Static Properties of Commercial Real-Time and Embedded Systems

Results from the MARE Project

[Measurements of Actual Real-Time and Embedded Programs]

Release 2
November 25, 1998

Jakob Engblom

Hard-Real Time Systems Group,
Department of Computer Systems,
Uppsala University, Sweden

Author's address:
Department of Computer Systems
P.O. Box 325
SE-751 05 Uppsala
Sweden
Email: `jakob@docs.uu.se`

ASTEC Technical Report 98/05
DoCS Technical Report 98/102

Abstract

We have used a modified C compiler to analyze a large number of real-life commercial C programs used in real-time and embedded applications. The total size of the sample used in this study is more than 300 000 non-comment lines of C code, from 13 different applications obtained from 6 different companies. Only the 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 code was written for 8- and 16-bit embedded systems.

The purpose of the study is to guide our future research into practically useful WCET analysis methods by identifying properties of real-time programs.

The programs studied were from application fields where parts of the program can be considered timing critical, like control loops and real-time signal encoding/decoding and protocol processing.

The most important conclusions for the design of WCET tools are that:

- *It is not reasonable to assume that an entire real-time system is available as source code. Parts of the system will very likely only be delivered as object files, for example OS code, libraries, and company-specific reusable components.*
- *Most functions in a program have very simple control flow. This fact should be exploited to optimize WCET analysis.*
- *Automatically-generated code is being used, even for small processors. This has the following implications: parts of the program have not been written by humans, which means that it is not possible to ask programmers for help in WCET analysis. Furthermore, automatically generated code has a tendency to have a more complex structure than human-written code, which invalidates certain simplifying assumptions, such as "real-time programs do not use recursion".*

Rounding off the study, some other aspects of interest for embedded compiler implementers have been investigated: the distribution of variable sizes and the composition of C switch statements.

1. Introduction

The MARE (*Measurements of Actual Real-time and Embedded programs*) project is an attempt to quantitatively investigate how real real-time and embedded programs are written.

The work has been performed by Jakob Engblom, PhD Student at Uppsala University and IAR Systems AB, with helpful input from the other WCET researchers in the real-time systems group in Uppsala[1] and other WCET groups in Sweden[2].

1.1. Rationale

This study is based the observation that when researching worst-case execution time analysis or building embedded compilers, we often need information about how the programs we intend to process in our tools are written. There is little published data in the field, and therefore we have gathered a set of real programs and performed measurements on them.

Knowledge about how programs are written is useful for the following purposes:

- **Performing relevant research**: our goal is to produce a tool that can be used in industrial practice. This requires knowledge about how industry actually writes code. There is little to be gained from making simplifying assumptions not supported by reality.
- **Prioritizing research**: there are many research problems to be solved in the WCET field. Insight into how real programs are written allows correct priorities to be set – 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 – often with little empiric support. We hope to provide (at least some) relevant empiric measurements.

If we want to perform relevant research, we need a map of the world we are researching.

1.2. Material

Thanks to our industrial partners, we have been able to obtain actual source code from actual commercial software used in real-time and embedded programs. Unfortunately, we cannot distribute the code or name the companies involved, due to non-disclosure agreements. All the code was written in C, using ANSI C extended with compiler specific and machine specific keywords.

The total amount of analyzed code:

- 13 different applications. The application areas were telecommunications (including protocol management), vehicular control applications, and home and consumer electronics. On average, two applications are from the same source. Most of the applications are in use in products on the market today.
- 477 source files (`.c`), using 942 header files.
- 334600 lines of code (LOC): with a large span in sizes: the largest program was about 123000 LOC, the smallest around 2000 LOC.
- 5579 functions were defined[3].
- 17173 variables defined.

The processors used in the applications were:

- The 16-bit Hitachi H8 processors – called H8/16 in the discussion in Sections 4.3 and 3.8 – this includes a number of processors: H8S, H8/300H.
- The 16-bit Siemens C166 family.
- The 16-bit Mitsubishi MELPS 7700.
- The 8-bit Hitachi H8 processors – called H8/8 in the discussion in Sections 4.3 and 3.8 – this family of processors include the H8/300L, the plain H8/300, and the H8/500.
- The 8-bit Zilog Z-80.
- The 8-bit Motorola 68HC11.

[1] In particular, I would like to thank Andreas “Ebbe” Ermedahl and Hans Hansson.

[2] I have received a lot of help from Jan Gustafsson at Mälardalens Högskola (MDH) in Västerås and Thomas Lundqvist at Chalmers in Göteborg.

[3] In the C language, a variable or function is *defined* when it is given memory to reside in or code to implement it. An entity can only be defined once in a program. A function prototype *declares* a function, and an extern declaration of a variable declares it. An entity may be declared several times without harm.

We do not pretend or infer that this is an exhaustive or even representative sample of all real-time and embedded applications, but we do think that the study gives a rough impression on how real programs are written. Conclusions regarding the presence of certain features are probably accurate (since they have been observed in the program code), but it is harder to draw firm conclusions about the absence of features.

Note that we have measured *all* code in the studied programs, not just the time-critical parts. The motivation for this is that although only a small part of a program may be time-critical, performing WCET analysis on this part will probably require knowledge about the entire program. In a commercial environment, with a programmer working in a small part of a large project, assuming that each programmer has a good intellectual grasp of how the entire system operates and how it affects her small part is not reasonable.

1.3. Structure of this Report

This paper consists of the following sections:

- Section 2 discusses the technical and methodological aspects of the study: how we measured, and which measures were collected.
- Section 3 discusses the measurements relevant to WCET analysis, and the conclusions we have drawn.
- Section 4 discusses other measurements of interest, especially for embedded compiler builders.
- Section 5 provides a final discussion on the conclusions to be drawn from this study.

2. Methodology

The tool used in the MARE project was a research compiler created with the IAR Systems AB C/C++ compiler as a basis (we call it the MARE compiler). A statistics module replaced the code generator. The front-end, consisting of the C/C++ parser and function-level optimizer was kept. Only the C mode of the compiler was enabled. The optimizer was used in moderate size optimization mode, since size is usually the constraining factor in small embedded systems.

The backend of the MARE compiler receives an intermediate representation of the parsed and optimized program, and converts it to an internal format suitable for the analyses performed. Note that this has the implication that our measurements are performed on the optimized intermediate code for the program, and not on the source code. A number of standard compiler data structures are built: flow graph, dominator trees, etc.

The MARE compiler processes C source files one at a time, and outputs a number of files containing measurements (see Section 2.7 for the format of the files). These files are then joined into aggregate statistics files (one for each type of measurement).

The data items in the files are tagged to allow the source of each item to be determined (usually we trace a measurement back to the file, and maybe the function, where it was collected).

Since the source files were spread over a number of directories, a set of PERL programs and shell scripts were used to automatically gather the information. Special files were used to track the specific compiler settings necessary to compile each program. The whole system is reasonably fast, but slow if compared to ordinary C-compilers. We manage to compile our entire material in about two hours (on a 300 MHz Pentium II running Windows NT).

2.1. Why We Measure Optimized Intermediate Code

The statistics backend receives an *optimized* intermediate representation of the program. The reasons for applying the optimizer to the parsed code and to measure at the intermediate code level are the following:

First, the properties of intermediate optimized code are *more relevant* to future tool construction and research than source code properties:

- Our research plans are to insert the semantic analysis component of a WCET tool at the same level in the compiler: the optimized intermediate code.
- The optimizer is one of the problems a WCET tool has to deal with, and we wanted to see how the code looks after optimization.
- The analysis of intermediate code allows us to get to the essential properties of the program, not the accidental properties of the syntactical representation. We are not interested in indentation, variable naming, `while`-loops vs. `for`-loops, etc., since this does not affect machine analysis of code.

Second, there are some good reasons for not looking too much at the source code:

- Many relevant embedded programs are machine-generated, which generally makes them ugly and hard-to-read. Making statements about such programs based on the look 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 (changing looping constructs, for example), which means that studies based on the source code style and syntax are irrelevant. A WCET tool used in industrial practice would not be applied to an obfuscated form of a program.

Finally, performing the MARE analysis after parsing and optimization makes the tool *simpler*:

- We do not need to handle the complexities of the C preprocessor – all preprocessing has already been performed. Considering the use some programs make of the preprocessor, this is a significant simplification.[4]
- Erroneous code is never encountered – we get error handling for free from the modified compiler.
- The compiler front-end handles special ANSI C extensions – we use the keyword-handling mechanisms already present in the IAR C compiler to handle extended keywords.
- The set of operations to handle is smaller – we only need to handle the essential operations of a program: mathematical operations, comparisons, jumps, function calls, etc. We do not care about the different ways to express them syntactically.
- The resulting code is canonicalized by the optimizer, which makes the analysis simpler and faster, with fewer special cases to handle.

[4] Another problem with source code analysis: how should preprocessor macros be treated? Like special keywords, or like the constructions they actually expand to? They certainly affect the appearance of the program.

2.2. Mimicking other Compilers

The code we studied was written for a variety of embedded C compilers. This gives rise to certain problems, since there are many compiler-specific features used to write embedded programs, and several aspects of C semantics are implementation dependent. To handle this, we allow the MARE compiler to mimic certain aspects of a compiler. The mimicking involves configuring the following aspects:

- The size of types: how large is an `int`? How large is a double floating-point value?
- Special keywords for modifying function declarations, variable memory placement, etc.
- The available intrinsic functions. Intrinsic functions are functions which are converted into inline assembly code by the compiler, typically to allow access to features not expressible in the C language (like enable/disable interrupts) and to chip-specific features which cannot today be utilized by the compiler (like saturated arithmetic).

Since the IAR compiler on which the MARE compiler is based is an embedded C compiler, the support required to handle special keywords and intrinsic functions was already present. It was enough to declare the special keywords available: there was no need to change the parser (which would be needed for an ordinary C/C++ front-end). However, in some cases `#define`s had to be used to make certain keywords have a syntax acceptable to the IAR compiler. This does not affect the measurements, since we only change the syntax of the program and not the semantics.

2.3. Compilation Problems

There were some unexpected problems encountered when analyzing the programs, which in some cases required modification of the source code to allow the programs to be analyzed:

- In some cases, we were given more files than actually made up a program. To get correct results, we had to remove the extraneous files. This was performed manually, using include-hierarchy graphs to figure out which files actually belonged to a program.
- Our research compiler is based on a modern and quite picky front-end, and many typing errors which passed unnoticed before are now caught and reported as errors and/or warnings. In some cases, this forced us to modify the code, but the effect should be minimal. Usually, it was sufficient to add a few casts in the correct places, which does not affect our statistics. An especially common case is `const`-misuses – most old C compilers treat `const` as a recommendation, but today, in the spirit of C++, the checks are hard.
- Some programs are structured with `.c`-files including other `.c`-files, in which case we only processed the "root" `.c`-files, in order to avoid counting the same variables and functions twice.

2.4. Incomplete Programs

The file sets in the study only contained partial applications, or complete applications that rely on an operating system. This limited the usefulness of certain measurements, since only functions actually defined in the examined files have been subject to measurement (and not functions which are only declared and called):

- Function modifications using keywords: the statistics on function modification (see Section 4.1.1) indicate that function modification was quite uncommon. However, there were some instances of modified functions only being present in header files and never defined in the source files. These functions either belonged to the operating system or to parts of the program which we did not have available. This might cause some errors in the measurements.
- The same problem applies to variables: many memory-attributed variables are system entities (I/O, peripherals, MMU-control, etc.) which are never defined in an application, only declared (in machine-description header files) and referred to.

The fact that operating system code is not analyzed is not a problem: this study is about embedded applications, not about operating systems. Similarly, the lack of analysis of system entities is unproblematic, since the number of hardware registers depends on the chip used and not on how the program is written.

The incomplete applications may be a problem, since it is possible that the analyzed parts are in some way atypical. We do not think that this is the case, since there are no big differences between the results for the partial and the complete programs.

2.5. Available Information at the Intermediate Level

The following information about the program is available at the intermediate code level (after some basic control-flow analysis of the code):

- All declared and referenced symbols: their type, name, and other properties.
- The program flow structure, as seen by a compiler: basic blocks and the flow edges between them (the edges represent jumps and flow-through paths).
- Dominators and post-dominators.

- The call graph (for a single file at a time).
- All operations performed in the program (obviously), in the form of lists of intermediate instructions: `if`, `goto`, `add`, `sub`, `mul`, `store`, `load`, `load constant`… (see any compiler textbook).
- The references to variables and functions are symbolic: we have the names of all entities, but no memory addresses, since we have not generated code.

The following information is **not** available at the intermediate level (the most important difference from the source code is that all syntax is gone):

- “Loop kind”: `for` or `do-while` or `while` or `if/goto`: the difference cannot be told.
- `typedef` type names.
- Names of structure types, union types, and other aggregates.
- Comments
- No program-global information, unless the entire program is stuffed into a single file and then run through the research compiler.

The use of intermediate code means that we cannot detect issues like programming style, but only issues revolving around the data and actions actually carries out by a program, no matter how the effect is achieved.

2.6. Analysis Tools

The analysis of the collected measurements was performed using the Microsoft Excel spreadsheet. Graphs were used to get visual impressions of the data, and cross-tabulation and selection of data to look for relationships.

2.7. Format of Collected Data

This section describes all the measurements collected by the MARE compiler.

2.7.1. File.MARE

One line per file analyzed.

Field name	**Type**	**Meaning**
Program Information		
`Filename`	text	Name of the compiled file, including `.c` suffix.
`Programname`	text	Name of the program the file belongs to
`Application`	text	Name of the application area for the program
`TargetCPU`	text	Name of the CPU the program was written for
Variables		
`Variables`	int	Number of variables in the file
`ScopeGlobal`	int	Count of global variables
`ScopeStatic`	int	Count of static variables
`ScopeAuto`	int	Count of auto variables
`ScopeParam`	int	Count of parameters
`VarConst`	int	Count of `const`-declared variables
`VarVolatile`	int	Count of volatile-declared variables
`VarMemAttr`	int	Count of variables with explicit memory attributes set
`VarNoInit`	int	Count of variables with no-init attribute (variables which should not be cleared on program startup)
Variable types (number of variables of each type)		
`char`	int	8-bit, signed
`unsigned_char`	int	8-bit, unsigned
`short`	int	16-bit, signed
`unsigned_short`	int	16-bit, unsigned
`long`	int	32-bit, signed
`unsigned_long`	int	32-bit, unsigned
`float`	int	32-bit floating point numbers
`double`	int	64-bit floating point numbers
`struct`	int	All structures
`union`	int	All unions
`array`	int	All arrays
`pointer`	int	Pointers to data
`code_pointer`	int	Pointers to functions

General counters		
`Switches`	int	The number of switch statements in the file
`DecisionNests`	int	The number of decision nests in the program.
Function calls		
`FunctionCalls`	int	Total number of function calls
`StdLibCalls`	int	Calls to standard library
`UserLibCalls`	int	Calls to user library
`IntrinsicCalls`	int	Uses of intrinsic functions
`SameFileCalls`	int	Calls within the file
`FptrCalls`	int	Calls to function pointers
`ExternalCalls`	int	Calls to other (external non-library) functions
`RecursiveLoops`	int	Number of recursive loops in the call graph
`RecursiveFunctions`	int	Number of functions involved in recursive loops
Function declarations		
`Function`	int	Number of function declarations
`FunctionModified`	int	Number of modified functions
`FunctionMonitor`	int	Monitor-declared functions
`FunctionInterrupt`	int	Interrupt-declared functions
`FunctionTrap`	int	Trap-declared functions
`FunctionTargetTAttr`	int	Target-specific type attributes
`FunctionMemAttr`	int	Functions with memory attributes.
Function composition		
`FunctionInfinite`	int	Number of never-returning functions
`FunctionNoLoop`	int	Number of functions containing no loops
`FunctionNoDecide`	int	Number of functions containing no decisions, i.e. completely straight-line code – no loops, no decisions.
`FunctionUnstructured`	int	Number of functions containing unstructured flow graphs.
Loops and functions		
`GlobalLoopDepth`	int	Maximal depth of loops, including the effects of function calls inside loop bodies. Per file, which limits the value.

2.7.2. *File.MSWT – Switch information*

Contains several lines: one for each switch encountered in the file.

Field name	**Type**	**Meaning**
File identification		
`Filename`	text	Name of the compiled file, including `.c` suffix.
`Function`	text	Name of the function for the switch.
Information		
`Type`	text	Name of the deciding type (same names as used in the `.mare` file)
`Min`	int	Minimum value of switch (signed long interpretation).
`Max`	int	Maximum value of switch (signed long interpretation).
`MaxDelta`	int	Largest difference between two successive cases
`ActualCases`	int	The number of cases in the switch
`Span`	int	The size of the span [min, max]

2.7.3. *File.MLOP – Loop information*

Contains several lines: one for each loop-nest encountered in the file.

Field name	**Type**	**Meaning**
File identification		
`Filename`	text	Name of the compiled file, including `.c` suffix.
`Function`	text	Name of the function for the loop.
Information		
`Depth`	int	Depth of the loop nest.
`Exits`	int	Number of exit edges in the loop nest – both from inner

		loops to outer loops, and out from the loop nest altogether.
`Blocks`	int	Number of blocks in the (largest) innermost loop.

2.7.4. File.MIFS – If-nest information

Contains several lines: one for each if-nest encountered.

Field name	**Type**	**Meaning**
File identification		
`Filename`	text	Name of the compiled file, including .c suffix.
`Function`	text	Name of the function for the if-nest.
Information		
`IfDepth`	int	Depth of the if-nest

3. Results Relevant for WCET Analysis

The original motivation for the MARE project was to investigate whether it would be possible to simplify our initial approach to WCET analysis by ignoring hard-to-handle but unimportant aspects of the C language. In this section, we go through the results of the measurements we consider relevant for the design of WCET analysis tools[5]. In the next section, we will look at some other interesting observations which do not have any immediate interest for WCET analysis.

3.1. Recursive Calls

Recursive calls are theoretically equivalent to imperative loops, but are much more difficult to analyze in a language not designed for recursion (like C). The study found a total of 14 recursive call loops involving 18 functions in the (almost) 5600 functions we analyzed.

- 2 were in user-interface code managing menus.
- 1 was a filter which used list-processing and dynamic data.
- 1 was a recursive expansion used in text processing (naturally recursive algorithm).
- 10 were part of automatically generated code (from an SDL-tool).

What might be surprising is that these programs were written to run on small 8-bit processors (albeit with reasonable stack handling).

Obviously, user interface and text processing code is unlikely to be hard real-time critical (i.e. requiring WCET analysis). The automatically-generated sequences of code, however, are often protocol handlers which do have timing constraints on them.

Conclusion: recursion is probably not too common, but cannot be ignored in the long run. Ignoring it in a first version of the tool is not too limiting.[6]

3.2. Unstructured Flow Graphs

Unstructured flow graphs (also known as non-reducible graphs) make analysis of a program more complex. We found 18 instances of non-reducible flow-graphs in the examined functions. Since it is well-known that all programs can be written without resorting to unstructured flow graphs, the functions containing unstructured flow-graphs were examined to see why they were unstructured.

Result: most of the functions were automatically generated state machines implemented using gotos. The few remaining were human-written functions with no unstructured constructions in the source code; the compiler had introduced unstructured flows during the optimization process.

Thus we cannot reasonably expect all our programs to be perfectly structured, especially not when we use a high-powered optimizer. Furthermore, the automatically-generated code over which the programmers typically have very little control contains several instances of unstructured flow graphs.

Conclusion: we must be prepared to deal with unstructured flow graphs, and since this problem is easier than dealing with recursion, we should do so quite early in the development process.

Alternatively, we could discuss the quality of generated code with the code-generator vendors, and maybe try to avoid using destructuring optimizations for time-critical programs. This is a much more difficult solution however, especially since avoiding unstructured optimizations severely hampers the efficiency of an optimizer.

3.3. Loops

Loops are one of the principal difficulties in WCET analysis (the other two are conditional statements and function calls). Several measurements were performed in order to try to determine the difficulty level of loops to expect when (statically) analyzing real-time and embedded programs.

There were 1414 loop nests in the analyzed programs. Note that the way in which the loops are written in the source code is of no importance here: `for` loops, `while` loops, `do-while` loops, and loops constructed using `goto`s are all considered equivalent. This is the usual compiler construction view of loops as edges going back into the flow graph.

The difficulty of obtaining a loop bound for a loop is sometimes related to the source code: for a textbook-like `for` loop with an obvious bound is easy to analyze. However, it is still possible for the program to exit

[5] We are considering *high-level analysis* here: tools that analyze the program flow in order to find loop bounds, infeasible paths, etc.

[6] Another course of action would be to discuss the quality of the generated code with the companies producing code generators.

a loop prematurely using `break`. To be relevant, measurements of the difficulty of loop bounds analysis require a specific loop bounds analysis model to be assumed. The difficulty is always relative to a certain analysis, and cannot be measured as an absolute value. This is why no such measurements are included in the MARE project.

3.3.1. Loop nesting depth

The nesting depth of loops affect the complexity of the loop analysis. Our data indicates that few loop nests are very deep: 91 % of the analyzed were single-nested, 8 % double-nested, and only 1 % triple- or quadruple-nested. There were no loops with a loop nesting depth greater than four. Figure 1 shows the distribution of loop nesting depths.

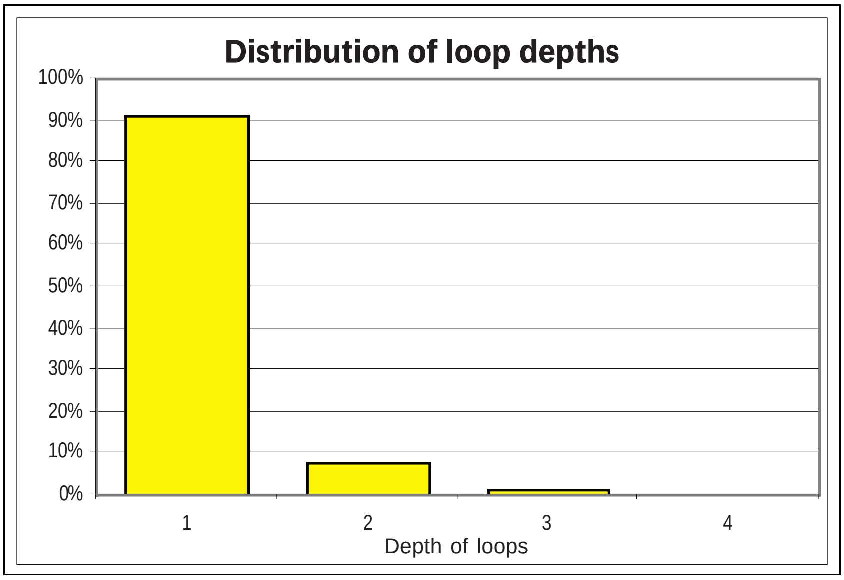

Figure 1

Note that the loop nesting depth of *programs* considered across functions certainly can exceed four. The present study was limited to file scope, and the greatest loop nesting depth across functions found was seven (for example, this could mean that a function containing a quadruple-nested loop was called from the innermost loop in a triple-nested loop).

Conclusion: we need methods for loop bound analysis and data structures to describe loop bounds which are linear (or close to linear) in the loop nesting depth. Methods and data structures which grow in size and time depending on the actual number of iterations of loops are not useful in the general case (since the expected size of the data and would be exponential in the depth of the loop nests, and the run time proportional to the run time of the program analyzed).

3.3.2. Number of exits

If loops have more than one exit, analysis is made slightly more complex. Extra exits typically take the form of `break` statements or `gotos` to labels beyond the end of the loop. The main problem for WCET analysis is to correctly account for the fact that only part of the loop body is executed on an early exit. Figure 2 shows the distribution of extra exits for all the loop nests in the analyzed programs.

- -1 means that the loop never exits: we have fewer exits than loops.
- 0 means that the loop nest has exactly the same number of exits as loop nesting levels, i.e. each loop has exactly one exit. This is a C loop written without any “`break`” statements.
- Positive numbers indicate that the loop nest contain extra exits.

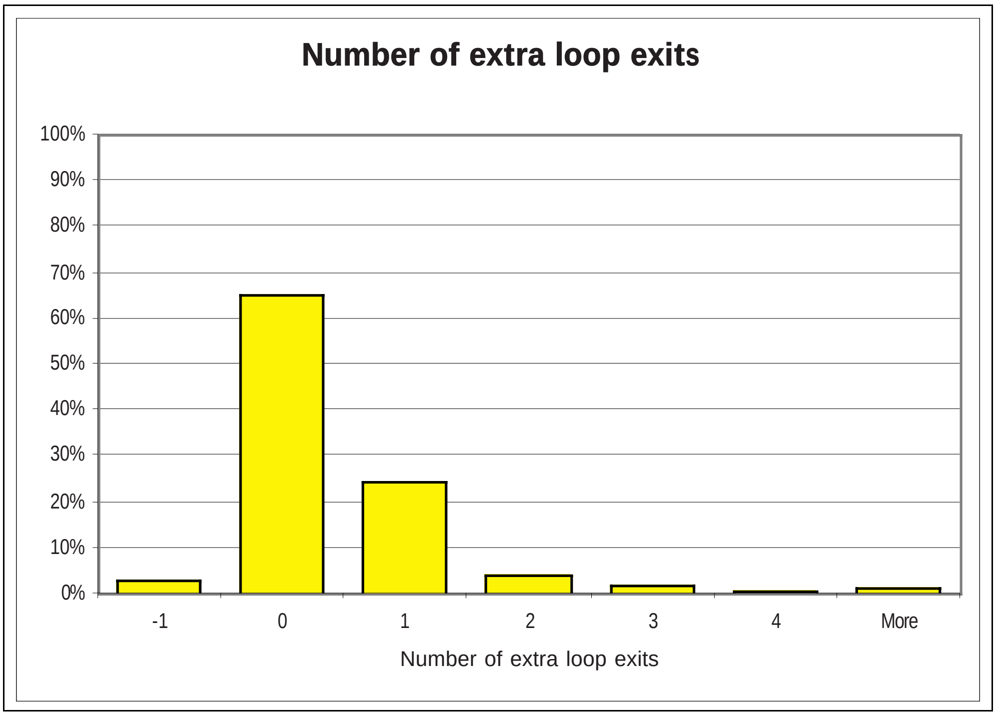

Figure 2

About 65 % of all loops contained the same number of exits as loop nesting levels, indicating well-behaved simple loops. About 3 % contained fewer exits, meaning that they never terminate. 25 % contained one extra exit. The loops with more than one extra exit comprised about 7 % of the loops.

Conclusion: there are several loops with multiple exits, even though single-exit loops dominate. A loop analysis method should be able to handle the case that a loop exits early in the body.

3.3.3. Loop complexity

The complexities of the bodies of the innermost loops of loop nests have been measured, to get a rough measurement on the complexity of the code in loops. The measurement used is the number of basic blocks in the body of the innermost loop. Figure 3 shows a graph of the results.

Only 11 % of all loops have simple, straight-line bodies (a single block, which is equivalent to a simple `do-while` loop, see Figure 4). The large number of loops (34 %) with two blocks represent simple `for`-style loops which begin with a check for exit, perform their work, and then loop back to start (see Figure 4). About 90 % of all loops have 10 or fewer blocks in the innermost loop.

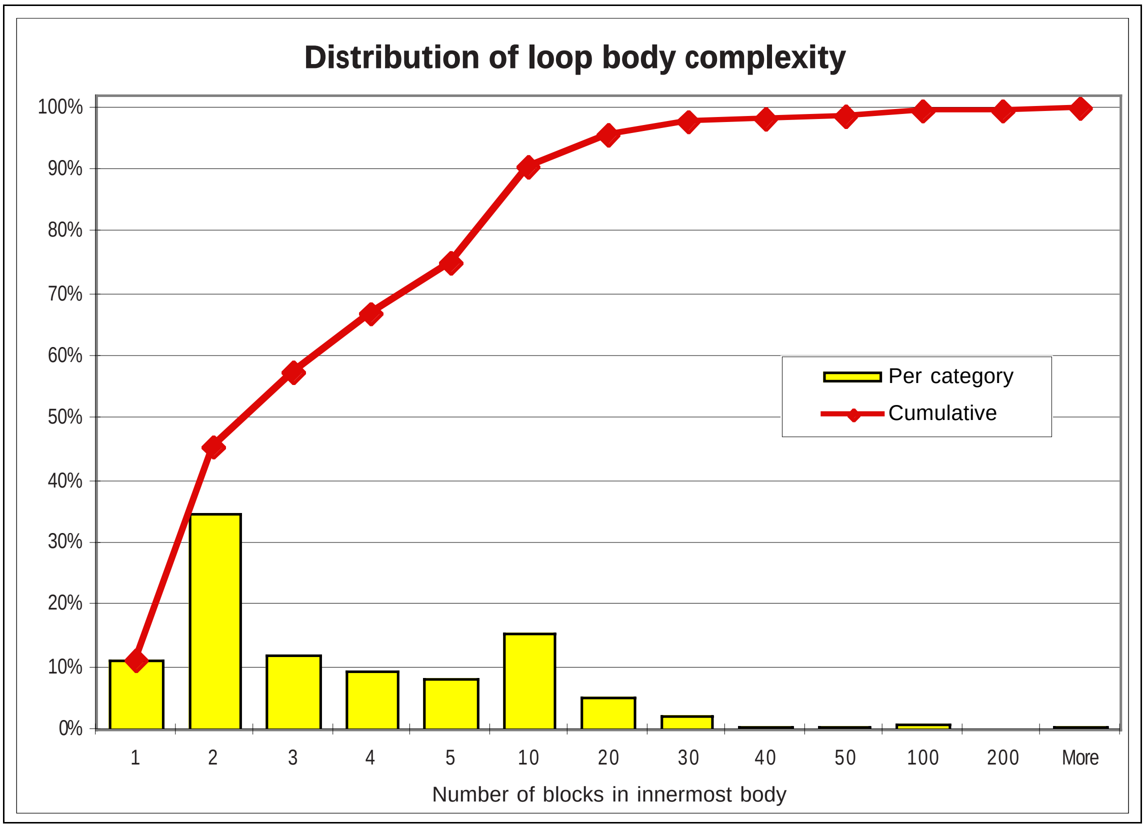

Figure 3

The loops with just one or two blocks are simple flow-through loops with no other decisions than the loop termination condition, which are simple to analyze. Loops with more than two blocks in the body are more complex, since they necessarily contain at least one decision more than the basic loop control.

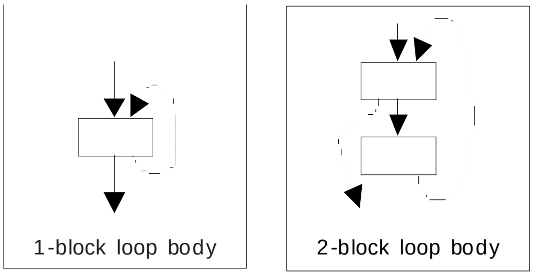

Figure 4: Structure of loop bodies

Conclusion: we cannot assume that loops have simple bodies. This points in the same direction as the conclusions in Section 3.3.2 – loop body analysis will have to handle complex loop bodies, with multiple exits.

3.4. Conditional Statements

Conditional statements are one of the main determiners of program flow. The complexity decision was measured by looking at the *decision depths* of each decision-nest.

Definition: a *decision nest* is a single-entry-single-exit fragment of the flow-graph where the bottom (exit) node post-dominates the top (entry) node of the flow graph fragment, and the top node dominates the bottom node. The intuition is to capture a region of the graph where the flow splits into several branches and then rejoins again.

Definition: the *decision depth* of a decision nest is the maximum number of splits in the flow graph that we can traverse on the path from the top to the bottom node. An ordinary if-then-else has a decision depth of one. A plain switch also has a depth of one. Nested ifs, switches, and loop constructs are necessary to produce greater depths.

The programs we measured contained 5604 decision nests. The distribution of the decision depths are shown in the graph Figure 5.

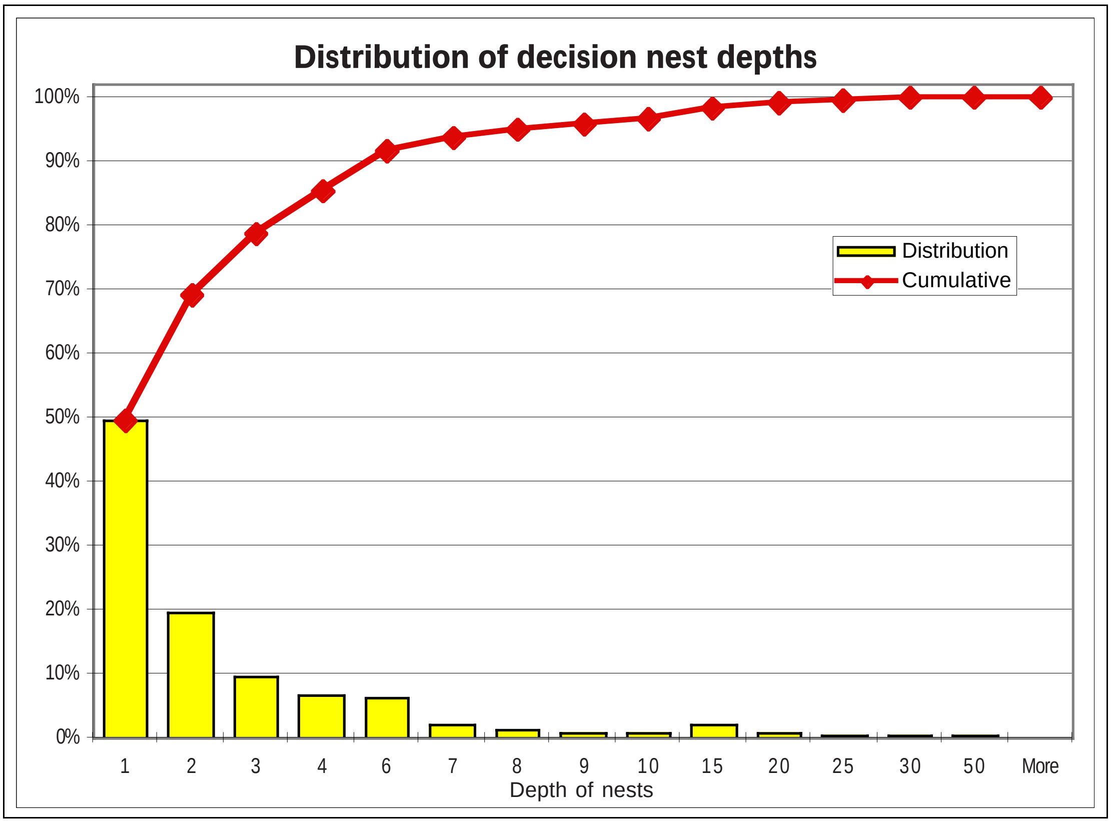

Figure 5

We note that almost half (49 %) of the decision nests are simple, but that there are some extremely complex nests. The problem with complex decision nests comes when we want to describe or analyze which paths can actually be taken: the *infeasible paths* problem.

The maximum observed depth in the investigated sample is 121, which means that there are 2^{121} possible paths through that particular nest. This is caused by a structure of the following type:

```
if( C1 )
{
 if( C2 ) {}
 if( C3 ) {}
 ...
}
```

In other words, inside an if-statement there is a (long) series of subordinate ifs, which means that the number of decisions made before the flow rejoins is 121. This kind of structure is reasonable, but hard to analyze, pointing to a very tricky problem for WCET analysis.

Conclusion: as for loops, we need compact ways of representing feasible and infeasible paths, and efficient ways to deduce the paths which do not need to enumerate all possibilities.[7]

One approach would be to use exact techniques for simple nests, and switch to less exact methods for more complex nests.

3.5. Function Calls

The third major problem for WCET analysis is function calls. The programs examined contain 13835 function calls. Figure 6 shows how the function calls are distributed are distributed between the following categories:

- External calls are to functions defined in other files (other parts of the same program, operating system, etc.).

[7] This is a requirement on path analysis. We have no good ideas on how to achieve this – considering the huge complexity of certain decision nests, it is by no means a trivial task.

- Function pointer calls are calls to function pointers.
- Same file calls are to function defined in the same file.
- Intrinsic calls are calls to intrinsic functions (for an explanation of intrinsic functions, see Section 2.1).
- Standard library calls are calls to the C standard library (`printf()` etc.).

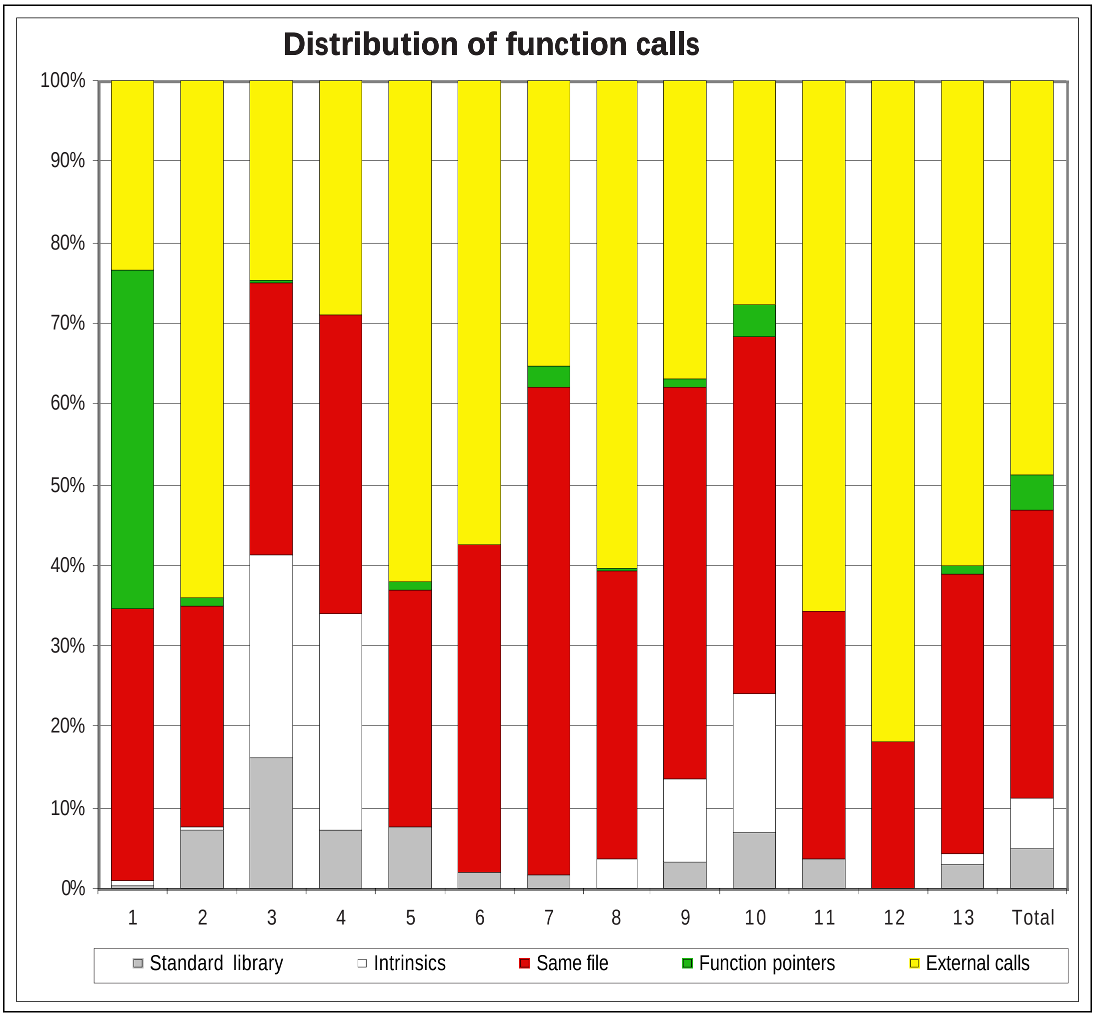

Figure 6: Types of function calls, for each program and for the total sample.

3.5.1. *External calls*

External calls pose a problem if the entire program is not available for analysis, which is highly unlikely, since most programs make use of libraries and/or operating systems which are unlikely to be delivered in source form.

However, to allow linking, the external code must be compiled with the same compiler as the application program (or at least a compiler which delivers the same intermediate format). This means that it is quite reasonable to have the manufacturers analyze their program parts with the same compiler/WCET tool as the application programmer is using, and then deliver the results as a part of the product package (just like operating systems are delivered in binary format today).

Conclusions:

- The first conclusion is that analysis of WCET across functions on a file-per-file basis is impossible.
- The second conclusion is that it would be advantageous if a WCET tool would allow “separate compilation” just like C compilers do. It should be possible to (partially) analyze each file in isolation, and then “link” the results.

3.5.2. *Function pointers*

Function pointers comprise a problem for WCET analysis (and all other forms of data and/or control-flow analysis), since a program can theoretically call *any* function in the program by using a function pointer (we ignore the pathological possibility of jumping to some random place in memory, since this should typically result in a crash). WCET analysis needs to at least limit the set of reachable functions, which can only be done by using global pointer analysis.

In program 1 in the graph, about 40 % of all function calls are via function pointers. This is because the program uses a large global dispatch table to allow parts of the program to be changed without forcing recompilation and re-linking of other parts. Four of the programs do not use function pointers at all, and the total incidence is about 4 %.

Conclusions:

- There is obviously a requirement to handle function pointers, since they are used in embedded and real-time programs.
- Some method for global pointer analysis needs to be employed in order to figure out whereto function pointers point.

3.5.3. *Intrinsics and the standard library*

Control over the compiler also gives control over the intrinsic functions and the standard library. Intrinsic function can be handled like any other code in the WCET analysis: it is simply a special way to enter certain machine instructions into the code.

The standard library is used in most programs (11 or 13 programs used it in this study). Typically, the standard library is the same for all programs written using the same compiler (it is delivered as a set of header files plus a set of compiled object files – access to the source code normally costs extra), and it would be sensible to pre-analyze it (just like for other libraries, as discussed in Section 3.5.1). In most respects, the standard library is just like other libraries, except that it is under the control of the compiler manufacturer (which in our case means the WCET tool manufacturer).

Conclusions:

- Intrinsic functions pose no problems.
- It would be advantageous if the standard library could be “pre-analyzed” to speed the WCET analysis process.

3.6. Non-terminating Functions

In general purpose computing, a function which never returns does not make sense, since a program is invoked to perform some given task, and is expected to terminate once that task is completed. In embedded programs, this is not necessarily the case, since a program is written as a set of task where certain tasks should run as long as the device is on. The main body of a task is typically written as a never-ending function.

Confirming this hypothesis is the fact that ten of the thirteen programs we investigated (and note that we are usually talking about fragments of complete applications) contained non-terminating (never-ending) functions.

Assuming that the non-terminating functions are tasks, it would make sense to estimate the time needed between two operating system calls (like a `delay` which puts the process to sleep, waiting for its next invocation). This would require the cooperation of the operating-system vendor in the creation of a WCET tool (which makes sense).

Note that non-terminating functions as defined here are functions which do not even contain the possibility of an exit (no return statement or flow that goes to the end of the function). Functions which may be non-terminating, but which do have (conditional) flows ending in an exit from the functions do not count as non-terminating. An non-terminating function must be written in a manner similar to the following:

```
void foo(void)
{
 while(true)
 {
   <no exit or return statements>
 }
}
```

The problem of terminating loop bound analysis for complex loops (where it is hard or impossible to automatically prove that all possible executions will always terminate) is different and much more difficult (equivalent to solving the halting problem). Here, heuristics and maybe user interaction will have to be used to correctly analyze the loops.

Conclusion: the WCET analysis should handle non-terminating functions gracefully, preferably by assuming that they are tasks and giving a timing for the body of the non-terminating loop.

3.7. Simple Functions

Many functions defined in computer programs are quite simple and do not include any control-flow change, like reading or setting a certain value, performing some calculation, or updating some global state.

In the programs we measured, we found that 33 % of all functions contained no control-flow change – i.e. the functions were simple flow-through, one-basic-block functions. Furthermore, 80 % of all functions contained no loops (this includes the 33 % flow-through). The remaining functions contained loops.

Figure 7 shows the results from our measurements:

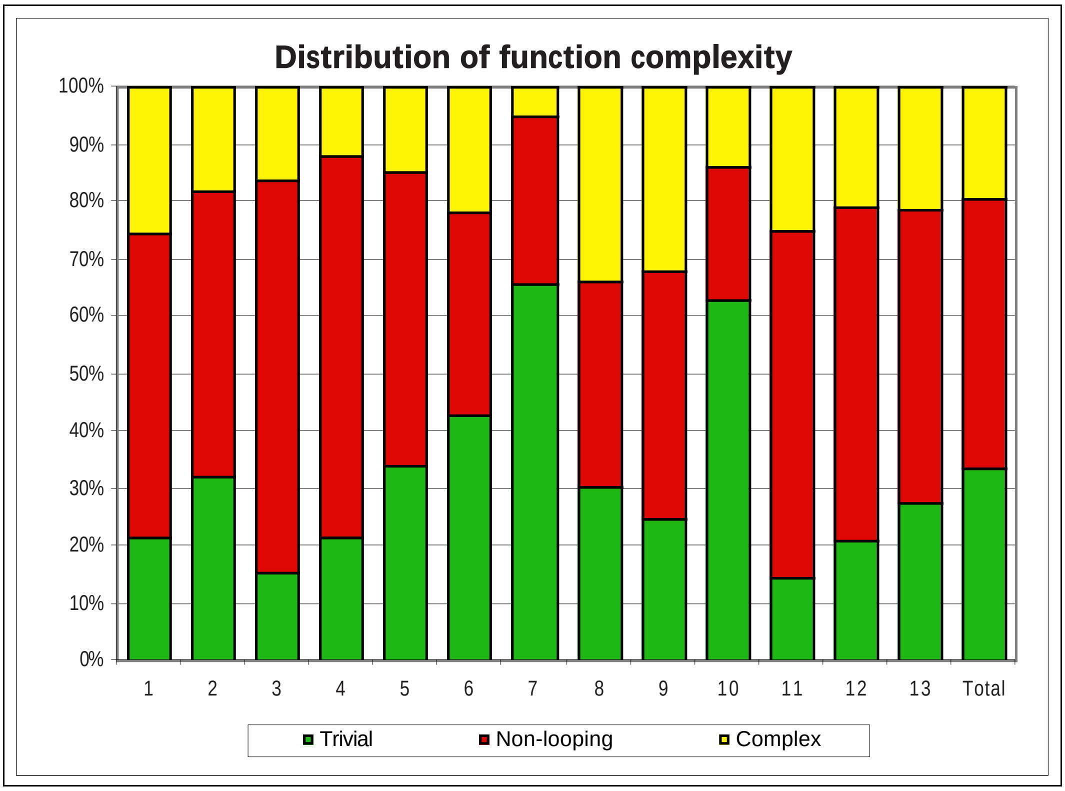

Figure 7: Function complexity for the 13 studied programs, and the total set.

These results are interesting for optimizing WCET tools: a function that lacks control-flow should have an approximately constant execution time (unless caches are taken into account). This should allow a tool to analyze the function only once, and then use the derived result for other calls to the same function. Furthermore, certain methods can be applied to non-looping code that might be more efficient than general methods.

Conclusions:

- A WCET tool should be optimized for only analyzing simple functions once.
- Different strategies could be employed depending on the complexity of a function.

3.8. Variable scopes

The scopes of variables can be a source of problems for WCET analysis. Especially global variables and static variables (local variables which retain their values across calls to the same function) make flow analysis based on data values difficult, since a much larger set of statements can affect the value of the variable.

The concept of scope for variables used in the MARE project is that defined by the C language. There are the following scopes available:

- *Parameters* are parameters to functions. They get their value when a function is called, and they disappear when the function returns.
- *Auto* variables are local to a function. They are appear (with a random value or initialized) when the function is called, and disappears when the function returns.
- *Static* variables are local to a function, but retain their value across function calls. They cannot be modified by any other functions.

- *Global* variables are declared outside any function, and can be modified by any function.

Figure 9 shows the distribution of variable scopes across the programs. No firm conclusions can be drawn from this data. The proportion of global data varies between 5 % and 60 %, but in general a majority of all variables are autos or parameters, i.e. local to function call.

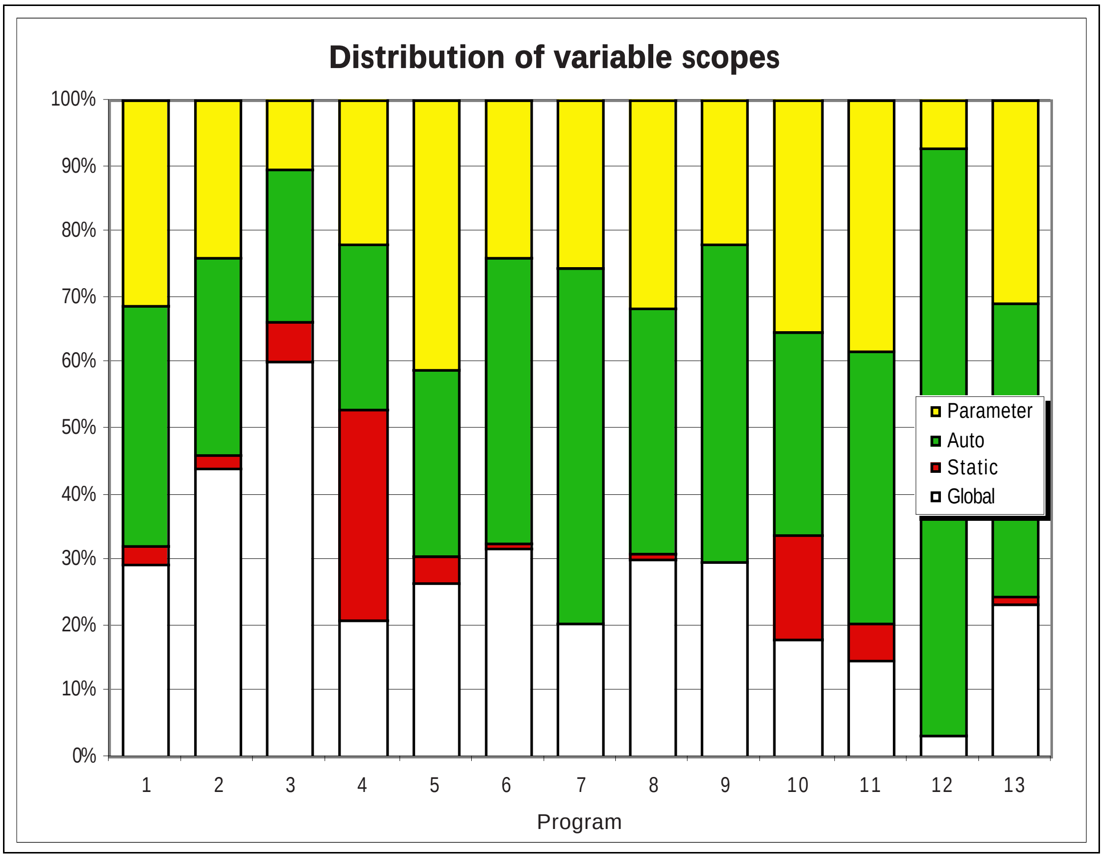

Figure 8: Distribution of variable scopes for the 13 programs and the total set.

Another possibility is that the type of processor has some effect on the scopes of variables used (a common assumption among embedded compiler writers is that programs for processors with poor stack handling, like the Z-80, would have a higher proportion of global and static variables to avoid time-consuming stack operations). Figure 9 shows the result.

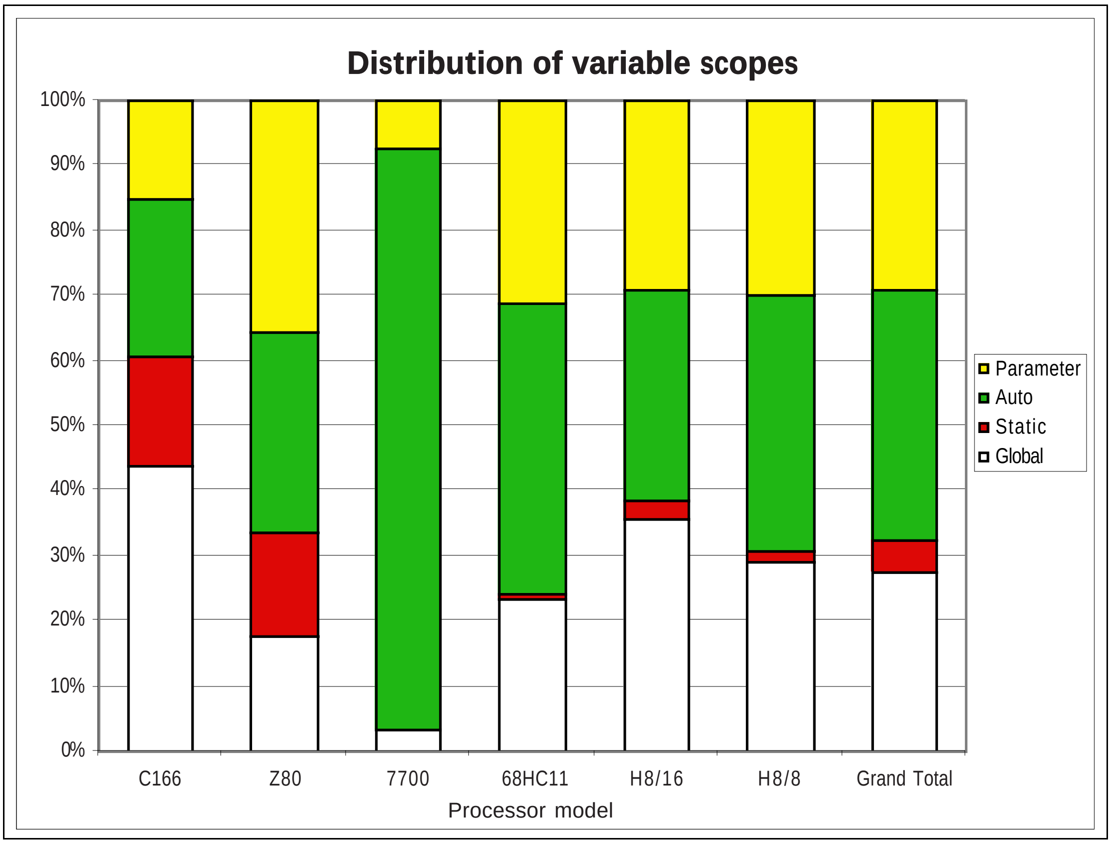

Figure 9

There are few conclusions to be drawn from this data. It seems that the variance has more to do with programming style than the processor used, especially the proportion of global data.

Conclusion: embedded programs do use static and global variables, and this difficulty has to be dealt with.

3.9. Variable types

Figure 10 shows how the relative frequency of the types of variables in the measured programs. The categories used are the following:

- Integer variables: `int`, `char`, `short`, `long`.
- Float variables: `float`, `double`, `long double`[8].
- Structures and unions.
- Arrays.
- Pointers, i.e. pointers to data.
- Code pointers, i.e. pointers to functions.

It is clear that integers and pointers are the most popular types of variables. For WCET analysis, the large number of data pointers is worrying. Note that most pointers are not problematic – pointers used to pass arguments into functions are usually quite easy to handle; the real problems are caused by global pointers, which can point anywhere. Any write to an unknown global pointer invalidates all data value analysis; this can only be handled by employing global points-to analysis (which is common in whole program analysis).

[8] Note that `long double` is hardly ever separate from `double`, and that all three floating point types often have the same size (32 bits). Also note that the few floating-point variables found were all of the `float` type.

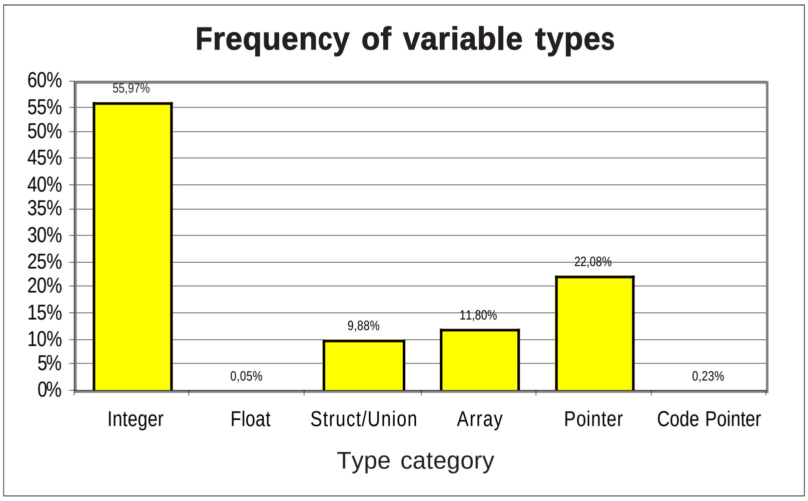

Figure 10

Conclusions:

- Float values can be ignored in a first version (this is probably a quite small simplification, however).
- The large number of pointers make points-to analysis a necessity for an industrial tool. It is likely that handling only simple pointers (function parameters and local pointers) is a reasonable level of ambition for prototype tools.

3.10. Automatically Generated Code

From the investigated programs, it is clear that automatically-generated code is being used in the development of real-time and embedded programs, even for small processors.

The use of automatically generated code makes it much harder for a WCET tool to rely on user annotations. Use annotations are often used today to determine loop bounds (and other relevant information), but the user is unlikely to understand the generated code well enough to give sensible bounds (especially since the code is really hard to read).

Conclusion: automatic analysis of programs is preferable to user annotation, at least for code which has not been written by humans.

4. Other Observations

The previous section discussed various observations and results relevant to the construction of WCET analysis tools. In this section, other observations will be discussed, mainly of interest for embedded compiler writers.

4.1. Embedded Compiler Features

Compilers for embedded systems typically provide features not found in compilers for general-purpose computing. It is interesting to see the frequency of use of such features.

4.1.1. Function modification

A common feature of embedded compilers is the possibility to modify function definitions and declarations to indicate that they are to be used as interrupt handlers, use special calling conventions, and other machine-specific features. This is very convenient for application programmers who do not have to care about setting up interrupt vectors, correctly disabling and enabling interrupts, etc.

About 4.5 % of the functions defined in the investigated code used function modifiers. Out of these, about one fourth were "interrupt" functions, usually with a vector number determining which interrupt vector to attach to the function. The other three fourths were calling convention/memory placements keywords associated with the banked memory models used on the Hitachi H8 and Motorola 68HC11.

It should be noted that the use of a real-time operating system didn't preclude the use of function modifiers. Interrupt functions are still set up, and calling conventions and memory placement modifiers are used.

4.1.2. Variable modification

Variables can be modified to indicate that they should be stored in certain memory areas. This was used on about 1,3 % of all variables.

4.2. Const

About 17 % of all variables were modified by the `const` keyword.

`const` is often used to indicate that a certain "variable" is actually a complex constant, especially for arrays and structures (since C does not support complex constants). The programmer's intention is that the data should be stored in ROM and not in RAM (which saves cost for microcontrollers which typically have quite a lot of ROM for programs and constant data, and only a small RAM to use as a scratch area).

A common idiom was the use of `static const struct` to define complex constants for use as arguments to functions.

4.3. Variable sizes

Figure 11 shows how the sizes of integer variables used vary with the type of processor (`char` means 8-bit values, `short` means 16-bit, and `long` 32-bit). The C166, 7700, and H8/16 are sixteen-bit processors, the others are eight-bit processors. The processors are introduced in a little more detail in Section 1.2.

It is hard to draw firm conclusions from this data. It is clear that smaller data types (`char` and `short`) dominate, and that `long` is very rare. However, `long` is definitely more common on the 16-bit processors (C166, H8/16, 7700) than on the 8-bit processors. The only 8-bit processor with a significant use of `long` is the 68HC11. Also note that two of the 16-bit processors, the 7700 and C166 only have around 50 % `char` variables, while all the other (including the16-bit Hitachis) have 75 % or more `char` variables.

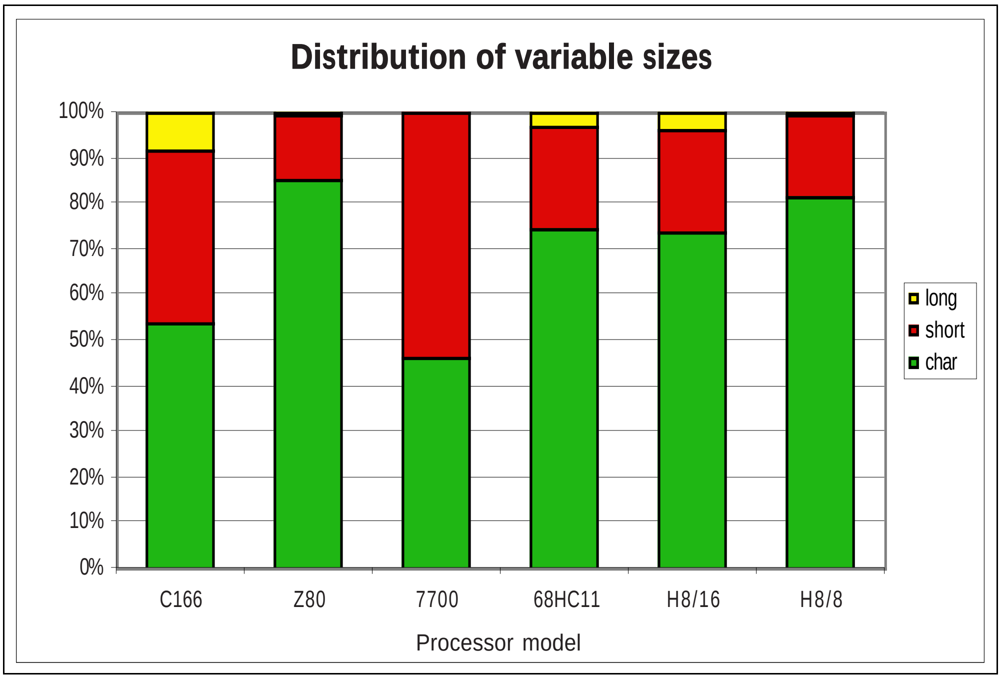

Figure 11

Conclusion: as could be expected, small data dominates programs for small processors.

4.4. Switches

The switches in the investigated programs have been analyzed with respect to a few simple properties. The purpose was to provide some hints as to how to efficiently implement switch tables in a compiler. We have performed the following measurements:

- The span of the switch: the numeric distance between the smallest and the largest value (inclusive).
- The actual number of cases used in the switch.
- Density: the proportion of values between the smallest and the largest switch case values actually covered by cases.

To clarify the concepts, consider the following (simplified) switch statement:

```
switch( X )
{
 case 1:
 case 3:
 case 4:
}
```

This has a *span* of four, an *actual case* count of three, and a *density* of 75 %.

The programs examined in the MARE project contained a total of 1122 switch statements.

Figure 12 shows the distribution of spans for the switch statements in the examined programs. Note the non-linear scale on the x-axis.

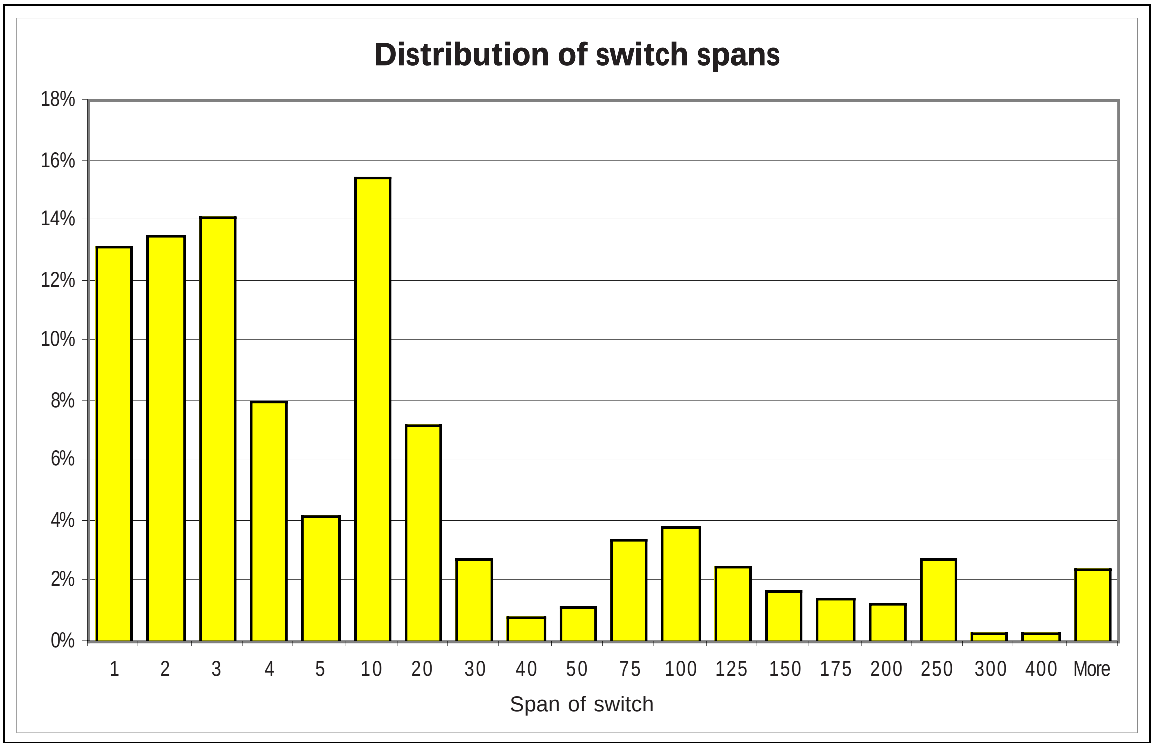

Figure 12

The large number of switches with only a single case is a symptom of automatically-generated code, where switches are in many cases used for decisions which could just as well be made using if-statements.

More than 50 % of all switches have a span less than 5, and 75 % less than 20. This indicates that most switches contain quite few cases.

Figure 13 shows the distribution of switch densities. It is clear that most switches are quite dense.

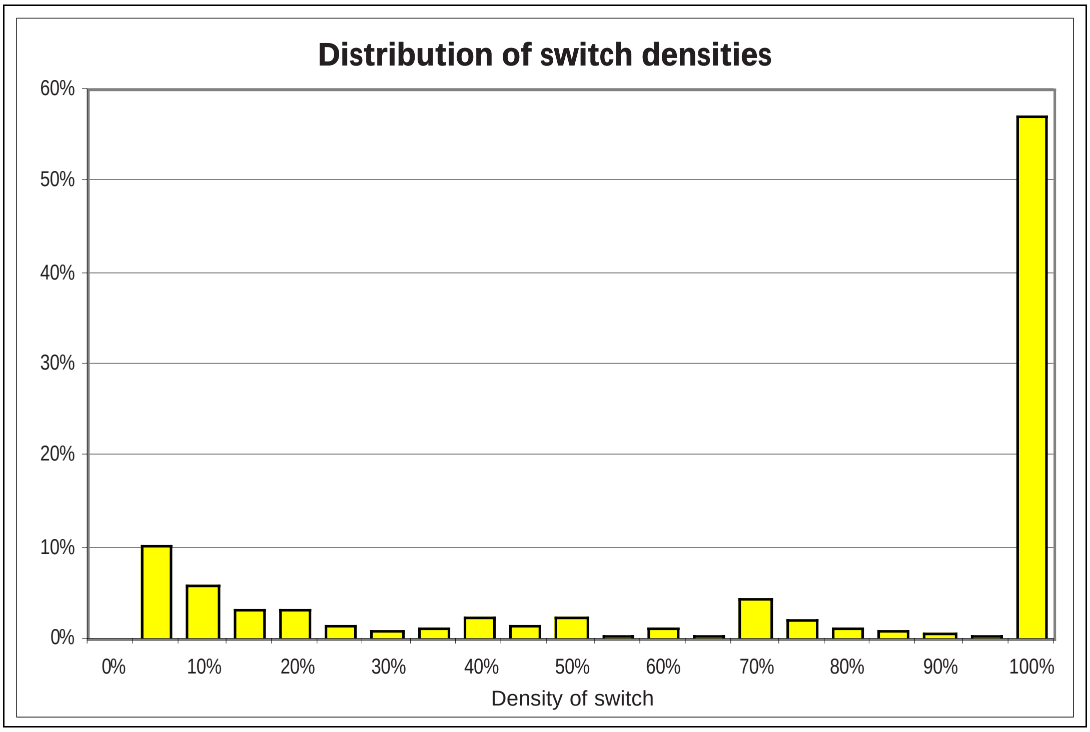

Figure 13

For code-generation purposes, the density of a switch determines which implementation is the most efficient (spacewise):

- For dense switches, a direct-indexed jump table can be used. In the analyzed programs, about 58 % of switches have a density over 90 %, which should mean that a simple jump table should be very efficient.
- For somewhat sparser switches, down to 50 %, a switch table is still efficient, and this makes up a further 23 % of the switches, for a total of 71 % of switches which can be efficiently handled by jump tables.
- For very sparse switches, the best implementation is a series of compares and jumps, or a table containing value-label pairs. About 22 % of the switches fall into this category, with a density of 20 % or less.
- For intermediate-density switches, there is no obvious implementation. The switches between 20 % and 50 % density only make up about 7 % of all switches, which means that this is quite a small problem.

A further question is whether the span and density is related in some way. The plot in Figure 14 shows a plot of span (on the logarithmic Y-axis) vs density (on the X-axis) for all the switches in the data.

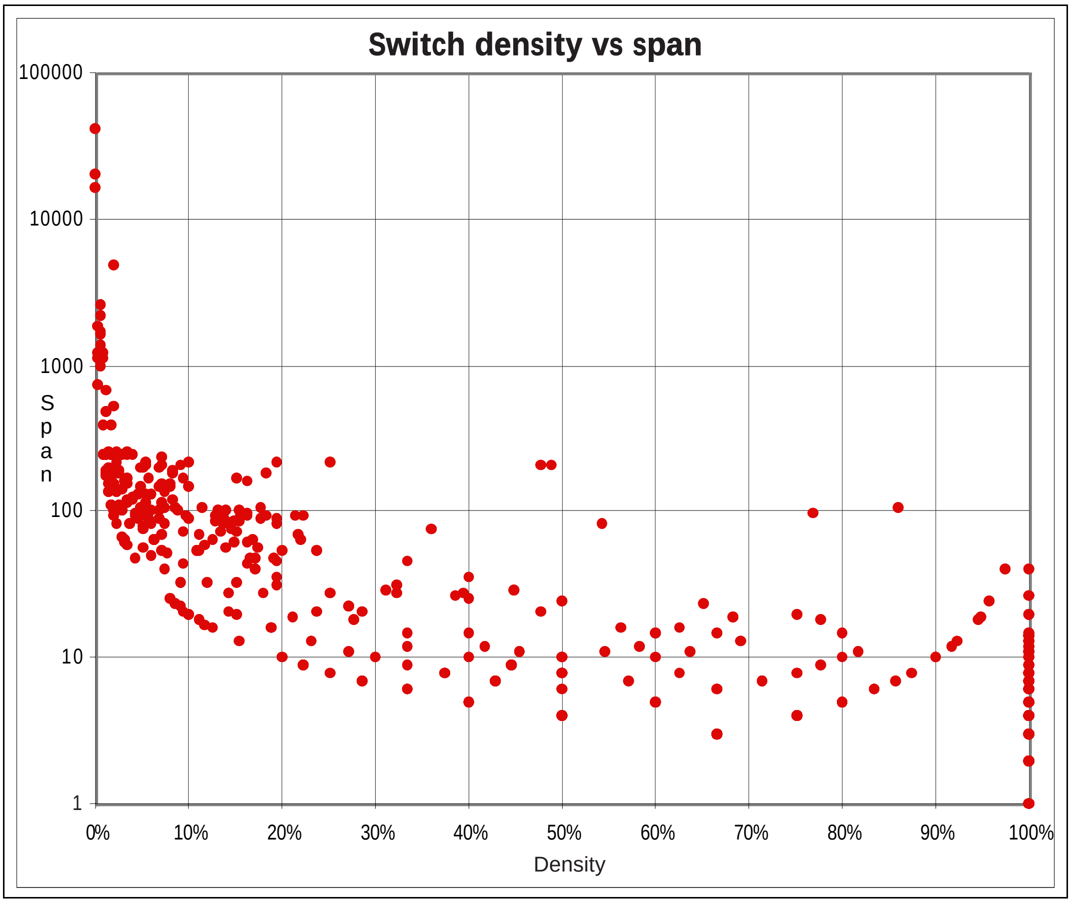

Figure 14

Some conclusions and observations:

- There are a few outliers: the points in the upper left of the diagram. The most extreme switch has a span of 40000 and only 7 actual cases. The other two have spans of 18000 (a jumble of hexadecimal constants used in an operating system) and 16384 (a bit-mask switch [`case 1`, `case 2`, `case 4`, …, `case 16384`]).
- Note that most of the switches actually belong on the 100 % axis (about 50 %). This indicates that the majority of the switches are dense and contain less than 50 cases. There are no very large and very dense switches.
- The nice smooth curve on the lower-left looks interesting, but it is only an unavoidable mathematical consequence of the relation between the plotted variables:
  - To make a switch with a low density, you need a large span. There is no way to construct a switch with a density of 10 % and a span less than 20 (the smallest is `case 1`, `case 20` – since you need at least two cases to create a span, and 2 is 10 % of 20).

- To make a really dense switch, you need at least 1/(1-*density*) cases: for a density of 90 %, you need at least a span of ten (`case 1, case 2, ..., case 10` – with one case missing). This rule does not apply to the 100 % dense switches: when all cases are covered, the span can be anything between one and eternity.

Figure 15 shows a plot of the switch density vs the number of actual cases.

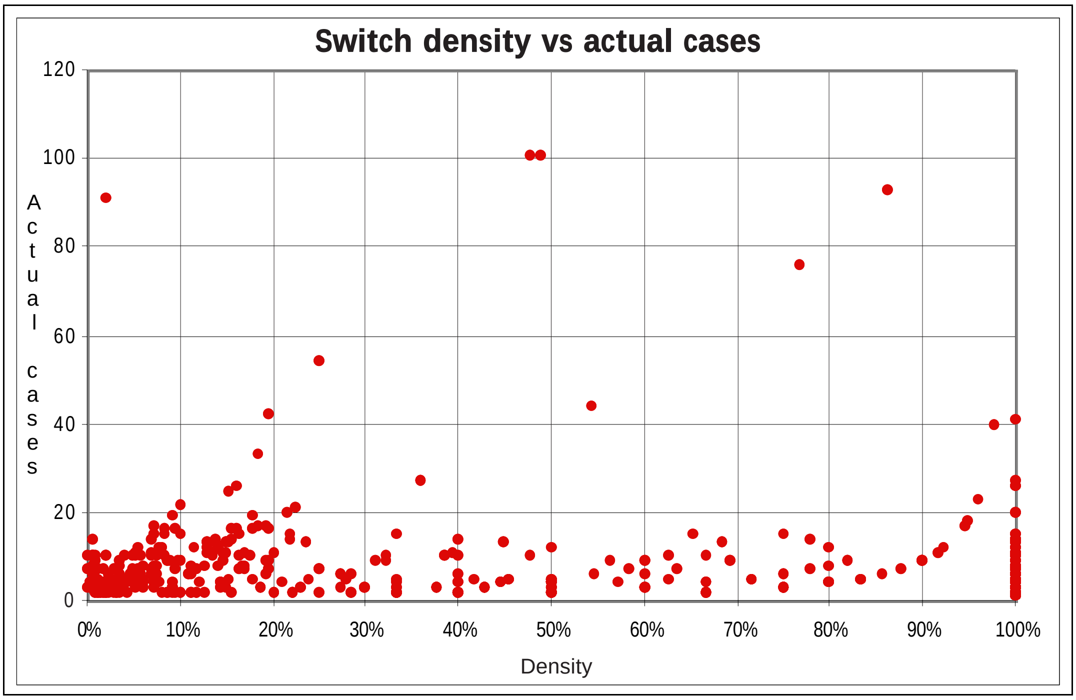

Figure 15

Some observations:

- Sparse switches in general contain few actual cases, which means that a compare-jump implementation will be very efficient.
- Most switches contain 25 cases or less. For an embedded compiler, there is little to be gained from optimizing switches with many cases.
- The most problematic switches, for code generation, are the sparse switches with many cases (between 20 % and 50 % density, which means that jump tables could get very large). There are a few of those in the data.

5. Conclusions

5.1. Information on Embedded Programs

The conventional wisdom in the embedded and real-time research community and industry is that “there is no such thing as a typical real-time/embedded program”. Even so, there are some observations we would like to make:

Embedded programming is closer to the hardware than ordinary desktop programming:

- The importance of storing data in ROM when possible makes for heavy use of `static const` variables.
- Because of performance and memory-consumption constraints, smaller data types are preferred. Memory is not cheap for embedded applications.
- The features of the microcontroller used have an impact on the code, since the microcontroller was chosen specifically for its features.

For more on embedded processors vs desktop processors, see [1].

The structure of the programs exhibit some interesting features:

- Non-terminating loops are used to create tasks. The presence of non-terminating loops in desktop programs is almost invariably a bug. In an embedded application, it is quite reasonable.
- A large number of functions are simple, which indicates that procedural abstraction is being used. Programs are not written as large spaghetti functions.
- `goto`s are not used indiscriminately, except by automatically-generated code.

Operating systems are used even on 8-bit processors – but note that the 8-bit processors used for the investigated programs are actually quite sophisticated. The Z-80, for example, has been used to implement multi-user minicomputers.

5.2. Conclusions for WCET Tools

We believe that this study has provided some very important guidance for our efforts to create a useful WCET tool. The following conclusions can be drawn:

Program structure and language features:

- Unstructured flow-graphs have to be handled. They appear both in automatically generated code and in ordinary code after optimization.
- Recursion will need to be handled, at least in its tail-recursive form. This can be postponed, but not for too long.
- Non-terminating functions require checks for termination in loop analysis methods.
- Function pointers must be handled to some extent. Probably a reasonable level of ambition is to handle function pointers with constant contents. Function pointers passed as arguments should be handled as well.
- Global variables are common and cannot be ignored.
- The information provided by language extensions (see Section 4.1) should be utilized, if possible.

Complexity of analysis:

- A majority of the examined functions contained no loops. This is interesting, since there are WCET analysis methods which are very efficient but require non-looping code [2]. Such methods could be used for simple functions, with more complex analysis methods applied only to more complex functions.
- For simple functions, it should be possible to calculate the WCET characteristics only once, and then use the result of this analysis wherever the function is called. This vision is reminiscent of Chapman’s work on the SPATS tool [3].
- The presence of very deep loops and very complex decision nests require attention to the complexity of the analysis methods.

The use of operating system, libraries available in binary only, legacy code, and automatically generated code makes the reliance on annotations in the code unrealistic. We must automate the analysis of program behavior. However, a programmer should be allowed to enter extra information to make the analysis more exact.

Incomplete programs: the conclusion with the greatest impact on the architecture of our WCET tool is the fact that we have to work with incomplete programs. This has the following implications:

- It must be possible to analyze files and functions in isolation – for example, to allow operating-system code to be analyzed by the vendor.
- It should be possible to analyze parts of a program, even though large parts may be completely unknown and not available for analysis. We cannot require access to the entire program.

- We need to make WCET analysis component-based and compositional.[9]

5.3. Summary and Future Work

The MARE project has given us more information than we originally hoped, and the results have been surprising at times.

The most important conclusion is the need for component-based WCET analysis. The second most important is that WCET analysis can be optimized by taking advantage of the fact that some parts of a program are very simple.

In the future, we plan to use the MARE tools and results for comparison to other bodies of code, for example the spec performance benchmarks, or benchmarks used to evaluate embedded compilers. It would be interesting to contrast code for small 8- and 16-bit processors to code written for large 32-bit processors. Finally, we would like to extend the base of programs used for the MARE project, in order to check if our conclusions still hold.

[9] These conclusions fit well with the ARTES (A network for Real-Time research and graduate Education in Sweden) [4] vision of the future of real-time and embedded programming as being component-based.

6. References

1. Manfred Schlett. *Trends in Embedded-Microprocessor Design*, IEEE Computer Magazine, Vol. 31, No. 8, August 1998, pp. 44-49.
2. Peter Altenbernd. *On the False Path Problem in Hard Real-Time Programs*. In Proceedings 8th Euromicro Workshop on Real Time Systems, 1996.
3. Roderick Chapman. *Worst-case timing analysis via finding longest paths in SPARK Ada basic-path graphs*. Technical Report YCS-94-246, Department of Computer Science, York University, October 1994.
4. ARTES (A network for Real-Time research and graduate Education in Sweden). *Programme Plan 1998-2002*, April 1998, page 3. Available on the web from `http://www.docs.uu.se/artes/`.