LwProf: Lightweight Profiling and Coverage Tool for Embedded Software

Evan Lojewski[1] **and Kristen R. Walcott**[2]
[1]EM Microelectronic-US Inc., Colorado Springs, Colorado, USA, Evan.Lojewski@emmicro-us.com
[2]University of Colorado - Colorado Springs, Colorado Springs, Colorado, USA, kwalcott@uccs.edu

Abstract— *Determining test coverage can be prohibitively expensive in resource constrained embedded systems. Traditional methods for determining test coverage often cannot be considered due to the resulting memory impact when enabling coverage. Even if the binary did fit within the size limitations of the target system, the execution slowdown can significantly change the behavior of a real-time system, giving test results that may not reflect the final application.*

In this work, we develop a tool, `LwProf`*, that implements an optimized version of traditional profiling techniques for embedded systems. The tool is compared to existing coverage tools to determine the difference in efficiency and effectiveness. When compared to traditional methods,* `LwProf` *has a factor of two reduction in code size, a factor of four reduction in data size, and an order of magnitude improvement on the execution speed overhead, enabling coverage and profiling on existing embedded systems.*

Keywords: gcov, profiling, coverage

1. Introduction

Test coverage is a standard metric for determining the quality of a test suite. This metric is determined by measuring which parts of the code have executed, verses which parts have not. Similarly, coverage can be extended into profiling by measuring not only if the code has executed, but how often it executes. Standard tools for determining test coverage such as gcov and llvm-cov [9] result in binaries that are both larger and slower than binaries with coverage disabled. While this overhead may be palatable in standard software development projects, it becomes prohibitively expensive to measure coverage in resource constrained systems such as Application Specific Integrated Circuits (ASICs) with embedded processors.

When ASICs are developed, they are often stripped down to the minimum feature set required for the target application. This is done in order to reduce per chip production costs, specifically by minimizing the die area (chip size). Reducing the frequency of an ASIC is one method that is used to reduce the die size. Another technique is the removal of RAM, ROM, Flash, or other hardware blocks that will go unused in the final application [3].

Traditional coverage tools such as gcov and llvm-cov result in larger and slower binaries. As the ASIC has a reduced amount of memory, often the coverage tool will result in a binary that cannot be linked. Even if there is enough memory available, the performance overhead can result in an inoperable system. As such, traditional coverage tools cannot be used in many of these resource constrained systems.

Other tools, such as THeME [1], utilize advanced hardware features to achieve coverage results. By using the hardware features, changes to the binary can be reduced or removed, resulting in minimal size and speed impacts to the code. These tools however do rely on additional hardware, increasing the die size and as such the overall ASIC cost. Due to manufacturing-cost requirements on the ASIC, these features are removed, resulting in hardware-assisted coverage not being possible in many ASICs. Thus, standard test coverage tools cannot be used due to the limited resources on the ASIC.

Firmware for an ASIC may have a test suite. However, test suite quality cannot easily be determined in many cases due to the lack of profiling units or expensive coverage metrics. When software is located in ROM, the quality of the software is extremely important as complete mask sets for creating a chip can be on the order of $100,000 to $1,000,000 depending on the process node [4]. As such, an alternative method needs to be developed that works within these constraints from a business standpoint.

To solve this problem, we developed a new tool. The tool, `LwProf`, utilizes an optimized version of traditional coverage techniques. The tool supports both an extremely light weight coverage option as well as a profiling option that can be used on embedded systems. As a result of the optimizations implemented, `LwProf` results in a factor of two improvement in the code-size overhead, while also reducing the execution overhead by an order of magnitude. `LwProf` not only reduces the barrier for coverage on ASICs, but it can also be directly used with standard software due to the reduced overhead leading to reducing test time for the developer.

In summary, the main contributions of the paper are:

- A Description of the architecture and purposes of `LwProf`, given existing tools and their effectiveness (Sections 2 and 3)
- Details of the development and implementation of `LwProf` (Sections 3 and 4).
- Evaluation and analysis of existing tools in terms of efficiency and effectiveness (Section 5).
- Comparison of existing tools with `LwProf` (Sections 6 and 7).

2. Background

Traditional coverage techniques were developed to run on computers with full operating systems, large amounts of memory, and other resources. These coverage techniques work by modifying software during compilation to inject probe points into the application. The probe points, or instrumentation sites, cause a payload to be recorded. This is often done with a special option in a compiler that enables instrumentation code to be emitted into the final binary.

When the instrumentation sites are chosen appropriately, these tools are able to record line, branch, function, and other types of coverage metrics. Tools like gcov and llvm-cov add the instrumentation code by injecting function calls into the code as needed for the coverage metric desired. As an example, the instrumentation site for function-level coverage is the prologue of each function, while sites for branch coverage are the *then* and *else* clauses of

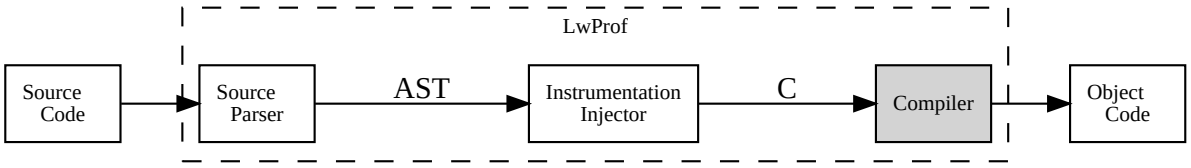

Fig. 1: The LwProf System Overview

each conditional. With traditional tools, the injected instrumentation calls into a coverage middleware library, recording that the instrumentation was executed successfully. Once the program has finished execution, the instrumentation payloads need to be saved and analyzed. These tools often perform file I/O operations as the program is exiting to record the coverage metrics to disk. All of these additional operations can result in an execution overhead of 10% to 30% [8].

Instead of a middleware library to record instrumentation sites, tools such as THEME [1] use hardware profiling units. Modern CPUs used within standard PCs as well as ARM-based phones include additional hardware that can enable hardware-assisted coverage and profiling. Once a profiling unit is setup, the execution path of a block of code can be recorded. Unlike traditional techniques, hardware-based coverage tools are able able to grab coverage information from unmodified software with some restrictions. Every time a branch is taken, the hardware unit can record the event. If a branch is not taken, however, the hardware unit will not record any information. This results in half of the branch decisions being lost. If the software is modified to include a dummy branch instruction in the fall-through case, then the hardware profiling can catch all of the decisions. The THEME [1] research showed that the additional dummy branch instruction had minimal impact on both size and speed. While this does reduce the software overhead, requiring hardware profiling units can impact the cost of ASICs due to die size increases.

3. Tool Architecture

In order to solve the coverage problems inherent with ASICs, we propose a new tool called LwProf. LwProf is designed to mitigate the costs incurred when using both transitional coverage based tools and hardware-assisted coverage tools. The overall architecture of LwProf can bee seen in Fig. 1.

In order to simplify the architecture of the tool, LwProf has three primary components: the Source Parser, the Instrumentation Injector, and the Compiler.

The primary purpose of the Source Parser is to convert the input C source code into an intermediate representation (the Abstract Syntax Tree) that can be easily understood by later portions of the tool. This intermediate representation includes useful information including the source-location of each function as well as the body of each *then* and *else* condition of an if statement.

Once the AST has been created, the Instrumentation Injector locates any blocks of code that need to be instrumented. Once located, a new, temporary version of the C source code is generated with instrumentation added in.

This temporary source code is passed into the final component - the compiler.

In order to be useful with ASICs, a couple of issues need to be mitigated. Primarily, the profiling implementation needs to

```
my_func:
  210a17c0          mov     %r9,%blink
  c0f1              push_s  %blink
  44cb 00000000r    mov_s   %r12,0    ; .mcount
  08020000r         bl      _mcount
  c0d1              pop_s   %blink
  7fe0              j_s.d   [%blink]
  d8a5              mov_s   %r0,165
```

Fig. 2: An example of a traditional instrumentation site as generated using the commercial compiler.

```
my_func:
  1e007043 00000000  stb    1,[0]  ;
      lwprof_my_func_c_37_39_1
  7fe0               j_s.d  [%blink]
  d8a5               mov_s  %r0,165
```

Fig. 3: An example of the LwProf coverage instrumentation site

be constructed without imposing a significant cost on the final product. That is, the code must fit in the same amount of ROM to ensure the die size does not increase. It must not use additional cycles, as this would require an increase in clock speed and power consumption to offset it. Hardware-based techniques mitigate the ROM and performance issues by adding an additional profiling unit into the ASIC, however this has the downside of increased die area. Additionally, these hardware methods are not portable across different pre-existing architectures as they require design changes and so cannot be readily used. Because of this, alternative methods needed to be developed to mitigate these issues.

In order to ensure a high level of portability across different systems, LwProf is designed as a software only solution. This also has the benefit that additional hardware features are not needed, reducing the likelihood of increased ASIC costs.

In Fig. 2, a traditional instrumentation site can be seen. The traditional instrumentation site results in three instructions being emitted with a total size of 14 bytes to store the instructions. Additionally, for smaller functions such as the one shown, the compiler needs to backup and restore the return address.

Similar to the dummy-branch optimization for hardware coverage, LwProf works around the execution overhead by replacing the middleware library call with a memory store instruction as seen in Fig. 3. This results in a single-instruction instrumentation site taking up just 8 bytes. REmoving the middleware call causes the execution overhead to become negligible. As a result, the instrumentation site for LwProf is lightweight in both size and execution overhead and has a high likelihood of being useful for

```
uint8_t my_func(void) {
    return 165;
}
```

Fig. 4: A standard C function.

```
unsigned char lwprof_my_func_c_37_39_1;
uint8_t my_func(void) {
    lwprof_my_func_c_37_39_1 = 1;
    return 165;
}
```

Fig. 5: A standard C function with the LwProf coverage instrumentation site.

ASIC development.

4. Implementation

The primary contribution of this paper is the implementation of the LwProf tool. As shown in Fig. 1, LwProf has three main components.

The most complex component, the Source Parse, is implemented by leveraging the LibTooling library (version 5.0.0) from the LLVM compiler infrastructure project [10]. By using LibTooling, LwProf gains the ability to parse C and C++ code, such as that shown in Fig. 4, at the same level as the LLVM-based clang compiler. Once the source files have been parsed, LibTooling generates an Abstract Syntax Tree (AST) that can be further processed by LwProf.

The second component, the Instrumentation Injector, uses the AST to determine instrumentation sites within the source code. Depending on the switches provided to LwProf, instrumentation can be injected at the beginning of each function, or within each *then* and *else* clause of an if statement. LwProf injects plain-old C code as shown in Fig. 5, ensuring a high level of portability across compilers. As can be seen, the emitted C code does not call into any middleware libraries, instead opting to set a variable to a constant value. Additionally, the data type of the variable can be configured, enabling smaller data-types to be used.

In addition to coverage information, a second mode to LwProf was implemented - the profiling mode. Example C code injected into a function can be see in Fig. 6. Instead of setting a memory location to 1, this mode increments the value stored in memory in order to record the number of times a function is called.

Lastly, the reader may have noticed that this implementation has the possibility of wrapping around once it reaches the maximum value for the specified data type. In order to mitigate this, a larger (such as 32bit) data types can be used resulting in higher memory overhead with no cycle overhead. If this is still undesirable, a simple overflow check can be added to ensure no wrapping occurs, at the expense of a single cycle and a single instruction in many CPU architectures.

```
unsigned char lwprof_my_func_c_37_39_1;
uint8_t my_func(void) {
    lwprof_my_func_c_37_39_1++;
    return 165;
}
```

Fig. 6: A standard C function with the LwProf profiling instrumentation site.

For the final component LwProf sends the instrumented source code to the original compiler. All LwProf specific command line arguments are stripped out, and the remaining arguments plus the modified source file are passed along. LwProf is thus able to call an arbitrary compiler, resulting in a transparent tool to the build system. This transparency allow for the user to seamlessly integrate LwProf into existing build systems, reducing the barrier for the developer to enable instrumentation within their project flow.

5. Evaluation

In order to determine the efficiency and effectiveness of LwProf, our tool is evaluated by comparing to a commercial tool and itself under different configurations based on the metrics and case study described here.

5.1 Metrics

Measuring the capabilities of each implementation was broken apart into three steps.

For memory-constrained systems, the increase in *code size* to enable coverage instrumentation can be prohibitively expensive. For each technique evaluated, the total increase in size, in addition to the per instrumentation size increase is evaluated. A number of tools can be used to determine the size. In this case, the objdump is used to dump the *.text* section sizes of a binary. Determining the overhead of an instrumentation tool is done by subtracted off the baseline size from the instrumented size as shown in Eq. 1.

$$size_{overhead} = size_{instr} - size_{baseline} \tag{1}$$

Similar to the code size, the *data size* impact is important for ASICs. The same Eq. 1 can be used to calculate the data size using the *.data* and *.bss* sections instead of *.text*.

Once the data size impact is calculated, the number of instrumentation sites can be determined. In the case of traditional coverage tools, each instrumentation site takes 4 bytes. When evaluating the LwProf results, the options passed to the tool must be considered to determine the per instrumentation data RAM overhead.

After the per instrumentation data size overhead is determined, the code overhead for each instrumentation sites can be determined by dividing the total code size by the number of sites as shown in Eq. 2.

$$SiteOverhead = \frac{size_{overhead}}{NumInstrumentations} \tag{2}$$

In addition to the per site overhead, the total binary size overhead was calculated as a percentage using Eq. 3.

$$PercentOverhead = \frac{size_{overhead}}{size_{baseline}} \tag{3}$$

On resource constrained system, the *execution time overhead* is also very important. If a system is already reaching its limits, then depending on the overhead, enabling instrumentation may cause the system to enter an overload condition. In this state events such as interrupts may be lost or mishandled, resulting in a misbehaving system. When this state is entered, the test cases being executed

may no longer reflect that of a non-instrumented binary. To measure the execution time overhead, the test system is placed into steady-state with baseline firmware with instrumentation disabled. Once the baseline load is measured, the firmware is re-built with instrumentation enabled and the load is re-measured. Next, the normalized load overhead was calculated using Eq. 4, allowing for a worse-case overhead to be determined.

$$NormOverhead = 1 - \frac{load_{instumented}}{load_{baseline}} \quad (4)$$

Finally, the overhead per instrumentation cycle is determined by dissembling the modified binary and counting cycles by hand.

The final metric that is evaluated is the *accuracy* of the coverage tool. For the purpose of this research, the original compiler implementation is used as the baseline.

$$Accuracy = 1 - \frac{abs(sites_{instumented} - sites_{baseline})}{sites_{baseline}} \quad (5)$$

Any deviation from the baseline (either more, or less) is considered an error as shown in Eq. 5.

5.2 Case Study

This research uses a proprietary firmware code base with a mature test suite as a case study. The code base was written for a customized CPU executing in a resource constrained system. To compile the firmware, commercial tool #1, a clang-based compiler is used. Additionally, support hardware exists for this code base that allows for non-intrusive measurements of the CPU load while executing the firmware. The various implementations in the next section are evaluated to ensure that the test results are not affected by the instrumentation.

The firmware, when running without instrumentation, can often show CPU loads of over 95% and ROM utilization of 99.6%. Due to the high CPU loads, the system is evaluated in a reduced performance mode where the average CPU load is close to 50% with instrumentation disabled. In order to accurately measure the code size impact, the available ROM for the case study was artificially increased to allow all libraries to properly link without having to shift code from ROM into RAM. The ROM increase allows for a direct comparison with all implementations.

6. Experimental Design

Five different instrumentation methods were evaluated for this experiment.

First, firmware with profiling disabled is evaluated. The baseline resulted in a binary with 131,440 bytes of code, of which 81,848 bytes of the binary were pre-compiled as a static library. As a result, the case study focused on the remaining re-compilable code of 49,592 bytes total as a baseline.

The second case evaluated is the coverage option in the commercial compiler that is enabled with the *-pg* compiler option. Once compiled, the official middleware library is linked into the firmware. The middleware library included with the compiler has many design choices that result in a configurable, but slow library. As an example, the library checks to determine if coverage is enabled, and only if so does it record that an instrumentation site was executed. While this seems reasonable, the number of cycles used to determine if coverage is enabled is larger than the number of cycles to actually record the result. In order to simplify the experiment, the middleware size overhead is not taken into account.

Table 1: Evaluation Results - Code Size

Implementation	Per Instr Code Overhead	Total Code Overhead
Baseline	-	0%
ct #1 -pg	14.87 bytes	19.26%
ct #1 -pg (optimized)	14.87 bytes	19.26%
`LwProf` (coverage)	7.51 bytes	10.68%
`LwProf` (profiling)	11.14 bytes	15.84%

The next step during evaluation was to optimize the commercial tool #1 middleware library to reduce the execution overhead. Unlike the default library, the optimized version simply stores a TRUE value to the passed in address and returns, recording the instrumentation with limited overhead. With the middleware library optimized, the firmware is re-evaluated to determine the overhead associated with the library itself.

Two variants of the `LwProf` tool were analyzed. The first variant, coverage-enable `LwProf`, was used as a best case scenario. The second variant enabled function-level profiling support in `LwProf`, allowing for the developer to determine how many times a function was called, instead of just if it was called.

7. Discussion of Results

In this section, we report the results obtained throughout the experimentation.

7.1 Code Size

The code size measurement results can be found in Table 1. For the purpose of this experiment, the size impact of the middleware library used for the commercial compiler was not included in these numbers. This allows for an easier comparison for just the instrumentation site overhead.

Both commercial tool #1 native and optimized implementations result in a code size increase of 14.87 bytes per instrumentation site. This number is the same for both version due the the middleware library, and not the instrumentation site, being optimized. This method enabled coverage and profiling metrics on a per function bases. Branch coverage is not supported. When enabled, three instructions (14 bytes total) are emitted to each compiled functions calling an instrumentation routine. When inspecting the assembly in Fig. 2, two additional instructions are seen due to the compiler needlessly issuing push_s and pop_s blink instruction, resulting higher overhead in when no function calls exist. As a result, the average instrumentation cost can be anywhere from 14 to 18 bytes, matching the values determined by measuring the code size. As most functions within the case study source code call other functions, the average overhead is closer to the smaller number.

Switching to `LwProf`, the instrumentation overhead can be reduced by a factor of two. When enabling function-level coverage, `LwProf` results in one instruction (8 bytes as shown in Fig. 3) being emitted with a total execution overhead of 3 cycles. When calculated over the full binary, however, a value of only 7.51 bytes was measured. This reduction is due to the commercial compiler optimizing the store instruction into a smaller form, depending on if additional memory accesses were done in the function being instrumented.

These size improvements directly translate to the total code overhead - the commercial tool #1 overhead of 19.26% was cut in half, resulting in an overhead of 10.68%.

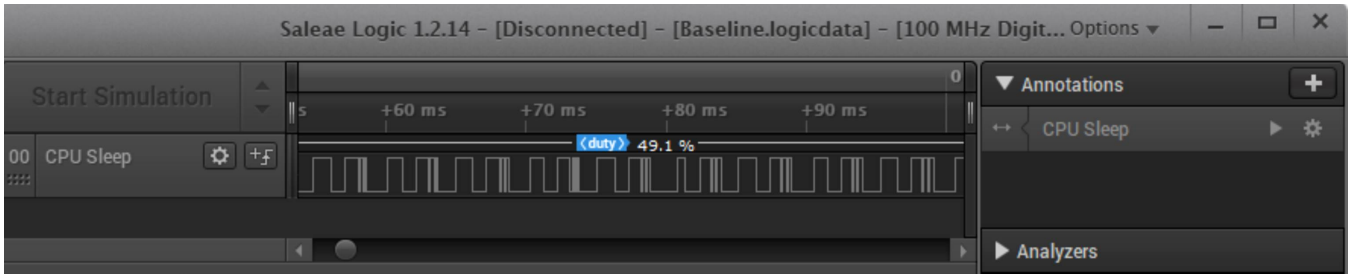

Fig. 7: Baseline CPU Load measurement using a logic analyzer

```
1 my_func:
2   42c3 00000000r   mov_s    %r2, 0       ;
        lwprof_my_func_c_37_39_1
3   8a20             ldb_s    %r1, [%r2]
4   d8a5             mov_s    %r0, 165
5   7124             add_s    %r1, %r1, 1
6   7fe0             j_s.d    [%blink]
7   aa20             stb_s    %r1, [%r2]
```

Fig. 8: An example of the LwProf profiling instrumentation site.

Table 2: Evaluation Results - Data Size

Implementation	Per Instr Data Overhead	Total Data Overhead
Baseline	N/A	N/A
ct #1 -pg	4 bytes	2568 bytes
ct #1 -pg (optimized)	4 bytes	2568 bytes
LwProf (coverage)	1 byte	705 bytes
LwProf (profiling)	1 byte	705 bytes

Lastly, if we look at the profiling version of LwProf, shown in Fig. 8, the code size increase goes up to 15.84%. This is due to two additional instruction being emitted for a total size of 12 bytes. As with the coverage version of LwProf, the actual measured value of 11.14 bytes is slightly lower due to the optimizations that the commercial compiler is able to make. This feature-enhanced version results in lower overhead than the official supported tool.

7.2 Data Size

The second metric measured is the data size metric. Results can be seen in Table 2. The commercial tool #1 instrumentation method allocates a 4-byte data type for each instrumentation site. As with the code size, the optimized middleware library is unable to improve on the data size for the optimized second case.

With LwProf, the size of the instrumentation counters can be configured, enabling an overhead of 1, 2, or 4 bytes per instrumentation site (705, 1410, or 2820 bytes total) depending on the developer needs. For the evaluation, a 1-byte data type size was chosen to record the coverage information, improving on the commercial implementation by a factor of four. When taking into account all instrumentation sites, close to 2KB of memory savings is realized. If the system under test is memory restricted, the memory savings can mean the difference between linkable firmware and a show stopper.

Table 3: Evaluation Results - Execution Time

Implementation	Per Instr Overhead	CPU Load	Normalized Overhead
Baseline	-	50.87%	0%
ct #1 -pg	48 Cycles	Error	-
ct #1 -pg (optimized)	10 Cycles	Error	-
LwProf (coverage)	3 Cycles	52.68%	3.44%
LwProf (profiling)	7 Cycles	55.14%	8.11%

7.3 Execution Time

The CPU load measurements for each of the test cases can be found in Table 3. Using a logic analyzer, in Fig. 7, the sleep output signal from the device was measured. When converted to CPU load, this means that the baseline firmware without coverage was active 50.87% of the time. This baseline measurement was achieved by enabling a subset of features in the firmware and placing the device into steady-state.

The commercial implementation results in 3 instructions emitted with an execution time of 5 cycles, not including the middleware library. When the middleware library is taken into account, the total execution time for each instrumentation is 48 cycles. When the commercial-instrumented code was loaded into the device and executed, the part resulted in a CPU exception. After analyzing the issue, it was determined that the -pg option adds incorrect instrumentation for interrupt routines, and as a result causes certain registers to be mangled that should not be. Because of this issue, no load information could be measured.

Similar to the native commercial implementation, the optimized version also has a 5 cycle execution hit due to the instructions being emitted. When the middleware library is replaced with an optimized version, however, the total execution time can be reduced down to 10 cycles. This is done by having the middleware library store the instrumentation result into memory and return. Similarly to the commercial native implementation, a CPU exception was encountered when this method was tested on hardware.

When running the LwProf version of the code the execution overhead is reduced to 3 cycles per instrumentation site. Once in steady-state, a 52.68% CPU load - a total overhead of 3.44% - was measured. As a result of the low overhead, coverage can be enabled even when the firmware is running close to its limits.

Similarly, when LwProf was switched to profiling mode, the execution overhead increases to 7 cycles per site resulting in a 55.14% CPU load (8.11% overhead) being measured.

When comparing the profiling mode to the coverage mode in LwProf, the higher 8.11% value matches the expected value of 8.03% based purely on the cycle amount for each instrumentation

Table 4: Evaluation Results - Accuracy

Implementation	Instrumentation Sites	Accuracy
ct #1 -pg	642	100%
ct #1 -pg (optimized)	642	100%
LwProf (coverage)	705	90.2%
LwProf (profiling)	705	90.2%

type.

7.4 Accuracy

The final item measured is the accuracy of the results. The existing test suite was modified to enable dumping of the instrumentation data after each test ran to completion. Each instrumentation site was then compared to determine if any sites were skipped. Unfortunately, the instrumentation injected by the commercial tool #1 -pg option mangled a register when instrumenting interrupt handlers. Since this implementation would not execute properly, no baseline data could be taken. As a result, the accuracy of each implementation could only be estimated. In Table 4, the commercial implementations are assumed to be 100% accurate. Since 642 sites were emitted within the commercial tool #1 version, and 705 sites were emitted with LwProf, the maximum accuracy that can be achieved is 90.2%.

In order to determine why there was a deviation in instrumentation sites between the commercial tool and LwProf, an analysis was done on the resulting assembly. During the analysis, a pattern was noticed with certain types of functions. If a function was marked as *static*, then there was a high likelihood for commercial tool #1 to skip adding instrumentation. In the event that the compiler completely inlines a function, then no instrumentation would be emitted for that function. As a result, the function no longer exists in the binary, and so no profiling or coverage information can be determined when using commercial tool #1. On the other hand, when LwProf injects instrumentation, the compiler to decide that a function should not be inline due to compiler thresholds. As a result, The LwProf instrumentation is emitted and information can be obtained about a function. Even if the LwProf code was inline, instrumentation would still be included in the final binary.

Similar to static functions, another class of functions was found to be ignored by the commercial tool. If a function is wrapped such that no prologue is emitted, then instrumentation can also be lost with commercial tool #1. In this case, the compiler with simply jump to the final function without adding instrumentation. Like the previous static case, this results in instrumentation being lost. When the same function is instrumented with LwProf, however, the compiler properly keeps the instrumentation and only issues a jump after the instrumentation site is executed.

Based on the two above cases, we can conclude the following: (1) LwProf ensures instrumentation is always added by doing so before the compiler can execute. (2) The commercial compiler implements instrumentation as a later stage in the optimization process. That is, commercial tool #1 performs optimizations with the pre-instrumented code. Because these optimizations are done pre-instrumentation, the compiler is able to remove small function calls.

While we initially assumed that the commercial compiler was 100% accurate, in reality it causes instrumentation information to be lost.

Finally, a third class of functions were found to cause an issue on LwProf. In some rare cases, a function can be completely implemented in a source file that is included like a standard header. When compiled, a functions without instrumentation is created in the resulting binary. In the analyzed function with this issue, it was also found that commercial tool #1 did not instrument it properly, however the reason for this was due to the previous inlining issues described above.

8. Threats to Validity

The primary threat to validity with this research is how each technique compares. In other words, how accurately do they measure coverage. This research initially assumed that the commercial compiler's -pg option is ideal and results in all functions being properly covered. The number of instrumentation sites injected by commercial tool #1 was 642, while the LwProf tool added 705, for an addition of 63 instrumentation sites. This discrepancy does lead to the possibility that the commercial tool's -pg option may not be as ideal as originally thought. Each instrumentation site emitted with commercial tool #1 and LwProf was analyzed, and multiple issues were found with the baseline commercial tool, while only one issue was found with the LwProf tool. As such, we can determine that while the two tools do note have the same results, that the LwProf results in better instrumentation.

Unfortunately, due to a bug in the commercial tool #1 implementation, we were unable to compare the CPU overhead between LwProf and commercial tool #1. As a result, only the hand-counted cycle numbers were used for comparison.

A second threat to validity is with regards to the case study. A proprietary code base was used for the case study, however the results may not translate to other code bases. While different code bases will result in different behaviors, the selected code base was originally developed for resource constrained ASICs. As such, it was designed to reduce the memory footprint and execution time, suggesting that it is a good analog to other code bases for ASIC firmware.

This research primary compares the LwProf tool to the instrumentation method implemented in the commercial tool #1. The results seen in this paper may not translate to other compilers such as GCC. That is, GCC may be much more optimal when enabling profiling. According to the gprof documentation for profiling with GCC, this implementation actually works the same way as the one in commercial tool #1. When the instrumentation option is enabled, both commercial tool #1 and GCC call into the _mcount routine with the calling address as an argument. As a result, it is safe to assume that the results in this paper will also significantly speed up profiling and coverage for GCC as is done with commercial tool #1.

9. Related Work

Various research has been done related to minimizing the overhead of measuring code coverage [1], [2], [5], [6], [7].

In THeME [1], a technique was developed for using the hardware profiling unit on modern processors to reduce branch coverage overhead. The technique showed that by adding a a single dummy branch instruction on each branch to account for fall-through

behavior yielded less than 2% code growth with minimal performance impact. LwProf builds on this research by replacing most compiler-based instrumentation with a lightweight version.

Techniques were developed by S. P. Kedia et al. [2] to split the instrumentation and analysis when measuring coverage into two separate phases. Thus, the file I/O phase only needs to be executed once after the test has completed, showing that both the memory impact and the execution time can be reduced. In LwProf, no analysis code is included in the final binary. Instead, hints are added to the symbol table for the binary, resulting in the ability to post-process data without impacting execution time of the software.

In Wu et al. [5], further research resulted in coverage testing with minimal memory impact. Their tool eXVantage limited the amount of information recorded so that only coverage information is stored. This information information is recorded in memory and later read out after the test completes instead of writing all coverage information to files. eXVantage also compacts the form in which coverage information is stored by ensuring that needless strings are not saved in memory. The technique shows that the performance overhead can be as low as 1%. LwProf uses a similar technique to reduce instrumentation overhead. By using the smallest supported data-type (8-bit char), LwProf can compress each coverage site to 1 byte with no execution time overhead. Similarly, no strings or file locations are directly recorded. Instead, this information is determined after the coverage information is dumped.

A method using self-modifying code was developed by J. Jenny Li et al. [6]. In their research, instrumentation code is removed from the execution path after it has been executed once. This results in compiled code with duplicate functions, one with coverage enabled and one without. The runtime environment modifies the executable in memory and replaces the slower coverage-enabled function with the faster coverage-disabled function after execution, resulting in the performance hit only happening once. This leads to a larger program binary with minimal execution time overhead. Due to the increased binary size, this technique is not usable for many resource-constrained embedded systems. Additionally, when placing code into and ASIC's ROM, self-modifying code becomes trickier since the ROM cannot be modified. Since the ROM is read-only, the instrumentation can only be removed if a trampoline is used within RAM to allow for the address to be modified at runtime. This trampoline technique has a similar overhead to LwProf, and as such the technique used in LwProf is more suited for ASICs.

10. Conclusion and Future Work

As shown, LwProf resulted in much lower execution and data size overhead than the officially supported instrumentation options. Likewise, the tool also supports both an extremely light weight coverage option, as well as a slightly slower profiling option. Both cases result in lower overhead than commercial tool #1.

Due to the simplicity, the LwProf instrumentation method also resulted in code that did not need to touch the CPUs register file ensuring that no internal state information was modified (excluding instrumentation ram). As a result, LwProf was able to run to completion without causing any exceptions. All other instrumentation methods modified register contents, which when analyzed, appeared to cause an issue with certain inline assembly blocks of code. LwProf is not only smaller and faster than commercial tool #1, but it has less possibilities for errors to be triggered within firmware under test.

The commercial compiler was assumed to be 100% accurate with regards to instrumentation. Once the two tools were compared, however, it can be seen that the accuracy of commercial tool #1 is reduced due to design decisions by the compiler vendor. On the other hand, we can see that the LwProf was unable to inject instrumentation into a certain class of functions. As such, we can conclude that in most cases, LwProf is actually more accurate than the baseline commercial tool. As a future work, LwProf could be enhanced to properly handle the class of functions that cannot be instrumented.

As an alternative to LwProf, the commercial tool #1 output could have been improved by writing a binary modification tool. This would involve locating all existing instrumentation sites in the code, and rewriting the assembly to perform a memory write instruction instead of a branch instruction. Due to the complexity and architecture-specific nature of the task, this is left for future exploration.

References

[1] K. Walcott-Justice, J. Mars, and M. L. Soffa, "Theme: A system for testing by hardware monitoring events," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012. New York, NY, USA: ACM, 2012, pp. 12–22. [Online]. Available: http://doi.acm.org/10.1145/2338965.2336755

[2] S. P. Kedia, A. Bhattacharjee, R. Kailash, and S. Dongre, "Coverage and profiling for real-time tiny kernels," in *2010 10th IEEE International Conference on Computer and Information Technology*, June 2010, pp. 1926–1931.

[3] J. Banker, A. Shanbhag, and N. Sherwani, "Physical design tradeoffs for asic technologies," in *Sixth Annual IEEE International ASIC Conference and Exhibit*, Sep 1993, pp. 70–78.

[4] B. J. Grenon and S. Hector, "Mask costs, a new look," in *22nd European Mask and Lithography Conference*, Jan 2006, pp. 1–5.

[5] X. Wu, J. J. Li, D. Weiss, and Y. Lee, "Coverage-based testing on embedded systems," in *Proceedings of the Second International Workshop on Automation of Software Test*, ser. AST '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 7–. [Online]. Available: http://dx.doi.org/10.1109/AST.2007.8

[6] J. J. Li, D. M. Weiss, and H. Yee, "An automatically-generated run-time instrumenter to reduce coverage testing overhead," in *Proceedings of the 3rd International Workshop on Automation of Software Test*, ser. AST '08. New York, NY, USA: ACM, 2008, pp. 49–56. [Online]. Available: http://doi.acm.org/10.1145/1370042.1370054

[7] R. Wu, X. Xiao, S.-C. Cheung, H. Zhang, and C. Zhang, "Casper: An efficient approach to call trace collection," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '16. New York, NY, USA: ACM, 2016, pp. 678–690. [Online]. Available: http://doi.acm.org/10.1145/2837614.2837619

[8] R. Santelices and M. J. Harrold, "Efficiently monitoring data-flow test coverage," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07. New York, NY, USA: ACM, 2007, pp. 343–352. [Online]. Available: http://doi.acm.org.libproxy.uccs.edu/10.1145/1321631.1321682

[9] llvm-admin team. (2017, September) llvm-cov - emit coverage information. [Online]. Available: https://llvm.org/docs/CommandGuide/llvm-cov.html

[10] L. Project. (2017, September) The llvm compiler infrastructure. [Online]. Available: http://llvm.org/