CoMOS: An Operating System for Heterogeneous Multi-Processor Sensor Devices

Chih-Chieh Han*, Michel Goraczko†, Johannes Helander†, Jie Liu†, Nissanka B. Priyantha†, Feng Zhao†
*University of California, Los Angeles, simonhan@cs.ucla.edu
†Microsoft Research,{michelg,jvh,liuj,bodhip,zhao}@microsoft.com

***Abstract*— This paper presents the architectural design and implementation of CoMOS, a component messaging based operating system for mobile sensing and communication devices with multiple, heterogeneous processors. Potential applications of these devices include personal voice or video services, health monitoring, and environmental sensing. To enable timely processing of environmental or user events and energy-efficient operations, the system uses a stack-based preemption mechanism and supports task migration and fine-grained resource scheduling. At the center of the architecture is a processor-agnostic programming abstraction in which applications are specified as sets of asynchronous tasks interacting with messages. Tasks respond to messages, or events, and may produce new messages. Message handling can be preempted on stack enabling timely response to high priority events with a very small memory overhead. The tasks are mapped to processors at compile time and can be migrated from one processor to another at runtime, allowing dynamic power optimization and load balance. CoMOS has been implemented on mPlatform, a modular multi-board sensing device with multiple MSP430 and ARM7 processors and radios, and has been used to support the development of a real-time 4-channel sound source location (SSL) application on the mPlatform. We describe the experimental results quantifying the overhead of the messaging, migration, and other critical components of CoMOS, and the end-to-end performance evaluation of the SSL application.**

I. Introduction

The modular architecture of multiple modules interconnected via reconfigurable buses provides an attractive design choice for supporting heterogeneous and extensible mobile embedded computing [24], [8], [22]. In this architecture, each individual module can specialize in providing one of the sensing, actuation, processing, storage, and communication capabilities, or a combination of these. Consider a scenario in which a mobile wellness monitoring device continuously senses and logs a person's fitness signals. One module of the device could use a low-power microcontroller such as MSP430 with low sleep current and low wakeup cost for data sampling, simple processing tasks such as filtering, and storage control. Upon detection of a significant physiological condition such as a heart irregularity, the device could wake up a more powerful module equipped with a 32-bit microprocessor such as ARM7 to respond to this infrequent, but computationally more demanding diagnostic task without delay. At the same time, another module, which specializes in wireless communication with a low-power medium access control (MAC), could switch from a low-power wireless sniffing mode to an active transmission mode of sending the alert and data to a remote caregiver. Concurrently, a cell-phone module might also be requested to support real-time voice communication with the caregiver if necessary. The availability of multiple, heterogeneous processors allow the device to respond to different events in a *timely* and *energy-efficient* manner. It is also more *flexible* and potentially more robust to local module failure, as in such cases the processing job of the module can be delegated to other modules.

One approach to programming multi-processor sensor devices is to develop applications in pieces directly on top of each local operating system of individual modules. For example, the Intel Stargate[1] with a 32-bit PXA250 processor can be connected to Mica2 sensor node with an 8-bit ATMEL microcontroller to support sensing and communication. The PXA250 processor runs Linux, while the Mica2 sensor node runs TinyOS [16]. There are several issues with this approach. First, performance optimization becomes more involved. Consider a simple data aggregation from wireless radio on Mica2. A user has to decide *a priori* where the data aggregation service should be implemented. Choosing an optimal location is non-trivial when more than one application is running on the device. Second, different operating systems impose different programming models. For example, Linux follows a process model while TinyOS is based on an event-driven model. Integrating programs created with difference programming models can be tricky, especially when multiple concurrency models, like threads and interrupt handling, are involved. One way to deal with this problem is to view a multi-processor device as a distributed system and provide a middleware that translates system calls for one OS into another OS. For example, Emstar [12] provides a thin "stub" layer for TinyOS running on Linux capable device so that program written in TinyOS can talk to Emstar through the stub, which translates low-level drivers such as radio, timers and LEDs to Emstar system calls. However, moving a component from Emstar to TinyOS is still difficult. A separate stub layer for the moved component must be in place to translate function interface into messaging interface. While effective for a small number of modules as in the Starget/Mica2 case, this approach would be difficult to scale up when the number of modules increases.

This points to the need for a processor-agnostic programming model in which an application is developed independent of the individual processors and interconnects. It should be the job of the scheduler and OS to map and schedule an application onto the processors, meeting timing and power requirements. In this paper, we present the architecture and implementation of CoMOS, a component messaging operating system. An application in CoMOS is specified as a set of asynchronous tasks interacting with messages. The tasks are event triggered in that they respond to input messages (or events), process them, and may produce new messages. Message handling can be preempted, enabling timely response to high-priority events. To support preemptive execution on memory constrained low-end microcontrollers, we choose a

[1]http://www.xbow.com/Products/productsdetails.aspx?sid=65

stack-based preemption mechanism rather than threads.

The CoMOS architecture is partly motivated by the need for moving computational tasks around to achieve optimal energy utilization when the parameters of an application change or a new application arrives, which is important for battery powered devices. For example, in a sound-source localization (SSL) application, when the desired sampling block latency of an acoustic application changes from 200 ms to 250 ms, it is more energy efficient to run a 512-point FFT task on a MSP430 than an ARM7 [23]. A uniform messaging interface, enabled by an underlying high-speed TDMA based CPLD bus and a message-routing table, provides transparency to on-module or cross-module communication. Late-binding of tasks to available resources decouples the task specification from the task execution. Task assignment can happen at compile time, where a constrained optimization scheduler assign each task to a processor, considering both power and deadline requirements, based on detailed energy models for processors, buses, and radios. The tasks execution can then be monitored at run time, and tasks can be migrated from one module to another to achieve balanced load, improved responsiveness, or optimized power utilization.

CoMOS has been implemented on the multi-processor sensing device *mPlatform* with multiple embedded processors and radios and has supported the prototyping of a real-time acoustic SSL application. Our experiments and performance evaluation have shown that the messaging, stack-based preemption, and migration incurred a small, tolerable amount of latency and memory overhead while providing the desired capabilities, as demonstrated by the end-to-end performance of the SSL application. The *contributions* of CoMOS are:

- It introduces a processor-agnostic programming abstraction based on a message mediated asynchronous task model to support application development on a heterogeneous multi-module system.
- It supports run-time dynamic migration of tasks thus giving a multi-processor embedded platform the ability to finely manage power, balance loads and mitigate possible local failures.
- It uses an asymmetric stack-based preemption mechanism to improve the responsiveness to time-critical events while incurring a minimal memory overhead.

The rest of the paper is structured as follows. Section II reviews prior work. Section III describes the use of modular hardware designs for energy efficiency. Section IV introduces the overall architecture and design choices of CoMOS, and Section V describes its design and implementation. Section VI presents an evaluation of CoMOS with micro-benchmarks and end-to-end application performance. Section VII discusses limitations of and possible improvements to CoMOS.

II. Related Work

1) Programming heterogeneous multi-module systems: Componentization is a way to provide abstractions to program embedded systems. The interaction among components can be synchronous, such as method calls, or asynchronous, such as message passing. In sensor network programming models, TinyOS/nesC [11] provide layered component abstraction and synchronous communication between modules. TinyGALS [6] builds an asynchronous message passing model on top of nesC. However, TinyGALS does not provide prioritization of tasks, nor preemptive execution. Port-based objects [25] uses asynchronous components to achieve reconfigurability. But message ordering is not guaranteed in the communication.

Asynchronous models are also common in integrating heterogeneous systems. EmStar [12] integrates TinyOS with a stub layer, which converts low-level TinyOS drivers to message handlers. The asynchronous nature of driver-hardware interaction, method calls for resource requests to the driver and hardware interrupt for completion, makes the integration transparent to both systems. However, generalizing the stub approach to every component can be very complex.

2) Component migration and loading: A number of approaches implement component migration using virtual machines. SensorWare [4] provides a mobile agent environment where scripts written in Tcl can migrate from one sensor node to another and spawn new scripts, with some limitations in communication and restart. SensorWare provides system support for serializing the script and Tcl variables associated with the script; there is no need to migrate neither program stack nor routing table. Agilla [9] is a middleware that provides mobile agent environment on mote-class devices. It supports migration with a stack-based byte-code interpreter, in which each task has a stack, registers and a heap. When an agent migrates, Agilla serializes the stack, registers and heap. Although the interpreters in these approaches provide a uniform execution environment, script execution can have a significant computational overhead [18]. The expressiveness of the task is limited to the underlying virtual machine or agent system. In contrast, CoMOS is based on a cross-processor messaging model, which does not rely on a VM interpreter. Further, the programming language for CoMOS is C, which is commonly used in embedded system programming.

CoMOS draws upon prior work on application loading such as TinyOS [16], Contiki [7], Impala [19], and SOS [13]. All four systems provide loading applications compiled in native instructions with different granularity. TinyOS supports remote application installation by replacing entire program image including OS and application. Contiki allows loading an application compiled in the ELF format into the kernel. Impala allows multiple applications to be loaded, but only one application can execute at any time. SOS allows multiple applications to be loaded and executed concurrently. The granularity of application loading for CoMOS is similar to SOS in the sense that multiple applications can be loaded remotely and executed concurrently. Each application is represented as multiple binary images with application wiring, and the loading can be at individual component level.

MMLite [14] and its distributed extension, The Microsoft Invisible Computing Embedded Web Services [15], use the Common Object Model (COM) for component loading and migration. The latter uses an automatic marshaler that converts between messages and procedural execution, and uses web services for distributed real-time scheduling.

3) Task scheduling: Task or message scheduling is at the core of an operating system. TinyOS uses foreground and

background scheduling. Foreground processing is scheduled by hardware via interrupt and background processing is scheduled in FIFO order by TinyOS task scheduler. Contiki, like TinyOS, provides foreground/background scheduling except that interrupt handlers only register callbacks to be called as soon as the current background task has completed the execution. Traditional threaded preemption can be optionally enabled to provide real-time operation. SOS provides a priority based message queue. Message handling in SOS has non-preemptive run-to-completion semantic. CoMOS message scheduling incorporates stack preemption on top of the messaging system. The messages that have higher priority will preempt current task execution immediately. This allows long running tasks be preempted for system responsiveness.

Mantis [2] provides traditional preemptive multi-threading and POSIX operating system interface. Preemption in Mantis is symmetric in that scheduler can swap back a previously preempted task. CoMOS stack preemption mechanism is asymmetric. When a task is preempted, switch back to the original task is not possible until interrupting task has completed. However, CoMOS does not require pre-allocation of the task stack. This allows more efficient memory utilization.

There is a rich body of research on multiprocessor and distributed real-time scheduling [17], [21], [23]. The preemptive execution mechanism and the messaging interface implemented in CoMOS can support static scheduling frameworks, such as rate-monotonic scheduling, and some dynamic scheduling frameworks, such as earliest deadline first (EDF).

III. Modular Multi-Processor Platforms

The targets for CoMOS are modular hardware architectures that can provide a Lego-like, plug-and-play capability to put together a sensor network platform tailored to specific application/research requirements. In contrast to some embedded system platforms that use a single processor to manage multiple add-on boards, multi-processor platforms, especially heterogeneous multi-processor designs, have several advantages. For example, DSP and network processors can perform domain specific operations very efficiently. Using them alongside general purpose CPUs can significantly improve multimedia and digital communication application performance. Another advantage of heterogeneous multi-processor designs is the flexibility in power scheduling, which we elaborate below.

A. Heterogeneous multi-processor energy efficiency

High-end embedded processors generally can deliver more instruction cycles per unit energy than low-end microcontrollers, when running at a full speed. However, high-end embedded processors, unless completely shut down, also consume more power and need more energy to wake up from a standby or sleep mode. The difference in the power consumption in different modes and their transitions as well as the difference in the energy efficiency motivate us to use a dedicated low-end microcontroller to handle frequent yet simple tasks.

Many mobile and embedded applications exhibit strong variations in the application workload. The earlier example of wellness monitoring scenario may switch from a low-power monitoring mode to a high-fidelity processing mode that requires orders of magnitude more processing cycles. Timing requirements and energy consumption among these modes can be very different, pushing the limit of a single processor even with dynamic voltage and frequency scaling.

Generally speaking, a more powerful microprocessor is more energy efficient in that it can finish a given job faster and with a smaller energy consumption than a less powerful microprocessor. However, putting the processor into sleep and waking them up require more energy. This gives rise to non-trivial scheduling problems about which processor to use for what job [23]. For example, assume that two microprocessors p and q (with characteristics similar to ARM7 and MSP430) are considered for a non-splittable job of L millions of instructions by deadline D. Figure 1 illustrates which processor is most energy efficient to use depending on the relationship between L and D, while taking into consideration the computation throughput (MIPS), energy throughput (MIPJ), mode transition time, and mode transition energy cost. Appendix gives the details of the analysis. A more complete treatment of the problem should also take into account of splittable workload the dependencies among pieces of work, and the voltage/frequency scale of the microprocessors. The problem can be formulated as an integer linear programming problem [23], which is NP hard. Even with this simple example, we can see that when the system workload and real-time requirements vary over time, it is desirable to shift the workload from one processor to another to achieve optimal energy utilization.

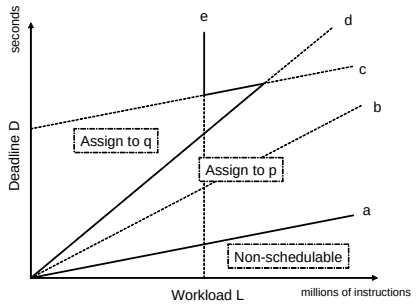

Fig. 1—Optimal assignment of non-splittable workload on two processors.

B. The mPlatform

The *mPlatform* is a realization of the multi-module hardware architecture [22]. It comprises a collection of stackable hardware modules that share a well defined common communication interface. Figure 2 shows a picture of a sample hardware setup.

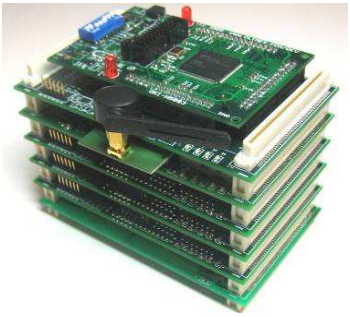

Fig. 2—A prototype of mPlatform.

Physically, each module consists of a circuit board and connectors that enable other modules to be plugged on both top and bottom of that module. Some of the *mPlatform* modules are general purpose processing boards with processors ranging from MSP430 to ARM7 and PXA270, while others are special purpose boards such as radio boards for wireless communication, sensor boards for sensing physical phenomena, and power boards for supplying power to a stack of modules. Each module, except for the power module, has a local processor. These local processors greatly enhance energy efficient handling of real-time events. The processor-per-module approach also allows a more flexible composition of different modules.

The *mPlatfom* uses a high speed parallel bus for inter-processor communication, with a maximum throughput of 64 MB/s. Each processor connects to this bus through a Complex Programmable Logic Device (CPLD), which implements a TDMA protocol for sharing the bus among modules. The bus protocol supports both unicast and broadcast message delivery. The CPLD bus employs hardware flow control to provide guaranteed message delivery. Due to the high throughput of the CPLD bus, for ARM and MSP430 class processors, usually the processors become the bottleneck during data transfers over the bus. The CPLD bus driver on each processor exposes a uniform `send_message()`, `receive_message()` interface for communicating over the bus. Since the bus is controlled by the CPLD, all but one of the processors can be powered of, and be powered back on by the remaining "active" processor by sending a message over the CPLD bus. This enables processors to be shut down and woken up as necessary to achieve the energy efficiency goals.

IV. Architectural Considerations

Heterogeneous multi-processor hardware design presents challenges and opportunities for operating systems. CoMOS is designed to meet the following goals:

- *Ease of programming and reconfiguration:* A wide range of processors, from low-power microcontrollers to higher-power 32-bit embedded processors, can coexist on the same platform and efficiently communicate with each other. To ease application development and to encourage reuse of software components, it is important to develop a simple, unified programming model that is transparent to the heterogeneity of the platform.
- *Real-time event handling:* Since a mobile platform is typically used in applications where it constantly interacts with the environment and user, the ability to promptly handle high-priority events is important.
- *Efficiency in resource utilization:* In many sensing and mobility applications devices are battery-powered. It is desirable to be able to shut down components when not in use and scale up or down operation voltage and/or frequency of the components according to changing task needs. In addition, applications should not have unbounded resource (e.g., memory) usage.

These goals lead us to use atomic software components and asynchronous message passing as the foundation of applications running on CoMOS. This architecture is motivated by the need to support a unified programming interface for multi-module devices and easy migration and loading of program components at runtime. A CoMOS application is written as a collection of loosely coupled software components, called *tasks*, which interacts by reacting to and generating *messages*. Some of these tasks can be executed at a number of different modules. The operating system takes the role of allocating sufficient resources to these tasks to meet application constraints.

A. Tasks

A task is an instance of a component that groups related functions together. A task is also assigned a priority, which determines the ordering of message processing within a module. Tasks have ports, which are their only communication interfaces. These include the communication with the operating system, such as initialization, time triggers, and error handling. The ports of a task are one-to-one bindings to specific processing functions, called *reactions* or *handlers*. A message delivered to a specific port is handled by the processing function assigned to the port.

Tasks are location transparent within a device. To achieve that, tasks have unique IDs across modules. When one task tries to send a message to another task, the message contains the destination task ID. It is the OS's responsibility to route the message to the destination. A key to the atomicity of tasks is that they maintain their own state, and there are no shared variables in the system. The state of a task is expected to be serialized and de-serialized if the task can be migrated.

B. Messaging

CoMOS applications are constructed by connecting input and output ports of tasks, which dictates how messages are routed in the system. This information can be stored in a message routing table that decouples task IDs from task locations. This level of indirection makes task migration relatively easy.

Messages carry their destination ports and the priorities of destination tasks. When an application spans across multiple modules, the bus is used to send messages from one module to another. This is handled by the OS without explicit knowledge to the application. Messages on the same module are sorted by their priority and then by their creation order in the OS scheduler. Higher priority messages are processed first. Messages in different modules are processes concurrently.

Message passing may create extra cost such as unnecessary content copying when sending to a local task. Thus practical systems usually convert some message passing into direct function calls when possible. This can either be carried out uniformly by the system or specifically by the applications. In the low-end modules, handling only a small number of messages and fairly simple applications, it is more memory efficient to leave the applications to handle message copying and marshaling directly. While it would be possible to have automatic marshaling of parameters on the more powerful modules, such as in the Invisible Computing framework [15], the current design choice favors the same programming interface on all the modules, thus using direct message processing across the modules.

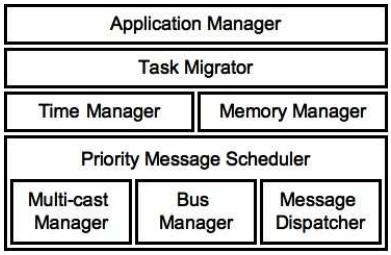

Fig. 3—CoMOS component break-down

C. Preemption model

The low-end modules are not limited just in computational capability but also in available memory. On microcontrollers without hardware memory management, thread stacks must be fully allocated in physical memory. Maintaining per-thread stack is very expensive considering the amount of available memory on low-end microcontrollers. Single thread execution, on the other hand, requires little stack space. But the run-to-completion execution semantics limits the fast response to high-priority events. An asymmetric preemptive threading model is proposed in [1]. This model allows interrupting the current computation, due to either I/O interrupt or higher-priority message processing, with another process on the same stack. While this allows the yielding of computation resources from low-priority tasks to high-priority tasks, it limits how priorities can be dynamically changed at run time, since switching back to the original computation is not possible until the intruding process has completed. Nevertheless this is sufficient to support many real-time requirements, while using memory extremely efficiently.

D. Migration

Task migration may appear easy in a component-oriented message passing system, since it is primarily an act of serializing states and changing message routing table. However, challenges arise when we must maintain the continuity of task execution. To ensure the transparency to high-level application, task migration has to be an atomic operation, that is all pending messages at the start of migration and all messages sent to the task during its migration must be received by the migrated task and in the correct order.

A task migration protocol is implemented in CoMOS, which coordinate among multiple modules to achieve both the atomicity of task migration and consistency of messages. By atomic, we mean that the task will either be at the destination module when migration has succeeded or at the source module when migration has failed but on not both modules. The details of the protocol are given in section V-D.

V. System Design and Implementation

In this section, we present the design and implementation of the CoMOS components to support the architectural requirements. Figure 3 shows the component break-down of current CoMOS implementation. All components have one instance running on each module of a multi-module sensor node.

Input Port	Condition
PORT_INIT	triggered when the task is created
PORT_FINAL	triggered when the task is killed
PORT_SERIALIZE	request state serialization for migration
PORT_DESERIALIZE	triggered when migration has completed
PORT_TIMEOUT[0..3]	triggered when TIMER_[0..3] has fired

TABLE I—Pre-defined input ports and their associated conditions.

A. CoMOS Tasks

1) Task Specification: A Task in CoMOS is described by the properties defined in `task_desc` structure. The values of these properties are used to instantiate a given task.

```
1 struct task_desc {
      program_id_t          prog_id;
3     pid_t                 pid;
      priority_t            priority;
5     port_range_t          num_ports;
      board_address_t       init_location;
7     size_t                init_data_size;
      size_t                init_data_offset;
9     task_desc_flag_t      flag;
  }
```

Here *prog_id* is the ID of the code image that implements the task. A Task has a unique ID among all modules on the sensor node; *pid* is the unique task identifier; *priority* is the priority assignment of the task. Each task has several output ports send out messages; *num_ports* is the number of output ports for the task; *init_location* specifies the initial module location for the task. A task may use a set parameters when it initializes (similar to the parameter list used in a C++ constructor); *init_data_size* and *init_data_offset* specifies the size and the location of an initialization string which is in fact a serialized version of these parameters. The *flag* is the bitwise OR of a set of task options such as `COMOS_TASK_MOVABLE`.

A task has a set of pre-defined input ports to receive messages generated by CoMOS. Some of them are system ports, as shown in Table I, others are application ports named as `PORT_0` through `PORT_MAX`. The current implementation supports a maximum of 128 input ports.

Listing 1 shows the execution part of a ”Hello World” task in CoMOS. The task creates a periodic timer that fires every 0.125 seconds after the task initialization (line 6 - 10). When requested time interval has passed, a message to port `PORT_TIMEOUT0` is generated for `hello_task`. CoMOS automatically associates `TIMER_0` with `PORT_TIMEOUT0`. The message handler then sends out “Hello World” message to output port `PORT_0` (line 14 - 19). Message handling in a task follows the run-to-completion semantic. That is the implementation of a task does not need to handle concurrency and synchronization explicitly. However, the execution of a task can be preempted when there are certain messages with higher priorities.

```
  ret_t hello_task( void *state, msg_t *msg )
2 {
      switch( msg->port ) {
4         //
          // First Message for the task
6         case PORT_INIT:
              timer_start(
8                 TIMER_0,
                  TIMEOUT_SECOND / 8,
10                TIMEOUT_SECOND / 8);
              break;
12        //
          // The Message every time timer fires
14        case PORT_TIMEOUT0:
              sched_send_msg(
16                "Hello World",
                  sizeof("Hello World"),
18                0,
                  PORT_0 );
```

```
20              break;
        }
22      return OK;
    }
```

Listing 1—A CoMOS task that sends out "Hello World" every 0.125 seconds

—

2) Task Creation: A CoMOS task is created by passing a task description to the message scheduler. Message scheduler, upon receiving task creation request, first calls the `sched_task_create()` with 5 parameters.

```
1 ret_t sched_task_create(
      task_t      task,
3     pid_t       pid,
      priority_t  p,
5     portlist_t* pl,
      port_range_t  num_ports );
```

First parameter, *task*, has following prototype.

```
ret_t task_name( void *state, msg_t *msg )
```

It indicates to the scheduler which massage handler to invoke when receiving a message for this task.

Second parameter, *pid*, is the unique identifier for this task. The pid should be unique among all modules on the sensor node. ID generation is not handled by `sched_task_create()` interface because CoMOS allows resource to be identified with ID. In section V-A.3, we describe how ID is generated when we describe application creation. Third parameter is the priority of the task to be created. Task priority assignment is also deferred to high-level. For example, our back-end tool assigns task priority according to global timing requirement among multiple applications. Fourth and fifth parameter are the *output* port table and the size of the table respectively. The output port table is used as a routing table to route messages from the task to the intended destinations. The level of indirection provided by the output port table allows us to separate the definition of message destinations from the task implementation. Apart from the output port table, `sched_task_create()` allocates and initializes the task control block (TCB) also. The TCB is used throughout the system to identify current executing task and speed up message scheduling and delivery. Creating a task does not immediately initialize it. Task initialization is implemented by a `PORT_INIT` message generated by the scheduler's `sched_task_init()` interface. This allows the tasks in an application to be created without worrying about missing messages. When initializing an application graph, *application manager* simply creates all the tasks and then issues `sched_task_init()` call to the message scheduler.

The C prototype of `sched_task_init()` is shown.

```
1 ret_t sched_task_init(
      pid_t       pid,
3     void*       init_data,
      size_t      init_data_size,
5     msg_flag_t init_data_flag );
```

In addition to generating `PORT_INIT` message, `sched_task_init()` interface allows the initialization string, *init_data* to be included for parameterized task initialization.

3) Application Creation: An application is composed of task images (the implementation in terms of processor specific instructions), task dependencies (output port routing tables and multi-cast group information), and initial conditions (task initial location and task initialization strings). An application description has six sections (refer to the line number in Listing 2 for concrete example).

- Application Description Header (line 13-14) specifies the number of tasks and multicast groups used.
- The Task Descriptions section (line 16 - 32) describes the properties of all the tasks in the application such as the task ID relative to this application, the priority of the task, and the number of output ports, as shown in `task_desc`.
- The implementation of the task is described as an ID to the program (line 18 and 27). The actual implementation of the task is defined separately as discussed in the previous section. This makes it easy to support different object formats such as ELF and COFF.
- Mcast Group Descriptions (line 34-35) provides the information regarding multi-cast group such as multicast group ID and number of members in the group.
- The routing tables (line 37-44) describe the destination task ID (relative to the application) and input port.
- Finally, Task Init Strings (line 46) includes the initialization strings for each task if not NULL.

Currently, application writers manually construct the application description as shown in listing 2.

```
1  static struct hello_world_app_desc {
       comos_app_desc             hdr;
3      comos_task_desc            task_desc[2];
       comos_mcast_desc           mcast_desc[1];
5      portlist_t                 task0_tbl[1];
       portlist_t                 task1_tbl[1];
7      portlist_t                 mcast1_tbl[1];
       task0_init_string_t        task0_init;
9  } app_desc;

11 comos_app_desc* get_app_desc( size_t *size )
   {
13     app_desc.hdr.num_tasks  = 2;
       app_desc.hdr.num_groups = 1;
15
       app_desc.task_desc[0].num_ports        = 1;
17     app_desc.task_desc[0].pid              = 0;
       app_desc.task_desc[0].prog_id          = TIMER_TASK_PROG_ID;
19     app_desc.task_desc[0].flag             = COMOS_TASK_MOVABLE;
       app_desc.task_desc[0].init_location    = BOARD_STARTING_ADDR;
21     app_desc.task_desc[0].priority         = 1;
       app_desc.task_desc[0].init_data_size   = sizeof(task0_init_string_t);
23     app_desc.task_desc[0].init_data_offset = offsetof(app_desc, task0_init);

25     app_desc.task_desc[1].num_ports        = 0;
       app_desc.task_desc[1].pid              = 1;
27     app_desc.task_desc[1].prog_id          = HELLO_WORLD_PROG_ID;
       app_desc.task_desc[1].flag             = COMOS_TASK_MOVABLE;
29     app_desc.task_desc[1].init_location    = BOARD_STARTING_ADDR + 1;
       app_desc.task_desc[1].priority         = 1;
31     app_desc.task_desc[1].init_data_size   = 0;
       app_desc.task_desc[1].init_data_offset = 0;
33
       app_desc.mcast_desc[0].group_id        = GROUP0;
35     app_desc.mcast_desc[0].num_entries     = 1;

37     app_desc.task0_tbl[0].dst_pid          = MULTICAST_PID;
       app_desc.task0_tbl[0].dst_inp          = GROUP0;
39
       app_desc.task0_tbl[0].dst_pid          = RADIO_PID;
41     app_desc.task0_tbl[0].dst_inp          = PORT0;

43     app_desc.mcast1_tbl[0].dst_pid         = 1;
       app_desc.mcast1_tbl[0].dst_inp         = PORT0;
45
       app_desc.task0_init.timer_period   = TIMEOUT_SECOND / 8;
47
       *size = sizeof(app_desc);
49     return (comos_app_desc*) app_desc;
   }
```

Listing 2—A CoMOS Hello World Application

—

Listing 2 is a CoMOS application that sends out "Hello World" every 0.125 seconds. The application is implemented with one timer task and one hello world task. Timer task is initialized with `timer_period` of `TIMEOUT_SECOND/8` seconds. When timeout message arrives at timer task, a multicast message is sent to `GROUP0`, which is wired to hello world

task's PORT0. When the message arrives at hello world task, a "Hello World" message is sent to RADIO_PID's PORT0.

In order for message passing and migration to work, a set of message routing tables are maintained at each module (these include the output port tables of each local task). Each routing table has two levels of information of the destination tasks: (1) the current task to module assignment, and (2) the input ports within a task. This two-level routing table simplifies task migration since the task is the unit of migration. A task is not specific to a particular architecture. The routing table allows late binding of tasks to modules, although some specific task numbers may be handled as constants by the routing table. One such specific task is a special control task that represents the module itself and facilitates migration requests and similar system functionality.

CoMOS supports creating applications dynamically through a special task called *application manager*. Like all tasks, the application manager has ports which can be connected to other messaging interface such as radio or serial port links.

When the *application manager* receives the application description, it first patches all IDs including task IDs and multi-cast group IDs. This is because all these IDs are described relatively to the application unless the ID falls into the range of global resource ID. The application manager then broadcasts patched application description to all other modules and parses the application description locally. For each task that has initial location local to the module, `sched_task_create()` is issued along with necessary information. Otherwise, `sched_task_create_remote()` is issued to inform the message scheduler the existence of this task on the remote module. For each multi-cast group, the members that are local to the module are added to the multi-cast table. Finally, *application manager* initializes each task that is local to the module with initialization string.

To avoid consistency problems during the creation of applications, the *application manager* is running at the highest priority in the system. Further, application is created in two stages: instantiation and initialization. Message scheduler buffers all messages for the task if the task has not yet initialized. These buffered messages will be dispatched once the task has handled the PORT_INIT message.

In application instantiation, `sched_task_create()` is used to to allocate necessary memory and create TCBs. In application initialization, `sched_task_init()` is used to create message to PORT_INIT.

B. Messaging

Tasks communicate only through messages. They provide a location transparent abstraction for tasks running on different modules. Messaging is also a key to achieve prioritization among local task execution.

1) Local Message Scheduling: CoMOS schedules messages according to the priorities of message destinations. The CoMOS message scheduler is invoked every time there is a new message, called a *preemption point*. For example, line 15 of listing 1 is a preemption point, which CoMOS message scheduler will be invoked. Every time CoMOS message scheduler is invoked, it checks whether the priority of the message destination is equal or higher than the sender. If so, the message is scheduled for immediate execution (i.e. preemption). Instead of normal preemption mechanism that saves registers and stack content, CoMOS preempts task on the global stack. Stack preemption is motivated by Stack-based Resource Policy[1]. That is, when the preemption does happen, CoMOS fetches the function pointer associated with the destination of the message and executes it immediately. This way, there is only one global execution stack instead of per-task private stack. The limitation of the stack-based preemption is that once the task is preempted, the scheduler cannot swap back the task until the intruding task finishes. When the task starts handling the message CoMOS sets a flag to guard against the swapping-back situation, which may be caused by the arrival of a message to a task that is already on the stack. When this flag is set, CoMOS queues all the messages for that task.

When the priority of message is lower than current executing task, the message is queued in the scheduler for later execution. The message queue is sorted according to destination priorities. When the current task has finished, message scheduler checks the priority on top of the execution stack against the head of the message queue. If the priority of the head of the message queue is higher, message scheduler dispatches the message. Otherwise, message scheduler simply does nothing and let the task on the top of the stack resume execution.

CoMOS handles the messages from the hardware interrupts differently. When a message is sent from the hardware interrupt, CoMOS only preempt the current executing task when the priority of the message is strictly greater. This is to preserve the execution order. The current task execution, resulted from a previous interrupt, either directly or indirectly, should take higher precedence.

2) Message Routing: Local message routing is implemented using message routing tables. CoMOS enables the transparency of messaging by having an output port routing table for each task. Output routing table maps output port to <`destination task ID, input port`> pair. This routing table is stored in the RAM and thus can be changed.

When a message is sent to a task on the remote module, the message scheduler forwards the message to the *bus manager*. The message scheduler makes this decision by finding the location of the message destination.

To enable low execution cost on discovering destination module, message scheduler stores the TCB of message destination along with each entry of routing table. For tasks that are remote to the current module, message scheduler creates a remote TCB that includes Task ID, priority of the task, and the location in terms of module address. When the message is sent to a particular port, the message scheduler looks up the destination module address in the TCB. If the destination module address is not the same as current module, the message scheduler forwards the message to the bus manager. The bus manager, if it is not currently sending another message, starts sending the message while blocking the current execution. We use this implementation because of the high speed CPLD bus. Using interrupt driven message transfer will waste CPU

cycles on interrupt dispatch for each byte. On the other hand, if the bus is busy sending messages, the new message will be buffered according to the priority of the destination task.

CoMOS supports multicast messages though a virtual task. Virtual task is a special task with special task ID but without the TCB. The virtual task for multicast messages has ID equal to `MULTICAST_PID`. When message scheduler sees the destination task ID equal to `MULTICAST_PID`, it forwards the message to a *multicast manager*. A multicast manager, upon receiving the message from message scheduler, first sends out the message over the bus.

Each multicast manager maintains the multicast group membership information of the tasks local to its own module. When a multicast manager receives a multicast message over the bus, or when a multicast manager finishes sending a multicast message over the bus, based on the multicast group ID, the multicast manager fetches the list of local tasks that belong to this group ID. It then duplicates the message and sends to each destination task according to the task priority.

We use this implementation because maintaining only the local multicast membership information can result in a significant memory saving, compared to maintaining the global membership at each multicast manager.

Consider our *task migrator* implementation. For a six module sensor node, there is a total of six *task migrator* instances: one instance for each module. The *task migrator* uses multicast messages to implement reliable task migration. If the *multi-cast manager* records only the task migrator instance local to the module, we save memory since we do not record the other five instances. In fact, the *task migrator* uses two multi-cast groups. This saves a total of 10 routing entries in the multi-cast table. Further, a smaller multi-cast table allows faster message delivery because there is no need to filter out unwanted entries. Obviously, when the multi-cast message is only for the local module, the energy for waking up bus and sending over the bus is wasted; however, this is a rare case.

C. Resource Management

Memories and timers are shared resources that must be carefully managed in resource constrained sensor nodes. CoMOS assumes that the sensor node is a multi-machine architecture in that every processor has dedicated memory and common peripherals such as hardware counters. CoMOS allocates such generic resource at the local processor where the task is residing at the time of request. When migration is requested, these generic resources are serialized in a processor independent manner and de-serialized on the destination module. CoMOS manages two hardware resources for the tasks on each module. The *memory manager* manages the heap memory, and the *time manager* provides a fixed number of virtual timers to each task. Resources such as flash devices for storage, I2C for communication, and additional hardware counters are assumed to be controlled by tasks. Since these hardware resources are physically bound to a specific module, the tasks that control these resources cannot be migrated. Such tasks have their `COMOS_TASK_MOVABLE` flag cleared to indicate this.

1) Memory Manager: When a task is migrated in or out, the memory for the task specific state must be allocated and de-allocated accordingly. Therefore, dynamic memory management not only makes programming easy, but also serves as an essential service for correct task migration. The key design decision then becomes whether the service should be exposed to the tasks. Static memory makes programming easy and free of memory leaks, while dynamic memory enables temporal sharing of memory. Dynamic memory was chosen also because it makes moving messages easy. Tracking memory ownership eases memory leak debugging and enables garbage collection.

The memory attached to the message is only copied when the task explicitly asks to do so. Task uses the following interface for taking the payload the message.

```
void *sched_take_msg( msg_t *msg )
```

The memory manager, upon receiving this call, first checks whether the memory is dynamically allocated. If so, the memory manager simply returns the memory back to the caller. If not, the memory manager dynamically allocates memory on behalf of the caller and does a deep copy of the message payload. This enables low latency messaging when both the sender and the receiver are on the same module. Memory is not shared among multiple tasks, which would otherwise force us to provide distributed shared memory in CoMOS and complicate the design.

Processors on the current mPlatform do not have memory management units (MMU), and the standard C library does not provide ownership information in the `malloc()` implementation. Therefore, we implemented memory management with ownership tracking that has the same semantics as the standard C library. To reduce external fragmentation, we chose an address-ordered first fit algorithm for memory allocation. To efficiently manage small metadata, such as timer control blocks, task control blocks, and routing entries, we implemented a slab allocator, which is commonly used to avoid internal fragmentations of OS metadata [3].

The memory interface is identical to libC. We repeat here for completeness.

```
1 void*    mem_alloc  ( size_t size );
  void*    mem_realloc   ( void *mem, size_t new_size );
3 void     mem_free   ( void *mem );
```

2) Time Manager: The *time manager* manages virtual timers for each task. In our current implementation, each task can have up to four timers. The time manager provides these virtual timers via one hardware counter on the local processor. The software timer is not ticked regularly from hardware; instead, a delta timer is implemented to avoid unnecessary CPU cycles associated with timer interrupt handling.

Furthermore, in order to reduce extra load on the bus, each processor maintains its local delta timer based on its own hardware clock. However, the problem of not using a single clock is that the global notion of time needs to be handled separately. In a long running system, each module would eventually have different notion of time because local clock crystals drift. To solve this problem, CoMOS uses time synchronization to ensure precise timing. Time synchronization is implemented as a special virtual task, which has ID `TIMESYNC_PID`. Messages sent to this task will be broadcast on the CPLD bus. The sender side CPLD driver recognizes this

special task ID and timestamps the messages with the local clock counter. When the destination side CPLD driver receives this special message, it compares the timestamp with its local clock. The difference of the two clocks is maintained to derive the global clock. The 32.768 KHz Real Time Clock (RTC) used in mPlatform has +/-20 PPM (parts per million) accuracy, which corresponds to a worse case of 40 PPM (a drift of 40 μs per second) frequency error between any two modules. The RTC resolution of a tick is 30.5 μs (1/32768 seconds). With 1Hz time synchronization frequency, we can achieve a global clock synchronization within +/-9.5 μs. Because tasks can send messages to `TIMESYNC_PID`, it is easy to modify the time synchronization frequency at runtime. For example, with low PPM clock crystals, time synchronization frequency can be reduced. Further, for better bus utilization, a task can implement adaptive time synchronization algorithms (e.g. [10]).

D. Task Migration

One unique service in CoMOS is the *task migrator* (TM), which ensures an atomic task migration. The task migrator provides atomic guarantee via following migration protocol.

1) Task migration REQUEST can be generated by either a task or a TM based on runtime resource utilization and application specific migration policies.
2) A REQUEST is first forwarded to the module where the task to be migrated resides. We call this module the *source module*.
3) Upon receiving the REQUEST at the source module, the source TM multicasts a FREEZE message with the task ID to be migrated and then issues a serialization command to the message scheduler, which will start the serialization of the task resources and will block all messages for this task.
4) Upon receiving the FREEZE message, other TMs issue a freeze command to their local message schedulers, which will block all messages sent toward this task. Then the TMs unicast reply messages back to the source TM.
5) When the source message scheduler has serialized all resources and the source TM has received all reply messages from other TMs, the source TM sends serialized resources to the destination module where the migrated task will be resumed.
6) Upon receiving the serialized resources at the destination TM, a de-serialize command with the serialized resources is sent to the destination message scheduler. The destination TM then multicasts the return value for the de-serialize command from the message scheduler to all other TMs.
7) Upon receiving the reply from the destination TM, the source TM checks whether the migration has succeeded. If not, the source TM tries to resume the task at the local module. If the task fails to resume, the task is killed and a KILLED message is multicast to all other modules. The non-source TMs wait for a successful reply value and issue UNFREEZE commands to their respective message schedulers, which will put blocked messages back to the message queue for delivery. Upon successfully de-serialization of the task, all TMs updates their local routing entries. For a failed de-serialization, the routing entries at all modules remain unchanged.

One important property of this protocol is that the messages sent before the FREEZE message will be buffered at the source task location and the message sent after the FREEZE message will be buffered at the senders. The messages that were buffered at the source will be forwarded to the new task location once the migration is complete. This ensures that at no time instance, the same task appears at both the source and the destination, which is an important property of atomic migration.

The source TM sends task virtual timers and task specific state. At the time of the serialization, the TM timestamps virtual timers with the global clock and stores the reminder timer ticks. When these serialized timers arrive at the destination, the time manager first computes the time that has been spent during migration by comparing the timestamp in the serialized timers and the global clock. Then, the TM subtracts those computed ticks from the virtual timer. If a timeout is fired during migration, the time manager then fires the message to `PORT_TIMEOUT` port of the task.

The serialization and de-serialization of the task state is associated with two special input ports: `PORT_SERIALIZE` and `PORT_DESERIALIZE`. The message scheduler, upon receiving a serialization command, sends a message to the `PORT_SERIALIZE` port. If the state of the task has already serialized, it ignores this message. Otherwise, the task allocates a continuous region of memory and serializes its state. Similarly, the message scheduler sends a message to `PORT_DESERIALIZE` port when it receives a de-serialize command from the TM. The task then de-serializes the content of the message according to the defined data structure for the task state. We depend on the tasks to perform serialization and de-serialization, because the task state may contain pointers to dynamically allocated memories, and without MMU support CoMOS cannot not detect them.

VI. Evaluation

CoMOS has been ported to the *mPlatform*. We have designed a set of experiments to evaluate the performance of CoMOS with respect to latency and memory overhead. The evaluation is carried out in the context of a complex real-time sensing application. The hardware platform used for the experiments is a five-module *mPlatform* prototype, where one module has an OKI ML67Q5003 processor (60MHz, 32kB RAM), while each of the other 4 sensing modules has a TI MSP430F1611 processor (6MHz, 10kB RAM). Each sensing module has a microphone and an amplifier to sense audio. One of the sensing modules also includes an 802.15.4 radio for wireless communication, and another sensing module includes a temperature sensor.

A. Application description

To stress test CoMOS, we devised an application scenario in which the *mPlatform* device is used simultaneously for audio conferencing in an office setting and for fire hazard

detection. We have implemented a 4-channel SSL algorithm together a fire-breakout-detection (FBD) algorithm on the top of CoMOS, whose task graph is shown in Figure 4. In addition to benchmarking the latency and memory overhead in messaging and migration, this setup allows us to evaluate the effectiveness of the preemption mechanism of CoMOS when multiple applications share processing and hardware resources.

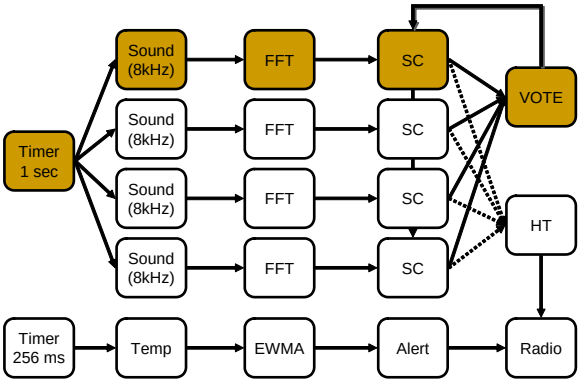

Fig. 4—The task graph for sound source localization (SSL) and fire breakout detection (FBD). Top four processing chains belong to SSL while the bottom processing chain is FBD.

SSL is a classic problem in sensing that detects the direction of a sound source using a microphone array, and it is at the center of applications such as teleconference and intelligent user interfaces. We use SRP-PHAT, a well-know algorithm for SSL [5], in a setup of 4 microphones forming a square. SRP-PHAT estimates the source location by computing the delays between arrivals of the audio signal at different microphones, via the maximization of the signal cross-correlation function. The signal processing is usually performed in the frequency domain because of more efficient processing and noise filtering. The FFT task applies Fourier transform to a block of 512 audio samples. The SC task performs noise power estimation, which is used to detect the presence of a voice. If more than two channels detect voices, through a voting procedure, the HT task is executed to determine the source location through correlation maximization. Concurrently, the FBD task chain periodically samples temperature sensor (Temp) at 4 Hz and performs exponentially weighted moving average (EWMA). The Alert task detects possible fire breakout using the energy component in the temperature readings. Even though FBD is computationally less intensive compared to SSL, whenever a fire is detected FBD gets a higher priority to send an alert.

B. Priority Assignment

Radio task had the highest priority since the system tasks running at highest priority use this task. We assigned the priority 1 and 2 to all the other tasks of SSL and FBD respectively.

C. Messaging Latency

We implemented a subset of the SSL application tasks, shown highlighted in Figure 4, on a single sensing module to evaluate the CoMOS Messaging induced latency overhead. In this implementation, when SC detects a valid sound source it sends out 1 kB of FFT results over the CPLD bus, otherwise SC updates the noise spectrum.

	Sound Source (ms)	Noise (ms)
CoMOS	253.0	287.0
Standalone	250.6	284.6
Overhead	0.94%	0.84%

TABLE II—SSL execution latency on single mPlatform module.

We implemented both the CoMOS-based version and a non-CoMOS version of this task graph to measure the overhead. For the non-CoMOS version, we unwrapped the CoMOS task framework and implemented the application as a standalone image. The standalone image was programmed in a single C file, to allow extensive compiler optimization. In the standalone version, all events such as the timer and Analog-to-Digital Converter(ADC) interrupts, were handled in hardware interrupt handlers. Further, the standalone version did not use dynamic memory. We consider our implementation of the standalone version to be the optimal in both speed and code size. We compiled both versions using the IAR V3.41 compiler, with the maximum speed optimization option enabled.

To measure the end-to-end latency of the CoMOS-based and standalone versions of the application, we set a GPIO pin of the processor high when the timer interrupt fired and clear the pin when SC finished processing.

Table II shows our latency measurement in milliseconds. Without CoMOS, the end-to-end latency was measured to be 250.6 ms when there is a sound source. With CoMOS, the latency was increased by 2.4 ms. In one round of the application execution, there were 6 messages and 25 OS events logged. OS event logging is the CoMOS mechanism for exporting information to the task scheduler. The task scheduler, which decides when and where a task should be migrated, is outside the scope of this paper, but we are interested in the overhead introduced due to OS event logging. Figure 5 shows the task graph with 25 events listed. Of those 2.4 ms, 1.4 ms were due to messaging, 0.863 ms were due to OS event logging, 0.137 ms were due to dynamic memory operations such as allocation, free, and transfer. CoMOS results in a 0.94% CPU overhead when a sound source is detected, and a 0.84% overhead otherwise. We observe that this $< 1\%$ overhead is a small price to pay, compared to the ease of application development and resource management enabled by CoMOS on multiprocessor platforms. The stack preemption makes the overhead small since this type of preemption only generates a small number of messages. With stack preemption, none of the task implementations needed to post messages back to itself for continuation while maintaining responsiveness. In fact, the FFT implementation was a simple matter of downloading from the web and wrapping it with CoMOS task framework. The whole process took us less than 10 minutes. Next we examine CoMOS messaging latency in more detail.

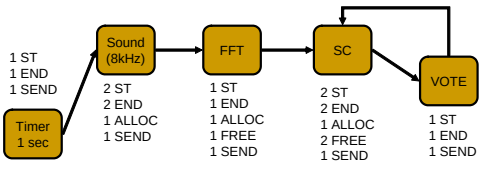

Fig. 5—The types and numbers of events recorded for each round of execution. ST: Message Start Event, END: Message End Event, SEND: Message Sent Event, ALLOC: Memory Allocation Event, FREE: Memory Free Event.

To further understand the latency of CoMOS messaging and OS event logging, we measured individual message latencies by setting a GPIO pin before posting a message and clearing the pin when the message arrived at the destination. Figure 6 shows the message latencies for each message destination. The solid bars show the latency without OS event logging and the hollow bars with logging. Here SC2 refers to the message received at SC from the Vote task. The suffix "Mcast" refers to multicast versions of the messages used in the multi-module SSL implementation shown in Figure 4.

We observe that messaging does not have a constant overhead. The overhead is highly dependent on the task graph, type of messages, and the priority setting. Messages to the Timer task were originated from hardware timer and the delta-timer. We observe the extra processing (127.8 μs overhead) involved in handling the delta-timer messages, compared to a normal message such as one destined to the FFT task (80.96 μs overhead). The extra processing for creating message to the Timer task is due to manipulating delta-timer list, which is a linked list storing the time duration to the next hardware timer interrupt.

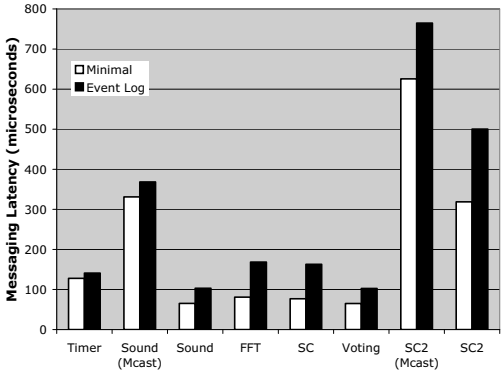

Fig. 6—Messaging latency in microseconds. For each result, the bar on the left is the latency without OS event logging. The bar on the right is the latency with OS event logging

We also observe that multicast messages result in high overhead. For example, multicast messages to Sound have 5 times the latency of unicast messages. This is because each multicast message results in a message broadcast over the CPLD bus. As we observe from the SC2 latency data, cyclic task graphs also result in higher overhead. SC2 (Mcast), for example, has incurred 625.7 μs. This is because SC was already preempted on the stack when it sent the message to VOTE task, causing the second message to be buffered in the queue until the entire message sending chain is unwrapped.

In Figure 6, the column on the right gives the latency with OS event logging for each destination task. In general, each message results in 3 logged events: message sent (SEND), message start (ST), and message end (END). SEND and ST are included in the latency measurement. The minimum event logging latency is measured to be 13 μs, but it can be as high as 181.37 μs as we observe in the SC2 unicast. This is because the latency also includes the END event of all preempted tasks. Messaging can generate several more events due to the associated memory management features. In the FFT and SC tasks, for example, the ownership of the memory gets transferred from the sender to the destination; this results in two more events: FREE and ALLOC events at the sender and the destination respectively.

System	Code Size	Data Size	Heap Size	Heap Usage (Max)
CoMOS	19424	518	6144	4498
Standalone	4104	4180	0	0

TABLE III—Memory Comparison in bytes

Component	Code Size	Data Size
Hardware Drivers	2934	292
Messaging Kernel	7562	64
Memory Management	1840	68
Virtual Timer	732	8
Migration Protocol	2548	86
Application	3808	0

TABLE IV—CoMOS Memory usage in bytes. The heap size of 6kB is excluded from memory management

We contend that the functionalities provided by messaging: context switch, memory management, and event logging more than justify the messaging overhead. For example, a message that implements a context switch has a 65.32 μs latency. This context switch includes software check for remote destination, multi-cast messages, and priority of current executing task. In contrast, thread based preemption, typically implemented using a setjump library call (17 μs), a longjmp (17 μs), and some scheduling algorithm ($\simeq$ 21.2 μs), incurs a latency of $\simeq$ 55.2 μs.

D. Memory Overhead

One important aspect of any operating system is the resulting memory overhead. We measured the code and data memory sizes of the application used in the latency measurements. Table III lists the memory footprint of the CoMOS-based and standalone implementations. CoMOS relied heavily on dynamic memory; therefore, the static data usage was small (430 bytes). In contrast, standalone version relied on static data. Standalone used 2048 bytes to store 512 samples of integer FFT conversion and 2078 bytes for storing noise power estimates. Since integer FFT results were only used as intermediate computation, having dynamic memory improves temporary memory usage. The sum of maximum heap usage and data size in CoMOS can be used to compare the memory usage against standalone application. In this particular application, the total overhead is 836 bytes. Part of memory overhead comes from static data (518 bytes), which we will explain later. The remaining overhead of 318 bytes is due to OS meta-data allocation based on slab allocation. In our current implementation, CoMOS allocate 8 objects for timer control block (14 bytes per object) and task control block (21 bytes per object) regardless of the numbers of blocks used.

Table IV shows the memory footprint of CoMOS version broken up into individual components. Application code size is slightly less than the standalone image size because libc runtime, math library and interrupt vectors are accounted for in the Messaging Kernel. CoMOS occupies 15616 bytes in program memory. On MSP430F1611, this is about 31.7% of total program memory (48kB). Even with the relatively complex sensing application we used here, the low end microcontroller can still accommodate 8 such applications together

with CoMOS, so we do not consider the program memory overhead to be a major issue.

E. Migration Latency

Since task migration is a key feature of CoMOS, low latency task migration becomes important. In this section, we measure the task migration latency of CoMOS using a task with variable task state. Here we define the migration latency to be the time between the instance a migration request arrives at a task and the instance the task finishes de-serializing its state at the destination. Note that in our definition, the program image size of the task is not relevant since we assume that the program image is already installed at the destination. There are two variables that can affect the migration latency: number of modules and the size of the task state. The number of modules affect the latency because CoMOS must wait for replies from all other modules before sending the serialized task resources to the destination module. Here, the CoMOS at the origin must process each reply sequentially, causing the latency to increase with the number of modules.The size of the task state affects migration latency because the state must be transmitted to the destination. Another possible source of latency is the delay due to bus contention. For example, a CoMOS message may get queued due to another message currently being transmitted; however, we can ameliorate this by extending the CPLD bus implementation to preempt an on going bus transmission when a high priority message arrives. To measure the task migration latency, we used an empty task that allocates a variable size task state during its initialization. To measure latency, a GPIO pin at the source was set when the migration request arrived, and a GPIO at the destination was set when the task finished de-serializing its state.

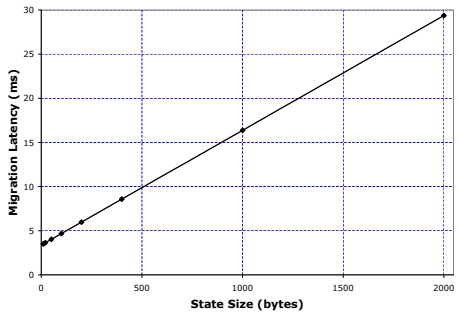

Fig. 7—Migration latency in terms of number of bytes in the task state.

Figure 7 shows the task migration delay versus the task state size. The data was measured on two sensing modules with MSP430 processors running at 6 MHz. Each data point corresponds to the average of 10 runs. We observed that, for a given number of modules, the latency of task migration is linearly proportional to the size of the task state. The plot intersects the Y-axis at 3.5 ms, which corresponds to a 3.5 ms fixed base latency. Each additional byte of task state add 0.013 ms to the migration latency. Although our simple task does not have any output ports, each out port adds two bytes to the state during migration. Each timer corresponds to 14 bytes of state, hence the 4 timers per task add a maximum of 0.73 ms to the migration latency. For example, migrating a task with 1 kB of state imposes a 17.23 ms delay. Since we assume task migration to be an infrequent activity, this overhead is well within acceptable limits.

We note that the task scheduler can use these latency results when deciding to migrate timing sensitive tasks. For example, if the deadline of an event is 10 ms and migrating the task to faster processor reduces processing time by 5 ms, depending on the size of task state, migration might offset the benefit from faster processor. Further, these results suggest that tasks with large state should be migrated less frequently.

F. End to End Application Performance Evaluation

To measure the end-to-end overhead introduced by CoMOS and to examine the responsiveness of the preemptive mechanism in CoMOS, we implemented the combined SSL and FBD application on a five-module mPlatform prototype with the task-to-module mapping shown in Figure 8.

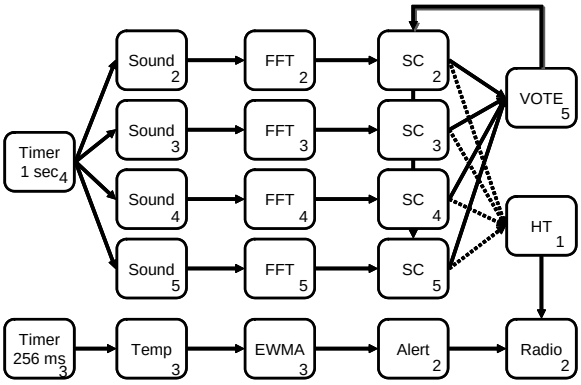

Fig. 8—Task-to-module mapping for SSL and FBD applications: module ID is shown on the bottom-right corner of each task. Module 1 corresponds to computational module (ARM board) and modules 2 to 5 correspond to sensing modules. Radio is on module 2 and temperature sensor is on module 3.

To measure the overhead introduced by CoMOS, we implemented the standalone version of the same application without CoMOS. One immediate observation was that implementing the standalone version was very tedious. We had to manually keep track of five different program images for five different modules as the task-to-module assignment was unique for each module. Every time we experimented with a different mapping, we had to carefully ensure the task-to-module mapping was correct. For CoMOS, we only had to make sure that the application description was correct and the task images were included. Changing task locations was as simple as changing an integer in the application description. When the system started, CoMOS initialized the application according to the application description. Further, for the standalone version, the application writer had to deal with low-level issues such as configuring various hardware interrupts to achieve the desired functionality.

Next we measured the processing overhead introduced by CoMOS. Since the CPU requirement for FBD related tasks is extremely low, we used only the SSL portion of the application to take measurements. We measured latency by setting a GPIO pin just after the timer hardware interrupt, and by clearing the GPIO pin after HT has completed (to measure latency for noise input, we set a GPIO on a sensing module just after SC completes the noise update).

	Sound Source (ms)	Noise (ms)
CoMOS	392.0	287.0
Standalone	389.6	284.6
Overhead	0.616%	0.836%

TABLE V—SSL execution latency on five mPlatform modules.

Table V shows latency measurements. The column labeled "Sound Source" gives the latency when a sound source was detected by the first pass of SC. The column labeled "Noise" gives latencies when SC identified the acoustic samples to be noise. Latency numbers for "Sound Source" column are higher because HT needs to perform hypothesis testing and compute the direction of the sound source. We observed that the overhead due to CoMOS is $< 1\%$ in both these instances. Hence, for data processing intensive application, CoMOS provides its services with negligible computation and latency overhead.

Next we examined the effectiveness and the correctness of stack preemption mechanism by invoking the FBD related tasks while SSL tasks are busy executing. While the SSL related tasks are running, we increased the temperature of the air around the temperature sensor. Each time FBD detected a "fire", we measured the execution time of the FBD related task chain by setting a GPIO pin when the temperature sampling was completed, and clearing the GPIO pin when Alert finished processing the EWMA result. The FBD execution time was always less than 10 ms. Since the execution time of the SSL related FFT task is 100 ms, this shows that the stack preemption mechanism is working properly and makes the system responsive enough for efficient real-time event handling.

VII. Concluding Remarks

CoMOS enables a cross-module uniform programming abstraction, resource efficiency, and real-time responsiveness for heterogeneous multi-processor platforms by using component messaging, migration, asymmetric stack preemption mechanisms.

The task component messaging model allows an application to span across multiple processor modules in a transparent manner. Messaging over the bus incurs little overhead, as small as 300μs, comparing to local message communication. Flow control mechanism employed in the CPLD bus has simplified the implementation of CoMOS significantly. Further, the broadcast nature of the bus enables us to efficiently implement multicast messaging and create multicast group members specific to the modules, which reduces memory consumption. The memory overhead of CoMOS is also low. One MSP430, the CoMOS kernel occupies 15616 bytes out of 48kB program memory and incurs 836 bytes out of 10kB data memory to support prioritized execution, dynamic memory allocation, task migration, and virtual timers. Comparing to 678 bytes in SOS and 500 bytes in Mantis, CoMOS has slightly higher memory usage, but CoMOS uses 292 bytes for hardware drivers.

Currently, CoMOS task migration does not including the transferring of task images. It assumes the target code is already on the destination boards. For a task code size of 1kB, the additional latency for migrating the code (task implementation) can be of 14 ms, comparing to 3.5 ms base cost for migrating the state and message routing table. When migrating tasks code cannot fit in every destination processor, we should take into account the latency incurred for transmitting task code. Furthermore, any task that will migrate across heterogeneous processors will have to have the code image for those processor architectures. This introduces extra complexity for preparing the executables. Currently, mPlatform has only two processor architectures: ARM7 and MSP430. Therefore, application code image is at most twice as big as the homogeneous architecture. Future multi-processor platform should limit the amount of heterogeneity in the system or go with a virtual machine approach.

CoMOS asymmetric stack preemption solves long running task problem with very low memory requirement. Traditional preemptive multi-threaded execution model will allocate at least 128 bytes per-task private stack. So the 5 tasks in SSL would incur 640 bytes additional memory overhead, which is significant on MSP430. CoMOS moves per-task stack into global stack. The assumption is that not all tasks will be on the stack simultaneously; hence, the overall stack utilization is low. On the other hand, when CPU utilization is high, this assumption may not hold true. In which case, traditional threading model may be a better design choice.

Stack-based preemption mechanisms, although restrain the capability of changing priorities dynamically, is powerful enough to support simple real-time scheduling frameworks such as rate-monotonic and earliest deadline first schedules. The major difference between our implementation and Baker's [1] is that we do not need to use priority ceiling to prevent possible dead lock introduced by the sharing of the global stack. Thanks to our choice of the asynchronous task model. CoMOS message scheduler automatically queues message to the task on the stack. As a consequence, there is no need to assign priorities to all resources in the system.

CoMOS creates task control blocks (TCB) for every task in the system to avoid the cost of discovering task location at runtime. When the number of tasks is large, this will result in poor memory utilization. Currently, each remote TCB costs 6 bytes of memory. When more information is needed in remote TCB, online discovery of task location may be a better design choice.

Appendix

Here we provide a detailed analysis of the multi-processor energy efficiency advantage mentioned in Section III-A. To simplify our discussion, we assume that each processor has one mode, in terms of voltage and frequency scaling. We introduce the following parameters for a processor *i*:

- N^i is the *instruction throughput* of the processors, in terms of millions of instructions per second (MIPS). For a workload of L instructions, the execution time is $T^i(L) = L/N^i$.
- M^i is the *power throughput* of the processors, in terms of million instructions per Joule (MIPJ). So, the energy spent in executing L instructions is $E^i(L) = L/M^i$.
- P^i_A: power consumption in the *active* mode, where the processor is actively running a task;

- P_I^i: power consumption in the *idle* mode, where the processor is executing NOP;
- P_S^i: power consumption in the *standby* mode, where the clock to the processor is turned off;
- P_W^i: power consumption in the transition between active and standby modes. This include the average power spends on both going to standby and waking up from standby to the active mode. We use T_W^i to denote the time spent in the transition. So the total energy cost for going into and waking up from the standby mode is $E_W^i = P_W^i \cdot T_W^i$.

Breakeven time T_{be}^i is defined as the amount of idle time such that the energy spent in the idle mode is the same as the energy spent in standby mode and the transition mode [20]. That is $T_{be}^i = (E_W^i - P_S^i \cdot T_w^i)/(P_I^i - P_S^i)$. Consider that L on processor i must be finished by a deadline D. If the slack time, $D - T^i(L) > T_{be}^i$, then it is more energy efficient to switch to the standby mode. Otherwise, the processor should stay in the idle mode until the next activation of the task.

With the above setup, we consider two heterogeneous processor, p and q, where p is more powerful and power efficient than q. That is, $M^p > M^q$ and $N^p > N^q$. We further assume that q is an ultra-low-power microcontroller, (e.g MSP430), such that the energy cost in the standby mode and the mode transitions are negligible. I.e. $P_S^q = 0$, $E_W^q = 0$, so $T_{be} = 0$.

Assuming a periodic non-splitable workload with period D and deadline D, Figure 1 is obtained as following:

- The areas below line $a : D = L/N^p$ is non-schedulable, since not even p can meet the deadline.
- In the areas above line a but below line $b : D = L/N^q$, the workload can only be assigned to p since q cannot finish it on time.
- In the area above line b, both processors can finish the workload on time. Line $c : D = (1/N^p) \cdot L + T_{be}^p$ defines the breakeven line for p. That is, above this line, p can turn into the standby mode after finishing the work, while below the line, p will stay at idle after finishing the work. Notice that c is parallel to a.
- The area below c but above b is further split by Line $d : D = (\frac{1}{M^q \cdot P_I^p} - \frac{1}{M^p \cdot P_I^p} + \frac{1}{N^p})L$, where below d, it is more power efficient to use p without standby and above d, it is more efficient to use q.
- The area above c can be further split by the vertical line $e : L = E_W^p/(\frac{1}{M^q} - \frac{1}{M^p})$, where to the left of this line, it is more efficient to use q and to the right of this line, it is more efficient to use p.

It is easy to show that the intersection of lines c and d is always to the right of line e, unless when $P_S^p = 0$, the three lines intersect at a single point. So the lines are always arranged as shown in Figure 1.

REFERENCES

[1] T. P. Baker. A stack-based resource allocation policy for realtime processes. In *IEEE Real-Time Systems Symposium*, pages 191–200, 1990.

[2] 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. *Mob. Netw. Appl.*, 10(4):563–579, 2005.

[3] J. Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer*, pages 87–98, 1994.

[4] A. Boulis, C.-C. Han, and M. B. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *MobiSys'03*, pages 187–200, 2003.

[5] M. Brandstein and H. Silverman. A robust method for speech signal time-delay estimation in reverberant rooms. In *Proc. of ICASSP'97*, 1997.

[6] E. Cheong, J. Liebman, J. Liu, and F. Zhao. Tinygals: A programming model for event-driven embedded systems. In *Proceedings of 18th ACM Symposium on Applied Computing (SAC03)*, pages 698–704, 2003.

[7] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors*, 2004.

[8] N. Edmonds, D. Stark, and J. Davis. Mass: modular architecture for sensor systems. In *IPSN '05*, page 53, Piscataway, NJ, USA, 2005. IEEE Press.

[9] C.-L. Fok, G.-C. Roman, and C. Lu. Rapid development and flexible deployment of adaptive wireless sensor network applications. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'05)*, pages 653–662. IEEE, June 2005.

[10] S. Ganeriwal, D. Ganesan, H. Shim, V. Tsiatsis, and M. B. Srivastava. Estimating clock uncertainty for efficient duty-cycling in sensor networks. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 130–141, New York, NY, USA, 2005. ACM Press.

[11] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *Proc. of PLDI*, 2003.

[12] L. Girod, T. Stathopoulos, N. Ramanathan, J. Elson, D. Estrin, E. Osterweil, and T. Schoellhammer. A system for simulation, emulation, and deployment of heterogeneous sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 201–213, New York, NY, USA, 2004. ACM Press.

[13] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A dynamic operating system for sensor nodes. In *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*, pages 163–176, New York, NY, USA, 2005. ACM Press.

[14] J. Helander. Deeply embedded xml communication: towards an interoperable and seamless world. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 62–67, New York, NY, USA, 2005. ACM Press.

[15] J. Helander and A. Forin. Mmlite: a highly componentized system architecture. In *EW 8: Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*, pages 96–103, New York, NY, USA, 1998. ACM Press.

[16] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *ASPLOS-IX*, pages 93–104. ACM Press, 2000.

[17] A. Khemka and R. K. Shyamasundar. An optimal multiprocessor real-time scheduling algorithm. *Journal of Parallel and Distributed Computing*, 43(1):37–45, 1997.

[18] P. Levis and D. Culler. Mate: A tiny virtual machine for sensor networks. In *International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA*, Oct. 2002.

[19] T. Liu and M. Martonosi. Impala: a middleware system for managing autonomic, parallel sensor systems. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 107–118. ACM Press, 2003.

[20] Y.-H. Lu and G. D. Micheli. Comparing system-level power management policies. *IEEE Design and Test of Computers*, 18(2):10–19, March/April 2001.

[21] J. Luo and N. K. Jha. Power-conscious joint scheduling of periodic task graphs and aperiodic tasks in distributed real-time embedded systems. In *ICCAD*, pages 357–364. IEEE Press, 2000.

[22] D. Lymberopoulos, B. Priyantha, and F. Zhao. A flexible and efficient architecture for sharing data in stack-based sensor network platforms. Technical Report MSR-TR-2006-142, Microsoft Research, 2006.

[23] S. Matic, M. Goraczko, J. Liu, D. Lymberopoulos, B. Priyantha, and F. Zhao. Resource modeling and scheduling for extensible embedded platforms. Technical report, Microsoft Research, 2006.

[24] B. Schott, M. Bajura, J. Czarnaski, J. Flidr, T. Tho, and L. Wang. A modular power-aware microsensor with 1000x dynamic power range. In *IPSN '05*, page 66, Piscataway, NJ, USA, 2005. IEEE Press.

[25] D. Stewart, R. Volpe, and P. Khosla. Design of dynamically reconfigurable real-time software using port-based objects. *IEEE Trans. on Software Engineering*, 23(12):759–776, 1997.