Nonintrusive Precision Instrumentation of Microcontroller Software

Ben L. Titzer
UCLA
Computer Science Department
titzer@cs.ucla.edu

Jens Palsberg
UCLA
Computer Science Department
palsberg@ucla.edu

Abstract

Debugging, testing, and profiling microcontroller programs are notoriously difficult. The lack of supporting software such as an operating system, a narrow interface to the hardware chip, and delicately timed sequences of code present significant challenges which can be exacerbated by the presence of additional debugging or profiling code. In this paper we present a solution to the precision instrumentation problem for microcontroller code that is based upon our open, flexible simulator framework, Avrora. Our simulator preserves all timing and behavior of the instrumented program while allowing precision measurement of application-specific quantities.

Categories and Subject Descriptors I.6 [*Computing Methodologies*]: Simulation and Modeling

General Terms Experimentation, Performance

Keywords Sensor networks, instruction-level simulation, cycle-accurate, parallel simulation, instrumentation, debugging, monitoring, profiling

1. Introduction

For many embedded applications, *microcontrollers* can be used in place of application-specific integrated circuits (ASICs), which significantly lowers cost and increases flexibility. Microcontrollers are small, limited-power processors that represent systems on a chip: a central processing unit, main memory, and I/O devices in one package. They generally have limited computational power and resources but correspondingly low cost, which makes them an ideal fit for high-volume, low-price electronics where they may serve as the central or auxiliary control (e.g., a fuel injection system in an internal combustion engine).

Compared to desktop applications, debugging, testing, understanding, and measuring programs that run on a microcontroller can be a significant challenge because it is hard to observe the software operation in detail. One approach is to instrument programs at the source level manually to dump debugging or profiling information to a narrow interface such as a serial port or individual hardware pin. However, often these interfaces are partially controlled by the software itself and can fail mysteriously. Such instrumentation can also alter the timing and behavior of the underlying program being instrumented, distorting or ruining results. Alternatively, many microcontrollers provide an in-circuit-emulation (ICE) mode that can be accessed through a protocol such as JTAG, which allows the hardware to be paused, stepped, and resumed. The interface is often low-level and may be utilized by a debugger such as gdb [8]. However, the interface is insufficient for complex profiling tasks or debugging the program in the midst of a complex environment scenario where external devices are present.

A popular approach to debugging and testing microcontroller programs is to use an instruction-level simulator that is able to interpret the microcontroller program accurately and report a trace of various aspects of its execution. The relevant state of the program is “dumped” as output as the program executes, and the output is observed manually or post-processed by external tools. However, this simulator-based approach can require significant modifications to the simulator to allow the reporting of all relevant information and it can generate large amounts of output from which it can be difficult to reconstruct complex behavior.

In this paper we present a new approach to instrumenting microcontroller software. In contrast to previous work, our approach has all of the following six properties:

- **modular:** the instrumentation can be written without editing the source or target code of the microcontroller;
- **efficient:** the instrumentation introduces a performance overhead that is small and proportional to the computation performed by the instrumentation itself;
- **flexible:** the instrumentation can be inserted statically or dynamically;
- **high level:** the instrumentation can be written in a high-level language and use complex data structures;
- **nonintrusive:** the instrumentation does not alter behavior, timing, or energy usage of the microcontroller software;
- **comprehensive:** the instrumentation can report on locations in memory, individual control points, events from the simulation itself (such as reception of packets in a sensor network simulation), timing counts, energy counts, etc.

Our simulator-based approach offers three core instrumentation mechanisms, *probes*, *watches*, and *events*, that are inspired by the Observer pattern [10] to allow dynamic insertion of program instrumentation. Dynamic instrumentation allows adding arbitrary profiling behavior at any point in the program or in response to any type of event at any point in simulation time, without making any modifications to the simulator or influencing the program’s behav-

	modular	efficient	flexible	high level	nonintrusive	comprehensive
Manuel instrumentation	no	yes	yes	yes	no	no
Automatic instrumentation	yes	yes	yes	yes	no	no
Simulation traces	yes	no	no	yes	yes	yes
Hardware monitors	yes	yes	no	no	yes	no
Debuggers	yes	no	yes	yes	no	no
Avrora	yes	yes	yes	yes	yes	yes

Figure 1. Six approaches to instrumentation

ior or timing. We have implemented our approach in Avrora [19], which is a cycle-accurate instruction-level simulator for the AVR microcontroller and for networks of sensor nodes built on the AVR microcontroller.

In the following section, we discuss several traditional approaches to debugging and testing and we argue why none of them have all six properties listed above. In section 3 we discuss the architecture of Avrora and how it lends itself naturally to instrumentation. In section 4 we introduce our instrumentation mechanisms and their interfaces within the simulation, and in section 5 we discuss efficient implementation strategies. In section 6, we evaluate the performance overheads introduced by common types of instrumentation, including per-instruction profiling and periodic sampling. Finally, in section 7 we discuss applications such as monitoring stack-allocated quantities and report on our experience implementing a debugger backend and several profiling tools.

2. Previous Work

We now discuss five previous approaches to instrumenting programs and discuss their relevance to microcontroller programs. The five approaches are: manual and automatic source instrumentation, dynamic binary instrumentation, trace-based simulation, hardware performance counters, and debugging. For each approach, we assess which of the six desired properties (as listed in Section 1) it has, and what shortcomings it has. A summary of the discussion is shown in Figure 1.

2.1 Manual Instrumentation

Often programmers trace the execution of their program by manually inserting instrumentation statements that may print out the program location and value of one or more program variables, or toggle a physical pin on the microcontroller when they are reached. Thus the program generates its own trace that can be inspected to understand or verify its behavior as it executes on the actual hardware device. Some microcontrollers provide a serial interface to which print statements in the program write their output; a workstation or server connected on the other end displays the output on a user's terminal or stores it in a file. Often this requires linking in additional code to manage the serial port as well as the code to render program quantities such as integers and floating point to a string representation. In extreme cases and for small devices, the addition of this extra code may no longer allow the program to actually fit in the limited code space of the device. Lacing the code with printf's also subtly distorts its behavior and may radically affect its timing. It can sometimes lead to the dreaded "Heisen-bug" effect, where a bug in the program disappears depending on what instrumentation is enabled when the program is compiled.

Instrumenting to measure a program can also be done manually when a programmer gathers profiling information by inserting code to track various quantities he or she may be interested in such as loop iterations, function calls, data structures, or timing. The presence of such code can often influence the results, introducing imprecision in the measurement. As with any addition to the program, the instrumentation code or data collected may inadvertently break the program or cause it to no longer fit on the device due to limited space.

Jeffery and Griswold [12] presented an approach to execution monitoring in which the instrumentation code is written in the same language as the monitored program, in a modular way. The instrumentation code is written in a co-routine, and every time an event of interest occurs control is transferred to the instrumentation code. Thus, the only extra code that appears in the monitored program are statements for yielding control to the instrumentation co-routine. A related approach can be achieved using aspect-oriented programming [13] in which instrumentation code, for example, can be written in a separate aspect and weaved in with the monitored code using an aspect weaver. Aside from the modularity achieved using co-routines and aspects, those approaches suffer from the same drawbacks as other approaches to manual source instrumentation.

To summarize: manual instrumentation can break the program, is labor-intensive, is poor at debugging, understanding, and testing, but good at measuring application-specific quantities.

2.2 Automatic Instrumentation

Software is automatically instrumented in many contexts for many purposes. Let us begin with automatic source instrumentation. Templer and Jeffery [18] presented the CCI tool, which inserts code into C programs for obtaining events that are useful for monitoring. Auguston [3] presented a tool for instrumenting C programs such that assertions can be checked at run-time or such that a trace can be saved and later post-processed. Such tools obviate the need for editing the monitored program and they often provide a domain-specific language [18, 3, 4] for writing the instrumentation code, and tools that can translate the instrumentation code to the language of the monitored program. The main problem with this approach is that it does insert code into the monitored program and therefore changes the timing properties, increase code size, etc. Also, the instrumentation code can only have access to what is available at the source level of the language of the monitored program, not quantitative measures such as cycle counts, energy counts, etc.

It is also possible to do automatic binary instrumentation. Of particular interest is *dynamic* binary instrumentation which can be done either using run-time code generation, as done by Hollingsworth, Miller, Goncalves, Naim, Xu, and Zheng [11] or using operation-system support, as done by Tamches and Miller [17], Moore [14], and Cantrill, Shapiro and Leventhal [6], Also for binary instrumentation may one use a domain-specific language for writing the instrumentation code [11, 14, 6]. Dynamic binary instrumentation is not suited for microcontrollers, where machine code is often immutable, and there is no supporting software such as an operating system which many such techniques require.

2.3 Simulation Traces

Another approach is to use an instruction-level simulator, such as SimpleScalar [5], which can execute the machine code program and produce a trace of its execution. Simulation offers the advantage of reproducibility, which solves the "Heisen-bug" problem, and the ability to inspect the execution in fine-grained detail by producing a detailed trace as the output of simulation. The trace can then be inspected either manually or by automated tools external to the simulator to glean the desired information about the program's execution. Some simulators offer various output options to be selected at the beginning of executing the simulation such as tracing each instruction, memory access, stack operation, interrupt, device event, etc, and may offer higher-level services such as counting instruction executions, cache misses, or other things.

One main disadvantage is that even simple monitoring tasks can have dynamic behavior that is not easy to capture with a simulation trace. In such a case all of the relevant trace information must be enabled at the beginning of execution and sifted through later, since they cannot be enabled and disabled dynamically. For example, the simple task of monitoring a local variable in a function is complex; the variable may at times be stored in a register and at other times on the stack, depending on how the compiler allocates registers, and while on the stack it might be modified indirectly through a pointer access. Worse, the variable is only in scope during the lifetime of the function call; it is not adequate to simply monitor a single memory location in the stack. Enabling all tracing to capture all such behavior leads to the problem that the trace output can be huge; programs may execute many millions of instructions and generate gigabytes of data quickly. Generating such an amount of data and processing it slows down simulation considerably, rendering it impractical for long-running simulations. In our approach, we solve this problem by allowing instrumentation to be enabled and disabled dynamically.

Another disadvantage is that such an approach may require intrusive modifications to the simulator if the relevant state for an instrumentation task is not available outside of the simulator. Intrusive modifications require access to the simulator source code and some knowledge of its implementation. Our approach requires no such modifications, and only knowledge of a clear, well-defined interface to the state of the simulation.

To summarize: simulation traces can be slow and cumbersome and poor at debugging, understanding, but sufficient for testing and measuring, as long as it is not application-specific.

2.4 Hardware Monitors

Many hardware designs have additional logic that allows debugging or profiling of a program as it executes. For example, modern desktop CPUs record hardware profiling information such as branch frequencies and cache misses which can be used to understand the performance of an executing program. Ammons, Ball, and Larus [2] used hardware counters on the Sun UltraSPARC processor to do flow sensitive and context sensitive path profiling. Adl-Tabatabai, Hudson, Serrano, and Subramoney [1] used hardware performance monitors on the Intel Itanium 2 processor to do gather information that can help improve prefetching. However, in embedded systems where chip size and power consumption are important, such logic is rare, and programs cannot be profiled in this way.

Fortunately, most CPUs and microcontrollers do support a hardware interface that allows a program to be loaded and run. Usually called in-circuit emulation or ICE, logic built into the device offers an external physical interface for programs to be loaded, run, paused, stepped, and resumed, as well as the state of the registers, memory, and on-chip devices to be inspected. A source-level debugger can be connected to this hardware interface and allow debugging of the program running on the actual chip. However, one disadvantage is that complex application-specific measurements of the software cannot be performed through this interface (e.g. measuring hits in a software-implemented hash table).

Also, many microcontroller programs interact with external devices that are wired directly to the microcontroller. Often these devices are not designed to participate in the debugging of the program; they cannot be paused, inspected, and resumed as the microcontroller itself can. Therefore debugging programs in the presence of external devices can be difficult. Troubleshooting timing problems can therefore be frustrating and difficult, especially when external devices are involved.

To summarize: hardware-supported instrumentation is good at debugging (with limitations for external devices) and understanding, but poor at measuring and testing application specific quantities and is often not available for embedded systems.

2.5 Debuggers

Many debuggers are able to attach to either a hardware-based debugging interface such as JTAG or to an instruction level simulator. Either generally offers a comparable interface; breakpoints, watchpoints, and read/write of various memory locations. Debuggers make use of source-level information preserved by the compiler that describes the mapping of the machine code and memory addresses back to the source code. However, most compilers are unable to preserve complete debugging information when the program is compiled at the highest optimization levels, often leading to separate "debug" and "optimized" builds.

A debugger can be an excellent tool for interacting with an executing program and may offer limited profiling and monitoring of source-level behavior, but most debuggers are not suited to complex monitoring tasks, for automated testing, and are often not extensible in what behavior can happen at breakpoints and watchpoints. Recent progress was reported by Ducassé [9] who presented an approach to debugging for C in which instrumentation is written in Prolog and queries can be executed at breakpoints. However, debuggers are often limited by the quality of source-level mapping provided by the compiler and may be confused by advanced optimizations such as code motion.

Inspired by the shortcomings of the previous approaches to debugging, testing, and measuring, we will present an approach that has all six desired properties. We have implemented our approach in the Avrora simulator [19].

3. The Avrora Simulator

Avrora is a flexible simulator framework. In this section we briefly discuss aspects of Avrora's software architecture relevant to designing and implementing instrumentation mechanisms; the discussion in this section is based on our paper [19] and is provided for convenience and completeness.

Avrora is implemented in Java; its object-oriented design lends itself to encapsulating the principal concepts in simulation such as instructions, devices, and state in an intuitive manner.

Figure 2 shows a structural diagram of the software architecture. The encapsulation mirrors the physical design of a device; a microcontroller contains an interpreter capable of executing instructions and on-chip devices such as timers, a serial controller, etc, and is itself contained in a larger platform abstraction which adds external devices.

3.1 Device Simulation

One primary problem of developing and testing a microcontroller program that controls an external device is that often the target device cannot be paused and stepped like the microcontroller in ICE mode. Stopping the microcontroller in the middle of its interactions with the external device may give unexpected and incorrect results. In order to address this problem, Avrora's architecture allows a model of each device to be built in software that emulates the device and allows the program to interact with it. Each device model is written in Java and connects to the main simulation

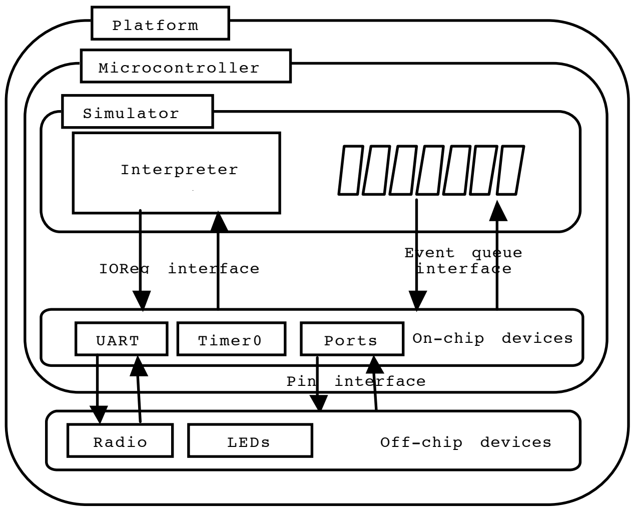

Figure 2. Avrora's software architecture

through interfaces that separate it from the details of the simulator implementation.

On the AVR architecture, the program interacts with internal devices through memory-mapped I/O registers. In the simulation, each I/O register is represented as a Java object that has both `read()` and `write()` methods. Each device can then implement its own behavior for reading and writing the registers associated with that device, without any knowledge of the rest of the simulator implementation.

For external devices, the program can read or write the logic value of an individual pin on the microcontroller chip or can communicate through one of the more complex interfaces such as the serial port. In the simulator architecture, external devices can be implemented as Java classes and connected to individual pins or to a serial port interface, and thus are separated from the simulator details.

These interfaces in Avrora's software architecture make it easy to implement new device models and connect them to the simulation without requiring modification or recompilation of the simulator, as, like any Java class, new device classes can be loaded dynamically and wired into the simulation. In this way, both the simulator and the devices are insulated from changes in implementation. We demonstrate in section 6 that this increased modularity does not sacrifice performance by comparing Avrora to two other AVR simulators with comparable features.

3.2 Event Queue

In order to achieve a high degree of accuracy and closely match the hardware, cycle-accurate simulation is needed. The precise timing of individual instructions and devices can be important for accurate performance accounting, testing real-time properties, and correct emulation of device operation such as timers that are derived from the main clock.

Building a cycle-accurate interpreter for microcontrollers is straightforward, since most microcontrollers have an ISA where each instruction (with the exception of branches) takes a fixed number of cycles to execute and there are no pipeline stalls.

However, microcontroller programs often depend heavily on the timing of on-chip and off-chip devices. On AVR, some examples include: a timer might be programmed to increment a register every 16 cycles and trigger an interrupt when the count reaches a maximum value; the UART chip communicates over the serial port by writing bits out one by one on a pin at a fixed and known clock rate; the EEPROM write sequence offers only a strict timing window for the program to supply the data in order to work correctly. Also, controlling external devices can often have complex timing behavior that should be modeled accurately in simulation.

One naive interpreter implementation would be to simply notify each device after every clock cycle, and each device would decide what work (if any) should be done. While correct in terms of timing, this strategy complicates device implementations and leads to poor simulation performance. Avrora solves this problem by exposing an event queue that is timed by the main microcontroller clock. Devices perform work at the correct time by inserting events into this queue. For example, when the timer is programmed to increment a register every 16 cycles, it simply inserts an event into the queue 16 cycles into the future. When the interpreter reaches that time in the future, the timer event is fired. The event increments the register and inserts itself again into the queue 16 cycles into the future.

Avrora implements the event queue as an efficiently maintained delta queue, with the nearest event in the future at the head. After each instruction, it subtracts the cycles consumed by that instruction from the count at the head of the queue and fires the event when it reaches zero, moving the head to the next link.

The event queue also enables the sleep optimization where when the microcontroller is in sleep mode, the simulator only needs to processes events in the queue in order, skipping large amounts of idle simulation time.

We now turn to the contribution of this paper: design and implementation of nonintrusive precision instrumentation.

4. Instrumentation Mechanisms

For simulation-based approaches, a simulator's usefulness is related to the instrumentation services that it provides. For example, systems such as shade [7] and SimpleScalar [5] provide instruction counts, cache misses, execution time, and other quantities envisioned by the designers of the simulator, as well as the ability to dump traces as output. Users are limited by what the simulator provides or what they can reconstruct by post-processing the output of simulation.

In Avrora, we provide three general mechanisms: probes, watches and events. These represent instrumentation points instead of fixed services. Common services can be implemented in terms of these mechanisms while additionally allowing new types of instrumentation to be added easily by users of the system—without modifying or recompiling any code. This enables detailed, application-specific results and solves an open extensibility problem not previously recognized in this domain.

4.1 Probes

Many instrumentation tasks require some behavior to be triggered when a particular location in the program is reached. For example, a debugging task might set a breakpoint at a particular instruction corresponding to a line of source code, or a profiling task may keep a count of the number of times a particular function is called. A typical timing measurement of interest might be the number of clock cycles between the program entering one location, like the start of an interrupt handler, and it reaching another location, such as the end of the interrupt handler. More complex tasks are also possible. Rather than attempt to foresee all possible scenarios when building our simulator, we instead choose to offer a point of extensibility called a probe, which allows arbitrary behavior to inserted at any location in the program, as illustrated in Figure 3.

The simulator exposes an interface `Simulator.Probe` that allows a Java object implementing the interface to be inserted at a

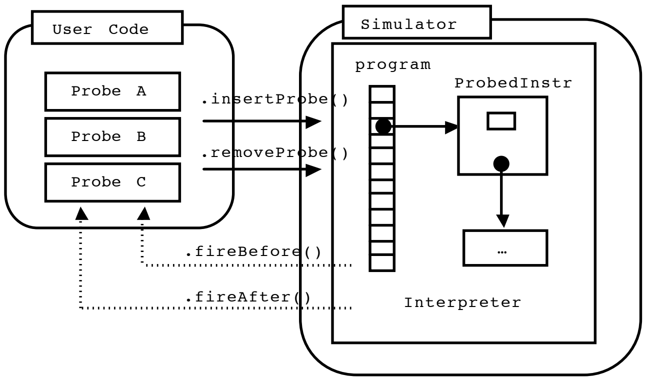

Figure 3. Instrumentation with probes

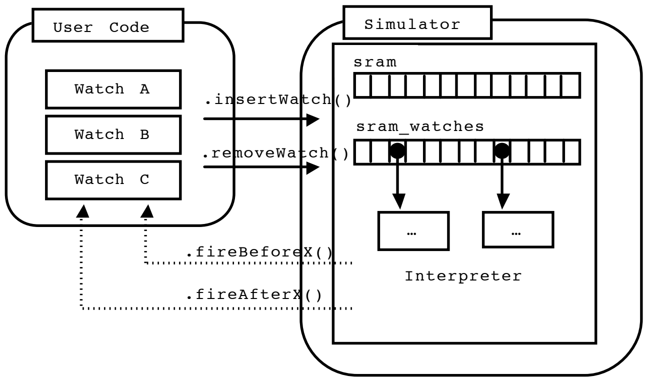

Figure 4. Instrumentation with watches

particular location in the program. The `Simulator` class exposes two methods:

`insertProbe(Probe p, int addr)` and
`removeProbe(Probe p, int addr)`

that allow probes to be inserted and removed from a program location dynamically during the execution of the program. When the program reaches that location, the probe's `fireBefore()` is called before the instruction executes, and the Instr object representing the instruction at the location, the address of that instruction, and a `State` object representing the state of the simulation are passed to the method. After the probe returns, the instruction is executed by the interpreter. After the instruction has finished execution, the probe's `fireAfter()` method is called, again passing the instruction, its location, and the state of simulation as parameters to the call. Since the probe executes in the virtual machine running the simulation, and not as part of the executing program, it cannot inadvertently alter the program's execution. The `State` interface provides methods to access the state of registers, memory, and devices and enforces that the probe can only read the state of the program being simulated and not alter it. Since the probe runs "in the simulation", it does not affect the timing of the underlying program.

The simplest probe may just increment a counter for that instruction, as shown in Figure 5. A more complex probe might pause the simulation (such as a breakpoint), inspect the registers or memory, or insert or remove other probes. Since a probe is a Java class

```
public class ProgramProfiler
       implements Simulator.Probe {
  public final Program program;
  public final long icount[];
  public ProgramProfiler(Program p) {
    int size = p.program_end;
    icount = new long[size];
    program = p;
  }
  public void fireBefore(Instr i, int address,
                         State state) {
    icount[address]++;
  }
  public void fireAfter(Instr i, int address,
                        State state) {
    // do nothing.
  }
}
```

Figure 5. A program profiler probe

```
public class MemoryProfiler
       implements Simulator.Watch {
  public final long rcount[];
  public final long wcount[];
  public MemoryProfiler(int size) {
    rcount = new long[size];
    wcount = new long[size];
  }
  public void fireBeforeRead(Instr i, int address,
                             State state,
                             int data_addr,
                             byte value) {
    rcount[data_addr]++;
  }
  public void fireBeforeWrite(Instr i, int address,
                              State state,
                              int data_addr,
                              byte value) {
    wcount[data_addr]++;
  }
  public void fireAfterRead(Instr i, int address,
                            State state,
                            int data_addr,
                            byte value) {
    // do nothing
  }
  public void fireAfterWrite(Instr i, int address,
                             State state,
                             int data_addr,
                             byte value) {
    // do nothing
  }
}
```

Figure 6. A memory profiler watch

just like other parts of the simulator, it can be loaded dynamically like other classes in Java; the simulator need not be modified or recompiled to allow the probe to be inserted. Our technique therefore benefits greatly from the inherent flexibility in the Java language. For simulators implemented in C or C++, this level of modularity could be achieved by allowing the user to specify a dynamic library containing instrumentation code that the simulator would link in through `dlopen` and `dlsym` calls, removing the need to recompile the simulator to add instrumentation. We show some common uses of probes in section 6.

4.2 Watches

Just as in many cases an instrumentation task requires some actions to be taken when a particular location in the program is reached, some tasks require actions to be taken when a particular memory location is accessed. For example, many debuggers allow a "watchpoint" to be inserted on a variable in the program, and when that variable is modified, the program will be paused. This cannot be done simply by adding probes throughout the code of the program because memory locations can be modified indirectly through pointers or may reside in the stack. In this case, the most natural and general mechanism is to allow another point of extensibility called a watch to be inserted at the particular memory location, as illustrated in Figure 4. Reads and writes to that memory location will then trigger the actions in the watch.

Similar to the case with probes, the simulator exposes an interface `Simulator.Watch` that allows a Java object implementing the interface to be inserted at a particular memory location in memory. The `Simulator` class exposes two methods:

`insertWatch(Watch w, int addr)` and

`removeWatch(Watch w, int addr)`

that, just as for probes, allow watches on memory locations to be inserted and removed dynamically as the program executes. When the program attempts to read that memory location, the watch's `fireBeforeRead()` method is called, the read is performed, and then the method `fireAfterRead()` is called. Similarly, when the program attempts to write to that location, the `fireBeforeWrite()` and `fireAfterWrite()` methods before and after the write, respectively.

The simplest watch may just increment a counter for that memory location, as shown in Figure 6. Watches can be used to implement watchpoints for debuggers, value profiling for data structures, monitoring reference locality for cached architectures, or can enable other types of instrumentation dynamically by inserting or removing probes or other watches in the program. This dynamic property can be key in supporting many application-specific profiling tasks such as monitoring each item in a list, the efficiency of a hash table, accesses to a matrix, etc. We show some common uses of watches in section 6.

4.3 Events

One often-overlooked aspect in instrumentation schemes is the accurate accounting of time. For delicately timed code, interaction with devices, or for precise performance measures, accurate timing is essential. While instrumenting locations in the code and memory of an executing program is essential, in some instrumentation applications "instrumenting" time in the same way is useful.

For example, one application is periodic profiling of the program. For a course-grained approximation of which locations in the code are executed the most frequently, the program counter can be sampled periodically with an event. At the beginning of the program, an event is inserted in the queue for 1000 cycles (for example) in the future. When this event fires, it will record the program counter value and determine which function contains that program counter. The function is then charged with a "tick", and the event is re-inserted into the queue for 1000 cycles in the future. In this way, functions accumulate "ticks", which approximates the time spent in each.

Following the theme of the `Simulator.Probe` interface and the `Simulator.Watch` interface, the `Simulator` class exposes an interface `Simulator.Event` that allows a Java object implementing the interface to be inserted into the event queue. The interface has a single method, `fire()`, that the simulator calls when the time for the event to be fired is reached.

5. Implementation Techniques

In this section, we discuss the efficient implementation of the instrumentation mechanisms. For probes and events, the overhead incurred is proportional to the amount of instrumentation present, with no overhead if probes and events are not used. The support for watches imposes a negligible overhead regardless of use.

5.1 Probe Implementation

Avrora implements probes with minimal simulation overhead. Since probes are tied to the execution of individual instructions, the interpreter must be modified to support executing probes before and after (potentially) any instruction in the program. Probes also be inserted and removed dynamically, so the interpreter must be flexible enough to allow this.

Avrora represents each instruction in the program as an instance of a Java object descending from the `Instr` class. Each type of instruction is represented by a class inside of `Instr` that also extends `Instr`. For example, the `Instr.ADC` class represents instances of the "adc", or add with carry, instruction. Instances of each instruction class store the operands (registers, immediates, etc) as fields in the object. When executing instructions, a main loop controls execution by keeping track of the current instruction, calling a method on the instruction for it to be executed, and then advancing to the next instruction. (Originally we expected the cost of the call to be a limiting factor for interpreter performance; however, we found that on modern Java virtual machines that cost is negligible compared with the other costs of interpreting the instruction).

The naive strategy to add probes to an instruction-level simulator would alter the main interpretation loop to check if the instruction to be executed has any attached probes and if so fire calling the probes' `fireBefore()` methods, execute the instruction, and then call the `fireAfter()` methods. This imposes overhead on the simulation because the tests for the presence of probes happen for each instruction executed, regardless of whether that instruction is being probed or not.

We solve the efficiency problem by introducing a new class, `ProbedInstr`, that extends `Instr` and, like other instructions, has a method that is called to execute it. When a probe is inserted at a particular instruction in the program, a new `ProbedInstr` instance is created that replaces that instruction and contains references to the original instruction and the probe(s) attached to the instruction. Later, when that instruction is executed, the `ProbedInstr` is executed instead of the original instruction. It will first call the probes before execution, then call the original instruction's execute method, and then call the probe(s) after the instruction is finished. In this way, the main loop of the interpreter need not be altered and the performance of non-probe instructions is not degraded, since interpreting each instruction already involves one indirect call.

This implementation technique for probes is widely applicable; even in an interpreter which uses a large switch statement with various integer codes to differentiate between instruction types, the same effect can be derived by adding an extra instruction type for a "probed instruction", adding a new case in the switch statement, and replacing the original instruction in the program with a probed instruction when the probe is inserted. This new case in the switch statement therefore adds no overhead to the execution of the other instructions.

Our implementation technique has the important property that probing the executing program has overhead which is proportional to the dynamic count of probes fired; simulation without instrumentation runs at full speed. We validate this claim in section 6 by showing that our simulator outperforms two other widely available AVR simulators.

5.2 Watch Implementation

The simulator must implement watches in a way that attempts to minimize the performance overheads in simulation while still preserving the ability to dynamically insert and remove watches.

In Avrora, the memory (SRAM) of the microcontroller is represented by a simple array of bytes `sram[]` indexed from 0 to the highest memory address. In the case of the AVR architecture, this memory is byte addressable and the largest currently available memories are 4KB in size. The byte array that represents SRAM is stored in the interpreter and is read and written by the instruction implementations through the `getDataByte()` and `setDataByte()` methods. These methods were modified in order to support watches. In addition to the array `sram[]` for storing the actual value of memory locations, a parallel array `sram_watches[]` is kept where each element stores a (possibly empty) list of watches for each memory address. The reference to this array is non-null only when there are active watches at any memory location; it is not allocated until the first watch is activated.

The `getDataByte()` and `setDataByte()` methods are implemented such that the reference to the `sram_watches[]` array is tested first. If this reference is `null`, then there are no currently active watches, and the read or write happens normally. If the array is present, then the list at the location corresponding to the memory address being accessed is tested. If this element is `null`, then there are no watches for this memory address, and the read or write can happen normally. If the list is not `null`, then the watches are called before and after the access occurs. For large memories, it may be necessary to have a more space efficient implementation of the `sram_watches[]` array, such as a two-level array, a tree, or a hash table. We have not found this to be necessary, since AVR memories are 4KB or less.

While for probes we can completely eliminate overhead for non-probed instructions, there is still a small cost on each memory access to check whether there are 1) any watches for the entire memory and 2) any watches for that address. We measure this overhead in section 6 for computationally intensive and memory intensive benchmarks. For most programs, the cost of simulating memory accesses is small compared to the other costs of simulation, and therefore the extra overhead for supporting memory watches is negligible.

5.3 Event Implementation

As discussed in section 3, an event queue tracks individual clock cycles for each instruction and allows cycle-accurate interpretation of instructions and device implementations. Instrumentation events are treated just as another type of event in the simulation, whether it be work to be performed by a device, communication, etc. In this way the event queue mechanism is reused, and no extra cost is paid for apply it to profiling.

6. Experimental Results

In this section, we evaluate the performance overheads introduced in implementing the instrumentation mechanisms over a variety of common instrumentation tasks. For all experiments, we use the same machine: a dual 3.06ghz Xeon server with 4GB of memory running Linux kernel 2.6.8. To run Avrora, we use the publicly available Sun Hotspot JVM version 1.4.2-05 in client mode, unless specified otherwise. For benchmarks, we use: i) three computationally intensive programs taken from the Livermore loops series: livermore1, livermore2, and livermore5; ii) a C implementation of bubblesort for arrays of 16-bit integers; iii) a device-intensive TinyOS sensor network program that uses the radio: CntToRfm; iv) an interrupt-driven sleep-often program that simply toggles LEDs periodically: Blink.

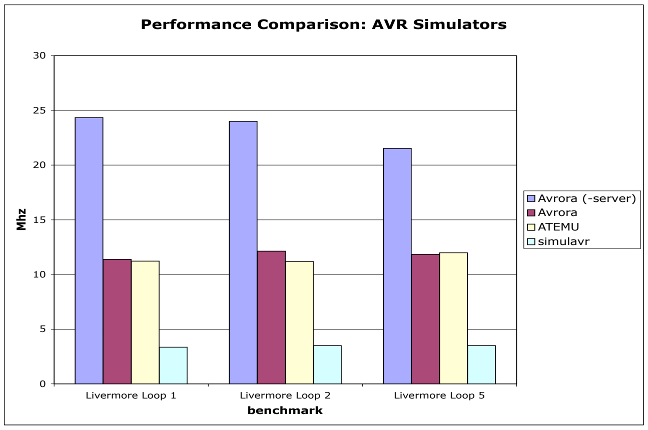

Figure 7. Performance Comparison

6.1 Simulator Performance

First, it is important to establish that our techniques can be used in a high-performance simulation environment, since evaluating an implementation technique in a slower simulator can mask overheads in the already poor performance of simulation. For this reason, we first show benchmarks that demonstrate our simulation framework has excellent performance compared to two other AVR simulators. The comparison is taken from our paper [19].

The first simulator is simulavr [16], an open-source AVR simulator that supports a variety of microcontroller models and on-chip devices and is widely used in the open-source community. It supports source-level debugging with gdb by communicating over a socket.

The second, ATEMU [15], is a sensor network simulator that is able to simulate mica2 motes including the ATMega128 AVR microcontroller, its on-chip devices, and an external CC1000 radio. It has its own custom debugger interface implemented in GTK+.

All three simulators interpret the full AVR instruction set, are cycle-accurate, and support most of the on-chip devices of the microcontroller. While Avrora is implemented in Java, ATEMU and simulavr are implemented in C and were compiled with gcc version 3.2.2 for our benchmarks.

Figure 7 shows the performance results comparing the three AVR simulators. First, we see that Avrora is sensitive to the Java Virtual Machine's performance; it has nearly double the performance when run on the more sophisticated server VM. Second, ATEMU and Avrora have comparable performance when Avrora is run on the client (default) version of the JVM, but Avrora outperforms ATEMU by up to a factor of two when run on the server VM. The other simulator, simulavr, lags behind the other two simulators significantly. All Avrora results include full instrumentation support; no instrumentation mechanisms were removed from Avrora to make it run faster.

Avrora and ATEMU are sensor network simulators; both are capable of simulating many sensor nodes running in parallel. Avrora scales well to large networks with high device activity, outperforming ATEMU by as much as a factor of 20, as reported in our paper [19]. This scalability helps to establish that Avrora is fast and scalable enough that measuring overheads in this context gives meaningful results.

6.2 Watch Support Overhead

Next we evaluate the performance overhead introduced in supporting watches, namely the extra checks performed when memory is read or written by the program. This overhead is proportional to the frequency of memory access by the program. For this reason, we a computationally intensive benchmark livermore1 that accesses memory infrequently, and a memory-intensive benchmark bubblesort.

Watches	livermore1	bubblesort
baseline	0.0%	0.0%
none	3.1%	15.8%
present	3.4%	24.0%
empty	3.9%	43.8%
count	4.0%	46.0%

After instrumenting each program, we found that bubblesort accesses memory once every 4.23 cycles on average, and livermore1 averages one access per 218 cycles. We first ran our simulator without the watch support by removing related checks in the interpreter to obtain a baseline performance number ("baseline"). We then evaluate four scenarios: when no watches are present ("none": `sram_watches[]` array is `null`), when one watch is present but does not ever fire ("present"), when an empty watch fires for every memory access ("empty"), and when a watch fires for every memory access and records the read/written count for every memory address ("count"). The numbers in the table represent the percentage slow-down from the baseline.

6.3 Exact Execution Profile

The first and simplest type of instrumentation task we evaluate is an exact execution profile of the entire program. The instrumentation collects an execution count and a cycle count for every instruction in the program. The instrumentation does this by inserting a probe at every instruction. During simulation, the probe is invoked for every instruction executed and records a counter and the total cycles consumed for each instruction in an array indexed by the instruction's address. This information is reported to the user in a table at the end of simulation. The benchmark used is livermore1.

Probes	Time	Overhead
none	63.3s	0.0%
empty probe	74.6s	17.8%
count	84.4s	33.4%
count + cycles	117.4s	85.5%

It is important to note that most instrumentation tasks do not require work to be done for every instruction executed. For example, to obtain an execution count and cycles consumed for the whole program on a basic block level, probes would only need to be inserted at the beginning and end of each basic block. For a function granularity measurement, probes would only need to be inserted at the beginning and end of each method.

6.4 Periodic Execution Profile

Exact profiling can place a significant overhead on simulation time, especially if the each sample is expensive to compute, such as a call chain. While short simulations may not be affected much, the overhead may be unacceptable for a simulation that lasts hours or days. One popular approach to the dynamic profiling problem is to use periodic sampling, where the program counter location, the call stack, or other criteria is sampled periodically, such as every n cycles or milliseconds. This gives an approximate answer that is close to the exact profile with high probability but with significantly less overhead because samples are taken less often. We ran a similar experiment to measure the overhead of this approach using four different periods: 1, 10, 100, and 1000 cycles. This instrumentation is implemented by an event that is placed in the queue of the simulation. When this event fires, it will read the value of the program counter, update a count for that instruction, and then reinsert itself into the queue for the next period.

Period	Time	Overhead
none	63.0s	0.0%
1 cycle	189.2s	200.2%
10 cycles	79.7s	26.4%
100 cycles	66.0s	4.7%
1000 cycles	64.0s	1.5%

As expected, the overhead is severe for sampling each cycle, but decreases rapidly as the frequency of sampling is reduced. For a sampling period of 1000 cycles (7372 samples per second on mica2), the overhead imposed on simulation is just 1.5%.

7. Applications and Experience

Up to this point, our discussion has been limited to measuring quantities that are not application specific, such as the execution profile for an entire program, or the memory behavior for a whole program. We would like to be able to measure specific properties of the program that exist at the source level, such as the percentage of hits in a hash table, the length of the longest path in a tree, the time from one point in the program to another, or the most common value passed to a function. For an approach based on simulation traces, this can require a significant amount of post-processing by application-specific scripts.

Avrora supports these types of tasks by providing probes, watches, and events that are 1) comprehensive, allowing access to all program locations and data, 2) targeted, meaning individual locations can be singled out for instrumentation, and 3) dynamic, allowing instrumentation to be inserted and removed depending on the behavior of the program.

7.1 Dynamic Instrumentation

Dynamic instrumentation is key for application-specific instrumentation tasks where monitoring the dynamic properties of code and data structures at the source level is important. For example, in low-level systems code written in C, it is common to allocate buffers on the stack to avoid the extra overhead of allocating a block of memory from the heap and then freeing it after it is no longer used. In embedded systems it is common to use this technique when dynamic allocation of memory from the heap is not supported. Suppose we are interested in monitoring the contents of one such buffer. It is clear that static instrumentation is inadequate: we cannot determine ahead of time which memory addresses the buffer will occupy when it is allocated, and moreover, the addresses the buffer occupies will be reused as the stack grows and shrinks during the execution of the program. The problem reduces to the fact that the buffer has a scoped lifetime; it is only live while the function that allocated it is live. We can use the dynamic instrumentation mechanisms of Avrora to solve this problem.

Suppose that F is a C function that allocates a buffer `buf` in its stack frame and that we are interested in tracking the accesses to `buf` during execution. First we determine the starting point of F in the assembly code. Next, we determine the location of `buf` in F's stack frame as an offset from the stack pointer, which can be done by examining the debugging information generated by the compiler. Third, by automatically tracing the basic blocks from the beginning of the function to basic blocks that end with a return instruction, we determine the exit points of the function F. Now, we can write a probe $P_{F\text{-start}}$ that is inserted at the instruction at the beginning of F. This role of this probe is to insert a watch

W on the memory locations corresponding to `buf` when F begins execution. $P_{F\text{-start}}$ computes the location of `buf` by reading the the stack pointer from the `State` object and adding the known offset. When W has been installed on the buffer, any accesses to the buffer from any point in the program, even through pointers, will cause W to be called before and after the access. When F returns, we would like the watch to be removed, since that memory in the stack will be reused. We do this by inserting another probe $P_{F\text{-end}}$ at the exit points of F that will remove W from the memory locations when F exits.

In this way, we can achieve precise, application-specific instrumentation without modifying the source code or the binary. It only requires reading the debugging information generated by the compiler in order to determine where to insert probes and watches into the simulation.

Another example is complex runtime assertion checking. For example, we might have a data structure in our program that must obey some invariant, such as members of the *red_list* must all be *red*. Using the Avrora instrumentation mechanisms, we can write a system of dynamic watches that check this property each time an element is modified. When an item is inserted into the list, a probe fires that first checks whether the element is *red*. Then a watch is placed on that element so that if the element is ever modified indirectly, the watch can check that the *red* property still holds. When the item is removed from the list, we can remove the watch, and subsequent modifications to the element will not be checked.

7.2 Debuggers and Beyond

We demonstrated the comprehensiveness of our system by implementing several common instrumentation tasks in terms of probes and watches, allowing us to provide useful services without modifying the simulator code in order to support them.

One important service we wanted to offer was source-level debugging. Avrora supports this by accepting connections from gdb, which defines a protocol for remote debugging over a socket or a serial port. The debugger sends commands over this connection to insert breakpoints and watchpoints, inspect variables, and step individual instructions or source statements. We implemented a monitor that accepts gdb connections, decodes these commands, and implements each command with probes and watches, without modifying the simulator's code. For this task, dynamic insertion and removal of probes is essential, since the user may insert or remove breakpoints in the program as he or she interacts with it. The table below compares the number of lines of Java code, including comments and blank lines, needed to implement three common instrumentation tasks in Avrora. For the gdb server, we required only 570 lines of code, the majority of which handles the decoding of the serial protocol, 100 lines of which are probe and watch code.

Monitor	Total	Probes
gdb server	570 lines	100 lines
A-B timing	200 lines	20 lines
exact profile	200 lines	40 lines

Another common instrumentation task in real-time systems is A–B timing, where the time required for the program to execute starting at location A and ending at location B is measured. Locations A and B might be the start and end of an interrupt handler, for example. A–B is implemented in a very straightforward manner as two probes: the probe at location A records the start time, and the probe at location B records the end time, subtracting the time at A to obtain the result. The times for various traces can be averaged or the distribution plotted. We implemented a monitor that allows simultaneous A–B timing for an arbitrary number of pairs A and B in the program, as well as A–X timing, which records the time from one location in the program to every other location in the program.

8. Conclusion and Future Work

We identified six major properties that are important in an instrumentation system: modularity, efficiency, flexibility, high-levelness, nonintrusiveness, and comprehensiveness. We demonstrated how previous systems have not been able to have all these properties simultaneously. We identified three major extension points in simulation that can allow instrumentation to be added with a system that satisfies all six of these criteria. Additionally, we believe our system makes significant progress on two principal criteria of importance to microcontroller programs: nonintrusiveness and flexibility, while still retaining precision.

Our system is based on instrumenting machine code and raw memory. Its primary weakness is that accomplishing source-level instrumentation tasks manually can be tedious and requires detailed debugging information from the compiler. We would like to extend the system to make it easier to express source-level monitoring tasks and automatically translate the instrumentation to the appropriate machine-level probes and watches which are inserted at the correct points in the program.

Our experience has been that collecting information with probes is relatively easy compared to the tasks of organizing, correlating, processing and displaying that data, as seen in the ratio of monitor code to actual probe code in our demonstrative examples. Libraries of commonly used functionality could aid greatly in reducing the effort spent in this area.

Dynamic instrumentation is powerful, but it is also more difficult and tedious to implement because it is written at the machine level and probes may insert or remove probes or watches. It can be tricky to get the right probes in the right points in the program at the right time. Also, the instrumentation is also triggered by the underlying program's behavior, which in itself can be quite complex. A domain specific language for writing such instrumentation may help to simplify complex dynamic instrumentation tasks.

We are also interested in performing a more comprehensive study for larger embedded programs and more complicated profiling tasks, especially for embedded systems with larger memories. While our initial work has focused on microcontrollers, there is a large potential for applicability to these larger embedded platforms and this remains as future work.

Acknowledgments

We thank Daniel Lee for implementing many of the device simulations in Avrora including the CC1000 radio, Simon Han for valuable timing validation work against real devices as well as bug reports and suggestions for improvements to Avrora, and Todd Millstein for comments on an early draft of this paper. We were partially supported by the NSF ITR award number 0427202.

References

[1] Ali-Reza Adl-Tabatabai, Richard L. Hudson, Mauricio J. Serrano, and Sreenivas Subramoney. Prefetch injection based on hardware monitoring and object metadata. In *Proceedings of PLDI'04, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 267–276, 2004.

[2] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of PLDI'97, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, 1997.

[3] Mikhail Auguston. Assertion checker for the C programming language based on computations over event traces. In *Proceedings*

of AADEBUG'99, International Workshop on Automated Debugging, 2000.

[4] Mikhail Auguston, Clinton Jeffery, and Scott Underwood. A framework for automatic debugging. In *Proceedings of ASE'02, Automated Software Engineering*, pages 217–222, 2002.

[5] Todd M. Austin, Eric Larson, and Dan Ernst. Simplescalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, 2002.

[6] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of USENIX Annual Technical Conference, General Track*, pages 15–28, 2004.

[7] Bob Cmelik and David Keppel. Shade: a fast instruction-set simulator for execution profiling. In *Proceedings of SIGMETRICS'94, Conference on Measurement and Modeling of Computer Systems*, pages 128–137, 1994.

[8] The GDB developers. GDB: The GNU project debugger. http://www.gnu.org/software/gdb.

[9] Mireille Ducassée. Coca: an automated debugger for C. In *Proceedings of ICSE'98, International Conference on Software Engineering*, pages 504–513, 1999.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[11] Jeffrey K. Hollingsworth, Barton P. Miller, Marcelo J. R. Goncalves, Oscar Naim, Zhichen Xu, and Ling Zheng. Mdl: A language and compiler for dynamic program instrumentation. In *Proceedings of PACT'97, Conference on Parallel Architectures and Compilation Techniques*, pages 201–213, 1997.

[12] Clinton L. Jeffery and Ralph Griswold. A framework for execution monitoring in Icon. *Software – Practice & Experience*, 24(11):1025–1049, November 1994.

[13] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of ECOOP'01, European Conference on Object-Oriented Programming*, pages 327–355. Springer-Verlag (*LNCS* 2072), 2001.

[14] Richard J. Moore. A universal dynamic trace for Linux and other operating systems. In *Proceedings of USENIX Annual Technical Conference, FREENIX Track*, pages 297–308, 2001.

[15] Jonathan Polley, Dionysys Blazakis, Jonathan McGee, Dan Rusk, John S. Baras, and Manish Karir. ATEMU: A fine-grained sensor network simulator. In *Proceedings of SECON'04, IEEE Conference on Sensor and Ad Hoc Communications and Networks*, 2004.

[16] Theodore A. Roth. Simulavr: an AVR simulator. http://savannah.nongnu.org/projects/simulavr.

[17] Ariel Tamches and Barton Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proceedings of OSDI'99, Symposium on Operating Systems Design and Implementation*, pages 117–130, 1999.

[18] Kevin Templer and Clinton L. Jeffery. A configurable automatic instrumentation tool for ANSI C. In *Proceedings of ASE'98, Automated Software Engineering*, pages 249–259, 1998.

[19] Ben L. Titzer, Daniel K. Lee, and Jens Palsberg. Avrora: Scalable sensor network simulation with precise timing. In *Proceedings of IPSN'05, International Conference on Information Processing in Sensor Networks*, 2005. To appear.