Timing Analysis for Sensor Network Nodes of the Atmega Processor Family *

Sibin Mohan[1], Frank Mueller[1], David Whalley[2] and Christopher Healy[3]

[1] Dept. of Computer Science, Center for Embedded Systems Research,
North Carolina State University, Raleigh, NC 27695-7534, mueller@cs.ncsu.edu
[2] Dept. of Computer Science, Florida State University, Tallahassee, FL 32306, whalley@cs.fsu.edu
[3] Dept. of Computer Science, Furman University, Greenville, SC 29613, chris.healy@furman.edu

Abstract

Low-end embedded architectures, such as sensor nodes, have become popular in diverse fields, many of which impose real-time constraints. Currently, the Atmel Atmega processor family used by Berkeley Motes lacks support for deriving safe bounds on the WCET, which is a prerequisite for performing real-time schedulability analysis. Our work fills this gap by providing an analytical method to obtain WCET bounds for this processor architecture.

Our first contribution is to analyze both C and NesC code, the latter of which is unprecedented. The second contribution is to model control hazards and variable-cycle instructions, both handled more efficiently by our approach than by previous ones and results in up to 77% improvement in bounding the WCET. The results demonstrate that our timing analysis framework is able to tightly and safely estimate the WCET of the benchmarks while simulator results are shown to not always provide safe WCET bounds. While motivated by the Atmel Atmega series of processors, results are equally applicable to low-end embedded processors.

This work is, to the best of our knowledge, the first set of experiments where timing results are contrasted from execution on an actual processor, from a cycle-accurate simulator and from a static timing analyzer. Furthermore, making our timing analysis toolset available to the Atmel Atmega processor family is a significant contribution towards addressing a documented need for tool support for sensor node architectures commonly used in networked systems of embedded computers, or so-called EmNets.

1. Introduction

Networked systems of embedded computers, referred to as EmNets [10], usually consist of one or several computationally powerful nodes called *base stations* and a large number of inexpensive, low capacity nodes called *sensors* (or *sensor nodes*). The nodes in a wireless sensor network communicate through wireless links, typically with rather limited bandwidth. Such EmNets have extensive applications ranging from civilian to military operations, many of which are subject to timing constraints: On the one hand, applications themselves generally pose requirements on the frequency of sensing; on the other hand, constraints on energy consumption and effective communication are also subject to timing constraints and clock synchronization assumptions. In recent years, low-end sensor nodes, especially the ones from Berkeley [16], have become popular and they find applications in diverse fields. These sensor nodes use the Atmel Atmega series of processors [2]. Such sensor nodes and networks composed of these nodes find applications in real-time environments.

Research on timing constraints for sensor node architectures is still preliminary; except for ad-hoc solutions specific to applications, there is a lack of software support for handling timing constraints. This paper addresses this issue by providing automated tool support for determining bounds on the worst-case execution time (WCET) of programs, tasks and even smaller program scopes for applications on Atmel architectures. These WCET bounds are particularly useful in real-time applications for EmNets.

In real-time systems, particularly hard real-time systems, violations of temporal constraints may have irreparable effects on the system, its environment, or both. Real-time systems theory reasons about the schedulability of task sets, *i.e.*, by performing offline schedulability tests, which can determine if all deadlines of a set of tasks can be met. Task parameters have to be known beforehand, *i.e.*, parameters such as period of each task, the worst-case execution time (WCET), and so on. Periods of tasks are determined from the operating environment, such as temporal constraints on sensors, actuators and other parts of the system. Determining the WCET for tasks is a non-trivial effort due to software complexity, non-determinism of inputs and hardware complexity with unpredictable execution behavior.

Various approaches to determine the WCET of tasks exist, such as experimental methods, which are considered unsafe or constrained to probabilistic analysis [30, 4]. Static analysis methods have also been developed to derive safe

* This work was supported in part by NSF grants CCR-0208581, CCR-0310860, CCR-0312695, EIA-0072043, CCR-0208892, CCR-0312493 and CCR-0312531.

WCET bounds, and they model hardware components, *e.g.*, the processor pipeline, caches, etc. They model the flow of code through various hardware components and use interprocedural program representation and longest control-flow paths to obtain an upper bound on the number of cycles for any execution.

In this paper, we present a timing analysis framework that analyzes programs, such as EmNets applications, for the Atmel architecture to obtain worst-case execution times. We present a unique approach in that a three-tiered system of experiments is carried out to verify the correctness of the WCET numbers obtained using our timing analyzer. First, we obtain execution times from the actual Atmel hardware itself. Second, we obtain execution times from a cycle-accurate simulator from the processor manufacturer. Finally, we run our benchmarks through our timing analysis framework to obtain WCET estimates. Our timing analysis framework is able to tightly and safely estimate the WCET for the various benchmarks. We also show that the methods used are scalable. When the input set sizes are scaled, the WCET estimates closely follow the actual execution cycles of the processor.

We have also adapted our framework to provide WCET estimates for NesC [11], the preferred language for programming sensor network applications. We are able to obtain accurate WCET estimates for both – NesC code as well as C code. We have also created an abstract NesC interface, which makes timing analysis of any piece of synchronous NesC code very straightforward. To the best of our knowledge, the interoperability of our timing analysis framework with NesC is unique.

The timing analysis framework we use was initially developed for the Micro SPARC I [29] architecture. It provides accurate and tight WCET estimates comparable to other WCET methods and tools available. Since the Atmel architecture is simpler than the SPARC in that it does not have caches and that it has only two pipeline stages, we believed that a simple one-to-one port was all that was necessary to adapt it for the Atmel architecture. Contrary to our intuition, we found that important enhancements were required even for a simple architecture as the Atmel Atmega to obtain tight WCET bounds as is demonstrated in the following. Each singular contribution resulted in 5% to 40% improvement in accuracy of bounding the WCET.

One significant problem faced by WCET tools is to cope with instructions that take a variable number of cycles to execute. Certain instructions may take a different number of cycles, depending on the the state of the system, such as status of a condition code register, or whether a branch is taken or not, and so on. Traditionally, such instructions cause WCET tools to over-estimate the WCET numbers, as they always assume the largest possible number of cycles for such instructions. This typically results in large overestimations for any applications when variable-cycle instructions are found in loops. We solve this problem by enhancing the path analysis to accurately account for the overhead of variable-cycle instructions depending on the actual execution context. Hence, we are able to obtain 30-40% tighter bounds on the WCET for Atmel processors.

While motivated by the Atmel architecture, these improvements transfer equally to other low-end embedded architectures that commonly have control hazards and variable-cycle instructions.

One of the lessons learned is that the design of static timing analysis tools should be accompanied by verifications of correctness for a set of benchmarks before applying these techniques to larger applications. Verification can be realized by comparing to a cycle-accurate simulator or actual embedded hardware, as demonstrated by our experiments.

2. Static Timing Analysis

Real-time schedulability analysis for any hard real-time system requires the WCET to be known beforehand and safely bounded. This is so that the feasibility of scheduling a task set can be determined given a scheduling policy, such as rate-monotone or earliest-deadline-first scheduling [19]. If dynamic timing analysis methods were used based on experimental or trace-driven approaches, the safety of the WCET values obtained cannot be guaranteed [30]. Also, it is difficult to determine worst-case input sets for even moderately complex tasks such that the WCET may be obtained, and performing exhaustive testing over wide ranges of the input space is infeasible, except for trivial cases. Even if the worst-case input set is known, interactions between hardware and software might inhibit programs from exhibiting their worst-case behavior. Architectural complexity is the main reason for such unpredictability, in particular complex pipelines and caching mechanisms.

The alternative to dynamic timing analysis, of course, is Static Timing analysis. While various approaches have been studied and proposed, we constrain ourselves to the toolset we use, without loss of generality [13, 22, 32]. WCET bounds obtained by static timing analysis guarantees the upper bounds on computation times of tasks. To achieve this, static timing analysis models the traversal of all possible execution paths in the code and determines timing independent of program traces or values of program variables. Loop bodies require only few traversals,and the worst-case behavior of the loop is obtained by an efficient fixed-point approach. The behavior of architectural components along execution paths are captured as the execution paths are traversed. Paths are composed to form functions, loops, etc. Finally, the entire application is covered to derive a bound on the worst-case execution cycles (WCEC) and the WCET.

An overview of the organization of this timing analysis framework is illustrated in Figure 1. Modifications to an optimizing compiler result in the production of control-flow

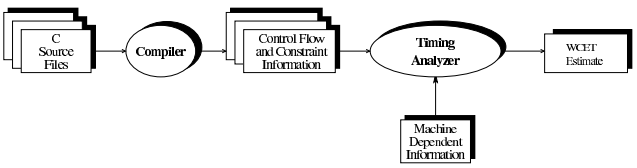

Figure 1. Static Timing Analysis Framework

and branch constraint information as a side effect of the the compilation process. The GCC compiler for the AVR instruction set architecture (ISA) is used to compile real-time applications into assembly code. Control-flow graphs and instruction and data references are extracted from this assembly code. A prerequisite for performing static timing analysis is that upper bounds on the number of iterations performed by various loops in the program be provided, either through program analysis or provided by the user.

The timing analyzer uses the control flow, the constraint information, and architecture-specific information (*e.g.*, pipeline characteristics) to calculate bounds on the WCET. While the analysis framework also supports static analysis of caches and modeling of static branch prediction [13, 1], these components are omitted here since the Atmel AVR processor family does not support them.

The control-flow information, the loop-bounds, optional caching categorizations, and the pipeline descriptions are used by the timing analyzer, to derive WCET bounds. The effects of structural hazards (instructions dependencies due to constraints on functional units resulting in stalls), data hazards (load-dependant instruction stalls if a use immediately follows a load instruction), branch prediction and cache misses (derived from caching categorizations) are considered by the pipeline simulator for each execution path through a loop or a function. Static branch prediction can be easily accommodated by WCET analysis - the misprediction penalty is added to the non-predicted path. Path analysis, explained below, selects the longest execution path. Once timing results for alternate paths in a loop are available, a fixed-point algorithm that quickly converges in practice is employed to safely bound the time for all iterations of the loop based on the loop body's cycle counts.

Path analysis for only a few iterations provides the necessary data for the fixed-point algorithm. If the longest path for the first iteration has been determined, the next-longest path for the next iteration can be determined, and this difference may occur only due to caching effects, if any. Lengths of these paths monotonically decrease due to the above-mentioned cache effects, and once a fixed-point is reached, subsequent loop iterations can be safely approximated by this fixed-point timing value, as shown in [13]. While combining the longest paths of consecutive iterations, care must be taken to account for overlap between the tail of the preceding path and the head of the succeeding path. Not considering this overlap is tantamount to draining the pipeline between iterations.

The timing analyzer ultimately derives WCET bounds using this fixed-point approach, first for each path, then for each loop, and finally for each function. A timing analysis tree is constructed, such that each loop or functions corresponds to a node of the tree. The tree is processed in a bottom-up manner, *i.e.*, the WCET for an outer loop nest/caller function in the tree is not calculated until the times for all of it's inner loop nests/callees are known. Hence, the timing analyzer predicts the WCET for tasks by first analyzing lower-level loops and functions before proceeding to higher-level loops and functions, eventually reaching the root of the tree (*e.g.*, main()). The timing analysis tree provides a convenient mechanism for obtaining the WCET for specific scopes, in particular for sub-tasks. The material in this section shows that static timing analysis is a non-trivial task, even for a simple architecture such as that of the Atmel processors.

3. Adapting to Architectural Features

While the timing analysis framework was being adapted to the Atmel Atmega architecture, several enhancements to the existing framework were deemed necessary. While the original framework provided safe WCET bounds, they were not necessarily tight bounds. A number of important enhancements resulted in significant improvements in terms of obtaining tight WCET bounds:

Variable-cycle instructions: Certain instructions take a variable number of cycles depending on the context of their execution. We adopted our framework, specifically the path-merging logic, to account for these instructions as explained in Subsection 3.1.

Pipeline overlap handling for adjacent loop iterations: The pipeline structure of adjacent loop iterations must be matched so to reflect the processor pipelines without unnecessary stalls. During path analysis, pipeline information is calculated for traces of straight-line code, *i.e.*, sequences of basic blocks, so-called paths. When considering paths of consecutive loop iterations, timings of each pipeline stage for the first and last instruction in a path need to be considered. The objective is to compose the trailing pipeline step-curve of a path with the leading step-curve of the next path. This composition was being handled incorrectly for the Atmel Atmega architecture. We performed modifications to eliminate unnecessary stalls that were affecting the tightness, albeit not the correctness of the WCET bounds, as explained in Subsection 3.2.

3.1. Variable-Cycle Instructions

Consider a simple conditional branch instruction where a branch is either taken or not taken. Let us assume that if the branch is taken, it completes in two cycles; if it is not taken, the branch completes in one cycle. The reason for the difference is given by the overhead imposed on resolving the target of the branch. Traditional static timing analysis meth-

ods would assume worst-case behavior and estimate that the branch would execute in two cycles, irrespective of the behavior of the instruction. This would lead to the WCET not being tight enough and could lead to gross over-estimation, especially if branches are embedded within loops, which is inevitable for the Atmel Atmega architecture. Such behavior is typically observed for instructions that modify the control flow of the program.

We have enhanced the timing analysis framework to take this into account and estimate the number of execution cycles required by the branch instruction correctly. As explained before, once the timing of alternate paths is complete, the fixed-point algorithm takes over and works towards convergence. At this stage, alternate paths created by instructions that modify the control flow of the program are analyzed. By analyzing alternate paths created as the result of a single instruction's modification of the control flow, we can accurately determine the number of cycles taken by that instruction. For example, the above branch instruction would take one cycle to execute in the path that the execution falls through (branch not taken) and two cycles in the path that does not fall through (branch taken). Hence, an adjustment can be made to one of the alternate paths so that it reflects the true nature of execution. Suppose it had been decided that we always assign two cycles to the branch instruction. We would then reduce the WCEC of the path in which the branch is not taken by one so that it is equivalent to the branch instruction taking one cycle to execute. This is shown in Figure 2. The fixed-point algorithm would now analyze the two paths and accurately obtain WCET information. This enhancement to the static timing analysis framework results in tighter, more accurate WCET estimates for the tasks being examined.

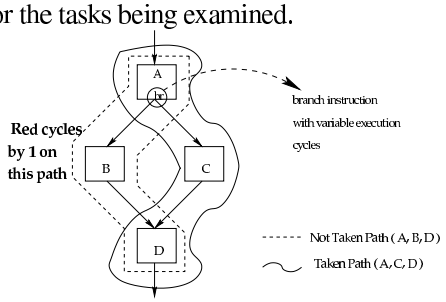

Figure 2. Modeling Instructions with Variable Execution Times

3.2. Pipeline Modeling across Loops Iterations

When timing analysis is performed for loop bodies, care must be taken to ensure that the pipeline structure of adjacent loop iterations is being correctly composed, as shown in Figure 3(b). Suppose the two pipeline structures were matched incorrectly, as shown in Figure 3(c). The WCET estimates obtained would be inaccurate and not representative of the execution on the real architecture. In most cases, the errors would accumulate over all loop iterations, and the WCET estimate obtained would be a gross overestimation. This is the case for the example shown in Figure 3(c)(i). Even more seriously, an unsafe underestimation may occur if only the leading edge (IF stage) of the pipeline is considered for composition, as shown in Figure 3(c) (ii). While we did not observe the latter case, the former case was encountered as a result of the initial porting efforts of our analysis framework. Our timing analysis framework was composing by overlapping stages, but only to a limited extent. We observed that an extra cycle was being introduced at the end of every iteration for certain loops. The composition was enhanced by improving the logic that performs the analysis for adjacent loop iterations to remove the extra cycle.

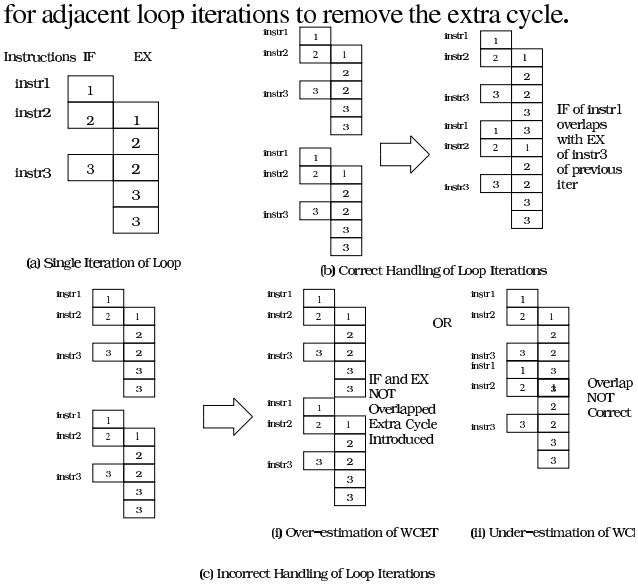

Figure 3. Composing Pipeline Behavior of Adjacent Loop Iterations

4. Features of the Atmega Architecture

We shall limit our discussion to the Atmega128 / Atmega103 processors, two low-power CMOS 8-bit micro controllers based on the AVR enhanced RISC processors.

Architectural Model: The AVR uses a Harvard architecture with separate memories and buses for program and data. Instructions located in the program memory are executed with a single level of pipelining. As one instruction is being executed, the next instruction is fetched, *i.e.*, instructions execute every clock cycle. Thus, the pipeline has just two stages – Instruction Fetch (IF) and Execute (EX).

The AVR processors have two main memory spaces, the Data memory and the Program memory space. It also features an EEPROM memory for data storage. All three memory spaces are linear and regular. The Atmega128 chip has 128kB of on-chip in-system programmable flash memory for program storage, and is organized as 64K x 16 bits. AVR processors do not have a cache memory.

Instruction Set: All AVR instructions are either 16 bits or 32 bits wide. This is the reason why the flash is orga-

nized as 64K x 16. The AVR instruction set supports a variety of addressing schemes for a total of thirteen different addressing modes.

All operations are integer-based. The AVR instruction set does not support floating-point operations but requires their emulation in software. All instructions take either one or two cycles to execute, except certain types of instructions, and their timing is listed in Table 1.

Instructions	Function	Exec. Cycles
rcall, icall, call	Subroutine calls	3/4 *
eicall	Extended indirect call	4
ret, reti	subroutine returns	4/5 *
cpse	compare, skip if equal	1/2/3 *
sbrc, sbrs, sbic, sbis	skip if bit is set/clear	1/2/3 *
brxy	conditional branches	1/2 *
lpm, elpm	load program memory	3

Table 1. Timing of AVR Instructions: (* denotes variable execution times)

The instruction set also has a rich set of conditional branch instructions. All conditional branch instructions either take two registers as operands or operate based on the status of flag bits. It also has certain *skip* instructions, which decide whether to skip the next instruction or not based on whether certain flags are set or clear. All load/store instructions directly access memory. Hence, they take a fixed number of cycles. We assume that every memory reference is for a reference in the internal memory and, hence, takes two cycles to complete. This assumption is consistent with the programming environment for the Atmel Atmega family. Thus, the memory latency is bounded and constant as opposed to other conventional processors. This makes the task of Timing Analysis easier.

We see in Table 1 that certain instructions take a variable number of cycles.

- The *call* instructions take four or five cycles based on the PC size used. For a 16 bit PC, these instructions take 4 cycles; for a 22 bit PC, they take 5 cycles.
- The same overhead accounting applies to the *ret* instruction.
- The *cpse, sbrc, etc.* instructions use a varying number of cycles based on two factors: Result of evaluation of the condition (true/false) and the size of the instruction to be skipped. For example, if the condition is false, then they take one cycle. If it is false and the instruction to be skipped is one word in size, then they take two cycles. And if it is false and the instruction to be skipped is two words long, then they take 3 cycles. All branch instructions take either one or two cycles, based on whether the branch is not taken, or taken, respectively.

The AVR instruction set also includes many logical as well as bit-operational instructions.

Timing Analysis for AVR Processors: Table 1 illustrates that only certain instruction types exhibit timing characteristics that are different from the majority. Most other instructions either take one or two cycles to execute. For these instructions, we can divide them into instructions that take one cycle and instructions that take two cycles. Once we determine which of these two categories an instruction falls into, instructions of the same category can be treated identically and the timing added up. For the instructions in Table 1, special handling is required, particularly for those that have variable execution times. These instructions are handled after path analysis, during the path-merging stage, as explained in Section 2.

Since the AVR processors do not have a cache, *all* instructions are treated as if they were hits in terms of their timing behavior. Since the memory latency is fixed, we see that this simplification captures the behavior of the system accurately. Loop bounds for the various loops, if any, in the programs are provided as input to the timing analysis framework. A description of the simple pipeline of the AVR processors was also provided as input to the framework to port the pipeline simulator of the timing analyzer to the AVR processor family.

5. Experimental Setup

We have used a three-tiered experimental setup to perform and validate our results. To our knowledge, this is the first time that such an approach has been utilized. The first set of experiments was carried out on the MICA Berkeley Sensor nodes [16], which utilize the Atmel Atmega 128L micro-processor. The experiments were conducted by providing worst-case inputs to the programs. The second set of experiments was carried out on a cycle-accurate simulator, *AVR Studio*, supplied by the processor manufacturer. The final set of experiments was based on static analysis with the timing analysis framework.

We chose five benchmarks from the C-Lab embedded WCET suite for the experiments [5], which is widely used for assessing the capability of timing analysis tools. We further analyzed three NesC programs, one synthetic benchmark and two commands from the TinyOS security layer, as depicted in Table 2.

The "ArraySum" benchmark (NesC) is a synthetic benchmark that calculates the sum of the elements of an integer array. "RC5.encrypt" and "RC5.decrypt" are the RC5 encryption and decryption functions, respectively, part of the SPINS security layer of TinyOS [24]. Section 6 provides details on timing analysis for NesC.

All worst-case measurements are expressed in terms of processor cycles. To obtain accurate timing results on the hardware, we used interrupt-driven routines activated by hardware counter overflows. At the start of the execution

C Benchmark	Function
sum array	Sum and count of positive and negative numbers in an array.
fibcall	Generate the n^{th} Fibonacci number.
insertsort	Implementation of Insertion Sort.
matrix mult	Matrix Multiplication.
bubble sort	Implementation of Bubble Sort.
NesC Benchmark	**Function**
ArraySum	Sum of numbers in an array.
RC5.encrypt	RC5 encryption
RC5.decrypt	RC5 decryption

Table 2. Benchmarks used in Experiments

of the benchmark, we initialize a pair of 16-bit counters to zero and allow one of them to increment by one on every cycle (cycle counter). If this counter overflows, we increment the second counter (overflow counter) and reset the first counter. Thus, when execution of the benchmark completes, the combination of the two counters gives us the total time for the benchmark. Of course, considerations have to be made for the overheads of the timing code itself. We explain this in some detail below: Let

$O_1 = overhead\ for\ starting\ and\ stopping\ timers\ [cycles]$

$O_2 = overhead\ per\ invocation\ of\ overflow\ handler\ [cycles]$

$x = value\ of\ cycle\ counter$

$y = value\ of\ overflow\ counter$

We note that an overflow occurs once every *65,536* cycles, because we use a 16 bit counter as the cycle counter.

Then, the total execution time for the benchmark is obtained as follows:

$$total_time = y * 2^{16} + x$$

Accounting for the start and stop overhead, we get:

$$wcet' = total_time - O_1$$

Now, accounting for the overflow handling overhead, we obtain our final WCET estimate:

$$wcet = wcet' - (y * O_2)$$

We account for the overhead of timing code in results for the hardware experiments. Once this has been taken care of, the code used on all three platforms is identical.

The cycle-accurate simulator has its own interactive GUI, which provides various processor statistics, such as execution cycles and execution time. To obtain timing information from the simulator, we first feed it the code identical to that used for the hardware experiments. We then set breakpoints before and after significant points in the code and calculate the difference in the execution time, as shown on the GUI, to obtain the WCET (or, more precisely, the WCEC in this case). For both of the above, worst-case input sets were manually determined for the execution so that the programs may exhibit worst-case behavior.

Our final set of experiments was to run the benchmarks through the timing analysis framework. The same assembly files that were executed on the hardware and the simulator was provided to the timing analysis framework along with the loop bounds for the various loops in the code. The instruction categorizations were hardwired as *always hits*. The control-flow information was extracted by a preprocessing tool, similar to a compiler back-end, which is part of the framework. The results, along with pipeline information, were fed into the timing analyzer, which decomposed the program into paths and performed WCET analysis as explained in Section 2.

Results from all three sets of experiments were tabulated. The results themselves and related analysis are provided in the next section. We also conducted experiments to study the effects of varying input sets and loop bounds on the WCET estimates produced by the timing analysis framework.

Instructions Extraneous to Loop Bodies: The timing analysis framework calculates the WCET for loops, functions and the main program. It does not provide WCET estimates for single instructions or arbitrary regions of code. The timing obtained from the hardware and the simulator can be for any arbitrary piece of code. Hence, there can be a mismatch between the results obtained from the timing analyzer and the hardware and/or the cycle-accurate simulator. Even with careful placement of the timing code for the hardware and breakpoints for the simulator, differences can occur. The main reason for the mismatch is code that is extraneous to the loops but still integral to the execution, such as loop initialization code, code inserted to pass arguments to functions, and so on, as shown in Figure 4. Care must be taken to adjust for the timing of these extra instructions in the final results obtained from the hardware/simulator. A similar problem is given by the overhead of reading the timer and handling counter overflow exceptions on the actual hardware, which incurs overhead. The former poses only a small, constant overhead while the latter overhead aggregates over longer executions. We explicitly compensate for both of these effects when comparing timings from the actual hardware with the cycle-accurate simulator or the timing analyzer.

6. Timing Analysis for NesC

Typical NesC programs include a variety of constructs such as commands, events, tasks, etc. Even though our following description for statically deriving WCET bounds focuses on commands, the methodology is equally applicable to the timing of the body of events once these events have been triggered, as is required by schedulability analysis of synchronous and asynchronous activities in real-time systems [20]. NesC commands are analogous to C functions in that they execute code synchronously. Typically, the NesC compiler converts given NesC code to intermediate C code

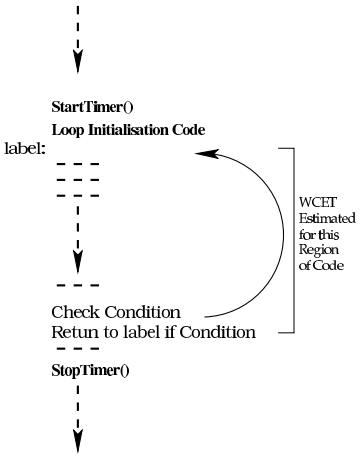

Figure 4. Timing Adjustment for Extra Cycles (Bold: Code only Present when Timing Executions on Hardware)

and then builds an executable for the AVR motes. Since we deal only with synchronous commands, the C code generated can be examined and information, such as loop bounds, can easily be obtained. The resulting intermediate AVR assembly code and loop bounds information obtained from the C file provide the inputs to the static timing analysis framework (see Section 2).

For actual execution on the hardware, we again add calls to our hardware timer routines in the NesC code so that we obtain accurate execution times on the hardware. Adjustments are made for the overhead of calling the timing routines.

To facilitate the timing NesC code, we have created a simple interface in NesC, which provides an abstraction for executing the actual benchmark. We time the call to the *execute()* method of the interface. Any code that executes within this method or is called from this method is timed, both in hardware and by the timing analyzer. Since we abstract from the actual benchmark, we can time various NesC modules as long as they implement a facility similar to the execute() interface and contain commands.

7. Results

The results from the three sets of experiments are summarized in Table 3. The execution times for these benchmarks range from a few hundred cycles to several million cycles. Results for the Mica Motes show execution cycles for the benchmarks before handling of the loop-extraneous instructions and after the extra cycles have been accounted for in columns titled "Before Adjustment" and "After adjustment" under the "Mica Motes" results, respectively.

The upper part of Table 3 shows the results for the C benchmarks whereas the lower part of the table shows results from the execution and analysis of NesC benchmarks as explained in Section 6.

The column titled "Before Adjustment" under "Simulator", depicts the raw results from the simulator whereas the column titled "After Adjustment" takes the same overheads as before into account to adjust the raw numbers. The column "Ratio" under "Simulator" depicts the ratio between adjusted simulator and adjusted Mica results. The remaining columns depict the results of the timing analyzer. They demonstrate increasingly tight WCET estimates as various special cases were handled (see Section 3). The initial results in columns titled "Initial Results" and "Ratio" under "Timing Analyzer" indicate the WCET estimates obtained from the timing analyzer *before* any of the special cases were handled, both in cycles and as a ratio relative to the adjusted Mica numbers. The column titled "After Pipeline Fix" and the following "Ratio" column show the results after adjacent loop iteration were adjusted to remove pipeline stalls between paths (see Subsection 3.2). Finally, the column "After Variable Instruction Fix" and the succeeding "Ratio" column report the WCET estimate obtained from the timing analysis framework obtained after the timing analyzer's path-merging logic was enhanced to account for instructions that have variable execution times (see Subsection 3.1). All "Ratios" indicate the ratio of the execution cycles in the preceding column to the corresponding entries of execution cycles "After Adjustment" for "Mica Motes".

Let us first focus on the results regarding the timing analyzer. The original timing analyzer framework reported WCET estimates for the AVR architecture. However, these results were not tight. By studying the architecture and the instruction set, we were able to identify a number of cases (see Section 3) that required enhancements of the timing analyzer. After adding the logic to ensure proper overlapping of adjacent loop iterations, WCET estimates became much tighter (Column "After Pipeline Fix" Table 3). However, the WCET estimates were still being over-estimated. After addressing shortcomings in the handling of instructions with variable execution times, the final results (Column "After Variable Instruction Fix" of Table 3) show very tight and close estimates of the WCET compared to the execution numbers obtained from the hardware. Not only were we able to obtain tight WCET estimates for the AVR architecture, we also enhanced the capabilities and the value of our timing analysis framework in the process.

The execution on the Mica hardware resulted in nearly identical results to the cycle-accurate simulator. However, we see that the simulator may slightly underestimate the WCET, which, however small, is not safe in a real-time environment. This differences in timing between the hardware and the simulator is fully repeatable. Hence, the "cycle-accurate" simulator is not entirely fit for usage in the context of real-time schedulability analysis as a base to obtain the WCET of tasks. Hence, we confirm unsafe differences between simulators and hardware reported in prior work,

C Benchmark	Mica Motes		Simulator			Timing Analyzer					
	Before Adjustment	After Adjustment	Before Adjustment	After Adjustment	Ratio	Initial Results	Ratio	After Pipeline Fix	Ratio	After Var. Instr. Fix	Ratio
sum array	141,524	141,500	141,521	141,497	0.99	161,498	1.14	141,500	1.00	141,600	1.00
fi bcall	151	145	146	140	0.96	258	1.78	202	1.39	146	1.01
insertsort	1,629	1,613	1,622	1,606	0.99	1,978	1.23	1,880	1.17	1861	1.15
matrix mult	1,851	1,845	1,848	1,842	0.99	2,318	1.26	2070	1.12	1,878	1.01
bubble sort	3,628,249	3,628,239	3,628,249	3,628,239	1.00	3,900,998	1.08	3,650,000	1.01	3,776,518	1.04
NesC Benchmark	Mica Motes		Simulator			Timing Analyzer					
	Before Adjustment	After Adjustment	Before Adjustment	After Adjustment	Ratio	Initial Results	Ratio	After Pipeline Fix	Ratio	After Var. Instr. Fix)	Ratio
ArraySum	86	81	97	92	1.14	105	1.30	87	1.07	88	1.09
RC5.encrypt	15,956	15,951	15,951	15,946	1.00	17,958	1.13	16,088	1.00	16,088	1.00
RC5.decrypt	15,860	15,855	15,855	15,850	1.00	17,982	1.13	16,112	1.01	16,122	1.01

Table 3. Table of Results from all Experiments.

which has been attributed most likely to minor omissions in the accuracy of architectural model within simulators [21].

The results also demonstrate that the timing analyzer is able to *tightly* and *safely* bound the WCETs for the benchmarks tested. We see that for most of the benchmarks the WCET estimates closely match the execution times obtained from the hardware and the simulator. For three of the five benchmarks, the timing analyzer safely overestimates by less than 2% relative to the actual execution.

Furthermore, timing analysis for NesC code is accurate. The results from the timing analyzer match hardware execution times very closely. Differences are within 9% for the synthetic array summation benchmark and nearly identical (1% or less) for the encryption and decryption algorithms.

For the experiments with varying input sizes, we conducted experiments to assess the scalability of our approach for the *fibcall* benchmark. We compare results from the timing analyzer against results from the hardware, as depicted in Figure 5.

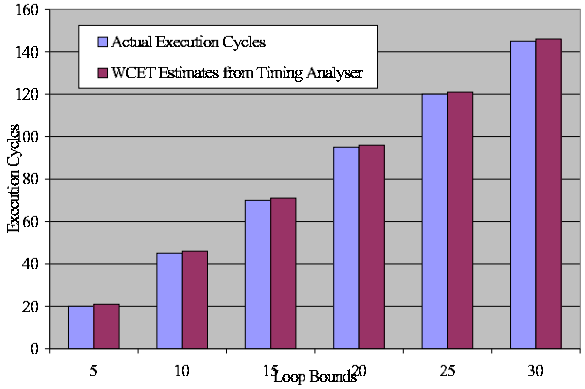

Figure 5. Scalability of Timing Analyzer for Varying Input Sizes

The figure shows that the tightness of WCET bounds does not deteriorate with increasing input sizes. We observe a constant, small overestimation of the actual execution times by the WCET estimates. Hence, our framework is not only reliable and tight for a wide range of programs, but it is also independent of varying input sizes and the resulting changes in the number of iterations.

The three-pronged approach to performing these experiments underscores the reliability of our framework. This also shows the need for verifying timing analyzers against either actual hardware or cycle-accurate simulators, or even both. In fact, a comparison against only a cycle-accurate simulator can sometimes lead to slight underestimations, as seen in the results. Hence, we favor a comparison with the actual hardware over a cycle-accurate simulator, even if the latter is supplied by the manufacturer. Overall, the WCET bounds obtained by our timing analyzer are both safe and tight as they closely follow actual execution times.

Let us discuss the merits of the three approaches in a more general sense. While obtaining cycles from the actual hardware provides the actual data needed, cycle level simulators can also be useful. First, problems with a timing analyzer can be compared to diagnostic output from a simulator to track down problems regarding inaccurate estimations. In contrast, only limited information regarding the execution can be typically obtained from hardware, such as the number of cycles. Second, simulators are often easier to interact with than actual hardware. Tests are typically easier to setup and devise with a simulator. Finally, sometimes the actual hardware may not be available. A simulator may be executed on different general-purpose processors.

Ideally, a cycle-accurate simulator should be used to verify the accuracy of a timing analyzer and the actual hardware should be used to verify the accuracy of the simulator. But our study shows that one should not blindly rely on so-called cycle-accurate simulators.

8. Related Work

Many research groups have addressed the the problem of timing analysis. Methods of analysis range from unoptimized programs on simple CISC processors over optimized

programs on pipelined RISC processors and uncached architectures to instruction and data caches [23, 25, 12, 18, 15, 22, 32, 17]. In fact, our work is probably most closely related to that of Harmon *et al.* [12] in terms of comparing the actual hardware. However, instead of using time-consuming reverse engineering methods (requiring error-prone data acquisition with, *e.g.*, an oscilloscope), we demonstrate that data sheets together with observing actual hardware may be sufficient for the design of a timing analyzer. These methods are used to obtain safe bounds on the WCET as discrete values in a non-parametric fashion.

Parametric timing analysis by Vivancos *et al.* describes techniques to handle variable loop bounds [28]. This allows dynamic schedulers to re-assess the WCET based on loop bounds determined dynamically during program execution. In independent work, path expressions were used to combine a source-oriented parametric approach of WCET analysis with timing annotations, verifying the latter with the former, in work by Chapman *et al.* [6]. Bernat and Burns recently proposed algebraic expressions to represent the WCET of programs [3].

The FAST framework by Seth *et al.* models processor frequency and incorporates parametric timing into the analysis framework [26]. Probabilistic and experimental approaches to obtaining the WCET for programs have also been proposed [30, 4].

Our timing analyzer framework builds on the concepts of prior work [22, 32, 14, 31, 15]. Modifications and enhancements have been made to the framework to work with various different architectures and instructions sets. The AiT tool [27], loosely based on our early work, particularly that on caching, is a commercially available timing analysis tool that has been shown to provide tighter estimates for the Motorola Coldfire architecture employed in Airbus planes.

Chen *et al.* recently presented methods to perform static timing analysis for embedded software [7]. They utilize Integer Linear Programming concepts to obtain the WCET for an embedded architecture. They cope with more complex architectures supporting branch prediction and predication, which are not issues in the AVR architecture. In other related work, Engblom *et al.* target a line of custom ASICs based on a standard CPU core based on the Atmega90 line [9]. Prior work handles pipeline effects and effects of variable-cycle instructions in a different manner [8]. In that work, the timing of any single instruction provides a baseline. Then, all possible subsequences of instructions of different lengths are examined to obtain savings in execution time due to instruction overlap in the pipeline. These savings are expressed as negative delta values for any instruction sequence of length two, three etc. This method, although exhaustive, is extremely time-consuming. In contrast, our work shows that it is sufficient to capture edge-effects between basic blocks, and only if they occur, as in case of the existence of variable-cycle instructions. Hence, our method is considerably more efficient in terms of analysis overhead.

9. Conclusion

In this work, we enhanced existing tools and developed new tools to incorporate them into a framework capable of performing timing analysis with the aim of obtaining WCET estimates for the Atmel Atmega architecture family. The Atmel AVR architecture and instruction set was targeted for static timing analysis as this architecture is widely used in the Berkeley Motes architecture — an architecture that is becoming ever more popular for use in a wide variety of applications. The timing analysis framework is capable of statically analyzing code compiled for the Atmel processors. It provides tight, safe, and accurate WCET estimates, which forms the base for subsequent real-time schedulability analysis. To verify our results, we performed a series of experiments using a three-tiered approach, which is unique, to the best of our knowledge. Timing results from the actual hardware, from a cycle-accurate simulator and from our timing analysis framework were studied and compared. We found that the cycle-accurate simulator is capable of underestimating the WCET at times. Hence, it does not provide a safe base for WCET calculations.

In contrast, our timing analysis framework was able to calculate WCET cycles with an over-estimation of less than 2% for programs with statically known loop bounds. We address the handling of variable-cycle instructions, path merging without pipeline stalls and accurate accounting for timing overhead. Our approach extends beyond the Atmel AVR family as the modeled architectural features are common in low-end embedded processors. By enhancing the entire framework, significantly tighter bounds were obtained by our timing analysis framework. Specifically, the enhanced pipeline modeling improves the accuracy of WCET bounds by 5 to 20%. Our effient modeling of variable-cycle instructions further improves the accuracy by 30 to 40% for total improvements of up to 77%.

Our framework is also capable of performing static timing analysis for NesC commands. The WCET results obtained for both NesC and C code are accurate as well as tight. This shows that our framework is flexible enough to bound the WCET of sensor network applications in NesC as well as C for the Atmel architecture, both at the level required by schedulability analysis of real-time systems.

To our knowledge, this is the first time that such a three-tiered assessment of timing experiments has been carried out. It demonstrates short-comings in so-called "cycle-accurate simulators" while static timing analysis provided safe bounds on the WCET. Furthermore, making our timing analysis toolset available to sensor node applications may be a significant contribution towards addressing a documented need for tool support for EmNets.

References

[1] A. Anantaraman, K. Seth, K. Patil, E. Rotenberg, and F. Mueller. Virtual simple architecture (VISA): Exceeding the complexity limit in safe real-time systems. In *International Symposium on Computer Architecture*, pages 250–261, June 2003.

[2] Atmel. Atmel avr 8-bit risc family. http://www.atmel.com/products/avr/.

[3] G. Bernat and A. Burns. An approach to symbolic worst-case execution time analysis. In *25th IFAC Workshop on Real-Time Programming*, May 2000.

[4] G. Bernat, A. Colin, and S. Petters. WCET analysis of probabilistic hard real-time systems. In *IEEE Real-Time Systems Symposium*, Dec. 2002.

[5] C-Lab. WCET benchmarks. Available from http://www.c-lab.de/home/en/download.html.

[6] R. Chapman, A. Burns, and A. Wellings. Combining static worst-case timing analysis and program proof. *Real-Time Systems*, 11(2):145–171, 1996.

[7] K. Chen, S. Malik, and D. I. August. Retargetable static timing analysis for embedded software. In *Proceedings of the International Symposium on System Synthesis (ISSS)*, October 2001.

[8] J. Engblom. Processor pipelines and static worst-case execution time analysis. *Uppsala Dissertations from the Faculty of Science and Technology*, 2002.

[9] J. Engblom, A. Ermedahl, M. Sjdin, J. Gustafsson, , and H. Hansson. Execution-time analysis for embedded real-time systems. In *STTT (Software Tools for Technology Transfer) special issue on ASTEC.*, 2001.

[10] D. Estrin, G. Borriello, R. Colwell, J. Fiddler, M. Horowitz, W. Kaiser, N. Leveson, B. Liskov, P. Lucas, D. Maher, P. Mankiewich, R. Taylor, and J. W. (eds.). *Embedded, Everywhere, A Research Agenda for Networked Systems of Embedded Computers*. Computer Science and Telecommunications Board, National Academies Press, 2001.

[11] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The NesC language: A holistic approach to networked embedded systems. In J. J. B. Fenwick and C. Norris, editors, *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI-03)*, volume 38, 5 of *ACM SIGPLAN Notices*, pages 1–11, New York, June 9–11 2003. ACM Press.

[12] M. Harmon, T. P. Baker, and D. B. Whalley. A retargetable technique for predicting execution time. In *IEEE Real-Time Systems Symposium*, pages 68–77, Dec. 1992.

[13] C. A. Healy, R. D. Arnold, F. Mueller, D. Whalley, and M. G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1):53–70, Jan. 1999.

[14] C. A. Healy, M. . Sjödin, and D. B. Whalley. Bounding loop iterations for timing analysis. In *IEEE Real-Time Embedded Technology and Applications Symposium*, pages 12–21, June 1998.

[15] C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *IEEE Real-Time Systems Symposium*, pages 288–297, Dec. 1995.

[16] J. Hill, R. Szewczyk, A. Woo, D. Culler, S. Hollar, and K. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

[17] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *IEEE Real-Time Systems Symposium*, pages 254–263, Dec. 1996.

[18] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, and C. S. Kim. An accurate worst case timing analysis for RISC processors. In *IEEE Real-Time Systems Symposium*, pages 97–108, Dec. 1994.

[19] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. of the Association for Computing Machinery*, 20(1):46–61, Jan. 1973.

[20] J. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[21] S. Montan and J. Engblom. Validation of a cycle-accurate cpu simulator against real hardware. In *ECRTS (Euromicro Conference on Real-Time Systems)*, 2000.

[22] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2/3):209–239, May 2000.

[23] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–61, Mar. 1993.

[24] A. Perrig, R. Szewczyk, V. Wen, D. Culler, and J. D. Tygar. SPINS: Security protocols for sensor networks. In *Seventh Annual International Conference on Mobile Computing and Networks (MobiCOM 2001)*, pages 189–199, 2001.

[25] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1(2):159–176, Sept. 1989.

[26] K. Seth, A. Anantaraman, F. Mueller, and E. Rotenberg. Fast: Frequency-aware static timing analysis. In *IEEE Real-Time Systems Symposium*, pages 40–51, Dec. 2003.

[27] S. Thesing, J. Souyris, R. Heckmann, F. R. andM. Langenbach, R. Wilhelm, and C. Ferdinand. An Abstract Interpretation-Based Timing Validation of Hard Real-Time Avionics. In *Proceedings of the International Performance and Dependability Symposium (IPDS)*, June 2003.

[28] E. Vivancos, C. Healy, F. Mueller, and D. Whalley. Parametric timing analysis. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Embedded Systems*, volume 36 of *ACM SIGPLAN Notices*, pages 88–93, Aug. 2001.

[29] D. L. Weaver and T. Germond. *The SPARC Architecture Manual – Version 9*. Prentice Hall, 1994.

[30] J. Wegener and F. Mueller. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-Time Systems*, 21(3):241–268, Nov. 2001.

[31] R. White, F. Mueller, C. Healy, D. Whalley, and M. Harmon. Timing analysis for data caches and set-associative caches. In *IEEE Real-Time Embedded Technology and Applications Symposium*, pages 192–202, June 1997.

[32] R. T. White, F. Mueller, C. Healy, D. Whalley, and M. G. Harmon. Timing analysis for data and wrap-around fill caches. *Real-Time Systems*, 17(2/3):209–233, Nov. 1999.