t-kernel: Provide Reliable OS Support for 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 networks (WSNs) is very difficult because of their stringent resource constraints, harsh energy budget, and demanding application requirements. We identify that three OS features – OS protection, virtual memory, and preemptive scheduling – will significantly improve the reliability of WSN systems and facilitate developing complex WSN software. However, due to the limitation of hardware, 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 load-time code modification and enhance the system abstraction visible to programmers. After the modification, the application and OS work in a collaborative way supporting the aforementioned features. Having implemented the *t-kernel* on MICA2 motes with an 8-bit processor and 4KB RAM, we evaluate its performance by measuring the overhead and execution speed. We analyze the CPU utilization in sensor network applications, and verify that, though CPU-bound computation tasks may slow down 0.5–4 times, the performance of applications under typical workloads does not degrade. The *t-kernel* significantly enhances developers' ability to design sophisticated applications and protects WSNs from accidental programming errors. To the authors' best knowledge, the *t-kernel* is unique in the follow ways: it performs efficient binary translation on highly resource constrained sensor nodes with only 4KB RAM, it provides software based virtual memory without repeatedly writable swapping devices, and it protects OS from application error without memory protection or privileged execution hardware.

1 Introduction

Many wireless sensor networks (WSNs) are based on mote-class devices, e.g., the Berkeley motes[32]. 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 and some other networked low-power systems – likely with increasingly small form factors [17, 8]. While some WSNs will indeed migrate to more expensive and capable devices, this 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 can be quite sophisticated. Currently, the OS' support to application development and runtime execution are extremely limited. By studying typical applications and anticipating the future needs of these systems, we identify that OS protection (boundary between OS and application), virtual memory, and preemptive scheduling are three standard features found in conventional OSs that would significantly improve WSN development and execution. The problem is that many microcontrollers do not have hardware support for any of these key features. Hence, we design the *t-kernel* that supports virtual memory by using flash and no memory management hardware, provides priority preemptive scheduling and guarantees the safety of OS code from bugs or problems in application code. We use a load-time modification approach that modifies the necessary native instructions in order to provide an enhanced system abstraction without supporting hardware. The result is that, without assuming advanced hardware support, the *t-kernel* can guarantee solid OS control against faulty application code, perform preemption at 16 priority levels, and support a 64KB virtual memory space over 4KB physical memory. With the inherent advantages that these mechanisms impart, this raises the level of abstraction and system support for developing reliable and sophisticated WSN applications.

The load-time code modification necessarily expands code size and incurs run-time overhead. We have fully implemented the *t-kernel* on Berkeley MICA2 motes and studied its performance. Application modules are evaluated with and without the *t-kernel* and slowdowns are identified. These slowdowns are the costs of using pure software solution to obtain the significant benefits of virtual memory, preemption, and OS protection. Observing common WSN applications spending more than 90% of CPU time in idle state, we verify

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.

that the slowdown does not degrade the performance of typical WSN applications. Moreover, evaluation reveals that the *t-kernel* is one order of magnitude faster than the virtual machine based approach, which is an alternative way to enhance system abstraction.

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 describe 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 summarizes the paper and briefly discusses future work.

2 Motivation

It is counterintuitive that energy-and-cost-efficient platforms require advanced OS features. Many embedded systems worked without them. However, with the advent of WSNs and, generally, networked embedded systems, a new design context has emerged with very-low-power embedded microcontrollers and complex application requirements. To generalize major application requirements and illustrate the need for advanced OS features, we study three real-world scenarios and illustrate the difficulties researchers encountered when developing high-quality WSN systems.

2.1 Scenario 1: memory shortage

VigilNet [28] 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 NesC [24] 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 management overlay scheme is well known to be a deprecated method because it is application specific, inefficient, labor-intensive, and error-prone. Some recently sensor nodes have 8KB–10KB RAM, but the memory scarcity is still limiting the development of sophisticated WSN software.

2.2 Scenario 2: Inaccurate sampling

When some researchers were developing acoustic processing algorithms for WSNs, they found that the timings of the microphone samplings were inaccurate. Further investigation discovered that it was due to the FIFO task scheduling in the OS. When there is a long-running computation task, other tasks, including the acoustic sensing tasks, have to remain in the FIFO queue longer, resulting in a timing discrepancy.

This scenario showcases a mismatch between the OS' scheduling policy and the application requirement of processing time-based events. One possible solution is to avoid using tasks and have acoustic sampling ride on clock interrupts. However, this is more like a workaround rather than a real solution. In large applications, time-based events are heavily used for myriad of functions, such as periodic sampling, wait time-out, and state transition. For example, VigilNet uses 27 timers to drive 27 time-based events. If they are all directly hooked to clock interrupts, the system's performance suffers, and race conditions teem. Moreover, after moving several time-based processing logics into clock interrupts, the programmer will find the same conflict happens in the clock handler's switch statements that calls these logics. So the problem is not solved – it just moves from the OS scheduler to the interrupt handler.

2.3 Scenario 3: OS control

The Extreme Scaling [20] project studies large-scale WSNs of more than 1,000 nodes. In the assumed application domain, such a network can be deployed by aircraft in an inaccessible area and operate in an unattended manner. For maintenance, it is essential that the nodes be always responsive to wireless control requests, such as reprogramming commands. However, this turns out to be difficult because the OS and application share the same memory space and have the same privilege. As a result, 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 employed a sanity operations by hardware – a grenade timer automatically restarts the sensor node periodically [20]. After restarting, the boot loader listens to the wireless channel before executing application code. This ensures that the administrators can reprogram the network but complicates application design because it must consider such periodic rebooting. Moreover, the periodic restarting has a fairly coarse control granularity. Not wanting a node to restart too frequently, thus 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.

2.4 Design context analysis

The three scenarios discussed above illustrate difficulty issues programmers encounter while developing large-scale real-world WSN applications. We notice that, in PCs and servers, these problems do not exist because the hardware is much more advanced. For example, most PCs have sufficient physical memory and virtual memory hardware. Similarly, if the hardware supports privileged instructions, the OS can control the computer and periodically perform system maintenance services by exclusively handling clock interrupts.

However, such advanced hardware support does not exist on many sensor network platforms. One important reason is energy consumption. While a Pentium 4 processor can easily consume 90W power, and a handheld device typically uses 1W power, some 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. In the past few years, some high-end sensor nodes have scaled up with the technology progress to employ better processors or co-processors [2, 25, 48]. But many still stay with a simple hardware architecture without privilege support, virtual memory translation, or memory protection [20, 5, 21, 42]. This is also true for some recent microcontrollers designed specifically for sensor networks. Even when high-end sensor nodes are used, it is not unusual that

the designers incorporate very low-power sensors to form a hybrid network [33, 43]. Hence, the technology development does not diminish the class of sensor nodes featuring simple hardware and very low power consumption. This class of sensor nodes includes most of the typical sensor network platforms, including MICA, MICA2, XSM, ExScal, MICAz, and so on. These sensor nodes target for very low-power operations or long-lifetime systems.

Besides energy efficiency, the accompanying requirements of small form factor and low cost also contribute to the modest hardware configuration of very-low-power sensor nodes [44, 33]. In general, with energy, form factor, and cost considered, it is undesirable, if not impossible, to upgrade the hardware configuration. Hence, the resource constraints and the lack of advanced hardware features will continue to be the reality in future on these platforms. Such energy-and-cost-efficient platforms is what this work targets for.

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 wider range of services, including topology control, routing, aggregation, network management, power management, security, and maintenance [28, 39, 16, 23, 35]. 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 three scenarios discussed before.

We propose to design a new OS kernel to close the gap between the application and hardware. Specifically, we identify OS protection, virtual memory, and preemptive scheduling to be three OS features that can significantly enhance the functionality and performance of WSNs. Therefore, the new OS kernel, *t-kernel*, should support these features without assuming advanced hardware support.

3 Assumptions, challenges, and approach

Compared to the PC industry, microcontrollers used in WSNs and embedded systems have a much wider variety [32, 13, 21, 46]. To ensure wide platform support and facilitate future porting, a new OS kernel design for this area must carefully choose its assumptions on hardware so that they represent a wide range of platforms and do not constrain future low-power designs. We choose the following assumptions about the hardware.

- Assumption R: Reprogrammable – The system allows write data into some memory space and execute it as program.
- Assumption E: External nonvolatile storage – Low-power, nonvolatile, and relatively large-capacity external storage is available, and it is 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 will use flash as a representative of nonvolatile storage, but the discussions apply to general read-friendly nonvolatile storage devices.

Note that the requirements for REM computers are fairly minimal. If the hardware has advanced features, such as privilege support, an OS developed for REM computers can still run on the better hardware, and optimization can be conducted to take advantage of the advanced features. Hence, by assuming a minimal set of hardware features, we enhance the portability and facilitate the construction of a general-purpose OS for energy-and-cost-efficient computers, which often prefer to scale down the energy consumption, instead of capacity, as technology improves.

There are three major challenges in designing *t-kernel* for a REM computer: stringent resource constraints, possibly write-unfriendly external nonvolatile storage (flash), and lack of hardware support. With these challenges, it is impossible to use traditional OS design techniques to implement OS protection, virtual memory, and preemptions on REM computers. Hence, a new design is needed.

After a program is written, three procedures combined together determine its behavior – compiling/static linking, loading/dynamic linking, and running on a specific architecture. Potentially, we can change any of these three procedures to affect the program's behavior so that it behaves as if it were running on a system with a higher abstraction (e.g., more memory). Traditionally, compile-time and run-time techniques have been extensively exploited for this purpose. Take virtual memory as an example, the address translation can be accomplished by compiling memory access statements to be function calls, like some implementations of distributed virtual memory systems. Alternatively, such translation can be conducted at run time by virtual memory hardware, or by a virtual machine. The *t-kernel* chooses to perform load-time processing to enhance the system abstraction, for the following reasons.

- We cannot assume a "correct" compiler has generated the application code. As a general OS kernel, the *t-kernel* executes application programs that may have been directly written in assembly or machine code.
- Compilers can only perform static checks. It is very hard for them to detect many software faults (e.g., infinite loops) without limiting the programming language's flexibility or capability.
- Run-time techniques, such as virtual machines, involve high overhead as the harsh resource constraints limit the level of optimization we can use.

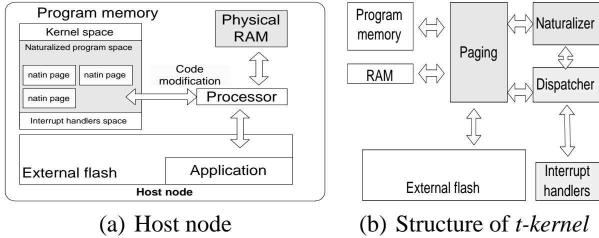
Hence, we design the *t-kernel* to employ extensive load-time processing to support OS protection, virtual memory, and preemptive scheduling. Following this new approach, the kernel has distinct ways of organizing resources and coordinating operations.

4 Design

This section introduces the design of the the *t-kernel*. First, we give a high-level description of the *t-kernel* and define terminology in Subsection 4.1. Then Section 4.2 and 4.3 discuss the OS protection and virtual memory in detail. Finally, Subsection 4.4 discusses scheduling and some important design issues.

4.1 Overview

As an operating system, the *t-kernel* resides on a sensor node, henceforth called a **host node**, and boots the host node when it is turned on. Figure 1(a) shows the hardware and software components in a host node. The program memory is the memory space from which the processor fetches instructions, and the *t-kernel* resides in the Kernel space in the program memory. Logically, the program memory is a different memory structure than the physical RAM (i.e., Harvard architecture) [32, 21]. Physically, they can be a section in one memory structure shared with data memory.



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

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

The *application* is in native instructions of the host node's hardware architecture, and resides in the external flash. After initializing its own working environment, the *t-kernel* loads and runs the application. During the lifespan of a host node, the user application may change via wireless reprogramming [35, 41]. In usual situations, however, the *t-kernel* does not change, and is responsible for maintaining the host node in a healthy operating state to respond to wireless control commands.

In order to ensure the stable operation of a host node without relying on the application, the *t-kernel* modifies application code when loading and dispatching the application for execution. We roughly define a small block of consecutive instructions as a code page, which, on specific machines, can be conveniently specified to correspond to a physical page in the external flash. When the control flow reaches a new code page, the *t-kernel* reads that page 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 per-instruction code modification the *t-kernel* performs on a small-memory platform to significantly enhance system abstraction. During the process of naturalization, only a small fraction of instructions in the application code are changed to different types or numbers of instructions. Most of the natins are the same as their corresponding instructions in the application code. Naturalized instructions guarantee not to compromise the host node, thus they are trusted and executed without restraints.

Blocks of natins are organized into *natin pages* residing in the naturalized application space in the program memory. When one natin branches to or calls an address in another code page, the *t-kernel* helps transfer the control flow. If there is no natin page for the destination address, the *t-kernel* reads the corresponding code page and naturalizes it. This procedure implies that naturalization is an incremental, and one-time process. Application code pages are naturalized incrementally when control flow reaches them. After a natin page is created, it is re-used when the control flow reaches the corresponding application code again. Hence, a natin page is naturalized only once. An exception is the "bridging" process, to be discussed in Section 4.2.2, which re-writes the branch destination addresses in some natin pages.

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 naturalize the code page. 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 OS control

Control and data 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 subsection, we introduce the naturalization process and explain how it guarantees OS control. The kernel data protection will be introduced in Section 4.3 with virtual memory.

4.2.1 Kernel/application transition

The dispatcher keeps track of the program counter of the application and dispatches the corresponding natin page. The program counter, denoted as **VPC** in *t-kernel*, points to an address in the application code. Meanwhile, we call the actual program counter on the processor in the host node the host program counter, or **HPC**. The VPC begins with the starting address of the application. When a natin page is executed, it guarantees a return to the dispatcher sooner or later with information about the next VPC. After returning to the kernel, the dispatcher repeats the naturalize/dispatch process with the next VPC.

In order to guarantee that the execution of a natin page returns to the dispatcher, the naturalization process modifies all branching 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 bit 24-31 of DEST
ldi r30, DEST2; load bit 16-23 of DEST
ldi r29, DEST1; load bit 8-15 of DEST
ldi r28, DEST0; load bit 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 natin page that services the destination VPC.

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 place in the natin page that corresponds to the destination VPC. A straightforward solution similar to traditional context switching is to restore

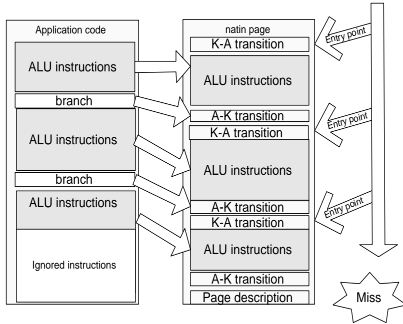


Figure 2. Application code page and natin page

register r28–r31 and machine state flags from the stack, and then point HPC to the entry point. 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 in 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 it services the destination VPC, and continues to execute if it does; otherwise, it jumps to the next entry point. Therefore, the seek for the suitable entry point is conducted by the natin page by a **cascading branch chain**. 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 branches hit 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. The kernel puts the destination VPC's last 8 bits in r29; 2. The kernel restores r28; 3. kernel uses a register indirect jump to jump to the natin page using register r30 and r31; 3. The natin page restores r30; 4. the cascading branch chain executes until there is a hit (a match of r29 and the entry point's 8 least significant bits); 5. the natin page restores the status register, r29, and r31. After this point, the natin page resumes the execution of the application code. When jumping to the natin page, the *t-kernel* makes sure there is no ambiguity on the high bits of the the destination VPC, hence a match on r29 ensures a hit. If the cascading branch chain does not hit the entry point, it jumps back to kernel.

Illustrated in this case, interweaving concerns 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 Bridging

After modifying branching instructions to perform A-K transitions, the kernel is guaranteed to get hold of the CPU, either because branching happens or because the control flow reaches the end of a natin page. Hence, the kernel is guaranteed to take hold of CPU very frequently.

However, such transitions involve some overhead and can significantly reduce the computation speed because branch instructions are common. To promote performance, transitions are classified into several types and handled differently. One type of A-K transitions is called "home transitions". Such transitions always jump back to a position in the dispatcher called "homeGate". Another type of A-K transitions, called "town transitions", is designed to speed up the branches. The first time a town transition occurs for a specific VPC, it returns to a specific location in the dispatcher called "townGate". The "townGate" logic examines the source and destination of the branch, and re-writes the natins in the source to increment an 8-bit system counter, and, if the counter is not zero, directly jump to the destination; otherwise, it jumps to "homeGate".

Hence, after the first execution of a town transition, the natins for that town trap changes to a jump directly to the destination most of the time. We call such a process **bridging**, since it directly links an A-K transition and a K-A transition and form a direct natin to natin jump. We call such a jump an **A-A transition**.

Among branching instructions, conditional branches and "rjmp" (relative jump) are modified to be town transitions. Some other branching instructions are transformed into home transitions. Returns are also modified to be home transitions to prevent applications from abusing return addresses. As the majority of branches are conditional and most of the town transitions are fast A-A transitions, the application's execution speed is significantly accelerated. Meanwhile, after no more than 255 A-A transitions, the system counter will reach zero and an A-K transition will be performed. Hence, the kernel is still guaranteed to occupy CPU fairly frequently – calculation reveals that, in every 8.8ms, the *t-kernel* will get hold of CPU at least once.

4.2.3 HPC/VPC look-up

An efficient look-up for the HPC from a specific VPC is a key to improving performance. In the *t-kernel*, the look-up is performed at three levels, illustrated in Figure 3. The slowest, but most reliable level, is at the program memory itself. Each VPC is hashed to a number of natin pages in the program space, and each natin page's cascading branch chain contains all the entry points in the page. Hence, the *t-kernel* looks up the natin page that services the VPC by looking at all the possible natin pages and check whether any of them services the VPC. In the middle level, the *t-kernel* maintains a 2-associative VPC table in RAM. The table is indexed by part of the VPC address, and each line in the table has two entries, each of which contains information about the location of the natin page and a tag describing its corresponding VPC. If either of the entries' tag matches the VPC's, the *t-kernel* uses the natin page described in that entry. The fastest level is a VPC look-aside buffer. It is a direct-mapped buffer indexed by several bits of the VPC (6 bits in the current implementation), and each entry in it contains the natin page number that

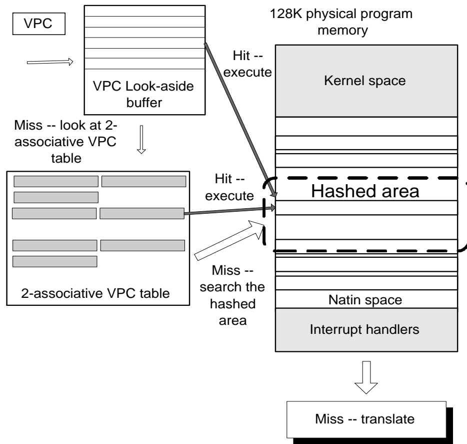


Figure 3. Three-level loop up for a VPC

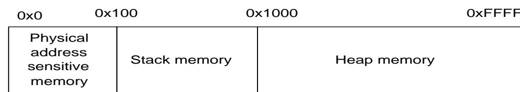


Figure 4. Example of virtual data memory configuration

services the VPC. Direct-mapped, the VPC look-aside buffer is fast, but has a higher miss ratio than the 2-associative VPC table.

4.3 Naturalized virtual memory

The memory space *t-kernel* provides to the application is a flat virtual memory much larger than the physical data memory of the host node. The virtual-physical memory address translation, boundary check, and memory swapping are handled by naturalized instructions without virtual memory and exception hardware. To distinguish such a virtual memory from traditional virtual memory, we call it naturalized virtual memory.

4.3.1 Memory areas

To promote speed, *t-kernel* defines three types of memory areas with different attributes, and differentiates memory accesses to these areas.

- Heap memory: swappable and relocatable.
- Stack memory: relocatable but not swappable. For efficiency, this memory area should be physically contiguous.
- Physical address sensitive memory: not swappable and not relocatable.

The specification of these memory areas in the naturalized virtual memory space is configurable for specific systems. Figure 4 shows an example of memory configuration of a 64KB virtual memory space.

4.3.2 Naturalization of memory accesses

The naturalizer treats memory accesses to different memory areas differently, making the common case fast. 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 almost as fast as native instructions. Accesses to the physical address sensitive memory area is also very fast since the virtual and physical addresses are the same.

The accesses to the heap memory area are the slowest. 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 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 that 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 of such a program statically to the data frame area for small memory programs.

When the address of the memory access is not known at load time, the naturalization process modifies the memory access into a branch structure. First, the memory address is compared to the boundary of the stack memory area. Then, if it is in the stack memory area, the memory access instruction is executed as a native instruction. Otherwise, the naturalized code performs an A-K transition, and the *t-kernel* handles the memory access as a heap memory area access.

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 is a systematic work involving 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 “push” and “pop” instructions access the stack memory area and are executed at native speed without boundary checks. The naturalization process assures that the application cannot execute a number of pops and pushes to invade the kernel heap.

4.3.4 Swapping

Swapping is a grand challenge to the naturalized virtual memory because the external flash is not write-friendly – after 10,000 writes, a flash page can no longer be used. Hence, its design principles and scheduling policy are different than those for traditional hard disks, where a sector can, conceptually, be written infinite times.

The same number of swap-outs, if directed to different pages, can either succeed or destroy the flash. Hence, two important performance metrics for naturalized virtual memory are lifetime and utilization. We define the lifetime of an external flash, denoted as L_f, as the number of swap-outs it services before any part of it becomes unstable. When the swap-outs are evenly distributed to all the pages in the flash, it accomplishes its maximum lifetime, denoted as L_{max}. In a small-memory system, L_{max} is very difficult to accomplish

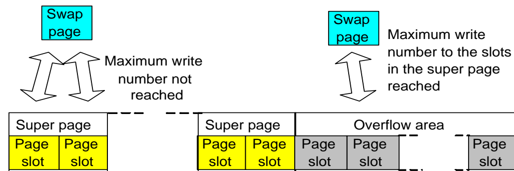


Figure 5. Swapping area organization

due to limited indexing capability. We define the utilization of a flash as L_f/L_{max}.

Figure 5 illustrates the organization of the swap area on the external flash. The super pages represent an associative cache where pages can be quickly allocated by looking at the swap table. Hence, swapping to one of the page slots in the super pages are fast. We call these swaps fast swaps. A soft locking mechanism improves the fairness of super pages swaps – when one super page cannot service swap-outs any more, the *t-kernel* tries to use other super pages. The overflow area provides a way to increase the fairness of the utilization of the flash – when a super page for a data frame is exhausted, the *t-kernel* swaps the data frame to the overflow area in a sequential manner. The seek in an overflow area with N pages is much slower with a complexity of O(N).

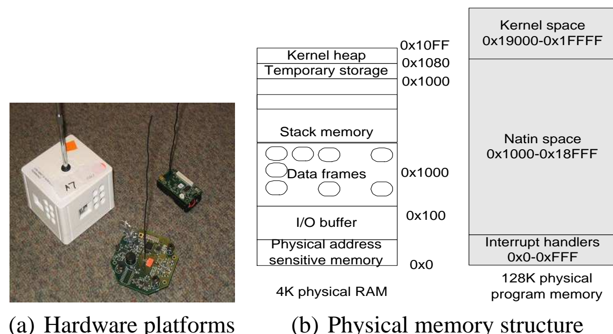
We expose a system parameter, A_f, which is the associativity of the super page and can be configured by system administrators. Generally, the larger the A_f, the faster the swap speed, but the shorter the flash lifetime, and vice versa. The default value of A_f is 2. Section 6.4 quantitatively depicts the relation between A_f and L_f. With the default setting, the flash can service 6.4 million swaps and 20% of them are fast swaps. The application designer can tune A_f to adjust the swap speed and flash lifetime to meet application requirements.

4.4 Discussion

We will only briefly describe the preemptive scheduling in this paper. In the *t-kernel*, tasks are the schedulable units. We maintains a single stack for all the tasks as well as the application and *t-kernel* itself. The task scheduling happens in two situations – when a task is posted and when a task is completed. If the new task's priority is higher than the task that is currently running, the *t-kernel* schedules the new task to run. Put another way, the current low-priority task is preempted. Otherwise, if the new task's priority is lower, the *t-kernel* resumes the application's execution.

From the description of the design, we can clearly see that many design choices are affected by the harsh resource constraints, especially, the small memory and write-unfriendly external flash on REM computers. Another driving factor is performance – Though resource constrained, it is still necessary for the *t-kernel* to run at an acceptable speed so that applications' performance do not noticeably degrade. If the kernel is not carefully designed, the application can easily slow down for 100 times when using software solutions for problems traditionally requiring hardware support. This is a grand challenge and affects almost every design choice. We study the performance in detail in Section 6.

Interrupt handling in *t-kernel* is conducted by the kernel and application together. The kernel interrupt handlers' function is simply to catch the interrupt and invoke the application-provided handler. Hence, applications essentially takes charge of the interrupt handling. This gives the application developers the maximum flexibility.



(a) Hardware platforms (b) Physical memory structure

Figure 6. The *t-kernel* implementation

The flexibility *t-kernel* gives to application developers is not limited to controlling interrupts. The *t-kernel* provides the developers an enhanced system abstraction with restraints hidden but instruction set unchanged. So developers have the freedom of using any instructions the host node supports to manipulate hardware. Such flexibility and extensibility are important advantages of the *t-kernel* compared to other approaches.

5 Implementation

We implemented *t-kernel* for the ATmega128L microcontroller [1], which is broadly used in many WSN platforms [32, 20, 11, 4, 3]. We have tested *t-kernel* on MICA2 family motes, including MICA2, XSMv1, and XSMv3 (ExScal) motes, as shown in Figure 6(a). Using *t-kernel* on other ATmega128L based system may involve replacing some hardware-specific routines (such as flash driver). As discussed in Section 3, porting to platforms with a similar or better microcontroller is definitely possible, given that the REM assumptions are met. Table 1 lists specifications of the hardware and some system parameters for *t-kernel* on MICA2. We intend *t-kernel* to be an open-source software that can benefit the community of sensor network researchers and generally embedded system developers.

Hardware parameters	Data RAM	4KB
	External flash	512KB
	Program memory	128KB
OS parameters	Virtual memory	64KB
	Data frame	64 frames
	Look-aside buffer	64 entries
	2-associative VPC	256 entries
	System stack	1KB
	I/O buffer	516 bytes
Implementation details	Code size (source)	10,056 lines
	Code (binary)	28,950 bytes

Table 1. System characteristics

The current *t-kernel* implementation on MICA2 motes supports 64KB of virtual data memory (16 times of the physical memory). The configuration of the virtual memory space is shown in Figure 4, and the organization of the physical memory is shown in Figure 6(b). Empirical study leads us to choose bit7-bit2 of the VPC to index VPC look-aside buffer, and bit15-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 [28], 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. Specifically, we quantitatively evaluate *t-kernel* in the following aspects:

- Measure the overhead of transitions.
- Measure the execution time of programs with different branch frequencies.
- Measure the speed of naturalized virtual memory – stack memory access time, heap access time without swapping, heap access time with swapping.
- Evaluate the relative execution time of kernel benchmark programs and assess the overall impact of slowdown on applications.
- Verify the execution speed with typical and stressed workloads.
- Analyze the performance of virtual memory swapping, including swapping efficiency and the lifetime of the swapping device.
- Analyze energy efficiency.
- Compare *t-kernel*'s performance with Maté
- Verify the OS protection against a number of application errors.

6.1 Overhead of naturalization

During the naturalization process, most of the instructions in the application's binary code are literally copied to the naturalized program page and are henceforth executed at native speed. Meanwhile, the kernel transitions, the naturalized virtual memory, and the naturalization process itself still have a significant impact on the execution speed of an application. We discuss the overhead of kernel transitions and the naturalization process in this subsection. The overhead of naturalized virtual memory will be discussed in Section 6.2.

We use several measurement programs written in assembly to study the overhead of kernel transitions. The "*iter*" program is a loop with only logic/arithmetic instructions in the loop body and a "brcs" (branch if carry is set) instruction to branch back to the start point of the loop. The loop body has several configurations. At minimum, the logic/arithmetic instructions in the loop body only increment the loop iteration variable. Other configurations contain different numbers of extra arithmetic/logic instructions in the loop body. We measure the execution time of *iter* iterating 8,388,608 (0x800000) times with 4–20 extra instructions in the loop body. The results are shown in Table 2.

# of extra	Native (sec.)	*t-kernel* (sec.)
0	12.05	33.83
4	16.58	38.28
8	21.21	43.07
12	25.71	49.68
16	30.26	54.32
20	34.74	58.79

Table 2. Execution time of *iter*

If we denote the execution time of a kernel transition per

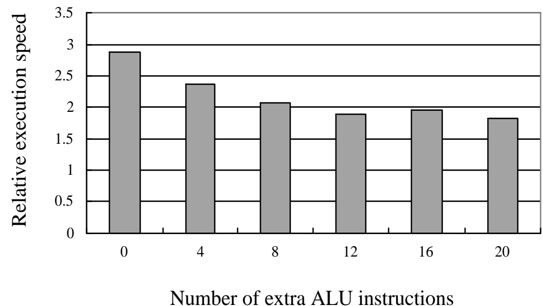


Figure 7. Relative execution time of *iter*

branch as t_f, the time for naturalizing the code as T_n, the time of 4 arithmetic/logic instructions as t_a, the time for the loop handling logic (the compare and branch) as t_l, and the time for all other instructions as T_o, we can deduce T_f from the measured data by solving the following equations.

$$T_o + t_l * 8388608 = 12.05$$
$$T_o + (t_l + t_a) * 8388608 = 16.58$$
$$T_o + T_n + (t_l + t_f) * 8388608 = 33.83$$
$$T_o + T_n + (t_l + t_f + t_a) * 8388608 = 38.28$$

The result is that $t_f = 2.6\mu s$. The ATmega128 microcontroller on MICA2 runs at 7.3827MHz. Hence, the execution time of a kernel transition is, on average, around 20 cycles. At minimum, a kernel transition saves/restores registers, checks the stack pointer, and increments the system counter. It sometimes also needs to look up the destination address, trigger the naturalization of a new page, and re-link naturalized pages. Considering the numerous tasks, it is quite efficient to have an average execution time of 20 clock cycles.

Obviously, the more frequent branching instructions occur in an application, the higher overhead (in the amount of t_f time per branch) the kernel transition imposes on the execution time. For an extremely branch-intensive application like *iter* with zero extra instructions, the application's execution time may be 2.8 times as long as in native instructions, i.e., the application execution on the *t-kernel* has a 1.8 times slowdown. If the application is less branch-intensive, the overhead is less, and the application executes faster.

We define the **relative execution time** of the execution of a program to be the ratio of its execution time on the *t-kernel* to the time in native instructions. Hence, a relative execution time of 3 means a 2 times slowdown. Henceforth, we use the relative execution time, not the execution time, to evaluate the execution speed because the former is less dependent on the size of the dynamic trace of a program. Figure 7 interprets the data in Table 2 into relative execution time *iter* in different configurations, representing the execution speed of applications with different proportions of branching instructions.

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 *t-kernel*. *iter.heap* accesses the heap memory area without incurring swapping. *iter.swap* sweeps 16KB of the naturalized virtual memory space (from 0x2000 to 0x6000) with steps of 16 bytes to ensure there is

Name	Numberof accesses	Native (sec.)	Cycle	*t-kernel* (sec.)	Cycle
iter.mem	8388608	5.74	2	26.5	16
iter.heap	8388608	N/A	N/A	26	15
iter.swap	1024	N/A	N/A	2	149815
iter.nomem	8388608	3.42	N/A	9	N/A

Table 3. Memory access performance

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

a page fault for each memory access and swapping for some of them. To isolate the execution time of non-memory access 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 with no memory access instructions. Table 3 summarizes the evaluation results.

Based on these measurements, we calculate that, in the *t-kernel*, the slowest stack access takes 16 cycles, the heap access without swapping takes 15 cycles, and the heap access with swapping takes 149,815 cycles. We can also calculate the swap-out time to be 20.3ms. According to the flash's chip's data sheet, an erase/write cycle takes approximately 20ms. Hence, the swap-out time is comparable to the hardware's limit. This is a similar situation to traditional virtual memory where a swap to disk costs hundreds of thousands of CPU cycles.

6.3 Assess application execution speed

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 [30]. 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.

All these programs were written in TinyOS' programming language – NesC [24]. To keep binary code of these programs to be the same as it is in current applications, we retain the compiler settings and still build them to be TinyOS applications. The build process only changes several linker parameters to locate the ".data" and ".bss" memory sections to pre-defined heap memory areas. Generated binary images are uploaded to MICA2 motes, and are executed directly (with TinyOS) or executed with *t-kernel*.

Figure 8 compares the relative execution time of the kernel benchmark programs. Different programs' performance differ noticeably. *am* has a relative execution time of 3.35, comparable to *iter* with 4 extra instructions in the loop body. However, *timer*'s relative execution time is 4.6. This corresponds to a 3.6 times slowdown, which is higher than the slowdown due to kernel transitions illustrated in Figure 7. The reason is that *timer* has frequent heap memory accesses, which are slower than arithmetic/logic instructions and stack

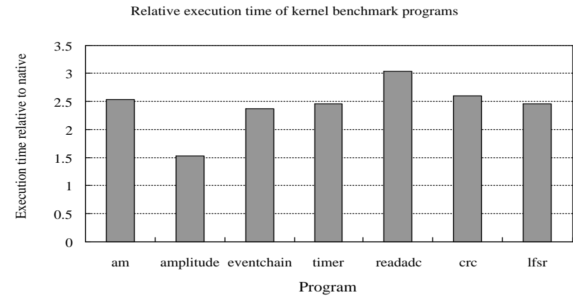


Figure 8. Performance of kernel benchmark programs

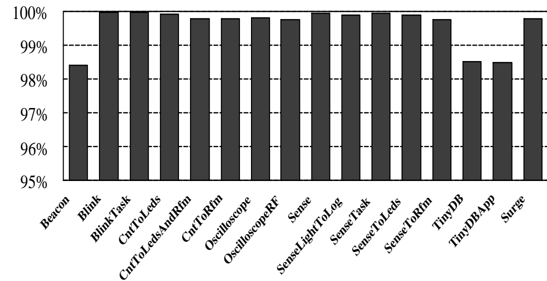


Figure 9. CPU idle ratio of TinyOS applications

memory accesses, as will be studied in Section 6.2.

We observe, however, a higher slowdown (3.67 times) for *crc* even though it has no heap accesses. This is due to a combination of "sbrs" and "rjmp" instructions close to the end of a natin page, which makes efficient handling impossible. This is an example where the situation of the some constraints makes it difficult to perform efficient naturalization. It also suggests an opportunity for code optimization if we have a *t-kernel* compiler.

The programs *lfsr* and *readadc* have slowdowns of 3.4 and 2.3 times, respectively. With a relative execution time of 1.47, *amplitude* shows the best performance among the benchmark programs.

Overall, we expect CPU-bound computation to have a relative execution time of 1.5–5 on *t-kernel*, corresponding to a slowdown of 0.5–4 times. This is a noticeable overhead. However, most of current sensor network applications are not CPU-bound. Hence, the applications' performance do not slow down as much.

To determine whether the computation slowdown is acceptable to sensor network applications, we examine the typical work load of WSN applications. Shnayder et al. listed the energy breakdown of 16 TinyOS applications [50]. Conservatively assuming the CPU's power in the idle and active state are the same, we deduce the amount of time the CPU spends on idle and active state, as shown in Figure 9.

We find that, without exception, the applications spend 98% of the CPU time in idle mode. This result is not surprising – WSN systems are not designed for heavy computation workload, and sensible developers seldom write programs to stress the CPU. Moreover, many WSN applications are intrinsically I/O intensive. Obviously, with a CPU utilization of less than 0.02, a slowdown of 0.5–4 times does not noticeably degrade the applications' performance.

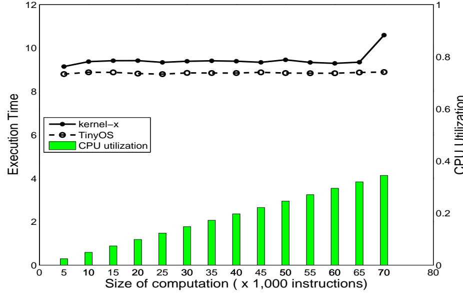


Figure 10. Execution Time and CPU utilization under different workloads

Note that the estimation in Figure 9 is very conservative – Usually, the CPU idle power is no more than 40% of that for the active state, and some time in the active state is spent on spin lock logic where the amount of time, not the number of instructions, counts. Hence, the CPU utilization is, in reality, even lower than 0.02.

Generally, for energy-and-cost-efficient sensor nodes running typical applications, memory and energy, not CPU cycles, are scarce resources and often the bottleneck of system performance. 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, that 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). So, the PeriodicTask application captures the typical work mode in most of sensor network applications. Note that existing TinyOS applications, like those listed in Figure 9, have very low CPU utilization. Trivially, these applications run as smooth in *t-kernel* as in TinyOS. To reveal the limit of *t-kernel*, we vary the amount of computation in each task to examine not only typical workload, but also stressed situations.

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 CPU utilization from 0.02 to 0.34. Let's roughly partition the workload spectrum to light (0.1 or less CPU utilization), moderate (0.1–0.2 utilization), stressed (0.2-0.3 utilization), and very stressed (0.3 or higher utilization). When the CPU utilization is low, the total execution time of a number of periodic tasks is dominated by the intervals between task arrivals. When the CPU utilization is very high, the computation time dominates the total execution time. Figure 10 shows the execution time of 300 computation tasks of different computation sizes. With each configuration, the execution times with TinyOS and with *t-kernel* are plotted The CPU utilization corresponding to the computation size is also shown in the same graph.

As Figure 10 shows, with light and moderate workload, the overall execution time is constantly around 9 seconds. The execution time with *t-kernel* is slightly longer because of the naturalization overhead for the periodic tasks. With stressed and very stressed workloads, the execution time with *t-kernel* stays around 9 seconds before the CPU utilization reaches 0.30, then shows an increase when the utilization grows higher than 0.3.

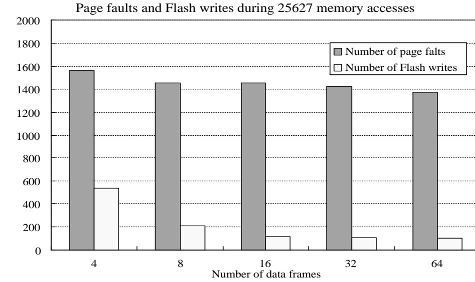


Figure 11. Virtual memory performance for slidingwin

Hence, when the CPU utilization is lower than 30%, the execution of PeriodicTask does show a difference than that with TinyOS. When the CPU utilization is higher than 30%, the PeriodicTask still runs well functionally, but may see a degraded performance in execution time and energy consumption. Hence, *t-kernel* is most suitable for applications with light or moderate workloads. Since the light workload (particularly, the low end of it) is what we usually encounter, most of the sensor network applications can be run in *t-kernel* without experiencing performance degradation.

The benefit of *t-kernel*, on the other hand, is very obvious – now the developers of the WSN can take advantage of enhanced and well defined system abstraction, including reliable OS control over the host node, expanded virtual memory space, and flexible task scheduling. In the experience of developing and deploying WSN systems in the past few years, we observe that the aforementioned features are so desirable that the benefits by far outweigh the cost in most situations. In addition, note that the kernel benchmark programs are compiled with a compiler without optimization for *t-kernel*. Future work has been planned to develop a compiler that can provide *t-kernel* specific optimizations and code generation to significantly reduce the overhead. This will make the *t-kernel* an even more favorable choice.

6.4 Performance of virtual memory swapping

We have studied the execution speed of applications with *t-kernel*. In the next few sections, we extend the evaluation to include other metrics, such as lifetime and energy efficiency. In this section, we study the performance virtual memory swapping.

Figure 11 shows the number of page faults and writes (swap-outs) to the external flash for the "slidingwin" application. This program mimics the operation of a network protocol with sliding window control. This program represents a harsh scenario for a virtual memory system because it has a big memory footprint without obvious locality. Five configurations of the data frame buffer sizes are compared. As we can see, even though the number of page faults remain approximately the same, as the number of data frames increases, the number of writes (swap-outs) to the external flash reduces significantly.

Figure 12 illustrates the trade-off between lifetime and swapping efficiency, assuming that total number of data frames is 64, and the swap area on the flash has 1024 pages. Note that the calculation in Figure 12 assumes the worst case – all accesses to a page are writes to the same data frame in it, all the 64 data frames are used and they are in 64 differ-

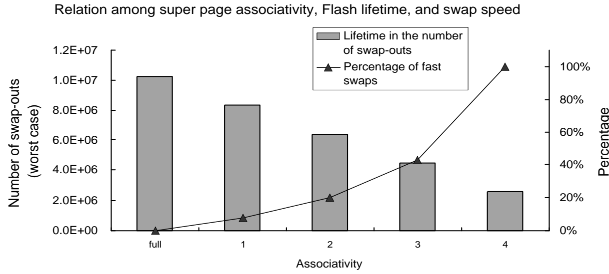


Figure 12. Tradeoff between lifetime and efficiency

ent pages. The result shows that, as the associativity of super pages (A_f) increases, the swapping efficiency (the percentage of fast swaps) increases, but the lifetime decreases. Based on this quantitative relation, the system designers can choose a setting most suitable for the application.

6.5 Energy effi ciency

One motivation of designing *t-kernel* is to avoid relying on complex hardware so that we can 1. utilize a broad range of existing microcontrollers 2. leave room for future low-power hardware designs. For the second reason, we must verify that, with the execution overhead, *t-kernel* is still more energy efficient than a hardware solution with similar capability. In this section, we compare the energy efficiency of *t-kernel* with a potential hardware solution. This hardware solution is a mote class device enhanced with a processor supporting privilege support and 64K SRAM.

The system energy consumption depends on the hardware technology, the fabrication process, the power management scheme, and the workload. Hence, it is necessary to make some assumptions. We list the assumptions and explain the rationale below. The assumptions are based on experimental results on MICA/MICA2 sensor nodes and estimations of practically achievable performance in a few years.

The sleep mode leakage power is the ultimate limit on the lifetime of a sensor node. Based on experimental data on MICA and MICA2 platforms, the leakage current in deep sleep mode ranges from 30μA to 100μA [44, 26]. The minimum, according to the microcontroller's datasheet, is 5μA. We assume the leakage current of a sensor node with 4K SRAM is 30μA, to reflect a low leakage current that is practically achievable. With the same technology, the SRAM leakage power is approximately proportional to size. Also, study revealed that the SRAM leakage power can account for 90% of the total leakage power in a microcontroller [19]. Hence, we conservatively assume the leakage current on a sensor node with 64K SRAM to be at least double of that with 4K SRAM, i.e., 60μA.

The active current, with the microcontroller and some sensors working, is typically 10–20mA, among which the microcontroller uses 8mA[50]. When the microcontroller is waiting for some events, it can enter an idle mode which reduces 4.8mA current. Hence, we assume the active current is 15mA and idle current is 10.2mA.

Most current sensor network applications let sensor nodes periodically sleep and wake up [28, 29, 44, 34]. The duty cycle typically ranges from less than 1% to 10%. We here assume a duty cycle of 1%, which should be the goal for most of applications lasting a long period of time, e.g., several months. When a sensor node wakes up, it samples the sensors and listens to the radio channel for a period ranging from a few dozen milliseconds to a few seconds [54, 28]. During the wake-up period, its operation is event driven, much like the Periodic Task discussed before. Hence, with a typical workload, *t-kernel* does not increase the length of the wake-up period. In this evaluation, we assume the workload represents 2% CPU utilization. However, when the microcontroller is waiting for interrupts, it typically enters the idle mode to save energy. Running *t-kernel* may reduce the amount of time spent in idle mode. This is where it may increase the energy consumption. Here we assume *t-kernel* uses three times as many clock cycles to perform the computations as the native code does. Hence, the microcontroller spends correspondingly more time in active mode when running *t-kernel*.

Another place *t-kernel* may increase energy consumption is virtual memory swapping – writing to flash is an expensive operation [45]. According to its datasheet, the flash on MICA2 consumes 1.2mJ energy when writing one page [1]. As mentioned before, *t-kernel* tries to minimize the number of swaps. Here we assume a swapping frequency of one swap per second during the wake-up period – and average this energy consumption to be a static current of 0.4mA current in the wake-up period.

With these assumptions, we can calculate the average power consumption of two systems – one with simple mote class hardware, but running *t-kernel*, and the other with enhanced hardware running native code. The simple hardware with *t-kernel* consumes 0.416mW, and the enhanced hardware consumes 0.487mW. Hence, compared to enhanced hardware, the *t-kernel* solution is 17% more efficient in energy consumption. In future, as some overhead is reduced by an optimizing compiler (Section 8) for *t-kernel*, the energy benefit will be come more significant.

Note that energy efficiency is one of the factors favoring simple hardware. Other reasons include cost and form factor. 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 [15]. In a low-end microcontroller, SRAM can account for 76% of the transistor count [19]. At present, the form factor constraint has not been as severe as energy. It will be a serious concern for dust-sized devices.

6.6 Comparison to the virtual machine approach

Virtual machines provide another level of abstraction above the raw hardware and can also provide enhanced features not directly supported by hardware. Having designed Maté and ASVM as virtual machine on TinyOS, Levis et. al. provided a way to execute WSN applications in a controlled environment and mentioned the possibility of supporting "functionality equivalent of the benefits a virtual memory system brings" [39, 40]. It is instructive to compare the *t-kernel* and the virtual machine approach and examine their difference on performance and other aspects.

Maté provides a stack based "virtual" architecture and interprets "capsules" of Maté bytecode. With a concise and easy-to-use language, it facilitates code mobility and is very

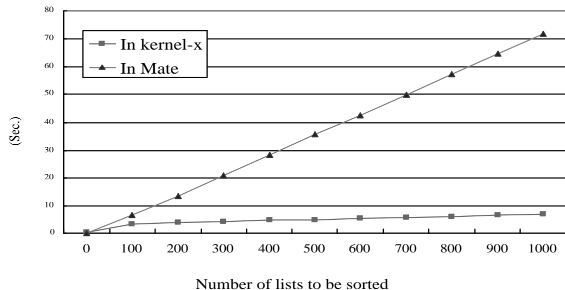


Figure 13. Execution time of insertion sorting

efficient data collection applications. Maté can execute the application program in a "safe" way since the bytecode can be analyzed at interpretation time. ASVM enhances the flexibility and efficiency of Maté by allowing programmers to re-construct the VM to incorporate application specific "handlers". Because the "handlers" essentially introduce application code into the OS, ASVM becomes less general-purpose and more application specific. We study Maté's performance with an insertion-sorting program, which is a popular choice for sorting small-data sets. The major purpose is to analyze the pros and cons of these two approaches.

Figure 13 plots the performance of insertion sorting in *t-kernel* and in Maté. The data shows that the *t-kernel* has a moderate increase in execution time when the number of lists increase from 0 to 100. This is because of the load-time naturalization overhead when new application code is used. After that, the execution time increases very slowly, because the naturalization process is a one-time overhead. On the other hand, Maté shows a constantly larger increase of execution time. Due to limited resource, sophisticated virtual machine optimization is infeasible on MICA2 motes. As a result, the overhead does reduce and the execution time grows at the same pace throughout the execution of an application. Overall, performance data shows that the load-time modification approach is faster than the the virtual machine approach when the number of lists increases.

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, *t-kernel* and virtual machines work at different levels and can combine together to benefit from each other's advantages. The former provides reliable kernel foundation and an extensible platform for developing new hardware drivers and system algorithms. The later provides expressive high-order language for application logic. Figure 6.6 shows a possible five-tier system architecture.

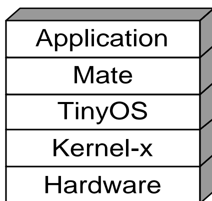


Figure 14. Five-tier system architecture

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. Table 5 lists a number of

Program error	Without *t-kernel*	With *t-kernel*
Left recursion	Crash	OK
Bad pointer	May crash	OK
Bad return address	May crash	OK
Memory leak	Crash	OK
Bad branch destination	May crash	OK
Invalid instruction	Crash	OK
Bad array index	May crash	OK
Infi nite loop	Halt	OK

Table 5. Controllability of sensor node with application errors

program faults and examine whether the sensor node is controllable when errors and failures arise due the these faults. With the *t-kernel*, the sensor nodes are always controllable despite these application errors because the OS is protected. The errors may make the application exhibit wrong behavior. But the OS' operation is not affected and critical system services, such as wireless control commands, are still responsive if they are implemented in kernel space.

Note there may be a latency between an application error and when the OS detects an application crash. Sometimes the latency can be very long. For example, if the application jumps to the second half of a two-word instruction, the second word may constitute a valid instruction syntactically. Hence, the application may continue to execute with a wrong logic for a long time. The *t-kernel* cannot detect all logic errors, or wrong behaviors, which is, in generally, an uncomputable problem. Instead, the *t-kernel* provides a reliable method to correct the errors after these errors are observed by programmers or system administrators.

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 [32], a number of OSes have been developed for WSNs and networked low-power systems. MANTIS and Contiki are two recent projects providing multithread support on MICA2 motes [6]. Not supporting virtual memory support, 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 [27]. The current implementation of SOS does not support OS protection. It has been discussed that future versions may incorporate this feature. 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 [37]. It has been ported to over 40 microprocessors including ATmega128. In the smallest configuration, it can be reduced to 2KB code and 200 bytes of data space, not including the stack space. The μC/OS provides a number of key functionalities needed by networked embedded applications, such as multitasking, synchronization, and timer management. However, when components providing these functionalities are included, the code and data sizes increase significantly beyond those in the minimal configuration. In a typical configuration allowing 10 tasks, the data size becomes 3K, not including 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" OSs – a decent set of features are only practically available when the platform has enough resources to accommodate all the corresponding modules. Sophisticated 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) [39] in Section 6.6. Other virtual machines include MagnetOS [10], whose "signal system image" unifies the whole network as one distributed machine [51], and SensorWare [12], which disseminates "script" code over the network. Kwon et al. developed a mobile agent platform called ActorNet [36]. ActorNet supports provides a virtual execution environment and supports virtual memory.

Reijers et al.[47] designed a scheme for code distribution on the EYES platform. Unlike virtual machines, Reijers' network-distributed code is not interpreted, but rewrite new "patches" into the program memory.

The task model in the *t-kernel* learned from the study and practice of using lightweight fibers [53, 7]. Like Aegis(Exokernel) [22] and Nemesis [38], *t-kernel* endeavors to avoid unnecessary high-level hardware abstractions, though *t-kernel* works on a much more resource-constrained platform. *t-kernel* advances further along this direction by exposing the full architectural features of the hardware platform to the application and forwarding all interrupts to the application.

Binary translation systems modify binary code to change platform abstractions and have been used for run-time optimization, architectural compatibility, and fast simulations [9, 14, 31, 18, 49]. However, due to the very different design context, most of the algorithms in this area cannot be used in the *t-kernel*.

Providing a "safe" execution environment, the approach closest to *t-kernel* is the sandboxing in software fault isolation [52]. They both modify part of untrusted code at program load time. However, these two methods are still significantly distinct in many ways. For example, sandboxing's major target is to regulate programs' memory access behavior so that they do not write memory or execute code outside their own memory spaces; In contrast, naturalization has a much more ambitious goal of providing enhanced abstraction that hardware does not support. For example, sandboxing rely 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. In addition, the *t-kernel* and sandboxing are different in their assumptions, mechanisms, extent, emphasis of performance metrics, resource constraints, and transformation complexity.

8 Conclusion and future work

We view the *t-kernel* as a small but solid step toward the design of robust, reliable, and energy efficient operating systems for WSN and general networked low-power computing devices. Such platforms play an important role in today and tomorrow's market, and impose numerous challenges.. Motivated by and based on observations on real-world WSN systems, the *t-kernel* is a novel solution for the unique design context. It has multiple contributions to the research on system software for energy-and-cost-efficient computers, including efficient binary translation with very small memory, efficient software based virtual memory, swapping on write-unfriendly devices, preemptive scheduling without clock interrupts, and OS protection without memory protection or privileged execution hardware. The design and implementation of the *t-kernel* not only provide a reliable OS kernel for WSN, but also prove the feasibility of using a pure software solution to efficiently implement several core OS functionalities that traditionally rely on hardware support.

As future work, an optimizing compiler for *t-kernel* is an important and promising step to accomplishing better performance. Some simple techniques, such as marking beginning of basic blocks in the binary code or reserving a register for OS use, can greatly help the naturalization process. Generally, we expect that a protocol between compiler and OS, in the form of descriptive tags in the binary code, can be defined to transfer compile-time and link-time knowledge to the OS. This will only slightly increase the code size, but will significantly improve the system performance.

Hiding certain resource constraints from programmers, *t-kernel* enables us to incorporate many necessary system services, such as network management, debugging and monitoring, and virtual execution environments, to support sensor network software development. Many such services have been proposed and implemented, but they could not co-exist in a complex application due to resource constraints. In addition, future work has been planned to support real-time specifications in *t-kernel*. Overall, the *t-kernel* enables us to construct a reliable and easy-to-program software architecture that facilitates the development of large-scale and high-quality WSN systems.

9 References

[1] Atmel site. *http://www.atmel.com*.

[2] Intel imote. `http://www.intel.com/research/exploratory/motes.htm`.

[3] Mica2dot mote. `http://www.xbow.com/Products/productsdetails.aspx?sid=73`.

[4] Micaz mote. `http://www.xbow.com/products/productsdetails.aspx?sid=101`.

[5] Mote IV. `http://moteiv.com/`.

[6] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han. MANTIS: System support for multimodal networks of in-situ sensors. *Proc. of 2nd ACM Intl. Workshop on Wireless Sensor Networks and Applications (WSNA)*, 2003.

[7] A. Adya, J. Howell, M. Theimer, B. Bolosky, and J. Douceur. Cooperative task management without manual stack management. In *Proc. of the USENIX Annual Technical Conf.*, June 2002.

[8] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. A survey on sensor networks. *Computer Networks*, 38, 2002.

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

[10] 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 *Operating Systems Review, ACM, 36(2):1-5*, Apr. 2002.

[11] 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)*, Jan. 2004.

[12] A. Boulis, C. Han, and M. Srivastava. Design and implementation of a framework for programmable and effi cient sensor networks. In *Proc. of Intl. Conf. on Mobile Systems, Applications, and Services (MobiSys)*, May 2003.

[13] A. Chandrakasan, R. Min, M. Bhardwaj, and S. Cho A. Wang. Power aware wireless microsensor systems. *Keynote Paper ESSCIRC*, Sept. 2002.

[14] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, and J. Yates. FX!32: A profi le-directed binary translator. *IEEE Micro*, 18(2):56–64, Mar/Apr 1998.

[15] 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), Nov. 2001.

[16] J. Considine, F. Li, G. Kollios, and J. W. Byers. Approximate aggregation techniques for sensor databases. In *Proc. of the 20th IEEE Int'l Conference on Data Engineering (ICDE '04)*, 2004.

[17] D. Culler, D. Estrin, and M. Srivastava. Overview of sensor networks. *IEEE Computer, Special Issue in Sensor Networks*, Aug 2004.

[18] 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. Symposium on Code Generation and Optimization*, 2003.

[19] 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*, Dec. 2001.

[20] 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 Fourth Intl. Conf. on Information Processing in Sensor Networks (IPSN'05)*, 2005.

[21] V. Ekanayake, C. Kelly 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*, Oct. 2004.

[22] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proc. of Symposium on Operating Systems Principles*, pages 251–266, 1995.

[23] D. Estrin, R. Govindan, J. S. Heidemann, and S. Kumar. Next century challenges: Scalable coordination in sensor networks. In *Mobile Computing and Networking*, pages 263–270, 1999.

[24] 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 (PLDI)*, June 2003.

[25] P. B. Gibbons, B. Karp, Y. Ke, S. Nath, and S. Seshan. Irisnet: An architecture for a world-wide sensor web. *IEEE Pervasive Computing*, 2(4), Oct.-Dec. 2003.

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

[27] C. Han, R. Kumar, R. Shea, E. Kohler, and M. B. Srivastava. A dynamic operating system for sensor nodes. In *Proc. of MobiSys'05*, 2005.

[28] T. He, S. Krishnamurthy, J. A. Stankovic, T. Abdelzaher, L. Luo, Radu S., T. Yan, L. Gu, G. Zhou, J. Hui, and B. Krogh. VigilNet: An integrated sensor network system for energy-effi cient surveillance. *ACM Transaction on Sensor Networks*, 2005.

[29] 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.

[30] M. Hempstead, D. Brooks, and M. Welsh. Tinybench: The case for a standardized benchmark suite for tinyos based wireless sensor network devices (poster). *1st IEEE Workshop on Embedded Networked Sensors (EmNets'04)*, Nov. 2004.

[31] M. W. Hicks, J. T. Moore, and S. Nettles. Dynamic software updating. In *SIGPLAN Conf. on Programming Language Design and Implementation*, pages 13–23, 2001.

[32] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for network sensors. *Proc. of ASPLOS 2000*, Nov. 2000.

[33] W. Hu, V. N. Tran, N. Bulusu, C.-T. Chou, S. Jha, and Andrew 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)*, Apr. 2005.

[34] J. Huang and S. Mishra. A sensor based tracking system using witnesses. In *Proc. of the 1st Intl. Workshop on Services and Infrastructures for the Ubiquitous and Mobile Internet (SIUMI'05)*, June 2005.

[35] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proc. of the 2nd ACM Intl. Conf. on Embedded Networked Sensor Systems (SenSys'04)*, pages 81–94, Nov. 2004.

[36] Y. Kwon, S. Sundresh, K. Mechitov, and G. Agha. Actornet: An actor platform for wireless sensor networks. In *to appear in Autonomous Agents and Multiagent Systems (AAMAS)*, 2006.

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

[38] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. T. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal of Selected Areas in Communications*, 14(7):1280–1297, 1996.

[39] P. Levis and David Culler. Maté : a virtual machine for tiny networked sensors. In *Proc. of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, Dec. 2002.

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

[41] P. Levis, N. Patel, S. Shenker, and D. Culler. Trickle: A self-regulating algorithm for code propogation and maintenance in wireless sensor network. *In Proc. of the 1st USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2004)*, 2004.

[42] 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)*, Apr. 2005.

[43] J. Liu and F. Zhao. Towards semantic services for sensor-rich information systems. In *Proc. of the 2nd IEEE/CreateNet International Workshop on Broadband Advanced Sensor Networks (Basenets'05)*, Oct. 2005.

[44] Alan Mainwaring, Joseph Polastre, Robert Szewczyk, David Culler, and John Anderson. Wireless sensor networks for habitat monitoring. *ACM Intl. Workshop on Wireless Sensor Networks and Applications*, Sept. 2002.

[45] 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)*, Apr. 2006.

[46] K. O'Hara and T. Balch. The gnats – low-cost embedded networks for supporting mobile robots. March 2005.

[47] N. Reijers and Koen Langendoen. Effi cient code distribution in wireless sensor networks. In *Proc. the 2nd ACM Intl. Conf. on Wireless sensor networks and applications*, 2003.

[48] A. Savvides and M. B. Srivastava. A distributed computation platform for wireless embedded sensing. In *Proc. of Intl. Conf. on Computer Design*, 2002.

[49] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. Davidson, and M. Soffa. Retargetable and reconfi gurable software dynamic translation. In *In Proc. of the Intl. Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, 2003.

[50] V. Shnayder, M. Hempstead, B. Chen, G. Werner-Allen, and M. Welsh. Simulating the power consumption of large-scale sensor network applications. In *Proc. of the 2nd ACM Conf. on Embedded Networked Sensor Systems (SenSys'04)*, Nov. 2004.

[51] E. G. Sirer, R. Grimm, A. J. Gregory, and B. N. Bershad. Design and implementation of a distributed virtual machine for networked computers. In *Proc. of Symposium on Operating Systems Principles*, pages 202–216, 1999.

[52] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Effi cient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.

[53] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. *In Proc. of the 1st USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04)*, Mar. 2004.

[54] P. Zhang, C. Sadler, S. Lyon, and M. Martonosi. Hardware design experiences in ZebraNet. In *Proc. of the 2nd ACM Intl. Conf. on Embedded Networked Sensor Systems (SenSys'04)*, Nov. 2004.