OS: An embedded multithreaded operating system for wireless micro sensor platforms. *ACM/Kluwer Mobile Networks and Applications (MONET), Special Issue on Wireless Sensor Networks*, 10(4):563–579, Aug. 2005.

[5] A. Boulis, C. Han, and M. Srivastava. Design and implementation of a framework for programmable and efficient sensor networks. In *Proc. of Intl. Conf. on Mobile Systems, Applications, and Services (MobiSys)*, pages 187–200, San Francisco, CA, May 2003.

[6] L. T. Clark, E. J. Hoffman, J. Miller, M. Biyani, Y. Liao, S. Strazdus, M. Morrow, K. E. Velarde, and M. A. Yarch. An embedded 32-b microprocessor core for low-power and high-performance applications. *IEEE Journal of Solid-State Circuits*, 36(11):1599–1608, Nov. 2001.

[7] J. Dehnert, B. Grant, J. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The Transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proc. of the Intl. Symp. on Code Generation and Optimization*, pages 15–24, San Francisco, CA, 2003.

[8] L. Doherty, B. A. Warneke, B. Boser, and K. S. J. Pister. Energy and performance considerations for smart dust. In *Intl. Journal of Parallel and Distributed Sensor Networks*, pages 121–133, Dec. 2001.

[9] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proc. of the 29th Annual IEEE Conf. on Local Computer Networks*, pages 455–462, Tampa, FL, Nov. 2004.

[10] P. Dutta, M. Grimmer, A. Arora, S. Bibyk, and D. Culler. Design of a wireless sensor network platform for detecting rare, random, and ephemeral events. In *Proc. of the 4th Intl. Conf. on Information Processing in Sensor Networks (IPSN'05)*, pages 497–502, Los Angeles, CA, 2005.

[11] K. Ebcioglu, E. Altman, M. Gschwind, and S. Sathaye. Dynamic binary translation and optimization. *IEEE Transactions on Computers*, 50(6):529–548, June 2001.

[12] V. Ekanayake, C. K. IV, and R. Manohar. An ultra-low-power processor for sensor networks. *Proc. of the 11th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 27–36, Oct. 2004.

[13] D. Estrin, R. Govindan, J. S. Heidemann, and S. Kumar. Next century challenges: Scalable coordination in sensor networks. In *Proc. of the 5th ACM/IEEE Conf. on Mobile Computing and Networking*, pages 263–270, Seattle, WA, 1999.

[14] D. Gay, P. Levis, R. Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. of Programming Language Design and Implementation*, pages 1–11, San Diego, CA, June 2003.

[15] L. Gu and J.A.Stankovic. Radio-triggered wake-up for wireless sensor networks. *Real-Time Systems 29(2-3)*, pages 157 – 182, Mar. 2005.

[16] L. Gu and J. A. Stankovic. *t-kernel*: A translative OS kernel for sensor networks. In *UVA CS Tech. Report CS-2005-09*, 2005.

[17] C. Han, R. Kumar, R. Shea, E. Kohler, and M. B. Srivastava. A dynamic operating system for sensor nodes. In *Proc. of the 3rd Intl. Conf. on Mobile systems, Applications, and Services*, pages 163–176, Seattle, Washington, June 2005.

[18] T. He, S. Krishnamurthy, J. A. Stankovic, T. Abdelzaher, L. Luo, R. Stoleru, T. Yan, L. Gu, G. Zhou, J. Hui, and B. Krogh. VigilNet: An integrated sensor network system for energy-efficient surveillance. *ACM Trans on Sensor Networks*, 2(1):1–38, 2006.

[19] J. Heidemann and W. Ye. Energy conservation in sensor networks at the link and network layers. *USC/ISI Tech. Report ISI-TR-2004-599*, 2004.

[20] M. Hempstead, D. Brooks, and M. Welsh. TinyBench: The case for a standardized benchmark suite for tinyos based wireless sensor network devices (poster). In *Proc. of the 29th Annual IEEE Conference on Local Computer Networks*, pages 585–586, Tampa, FL, Nov. 2004.

[21] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for network sensors. In *Proc. of the 9th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 93–104, Cambridge, MA, Nov. 2000.

[22] W. Hu, V. N. Tran, N. Bulusu, C.-T. Chou, S. Jha, and A. Taylor. The design and evaluation of a hybrid sensor network for cane-toad monitoring. In *Proc. of the 4th Information Processing in Sensor Networks (IPSN 2005)*, pages 503–508, Los Angeles, CA, Apr. 2005.

[23] Y. Kwon, S. Sundresh, K. Mechitov, and G. Agha. ActorNet: An actor platform for wireless sensor networks. In *Proc. of the 5th Intl. Joint Conf. on Autonomous Agents and Multiagent Systems (AAMAS)*, Hakodate, Japan, May 2006.

[24] J. Labrosse. MicroC/OS-II, the real-time kernel, 2nd edition. *ISBN 1-57820-103-9*.

[25] P. Levis, D. Gay, and D. Culler. Active sensor networks. In *Proc. of the 2nd USENIX/ACM Symp. on Network Systems Design and Implementation (NSDI)*, Boston, MA, May 2005.

[26] J. Lifton, M. Broxton, and J. A. Paradiso. Experiences and directions in pushpin computing. In *Proc. of IEEE/ACM Conf. on Information Processing in Sensor Networks (IPSN/SPOTS'05)*, pages 416–421, Los Angeles, CA, Apr. 2005.

[27] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. In *ACM Intl. Workshop on Wireless Sensor Networks and Applications*, pages 88–97, Atlanta, GA, Sept. 2002.

[28] C. B. Margi, V. Petkov, K. Obraczka, and R. Manduchi. Characterizing energy consumption in a visual sensor network testbed. In *Proc. of the 2nd Int. IEEE/Create-Net Conf. on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom 2006)*, March 2006.

[29] G. Mathur, P. Desnoyers, D. Ganesan, and P. Shenoy. Ultra-low power data storage for sensor networks. In *Proc. of IEEE/ACM Conf. on Information Processing in Sensor Networks (IPSN/SPOTS'06)*, pages 374–381, Nashville, TN, Apr. 2006.

[30] L. Nachman, R. Kling, R. Adler, J. Huang, and V. Hummel. The Intel Mote platform: a bluetooth-based sensor network for industrial monitoring. In *Proc. of the 4th Intl. Symp. on Information Processing in Sensor Networks (IPSN'05)*, pages 437–442, Los Angeles, California, 2005.

[31] P. Patel, A. Whitaker, D. Wetherall, J. Lepreau, and T. Stack. Upgrading transport protocol using untrusted mobile code. In *Proc. of the 19th ACM Symp. on Operating Systems Principles*, pages 1–14, Bolton Landing, NY, Oct. 2003.

[32] N. Reijers and K. Langendoen. Efficient code distribution in wireless sensor networks. In *Proc. the 2nd ACM Intl. Conf. on Wireless Sensor Networks and Applications*, pages 60–67, San Diego, CA, 2003.

[33] B. Titzer, D. Lee, and J. Palsberg. Avrora: Scalable sensor network simulation with precise timing. In *Proc. of the 4th Intl. Conf. on Information Processing in Sensor Networks (IPSN'05)*, pages 477–482, Los Angeles, CA, 2005.

[34] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proc. of the 14th ACM Symp. on Operating Systems Principles*, pages 203–216, Asheville, NC, Dec. 1993.