Design and Implementation of TinyGALS: A Programming Model for Event-Driven Embedded Systems

by Elaine Cheong

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Edward A. Lee

Research Advisor

* * * * * *

David E. Culler

Second Reader

Abstract

TinyGALS is a globally asynchronous, locally synchronous model for programming event-driven embedded systems. At the local level, software *components* communicate with each other synchronously via method calls. Components are composed to form *modules*. At the global level, modules communicate with each other asynchronously via message passing, which separates the flow of control between modules. TinyGUYS is a guarded yet synchronous model designed to allow thread-safe sharing of global state between modules without explicitly passing messages. This programming model is structured such that code for all intermodule communication, module triggering mechanisms, and access to guarded global variables can be automatically generated from a high level specification. We present language constructs for this specification, as well as a detailed description of the semantics of this programming model. We also discuss issues of determinacy that result from the event-driven nature of the target application domain.

We have implemented the programming model and tools for code generation on a wireless sensor network platform known as the Berkeley motes. We present details of the code generation process, which is designed to be compatible with components written for TinyOS, a component-based runtime environment for the motes. We describe a redesign of a multi-hop ad hoc communication protocol using the TinyGALS model.

Contents

1 Introduction

Emerging embedded systems, such as sensor networks [41], intelligent vehicle/highway systems [23], smart office spaces [38, 37], peer-to-peer collaborative cell phones and PDAs [14], are usually networked and event-driven. In these systems, the communication network is typically formed in an *ad hoc* manner, and the embedded computers must react to many (typically unstructured) stimuli, including physical events, user commands, and messages from other systems. In other words, external events drive the computation in the embedded system.

As the complexity of networked embedded systems grows, the costs of developing software for these systems increases dramatically. Embedded software designers face issues such as maintaining consistent state across multiple tasks, handling interrupts, avoiding deadlock, scheduling concurrent threads, managing resources, and conserving power [17, 26]. Typical technologies for developing embedded software, inherited from writing device drivers and optimizing assembly code to achieve a fast response and a small memory footprint, do not scale with the application complexity. In fact, it was not until recently that "high-level" languages such as C and C++ replaced assembly language as the dominant embedded software programming languages. Most of these high-level languages, however, are designed for writing sequential programs to run on an operating system and fail to handle concurrency intrinsically.

Modern software engineering practices advocate the use of software components such as standard libraries, objects, and software services to reduce redundant code development and improve productivity. However, when developing a component it is not foreseeable whether it should lock any resources. A software component that does not include synchronization code may not be thread-safe, and may exhibit unexpected behavior when composed with other components. On the other hand, if a software component is developed with resource synchronization in mind, then it may be overspecified in an application that

1

does not require synchronization, which will result in a large footprint and slow execution.[1]

Unlike software for traditional computer systems, software for embedded systems is application-specific and often encapsulates domain expertise, especially when it must process sensor data or control actuators [27]. As a result, although concurrency has long been a key research theme for systems software such as operating systems and distributed computing middleware [6, 33, 32], formal treatment of them in embedded systems has largely been ignored by mainstream computer science researchers until recently. The application-specific nature of embedded software, combined with tight constraints on available processing power and memory space, make traditional approaches such as the use of layers of abstraction and middleware less applicable. A more promising approach is to use formal concurrency models to construct and analyze software designs, and to use software synthesis technologies to generate application-specific scheduling and execution frameworks that give designers fine-grained control of timing, concurrency, and memory usage.

One example of this more formal approach are the synchronous languages for reactive systems [9]. Languages such as Esterel [10] allow users to specify a system using the notion of global ticks and concurrent zero delay reactions. Their models are specific enough that the concurrency in the system can be compiled away, and the system behaves like a state machine at run time. Actor-oriented designs, such as those seen in the Ptolemy II framework [11], integrate a rich set of sequential and concurrent models, use a system-level type system to check their composability [28], and compile away as much concurrency as possible to obtain run-time efficiency and predictability [12]. These approaches give us a way to formally coordinate the concurrency inherent in the system.

---

[1]Herlihy proposes a methodology in [19] for constructing non-blocking and wait-free implementations of concurrent objects. Programmers implement data objects as stylized sequential programs, with no explicit synchronization. Each sequential operation is automatically transformed into a non-blocking or wait-free operation via a collection of synchronization and memory management techniques. However, operations may not have any side-effects other than modifying the memory block occupied by the object. This does not address the need for inter-object communication when composing components. Additionally, this methodology requires additional memory copying, which may become expensive for large objects.

Not all concurrency can be compiled away in all applications, especially when the rate of input events does not match the processing speed of the embedded computers or the real-time requirements of the applications. Then, multiple threads of execution are needed at run time. In these situations, it is not uncommon to use asynchronous models to coordinate sequential reactions. These coordination models must address how to enable concurrent tasks to exchange messages and/or share global state. In the PBO (port-based object) model as implemented in the Chimera real-time operating system [35], data are stored in a global space. A concurrent task, called a PBO, is free to access the data space at any time. Data in the space are persistent, and PBOs are triggered only by time with no explicit message passing among them. In the POLIS co-design [7] approach, software modules are generated as asynchronous tasks that pass messages through buffers of size one. There is no global data space in which to share data; information is only exchanged in the communication messages. Event-driven models for servers [40], such as SEDA (staged event-driven architecture) divide processing into a network of event-driven stages connected by explicit queues [39].

In this report, we describe a globally asynchronous, locally synchronous (GALS) approach for programming event-driven embedded systems. This approach provides language constructs that allow designers to specify concurrency explicitly at the system level while maintaining the sequential execution of basic components. It also enables the generation of an application-specific operating system (or more precisely, an execution framework) to provide a thread-safe execution environment for the components.

Terms such as "synchronous," "asynchronous," and "globally asynchronous, locally synchronous (GALS)" mean different things to different communities, thus causing confusion. The circuit and processor design communities use these terms for synchronous and asynchronous circuits, where synchronous refers to circuits that are driven by a common clock [22]. In the system modeling community, synchronous often refers to computational steps and communication (propagation of computed signal values) that take no time (or, in practice, very little time compared to the intervals between successive arrivals of input signals. GALS then refers to a modeling paradigm that uses events and handshaking to

integrate subsystems that share a common tick (an abstract notion of an instant in time) [8]. Our notions of synchronous and asynchronous are consistent with the usage of these terms in distributed programming paradigms [31]. We use synchronous to mean that the software flow of control transfers immediately to another component and the calling code blocks awaiting return. Steps do not take infinite time; control eventually returns to the calling code. Asynchronous means that control flow does not transfer immediately to another component; execution of the other component is decoupled.

Our programming model, called *TinyGALS*, uses two levels of hierarchy to build an application on a single processor. At the application level, *modules* communicate with each other asynchronously via message passing. Within each module, *components* communicate synchronously via method calls, as in most imperative languages. Thus, the programming model is globally asynchronous and locally synchronous in terms of the method call mechanisms. If modules exchange data that is frequently updated, then message passing at the global level may become inefficient. In our model, a set of guarded yet synchronous variables (called *TinyGUYS*) is provided at the system level for asynchronous modules to exchange global information "lazily". These variables are thread-safe, yet components can quickly read their values. In this programming model, application developers have precise control over the concurrency in the system, and they can develop software components without the burden of thinking about multiple threads.

In a reactive, event-driven system, most of the processor time is spent waiting for an external trigger or event. A reasonably small amount of additional code to enhance software modularity will not greatly affect the performance of the system. Automatically generated code reduces implementation and debug time, since the developer does not need to reimplement standard constructs (e.g. communication ports, queues, functions, guards on variables), and the generated code will not contain errors.

In this report, we describe a method for generating code for applications that use the TinyGALS programming model. In this software synthesis technique, code for communication and scheduling is automatically generated from applications specified using the

TinyGALS language constructs, as well as code for guarding access to TinyGUYS global variables. We have implemented this for use on the MICA motes [15], a wireless sensor network platform. The TinyGALS programming model and code generation facilities can greatly improve software productivity and encourage component reuse.

Our model is influenced by the TinyOS project at the University of California, Berkeley [21]. We use the TinyOS component specification syntax in our tools so that users can reuse existing TinyOS v0.6.1 [5] components in TinyGALS applications. TinyOS components provide an interface abstraction that is consistent with synchronous communication via method calls. However, unlike our model, concurrent tasks in TinyOS are not exposed as part of the component interface. Lack of explicit management of concurrency forces component developers to manage concurrency by themselves (locking and unlocking semaphores), which makes TinyOS components extremely difficult to develop.

Our approach differs from coordination models like those discussed above in that we allow designers to directly control the concurrent execution and buffer sizes among asynchronous modules. At the same time, we use a thread-safe global data space to store messages that do not trigger reactions. Components in our model are entirely sequential, and they are both easy to develop and backwards compatible with most legacy software. We also do not rely on the existence of an operating system. Instead, we generate the scheduling framework as part of the application.

The remainder of this report is organized as follows. Section 2 describes the TinyGALS language constructs and semantics. Section 3 discusses issues related to determinacy of a TinyGALS program. Section 4 explains a code generation technique based on the two-level execution hierarchy and a system-level scheduler. Section 5 gives an example of developing a multi-hop routing protocol using the TinyGALS model. Section 6 discusses related work, emphasizing relevant embedded software models. Section 7 concludes this report and describes directions for future work.

2 The TinyGALS Programming Model

TinyGALS is based on a model of computation that contains globally asynchronous and locally synchronous communication. We use this architecture to separate the rates of control of the system – the reactive part and the computational part – via asynchrony. Thus, the system can finish reacting to an event without having to wait until full processing of any tasks related to the event completes. Processing can be deferred until a time at which the system is not busy reacting to events.

A TinyGALS program contains a single application composed of modules, which are in turn composed of components. Execution in TinyGALS is driven by events. In Tiny-GALS, there are three types of events: (1) a hardware interrupt, (2) the transfer of control flow between two components via a method call, and (3) a message passed between two modules, whose data is encapsulated as a token. In the third case, we sometimes use the terms event and token interchangeably.

We first present a simple example and provide an informal, intuitive description of the TinyGALS architecture. We then describe each of the language constructs (components, modules, and application) and their semantics.
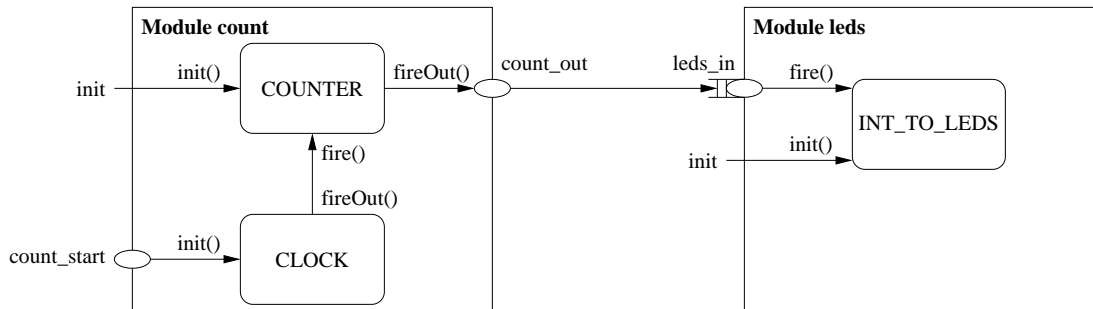
2.1 Introduction: An Example

Figure 1: An example application with two modules.

6

The example shown in Figure 1 is a simple application in which a hardware clock updates a software counter, whose binary value is displayed on a set of LEDs. The TinyGALS implementation of this application contains two modules: `count` and `leds`. Module `count` contains two components: `COUNTER` and `CLOCK`. Module `leds` contains a single component, `INT_TO_LEDS`.

At runtime, the system framework initializes the `COUNTER` and `INT_TO_LEDS` components via the initialization ports (both named `init`) of modules `count` and `leds`, respectively. The runtime system then activates execution of the application via the system start port, which in this example begins at the `count_start` input port of module `count`.

The hardware clock encapsulated by the `CLOCK` component periodically produces an event on the link between the `CLOCK` and `COUNTER` components via an interrupt service routine. The link represents a method call originating from the `CLOCK` component. The `COUNTER` component will respond to the call immediately by processing the event and generating another event at the `count_out` output port of module `count`. The token (encapsulated data) corresponding to the event produced at `count_out` is stored in the input port to which the output port is connected, `leds_in`. Later, the runtime scheduler will activate the `leds` module in response to this event. This possibly delayed response corresponds to an asynchronous communication between modules and allows for decoupling of the execution of modules. Finally, the `INT_TO_LEDS` component will process the event and display the value on the hardware LEDs.

2.2   TinyGALS Language Constructs

A TinyGALS program contains a single application composed of modules, which are in turn composed of components. In this section we describe the language constructs for each of these (components, modules, and application). Semantics are discussed in Section 2.3.

2.2.1 TinyGALS Components

Components are the most basic elements of a TinyGALS program. A TinyGALS component C is a 3-tuple:

$$C = (V_C, X_C, I_C), \tag{1}$$

where V_C is a set of internal variables, X_C is a set of external variables, and I_C is a set of methods that constitute the interface of C. The internal variables carry the state of C from one invocation of an interface method of C to another. The external variables can be accessed by C through read and write operations.[2] The set of methods I_C is further divided into two disjoint sets: $ACCEPTS_C$ and $USES_C$. The methods in $ACCEPTS_C$ can be called by other components (these are the inputs of component C), while the methods in $USES_C$ are those needed by C and may possibly belong to other components (these are the outputs of component C).[3] Thus, a component is like an object in most object-oriented programming languages, but with explicit definition of the external variables and methods it uses. Syntactically, a component is defined in two parts – an interface definition and an implementation.

Figure 2 shows a fragment of the code for the interface definition of the `COUNTER` component shown in Figure 1, while Figure 3 shows a fragment of the code for its implementation.[4] In Figure 2, we see that the component has two *ACCEPTS* methods and one *USES* method. In Figure 3, we see that the component accesses an internal variable of type `short` named `_counter`. Using the tuple notation given in Equation 1, the `COUNTER` component can be defined as $C = (V_C = \{\_counter\}, X_C = \emptyset, I_C = \{init, fire, fireOut\})$.

---

[2]External variables are implemented as TinyGUYS. See Section 2.4 for more information.

[3]Using TinyOS conventions, $ACCEPTS_C$ is the set formed by the union of the TinyOS `ACCEPTS` and `HANDLES` methods, and $USES_C$ is the set formed by the union of the TinyOS `USES` and `SIGNALS` methods.

[4]The syntax is similar to that of TinyOS.

```
// Component interface for COUNTER.
COMPONENT COUNTER;

ACCEPTS{
  char init(void);
  void fire(void);
};
USES{
  char fireOut(short value);
};
```

Figure 2: Fragment of code for interface definition of the COUNTER component shown in Figure 1.

2.2.2 TinyGALS Modules

Modules are the major building blocks of a TinyGALS program, encompassing one or more TinyGALS components. A TinyGALS module M is a 6-tuple:

$$M = (COMPONENTS_M, INIT_M, INPORTS_M, OUTPORTS_M, PARAMETERS_M, LINKS_M),$$
(2)

where $COMPONENTS_M$ is the set of components that form the module; $INIT_M$ is a list of initialization methods that belong to the components in $COMPONENTS_M$; $INPORTS_M$ and $OUTPORTS_M$ are sets that specify the input ports and output ports of the module, respectively; $PARAMETERS_M$ is a set of variables external to the components[5]; and $LINKS_M$ specifies the relations among the interface methods of the components (I_C in Equation 1) and the input and output ports of the module ($INPORTS_M$ and $OUTPORTS_M$). Section 2.3.3 describes links in more detail, including which configurations of components within a module are valid.

Modules are different from components; $INPORTS_M$ and $OUTPORTS_M$ of a module M are not the same as $ACCEPTS_C$ and $USES_C$ of a component C. While $ACCEPTS_C$

---

[5]Refer to information on TinyGUYS in Section 2.4.

```
// Component implementation for COUNTER.
char init(){
  _counter = 0;
  return 1;
}
void fire(){
  _counter++;
  CALL_COMMAND(fireOut)(_counter);
}
```

Figure 3: Fragment of code for implementation of the COUNTER component shown in Figure 1. This provides the method bodes for the interface defined in Figure 2. The internal variable _counter is of type short.

and $USES_C$ refer to method calls and may be connected to ports in $INPORTS_M$ and $OUTPORTS_M$, $INPORTS_M$ and $OUTPORTS_M$ are not executable. Additionally, ports in $INPORTS_M$ that are connected to ports in $OUTPORTS_M$ contain an implicit queue (see Section 2.3.4).

Initialization methods in $INIT_M$ are slightly different from input ports in $INPORTS_M$; they do not contain implicit queues since buffering of data is not needed during initialization.

Figure 4 gives the module definition code for the module count in the application shown in Figure 1. The definition states that module count contains two components, COUNTER and CLOCK. Note that component CLOCK accepts init() and uses fireOut(); component COUNTER accepts init() and fire(), and uses fireOut() (see Figure 2. The init section states that the init() method of component COUNTER belongs to the initialization list of the module. The connection section at the bottom of Figure 4 declares that the fireOut() method of component CLOCK is mapped to the fire() method of component COUNTER; the fireOut() method of component COUNTER is mapped to the count_out outport of module count; and that the count_start inport of module count is mapped to the init() method of component CLOCK.

```
// Module definition for count.
include components{
  COUNTER;
  CLOCK;
};
init{
  COUNTER:init;
};
ports in{
  count_start;
};
ports out{
  count_out;
};
CLOCK:fireOut COUNTER:fire
COUNTER:fireOut count_out
count_start CLOCK:init
```

Figure 4: The definition of the `count` module shown in Figure 1.

Using the tuple notation given in Equation 2, the `count` module can be defined as

$$
\begin{aligned}
M = (COMPONENTS_M &= \{COUNTER, CLOCK\}, \\
INIT_M &= [COUNTER : init], \\
INPORTS_M &= \{count\_start\}, \\
OUTPORTS_M &= \{count\_out\}, \\
PARAMETERS_M &= \emptyset, \\
LINKS_M &= \{(CLOCK : fireOut, COUNTER : fire), \\
&\quad (COUNTER : fireOut, count\_out), \\
&\quad (count\_start, CLOCK : init)\}).
\end{aligned}
$$

The code for the module `leds` is shown in Figure 5 for completeness. We will discuss the semantics of the execution of components within a module in more detail in Section 2.3.3.

11

```
// Module definition for leds.
include components{
  INT_TO_LEDS;
};
init{
  INT_TO_LEDS:init;
};
ports in{
  leds_in;
};
ports out{
};
leds_in INT_TO_LEDS:fire
```

Figure 5: The definition of the `leds` module shown in Figure 1.

2.2.3 TinyGALS Application

At the top level of a TinyGALS program, modules are connected to form a complete application. A TinyGALS application A is a 5-tuple:

$$A = (MODULES_A, GLOBALS_A, VARMAPS_A, CONNECTIONS_A, START_A), \quad (3)$$

where $MODULES_A$ is a list of modules in the application; $GLOBALS_A$ is a set of global variables; $VARMAPS_A$ is a set of mappings, each of which maps a global variable to a parameter (in $PARAMETERS_{M_i}$) of a module in $MODULES_A$[6]; $CONNECTIONS_A$ is a set of the connections between module output ports and input ports; $START_A$ is the name of an input port of exactly one module in the application. Section 2.3.4 describes which configurations of modules within an application are valid.

Figure 6 shows the code for defining the application shown in Figure 1. The application contains two modules, `count` and `leds`. The output port `count_out` of module `count` is connected to the input port `leds_in` of module `leds`. The definition declares that the connection between `count_out` and `leds_in` has a FIFO queue of size 50. The last

---

[6]Refer to information on TinyGUYS in Section 2.4.

12

```
// Application definition.
include modules{
  count;
  leds;
};
count_out -> leds_in 50
START@ count_start
```

Figure 6: Definition of the application shown in Figure 1.

line in the definition says that the system start port is `count_start`. Note that arguments (initial data) may be passed to the system start port on the same line.

Using the tuple notation given in Equation 3, the example application can be defined as

$$
\begin{aligned}
A = (MODULES_A &= [count, leds], \\
GLOBALS_A &= \emptyset, \\
VARMAPS_A &= \emptyset, \\
CONNECTIONS_A &= [(count\_out, leds\_in)], \\
START_A &= count\_start).
\end{aligned}
$$

2.3   TinyGALS Semantics

In this section, we discuss the semantics of execution within a component, between components within a module, and between modules within an application. We also include a discussion of the conditions for well-formedness an application.

2.3.1   Assumptions

The TinyGALS architecture is intended for a platform with a single processor. All memory is statically allocated; there is no dynamic memory allocation. A TinyGALS program

13

runs in a single thread of execution (single stack), which may be interrupted by the hardware. *Reentrant* code can have multiple simultaneous, interleaved, or nested invocations which will not interfere with each other. In this section, we assume that interrupts are not reentrant, but that an interrupt is masked while servicing it (interleaved invocations are disabled). However, other interrupts may occur while servicing the interrupt.[7] There are no other sources of preemption other than hardware interrupts. When using components in which interrupts are enabled in the interrupt handler, we must take special care in placing constraints on what constitutes a valid configuration of components within a module in order to avoid unexpected reentrancy, which may lead to race conditions and other nondeterminacy issues. We assume the existence of a clock, which is used to order events.

2.3.2   TinyGALS Components

There are three cases in which a component *C* may begin execution: (1) an interrupt from the hardware that *C* encapsulates, (2) an event arrives on the module input port linked to one of the interface methods of *C*, or (3) another component calls one of the interface methods of *C*. In the first case, the component is a *source component* and when activated by a hardware interrupt, the corresponding interrupt service routine is run. Source components do not connect to any module input ports. In the second case, the component is a *triggered component*, and the event triggers the execution of the method (in $ACCEPTS_C$) to which the input port is linked. Both source components and triggered components may call other components (via the methods in $USES_C$), which results in the third case, where the component is a *called component*. Once activated, a component executes to completion. That

---

[7]The *avr-gcc* compiler for the Atmel AVR microcontroller on the MICA motes provides two macros for writing user-defined interrupt service routines (locations specified in the interrupt vector table). The SIGNAL macro indicates that the specified function is a signal handler; interrupts are disabled inside the function. The INTERRUPT macro indicates that the specified function is an interrupt handler; interrupts are enabled inside the function. In the TinyOS v0.6.1 distribution, the following components use the INTERRUPT macro on the MICA motes: CLOCK, RF_PROXIMITY, TIMESTAMP, LOGGER, UART (transmission only); the following components re-enable interrupts inside of the SIGNAL macro on the MICA motes: MIC and ADC.

is, the interrupt service routine or method finishes.

Reentrancy problems may arise if a component is both a source component and a triggered component. An event on a linked module input port may trigger the execution of a component method. While the method runs, an interrupt may arrive, leading to possible race conditions if the interrupt modifies internal variables of the same component. Therefore, to improve the ease of analyzability of the system and eliminate the need to make components reentrant, source components must not also be triggered components, and vice versa. The same argument also applies to source components and called components. Therefore, it is necessary that source components only have outputs ($USES_C$) and no inputs ($ACCEPTS_C$). Additional rules for linking components together are detailed in the next section.

In Figure 3, `_counter` is an internal variable. Each time the `fire()` method of `COUNTER` is called, the component will call the `fireOut()` method with the value of the internal variable `_counter` as its argument. The `CALL_COMMAND` macro indicates that the `fireOut()` method will be called synchronously (we will explain this further in the next section). The component only needs to know the type signature of `fireOut()`, but it does not matter to which component the method is linked.

2.3.3 TinyGALS Modules

Flow of control between components within a TinyGALS module occurs on links. A link is a relation within a module M between a component method (in $USES_C$) and another component method (in $ACCEPTS'_C$), between an input port of the module (in $INPORTS_M$) and a component method (in $ACCEPTS_C$), or between a component method (in $USES_C$) and an output port of the module (in $OUTPORTS_M$). Links represent synchronous communication via method calls. When a component calls an external method (in $USES_C$) through `CALL_COMMAND`, the flow of control in the module is immediately transferred to the callee component or port. The external method can return a value through `CALL_COMMAND` just

15

as in a normal method call.[8] The graph of components and links between them is an abstraction of the call graph of methods within a module, where the methods associated with a single component are grouped together.

The execution of modules is controlled by a scheduler in the TinyGALS runtime system. There are two cases in which a module M may begin execution: (1) a triggered component is activated, or (2) a source component is activated. In the first case, the scheduler activates the component attached to an input port of M in response to an event sent to M by another module. In the second case, M contains a source component which has received a hardware interrupt. Notice that in this case, M may be interrupting the execution of another module. A module is considered to be finished executing when the components inside of it have finished executing and control has returned to the scheduler.

As discussed in the previous section, preemption of the normal thread of execution by an interrupt may lead to reentrancy problems. Therefore, we must place some restrictions on what configurations of components within a module are allowed.

Cycles within modules (between components) are not allowed, otherwise reentrant components are required.[9] Therefore, any valid configurations of components within a module can be modeled as a directed acyclic graph (DAG). A *source DAG* in a module M is formed by starting with a source component C in M and following all forward links between C and other components in M. Figure 7 shows an example of a source DAG inside of a module. A *triggered DAG* in a module M is formed by starting with a triggered component C in M and following all forward links between C and other components in M. Figure 8 shows an example of a triggered DAG inside of a module.

In general, within a module, source DAGs and triggered DAGs must be not be connected. In Figure 9, the source DAG ($C1$, $C3$) is connected to the triggered DAG ($C2$, $C3$). Suppose $C2$ is triggered by an event on the input port of M and calls $C3$. Reentrancy

---

[8]In TinyOS, the return value indicates whether the command completed successfully or not.

[9]Recursion within components is allowed. However, the recursion must be bounded for the system to be live.
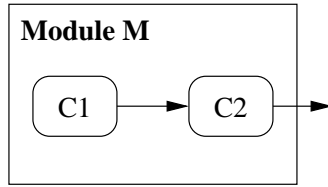
Figure 7: A source DAG inside Module M. This DAG is activated by a hardware interrupt.
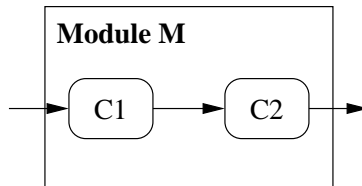

Figure 8: A triggered DAG inside Module M. This DAG is activated by the arrival of an event at the module input port.

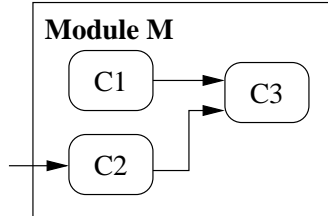problems may occur if an interrupt causes $C1$ to preempt $C3$.

Figure 9: Not allowed; source DAG connected to triggered DAG.

If all interrupts are masked during interrupt handling (interrupts are disabled), then we need not place any additional restrictions on source DAGs. However, if interrupts are not masked (interrupts are enabled), then a source DAG must not be connected to any other source DAG within the same module.

Triggered DAGs can be connected to other triggered DAGs, since with a single thread of execution, it is not possible for a triggered component to preempt a component in any other triggered DAG. Recall that once triggered, the components in a triggered DAG will execute to completion. We must also place restrictions on what connections are allowed between component methods and module ports, since some configurations may lead to

17

nondeterministic component firing order.

Let us first assume that both module input ports and module output ports are totally ordered (we assign the order to be the same as the order specified in the `ports in` and `ports out` sections of the module definition file). However, we assume that components are not ordered. As discussed earlier, the configuration of components inside of a module must not contain cycles and must follow the rules above regarding source and triggered DAGs. Then module input ports may either be associated with one method of a single component C (in $ACCEPTS_C$) or with one or more module output ports. Likewise, outgoing component methods ($USES$) may be associated with either one method of a single component C (in $ACCEPTS_C$) or with one or more module output ports.[10] Incoming component methods ($ACCEPTS$) may be associated with any number or combination of component methods ($USES$) and module input ports, but they may not be associated with module output ports. Likewise, module output ports may be associated with any number or combination of outgoing component methods ($USES$) and module output ports.

If neither module input ports nor module output ports are ordered, then module input ports and outgoing component methods may only be associated with either a single method or with a single output port.

In Figure 4, the connection section at the bottom of the module definition declares that whenever the `count_start` input port is triggered (which will be explained in the next section), the `init()` method of CLOCK will be called; whenever CLOCK calls `fireOut()`, the method `fire()` of COUNTER will be called; and whenever COUNTER calls `fireOut()`, an event will be produced at the `count_out` output port.

─────────────────────

[10]In the existing TinyOS constructs, one caller (outgoing component method) can have multiple callees. The interpretation is that when the caller calls, all the callees will be called in a possibly non-deterministic order. One of the callee's return values will be returned to the caller. Although multiple callees are not part of the TinyGALS semantics, it is supported by our software tools to be compatible with TinyOS.

2.3.4 TinyGALS Application

The execution of a TinyGALS system begins with initialization of all methods specified in $INIT_{M_i}$ for all modules M_i. The order in which modules are initialized is the same as the order in which they are listed in the application configuration file (e.g., as in Figure 6). The order in which methods are initialized for a single module is the same as the order in which they are listed in the module configuration file (e.g., as in Figure 4 or Figure 5).

Execution of the application begins at the system start port ($START_A$), which is a module input port declared in the `START@` section of the application configuration file. After module initialization, the TinyGALS runtime system triggers the system start port exactly once. Additional start triggers are not used since there is a single thread of execution. If initial arguments were declared in the application configuration file, these are passed to the component input method that is linked to the system start port at this time. For example, in Figure 6, the application starts when the runtime system triggers the input port `count_start` of module `count`. The components in the triggered DAG of the starting module execute to completion and may generate one or more events at the output port(s) of the module, which we discuss next. During execution, interrupts may occur and preempt the normal thread of execution. However, control will eventually return to the normal thread of execution.

During application execution, communication between module ports occurs asynchronously via FIFO queues. When a component within a module calls a method that is linked to an output port, the arguments of the call are converted into tokens. For each input port connected to the output port, a copy of the token is placed in its FIFO queue. Later, a scheduler in the TinyGALS runtime system will remove a token from the FIFO queue and call the method that is linked to the input port with the contents of the token as its arguments. The queue separates the flow of control between modules; the call to the output port will return immediately, and the component within the module can proceed. Communication between modules is also possible without the transfer of data. In this case, an empty message (token) transferred between ports acts as a trigger for activation of the receiving

19

module. Tokens are placed in input port queues atomically, so other source components cannot interrupt this operation. Note that since each input port of a module *M* is linked to a component method, each token that arrives on any input port of *M* corresponds to a future invocation of the component(s) in *M*. When the system is not responding to interrupts or events on input ports, the system does nothing (i.e., sleeps).

The TinyGALS semantics do not define exactly when the input port is triggered. We discuss the ramifications of token generation order on the determinacy of the system in Section 3. Our current implementation processes the tokens in the order that they are generated as defined by the hardware clock. Tokens generated at the same logical time are ordered according to the global ordering of module input ports, which we discuss next. The runtime system maintains a global event queue which keeps track of the tokens in all module input port queues in the system. Currently, the runtime system activates the modules corresponding to the tokens in the global event queue using FIFO scheduling. More sophisticated scheduling algorithms can be added, such as ones that take care of timing and energy concerns.

In the previous section, we discussed limitations on the configuration of links between components within a module. Connections between modules are much less restrictive. Cycles are allowed between modules. This does not lead to reentrancy problems because the queue on a module input port acts as a delay in the loop. Module output ports may be connected to one or more module input ports, and module input ports may be connected to one or more module output ports. The single-output-multiple-input connection acts as a fork. For example, in Figure 10, every token produced by `A_out` will be duplicated and trigger both `B_in` and `C_in`. Tokens that are produced at the same "time", as in the previous example, are processed with respect to the global input port ordering. Input ports are first ordered by module order, as they appear in the application configuration file, then in the order in which they are declared in the module configuration file. The multiple-output-single-input connection has a merge semantics, such that tokens from multiple sources are merged into a single stream in the order that the tokens are produced. This type of merge does not introduce any additional sources of nondeterminacy. See Section 3 for a discussion

20

of interrupts and their effect on the order of events in the global event queue.
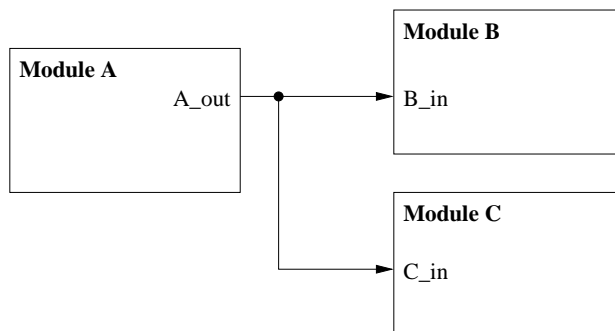

Figure 10: Single output multiple input connection.

2.4   TinyGUYS

The TinyGALS programming model has the advantage that the modules become decoupled through message passing and are easier to develop independently. However, each message passed will trigger the scheduler and activate a receiving module, which may quickly become inefficient if there is global state which must be updated frequently. *TinyGUYS* (Guarded Yet Synchronous) is a mechanism for sharing global state, allowing quick access but with protected modification of the data.

One must be very careful when implementing global data spaces in concurrent programs. Many modules may access the same global variables at the same time. It is possible that while one module is reading the variables, an interrupt may occur and preempt the reading. The interrupt service routine may modify the global variables. When the module resumes reading the remaining variables after handling the interrupt, it may see an inconsistent state. In the TinyGUYS mechanism, global variables are guarded. Modules may read the global variables synchronously (i.e., without delay). However, writes to the variable are asynchronous in the sense that all writes are delayed. A write to a TinyGUYS global variable is actually a write to a copy of the global variable. One can think of this as a write buffer of size one. Because there is only one buffer per global variable, the last module to write to the variable "wins", i.e., the last value written will be the new value of the global

21

```
// Application def.          // Module def. for A        // Component impl. for A1
include modules{             include components{         ...
  A;                           A1;                       void fire() {
  B;                         };                            ...
};                           parameters{                   int a;
globals{                       int A_param;                a=PARAMETER_GET(A_param);
  statevar;                  };                            a++;
};                           ...                           PARAMETER_PUT(A_param)(a);
statevar <=> A_param                                       ...
statevar <=> B_param                                     };
...
```

Figure 11: Defining and accessing TinyGUYS variables.

variable. TinyGUYS variables are updated atomically by the scheduler only when it is safe (e.g., after one module finishes and before the scheduler triggers the next module). One can think of this as a way of formalizing race conditions. We discuss how to eliminate race conditions in Section 3.1.

TinyGUYS have global names ($GLOBALS_A$) that are mapped to the parameters ($PARAMETERS_M$) of each module M. If a component C uses a parameter, it must declare it as an external variable (X_C). Figure 11 shows some sample code for defining and accessing TinyGUYS variables. The left-hand column of Figure 11 shows an application definition, which contains a list of global variable names, as well as a list of mappings from global names to local names. These local variables must be defined as parameters of the modules; this is shown in the center column of Figure 11. Local variables can be accessed within a component by using special constructs, PARAMETER_GET and PARAMETER_PUT, as shown in the right-hand column of Figure 11. This style of declaration, in which the types of the global variables must be declared at the module level, is slightly awkward. This means that the types must be declared for each module that uses the global variables. It also increases the fragility of the component code, since the components do not know the types of the global variables. In the future, we plan to improve this mechanism to use scoping, perhaps in a way similar to scoped parameters in Ptolemy II [11]. We could also use the Ptolemy II type system to improve the way in which types must be declared in TinyGALS.

22

3 Discussion

In this section we discuss issues related to determinacy of a TinyGALS program. We begin with definitions for a TinyGALS system, system state (including quiescent system state and active system state), module iteration (in response to an interrupt and in response to an event), and system execution. We also review the conditions for well-formedness of a TinyGALS system.

Definition 1 (System). A *system* consists of an application and a global event queue. Recall that an application is defined as:

$$A = (MODULES_A, GLOBALS_A, VARMAPS_A, CONNECTIONS_A, START_A).$$

Recall that the inport associated with a connection between modules has a FIFO queue for ordering and storing events destined for the inport. The global event queue provides an ordering for tokens in all inport queues. Whenever a token is stored in an inport queue, a representation of this event (implemented as an identifier for the inport queue) is also inserted into the global event queue. Thus, events that are produced earlier in time appear in the global event queue before events that are produced later in time (with respect to the system clock). Events that are produced at the same time (e.g., as in Figures 12 or 10) are ordered first by order of appearance in the application modules list ($MODULES_A$), then by order of appearance in the modules inports list ($INPORTS'_M$, which is an ordered list created from the modules inports set $INPORTS_M$).
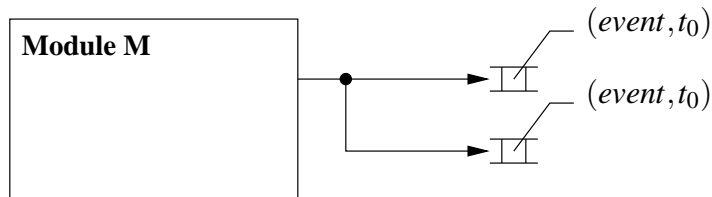

Figure 12: Two events are produced at the same time.

Definition 2 (System state). The *system state* consists of four main items:

23

1. The values of all internal variables of all components (V_{C_i}).

2. The contents of the global event queue.

3. The contents of all of the queues associated with module input ports in the application.

4. The values of all TinyGUYS (*GLOBALS$_A$*).

Recall that the global event queue contains the events in the system, but the module input ports contain the data associated with the event, encapsulated as a token.

There are two distinct kinds of system state: quiescent and active.

Definition 2.1 (quiescent system state). A system state is *quiescent* if there are no events in the global event queue, and hence, no events in any of the module inport queues in the system.

Definition 2.2 (active system state). A system state is *active* if there is at least one event in the global event queue, and hence, at least one event in the queue of at least one module inport.

Note that a TinyGALS system actually starts in an active system state, since execution begins by triggering a module input port.

Execution of the system can be partitioned into module iterations based on component execution.

Definition 3 (Component execution). A source component is activated when the hardware it encapsulates receives an interrupt. A triggered or called component C is activated when one of its methods (in *ACCEPTS$_C$*) is called. *Component execution* is the execution of the code in the body of the interrupt service routine or method through which the component has been activated.

Note that the code executed upon component activation may call other methods in the same component or in a linked component. Component execution also includes execution of all external code until control returns and execution of the code body has completed.

Definition 4 (Module iteration). An *iteration* of a module M is the execution of a subset of the components inside of M in response to either an interrupt or an event.

We define these two types of module iterations in more detail, including what we mean by "subset of components".

Definition 4.1 (Module iteration in response to an interrupt). Suppose module M is iterated in response to interrupt I. Let C be the component corresponding to interrupt I. Recall from Section 2.3.2 that C therefore must be a source component. Create a source DAG D by starting with C and following all forward links between C and other components in M. Iteration of the module consists of the execution of the components in D beginning with C. Note that iteration of the module may cause it to produce one or more events on its output port(s).

Definition 4.2 (Module iteration in response to an event). Suppose module M is iterated in response to an event E stored at the head of one of its inport queues, Q. Let C be the component corresponding to Q. Recall from Section 2.3.2 that C therefore must be a triggered component. Create a triggered DAG D by starting with C and following all forward links between C and other components in M. Iteration of the module consists of the execution of the components in D beginning with C. As with the interrupt case, iteration of the module may cause it to produce one or more events on its output port(s).

We can now discuss how to choose the module iteration order.

Definition 5 (System execution). Given a system state and zero or more interrupts, *system execution* is the iteration of modules until the system reaches a quiescent state. The order in which modules are executed is the same as the order of events in the global event queue.

Conditions for well-formedness Here, we summarize the conditions for well-formedness of a system, as discussed in Section 2.3.

1. Source (interrupt-driven) components must only have outputs, they may not have inputs. In other words, source components may not also be triggered components (triggered by an event on a module input port) nor called components (called by other components).

2. Cycles among components within a module are not allowed, but loops around modules are allowed.

3. Within a module, component source DAGs and triggered DAGs must be disconnected.

4. Within a module, component source DAGs must not be connected to other source DAGs, but triggered DAGs may be connected to other triggered DAGs. We assume that an interrupt whose handler is running is masked, but other interrupts are not masked.

5. Within a module, outgoing component methods may be associated with either one method of another component, or with one or more module output ports.

6. Within a module, module input ports may be associated with either one method of a single component, or with one or more module output ports.

3.1 Determinacy

Given the definitions in the previous section, we first discuss determinism of a TinyGALS system in the case of a single interrupt when in a quiescent state. We then discuss determinism for one or more interrupts during module iteration in the cases where there are no global variables and when there are global variables.

26

In our intuitive notion of determinacy, given an initial quiescent system state and a set of interrupts that occur at known times, the system will always produce the same outputs and end up in the same state after responding to the interrupts.

Theorem 1 (Determinacy). *A system is determinate if for each quiescent state and a single interrupt, there is only one system execution path.*
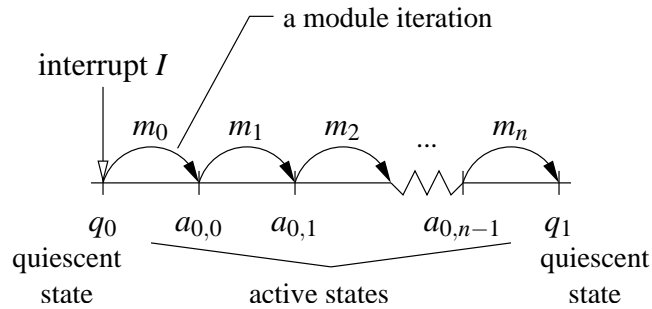

Figure 13: Single interrupt.

Recall that a TinyGALS system starts in an active system state. The application start port is a module input port which is in turn linked to a component C inside the module. The component C is a triggered component, which is part of a DAG. Components in this triggered DAG execute and may generate events at the output port(s) of the module. System execution proceeds until the system reaches a quiescent state. From this quiescent state, we can analyze the determinacy of a TinyGALS system.

Figure 13 depicts iteration of a TinyGALS system between two quiescent states due to activation by an interrupt I_0. A TinyGALS system is determinate, since the system execution path is the order in which the modules are iterated, and in each of the steps $m_0, m_1, \ldots, m_n$, the module selected is determined by the order of events in the global event queue.

What if we consider the case in which we have one or more interrupts *during* a module iteration, that is, between quiescent states, as is usually true in an event-driven system?

27

Determinacy of a system without global variables. We will first examine the case where there are no TinyGUYS global variables.

Let us consider a module M that contains a component C which produces events on the outports of M. Suppose the iteration of module M is interrupted one or more times. Since source DAGs must not be connected to triggered DAGs, the interrupt(s) cannot cause the production of events on outports of M that would be used in the case of a normal uninterrupted iteration. However, the interrupt(s) may cause insertion of events into other module inport queues, and hence insertions into the global event queue. Depending on the relative timing between the interrupts and the production of events by C at outports of M, the order of events in the global event queue may not be consistent between multiple runs of the system if the same interrupts occur during the same module iteration, but at slightly different times. This is a source of non-determinacy.

If we wish to reduce the non-determinacy in the system, a partial solution is to delay producing outputs from the module being iterated until the end of its iteration. If we know the order of interrupts, then we can predict the state of the system after a single module iteration even if it is interrupted one or more times. Figure 14 shows a system execution in which a single module iteration is interrupted by multiple interrupts. In our notation, $a^i_{j,k}$ refers to an active system state after an interrupt I_i starting from quiescent state q_j and after module iteration m_k. In Figure 14, the superscript in $a^x_{j,k}$ is a shorthand for the sequence of interrupts $I_0, I_1, I_2, \ldots, I_n$.
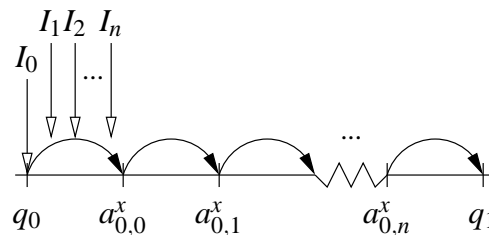

Figure 14: One or more interrupts where modules have delayed output.

In order to determine the value of active system state $a^x_{j,k}$, we can "add" the combined system states. Suppose active state $a^1_{0,0}$ would be the next state after an iteration of the

28

module corresponding to interrupt I_1 from quiescent state q_0, and that active state $a_{0,0}^2$ would be the next state after an iteration of the module corresponding to interrupt I_2 from q_0. This is illustrated in Figure 15.
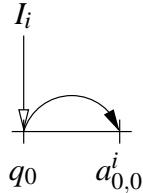

Figure 15: Active system state after one interrupt.

We assume that the handlers for interrupts $I_1, I_2, \ldots, I_n$ execute quickly enough such that they are not interleaved (e.g., I_2 does not interrupt the handling of I_1). Then the system state before the iteration of module M in response to interrupt I_0 has completed but after interrupts I_1 and I_2 would be $a_{0,0}^1 + a_{0,0}^2$, where the value of this expression is the system state in which the new events produced in active system state $a_{0,0}^2$ are inserted (or "appended") into the corresponding module inport queues in active system state $a_{0,0}^1$. We can extend this to any finite number of interrupts, I_n, as shown in Figure 16. It is necessary that the number of interrupts be finite for liveness of the system. From the performance perspective, it is also necessary that interrupt handling be fast enough that the handling of the first interrupt I_0 completes in a reasonable length of time. If the interrupts are interleaved, we must add the system state (append module inport queue contents) in the order in which the interrupt handlers finish.

Another solution, which leads to greater predictability in the system, is to preschedule module iterations. That is, if an interrupt occurs, a sequence of module iterations is scheduled and executed, during which interrupts are masked. One can also queue interrupts in order to eliminate preemption. Then, system execution is deterministic for a fixed sequence of interrupts. However, both of these approaches reduces the reactiveness of the system.

Determinacy of a system with global variables. Let us now discuss system determinacy in the case where there are TinyGUYS global variables ($GLOBALS_A$).
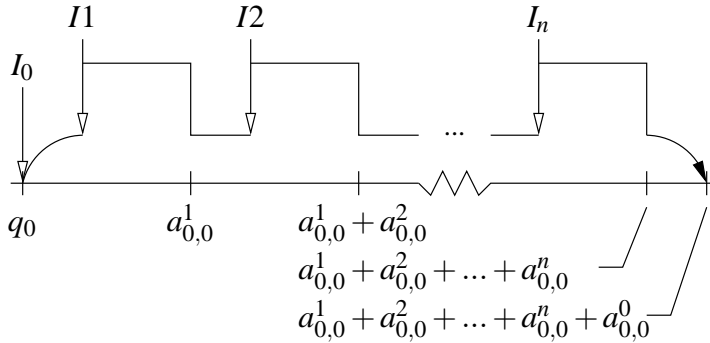
29

Figure 16: Active system state determined by adding the active system state after one non-interleaved interrupt.

Suppose that module M writes to a global variable. Also suppose that the iteration of module M is interrupted, and a component in the interrupting source DAG writes to the same global variable. Then without timing information, we cannot predict the final value of the global variable at the end of the iteration. (Note that when read, a global variable always contains the same value throughout an entire module iteration). As currently defined, the state of the system after the iteration of module M is interrupted by one or more interrupts is highly dependent on the time at which the components in M write to the global variable(s). There are several possible alternatives eliminate this source of nondeterminacy.

Solution 1 Allow only one writer for each TinyGUYS global variable.

Solution 2 Allow multiple writers, but only if they can never write at the same time. That is, if a component in a triggered DAG writes to a TinyGUYS global variable, no component in any source DAG can be a writer (but components in other triggered DAGs are allowed since they cannot execute at the same time). Likewise, if a component in a source DAG writes to a TinyGUYS global variable then no component in any triggered DAG can be a writer. Components in other source DAGs are only allowed to write if all interrupts are masked.

Solution 3 Delay writes to a TinyGUYS global variable by an iterating module until the end of the iteration.

30

Solution 4 Prioritize writes such that once a high priority writer has written to the TinyGUYS global variables, lower priority writes will be lost.

Summary   We have just discussed system determinacy in the case of a single interrupt and determination of system state when a module iteration is interrupted by one or more interrupts.

If we do not use one of the solutions presented, the system state at the completion of an interrupted module iteration depends on the timing of the interrupts. Without prescheduling module iterations, the next quiescent system state after an interrupted module iteration is also highly dependent on module execution times and the timing of the interrupts.

However, event-driven systems are usually designed to be reactive. In these cases, interrupts should be considered as high priority events which *should* affect the system state as soon as possible.

4   Code Generation

Given the highly structured architecture of the TinyGALS model, code for scheduling and event handling can be automatically generated to release software developers from writing error-prone concurrency control code. We have created a set of code generation tools for the MICA motes that, given the definitions for the components, modules, and application, will automatically generate all the code necessary for (1) component links and module connections, (2) system initialization and start of execution, (3) communication between modules, and (4) TinyGUYS global variable reads and writes.

In this section, we also give an overview of the implementation of the TinyGALS scheduler and how it interacts with TinyOS, as well as data on the memory usage of TinyGALS.
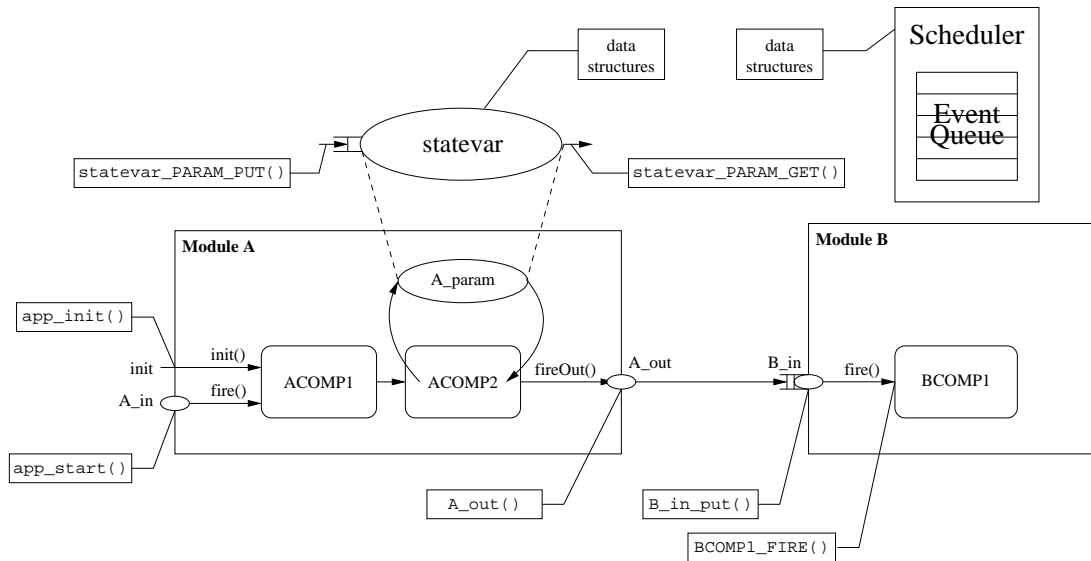
Figure 17: A code generation example.

Throughout this section, we will use the example system illustrated in Figure 17. This system is composed of two modules: A and B. Module A contains two components, ACOMP1 and ACOMP2. Component ACOMP2 accesses a parameter and has a definition similar to the definition of component A1 given in Figure 11. Module B contains a single component named BCOMP1, which has a function called fire() that is connected to the input port B_in of module B. Figure 17 also summarizes all functions and data structures generated by the code generator.

4.1   Links and Connections

The code generator will generate a set of aliases which create the maps for the links between components, as well as the maps for the connections between modules. In the example, an alias will be generated for the link between ACOMP1 and ACOMP2, and for the connection between modules A and B.

4.2   System Initialization and Start of Execution

The code generator will create a system-level initialization function called `app_init()`, which contains calls to the *INIT* methods of each module in the system. The order of modules listed in the system definition determines the order in which the *INIT* methods will be called. The `app_init()` function is one of the first functions called by the TinyGALS runtime scheduler before executing the application. In Figure 17, only module `A` contains an initialization list. Therefore, the generated `app_init()` function will only contain a call to `ACOMP1:init`.

The code generator will also create the application start function, `app_start()`. This function will trigger the input port of the module defined as the system start port. In the example, `app_start()` will contain a trigger for the input port `A_in` of module `A`.

4.3   Communication

The code generator will automatically generate a set of scheduler data structures and functions for each connection between modules.

For each input port of a module, the code generator will generate a queue of length `n`, where `n` is specified in the application definition file. The width of the queue depends on the number of arguments to the method that is connected to the port. If there are no arguments, then as an optimization, no queue is generated for the port (but space is still reserved for events in the scheduler event queue). A pointer and a counter are also generated for each input port to keep track of the location and number of tokens in the queue. For the example in Figure 17, the definition of port `B_in` will result in the generation of a port queue called `B_in_0[]` (assuming port `B_in` is linked to a function with a single argument), as well as the generation of `B_in_head` and `B_in_count`.

For each output port of a module, the code generator will generate a function that has

33

the same name as the output port. This outport function is called whenever a method of a component wishes to write to an output port. The type signature of the outport function matches that of the method that connects to the port. For each input port connected to the outport, a `put()` function is generated which handles the actual copying of data to the inport queue. The `put()` function also adds the port identifier to the scheduler event queue so that the scheduler will activate the module at a later time. The outport function calls the appropriate `put()` function for each connected inport. In the example of Figure 17, for the outport `A_out` of module `A`, a function `A_out()` is generated, which in turn calls the generated function `B_in_put()` to insert data into the queue.

If the queue is full when attempting to insert data into the queue, there are several strategies that can be taken. We currently take the simple approach of dropping events that occur when the queue is full. However, an alternate method is to generate a callback function which will attempt to re-queue the event at a later time. Yet another approach would be to place a higher priority on more recent events by deleting the oldest event in the queue to make room for the new event.

For each connection between a component method and a module input port, a function will be generated with a name formed from the name of the input port and the name of the component method. When the scheduler activates a module via an input port, it first calls this generated function to remove data from the input port queue and pass it to the component method. In Figure 17, module `B` contains an input port `B_in`, which is connected to the `fire()` method of component `BCOMP1`. The code generator will create a function called `BCOMP1_FIRE()`, which removes data queued in `B_in_0[]`, modifies `B_in_head` and `B_in_count`, and calls `BCOMP1:fire`.

4.4 TinyGUYS

The code generator will generate a pair of data structures and a pair of access functions for each TinyGUYS global variable declared in the system definition. The pair of data

structures consists of a data storage location of the type specified in the module definition that uses the global variable, along with a buffer for the storage location. The pair of access functions consists of a `PARAM_GET()` function that returns the value of the global variable, and a `PARAM_PUT()` function that stores a new value for the variable in the variable's buffer. A generated flag indicates whether the scheduler needs to update the variables by copying data from the buffer.

In the example of Figure 17, a global variable named `params.statevar` will be generated, along with a buffer named `params_buffer.statevar`. The code generator will also create functions `statevar_PARAM_GET()` and `statevar_PARAM_PUT()`.

4.5   Scheduling

Execution in the system begins in the scheduler, which performs all of the runtime initialization.[11] Figure 18 describes the TinyGALS scheduling algorithm. There is a single scheduler in TinyGALS which checks the global event queue for events. When there is an event, the scheduler first copies buffered values of into the actual storage for any modified TinyGUYS global variables. The scheduler removes the token corresponding to the event from the appropriate module input port and passes the value of the token to the component method linked to the input port. When there are no events in the global event queue, then the scheduler runs any posted TinyOS tasks. When there are no events or TinyOS tasks, then the system goes to sleep. In summary, the TinyGALS scheduler is a two-level scheduler. TinyGALS modules run at the highest priority, and TinyOS tasks run at the lowest priority.

---

[11]In TinyGALS, we eliminate the `MAIN` component of TinyOS and instead place this functionality in the initialization sequence of the scheduler.

> **if** there is an event in the global event queue **then** {
>> **if** any TinyGUYS have been modified
>>> Copy buffered values into variables.
>>
>> **end if**
>> Get token corresponding to event out of input port.
>> Pass value to the method linked to the input port.
>
> **else if** there is a TinyOS task **then** {
>> Take task out of task queue.
>> Run task.
>
> **end if**

Figure 18: TinyGALS scheduling algorithm.

4.6   Memory Usage

Tables 1 and 2 show the sizes of the generated functions and variables for the system shown in Figure 17. Note that the system contains one port. Here, we assume that the method linked to `A_out` writes values of type `short` and returns a confirmation value of type `char`. Additionally, the queue connected to inport `B_in` is of size 50 (i.e., it holds 50 elements of type `short`). Thus, memory usage of a TinyGALS application is determined mainly by the user-specified queue sizes and total number of ports in the system. The Tiny-GALS communication framework is very lightweight, since event queues are generated as application-specific data structures.

Table 3 compares the sizes of the TinyGALS and original TinyOS portions of the scheduler. The code size of the TinyGALS scheduler is only 114 bytes. For backwards compatibility with TinyOS tasks, we include the original 86 byte TinyOS scheduler. If TinyOS tasks are not used, the scheduler is about the same size as before. Refer to Section 6.1 on why TinyOS tasks are made obsolete by the TinyGALS model.

---

[12]The queue holds 50 elements of type `short`.

Table 1: Sizes of generated functions

Function Name	Bytes of code (decimal)
app_init()	58
app_start()	6
A_out()	12
B_in_put()	160
BCOMP1_FIRE()	98
A_param_PARAM_GET()	10
A_param_PARAM_PUT()	16

Table 2: Sizes of generated variables

Variable Name	Bytes (decimal)
eventqueue_head	2
params	2
entrypoints	2
eventqueue_count	2
eventqueue [12]	100
ports [12]	104
params_buffer_flag	1
params_buffer	2

Table 3: Size of scheduler (number of bytes in decimal)

TinyGALS	114
TinyOS	86
Total	200

5 Example

The TinyGALS programming model and code generation tools have been implemented for the Berkeley motes [21], a wireless networked sensor platform, and are compatible with existing TinyOS v0.6.1 components [5]. In this section, we give an example of the redesign of a multi-hop *ad hoc* communication protocol known as the beaconless protocol, or BLESS.

The specific generation of motes, called *MICA*, that we used in this example has a 4MHz ATMEGA103L microcontroller, 128KB programming memory, 4KB data memory, 5 types of sensors, and an RF communication component. MICA motes operate off of two AA batteries. BLESS is a single base multi-hop protocol, where a single base station collects information from a distributed wireless sensor network. BLESS dynamically builds a routing tree rooted at the base station. A sensor node finds its *parent* node in the routing tree by listening to the network traffic.

(a) All nodes that can hear the base station label themselves as one hop from the base.

(b) When a one-hop node sends a message, unlabeled nodes that overhear this message will label themselves as two-hop nodes.

(c) Three-hop nodes label themselves.

Figure 19: Illustration of the BLESS protocol.

The base station periodically sends a beacon message. Those nodes that directly receive

38

this message will label themselves one hop from the base, as in Figure 19(a). When a one-hop node sends an information message, some nodes that did not hear the base will overhear the one-hop node's message, as shown in Figure 19(b). These nodes will label themselves two-hop nodes and will remember to send their information message to the one-hop node they just overheard, which is now their parent node The one-hop node will forward any packets destined for itself to the base. After a two-hop node has sent an information message the process will repeat and three-hop nodes will be designated, and so on, as illustrated in Figure 19(c). Thus the information messages destined for the base also function as routing beacon messages. A timeout clock is used to re-initialize the parent list to empty, which allows nodes to tolerate unreliable links as well as nodes that join and leave the network dynamically

From a single node point of view, the BLESS protocol responds to three types of input events – information messages sent by the local application, network messages overheard or to be forwarded, and the timeout event to reset the parent variables. The three threads of reaction are intertwined. For example, while a node is busy forwarding a message from its child node, its own application may try to send a message and may modify the message buffer. It is also possible that, while a node sends an information message, an overheard message may arrive, which will change the node's parent and in turn the hop number of this node.

The BLESS protocol is implemented in TinyOS v0.6.1 as a single monolithic component with eight method call interfaces that handle all three threads of reaction internally. All methods are non-reentrant. To avoid concurrent modification of message buffers, the component makes extensive use of a *pending* lock. Six of the eight methods contain code that adjusts behavior accordingly after setting and/or checking the pending lock. Even so, due to the complexity of managing message queues, the queuing of events is left to application designers who use this component.

Using the TinyGALS model, we redesigned and implemented the BLESS protocol on

the motes as three separate modules: `BLESS_Start`, `BLESS_Receive`, and `BLESS_Send`[13], each of which only deals with a single thread of reaction. In addition, there are a number of global variables that use the TinyGUYS mechanism and are accessed from within at least two of the modules. This structure is displayed in Figure 20.
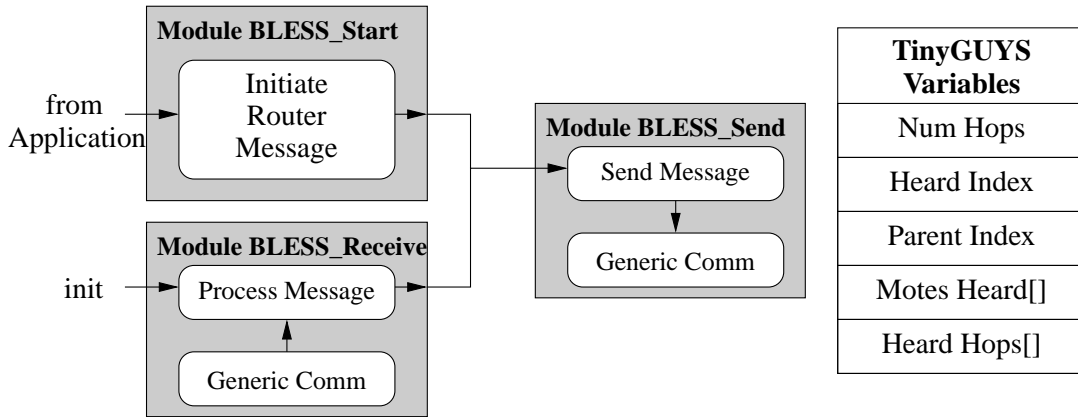

Figure 20: Structure of BLESS implementation under TinyGALS. Each shaded rectangle represents a module. The guarded global variables are shown on the right.

The `BLESS_Start` module contains a single component that prepares an outgoing message initiated by the local application. The input port of `BLESS_Start` is connected to this application module and the output port is connected to the sending component within `BLESS_Send`. The `BLESS_Start` module reads the guarded global variables to determine the current parent node and set up the outgoing message.

The `BLESS_Receive` module is comprised of two components: `Generic Comm`[14] receives messages from the radio, and `Process Message` processes those messages. The output port of `BLESS_Receive` is connected to the sending component within `BLESS_Send` to pass messages that must be forwarded to the base. The `BLESS_Receive` module accesses the guarded global variables both in order to update them and to use them when forwarding a message.

---

[13]Resetting the parent is done within a separate module triggered directly by the clock.

[14]`Generic Comm` is the same as `GENERIC_COMM`, an off-the-shelf TinyOS component that implements the network stack.

The `BLESS_Send` module is comprised of two components, one for reformatting messages, `Send Message`, and the second for sending messages over the radio, `Generic Comm`. No global variables are used.

Table 4 shows the sizes of the redesigned BLESS implementation, as well as the original BLESS implementation. For fair comparison, we only account for the sizes of the components inside of the modules. The last row of the table shows the total application sizes for two implementations of a target counting application which uses each BLESS implementation.

From the table, we can deduce that 1170 (3366 - 2196 = 1170) additional bytes of code appear in the redesigned BLESS implementation. This difference is a 53.28% (1170/2196) increase in code size from the original implementation. However, this difference contributes only 9.156% (1170/12778) to the total code size of the new application. TinyGALS sacrifices some space in memory for improved software modularity and code reuse. Applications that would take one week to implement now take us only 3 days to implement.

In the above example, the `BLESS_Send` module is conveniently used by both of the other two modules to send the specialized router message. Additionally, we avoid potential problems with shared state and interrupts through the use of TinyGUYS global variables. TinyGUYS simplifies the code writing and debugging process by eliminating the need for explicit locks. TinyGALS also provides easy management of message queues. In the TinyGALS architecture of the above example, when a message is to be sent through `Send Message`, a request is queued at the input port to the `BLESS_Send` module. When processing time is available, `BLESS_Send` will run after dequeuing the messages from its input port. This same call in the existing TinyOS implementation would result in the message being sent immediately, or, if busy, not at all, due to its usage of a single pending lock. Use of the TinyGALS programming model leads to programs with better code structure and improved communication network reliability.

41

Table 4: Code size comparison of BLESS implementations (bytes in decimal)

redesigned BLESS		original BLESS	
`BLESS_Start`	540	`BLESS_COMP`	2196
`BLESS_Receive`	2730		
`BLESS_Send`	96		
Total	3366		
New Application	12778	Old Application	10582

6   Related Work

In this section, we summarize the features of several operating systems and/or software architectures and discuss how they relate to TinyGALS. We begin with TinyOS, which is closely tied with the component model of TinyGALS. We then discuss PBOs (port-based objects) and FPBOs (featherweight port-based objects), whose SVAR (state variable) mechanism influenced the design of TinyGUYS. We discuss the Click Modular Router project, which has interesting parallels to the TinyGALS model of computation. We also discuss Ptolemy II and the CI (component interaction) domain, as well as TM (Timed Multitasking), which are areas that we plan to explore in more detail in extensions to the existing TinyGALS model and implementation.

6.1   TinyOS

TinyOS is an operating system specifically designed for networked embedded systems. [20] describes TinyOS in detail. TinyOS provides a set of reusable software *components*. A TinyOS application connects components using a *wiring specification* that is independent of component implementation. Some TinyOS components are thin wrappers around hardware, although most are software modules which process data; the distinction is invisible to the developer. Decomposing different OS services into separate components allows unused services to be excluded from the application.

There are two sources of concurrency in TinyOS: *tasks* and *events*. Tasks are a deferred computation mechanism. Components can *post* tasks; the post operation returns immediately, deferring the computation until the scheduler executes the task later. Tasks run to completion and do not preempt each other. Events signify either an event from the environment or completion of a *split-phase* operation. Split-phase operations are long-latency operations where operation request and completion are separate functions. *Commands* are typically requests to execute an operation. If the operation is split-phase, the command returns immediately and completion will be signaled with an event; non-split-phase operations do not have completion events. Events also run to completion, but may preempt the execution of a task or another event. Resource contention is typically handled through explicit rejection of concurrent requests. Because tasks execute non-preemptively, TinyOS has no blocking operations. TinyOS execution is ultimately driven by events representing hardware interrupts.

In Section 1, we introduced TinyOS and its influence on the TinyGALS model. Later sections and footnotes contain details on the ties between TinyOS and TinyGALS.

In TinyOS, many components such as `PHOTO` and other wrappers for device drivers are "split phase", which means that they are actually both source and triggered components. A higher level component will call the device driver component to ask for data. This call will return immediately. Later, the device driver component will interrupt with the ready data. The hidden source aspect of these types of components may lead to TinyOS configurations with race conditions or other synchronization problems. Although the TinyOS architecture allows components to reject concurrent requests, it is up to the software developer to write thread-safe code. This job is actually quite difficult, especially after components are wired together and may have interleaved events. In Sections 2.3 and 3, we showed how the TinyGALS component model enables users to analyze potential sources of concurrency problems more easily by identifying source, triggered, and called components and defined what kinds of links between components are valid.

The TinyGALS programming model removes the need for TinyOS tasks. Both trig-

gered modules in TinyGALS and tasks in TinyOS provide a method for deferring computation. However, TinyOS tasks are not explicitly defined in the interface of the component, so it is difficult for a developer wiring off-the-shelf components together to predict what non-interrupt driven computations will run in the system. In TinyOS tasks must be short; lengthy operations should be spread across multiple tasks. However, since there is no communication between tasks, the only way to share data is through the internal state of a component. The user must write synchronization code to ensure that there are no race conditions when multiple threads of execution access this data. TinyGALS modules, on the other hand, allow the developer to explicitly define "tasks" at the application level, which is a more natural way to write applications. The asynchronous and synchronous parts of the system are clearly separated to provide a well-defined model of computation, which leads to programs that are easier to debug. The globally asynchronous nature of TinyGALS provides a way for tasks to communicate. The developer has no need to write synchronization code when using TinyGUYS to share data between tasks; the code is automatically generated by the TinyGALS tools.

6.2   Port-Based Objects

The port-based object (PBO) [35] is a software abstraction for designing and implementing dynamically reconfigurable real-time software. The software framework was developed for the Chimera multiprocessor real-time operating system (RTOS). A PBO is an independent concurrent process, and there is no explicit synchronization with other processes. PBOs may execute either periodically or aperiodically. A PBO communicates other PBOs only through its *input ports* and *output ports*. *Configuration constants* are used to reconfigure generic components for use with specific hardware or applications. PBOs may also have *resource ports* that connect to sensors and actuators via I/O device drivers, which are not PBOs.

Communication between PBOs is performed via state variables stored in global and local tables. Every input and output port and configuration constant is defined as a state

variable (SVAR) in the global table, which is stored in shared memory. A PBO can only access its local table, which contains only the subset of data from the global table that is needed by the PBO. Since every PBO has its own local table, no synchronization is needed to read from or write to it. Consistency between the global and local tables is maintained by the SVAR mechanism, and updates to the tables only occur at predetermined times. Configuration constants are updated only during initialization of the PBO. The state variables corresponding to input ports are updated prior to executing each cycle of a periodic PBO, or before processing each event for an aperiodic PBO. During its cycle, a PBO may update the state variables corresponding to output ports at any time. These values are only updated in the global table after the PBO completes its processing for that cycle or event. All transfers between the local and global tables are performed as critical sections. Although there is no explicit synchronization or communication among processes, accesses to the same SVAR in the global table are mutually exclusive, which creates potential implicit blocking. Spin-locks are used to lock the global table, and it is assumed that the amount of data communicated via the ports on each cycle of a PBO is relatively small. It is guaranteed that the task holding the global lock is on a different processor and will not be preempted, thus it will release the lock shortly. If the total time that a CPU is locked to transfer a state variable is small compared to the resolution of the system clock, then there is negligible effect on the predictability of the system due to this mechanism locking the local CPU. Since there is only one lock, there is no possibility of deadlock. A task busy-waits with the local processor locked until it obtains the lock and goes through its critical section.

Echidna [2] is a real-time operating system designed for smaller, single-processor, embedded microcontrollers. The design is based on the *featherweight port based object* (FPBO) [34]. The application programmer interface (API) for the FPBO is identical to that of the PBO. In an RTOS, PBOs are separate processes, whereas FPBOs all share the same context. The Chimera implementation uses data replication to maintain data integrity and avoid race conditions. Echidna implementation takes advantage of context sharing to eliminate the need for local tables, which is especially important since memory in embedded processors is a limited resource. Access to global data must still be performed as a

critical section to maintain data integrity. However, instead of using semaphores, Echidna constraints when preemption can occur.

To summarize, in both the PBO and FPBO model, the software components only communicate with other components via SVARs, which are similar to global variables. Updates to an SVAR are made atomically, and the components always read the latest value of the SVAR. The SVAR concept is the motivation behind the TinyGUYS strategy of always reading the latest value. However, in TinyGALS, since components within a module may be tightly coupled in terms of data dependency, updates to TinyGUYS are buffered until a module has completed execution. This is more closely related to the local tables in the Chimera implementation than the global tables in the Echidna implementation. However, there is no possibility of blocking when using the TinyGUYS mechanism.

6.3   Click

Click [25, 24] is a flexible, modular software architecture for creating routers. A Click router configuration consists of a directed graph, where the vertices are called *elements* and the edges are called *connections*. In this section, we provide a detailed description of the constructs and processing in Click and compare it to TinyGALS.

Elements in Click   An element is a software module which usually performs a simple computation as a step in packet processing. An element is implemented as a C++ object that may maintain private state. Each element belongs to one *element class*, which specifies the code that should be executed when the element processes a packet, as well as the element's initialization procedure and data layout. An element can have any number of input and output *ports*. There are three types of ports: *push*, *pull*, and *agnostic*. In Click diagrams, push ports are drawn in black, pull ports in white, and agnostic ports with a double outline. Each element supports one or more *method interfaces*, through which they communicate at runtime. Every element supports the simple packet-transfer interface,

46

but elements can create and export arbitrary additional interfaces. An element may also have an optional *configuration string* which contains additional arguments that are passed to the element at router initialization time. The Click configuration language allows users to define *compound elements*, which are router configuration fragments that behave like element classes. At initialization time, each use of a compound element is compiled into the corresponding collection of simple elements.

Figure 21 shows an example Click element that belongs to the *Tee* element class, which sends a copy of each incoming packet to each output port. The element has one input port. It is initialized with the configuration string "2", which in this case configures the element to have two output ports.

Figure 21: An example Click element.

Connections in Click   A connection represents a possible path for packet handoff and attaches the output port of an element to the input port of another element. A connection is implemented as a single virtual function call. A connection between two push ports is a *push connection*, where packet handoff along the connection is initiated by the source element (or source end, in the case of a chain of push connections). A connection between two pull ports is a *pull connection*, where packet handoff along the connection is initiated by the destination element (or destination end, in the case of a chain of pull connections). An agnostic port behaves as a push port when connected to push ports and as a pull port when connected to pull ports, but each agnostic port must be used as push or pull exclusively. In addition, if packets arriving on an agnostic input might be emitted immediately on an agnostic output, then both input and output must be used in the same way (either push or pull). When a Click router is initialized, the system propagates constraints until every agnostic port has been assigned to either push or pull. A connection between a push port

47

and a pull port is illegal. Every push output and every pull input must be connected exactly once. However, push inputs and pull outputs may be connected more than once. There are no implicit queues on input and output ports (or the associated performance and complexity costs). Queues in Click must be defined explicitly and appear as *Queue* elements. A *Queue* has a push input port (responds to pushed packets by enqueueing them) and a pull output port (responds to pull requests by dequeuing packets and returning them).

Another type of element is the Click packet scheduler. This is an element with multiple pull inputs and one pull output. The element reacts to requests for packets by choosing one of its inputs, pulling a packet from it, and returning the packet. If the chosen input has no packets ready, the scheduler will usually try other inputs. Both *Queues* and scheduling elements have a single pull output, so to an element downstream, *Queues* and schedulers are indistinguishable. This leads to an ability to create virtual queues, which are compound elements that act like queues but implement more complex behavior than FIFO queuing.

Click runtime system   Click runs as a kernel thread inside the Linux 2.2 kernel. The kernel thread runs the Click router driver, which loops over the task queue and runs each task using stride scheduling [36]. A *task* is an element that would like special access to CPU time. An element should be on the task queue if it frequently initiates push or pull requests without receiving a corresponding request. Most elements are never placed on the task queue; they are implicitly scheduled when their `push()` or `pull()` methods are called. Since Click runs in a single thread, a call to `push()` or `pull()` must return to its caller before another task can begin. The router will continue to process each pushed packet, following it from element to element along a path in the router graph (a chain of `push()` calls, or a chain of `pull()` calls), until it is explicitly stored or dropped (and similarly for pull requests). Placement of *Queues* in the configuration graph determines how CPU scheduling may be performed. For example, device-handling elements such as *FromDevice* and *ToDevice* place themselves on Click's task queue. When activated, *FromDevice* polls the device's receive DMA queue for newly arrived packets and pushes them through the configuration graph. *ToDevice* examines the device's transmit DMA queue for empty slots

48

and pulls packets from its input. Click is a pure polling system; the device never interrupts the processor.

Timers are another way of activating an element besides tasks. An element can have any number of active timers, where each timer calls an arbitrary method when it fires. Timers are implemented using Linux timer queues.
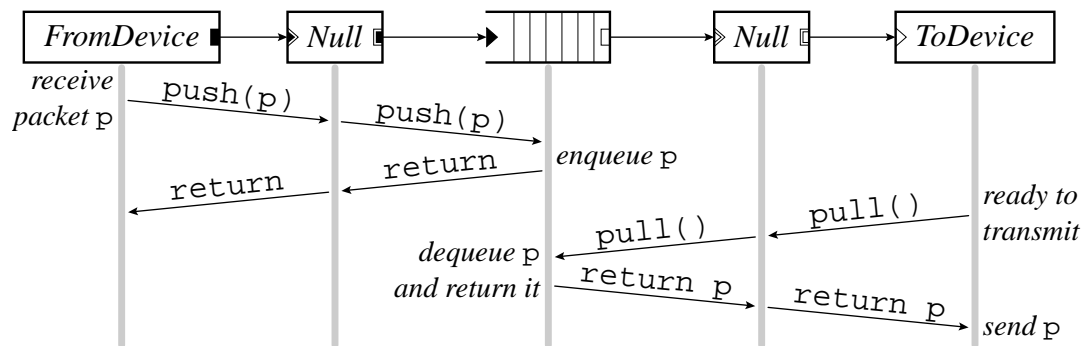

Figure 22: A simple Click configuration with sequence diagram.

Figure 22 shows a simple Click router configuration with a push chain (*FromDevice* and *Null*) and a pull chain (*Null* and *ToDevice*). The two chains are separated by a *Queue* element. The *Null* element simply passes a packet from its input port to its output port; it performs no processing on the packet. Note that in the sequence diagram in Figure 22, time moves downwards. Control flow moves forward during a push sequence, and moves backward during a pull sequence. Data flow (in this case, the packet p) always moves forwards.

Figure 23 illustrates the basic execution sequence of Figure 22. When the task corresponding to *FromDevice* is activated, the element polls the receive DMA ring for a packet. *FromDevice* calls push() on its output port, which calls the push() method of *Null*. The push() method of *Null* calls push() on its output port, which calls the push() method of the *Queue*. The *Queue* element enqueues the packet if its queue is not full; otherwise it drops the packet. The calls to push() then return in the reverse order. Later, the task corresponding to *ToDevice* is activated. If there is an empty slot in its transmit DMA ring, *ToDevice* calls pull() on its input port, which calls the pull() method of *Null*. The
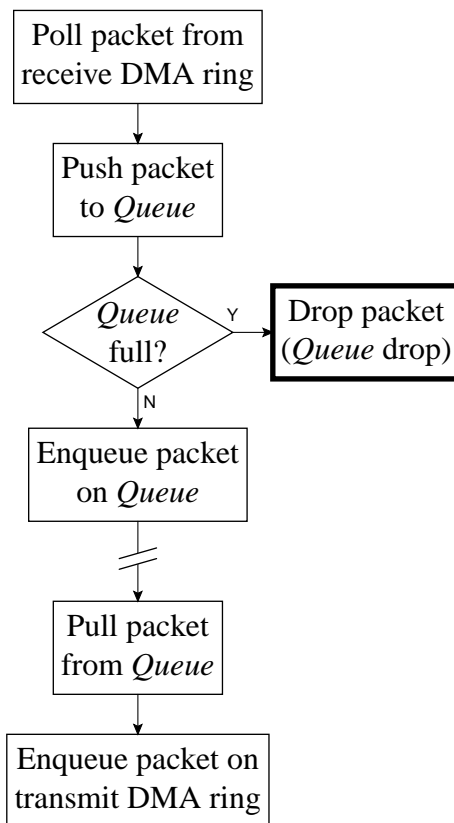
49

Figure 23: Flowchart for Click configuration shown in Figure 22.

`pull()` method of *Null* calls `pull()` on its input port, which calls the `pull()` method of the *Queue*. The *Queue* element dequeues the packet and returns it through the return of the `pull()` calls.

Overhead in Click Modularity in Click results in two main sources of overhead. The first source of overhead comes from passing packets between elements. This causes one or two virtual function calls, which involves loading the relevant function pointer from a virtual function table, as well as an indirect jump through that function pointer. This overhead is avoidable – the Click distribution contains a tool to eliminate all virtual function calls from a Click configuration. The second source of overhead comes from unnecessarily general element code. The authors of [25] found that element generality had a relatively small effect on Click's performance since not many elements in a particular configuration offered much opportunity for specialization.

Comparison of Click to TinyGALS An element in Click is comparable to a component in TinyGALS in the sense that both are objects with private state. Rules in Click on connecting elements together are similar to those for connecting components in TinyGALS – push outputs must be connected exactly once, but push inputs may be connected more than once (see Sections 2.3.3 and 3). Both types of objects (Click elements and TinyGALS components) communicate with other objects via method calls. In Click, there is no fundamental difference between push processing and pull processing in Click at the method call level of view; they are sets of method calls that differ only in name. However, the direction of control flow with respect to data flow in the two types of processing are opposite of each other. Push processing can be thought of as event-driven computation (if we ignore the polling aspect of Click), where control and data flow downstream in response to an upstream event. Pull processing can be thought of as demand-driven computation, where control flows upstream in order to compute data needed downstream.

Figure 24 helps to provide a more detailed analysis of the difference in control and data
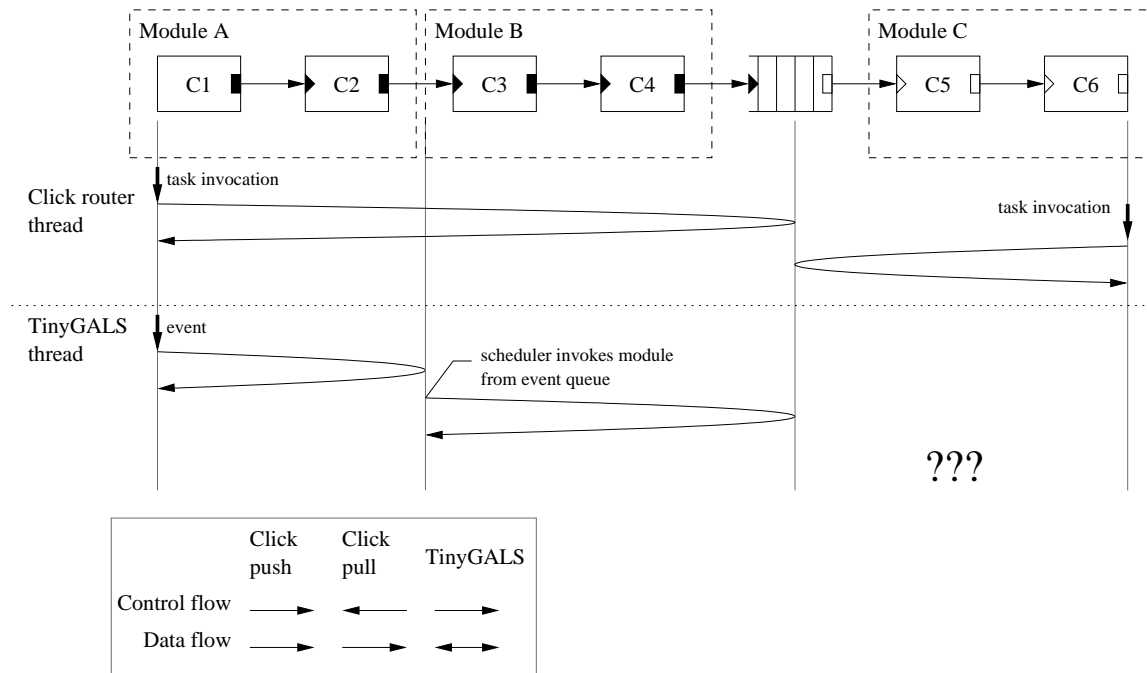
Figure 24: Click vs. TinyGALS.

flow between Click and TinyGALS. Figure 24 shows a push processing chain of four elements connected to a queue, which is connected to a pull processing chain of two elements. In Click, control begins at element `C1` and flows to the right and returns after it reaches the *Queue*. Data (a packet) flows to the right until it reaches the *Queue*. If we visualize this configuration as a TinyGALS model, with elements `C1` and `C2` grouped into a module `A` and elements `C3` and `C4` grouped into a module `B`, then we see that a TinyGALS module forms a boundary for control flow.

Note that a compound element in Click does not form the boundary of control flow. In Click, if an element inside of a compound element calls a method on its output, control will flow to the connected element (recall that a compound element is compiled to a chain of simple elements). In TinyGALS, data flow within a module is not represented explicitly. Data flow between components in a module can have a direction different from the link arrow direction, unlike in Click, where data flow is the same direction as the connection. Data flow between modules is always the same as the connection arrow direction, although

52

TinyGUYS provides a possible hidden avenue for data flow between modules.

Also note that the Click *Queue* element is not equivalent to the queue on a TinyGALS module input port. In Click, arrival of data in a queue does not cause downstream objects to be scheduled, as in TinyGALS. This highlights the fact that Click configurations cannot have two push chains (where the end elements are activated as tasks) separated by a *Queue*. Additionally, since Click is a pure polling system, it does not respond to events immediately, unlike TinyGALS, which is interrupt-driven and allows preemption to occur in order to process events. Much of this is due to the fact that Click's design is motivated by high throughput routers, whereas TinyGALS is motivated by a power- and resource-constrained platform; a TinyGALS system goes to sleep when there are no external events to which to respond.

Aside from the polling/interrupt-driven difference, push processing in Click is equivalent to synchronous communication between components in a TinyGALS module. Pull processing in Click, however, does not have a natural equivalent in TinyGALS. In Figure 24, elements C5 and C6 are grouped into a module C. If we reverse the arrow directions inside of module C, control flow in this new TinyGALS model will be the same as in Click. However, elements C5 and C6 may have to be rewritten to reflect the fact that C6 is now a source object, rather than a sink object.

In Click, execution is synchronous within each push (or pull) chain, but execution is asynchronous between chains, which are separated by a *Queue* element. From this global point of view, the execution model of Click is quite similar to the globally asynchronous, locally synchronous execution model of TinyGALS.

Unlike TinyGALS, elements in Click have no way of sharing global data. The only way of passing data between Click elements is to add annotations to a packet (information attached to the packet header, but which is not part of the packet data).

Unlike Click, TinyGALS does not contain timers associated with elements, although this can be emulated by linking a CLOCK component with an arbitrary component. Also

unlike Click, the TinyGALS model does not contain a task queue.[15]

Pull processing in sensor networks   Although TinyGALS does not currently use pull processing, the following example by Jie Liu given in [42] illustrates a situation in which pull processing is desirable for eliminating unnecessary computation. Figure 25 shows a sensor network application in which four nodes cooperate to detect intruders. Each node is only capable of detecting intruders within a limited range and has a limited battery life. Communication with other nodes consumes more power than performing local computations, so nodes should send data only when necessary. Node A has more power and functionality than other nodes in the system. It is known that an intruder is most likely to come from the west, somewhat likely to come from the south, but very unlikely to come from the east or north. Under these assumptions, node A may want to pull data from other nodes only when needed. Figure 26 shows one possible configuration for this kind of pull processing. The center component is similar to the Click scheduler element. This example also demonstrates a way to perform distributed multitasking. Node D (and others) may be free to perform other computations while node A performs most of the intrusion detection. This could be an extension to the current single-node architecture of TinyGALS.
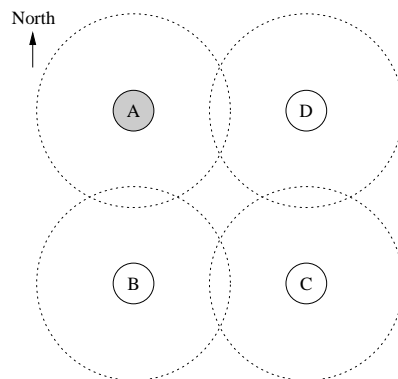

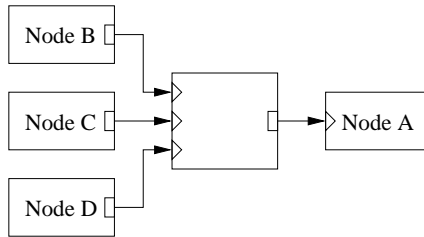Figure 25: A sensor network application.

Figure 26: Pull processing across multiple nodes; a configuration for the application in Figure 25.

6.4 Ptolemy and CI

The Ptolemy project [4] studies heterogeneous modeling, simulation, and design of concurrent systems. Modeling is the act of representing a system or subsystem formally. Executable models are constructed under a model of computation, which is the set of rules that govern the interaction, communication, and control flow of a set of components in the model. The Ptolemy II [11] software package includes implementations of a wide variety of models of computation, called *domains*. An executable entity in Ptolemy II is called an *actor*.

A new domain currently being developed in Ptolemy II, called CI (component interaction), models systems that contain both event-driven and demand-driven styles of computation. It is motivated by the push/pull interaction between data producers and consumers in middleware services such as the CORBA event service. CI actors can be active (i.e., they have their own thread of execution) or passive (they are triggered by an active actor). There is a natural correlation between the CI domain and Click. The MESCAL project [3] has created a tool called Teepee [30], which is based on Ptolemy II and implements the Click model of computation. The Ptolemy II actor library contains a *ClassWrapper* actor, which could be used to model TinyGALS components. We are currently investigating how CI and Click can be leveraged to implement an implementation of TinyGALS in Ptolemy II.

6.5   Timed Multitasking

Timed multitasking (TM) [29] is an event-triggered programming model that takes a time-centric approach to real-time programming but controls timing properties through deadlines and events rather than time triggers.

Software components in TM are called *actors*, due to the implementation of TM in the Ptolemy project. An actor represents a sequence of reactions, where a reaction is a finite piece of computation. Actor have state, which carries from one reaction to another. Actors can only communicate with other actors and the physical world through ports. Unlike method calls in object-oriented models, interaction with the ports of an actor may not directly transfer the flow of control to another actor.

Actors in a TM model declare their computing functionality and also specify their execution requirements in terms of *trigger conditions*, *execution time*, and *deadlines*. An actor is activated when its trigger condition is satisfied. If there are enough resources at run time, then the actor will be granted at least the declared execution time before its deadline is reached. The results of the execution are made available to other actors and the physical world only at the deadline time. In the cases where an actor cannot finish by its deadline, the TM model includes an overrun handler to preserve the timing determinism of all other actors and allow the actor that violates the deadline to come to a quiescent state.

A trigger condition can be built using real-time physical events, communication packets, and/or messages from other actors. Triggers must be *responsible*, which means that once triggered, an actor should not need any additional data to complete its finite computation. Therefore, actors will never be blocked on reading. The communication among the actors has an event semantics, in which, unlike state semantics, every piece of data will be produced and consumed exactly once. Event semantics can be implemented by FIFO queues. Conceptually, the sender of a communication is never blocked on writing.

[29] describes a method for generating the interfaces and interactions among TM actors

into an imperative language like C. There are two types of actors – an *interrupt service routines* (ISR) responds to external events, and a *task* is triggered entirely by events produced by peer actors. These two types do not intersect. In a TM model, an ISR usually appears as a source actor or a port that transfers events into the model. ISRs do not have triggering rules, and outputs are made immediately available as trigger events to downstream actors. An ISR is synthesized as an independent thread. Tasks have a much richer set of interfaces than ISRs and have a set of methods that define the split-phase reaction of a task. The TM runtime system uses an event dispatcher to trigger a task when a new event is received at its port. Events on a connection between two actors are represented by a global data structure, which contains the communicating data, a mutual-exclusion lock to guard the access to the variable if necessary, and a flag indicating whether the event has been consumed.

We suggested in Section 3.1 that a partial method of reducing non-determinacy due to one or more interrupts during a module iteration is to delay producing outputs from a module until the end of its iteration. This is similar to the TM method of only producing outputs at the end of an actor's deadline.

7 Conclusion

This report describes the TinyGALS programming model for event-driven multitasking embedded systems. The globally asynchronous, locally synchronous model allows software designers to use high-level constructs such as ports and message queues to separate the flow of control between modules that contain components composed of method calls. Guarded yet synchronous variables (TinyGUYS) provide a means of exchanging global data between asynchronous modules without triggering reactions. TinyGALS contains no blocking mechanisms in the language constructs, which means there is not potential of deadlock.

We showed that given a single interrupt, the path of a TinyGALS system from one

quiescent system state to the next quiescent system state can be predicted. Under multiple interrupts, we can predict the path of a TinyGALS system between module iterations if constraints are placed on the time at which events are produced at the output ports of a module, as well as on how many writers are allowed for TinyGUYS. Event-driven systems are highly timing-dependent, and we can consider interrupts to be high priority events that should affect the system state as soon as possible. However, the TinyGALS programming model allows us to examine the structure of the system to determine whether interrupts will change the state. For example, in Figure 27, we know that component $C1$ is not a source and that interrupts cannot change its state. Additionally, TinyGALS provides a task-oriented way of designing an application. In TinyOS, tasks are not visible at the user-level. In TinyGALS, the structure of the system makes tasks visible (i.e., tasks correspond to module(s)).
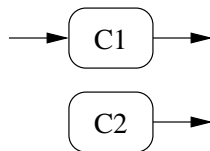
Figure 27: Interrupts cannot change the state of component $C1$.

TinyGALS is designed for applications in which the system must react to events as soon as possible. TinyGALS is also intended for use in resource-constrained systems, since generated TinyGALS code is quite small, and the system is designed to sleep when not reacting to events. TinyGALS is not designed for real-time applications, since there is currently no guarantee on the time it will take an interrupt handler to complete, should it be preempted by non-masked interrupts. TinyGALS and other systems that use a push model of computation are not meant for processing intensive applications that must avoid unnecessary chains of computation, since once activated, components run to completion.

The high-level constructs of the TinyGALS programming model are amenable to automatic code generation that releases designers from writing error-prone task synchronization code. We implemented this model for the Berkeley mote sensor network platform and described a multi-hop communication protocol redesigned using the TinyGALS model.

58

7.1 Future Work

We plan to reimplement the TinyGALS syntax and code generation tools to be compatible with nesC [18], the language written for TinyOS v1.0 and higher. We will then be able to take advantage of additional type checking provided by the nesC compiler. nesC will probably allow us to create a better way of declaring use of TinyGUYS global variables with scoping. David Culler notes that the TinyGALS scheduler could be implemented as a TinyOS task that posts itself and contains code to transfer data between components and activate them. We plan to investigate the feasibility of this approach with respect to the generation of code for TinyGALS module input ports.

We also plan to investigate how TinyGALS relates to the CI domain in Ptolemy II and how this can be leveraged to create a TinyGALS domain or runtime system, similar to that of TM.

Various improvements could be made to the TinyGALS framework. For example, a mechanism for writing to input ports whose queues are full could be added so that writes are blocking and will retry later when the queue is not full. The basic FIFO scheduling of modules in TinyGALS could be replaced with a priority scheduling algorithm with queue insertions.

We also wish to investigate the possibility of run-time reconfigurability of modules – how to replace or reconfigure modules in a TinyGALS system while it is executing. Another important direction of research is distributed multi-tasking, in which TinyGALS could be extended to execute in a distributed fashion on multiple nodes, rather than its current single-node implementation. An interesting area to investigate is that of *heterarchy* [16], in which components may belong to multiple concurrent and cross-cutting networks.

8 Acknowledgements

References

[1] The Click modular router project. http://www.pdos.lcs.mit.edu/click/.

[2] Echidna: A real-time operating system to support reconfigurable software on microcontrollers and digital signal processors. http://www.enee.umd.edu/serts/research/echidna.

[3] Mescal - modern embedded systems: Compilers, architectures, and languages. http://www.gigascale.org/mescal/.

[4] The Ptolemy project. http://ptolemy.eecs.berkeley.edu.

[5] TinyOS: a component-based OS for the networked sensor regime. http://webs.cs.berkeley.edu/tos/.

[6] G. R. Andrews. *Concurrent Programming: Principles and Practice*. Addison-Wesley Publishing Company, 1991.

[7] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, A. Sangiovanni-Vincentelli, E. M. Sentovich, and K. Suzuki. Synthesis of software programs for embedded control applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(6):834–849, June 1999.

[8] A. Benveniste, B. Caillaud, and P. L. Guernic. From synchrony to asynchrony. In J. C. Baeten and S. Mauw, editors, *10th International Conference on Concurrency Theory (CONCUR'99)*, volume 1664 of *Lecture Notes in Computer Science*, pages 162–177. Springer-Verlag, August 1999.

[9] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, January 2003.

[10] G. Berry. The foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pages 425–454. MIT Press, 2000.

[11] S. S. Bhattacharyya, E. Cheong, J. Davis, II, M. Goel, C. Hylands, B. Kienhuis, E. A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, B. Vogel, W. Williams, Y. Xiong, and H. Zheng. Ptolemy II: Heterogeneous concurrent modeling and design in Java. Memorandum UCB/ERL M02/23, University of California, Berkeley, Berkeley, CA, USA 94720, August 2002.

[12] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer, 1996.

[13] E. Cheong, J. Liebman, J. Liu, and F. Zhao. TinyGALS: A programming model for event-driven embedded systems. In *Proceedings of the Eighteenth Annual ACM Symposium on Applied Computing*, pages 698–704, March 2003.

[14] R. Comerford. Handhelds duke it out for the internet. *IEEE Spectrum*, 37(8):35–41, August 2000.

[15] Crossbow Technology, Inc. http://www.xbow.com/.

[16] W. Fontana and J. Padgett. Keck program statement for "evolution of complex structure and form", April 29 1999. http://home.uchicago.edu/j̃padgett/papers/sfi/keck.pdf.

[17] J. G. Ganssle. *The Art of Designing Embedded Systems*. Newnes, 1999.

[18] 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 *Proceedings of Programming Language Design and Implementation (PLDI) 2003*, June 2003. http://webs.cs.berkeley.edu/tos/papers/nesc.pdf.

[19] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[20] J. Hill. A software architecture supporting networked sensors. Master's thesis, University of California, Berkeley, Fall 2000.

[21] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 93–104. ACM Press, 2000.

[22] A. Iyer and D. Marculescu. Power and performance evaluation of globally asynchronous locally synchronous processors. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 158–168. IEEE Computer Society, 2002.

[23] W. D. Jones. Keeping cars from crashing. *IEEE Spectrum*, 38(9):40–45, September 2001.

[24] E. Kohler. *The Click Modular Router*. PhD thesis, Massachusetts Institute of Technology, November 2000.

[25] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[26] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[27] E. A. Lee. What's ahead for embedded software? *IEEE Computer*, 33(7):18–26, September 2000.

[28] E. A. Lee and Y. Xiong. System-level types for component-based design. In T. A. Henzinger and C. M. Kirsch, editors, *First International Workshop on Embedded Software (EMSOFT 2001)*, volume 2211 of *Lecture Notes in Computer Science*, pages 237–253. Springer-Verlag, October 2001.

[29] J. Liu and E. A. Lee. Timed multitasking for real-time embedded software. *IEEE Control Systems Magazine*, pages 65–75, February 2003.

[30] A. Mihal and K. Keutzer. Mapping concurrent applications onto architectural platforms. In A. Jantsch and H. Tenhunen, editors, *Networks on Chip*, chapter 3, pages 39–59. Kluwer Academic Publishers, 2003.

[31] T. J. Mowbray, W. A. Ruh, and R. M. Soley. *Inside CORBA: Distributed Object Standards and Applications*. Addison-Wesley, 1997.

[32] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, volume 2. John Wiley & Sons, 2000.

[33] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. John Wiley & Sons, 6th edition, 2001.

[34] D. B. Stewart and R. A. Brown. Grand challenges in mission-critical systems: Dynamically reconfigurable real-time software for flight control systems. In *Workshop on Real-Time Mission-Critical Systems in conjunction with the 1999 Real-Time Systems Symposium*, November 1999. http://www.embedded-zone.com/bib/position/rtmcs99.pdf.

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

[36] C. A. Waldspurger and W. E. Weihl. Stride scheduling: Deterministic proportional-share resource management. Technical Report MIT/LCS/TM-528, June 1995.

[37] M. Weiser. The computer for the twenty-first century. *Scientific American*, pages 94–104, September 1991. http://www.ubiq.com/hypertext/weiser/SciAmDraft3.html.

[38] M. Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):75–84, 1993.

[39] M. Welsh, D. Culler, and E. Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, pages 230–243. ACM Press, 2001.

[40] M. Welsh, S. D. Gribble, E. A. Brewer, and D. Culler. A design framework for highly concurrent systems. Technical Report UCB/CSD-00-1108, University of California, Berkeley, April 2000.

[41] F. Zhao and L. Guibas, editors. *Second International Workshop on Information Processing in Sensor Networks (IPSN 2003)*, volume 2634 of *Lecture Notes in Computer Science*. Springer-Verlag, April 2003.

[42] Y. Zhao. A study of Click, TinyGALS and CI. http://ptolemy.eecs.berkeley.edu/ ellen_zh/click_tinygals_ci.pdf, April 2003.