Why SpecInt95 Should Not Be Used to Benchmark Embedded Systems Tools

Jakob Engblom
Department of Computer Systems
Uppsala University
P.O. Box 325
SE-751 05 Uppsala, Sweden
+46-(0)18-471 73 44
jakob@docs.uu.se

ABSTRACT

The SpecInt95 benchmark suite is often used to evaluate the performance of programming tools, including those used for embedded systems programming. Embedded applications, however, are often targeting 8- or 16-bit processors with limited functionality, whereas SpecInt95 has no particular target architecture and a bias towards 32-bit systems. Hence, there are reasons to question the use of SpecInt95 for the evaluation of tools for embedded systems.

We present a comparative study of the static properties of a set of embedded application and the SpecInt95 benchmarks. The properties studied include: variable types, function argument lists, type of operations, and the use of local and global memory.

The study provides clear evidence that embedded applications and the SpecInt95 program suite differs significantly in several important areas. Hence, we conclude that using SpecInt95 to evaluate or compare tools for embedded systems is likely to be irrelevant or misleading, and that there is a clear need for a benchmark suite tailored for the embedded applications area.

1. INTRODUCTION

The SpecInt95 benchmarks [15] are often used to evaluate the performance and functionality of various programming and program analysis tools.

It is a common view in the embedded systems community that the SpecInt95 suite is not typical for embedded applications. However, there has been no firm evidence to support this view, and despite this gut feeling, the SpecInt95 suite (and older Spec suites) *is* being used to compare tools for embedded systems.

In this paper we present the results from a study which aims to quantify the appropriateness of the SpecInt95 benchmarks for evaluating embedded systems tools. Our method has been to measure the static properties of the SpecInt95 benchmarks, and compare the data to data collected from real embedded and real-time programs, thereby providing hard evidence on which the appropriateness of SpectInt95 can be judged.

Our study was performed by compiling programs with an instrumented compiler which gather statistics on the properties of the code.

Only *static* properties were measured, i.e. properties that can be determined by analyzing the *source code* of a program, without executing it. In contrast, *dynamic* properties are determined by observing the *execution* of a program.

Considering static properties is appropriate for evaluating analysis tools, since these tools operate on the program source code. The dynamic properties of programs are relevant when comparing the performance of code or processors, but not so much when comparing, evaluating, or designing programming tools.

One example of a programming tool affected by the static properties of the code is a compiler. The performance of a compiler is closely related to the static properties of the input: large or small functions, complex or simple data flow, etc.

Another example is worst-case execution time (WCET) analysis. WCET analysis is performed by statically analyzing the code, and both the run-time of the tool and the quality of the results are directly related to how well the algorithms employed fit the actual input.

The investigation of the SpecInt95 benchmarks and their relation to embedded programs is part of a study of embedded and real-time programs [4] with the purpose of gathering information that allows us to formulate requirements for industrial-strength worst-case execution time analysis [5].

The focus of thus study is on small embedded systems, based on eight- and sixteen-bit processors. Such processors are used in most of today's embedded systems[1].

The programming language used in this study is C, since this is the dominant programming language in the embedded systems field [14]. Note, however, that the analysis is performed on the

[1] In 1998, about 2 billion 8-bit and 1 billion 16-bit processors were sold, compared to 200 million embedded 32-bit processors, and just 100 million 32- and 64-bit CPUs for desktop computing.

intermediate code level, and hence it is relatively language independent.

Related work. We have not found any previous study dealing with the static properties of real commercial programs targeted for small embedded systems.

There has been some work on comparing the SpecInt95 benchmarks to other programs. In one such study, a group at the University of Washington compared the *dynamic* properties of Windows NT applications and the SPEC 95 benchmarks [12]. The conclusion was that the SpecInt95 benchmarks differ from the desktop programs regarding the pattern of function calls.

Statistics on the static characteristics of code in the Oberon system was collected by Lampe [11]. His study only covered the syntax of the source code, unlike our measurements of intermediate code. His conclusion was that all constructs in the language were used, thus indicating that there were no unnecessary features in the Oberon language.

This paper is organized as follows: in Section 2, we present the methodology and tools used for analysis. Section 3 contains information about the programs used in the study. In Sections 4 to 7, we compare the data for SpecInt95 and embedded systems for various categories of measurements. Section 8 discusses conclusions and gives directions for future work.

2. METHODOLOGY

The study was performed using a modified commercial C/C++-compiler from IAR Systems [6]. This provided us with a C parser suitable for handling embedded programs (including extended keywords etc.), and a source-level optimizer.

Each program (both embedded and SpecInt95) was compiled for its expected hardware platform. For the SpecInt95 benchmarks, we used settings corresponding to a generic 32-bit CPU. For the embedded programs, we used settings corresponding to the target platform for the programs.

Embedded systems compilers usually implement various extensions to the standard C language. Since each of our embedded systems programs is written for a specific compiler and a specific hardware platform, the compiler has to mimic the C variants used by several different compilers. For an example of typical features present in an embedded systems compiler, see the documentation for the IAR compiler [7].

In order to imitate a certain compiler for a certain hardware platform, the following settings in the compiler were changed:

- The size of the `int` type and the `double` type, and the various pointer types (code pointers, data pointers, pointers to different memory areas). For all embedded CPUs, `int` was set to 16 bits. Most pointers were also 16 bits. For the SpecInt95 programs, `int` and pointers were 32 bits.
- Keywords for modifying variable placement in memory (`near`, `far`, `huge`, `banked`,...), function calling conventions (`interrupt`, `trap`, `monitor`,...), and similar features used by embedded systems programmers.
- The list of *Intrinsic functions* used in the compiler. Intrinsic functions are very important for embedded systems compilers. An Intrinsic function invocation looks like a regular function call in the source code, but the generated code is a short sequence of assembly language. It is used to give C programmers access to interrupt handling and other special features of the hardware – without having to use inline assembler. Eleven of the studied embedded programs used intrinsic functions [5]. We ignore the code supposed to be generated by the intrinsic functions, but record that we have seen an intrinsic function call.

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

- Analysis at the intermediate code level allows us to analyze 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, and similar issues, since this does not affect the automatic analysis techniques (and programming tools) considered for this paper.
- The basic optimizations remove obvious clumsiness from the code: all constants have been folded, all expressions have been reordered in a canonical manner, etc. This makes it easier to compare programs written in different styles.

Furthermore, there are some factors indicating that source code analysis may give incorrect results:

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

Finally, performing the analysis after parsing and optimization makes our task simpler, since we do not have to handle program errors, and the parsing and optimization process simplifies the code. The intermediate language is much simpler than the C language (it uses a small number of well-defined operations and has an explicit control flow). However, the intermediate code is still quite far away from assembly-language, and is the same for all target processors, thus easing the task of comparing programs written for different target platforms.

Because of obvious space constraints, this paper only presents the results of measurements relevant for demonstrating differences between SpecInt95 programs and embedded real-time programs. The measurement results for the embedded programs are available as a technical report [4].

Furthermore, this study is limited to measurements that can be performed without extensive data and control flow analysis. This is both because such analyses are expensive to implement, and because the most interesting measurements require whole-program analysis, which is not possible in our present framework.

The data collected is relevant for indicating the *presence* of certain features in the studied programs. 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 embedded programs originated were telecommunications, vehicle control, and home

and consumer electronics. Unfortunately, due to non-disclosure agreements, we cannot name the companies involved, only present the results.

In the studied programs, there is a mix of timing critical code (including control loops, protocol management, coding and decoding) and non-timing critical code (e.g. user-interface and initialization code).

In total, our embedded systems programs consist of 13 different applications with a total of 334 600 lines of C source code (excluding comments). Note that the programs are written for *freestanding* C-implementations, i.e. the programs only use the libraries which do not need operating system support. The source code of the libraries are *not* part of the study.

The SpecInt95 programs used in the study were the following:

- `099.go`: a go-playing program.
- `124.m88ksim`: a Motorola 88000 CPU simulator.
- `129.compress`: a small file compression utility.
- `130.li`: a lisp interpreter.
- `132.ijpeg`: a JPEG compressor.
- `134.perl`: a PERL interpreter.
- `147.vortex`: an object-oriented database.

The total size of these programs are about 97 000 lines of source code excluding comments.

We were not able to get `126.gcc` to compile, since it required libraries not available on an embedded system, and the code was hard to push through our very picky C parser. Furthermore, including `gcc` would have skewed the results, since it is about the size of the other SpecInt95 programs combined.

4. VARIABLES

We have examined the variables *defined* in the programs, i.e. the variables actually coded into the programs. Variables that are only *declared*, like hardware I/O addresses and operating system entities, are not considered. In the embedded programs, we found approximately 17 200 variables (one variable per 20 lines of code), and in the SpecInt95 programs, about 19 700 (one variable for every five lines of code).

4.1 Variable Types and Data Allocation

We divided the variables into six categories: integer variables, floating point variables, structures and unions, arrays, data pointers, and code pointers. The frequencies of occurrence of the type categories in the SpecInt95 and embedded programs are shown in Figure 1.

The most noticeable difference is that the SpecInt95 programs use more pointers than the embedded programs, and that arrays and structures are more common in the embedded programs. Also note that code pointers are very rare, both in embedded applications and in SpecInt95. Furthermore, it is clear that both the considered SpecInt95 and embedded programs are integer programs. Almost no floating point variables are used.

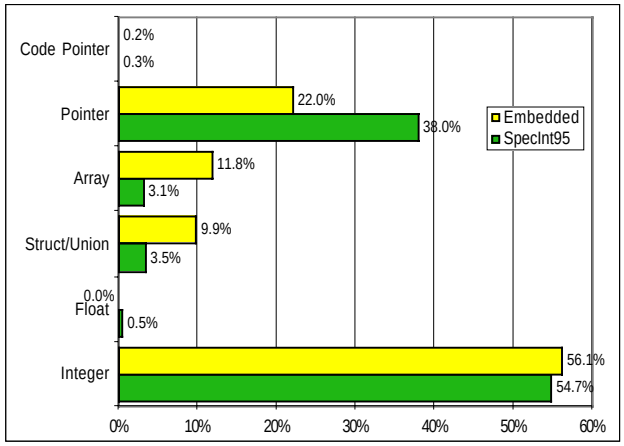

Figure 1: Distribution of variables across types

From the distribution of variable types and observations regarding the use of library functions and operating system calls, we draw the conclusion that the SpecInt95 programs use dynamic memory allocation for data, while the embedded programs allocate their data statically (as variables). This is consistent with the expectations of practitioners in the field [9].

This conclusion is based on (1) the larger number of data pointers and smaller number of `structs` and arrays in the SpecInt95 programs, and (2) that all SpecInt95 programs use the `malloc()` library call, while none of the embedded programs use `malloc()`, and only five of the embedded programs use operating-system dynamic memory services.

There are several explanations for why embedded programs use dynamic memory only sparingly. The first is the scarcity of memory in embedded systems (sometimes only a few hundred bytes of RAM are available). The second is the fact that most systems are quite static: data and code are stored in ROM, and the need for dynamically adjusting the data size is very small. Third, dynamic memory management makes the system less predictable, and predictability is usually very important for embedded systems [10].

4.2 Integer Variables

We consider the types of the integer variables in a program to be a good indicator of the size of data items the program processes. In order to investigate this, we have analyzed the types of the integer variables along the following two orthogonal axes:

- The size of the variable: `char` (8 bits), `short` (16 bits), or `long` (32 bits)[2]
- The signed or unsigned interpretation of the variables.

The results are shown in the two contrasting diagrams in Figure 2. The differences are striking:

- Embedded programs use mostly (97 %) 8-bit and 16-bit data. SpecInt95 programs use mostly 32-bit data (98 %).
- Embedded programs use a much higher proportion of unsigned variables: 87 % are unsigned, while only 47 % of the SpecInt95 variables are unsigned.

[2] We do not consider 64-bit processors, where `long` would be 64 bits in size.

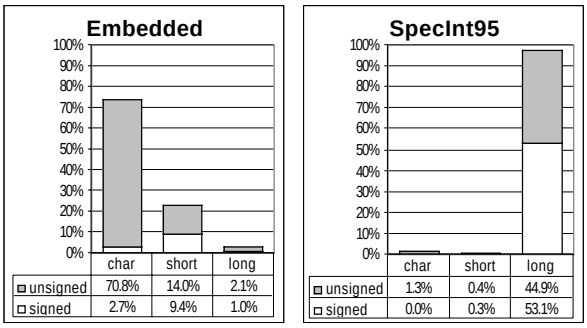

Figure 2: Contrasting integer types

The difference is to a large extent due to the different target machines: on a large 32-bit machine, an `int` maps to a machine register, and is the natural size for any numeric value. On an 8-bit embedded machine, the natural size is a byte (byte variables are declared as `char` or `unsigned char`), and only variables that do not fit in a byte are declared as `short` or `long`. Furthermore, most I/O unit interfaces use 8-bit values, which makes `char` variables appropriate for hardware interfacing.

The embedded programs rarely use the plain `int` type. Most of the code use macros designed to generate a variable of a particular size: `BYTE` for eight-bit values, `WORD` for sixteen-bit values, etc. Embedded systems programmers typically want precise control over the size of their variables.

Our conclusion is that handling small data items efficiently is much more important for embedded systems tools (and especially compilers) than for desktop systems tools.

4.3 Variable Scopes

Another contrast was found when we studied the scopes of program variables: embedded programs have a much higher proportion of global and static variables than the SpecInt95 programs. Figure 3 shows the distribution of variable scopes in detail.

Parameters are variables declared as parameters in function headers. *Auto* variables are variables local to a function (on a desktop system, they are typically allocated on the stack). *Static* variables are local variables declared with the `static` keyword, which means that they maintain their value between function calls, and that they are stored in static memory (together with the global variables)[3]. *Global* variables are declared in the global scope, outside of functions.

Figure 3 shows the statistics on the *number* of variables in each scope. A different picture emerges if the *amount of data* in each scope is considered. As shown in Figure 4, most of the data is kept in global scope, both in the embedded programs and the SpecInt95 programs. However, local data is more important in the SpecInt95 programs, which could indicate a different programming style.

Furthermore, the global data exhibits a different type signature from the local data (both for embedded and SpecInt95 programs). About 30 % of the global variables are arrays, compared to only one or two percent of the local variables. Pointers are much more common in the local scope than in the global scope. For the embedded programs, structures are six times as common in the global scope compared as in the local scope (for SpecInt95, structures are equally common locally and globally).

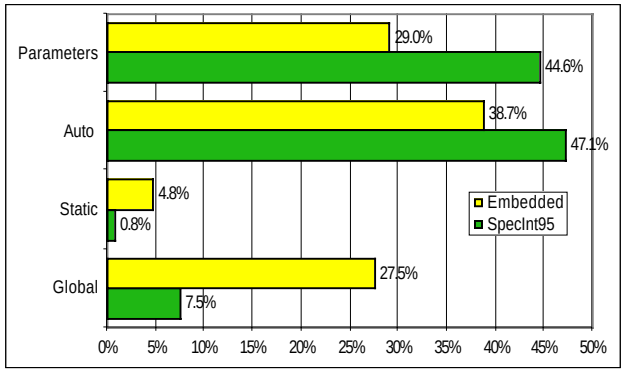

Figure 3: Distribution of variables across scopes

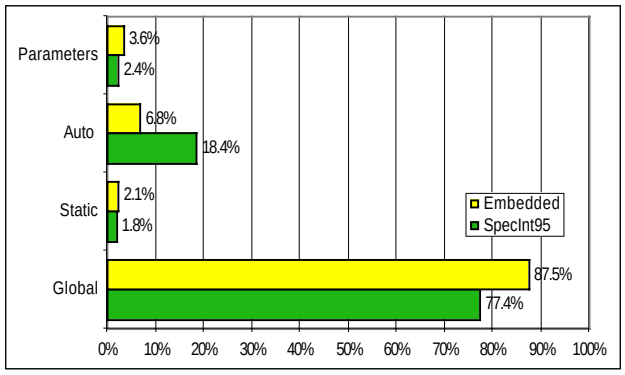

Figure 4: Distribution of data across scopes

The conclusion is that both embedded and SpecInt95 programs store large data globally. This trend is more pronounced for the embedded programs, especially for structures.

Embedded programs also use more global and static variables in general. This might be due to the fact that embedded compilers traditionally are weak in handling local data. Making a variable static or global has been more efficient than making it local, since stack handling is painful on most small machines. The compilers have not been capable of allocating a local variable statically[4], which has forced the programmers to code the `static` explicitly. Deficient tools have affected the style of programming.

5. OPERATIONS

The mix of operations used in the programs provides additional indications on how embedded programs and the SpecInt95 programs differ from each other.

[3] Variables static to a file are considered to be global variables, since they do not belong to a certain function.

[4] Allocating local variables and parameters statically is equivalent to imposing FORTRAN calling conventions on C, and requires careful analysis to avoid run-time errors. In theory, a compiler could determine when a variable does not need to be stored on the stack and allocate it in static memory, but to our knowledge, no compiler performs this optimization.

We have classified the operations present in the intermediate code of the analyzed programs into the following categories:

- *Logic:* shifts, `and`, `or`, `xor`, and bitwise negation (`not`).
- *Compares:* comparison operations used for making decisions. Equality, inequality, greater than, etc.
- *Arithmetic:* addition, subtraction, multiplication, division, modulo, and negation.
- *Pointer operations*: array indexing, access to structure members, etc.

Note that we have ignored branches and jump instructions. This is because the number of branches is highly dependent on the compiler and the target hardware: branches are accidental properties. For example, the presence of conditional moves, predicated instructions (as on the ARM [2]), or instructions to calculate with flags (as on the PowerPC [13]) drastically affects the number of branches in the code, without changing its meaning.

Also note that it is quite meaningless to reason about the number of loads and stores at the intermediate code level. The need for loads and stores is determined by the target architecture and can only be measured after code generation, which makes comparisons across hardware platforms meaningless.

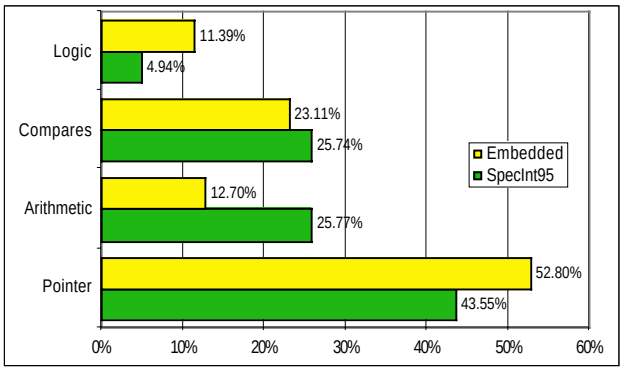

Figure 5: Distribution of operation categories

Figure 5 shows the result of the measurements. Note that pointer operations comprises about half of all operations, emphasizing how important good pointer handling is for a compiler (since many of the pointer operations can be removed during code generation). We are quite surprised to find that compares constituted a quarter of all operations, leaving only 25 % for the arithmetic and logic operations.

The most important difference between the embedded programs and the SpecInt95 programs is the use of arithmetic and logic operations. In embedded programs, logic and arithmetic operations are equally common, while the SpecInt95 programs use arithmetic operations five times as often as logic operations.

This is not surprising: much of the work in embedded systems is performed by testing, setting, and clearing bits in hardware registers.

Another contrast is that 70 % of the operations in embedded programs operate on unsigned values. Only 20 % of the SpecInt95 operations are unsigned. This is consistent with the data on the integer variable types presented in Section 4.2. Embedded programs tend to use unsigned data and operations.

The conclusion is that (to put it drastically) embedded programs manipulate bits while SpecInt95 programs calculate values, and that actual calculations and manipulations are a minority compared to operations needed to address data and make decisions.

6. FUNCTIONS

We have examined the *defined* functions in the programs, i.e. the functions whose bodies are present in the studied source code. The size of the sample was 2 713 (an average of 36 lines of code per function) functions for SpecInt95, and 5 597 functions for the embedded programs (an average of 60 lines of code per function).

6.1 Return Types

The return types of functions are distributed as shown in Figure 6. The categories are the following:

- *Void*: no return value.
- *Ptr*: data pointer.
- *Ulong*: unsigned long (32-bit) integer value.
- *Long*: signed long value.
- *Ushort*: unsigned short (16-bit) value.
- *Short*: signed short value.
- *Uchar*: unsigned char (8-bit) value.
- *Char*: signed char value.

Note that `float`, `struct`, and function pointers all had a frequency less than 0.5 %, and were left out of the graph in Figure 6.

The conclusion is that SpecInt95 functions return values twice as often as embedded functions, and that the return types are larger (mostly `long`s and pointers). The embedded functions typically do not return values (60 %), and the returned values are small, predominantly of type (`unsigned`) `char`.

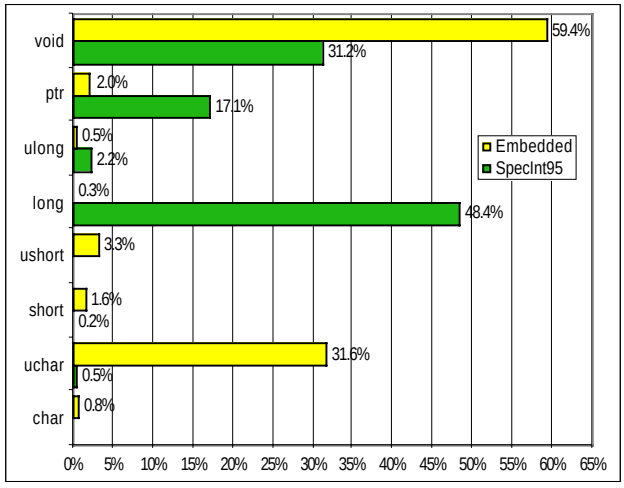

Figure 6: Distribution of return types for functions

6.2 Number of Parameters

The parameter lists of the functions were studied in order to determine the number of parameters to functions. The result is shown in Figure 7.

One striking result is that 51 % of the embedded functions take no parameters at all! Together with the result that 60 % of the embedded functions return no values, this indicates that functions with a type signature like "`void foo(void)`" are common.

Indeed, 35 % of the embedded functions are void-void. In the SpecInt95 programs, only 3.5 % of the functions are void-void.

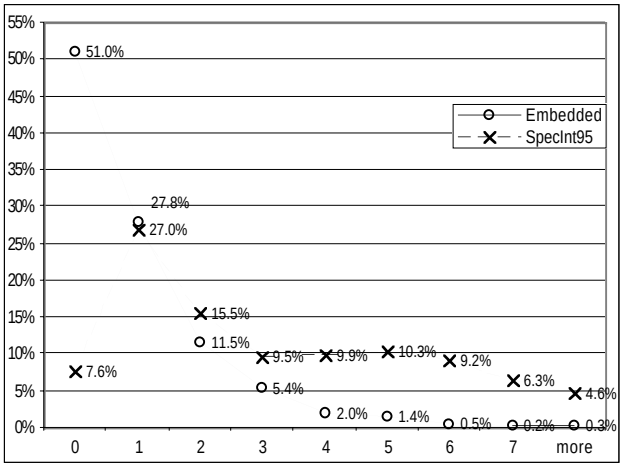

Figure 7: Relative frequency of parameter counts

This means that many functions in the embedded programs only perform side effects. We offer two explanations for this. The first is that the functions read or write global data instead of using the parameter and return value mechanisms in C. The second is that many functions simply perform some fixed task, like stopping a motor or turning a light on, which does not require parameters and does not produce any values.[5]

There are some other observations of special interest for compiler writers:

- In the embedded programs, allowing just three bytes of function parameters to be passed in registers would cover 87 % of all functions. For the SpecInt95 programs, 24 bytes, or six 32-bit registers, are needed to cover 90 % of all function parameter lists.
- The average size of a function argument is four bytes in the SpecInt95 programs. The average for embedded programs is just 1.6 bytes.

Our conclusion is that SpecInt95 programs tend to use proper functions: functions that take parameters, perform calculations, and return values, while embedded programs tend to use functions that do not return values or take parameters.

6.3 Complexity

In our original study of embedded programs [4], one of the most interesting results is that most functions have a rather simple control-flow. Four out of five functions do not contain loops, and one third of the functions were trivial (they consisted of a single basic block).

For the SpecInt95 programs, the results are different: only one sixth of the functions are trivial, while the complex functions (functions containing loops) make up one third of the total. Only two out of three functions are non-looping (compared to four out of five for the embedded programs).

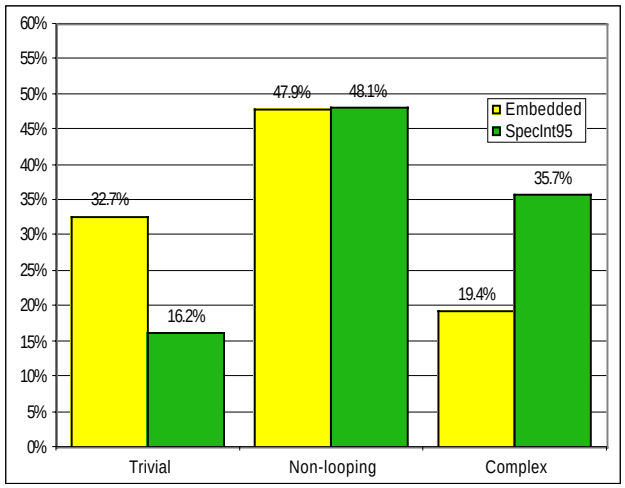

Figure 8: Distribution of function complexity

Figure 8 shows the comparison graphically. *Trivial* functions contain no decisions, *non-looping* functions contain decisions but no loops, and *complex* functions contain loops. Usually a complex function contains decisions, since the loop condition is a decision – unless the loop is non-terminating.

The conclusion is that embedded programs contain more simple functions than the SpecInt95 programs. The SpecInt95 programs appear to be slightly more complex.

7. OTHER MEASUREMENTS

The total number of measurements performed is quite large. Above, we have provided details on the most interesting differences between our set of embedded programs and the SpecInt95 programs. In this last section dealing with our measurements, we will discuss some less significant measurements.

7.1 Loops

Several measurements were performed on loops and loop nests:

- *Non-terminating loops*: there is only one non-terminating loop in the SpecInt95 suite[6] (compared to 44 in the embedded programs). This was expected, since embedded programs are usually built from tasks, where each task has a non-terminating function as its body. The SpecInt95 programs, in contrast, are usually designed to perform some calculation and then terminate.
- *Loop nesting depth*: the depth of loop nesting was slightly larger in the SpecInt95 programs. 91 % of the loops in the embedded programs are singly nested, while only 79 % are singly nested in the SpecInt95 programs. SpecInt95 loop nests are a little deeper than the embedded loop nests on average.
- *Loop complexity*: the number of basic blocks in loops and the number of extra exits (`break`-statements) were counted. The embedded programs and the SpecInt95 programs did not differ significantly.

[5] The explanations are based on experience and browsing through the source code. It is very hard to automatically analyze a function's behavior to determine whether it performs output to hardware: we cannot identify hardware registers accurately. It is also hard to determine whether global variables are used instead of parameters and return values, since this implies understanding the intent behind the code.

[6] It was the `main()` function for `m88ksim`. The CPU simulator was coded to run until the user interrupts the execution.

7.2 Decisions

A decision nest is a nest of `if` and `switch` statements. We measured the depths of the nests, as an approximate measure for the decision-making complexity of the programs. No significant difference between the embedded programs and the SpecInt95 programs could be found.

Looking specifically at the `switch` statements, we note that they are far more common in the embedded programs. Switches are present in 5 % of the decision nests in the SpecInt95 programs, and in 20 % of the decision nests in the embedded programs.

8. CONCLUSIONS AND FUTURE WORK

We have measured a large sample of commercial embedded and real-time programs, and compared the results to the same measurements performed on the SpecInt95 [15] benchmark suite. We have studied the static properties of the program code, which are more relevant than the dynamic properties for comparing and evaluating programming tools (especially program analysis tools).

Programs from SpecInt95 (and older SpecInt suites) are often used to evaluate programming tools, for both desktop systems and embedded systems. However, the comparisons in Sections 4 to 7 above indicate that the static properties of the SpecInt95 programs are quite different from those of real embedded programs.

Our conclusions are the following:

- It is dangerous to assume that comparisons between and evaluations of programming tools based on SpecInt95 programs give results relevant for embedded systems.
- There is a need for benchmarks that are more representative for embedded systems.

Embedded systems benchmarks would need to address the characteristics of embedded programs. In this paper, we have presented a number of points where embedded programs differ from desktop programs:

- The size of variables: embedded systems use variables tightly tailored to fit the data manipulated. `Char` variables are frequent, larger data types are used only when absolutely needed.
- Unsigned data dominate over signed data.
- Logical operations are more common (especiallly in relation to ordinary arithmetic).
- Many functions perform only side effects.
- Global data is used more frequently, both for variables (data which is changed at runtime) and large constant data[7].
- Direct interfacing to hardware is very common.
- Dynamic memory allocation is rarely used.
- Only the parts of the C libraries requiring no operating system support are used. The most notable omission is file handling.

In the future, we expect to use the information from this study to guide our research on worst-case execution time tools (and other programming tools for embedded systems), as detailed in [5].

We also plan to continue measuring other sets of programs, and comparing the results with our results for embedded programs and SpecInt95 programs.

The new EEMBC benchmark suite [3] is an attempt to provide the embedded systems world with an equivalent of the desktop systems' Spec benchmarks. Unfortunately, these benchmarks were released too late to be incorporated into the round of measurements presented here, but we plan measure them when the source code becomes available to us.[8]

It will be interesting to see how well the EEMBC benchmarks reflect our characterization of embedded systems. However, our guess is that we really need several sets of benchmarks depending on the capabilities of the target system. It seems very hard to construct a benchmark that can reasonably be used both on an 8051 [8] (no stack, small memory, horrible instruction set) and on an ARM [2] (fully modern RISC, large memory, good stack handling).

ACKNOWLEDGEMENTS

I would like to thank the following people for their help: *Thomas Lundqvist* at Chalmers, for helping me measure the SpecInt95 programs and for his ideas on what to measure. *Andreas Ermedahl* at DoCS for discussing the results and patiently listening to my explanations of each exciting new graph to come out of the printer. My advisors *Hans Hansson* and *Bengt Jonsson* for commenting and proofreading this paper. *Sang Min* and *Mikael Sjödin* at DoCS and *David Whalley* at Florida State University for commenting on drafts of this paper. *Carl von Platen*, *Anders Berg*, *Dan Hammerlid*, and *Mats Kindahl* at IAR Systems for their opinions on what would be interesting to measure. *IAR Systems* for letting us use their compilers as research tools.

This work has been performed with support from the Advanced Software TEChnology Center (ASTEC) [1], the Swedish National Board for Industrial and Technical Development (NUTEK), and IAR Systems [6].

REFERENCES

[1] Homepage for ASTEC: `http://www.docs.uu.se/astec`.

[2] *ARM7 Data Sheet*, ARM DDI 0020C, Advanced RISC Machines Ltd, December 1994.

[3] Homepage for the *EDN Embedded Microprocessor Benchmarking Consortium*: `http://www.eembc.org`.

[4] 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/Reports/tr_index.shtml`.

[7] The observation that many global variables are actually constants is based on experience and reading the code of our embedded programs.

[8] Unfortunately, the EEMBC consortium cannot provide a research license for their code, so we have to find a sponsor to pay the license fees they require.

[5] J. Engblom, “Static Properties of Commercial Embedded Real-Time Programs, and Their Implications for Worst-Case Execution Time Analysis”, To be published in *Proceedings of the Fifth IEEE Real-Time Technology and Applications Symposium (RTAS ‘99)*, IEEE Computer Society Press, June 1999.

[6] Homepage for *IAR Systems*: `http://www.iar.com`.

[7] Information about the IAR Systems 68HC11 compiler: `www.iar.com/download/ew6811.pdf`.

[8] *MCS 51 Microcontroller Family User’s Manual*, Order No. 272383-002, Intel Corporation, February 1994.

[9] N. Jones: “Efficient C code for eight-bit MCUs”, *Embedded Systems Programming Europe*, Miller Freeman Ltd, London, February 1999, pp. 18–30.

[10] D. Lafreniere, “An efficient dynamic storage allocator”, *Embedded Systems Programming Europe*, Miller Freeman Ltd, London, November 1998, pp. 34–42.

[11] J. Lampe: “Statistics about Modules of the Oberon System”, *Software-Concepts and Tools*, (1997) 18, Springer Verlag, pp. 27-34.

[12] D. C. Lee, P. J. Crowley, J-L. Baer, T. E. Anderson, and B. N. Bershad: “Execution Characteristics of Desktop Applications on Windows NT”. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, published as *ACM SIGARCH Computer Architecture News*, Vol. 26, No. 3 (June 1998), Pages 27-38.

[13] *PowerPC Microprocessor Family: The Programming Environment For 32-Bit Microprocessors*, Rev 1. Order No. MPCFPE32B/AD, Motorola Inc, January 19997.

[14] 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`.

[15] Homepage for SPEC (The Standard Performance Evaluation Corporation): `http://www.spec.org`.