

Static Properties of Commercial Embedded Real-Time Programs, and Their Implication for Worst-Case Execution Time Analysis

Jakob Engblom

Department of Computer Systems, Uppsala University
jakob@docs.uu.se

Abstract

We have used a modified C compiler to analyze a large number of commercial real-time and embedded applications written in C for 8- and 16-bit processors. Only static aspects of the programs have been studied, i.e. such information that can be obtained from the source code without running the programs.

The purpose of the study is to provide guidance for the development of worst-case execution time (WCET) analysis tools, and to increase the knowledge about embedded programs in general. Knowing how real programs are written makes it easier to focus research in relevant areas and set priorities.

The conclusion is that real-time and embedded programs are not necessarily simple just because they are written for small machines. This indicates that real-life WCET analysis tools need to handle advanced programming constructions, including function pointer calls and recursion.

1. Introduction

This paper describes the results of a study undertaken by the hard real-time systems group at Uppsala University. It is a part of our work on worst-case execution time (WCET) analysis for real-time programs. The study is called *MARE (Measurements of Actual Real-time and Embedded programs)*, and aims to collect quantitative measurements on how commercial software used in real-time and embedded systems is written.

Knowledge about how programs are written is useful for several purposes, including:

- **Guiding the selection of relevant research topics:** Our goal is to produce tools that can be used in industrial practice. This requires knowledge about how the engineers in industry actually write code. There is little to be gained from making simplifying assumptions not supported by reality. Also, frequent problems should be solved before less frequent problems.
- **Testing and benchmarking:** comparisons between different analysis methods should be performed using realistic benchmarks and examples – i.e. code which resembles real embedded programs.
- **Confirming and dismissing myths:** when discussing WCET analysis and its relation to real programs, researchers and people from industry tend to use

unverified assertions and assumptions about how "real programs" are written. We hope to provide at least some empiric data on how embedded programs are written.

The main motivation for this study was the need to guide our research into WCET analysis methods (a secondary motivation was simple curiosity). This paper presents our conclusions for WCET tools, based on the results of the study.

The study (and our research) focuses on small embedded systems: the eight- and sixteen-bit processors used in a majority of the world's embedded systems. The programming language used is C, since this is the dominant programming language in the embedded systems field [1].

We have not found any previous study dealing with the details of how code is written for real commercial programs targeted for small embedded systems.

The model for WCET analysis presented in [2] forms the conceptual basis for the discussion in this paper. This model divides WCET analysis into two types of analysis: *low-level analysis*, which deals with the amount of time taken to execute a straight-line segment of machine code, and *high-level analysis*, which deals with how to sequence the low-level segments (loop bounds, infeasible paths, etc.). It is mainly the high-level analysis that is affected by the characteristics of the source code.

This paper is structured as follows: Section 2 discusses measurement methodology. Section 3 describes the studied programs. Section 4 contains the measurement result. Section 5 discusses the implications of the measurements for WCET analysis and WCET research. Finally, Section 6 provides a summary, our final conclusions, and plans for future work.

A full description of the MARE study and the results can be found in [3].

2. Methodology

The MARE study was performed using a modified commercial C/C++-compiler (from IAR Systems AB [4]). This provided us with a C-parser suitable for handling embedded programs (extended keywords etc.) and an optimizer.

Embedded compilers usually implement various extensions to the standard C language, which poses a problem for a research compiler. The compiler has to mimic the C variants implemented by several different compilers. Each

program is in effect written for a certain compiler. For an example of typical features present in an embedded compiler, see [5].

The IAR C compiler was easy to modify to allow it to mimic the embedded compilers used for the studied programs. The compiler settings changed were:

- The size of the types `int`, `double`, and the various pointer types (code pointers, data pointers, pointers to different memory areas).
- Adding keywords for modifying variable placement in memory (`near`, `far`, `huge`, `banked`,...). function calling conventions (`interrupt`, `trap`, `monitor`,...), and similar features used by embedded programmers.
- The list of *intrinsic functions* used in the compiler (see Section 4.5 for more on intrinsic functions).

We collected our measurements at the *intermediate language* level, after parsing and source-level optimizations. This is motivated by the following:

- Our research plans are to insert the high-level analysis component of a WCET tool into a compiler. This tool will operate on the intermediate code.
- The optimizer is one of the problems a WCET tool has to deal with [2]. Thus, measurements *after* optimization are relevant.
- The analysis of intermediate code allows us to get to the *essential* properties of a program, not the *accidental* properties of its syntactical representation. We are not interested in indentation, variable naming, `while`-loops vs. `for`-loops, and similar issues, since this does not affect automatic analysis techniques.

Furthermore, some factors indicate that source code analysis is not always relevant:

- Many programs used in embedded systems are machine-generated, which (usually) makes them ugly and hard-to-read. The structure is often such that a human would (should) not write (`gotos`, very large functions, using `switch` for all decisions, and some other simple-to-generate but hard-to-read methods). Making statements about such code based on the structure of the source code is not very sensible.
- When we get programs to analyze from industry, they are often *obfuscated* – all function names and variable names are changed to meaningless combinations of digits and letters. More advanced obfuscators can even rewrite the code structurally (for example, rewriting loop constructions), which means that studies based on the source code style and syntax are irrelevant.

Finally, performing the analysis after parsing and optimization makes the tool simpler, since we do not have to handle errors and the parsing and optimization process simplifies the code. The intermediate language is much simpler than the C language. However, the intermediate code is still rather far away from assembly-language. The same intermediate language is used for all processors.

The data collected is relevant for indicating the *presence* of certain features in real programs – but less so for indicating the *absence*. Furthermore, comparisons on the frequency of presence of certain features are relevant (statements like “feature X is twice as common as feature Y”).

3. Studied Programs

The application areas from which the programs originated were telecommunications, vehicular control, and home and consumer electronics. Unfortunately, due to non-disclosure agreements, we can only present the results and not the source code, and we cannot name the companies involved.

The programs contain a mix of timing critical code (including control loops, protocol management, coding and decoding) and non-timing critical code (often user-interface).

We measured the *entire* programs, since we believe that WCET analysis has to deal with entire programs – even if just a small part of the application is timing critical. All of a program may influence the WCET of a small part of it, and a programmer cannot be expected to identify which parts of a program are relevant for a certain timing-critical section. Furthermore, timing analysis is a valuable development tool even for non-timing critical code. Note that this makes the study of general interest for embedded compiler writers, since we provide a characterization of the kinds of programs they have to compile.

The size of the body of code used in the study is the following:

- 334 600 non-comment lines of C source code.
- 477 source files, using 942 header files.
- 13 different application programs, from six different companies.
- 5 579 functions.
- 17 173 variables (not including hardware registers and operating system variables)

The following processors were used in the systems: the 8-bit Hitachi H8, Zilog Z-80, and Motorola 68HC11, and the 16-bit Siemens C166, Mitsubishi MELPS 7700, and Hitachi H8 (there are both 8-bit and 16-bit processors in the H8 family).

4. Results

In this section, we present the measurement results that we consider relevant to WCET analysis. Each subsection presents some results together with a short discussion on their implications for high-level WCET analysis.

The selection of measurements was guided by the needs of high-level analysis, and this set of results is not necessarily relevant for other purposes.

The set of possible measurements were also limited by the technical abilities of our research compiler. We have

no way to analyze complete programs: the compiler operates on a per-file basis. This precludes many interesting measurements, like how global data is used and the pattern of data flow and control flow.

4.1. Recursive Calls

Recursive calls are theoretically equivalent to imperative loops, but is much more difficult to analyze in a language not designed for recursion (such as C). The study found 14 recursive call loops involving 18 functions in the 5579 functions we analyzed.

Four of the loops were in user interface code, which is unlikely to be timing critical (i.e. requiring WCET analysis).

However, ten of the loops were found in machine-generated code used for managing communications protocols. This is more problematic, since such code is both hard to change and quite likely to be timing critical.

The presence of recursion was a little surprising, since recursion is typically considered a bad thing for embedded systems – especially for small processors [6].

It should be noted, however, that among embedded processors, the ones we studied are *not* among the weakest (they all have usable stacks, for example). There are processors where recursion is not possible to implement, since they do not have a stack (like the Microchip PIC family).

Our conclusion: Recursion is probably not too common, but it is not negligible. We think it is reasonable to ignore recursion in early WCET tools.

4.2. Unstructured Flow Graphs

Unstructured flowgraphs (also known as irreducible flowgraphs) cause problems for loop analysis, since they complicate the definition of the loop concept [7]. An unstructured flow graph is typically constructed by jumping into a loop body from outside the loop.

In the study, we found 18 unstructured flow graph fragments. Most of them were found in machine-generated code using `gotos` to implement state machines. A few were well-structured human-written functions that had been destructured by compiler optimizations¹.

Conclusion: Unstructured flow graphs cannot be ignored, and must be handled in high-level analysis. Changing code generators and optimizers not to generate unstructured code is not a realistic alternative.

¹ An example of a compiler optimization causing a well-structured flow graphs to become unstructured is loop unrolling. In order to handle a number of iterations not evenly dividing into the unrolling factor, the compiler might insert a jump into the middle of the loop.

In our experience, many jump optimizations cause unstructured flow graphs, typically because they are not designed to preserve the structure of the code, only to make it smaller and/or faster.

4.3. Loops

Loops are one of the principal difficulties in WCET analysis (the other two are conditional statements and function calls [2]). In this study, a loop is defined by a jump to a previously executed statement in the program flow graph (a *back edge* in compiler theory). Loops in unstructured flow graphs have not been considered.

Several measurements were made in order to determine how complicated the loops are in the studied programs. The measurements were performed per loop nest; there were 1414 loop nests in the analyzed programs.

Note that there are no measurements on the specific difficulty of determining loop bounds for the loop nests. This is because the difficulty of loop bounds analysis is related to a specific analysis method – it is not obvious how to assign an absolute value to the difficulty of a loop.

Depth of loop nests: the depth of a loop nest can affect both the runtime and precision of loop analysis. The loop nests found in the study were quite shallow: no nest was deeper than four, and more than 90 % of all loops had a depth of one (single-nested loop).

Note that the loop nesting level of *programs* can be (and is expected to be) much greater: if a function calls another function inside a loop, and the called function contains a loop, this creates a deeper loop nest than that of the individual functions.

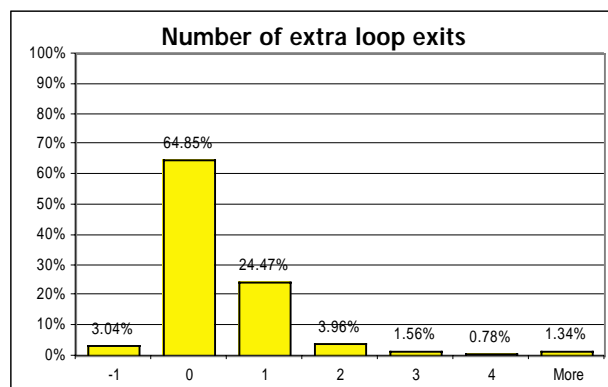


Figure 1

Number of loop exits: a simple loop has the same number of exits as the loop nesting depth: i.e. each constituent loop has exactly one exit. However, loops can have *several* exits (in C this is usually written using `break` statements) or *fewer* exits, in which case the loop never terminates. Figure 1 shows the distribution of extra loop exits. Zero extra exits means that the loop nest has no extra exits. Minus one² means that the loop is non-terminating and positive values that there are extra exits.

² Note that a loop nest can only have one exit to little: everything after the end of the innermost non-terminating loop is dead code – it is impossible to construct a loop nest with non-termination on several nesting levels.

Loop body complexity. Since WCET analysis is likely to concentrate on the loops in the program (intuitively, this is where most of the time is spent), it is interesting to investigate the complexity of the loop bodies. The complexity is estimated by counting the number of basic blocks in the innermost loop body of each loop nest (since this is where most of the time is likely to be spent).

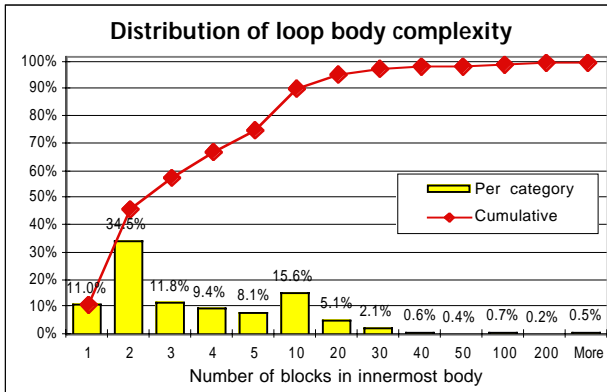


Figure 2

Figure 2 shows the distribution of loop body complexity, measured as the number of basic blocks in the innermost loop. Most bodies are quite simple (90 % have less than 10 blocks in the innermost loop), but there are some very complex loop bodies (with more than 200 blocks).

The loops with one block in the body are simple `do-while` loops where the loop condition is tested *after* each iteration. Loops with two blocks are typical for loops, where the loop condition is tested *before* each iteration (this adds a block to the loop body). Loops with more than two blocks contain at least one conditional statement in the loop body.

Conclusions about loops.

- Loop analysis local to a function needs to handle moderately nested loops, and loop analysis across functions need to handle very deep loop nests.
- The run time of loop analysis should not depend on the potential number of loop iterations but rather on the depth of the nested loops.
- Loops with no exits and loops with multiple exits are to be expected to occur with non-negligible frequency, and must be handled.
- Loop bodies can be both very simple and very complex. One solution is to adapt the analysis method after the complexity of the loop nest (using some heuristic to decide which loops are too complex to handle with high-precision methods. One possible heuristic is the number of basic blocks in the loop body).

4.4. Non-Terminating Functions

Non-terminating functions, functions that never return, are not very common in desktop and general purpose computing. In embedded systems, however, such functions are common, since they are often used to implement the bodies of tasks. The tasks run as long as power is applied to the system, and coding-wise they are non-terminating loops.

In total, 44 functions (of the 5579 in the study) were non-terminating. Ten of the thirteen applications we studied included non-terminating functions. Note that there could be other functions that never terminate, but where the optimizer³ cannot prove non-termination; this study only counts *provably* non-terminating functions. For a tool, this case can be handled reliably only by putting a bound on the analysis time, since proving non-termination in general is undecidable. An example of a provable non-terminating function is shown in Figure 3.

```

void foo(void)
{
    /* Perform initializations */
    init...

    /* Main non-terminating loop */
    while(1)
    {
        work...
        wait(period);
    }
}

```

Example of a non-terminating function. The `wait()` call calls the operating system. It returns to the function the next time the task is activated.

Figure 3

There is a relation between non-terminating functions and non-terminating loops. A function that never terminates must contain a loop (or unstructured flow graph fragment) that never terminates. However, the presence of a non-terminating loop does not make a function non-terminating, since it is possible that the function only enters the loop under certain circumstances.

Useful analysis of non-terminating functions requires heuristics to be applied to decide on a (terminating) part of the function to time (the WCET for the function as a whole is obvious: the uptime of the system). It is very likely that the time needed to execute the body of a non-terminating loop once is a measure relevant to the user.

Conclusion: Non-terminating functions are important in embedded systems, and should be handled gracefully. They cannot be ignored or defined as “pathological cases”.

³ We use the program analysis performed by the optimizer of the IAR C compiler to detect non-terminating loops.

4.5. Multiple Entry Points

Embedded programs can have *several* entry points, as opposed to the single `main()` function in ordinary desktop programs. This means that the call-graph for the program has several roots, one for each entry point to the program. Entry points can take the following shape:

- The `main()` function, called when the program starts.
- Functions used to define tasks are usually called from the real-time operating system (RTOS).
- Interrupt handling functions are invoked by the hardware or RTOS. Setting up interrupt handling and allowing functions to be declared as interrupt functions is a service offered by many embedded systems compilers.
- The operating system (if used) may employ callbacks to implement reactions to events.

Identifying these entry points automatically or with programmer assistance is necessary for a WCET analysis tool. The system designer can be relied on to know which functions are used as task bodies, callback handlers, or interrupt handlers.

An embedded compiler allows programmers to use special keywords to declare functions as *interrupt* or *trap* functions. Such functions are invoked when a hardware event happens, and are usually not called from other functions in the program. They can be identified by a WCET tool integrated into the compiler. For task bodies, however, there are usually no special keywords.

Conclusions:

- A WCET tool must be prepared to handle a program with multiple entry points.
- The user must be able to specify the functions used as entry points.
- The user must be able to specify what to time in the program: which function, called from which task? This is needed in order to establish the context for the WCET analysis.

4.6. Function Calls

Function calls are one of the factors that contribute to making WCET analysis more complex, since we need the WCET for called functions to determine the WCET for the calling function. In addition, the WCET of a function call may vary with the values of parameters and the context of the call.

The type of a function call influences the complexity of determining the WCET for the called function. We have examined the types of function calls, distinguishing between the following categories of calls:

- *External calls*: calls to functions in the same program, but outside the current source file. Handling such calls requires that the whole program is available for WCET analysis. The concept of external calls is a consequence of the use of *separate compilation* in C.

Every file is compiled separately, and not joined with other parts of the program until linking.

- *Function pointer calls*: calls to functions pointers cause problems for program analysis. If the set of functions a certain pointer can point to cannot be determined, the analysis has to assume that *any function* in the program (or even outside it) can be called.
- *Same file calls*: calls to functions in the same file. Easy to analyze since we have all information available at compile-time.
- *Intrinsic function calls*: intrinsic functions are not actually functions – they are a way to access low-level features of a processor that cannot be expressed in C (like interrupt handling). An intrinsic function call is replaced by a few inline assembly language statements. They are very simple to handle in WCET analysis since they just add a few instructions to the object code, without changing the flow.
- *Standard library calls*: calls to the C standard library (`printf()`, `strcat()`, etc.). The functions in the standard library are under the control of the compiler vendor, and should be analyzed when the libraries are *built*, not every time they are *used*.

Figure 4 shows the distribution of the types of function calls in the examined programs. The columns 1 to 13 represent the individual programs, and the *Total* column the overall result.

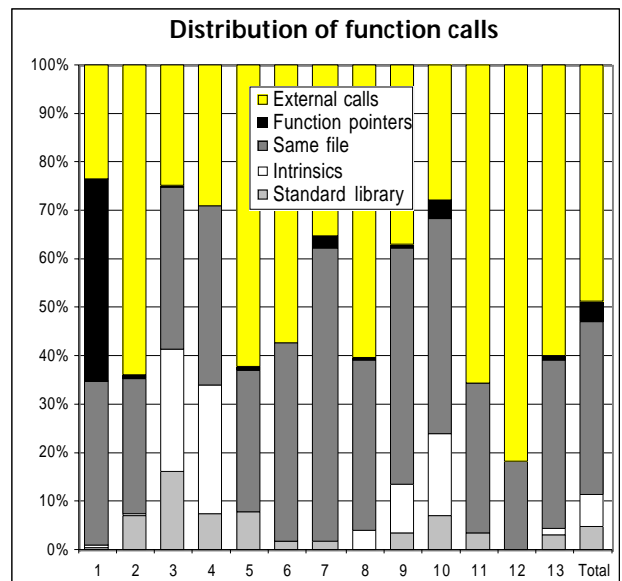


Figure 4

The large number of function pointer calls in program 1 is the result of a coding style that employs large jump tables (whose contents are constant and known). Nine of the 13 programs use function pointers.

Eleven programs use intrinsic functions, which indicate that this is a useful feature in embedded compilers. All programs use the standard library.

Conclusions:

- Function pointers cannot be ignored. Probably global pointer analysis is needed in order to determine the targets of pointers.
- The ubiquity of external calls indicates the necessity of performing analysis across entire programs: we cannot expect to perform WCET analysis within the confines of a single source file.
- Intrinsic functions and the standard library are useful tools for embedded programmers.

4.7. Function Complexity

Many functions in embedded programs are quite simple: reading some value from an input port, writing some actuator register, performing some common calculations – without control flow change. Such simple functions could allow simpler analysis techniques to be applied, or they may make a complex analysis run faster. To investigate the potential for optimization by taking advantage of simple functions, the control complexity of functions was measured.

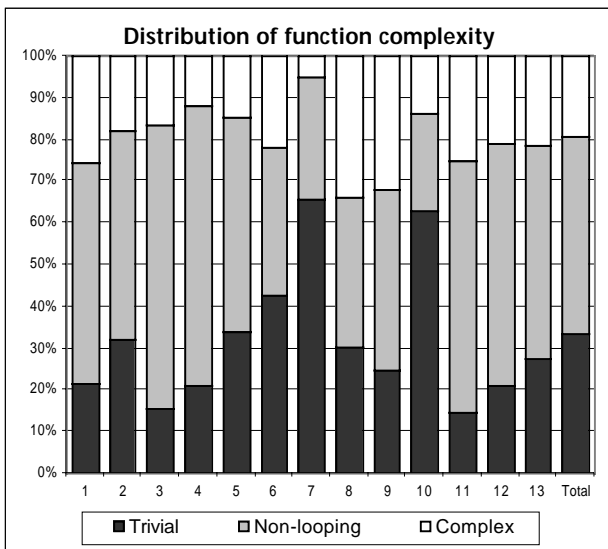


Figure 5

The functions in the study were classified into one of the following three categories:

- *Trivial*: functions without control flow – simple flow-through functions, containing just one basic block. No decisions, multiple exits, or loops.
- *Non-looping*: functions without loops, but with conditional control flow (*if*-statements and *switches*).
- *Complex*: functions containing loops or unstructured flow graph fragments. Note that a function containing

a loop usually contains a conditional flow – the loop condition (provided that the loop terminates). As stated above, there were only 44 non-terminating functions in the study, and they can be included in the *complex* category without distorting the data.

Figure 5 shows the distribution of function complexity for the individual programs (1 to 13) and the total for all programs.

For all programs studied, about one third of the functions are trivial (this varies between 15 % and 65 % for the individual programs). Almost half of the functions are non-looping, and only about one fifth contain loops.

For trivial functions, WCET analysis is quite simple: sum the execution times for the instructions. For certain systems, there are complications: instructions may take a variable amount of time, and if caches and pipelines are being used, the execution time can depend on the calling context. However, the analysis is still simpler than for more complex functions.

For non-looping functions, more efficient analysis methods than those needed to handle looping functions could be applied.

Conclusions: The fact that most functions are non-looping or trivial should be exploited for optimizing WCET analysis (and other program analysis).

4.8. Decision Nest Complexity

Conditional statements are an important factor in program analysis. In this study, the complexity of conditional statements have been measured by considering the depth of *decision nests*.

A decision nest is a nest of *if* and *switch* statements, where the outermost statement is on the top level of a function; the decision nest is not buried inside any other conditional statement, and is always executed if the function is executed. There were 5604 decision nests in the studied programs.

For a simple *if* or *switch* statement, the depth is one (regardless of the number of cases in the *switch*). For deeper nests, the depth is equal to the number of decisions on the path through the decision nest that makes the *maximum number of decisions*. The decision nest can be considered as a tree, and we look for the longest path in this tree. This property is illustrated in Figure 6.

<pre>foo() { if(C1) { if(C2) ... if(C3) ... } }</pre>	<pre>foo() { if(C1) ... if(C2) ... if(C3) ... }</pre>
One decision nest, with depth 3: we can take the C1-C2-C3 sequence of decisions.	Three decisions nests, each with depth 1.

Figure 6

This measurement agrees with the viewpoint of program analysis. The main impact of decision nesting depth on WCET analysis is in the determination of infeasible paths (paths that cannot be taken). The more complex the decision structure of the program, the more complex and time-consuming the analysis will be.

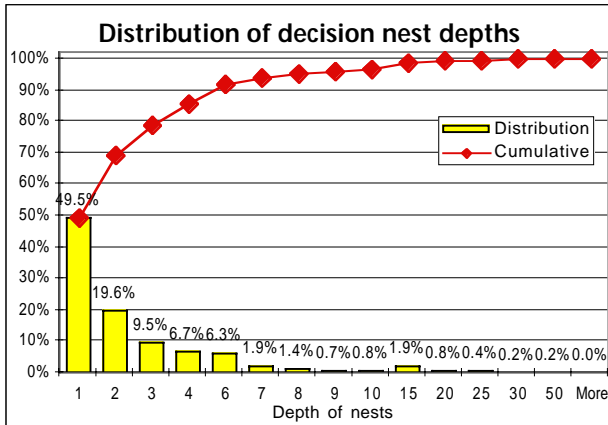


Figure 7

Figure 7 shows the distribution of decision nest depths. Note that about 50 % of the decision nests have a depth of one, i.e. they are simple `if` or `switch` statements, and that very complex nests are quite rare (as could be expected).

Conclusions:

- It is not safe to assume that embedded and real-time programs only have simple control flows.
- The run time of path analysis should not be exponential in the depth of the decision nests encountered, since this will make the analysis unusable for many real-life programs (for example, a naïve path enumeration would be exponential).

4.9. Variable Types

The “chunking size” of the data processed by a program is related to the type of system being programmed and the capacity of the processor. We have studied the types and sizes of variables in the programs in order to get a profile of how data is represented and processed in embedded programs.

17 173 variables were studied. Note that we only studied *program* variables. Variables that are just declared in header files and never defined in source files are not counted. This removes hardware I/O “variables” and operating system globals from the study.

The first measurement is the frequency of use of various *categories* of variables:

- Integer variables: `int`, `char`, `short`, `long`. These comprised 55.97 % of all variables.
- Float variables: `float`, `double`, `long double`. Hardly used at all: 0.05 % of the variables were floating point.

- Structures and unions. Frequency: 9.88 %.
- Arrays. Frequency: 11.8 %
- Pointers: pointers to data items. These were quite common, comprising 22.08 % of the variables.
- Code pointers: pointers to functions. Only 0.23 % of all variables.

To investigate how the capacity of the processor affects the type of variables used, we measured the sizes of integer variables used in the programs for each type of processor. The reason for studying the integer variables is that they are typically used for performing calculations, and thus should give a good idea of the size of data processed.

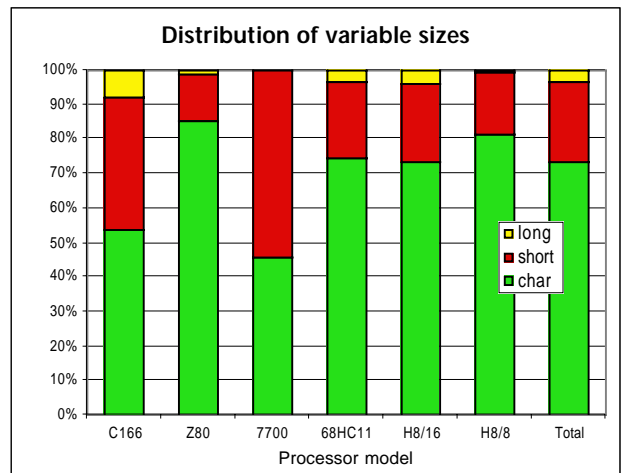


Figure 8

The result is presented in Figure 8 (the column labeled *H8/16* represents the sixteen-bit Hitachi H8 processors, and *H8/8* the eight-bit Hitachis). *Char* means an 8-bit value, *Short* a 16-bit value, and *Long* a 32-bit value. The plain C `int` type was mapped to either *short* or *long*, depending on the processor model (according to the size used in the original compiler used to write the program). The Z-80 and 68HC11 are 8-bit processors, and the C166 and 7700 are 16-bit processors.

The dominance of small data is clear: about 75 % of all variables used were single bytes (`char`), and less than 5 % were 32-bit words.

Conclusions:

- Data in embedded programs written for small processors is typically processed in small (8- and 16-bit) chunks. This is not surprising: for performance, it is clearly advantageous to manipulate data in the natural size for the processor, which is eight or sixteen bits. Furthermore, most I/O hardware communicate using byte-sized values, which makes it natural to use `char` variables to calculate outputs and read inputs.
- Data pointers are common, which means that pointer analysis is necessary to obtain good analysis results.

4.10. Summary and Demands

From the measurements presented in this section, we draw some conclusions regarding the requirements to be placed on WCET tools. A complete tool for WCET analysis must handle the following “difficult” program features:

- Recursion.
- Unstructured flow graphs.
- Function pointers and function pointer calls.
- Data pointers.
- Deeply nested loops.
- Multiple loop exits.
- Deeply nested decision nests.
- Non-terminating loops and functions.

Our classification of certain features as being “difficult” does not have a formal basis. It is based on “conventional wisdom” among teachers and researchers in the compiler field, and on observations of what is considered to be advanced compiler design.

In the next section, we compare this set of requirements to the present state of the research in WCET analysis.

5. Comparison with State of Research

Section 4 contains a large number of requirements and opportunities for WCET tools. In this section, we compare these with the present state-of-the-art in WCET research. Note that this is no attempt to list all contributions of all groups working in the WCET field. Rather, we give examples of approaches to each problem.

- **Recursion:** Recursion is in general forbidden, since it makes the program call-graph potentially unbounded. However, the approach for cache analysis and timing analysis described in [8] handles recursion and loops in a unified and elegant way. Examples of approaches forbidding recursion are [9] and [10].
- **Unstructured flow graphs:** In theory, ILP- or constraint-solution-based WCET techniques handle unstructured flow graphs [8], [9]. In practice, however, unstructured flow graphs are forbidden for other practical reasons [8]. Tree-based approaches cannot handle unstructured flow graphs, since they depend on well-formed loops [11].
- **Loops, analysis complexity:** The problem is to reduce the amount of information about a loop to a minimum. Instruction cache analysis models differentiating between the first and successive iterations of a loop offer a good precision–complexity trade-off for some cases [8], [10]. Better precision is achieved through the “calculated” data cache references in [10] (this approach condenses a list of hits/misses to a single hit/miss counter). Regarding loop bounds analysis, the automatic analysis presented in [12] analyzes each *potential* iteration of the loop, giving exponential complexity. The analysis in [13] typically

performs better (but it still has a problem for complex loop bodies).

- **Loops, multiple exits:** Analyzed for bounds in [13]. Timing effect handled in [9], [10].
- **Non-looping functions:** [14] describes an approach to WCET estimation tailored to non-looping code.
- **Decision nest complexity:** The analysis in [14] uses a branch-and-bound-like algorithm to overcome the problem, while [15] exhibits exponential behavior.
- **Whole program analysis:** All WCET analysis approaches dealing with caches have to assume complete knowledge of the program [8], [10]. Library functions are simply ignored in [8] – this method is applicable to all cases of unknown code, but depends on assumptions about the behavior of the unknown code.
- **Function pointer calls:** In general, WCET analysis approaches assume all function calls are known. [10] explicitly forbids function pointers.
- **Data pointers:** Unknown data pointers destroy data cache analysis, since they can result in writes to arbitrary memory. A common assumption is that all data accesses are to known locations [16]. In [17], a safe overestimation for the timing effect of cache accesses for unknown memory references is used.
- **Non-terminating loops and functions, and multiple entry points:** Not specifically mentioned in the literature, but can be considered as being a problem of defining what to time, and the assumptions about the system context for a program.

In conclusion, present WCET research is addressing some of the more difficult problems we have found in real code, but there is certainly a need for more research.

One issue that we find especially important is to determine *safe over-estimations* for rare and complex features. The approach should be to analyze most of a program exactly, and resort to approximations (instead of giving up) when very complex code is encountered. Just like in scheduling research, the direction forward must be to successively relax the limitations on the characteristics of analyzable problems (programs).

6. Conclusions and Future Work

The main conclusion from this study is that although embedded real-time programs are used to control quite small and often simple devices, and the programs themselves are quite small (at least compared to the desktop programs of today), the programs are in no way trivial. This is bad news for WCET analysts, since it makes it harder to create useful tools.

However, there are some encouraging tendencies: most measurements indicate that the largest part of a program is simple (typically between 75 % and 90 %). This should be exploited for analysis optimization (and by compilers).

Probably, applying different methods to different parts of a program is a viable strategy. The difficult parts may

require an approximate analysis to be tractable, while the simple parts may be analyzed in depth.

We have given an overview of how current WCET research handles the complexities. In many cases, there are useable methods, for other cases there are reasonable safe approximations, but there are some cases where more research is needed. In many cases, the issues are common to WCET analysis and compiler technology.

6.1. Implications for Systems Architecture

Apart from the requirements for specific analysis techniques and analyses for specific program features, there are some more general conclusions that can be drawn from this study. There are several facts that have an impact on how a useful WCET tool should be designed:

- The use of automatically generated code makes it untenable to rely on programmer annotations and feedback to determine loop bounds etc.
- The use of libraries and operating systems delivered in object-code form makes it necessary to consider analysis of systems without access to the complete source code. Some information must be supplied about the libraries and operating systems to enable the analysis. How this should be expressed is one area of research. Furthermore, some approximations must be used where no information is available.
- The use of function pointers, data pointers, and calls to functions external to a source file requires that the WCET analysis operates over entire programs. Global data-flow and points-to analyses are necessary support tools for WCET analysis.
- The presence of multiple entry points in a program must be handled, at least on the user-interface level.
- The variation in complexity between functions mandates the use of several different methods on the same program, preferably by automatic adaption. An adaptive framework would also allow the integration in one tool of the research presently being performed on analyzing certain classes of loops, algorithms, and programs.
- Most of the code we studied seemed to be written by disciplined programmers working using some software engineering process. This makes it tenable to design a set of *programming guidelines* for timing-predictable code. In many cases, embedded programmers already try to write code with predictable timing behavior [6], and a set of guidelines should stand a good chance of being used.

6.2. Future Work

This study has been a part of the planning process for our research into WCET analysis with a focus on tool-building, and our main future work is to continue our work on a WCET tool.

Regarding program measurements, we plan to use our research compiler to measure other classes of programs,

and to make comparisons between the classes. The first step is to study the well-known SPEC benchmarks [18], and investigate whether there exist substantial differences between this set of desktop computing programs and embedded programs. This project is in progress⁴.

It will also be interesting to investigate the new embedded benchmarks from the EEMBC (EDN Embedded Microprocessor Benchmarks Consortium) [19].

We would also like to cooperate with other WCET groups to generate a set of representative programs to use when comparing the efficiency and capabilities of WCET tools (in the field of verification of real-time systems using timed automata, such standard benchmarks have been very helpful in driving the development of ever more efficient tools).

Acknowledgements

This work has been supported by the competence center ASTEC (Advanced Software TEChnology) [20] and IAR Systems AB [4].

References

- [1] V. Seppänen, A-M. Kähkönen, M. Oivo, H. Perunka, P. Isomursu, and P. Pulli: *Strategic Needs and Future Trends of Embedded Software*, TEKES Technology Review 48/96, 1996. To order, check <http://www.tekes.fi>.
- [2] J. Engblom: *Worst-Case Execution Time Analysis for Optimized Code*, MSc Thesis DoCS 97/94, Uppsala University, September 1997. Available on the web: <http://www.docs.uu.se/~jakob>.
- [3] J. Engblom: *Static Properties of Commercial Real-Time and Embedded Systems – Results from the MARE Project*, ASTEC Technical Report 98/05, Uppsala University, October 1998. Available on the web: <http://www.docs.uu.se/astec>.
- [4] Homepage for IAR Systems AB: <http://www.iar.com>.
- [5] Information about the IAR 68HC11 compiler: <http://www.iar.com/download/ew6811.pdf>.
- [6] N. Jones: “Efficient C code for eight-bit MCUs”. *Embedded Systems Programming Europe*. Miller Freeman Ltd, London, February 1999, pp. 18–30.
- [7] S. S. Muchnick: *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers, San Francisco, 1997.

⁴ Due to the different reviewing time frames of workshops and conferences, the “future work” of investigating SpecInt95 was presented one month before the publication of this paper, at the ACM SIGPLAN LCTES Workshop in Atlanta in May 1999.

- [8] H. Theiling: *Über die Verwendung von ganzzahliger linearer Programmierung zum Suche nach längsten Programmpfaden*. Diplomarbeit, Fachbereich 14, Informatik, Universität des Saarlandes, Saarbrücken, September 1998.
- [9] P. Puschner and A. Schedl. *Computing maximum task execution times with linear programming techniques*. Technical report, Technische Universität, Institut für Technische Informatik, Wien, April 1995.
- [10] R. T. White, F. Müller, C. A. Healy, D. B. Whalley, and M. G. Harmon: "Timing Analysis for Data Caches and Set-Associative Caches", In *Proceedings of the IEEE Real-Time Technology and Applications Symposium: RTAS'97*, June 1997, pp. 192-202.
- [11] P. Puschner and A. Schedl. "A tool for the computation of worst case task execution times". In *Proc. of the 5:th EUROMICRO Workshop on Real-Time Systems*, 1993.
- [12] A. Ermedahl and J. Gustafsson. "Deriving Annotations for Tight Calculation of Execution Time", *Proceedings of Euro-Par'97*, LNCS 1300, pages 1298-1307, August 1997.
- [13] C. Healy, M. Sjödin, V. Rustagi, and D. Whalley. "Bounding Loop Iterations for Timing Analysis", In *Proceedings of the IEEE Real-Time Applications Symposium; RTAS'98*, June 1998, pp. 12-21.
- [14] P. Altenbernd. "On the False Path Problem in Hard Real-Time Programs", In *Proceedings 8th Euromicro Workshop on Real Time Systems*, 1996.
- [15] Y.-T. S. Li and S. Malik. "Performance analysis of embedded software using implicit path enumeration". In *Proceedings of the 32th Design Automation Conference*, pages 456-461, 1995.
- [16] T. Lundqvist and P. Stenström: *Integrating Path and Timing Analysis using Instruction-Level Simulation Techniques*, Technical Report 98-3, Dept. of Computer Engineering, Chalmers University, Gothenburg, February 1998.
- [17] S.-S. Lim, Y. H. Bae, C. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Ki. *An accurate worst-case timing analysis for RISC processors*. IEEE Transactions on Software Engineering, 21(7):593-604, July 1995.
- [18] Homepage for SPEC (The Standard Performance Evaluation Corporation): <http://www.spec.org>.
- [19] Homepage for EEMBC (The EDN Embedded Microprocessor Benchmarking Consortium): <http://www.eembc.org>.
- [20] Homepage for the ASTEC Competence Center: <http://www.docs.uu.se/astec>.
- [21] J. Engblom. "Why SpecInt95 Should Not Be Used to Benchmark Embedded Systems Tools", In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, May 1999.