Nucleos: a Runtime System for Ultra-Compact Wireless Sensor Nodes

Jiwon Hahn
University of California, Irvine
Irvine, CA 92697-2625, USA
jiwon.hahn@gmail.com

Pai H. Chou
University of California, Irvine, CA USA and
National Tsing Hua University, Hsinchu, Taiwan
phchou@uci.edu

ABSTRACT

Nucleos is a new runtime system for ultra-lightweight embedded systems. Central to Nucleos is a dispatcher based on the concept of *recursive threaded code*, which enables layers of abstraction from the runtime system and interrupt handlers to application tasks to be composed in a structured, powerful way, all with minimal program code. When used in conjunction with models of computation with behavioral transparency such as synchronous dataflow (SDF), Nucleos can support efficient memory allocation policies for multiple communicating actors with minimal runtime overhead. This recursive structure also lends itself to in-field code update and dynamic execution. Nucleos's low runtime overhead and low RAM/ROM requirements enable it to run on compact platforms previously unsupported by the most popular sensor OSes while still providing high flexibility and composability. In some cases, applications running on Nucleos actually outperform hand-crafted code running without an OS, thanks to non-obvious memory optimizations enabled by the SDF model of computation.

Categories and Subject Descriptors

C.3 [**Special-purpose and application-based systems**]: Real-time and embedded systems; D.4.7 [**Operating Systems**]: Organization and Design

General Terms

Design, Performance

Keywords

Wireless sensor networks, Operating Systems, virtual machines, recursive threaded code

1. INTRODUCTION

Minimizing the footprint of runtime support has been the goal of many since the early days of wireless sensor networks (WSN). Most WSN platforms are based on low-power, resource-constrained microcontroller units (MCU) with limited memory and processing capabilities. Traditionally, many of these MCUs are programmed directly for the bare machine without any operating system support. This approach eliminates all overhead, but such programs tend to be fragile, difficult to modify and maintain, and not easy to port to another hardware platform. In response, several runtime systems for WSN platforms have been developed.

Many runtime systems for WSNs follow the approach taken by TinyOS [15]: instead of a separate operating system and application, the runtime support and application code are compiled together as a monolithic executable. The runtime support includes primarily device drivers for the hardware and dispatching of hardware/software events. Memory is usually allocated statically in the form of overlays, which could preclude re-entrance or certain nested calls. To enable dynamic code update or higher level execution, additional layers of code such as Deluge or Maté must be compiled in. While successful as de-facto standard for a wide range of WSN platforms, TinyOS still occupies a relatively large footprint. For instance, recent ultra-compact WSN platforms with the popular 8051, 8052 cores typically have 4K RAM shared between data and code, too small for TinyOS. Even though TinyOS targets MCUs with 64KB ROM or larger, it is very difficult to use it in conjunction with a ZigBee stack, which occupies 64KB to 100KB by itself. Therefore, the problem with minimizing the footprint of the runtime support is important for not only ultra-compact platforms but also full-fledged ones requiring industry-standard protocol stack support.

Our runtime support must have the following properties:

1. occupies a minimal RAM and ROM footprint,
2. supports composition of tasks, not just independent threads of execution,
3. supports dynamic, in-field loading and invocation of tasks,
4. incurs minimal runtime overhead.

To accomplish all of these goals, we propose Nucleos, a runtime system based on the idea of *recursive threaded code*. Threaded code is a classical technique for building interpreters by representing the virtual instructions as the addresses of the corresponding subroutines [5]. The dispatcher itself performs a double-indirect call with auto-postincrement on a pointer to a table of addresses of native routines called *actors*. This approach avoids the cost of a switch statement and the arbitrary byte-code encoding for these routines. The novelty here is our *recursive* use of threaded code. That is, the dispatcher itself, just like any other routine, is invoked recursively to run another threaded code program, which in turn can invoke the dispatcher recursively to run yet another program on top. This is very powerful, because the same, minimal-cost but general dispatching mechanism can be used to build up layers of abstraction in the runtime system and application without additional code.

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.
EMSOFT'10, October 24–29, 2010, Scottsdale, Arizona, USA.
Copyright 2010 ACM 978-1-60558-904-6/10/10 ...$10.00.

Table 1: Code and data sizes (KB) of runtime systems for wireless sensor platforms

Runtime	code	data	Runtime	code	data
Nucleos	1	0.128	SOS	31	2.79
Contiki	40	2.0	SwissQM	33	3.00
DVM	13	0.178	ASVM	39	3.02
MantisOS	14	0.5+0.1× thread	Maté	40	3.20
TinyOS	21	0.60	Agilla	42	3.60
t-kernel	28	>2.0			

Our philosophy is that a runtime system essentially consists of a dispatcher of runtime policies for task execution or services. The dispatching *mechanism* is essential and must run on the node, while the *policy* can be computed by either the node itself as an application task or assisted by the host. In either case, the policy is represented as or embodied in the form of a *script* with the associated memory allocation scheme.

We have written several sensor applications modeled as synchronous dataflow (SDF) processes to be dispatched by Nucleos. To enable execution on resource-constrained WSN platforms, we implement host-assisted scheduling and memory optimization of these SDF processes, and then dynamically load and run them on the ultra-compact, $1cm^3$ WSN platform named Eco. Experimental results show that not only is Nucleos much more compact in both RAM and ROM usage than the monolithic approach, but it also incurs much less runtime overhead than other schemes.

In fact, in some cases, applications running on Nucleos actually ran faster than hand-crafted native code without an OS. Our analysis attributes this to the synergy between automatic memory optimization and Nucleos's composition support, as enabled by the SDF model of computation (MoC). The optimization opportunities in the presence of concurrent composition are difficult for the program to identify and perform manually, but the use of SDF provides much more structure to automated optimization. The minimal memory footprint and process composability features make Nucleos a compelling runtime system for not only ultra-compact WSNs but also any systems with non-trivial computing or protocol stacks that strain their resource use.

This paper is organized as follows. Section 2 discusses the related work followed by a brief overview of Nucleos in Section 3. Section 4 – Section 6 presents the details of Nucleos in the aspects of application/scheduling, communication, and its recursive dispatcher. In Section 7, we present our implementation, then evaluate our work in Section 8. Finally, we conclude in Section 9.

2. RELATED WORK

In this section, we first survey runtime systems for WSN platforms and then discuss memory management approaches.

2.1 Runtime Systems for WSN Platforms

We classify runtime systems for WSN platforms into monolithic systems, modular systems, and virtual machines.

2.1.1 Monolithic Systems

A monolithic system is one where the application code and runtime support library are compiled and linked together as a single, hence monolithic executable. TinyOS [15] programming is done in the nesC language, which provides constructs for hardware and software events. BTnut extends the open source Ethernut Nut/OS [1] with scheduling and memory management, events, synchronization, streaming I/O, and device drivers. Both TinyOS and BTnut are statically configurable to include exactly those software routines needed. However, even the bare minimum versions of these systems are still larger than 4–5KB, and both are rigid unless additional library code is linked to support dynamic update.

2.1.2 Modular Systems

By modular systems, we mean that the runtime system is able to dynamically load and invoke software components. On the lighter side, Contiki supports protothreads, which can be viewed as event-driven stackless coroutines whose state is represented by a single byte per thread [10]. It organizes the application as loadable units and is configurable for network protocol stack support. Although RAM usage is typically low (2KB), the firmware size is larger (40KB). In BerthaOS, applications are organized as *pfrags*, which are dynamically loadable, self-contained, fixed-sized code fragments of up to 2 KBytes each, and they are executed in round-robin. We assume similarly constrained platforms and similar dynamic loading assisted by an IDE, but we support a more general script instead of the hardwired policy.

Other sensor OSs assume more resources and offer more support. Mantis is a preemptive, multithreaded OS that supports priority-based thread scheduling with aggressive power management [7]. It is posed as an alternative to TinyOS type of event-triggered execution. It can run on nodes with 4KB RAM, though the code size is at least 14KB, which fits in TinyOS-class platforms but is still too large for our platform. SOS [13] supports not only dynamic loading but also enforces type-checking mechanism and memory buffers for interprocess communication. Unfortunately, this incurs very high overhead of 21 cycles compared to 4 cycles for direct function call, and 12 cycles for system calls. Moreover, dynamic memory allocation takes 69 cycles, and posting and dispatching of a message can take 562 cycles. To lower this overhead dealing with untrusted code, the *t-kernel* [11] supports "naturalization," or load-time binary translation, to ensure the newly loaded code does not compromise the system. It also supports preemption with 16 priority levels and 64KB virtual memory over 4KB physical memory; at the same time its code size is 28KB, twice that of Mantis OS.

2.1.3 Virtual Machines

Virtual machines (VMs) have been proposed on top of existing OSs. Maté [18], which runs on top of TinyOS, interprets assembly-like virtual instructions that can include user-defined routines and supports event-triggering. ASVM [19] addresses limitations of Maté by adding concurrency, customizability, and support for several languages. DVM [4] is a dynamically extensible VM built on top of SOS, though the interpretation overhead is relatively high. Several Java OSs have been proposed for WSNs, including Squawk [20] for the Sun SPOT node and Contiki's JVM [9]. They can take advantage of standard Java development tools. However, the interpreter and libraries require at least 350 KB of memory. SwissQM [21] offers a byte-code interface with 59 instructions, 37 of which are identical to those in Java and is Turing-complete. SwissQM targets data acquisition applications with concurrency support and byte-code compaction, though the interpretation overhead compared to native is unknown. Beside Java, a subset of Tcl called tinyTcl (64 KB) has also been proposed as part of SensorWare [8] to support modular code update. It is designed for reprogramming high-end sensor systems (≥327 KB ROM). CVM (Contiki VM) [10] is designed by the Contiki group for comparing their modular OS with a VM. Both result in a small program size, but the runtime overhead is high. CVM reports 87× slowdown vs. native code.

2.2 Memory Usage

TinyOS and Protothreads must reserve all potential memory at

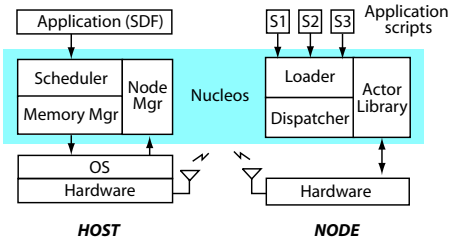


Figure 1: Nucleos framework

compile time. Most local variables must be declared as static. The problem is potentially poor memory utilization. A brute-force approach would be to let the programmer manually allocate memory based on knowledge of currently active tasks [14]. A common technique is *overlays*, meaning mapping different static locals to overlapping memory locations. However, it is error-prone and application dependent [11], as it prevents certain routines from calling others and rules out re-entrance or recursion. The t-kernel [11] supports general dynamic memory allocation. SOS supports allocation of communication buffers to enforce safe calls. These are all potentially costly operations. To enable efficient memory use on resource-constrained processors without memory management unit (MMU) hardware support, software-based virtual memory by data memory compression was proposed, which increases the amount of usable memory up to 50% [3].

A very different philosophy to OS-style concurrent programming is to use an explicit, higher-level model of computation (MoC). Lee [17] advocates abandoning threads, which not only pay a high price for the concurrency abstraction but their nondeterminism also complicates the software. Use of a MoC such as synchronous dataflow (SDF) [6, 22, 23] enables high-level, structured, formal composition, synthesis of provably correct programs, and optimization of resource usage. We exploit *behavioral transparency* properties in certain MoCs, including SDF, to determine and optimize the memory usage without having to simulate its execution. This is more general and formal than ad-hoc application-specific VMs.

3. NUCLEOS OVERVIEW

The name Nucleos actually refers to several different concepts: the *runtime framework*, the *runtime structure*, and the *kernel*. The runtime framework spans the nodes and the host. The host is a general-purpose computer that can send commands to the nodes, receive data, and also run the development tools. The nodes are the sensing systems with wireless links back to the host and is required to run only the bare minimum mechanism. The separation of policy and mechanism enables the policy layer to run on the host by default, or on the node if sufficient amount of resources are available. This section provides an overview of the *host-assisted* version, which imposes minimal resource requirements on the nodes. Each node is assumed to be connected to a base station in a star topology. In this paper, the term *node* indicates a system with limited resources, whereas a *host* may be either the central computer or a base station running an agent on behalf of the host. In either case, a host is a trusted entity and has access to all information about each node, from hardware configuration to firmware image. This enables the host to assist the node by processing the workload and to provide runtime interactivity.

Fig. 1 shows the Nucleos framework, which spans both the host and the node. On the host, the user loads the application currently modeled as a synchronous dataflow (SDF) graph. The host performs scheduling of SDF tasks and memory optimization, formats and transfers the code to the node. On the node side, the Nucleos kernel spawns three services upon startup to build up its runtime structure: communication to the host, script loader, and script dispatcher. These three components are necessary for loading and executing new applications on the fly. After receiving and loading the script segments, the dispatcher on the node starts executing the script. The actor code may have been just loaded dynamically or already installed in the actor library before deployment. At the heart of the Nucleos kernel is the dispatcher implemented in the form of a specialized threaded code engine. It invokes the actors in a given script with their corresponding parameters, including references to the communication buffer space.

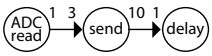


Figure 2: Example SDF Application

4. APPLICATION AND SCHEDULING

An application is represented as a script that is automatically generated from a task graph model. We use SDF for its ability to express data streaming and concurrent sensing behavior while enabling host-assisted memory optimization. Other MoCs with *behavioral transparency* can also be used [12]. The rest of this section summarizes the SDF model, implementation of actors, and representation of scripts.

4.1 Model of Computation: SDF

Synchronous dataflow (SDF) is a MoC for data-regular applications including many in DSP [6]. It models computation with a directed graph $G(V,E)$, where the vertices $v \in V$ represent *actors*, which are computing processes, and the edges $E \subseteq V \times V$ represent *channels*, which connect an output port of an actor to an input port of another actor. In general, the actors are *behaviorally transparent* in that when an actor is *fired* (i.e., executed for one iteration), it consumes and produces a known number of tokens as annotated on each of its I/O ports; in SDF, these numbers are constants. These fixed numbers are called *consume rate* and *produce rate*. Each *token* $tk \in E \times \mathbb{N}$ carries data and is uniquely identified by the channel and sequence number, which identifies token's creation order. In general, SDF graphs may be cyclic, though we use acyclic ones for the purpose of illustrating behavioral transparency without loss of generality.

Fig. 2 shows an example application modeled in SDF. The numbers show each actor's token consuming and producing rates. In this example, *ADCread* actor should be executed 3 times before *send* actor, and then *delay* actor should run 10 times. Although this graph yields only one possible schedule, different rates may yield many possible schedules. More details on SDF scheduling can be found in [12].

4.2 Actor Organization

The actor library contains a collection of compiled binary actors for composing an SDF application. These actors may be primitive, such as setting 0/1 values of a port, or may be nontrivial, such as encoding the sensor data. The level of abstraction is decided by the user.

To make actors reusable and composable, an implementation of an actor must refer to its input/output ports using pointers to the shared buffers rather than hardwired addresses. Instead of passing pointers to the actors (as function arguments), we implement

```
void actorname(void)
{
   PARAM_T repeat, wrptr;
   unsigned char data;

   repeat = popParam();
   wrptr = popParam();
   while (repeat--) {
      // create, process data here
      pushData(wrptr++, data)
      // write to data buffer
   }
}
```

Figure 3: Actor format

the actors to pro-actively fetch the buffer pointers as needed. This way, the dispatcher is kept very simple without having to handle the variable number of ports on each actor. Each actor may utilize five types of parameters including pointers to the buffers: *jump*, *repeat*, *read_ptr*, *write_ptr*, and *config*. *Jump* is the address to jump to within a hierarchical script. *Repeat* is the repeat factor, which defines how many times the corresponding actor is executed in series. When repeated, each invocation of actors accesses the buffer contiguously. The *repeat* parameter reduces the script size by collapsing multiple actor invocations (e.g., AAAA can be written as 4A) and is used in *single-appearance schedules* (SAS). The *read_ptr* and *write_ptr* parameters are the pointers to the buffer locations. Finally, *config* is for all other configuration data, such as the port value, power level, and the channel number. Each actor is written in a unified format as shown in Fig. 3. This format makes each actor's memory usage explicit, imposes a fixed number of input/output tokens, and allows the system to predict and compute the overall memory usage before run time.

4.3 Scripts

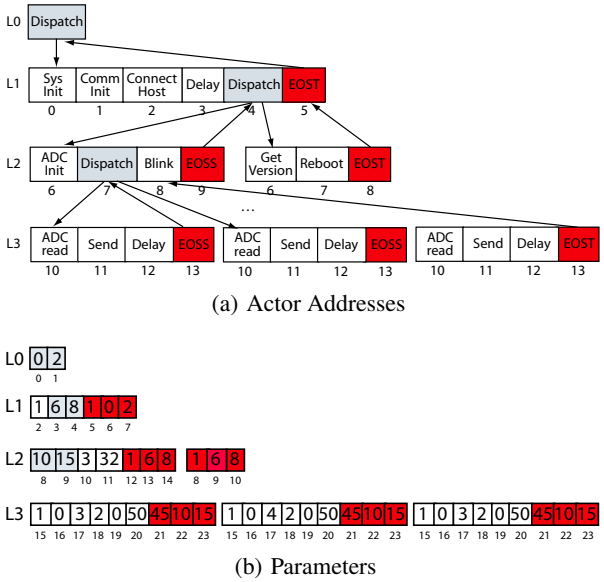
A script is a high-level program that defines of a sequence of actors to fire along with the parameters. A script can be written by a person textually or synthesized by a scheduler, and in either case it is compiled by our tool into a *threaded code* program. That is, the actors conceptually comprise the instructions in a virtual machine, and threaded code represents the program in terms of the actors' addresses (as opcodes) along with the parameters (as operands). This enables the dispatcher to be built efficiently, as explained in Section 6. Our tool lays out a script with all the actor addresses first, followed by all parameters, rather than interleaving them similar to assembly instructions (opcode, operands), as the former enables tighter packing.

Fig. 4 shows an example script. It is composed of multiple script segments, each shown as a group of blocks delimited by the shaded (red) ones. These scripts are dynamically composed and loaded to the node at runtime. More details on the highlighted actors and parameters and the script hierarchy of this example will be explained in Section 6.

4.4 Scheduling and Optimization

For an application modeled in SDF, the state of buffer requirements is completely determined by the firing sequence of the actors without requiring functional simulation of the actors. A variety of scheduling and buffer optimization algorithms can be used, and here we use one from our previous work [12]. Our approach is to first compute a schedule with low peak buffer demand and then compact the memory usage. Instead of dedicating buffers to each channel or actor statically, each actor is called with the addresses of the shared buffers with sufficient depth for consumption or production. We iterate through each step of the schedule and apply a buffer layout algorithm to find the most compact buffer map over time.

In addition to the shared buffer memory for inter-actor communication and each actor's own state, Nucleos also needs RAM for



(a) Actor Addresses

(b) Parameters

Figure 4: Script Example

storing scripts, i.e., the sequence of actor addresses and parameters. In a manner similar to the buffer optimization, script space can also be minimized by tracking the lifetime of the script segments. Whenever the execution of script segment is terminated, the script space is reclaimed and the new script overwrites the area.

5. COMMUNICATION

After an application is generated and optimized as a compact script, it is transferred and loaded to the node prior to the dispatch and actual execution. This section explains the built-in communication protocol between the host and the nodes, the packet format, and a novel implementation of the communication handler.

5.1 Operating Modes

The operating modes (or states) of the host thread and the node include *connect*, *load*, and *dispatch*. *Connect* is the initial phase of establishing a connection between a host and a node. *Load* is the phase of transferring and loading the application script. *Dispatch* is the execution phase of the script. Each mode binds to a specific RX parser similar to event handlers with incoming bytes as events. Each parser defines a valid range of expected input data and the corresponding action (i.e., data handling). The host and the nodes are synchronized by passing the state flags at the state transition points, and the received state flag triggers the mode transition and resets the RX parser accordingly.

5.2 Protocol

Fig. 5 shows the sequence diagram for the host/node communication protocol. The host first spawns a thread and waits until the node connects or times out and terminates. As soon as a node is powered on, it attempts to connect to the host by sending a CONNECT message to the host. Upon receiving the connection request from the node, the host enters LOAD mode to prepare the application script to load. The node waits until connection is confirmed by the host with a CONNECTED message and then sends a LOAD_SCRIPT to request the host to transfer the script. The host responds by sending the script to the node, waiting for the ack, and then enters *dispatch* mode. The node stores the received script

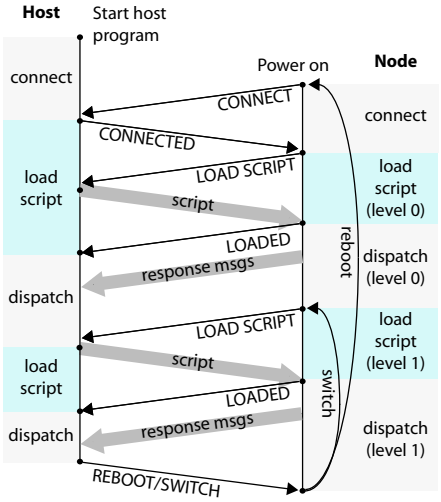


Figure 5: Protocol

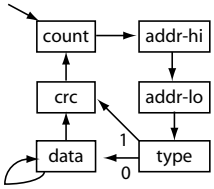


Figure 6: State diagram of the RX parser for Intel Hex format

at level 0 of the script hierarchy, acknowledges to the host with a LOADED message, and the host sends a REBOOT/SWITCH message to start the dispatcher to execute the script on the node. In *dispatch* mode, the node may need to send response messages back to the host for sampled data, query results, or node state. Anytime during the dispatch, the host can send a command for reboot or application switch.

5.3 Packet Format

The packet format consists of a command plus parameters. The command can be CONNECT, CONNECTED, LOAD_SCRIPT, SCRIPT, LOADED, RESPONSE, and BOOT/SWITCH, as shown in Fig. 5. The SCRIPT type contains code and must be handled in a special way. We use a binary version of the Intel hex format to represent the code, which can be either native code for the actors or the compiled scripts. A motivation of using such format is that the length of each data field can be adapted to the current buffer utilization. The Intel hex format consists of newline-separated lines of records, each of which consists of the fields *count*, *address*, *type*, *data*, and *crc*. *Count* indicates the length of data, *address* is the 2 byte address of the program memory (or buffer memory for the script) to store the data, *type* indicates whether the packet is the last one, *data* is the actual content to save, and *crc* is the error-detection checksum. Although the Intel hex by default is defined in human-readable ASCII text, we encode it in binary instead and achieve about 50% of the payload size.

5.4 Bufferless Packet Handling

The conventional way of handling data packets is to have the interrupt handler copy the data into a buffer and then perform the parsing outside the interrupt handler. While this minimizes the amount of time spent in interrupt handlers, this increases the buffering requirement. To address this problem, we structure the states of the parser's state machine (Fig. 6) as replaceable *plugins*, or functions that are called through a *function pointer* from inside the interrupt handler. To make a state transition, each plugin changes the plugin to that for the next state. This way, as the same interrupt handler gets invoked to handle each byte, it calls the plugins for the current state without having to buffer the bytes first.

Table 2: Buffer usage in each RX parsing state.

	count	addr-hi	addr-lo	type	data	crc
$r+0$	csum	-	-	-	-	$\rightarrow$
$r+1$	count	-	-	-	-	$\rightarrow$
$r+2$	c-var	-	-	-	$\rightarrow$	
$r+3$		addr-hi	-	-	-	$\rightarrow$
$r+4$			addr-lo	-	-	$\rightarrow$
$r+5$					data[0]	$\rightarrow$
...					data[...]	$\rightarrow$

```
// rx interrupt handler
void parser(void)
{
   unsigned char ch;
   ch = (unsigned char) rx_read();
   plugin(ch);
   // initially plugin = p_count
}
```

```
void p_count(unsigned char ch)
{
   // save checksum, c-var, ch
   // into Buffer[r+0 ... r+2].
   if (ch <= MAXDATASIZE)
      // within valid range
      plugin = p_addr_hi;
}
```

Figure 7: Implementation of bufferless packet handling

Fig. 7 shows parser code, where `parser()` serves as the interrupt handler for Rx. It calls the `rx_read` routine to read a byte from the communication port and passes it to the plugin parser, which is initially set to `p_count` corresponding to the "count" state in Fig. 6. To transition to the next state, `p_count` sets the plugin to `p_addr_hi`, which will be invoked by the interrupt handler upon receiving the next byte. Table 2 shows the memory usage during one round of the state diagram. In the table, dashes show occupied slots while arrows indicate that consumption of the token and thus freeing the slot. The routines can use a variable, named `r` in the leftmost column of the table, as the offset within the shared data buffer so that the host can determine the buffer usage map. The code size overhead of this plugin-style parser is 220 bytes. It shortens the response time and eliminates buffer locking. It also has the side benefit of consolidating the active intervals so that the MCU can have a longer idle interval for power management.

6. RECURSIVE DISPATCHER

A novel feature with Nucleos is the recursive dispatching structure to support formal MoCs. That is, the dispatcher is not only a bottom-level loop that invokes actors, but it can invoke itself, just like any other actor, to dispatch another script at a higher level. This is how we can use the minimal code to build up layers of abstraction in a consistent, efficient way. All of the same mechanisms including the memory assignment to the actors are reused at all levels; each script instance just needs its own memory space plus data. This section illustrates the script hierarchy with an example, the use of special actors to support this structure, the dispatcher's algorithm, and optimization of the dispatcher.

6.1 Illustration of Recursive Dispatch

To illustrate recursive dispatch, consider the example shown in Fig. 4 earlier. On power-up, the *dispatch* actor is run at level $L0$. It loads the script for the boot-up sequence at level $L1$, including

```
EOST()
1  P_w ← P_r ▷ to write back the repeating value
2  repeat ← POPPARAM()
3  if (repeat−−)
4     PUSHPARAM(repeat) ▷ save repeat
5     A_r ← POPPARAM() ▷ beginning of script
6     P_r ← POPPARAM()
7  else
8     TerminateScript ← 1
```
(a)

```
EOSS()
1  P_w ← P_r ▷ to write back the repeating value
2  repeat ← POPPARAM()
3  x ← POPPARAM() ▷ beginning of script
4  y ← POPPARAM()
5  if (repeat−−)
6     PUSHPARAM(repeat) ▷ save repeat
7     (A_r, P_r) ← (x, y) ▷ beginning of script
8  else
9     (A_w, P_w) ← (x, y) ▷ set write pointer
10    LOADSCRIPT()
11    (A_r, P_r) ← (x, y) ▷ set read pointer
```
(b)

```
EVENT()
1  if (F_0 == 1) ▷ check flag
2     Temp ← P_r ▷ save current pointer
3     P_r ← E_0 ▷ addr of event handling script
4     while 1
5        POPACTOR()() ▷ dispatch and execute
6        if TerminateScript ▷ end of script flag set
7           break
8  if (F_1 == 1)
9     ...
10 F ← [0, 0, ..., 0] ▷ unset flags
11 P_r ← Temp ▷ back to main thread
```
(c)

Figure 8: Pseudocode for (a) End-of-Script-and-Terminate (EOST) (b) End-of-Script-and-Switch (EOSS) (c) Event-Checking special actors.

system initialization, communication initialization, connection to the host, and delay; and spawns another instance of *dispatch* actor to execute the application at level *L*2.

Each instance of the dispatcher is maintained by two pointers: (a) the *program counter* to its script, and (b) reference to its parameter space. A script may have already been loaded or may need to be loaded into a *script segment*, and the specific instance of the dispatcher is initialized with the starting address of the script and reference to its parameter space. For ease of illustration, we show these addresses as a tuple of indices instead.

For instance, initially, the *L*0 base dispatcher has the default (actor address, param address) of *L*1, which is (0,2) before it starts executing in *L*1. The dispatcher at index 4 on *L*1 then is loaded with the tuple (6,8) before dispatching the script on level *L*2. The first segment on *L*2 performs ADC initialization, and then jumps to *L*3 (at (10,15)) to test ADC readings of different channels. The test applications to read each ADC channel are switched after a number of iterations (e.g., 45 in this example). When the execution of the last segment is over, the script is terminated, freeing up the memory used for both the script and data, and the script pointer returns back to *L*2. *L*2 script continues, and switches to the next segment, which is for version checking and reboot.

Although the regular sequence would be to terminate *L*2, roll back to *L*1 and eventually return to the base dispatcher at *L*0, this specific example ends with reboot, and directly restarts from *L*0; it is for testing the soft reboot function. As explained earlier, an actor fetches its own parameters sequentially. At the end of a script segment, the dispatcher either repeats the same script segment, switches to another script segment, or terminates and returns to the previous level. The following section explains these tasks in more details.

6.2 Special Actors for Dispatching Support

To keep the dispatcher's inner loop as simple as possible, we use special actors to handle conditional behavior and termination instead of testing these boundary conditions inside the dispatcher. The dispatcher actually does not know the length of the script and thus does not know when to stop the dispatch. Instead, we attach special actors called EOSS *(End of Script, Switch)* or EOST *(End of Script, Terminate)* at the end of the script segments. Figs. 8(a) and 8(b) show the algorithms for these actors.

To support repeating, the dispatcher first checks the current *repeat* value. If not zero, the value is decremented then pushed back to the script, and the script pointer (A_r, P_r) is reset to the beginning of the script. When the repeating value reaches zero, the script can either get replaced by a new script or just terminate. The EOST special actor sets the script termination flag to notify the dispatcher to stop. The EOSS special actor requests that new script replace the old script at the same script address, and resets the script pointer.

```
DISPATCHER()
1  x ← POPPARAM()                          ▷ actor to jump to...
2  y ← POPPARAM()                          ▷ parameter to jump to...
3  PUSHRET()                               ▷ save current pointers
4  Level++
5  (A_w, P_w) ← (x, y)                     ▷ set write ptr
6  LOADSCRIPT()
7  (A_r, P_r) ← (x, y)                     ▷ set read ptr
8  while 1
9     POPACTOR()()                         ▷ dispatch and execute
10    if TerminateScript                   ▷ end of script flag set
11       break
12 TerminateScript ← 0                     ▷ reset flag
13 Level−−
14 POPRET()                                ▷ reset pointers
```

Figure 9: Pseudocode for the Dispatcher.

To handle hardware interrupts, users may insert EVENT actor (Fig. 8(c)), which checks predefined flags (P_0, P_1, ...) for certain interrupts; interrupts are written to set flags only, and the handling routines are to be invoked shortly when event handling scripts are dispatched. Users should define the corresponding event handling scripts at the reserved script locations (E_0, E_1, ...). This enables flexible handling of external events with dynamically composable scripts.

6.3 Dispatching Algorithm

Fig. 9 shows the pseudocode of the dispatcher. It first pops parameters to get the target script address to jump to (lines 1–2). Then, the pointers (A_r and P_r) to the current script (READ) are pushed onto the return stack. After incrementing the level, the pointers (A_w and P_w) to the WRITE script are set to the target addresses (x, y) for where the script can be loaded (line 5). LOADSCRIPT is invoked to request and load the script from the host, and then the script pointers are set (line 7). The dispatcher enters an infinite loop to dispatch actors until a termination flag is set by an actor. The dispatcher resets the flag returns, pops the return address and returns to the previous script level.

6.4 Optimization of Dispatcher Core

The core optimization is critical to keeping the application per-

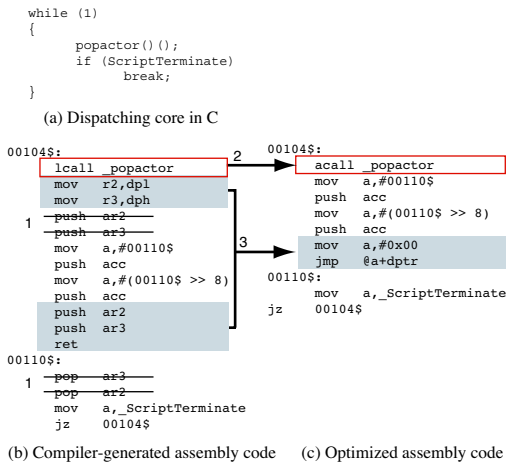


Figure 10: Code optimization at various levels for Nucleos kernel.

Table 3: Dispatching overhead of different core implementations

Implementations	code (bytes)	delay (cycles)
Switch/Case	99	151
Call-threaded	32	55
Direct-threaded (opt)	17	39

formance competitive. The most common and straightforward way to implement an interpreter is to use a switch-case statement to map each opcode to its corresponding primitive, but it incurs high overhead both in code size and execution delay. A more efficient way is to apply the idea of *threaded code* [5] by collecting all the incoming opcodes on a stack, then sequentially fetch and jump. A simple implementation in high level language such as C, looks like the following:

```
p = fetch_next(); // get the function pointer
(*p)(); // call the function pointed to by opcode
```

It fetches the pointer to the function that implements the opcode and invokes it. This implementation style is called *call threaded code* because the interpreter in fact *calls* the primitive with a subroutine-call instruction. In assembly generated from above code, the primitive address is pushed onto the stack then returns. Further optimization can be achieved by coding in assembly. This way, *direct threaded code* can be realized in its original form, i.e., by a *jump* instead of *call* instruction.

Fig. 10(a) shows the core of the dispatcher in C. The compiler generates the assembly code (Fig. 10(b)). We apply several levels of optimization to reduce the code as shown in Fig. 10(c). First, all redundant code is eliminated. Sometimes, reordering lines makes other code redundant. We also apply such reordering. Second, some opcodes are replaced with more efficient ones, by exploiting static knowledge (e.g., relative addresses of functions). Third, an alternative implementation is inserted. These methods are highlighted and numbered in the illustration of Fig. 10. This optimization is performed not only on the dispatching core but also throughout the kernel code. Table 3 shows the code size and delay of the different implementations – switch/case, call threaded code, and our optimized direct threaded code. Our optimized core is 17.2% of the code size and incurs 25.8% of the execution overhead of the straightforward switch/case implementation. The assembly optimization reduces the kernel size by 36.2% compared to the C implementation of the threaded code.

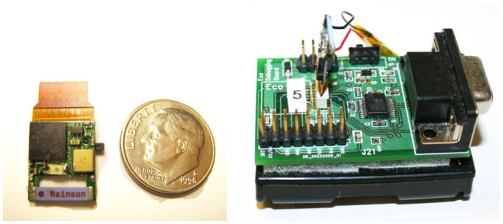


Figure 11: Eco node vs. a dime coin; Eco Debugging Board.

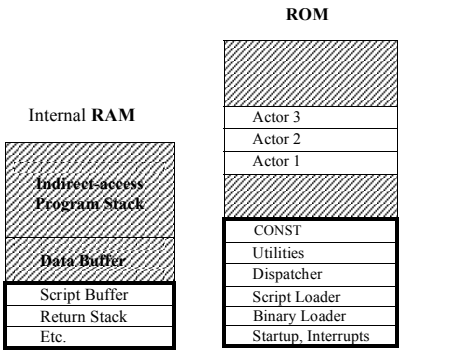


Figure 12: Memory structure of Nucleos on Eco node.

7. IMPLEMENTATION

We implemented Nucleos on Eco node v2.0 [24] shown in Fig. 11. The kernel implementations are written in a mix of C and assembly, where the assembly-optimized core is inserted as specially tagged functions and is compiled by SDCC [2]. In this section, we summarize the platform specification and the Nucleos memory partitions mapped to the Eco node's 8052 memory architecture. We also briefly describe *Ecosim* for simulation, followed by sample applications used in our experiments.

7.1 Platform

The MCU on the Eco node is the Nordic nRF24LE1. It is an integrated chip with an 8052-compatible core running at 16 MHz, a 2.4 GHz radio at 1 Mbps, a 9-channel ADC, 4 KB of on-chip RAM shared between program and data, and 256 bytes of internal RAM. On startup, the MCU loads its program code from the 4 KB external EEPROM. The MCU is connected to an integrated triaxial accelerometer and temperature sensor, a light sensor, and an LED on the same PCB. In our experiments, we mount an Eco node onto the Eco Debugging Board, which serves as a break-out board with a serial port adapter and provides an external power source to the Eco node.

7.2 Memory Structure

Fig. 12 shows the memory structure of Nucleos implemented on Eco. The memory reserved by the Nucleos kernel is framed in thick borders. As defined by 8052 architecture, the internal RAM of Eco is divided into 128 bytes of directly and indirectly addressable RAM (00-7FH) and 128 bytes of indirect-only RAM (80-FFH). The indirect-only portion is left available for the system stack starting at 80H. The higher part of the direct memory is allocated to our

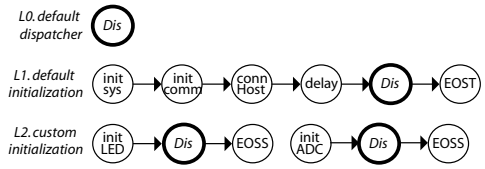
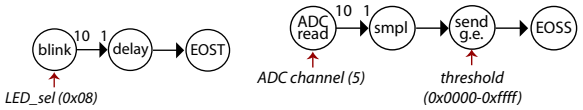
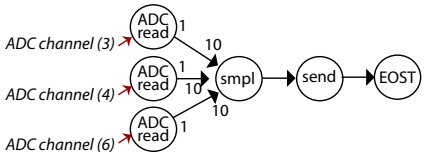


Figure 13: Initialization sequence in SDF hierarchy.



(a) App 1. Blinker (b) App 2. Light detector



(c) App 3. Motion tracker

Figure 14: Sample applications modeled in SDF.

shared data buffer. The RAM space reserved for the Nucleos kernel includes script buffer, return stack, global variables, and buffer pointers.

The Nucleos kernel resides at the bottom portion of the 4 KB ROM. It includes startup code and interrupts starting from address 0000H, and loaders for binary actors and scripts, the script dispatcher, and utilities used by actors, such as push/pop data. Constants include node's hardware and firmware versions, and predefined messages to the host. Although the regions in Fig. 12 are not shown in proportion, the Nucleos kernel consumes about 25% of the total 4 KB ROM, while the rest of the space remains available for the loadable actors.

7.3 Simulator

To help development and evaluation of the Eco platform, we developed *Ecosim* as a simulator for the 8052 core. It simulates the SDCC-generated assembly code. Written entirely in Python, *Ecosim* is cross platform and fast enough for compact codes. It is used to quickly obtain the code size and memory overhead of Nucleos components in terms of instruction cycles and bytes, respectively. We verified all simulated results by actual measurement.

7.4 Applications

Fig. 13 shows the SDF models for defining low-level components, starting from the default base dispatcher, the default initialization sequence, and the system initialization at one level above. The dispatchers are highlighted in bold, where each jumps to one level higher and returns after execution. $L0$ and $L1$ scripts are shared among different applications, whereas the system initialization can be replaced and customized to the specific needs of the application.

The sample SDF applications used in our experiments are shown in Fig. 14, where only non-uniform rates are tagged on each channel. Some of the actors are attached with an additional smaller (red) arrow for specifying the actor-specific configurable parameters, which are provided by the user. These parameters include selection of an LED, ADC channel, or setting a threshold value.

Fig. 14(a) shows the SDF graph for the blinker application, which keeps turning the LED on and off followed by a one-second delay (100 ms delay $\times$10). Fig. 14(b) shows the light detector application, where the ADC channel for the light sensor is read to retrieve the luminance reading, followed by sampling, and if the value is greater than or equal to a certain threshold, then it is sent to the host. Fig. 14(c) shows the motion tracker application, where the x, y, z acceleration values are continuously read on the corresponding ADC channels and sent to the host.

Table 4: Overhead of Nucleos kernel

Components	code (bytes)	delay (cycles)
Dispatch	82	39 per actor
Hierarchy Support	116	158
Ext. Memory Mgmt	16	24
Load Script	110	70+α per actor
Load Binary	580	690+α per actor
Total	904	N/A

8. EVALUATION

To evaluate Nucleos, we first analyze the overhead of the kernel in terms of code size and delay. Then, we show the memory consumption and performance of the sample applications running on Nucleos in comparison with the same applications written natively but in a structured way (e.g., using library modules). We also discuss the flexibility provided by Nucleos.

8.1 Kernel Overhead

The components of Nucleos include the dispatcher, hierarchical dispatching support, host-assisted memory management, and loader for scripts and binary. The dispatching mechanism entails fetching the script from the script buffer and the loop to invoke the actors. Recursive dispatching is supported by push/pop return stacks plus the end-of-script special actors. The loader handles the host-to-node communication protocol, parsing functions, and copying code to the script buffer and the EEPROM.

Table 4 shows a summary of the code size of each kernel component and the estimated delays. The total code size of the Nucleos kernel is 904 bytes. Delay of loading scripts and binary includes the communication delay α, protocol handling, and parsing. The dispatching of an actor in Nucleos costs 156 clock cycles (= 39 $\times$ 4), compared to 470 cycles for DVM [4] to fetch and decode an opcode, and 600 cycles in ASVM [19].

8.2 Memory Consumption

We compare memory consumption between applications running on Nucleos and native ones. The native implementations of the blinker, light detector, and motion tracker applications consume 470, 871, and 1003 bytes, respectively, including startup code, constants, library, and application. We divide memory sizes into application code and data sizes.

8.2.1 Application Size

The interface to the library can make a difference in the application size. The library routines for actors on Nucleos are designed to actively fetch parameters and data from the shared buffer, whereas those for native code are written to access parameters that are passed as function arguments. Nucleos actors are slightly larger in size due to extra code to fetch the parameters and the repeating factor. Table 5 shows the difference in the first corresponding

Table 5: Application size (bytes).

Program	Native			Nucleos		
	lib	app (binary)	total	lib	app (script)	total
Blinker	167	30	197	199	12	211
Light.	330	146	476	422	22	444
Motion.	353	249	602	360	41	401

Table 6: Data size

Program	Native	Nucleos	Savings
Blinker	0	0	0%
Light detector	28	22	21.4%
Motion tracker	33	26	21.2%

columns, ranging from 2.0% to 22%, which varies significantly according to the number of actors included. However, actors in Nucleos are *dynamically composable* with other actors.

When considering the entire executable, a Nucleos application can actually be significantly more compact than the corresponding native implementation written with a `main()` function, as shown in Table 5. Considering the total code size for an application, the more complex the application is, the more compact the Nucleos version is compared to the native. For instance, the Nucleos version of the motion tracker is 33% smaller than the native version.

8.2.2 *Data Size*

We compare the data memory consumption of Nucleos and native versions in Table 6. It shows that Nucleos is able to save up to 21.4% of data memory in applications that use more data buffers. The savings are due to scheduling of shared buffers based on data lifetime analysis. In contrast, native implementations must statically declare local buffers by the application programmer manually.

8.3 Application Performance

Table 7 shows the result of evaluating the application development cycle from compilation to execution. The compilation time is measured by the unix *time* command, which measures the elapsed time of compiling the source by SDCC when invoked from the command line prompt. The wall clock time is measured from the user's point of view. Nucleos applications are written as scripts that invoke existing actors and thus require no compilation or linking.

To measure the delay for loading applications with deterministic results while amplifying the effect of communication delays, we use a slower serial channel (19 Kbps) instead of wireless (1 Mbps). Latency due to loading native applications is measured by our custom binary transfer module, which transfers the binary code from the host to the node line by line and written to the EEPROM. The loading speed of Nucleos is measured by an on-chip timer. The timer values are read before and after the loadscript module, which starts by sending script request to the host, and finishes upon receiving the last byte of script. Since Nucleos scripts are much more compact and loaded directly to RAM, the loading delay appears to be orders of magnitude smaller than native, as shown in the second rows of Table 7. The execution delay is also measured by the on-chip timer. In both cases, the timer value is read in between executions.

Table 7: Time to compile, load, and execute applications.

Step	Blinker		Light detector		Motion tracker	
	Native	Nucleos	Native	Nucleos	Native	Nucleos
Compile	0.839 s	–	0.902 s	–	0.939 s	–
Load	29.5 s	0.202 s	35.7 s	0.671 s	41.5 s	0.369 s
Execute	1.63 s	1.79 s	5.1 ms	4.7 ms	51.0 ms	42.8 ms

The blinker application does not use any data, and therefore the Nucleos takes more time due to dispatch overhead. However, the other two applications use data buffers for reading and passing the sensor data between actors, and Nucleos benefits from the zero-copy mechanism. The speedup actually exceeds the dispatching overhead, and the total execution delay is even smaller than the native implementation. The speedup is amplified in the motion tracker application (delay reduced by 16%), which exhibits more usage of data buffer.

8.4 Flexibility

Nucleos offers flexibility with low overhead, partly due to the dynamically loadable scripts and modules, and partly due to dynamically defined buffer usage patterns, where buffer pointers are passed at runtime. This is enabled by having the actors conform to the design template (Section 4.2). The benefits of flexibility are explained as follows.

8.4.1 *Interactive Testing*

Interactivity provided by the dynamically loadable scripts proved useful for testing and debugging the sensor nodes. This is especially true for embedded systems that lack observability and controllability without simulators or JTAG or other external interface. In our anecdotal experience, we were able to interactively discover and fix a bug due to accessing the wrong I/O pin that was mistaken for a hardware component failure. Also useful is when fine-tuning delays. A common bug in MCU programs is caused by not waiting long enough for a slow device to finish its task. Scripting helps scanning and finding the proper delay parameter in the same manner the port value was scanned.

8.4.2 *In-Field Reconfiguration*

After deployment, the nodes may need to be reconfigured for various reasons, including fine-tuning the device parameters. For example, when the deployment site has strong RF interference, it may be desirable to boost the radio transmission power level or switch to a different frequency channel for more robust communication. Nucleos supports this type of in-field reconfiguration by allowing direct reload of a parameter; not even the entire script needs to be retransferred, thanks to the separate handling of actors and parameters within the same script. Although a system can be programmed in advance to handle such configurations, the size of such code can easily exceed that of Nucleos itself, and the generality that is rarely needed can become a burden especially for highly resource-constrained platforms.

8.4.3 *Software Update*

Software update may be done at two levels: scripts and actors. Scripts can be loaded with a modified sequence or parameters, and actor code can also be updated by writing into the EEPROM. Although technically possible to update the kernel this way, currently we write-protect the fixed-address region occupied by the Nucleos kernel. When actor code is modified, the linker can perform various optimizations including appending the modified binary to the end of the image or overwriting, based on the previous versions. Ongoing studies include efficient update of the actor library to minimize the update cost while retaining compact actor layout [16].

8.4.4 *Resource Management*

Due to the separation of policy and mechanism, the management

policy can be generated at runtime by either a remote system or on the same node itself. These policies are then represented as scripts or parameters for scripts. In the case of memory, the loaded policy controls the data buffer to be used in an efficient layout over time with minimal fragmentation. The actors can be viewed as memory manageable components, while the scripts can be viewed as plug-in management policies. The same concept can be applied to power and time. If applied, each actor instances may run the associated device at a certain power level or a certain CPU speed to reduce the overall energy consumption and increase the battery life of a node. Such power management or CPU clock management policies would be dynamically generated and plugged into the node at runtime. By giving users flexibility in controlling resource management with scripts, Nucleos is expected to be adaptable to a wide range of applications by effectively utilizing the platform's limited resources.

8.4.5 Limitations

Because the primary objective for Nucleos is to fit in very small platforms, the kernel does not provide memory protection, garbage collection, or fault handling. Most other compact runtime systems also lack similar features. Another limitation today is that we show examples in a restricted SDF model [12] for simplicity. However, we believe that this is largely orthogonal to Nucleos.

9. CONCLUSION

This paper presents a new runtime support system called Nucleos. Its use of recursive threaded code enables software to build up layers of abstraction using this self-similar mechanism. Its minimal memory footprint and low runtime overhead makes it applicable to ultra-compact WSN platforms. Moreover, even on larger platforms, there is a pressing need to support industry standard protocol stacks such as ZigBee, whose relatively large footprint of 64–100 Kbytes often leaves very little resource available for the application, let alone the OS. Nucleos can also fill this gap by supporting both vertical and horizontal composition of these software blocks dynamically. Our experience with Nucleos has shown that SDF represents a good match with many sensor applications by enabling structured modeling. More importantly, the high-level knowledge with SDF enables systematic, automatic optimization of memory buffers between concurrently actors. These optimizations are often too complex for humans to handle manually, and we have shown that the efficiency gained by optimized memory access dominates the small runtime overhead, enabling our code to outperform manually crafted code in several cases.

Acknowledgments

This work is sponsored by the National Science Foundation CAREER Grant CNS-0448668. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

10. REFERENCES

[1] Embedded ethernet. http://www.ethernut.de/.

[2] SDCC – Small Device C Compiler. http://sdcc.sourceforge.net/.

[3] L. S. Bai, L. Yang, and R. P. Dick. Automated compile-time and run-time techniques to increase usable memory in mmu-less embedded systems. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems (CASES'06)*, pages 125–135, October 2006.

[4] R. Balani, S. Han, R. Rengaswamy, I. Tsigkogiannis, and M. B. Srivastava. Multi-level software reconfiguration for sensor networks. In *EMSOFT'06*, pages 112–121, October 2006.

[5] J. R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973.

[6] S. Bhattacharyya, P. Murthy, and E. A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, Norwell MA, 1996.

[7] 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 & Applications (MONET'05)*, 10(4):563–579, August 2005.

[8] A. Boulis, S. Han, and M. B. Srivastava. Design and implementation of a framework for efficient and progr ammable sensor networks. In *Proceedings of the First International Conference on Mobile Systems, Applications, and Services (MobiSys'03)*, pages 187–200, May 2003.

[9] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt. Run-time dynamic linking for reprogramming wireless sensor networks. In *Proc. SenSys*, pages 15–28, November 2006.

[10] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proc. SenSys*, pages 20–42, November 2006.

[11] L. Gu and J. A. Stankovic. t-kernel: Providing reliable OS support for wireless sensor networks. In *Proc. SenSys*, pages 1–14, November 2006.

[12] J. Hahn and P. H. Chou. Buffer optimization and dispatching scheme for embedded systems with behavioral transparency. In *EMSOFT'07*, pages 94–103, October 2007.

[13] S. Han, R. Rengaswamy, R. S. Shea, E. Kohler, and M. B. Srivastava. A dynamic operating system for sensor nodes. In *Third International Conference on Mobile Systems, Applications and Services (Mobisys'05)*, pages 163–176, June 2005.

[14] T. He, S. Krishnamurthy, J. Stankovic, T. Abdelzaher, L. Luo, T. Yan, R. Stoleru, L. Gu, G. Zhou, J. Hui, and B. Krogh. VigilNet: An integrated sensor network system for energy efficient surveillance. *ACM Transactions on Sensor Networks (TOSN'06)*, 2:1–38, February 2006.

[15] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'00)*, pages 93–104, November 2000.

[16] J. Kim and P. H. Chou. Remote progressive firmware update for flash-based networked embedded systems. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, pages 407–412, San Francisco, CA, USA, August 2009.

[17] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[18] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. In *Proceedings of the 9th International Conference on Architect ural Support for Programming Languages and Operating Systems (ASPLOS'02)*, pages 85–95, October 2002.

[19] P. Levis, D. Gay, and D. Culler. Active sensor networks. In *Second USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2005)*, pages 343–356, May 2005.

[20] R. Mueller, G. Alonso, and D. Kossmann. Java^{TM} on the bare metal of wireless sensor devices – the squawk java virtual machine. In *Proceedings of the Second International Conference on Virtua l Execution Environments (VEE'06)*, pages 78–88, June 2006.

[21] R. Mueller, G. Alonso, and D. Kossmann. SwissQM: Next generation data processing in sensor networks. In *Proceedings of the Third Biennial Conference on Innovative Data Systems Research (CIDR'07)*, pages 1–9, January 2007.

[22] P. Murthy and S. Bhattacharyya. Shared memory implementations of synchronous dataflow specifications. In *Design, Automation and Test in Europe Conference and Exhibition (DATE'00)*, pages 404–410, November 2000.

[23] H. Oh and S. Ha. Fractional rate dataflow model for efficient code synthesis. *Journal of VLSI Signal Processing Systems*, 37(1):41–51, May 2004.

[24] C. Park and P. H. Chou. Eco: Ultra-wearable and expandable wireless sensor platform. In *Third International Workshop on Body Sensor Networks (BSN'06)*, April 2006.