In: Proceedings of the he 6th International Conference on Real-Time Computing Systems and Applications RTCSA'99, December 1999,

Making Worst Case Execution Time Analysis for Hard Real-Time Tasks on State of the Art Processors Feasible*

Stefan M. Petters Georg Färber
Institute for Real-Time Computer Systems
Prof. Dr.-Ing. Georg Färber
Technische Universität München, Germany
{**Stefan.Petters,Georg.Faerber**}**@rcs.ei.tum.de**

Abstract

In the development of embedded hard real-time systems the ability to guarantee worst case execution times is gaining importance and complexity. The fast evolving processor acceleration techniques constantly increase the gap between a processor which uses the accelerations and one which does not. Modeling cache, pipelines and other parts of the processor gets increasingly difficult and time consuming. To circumvent this problem especially in the domain of rapid prototyping of embedded hard real-time systems, we propose to lay more weight on measurement and less on modeling. By analyzing the control flow graph the compiler uses for optimization, a reduced control flow graph can be generated, which limits the paths to be measured. Using this information the object code of the program is being instrumented and then measured. By measuring all paths the reduced control flow graph indicates, predictability is achieved without using too pessimistic estimations.

1 Introduction

State of the art processors are equipped with various techniques to accelerate the execution. Main focus is to close the gap between processor speed and main memory: cache memory has been added first externally, now on chip and in several levels. Additional pipelines, multiple execution units, speculative execution and special features are build in new processors. The Pentium II even mimics externally a CISC machine but internally has a RISC core. The resulting RISC code is reordered in the CPU, so reliable prediction seems out of reach. The evolution of processors is driven by hard competition between the manufacturers, leading to the fact that the internals of a new CPU core are kept secret. On the other hand the modeling of such processors is ever more complex and is unsupported by the manufacturer. Due to this the modeling of a new processor takes longer while the evolution cycles of the processors get shorter. Many an application area is still closed to software real-time systems because the dependable prediction of uptodate processors is still impossible.

To avoid this problem we propose in this paper to take measurements instead of trying to model the processor to the very detail. The compiler has been modified to additionally emit the control flow graph (CFG) it has used for optimization. This CFG can be reduced by removing paths which are definitely known not to produce the worst case execution time (WCET). If the complexity of the resulting reduced CFG is still too large, the CFG is split into two or more parts to allow separate analysis of these. This leads to a larger overestimation of the WCET to be accepted to gain a feasible analysis. In the following stage the code is instrumented and executed. During the execution the execution time is measured. The instrumentation and measurement are repeated till all paths of the the CFG have been executed. Finally the influence of preemption is taken into account by an analysis of the timing specification of the environment.

The paper is organized as follows: The next section surveys related work, which is followed by a short description of the environment we are working with. In Section 4 our approach is presented in detail. A summary of the results is given before we indicate future work in the last section.

2 Related Work

Puschner and Schedl describe in [14] an approach where the control flow graph is analyzed to gain knowledge of the worst case execution path of a program. In contrast to our work, the execution time of basic blocks is assumed to be

*The work presented in this paper is supported by the *Deutsche Forschungsgemeinschaft* as part of a research program on "Rapid Prototyping of Embedded Systems with Hard Time Constraints" under Grant Fa 109/11-2.

known. A linear programming approach is taken to compute the worst case execution path and therefore the WCET.

Park and Shaw are assuming a very simple computation model in [12]. A cache-less processor is used and in the C subset under investigation no float and long word operations are allowed. Additionally the tasks run on a bare machine without operating system. Annotations are required to bound the maximal and minimal number of loop iterations. The work focuses more on the prediction of assembler code based on the C source than on the analysis itself.

In [9] Li, Malik and Wolfe use linear programming methods to determine the WCET. Each line of code is assigned a counter which holds the execution count of this line. Various (boolean and additive) equations are set up to describe the conditions which trigger the execution of code. This linear equation system is then solved. For small systems this works well, but the state space can grow vastly.

Formal proof methods are used by Chapman, Burns and Wellings in [1]. The program is run by symbolic execution. In the beginning all variables are undefined. As the program is executed, the range of values of a variable is reduced whenever possible. If a branching is reached, the state of the program is dumped on a stack-like structure. First one branch and then the other branch is executed. The number of loop iterations and behavior of subroutines in certain modes is described in annotations. If the state of the program during execution guarantees one and only one of the given modes, the analysis of this subroutine is simplified. This approach suffers on an exploding state space as well, but if a formal proof has to be done the additional effort is justifiable.

White, Müller and Harmon have modified the compiler to emit additional control flow information very similar to our technique in [16]. They model the cache and the processor very detailed. With 512 bytes the cache used in this paper is not very large. During array accesses the spatial locality is detected by the tool. It recognizes that certain array elements are in the same cache line and is therefore able to predict the presence of a corresponding amount of array elements in the cache. Temporal locality which is caused by repeated references of the same variable is also exploited. The problem is also the complexity of the method. The work of Ferdinand and Wilhelm in [4] bases on the previous approach. By using program restructuring methods to increase data locality and data dependency analysis the worst case bounds on cache misses are determined. As this method fits only restricted classes of programs it is complemented by persistence analysis.

The analysis in [15] is limited to straight-line code. This is code which contains no loops, procedure calls and only static variables. According to the authors Stappert and Altenbernd such code is produced by code generators. Analog to our approach the CFG is output of the compiler. The analysis is divided in longest execution path search, caching and pipeline analysis. The absence of loops simplifies significantly the analysis of the longest execution paths and the caching effects.

Ermedahl and Gustafson describe in [3] a method to automatically derive annotations on the number of loop iterations and mutual exclusive paths by analyzing the C source. Very similar to [1] a kind of symbolic execution is used, but in this case the goal is to derive annotations. To handle the complexity of the problem, the possible values of variables are abstracted if the discrete values are too *expensive*. Like in our approach fully optimized code is used. A potential combination with our tool is referenced in section 4.1.

Ermedahl et al. combine their approaches in [2]. A high level analysis searches for mutual exclusive paths and the number of loop iterations. The low level architecture dependent part is limited to analyzing caching and pipelining single basic blocks which conform to [15]. The analyzed programs were 10 to 80 lines long.

Lee et al. cover a side effect of caches in preemptive task systems in [8]. The restauration of the cache after a task has been preempted and is now reentering its code costs time which is not covered in traditional schedulability analysis. To bound the time for this tighter to the real one two techniques are presented. First the number of useful blocks is computed. This is the set of memory blocks needed by the program for further execution. The computation is done by a linear programming approach. Second a phasing of tasks is introduced to eliminated infeasible task interactions. In Section 4.4 a simpler method is proposed.

An approach which bases solely on the analysis of high level language is described in [10] by Liu and Gomez. To each procedure of the program under investigation a timing function is added, which returns additional to the *original* return value the time that was consumed by the function executed. The input parameters are limited to a known set of values. Combined with symbolic execution, a closer bound is gained. The measurement of execution is related to our approach.

3 Design environment

3.1 Target Architecture

Our target architecture REAR (cf. Figure 1) was designed to support rapid prototyping of real-time systems. The basis for this is the task classification model presented in [6], where each type of real-time task corresponds to a best suited processor type, in terms of performance and deterministic execution times. It is a configurable and scalable heterogeneous multiprocessor system consisting of standard off-the-shelf components, which are tightly coupled by a global PCI-Bus. The processing units' basic difference are

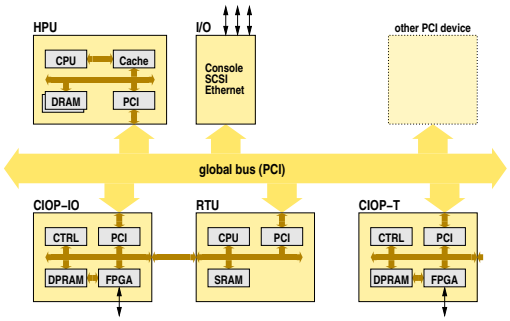

Figure 1. REAR Hardware Architecture

the means to achieve predictability in spite of interrupts and context switches:

The High Performance Unit HPU and Real-Time Unit RTU are both processor-boards connected to the global PCI-Bus. The HPU is based on a dual Pentium II machine and the RTU has a MIPS R4600 CPU.

Of more relevance for this paper are the Configurable I/O-Processors CIOPs. They each consist of one Xilinx FPGA and additional dual ported RAM and act as separate application specific processing units for tasks with deadlines too short to be met in software and provide a flexible way of linking the prototyping architecture to the embedding system. An additional CIOP is used exclusively for time measurements (see Section 4.3).

A more detailed description of REAR is given in [5].

3.2 Design Flow

The starting point is a formal specification of the embedded system in the Specification and Description Language SDL [7]. The timing requirements imposed by the embedding process are captured in annotations in SDL. All stimulating external events – i.e. events coming from the environment – and stimulating internal events, i.e. all timers which trigger the start of a task are annotated as additional timing information which describes the maximum number of events of a certain kind in a given interval.

Code generators produce both C code for the software part and VHDL for the hardware part. The software is then compiled, analyzed and instrumented to allow the recording of traces. These are used to determine the WCET (cf. Sec. 4). The WCET for the HW-part is predetermined during synthesis and can be read out from the synthesis tool.

From the SDL specification a real-time analysis model (RTAM) is generated using structural and behavioral information of the SDL specification. Together with the WCETs, deadlines and timing specification, a schedulability analysis is performed.

The design flow and it's interactions are described in greater detail in [13].

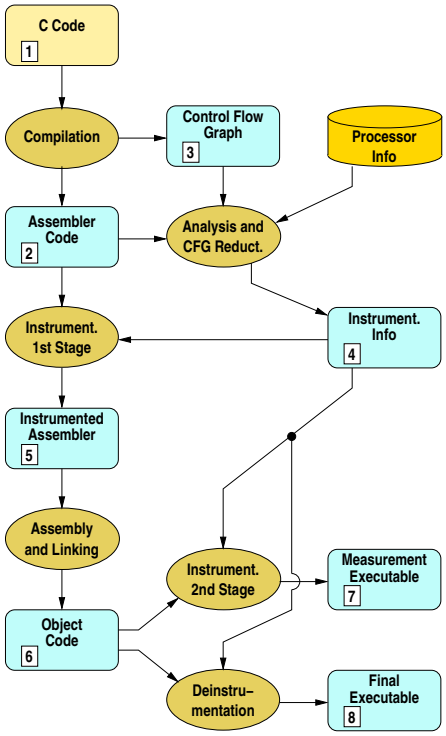

Figure 2. Tool Chain

4 Measurement approach

For the real-time analysis the WCET of tasks has to be known. As the complexity of measuring all possible paths inside yields to huge number of measurements, as intermediate representation measurement blocks are used. The finest granulation is a basic block. The CFG is generated by the compiler. After an analysis of the CFG it can be reduced i.e. paths known not to yield the WCET are removed. Information how the code is to be instrumented is extracted. The code is then instrumented and precise time stamps are taken. This step is repeated, till all paths defined before have been measured. The impact of preemption and interactions with the real-time operating system is covered in Section 4.4.

4.1 Control Flow Graph Analysis

Generation of the CFG As depicted in Fig. 2 the control flow graph [3] is automatically generated by the compiler. The compiler internally uses the CFG to optimize the code. In order to allow small changes to be made to the compiler itself, we use the gcc. This CFG reflects the state of the code after all optimizations have been made. Especially the fact that loop unrolling is done by the compiler to accelerate execution simplifies the later analysis. Additional debugging information allows an easy mapping of source code to assembler code [2]. This enables the integration of

an algorithm to identify mutual exclusive paths like the one presented in [3].

Infeasible and Non WCET Paths In a first step the CFG is cleaned of false paths. This is mainly due to two origins. The compiler adds always a path from a function call node to the exit node. This is to cover `assert` and `exit` statements. These paths are neglected since if this statement is reached in a hard real-time task, something seriously wrong has happened. An assert statement doesn't prolong the execution time of a correct program. Another source for false paths can be dead code, which should produce a warning either by the compiler or by our tool.

Non WCET paths are paths which yield the WCET under no circumstances. A simple example are bypass paths of `if` statements without a corresponding `else` statement. These paths can be skipped for the later measurement. Other cases have to handled more carefully. Even if a block is known to consume more time in the best case BCET than the alternative block in the worst case[1], the impact of the execution on consecutive blocks cannot be ignored. To skip the shorter path for measurement the gap between the BCET of the measured block and the WCET of the skipped should be given a reasonable size.

Limiting Complexity The reduced CFG is analyzed to search for the paths which have to be investigated by measurement. For large real-time applications the number of paths to measure is exceeding a manageable size.

To avoid that the measuring of the application takes several days, the application is broken into measurement blocks. A measurement block is a piece of code, which is measured isolated of history effects i.e. at the starting of a measurement block all caches and execution units are flushed. This introduces overhead but speeds up the process of design considerably. This partitioning has to be done by hand.

Loop Handling Figure 3 shows a control flow graph with a loop and two alternative paths inside this loop. If the loop is executed n times, the number c of combinations the alternatives are executed is

$$c = 2^n$$

The example shown in Figure 3 can be resolved in two ways. The CFG can either be measured as a whole or it has to be divide into measurement blocks. Which of both is done depends on the number of loop iterations. If this is small, a measurement in whole will be preferred in most

[1]The BCET and WCET of an assembler instruction have to be a priori known bounds which enclose the execution time of this instruction in all cases

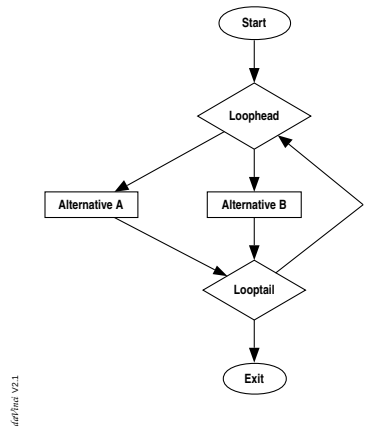

Figure 3. CFG of loop with alternatives

cases, while for larger numbers of iterations most likely the splitting will be used. For small loop bodies a third kind of analysis is possible. The loop can be instrumented to produce only a time stamp for each iteration without flushing the cache. The measurement can be accelerated, if the execution of the alternative A and B is forced in the following way:

$$(n * A), B, ((n-1) * A), (2 * B)...A, ((n-1) * B)$$

where $(n*A)$ stands for n-times the execution of alternative A in a row. This way only $n * (n+1)$ iterations of the loop are necessary. The WCET of the loop is the maximum of n consecutive loop iterations. Such a manipulation of measurement has to be done by hand.

A special case is represented in data invariant loops. These loops have a data invariant loop count and the alternatives inside the loop are not dependent on the type of data it uses. Thus such loops can be considered as one block.

Instrumentation Information The information of how the program is to be instrumented i.e. beginning and ending of super blocks and which paths have to be analyzed is put into a database [4]. The usage is described in the following section.

Figure 4 shows three stages in the reduction of the CFG. The first stage is the original CFG. The second shows only the paths for the measurement while the third displays the resulting measurement blocks.

4.2 Instrumentation and Deinstrumentation

The instrumentation (cf. Fig. 2) is done in several stages. The first step is to place procedure calls `sttrace(`*`id`*`)` in the assembler code. To force the number of loop iterations

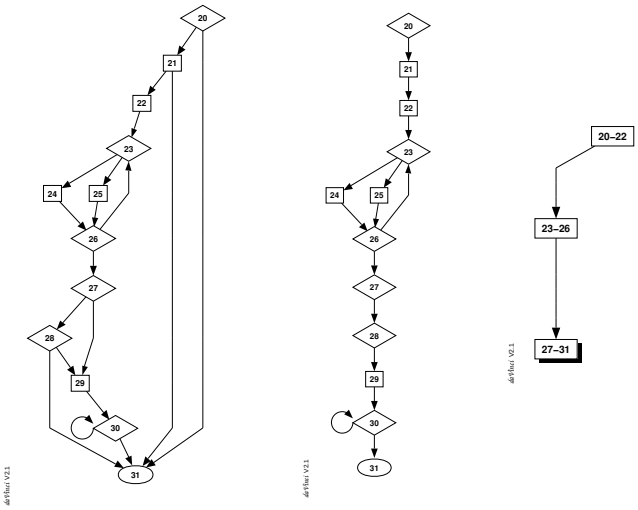

Figure 4. Example of a CFG reduction

additional code has to be placed at the tail of a loop. The `sttrace(`*`id`*`)` procedure is tagged with a unique *id* to correlate the time stamps with the code. The core of the procedure `sttrace(`*`id`*`)` is given for a Pentium II processor.

```
rdtsc                ; read timestamp
                     ; counter in eax
sall    $16,%eax     ; make room for id
movzx   8(%ebp),%edx ; get id from stack
orl     %edx,%eax    ; put word together
movl    PCI_ADR,%edx ; get address
movw    %eax,(%edx)  ; write tag
wbinvd               ; write back and
                     ; flush caches
rdtsc                ; same part as
sall    $16,%eax     ; above
movzx   8(%ebp),%edx
orl     %edx,%eax
movl    PCI_ADR,%edx
movw    %eax,(%edx)
```

The first time stamp record is made to mark the end of the measurement of the preceding block and the second is made to mark the beginning of the measurement of the following block (cf. Section 4.3). Each measurement block is analyzed separately but to speed up the measurement the tests are concatenated.

To get dependable WCETs the cache and all execution units are flushed. The cache is flushed on the Intel Pentium II by a single assembler instruction `wbinvd` which writes back the cache to main memory and then invalidates all cache levels. For the MIPS processor a little loop has to be inserted. The time for this operation is not constant on both processors. Therefore it is needed to take two time stamps before and after the flush.

By using this approach its possible to guarantee a WCET without knowing anything of the history the program has taken. If the history has a strong influence on the WCET of a certain code piece the result will be an overestimation of the WCET. To avoid the problem of accelerating the code by flushing the cache, for every instrumentation point an additive penalty has to be added to the WCET, which corresponds to the amount of time possibly gained by the cache flush.

After the assembler and linker have generated the object code 6 on the basis of the instrumented assembler code 5, the code is manipulated to execute the path prescribed in the instrumentation database 4. To force this behavior conditional jumps are substituted by `nop`s or by `jmp`. At the tail of loops additional code is added to force the number of loop iterations. To examine all paths several cycles of the 2nd stage of instrumentation and measurement have to be made. If all measurements are done the instrumented object code 6 is deinstrumented i.e. all calls to `sttrace(`*`id`*`)` and the instrumentation code at the loop tail are substituted by `nop`s. This ensures that the memory locations of code and data are exactly the same in the final executable as in the measurement executable. Thus we don't have to add additional penalties for uncertain alignment of code and data on cache line boundaries.

4.3 Measurement

To avoid preemption during the measurement, all triggering ISRs which can lead to a preemption by another task are instrumented to return immediately without starting a new task. The impact of preemption is handled in Section 4.4.

As mentioned in Section 4.2 recording of precise time stamps is needed. To achieve this additional hardware is used. The current approach of using a dedicated configurable I/O processor (CIOP-T) for timing measurement was driven by several requirements:

- High resolution (in the order of the processor cycle time),
- wide disambiguity range (several minutes to allow tracing long running applications),
- minimal overhead and interference with the system under investigation.

The first two items lead to the use of 64 bit trace records consisting of a 16 bit measurement block identifier, a 16 bit high resolution time stamp (usually from the processor's internal cycle counter [2]) and a 32 bit low resolution (system

[2]Most modern high performance microprocessors like the Alpha 21x64, the MIPS R4600 and the Intel Pentium support an internal 32 or 64 bit cycle counter.

global) time stamp from a counter implemented in the "timing CIOP". Overhead and interference were minimized by moving the tasks of combining and buffering the 64 bit trace records to the CIOP: The procedure `sttrace(id)` simply combines the 16 bit *id* and 16 bit from the high resolution time stamp counter and writes this 32 bit word to a dedicated CIOP register. [3] The write access triggers the CIOP to complete the trace record with the 32 bit low resolution time stamp and to store it in the trace record FIFO, which is implemented using the CIOP's dual ported RAM. For short test runs, this FIFO can hold all trace records and they may be read and stored or evaluated after the applications end, while large amounts of trace records will be read and stored during the applications run either by a low priority thread on one of the system's processing units, or on a dedicated processing unit, connected via the CIOP's I/O ports.

After the measurement is finished, the measured times are analyzed. The time stamp pairs of ending and starting of each measurement block are subtracted and compared to the previous found WCET for this block. If the new measured time is larger than it is assigned to the WCET of this measurement block.

The behavior of the system under investigation can only be controlled to some extend. For example the memory refresh cycles of the DRAM force the processor to wait eventually. To cover these effects an additional safety factor has to be added to the WCET of a block.

As a last step, the tasks reduced CFG is searched for the path with the largest WCET which is assigned as the WCET of the task. The measurement method is not depending on operating system or processor family. So it can be used for tasks either on the RTU or on the HPU.

4.4 Interactions with the RTOS

The operating system has to be instrumented as well. This is needed to keep track of system calls and interruptions of the running process i.e. preemption and interrupt service routines ISR.

The system calls have their own WCET each which is taken instead of the measured one. A detection of the start and the end of system calls is necessary to correct the measured time accordingly. This can either be achieved by wrapping system calls or by instrumenting the OS itself. This instrumentation is either with cache flush for blocking system calls and without for the non blocking ones.

Interruptions of the task can be seperated into two categories. One category is that of periodic interrupts of the RTOS e.g. clock ticks or periodic rescheduling. These will be the same in the running system. So their influence is already included in the measured time. A small penalty is added to the time to catch the jitter of these events and eventually additional events, which impact the task because of preemption.

The other category is that of sporadic interrupts and preemption by a task of higher priority. A sporadic interrupt has to be caught and the preemption avoided during measurement to keep the influence small on the measured time. An off line analysis of all preempting events is already covered in the timing specification (cf. Section 3.2). Each procedure is assigned the amount of memory it can potentially hold i.e. that could be lost by a cache flush. This is done hierarchically. If a procedure calls more than one procedure, only the one with the largest amount of memory is taken into account. Global variables are assumed to be in the cache and are therefore lost. If the amount of memory which can be potentially lost exceeds the cache size, the maximum cache size is taken. The resulting maximum memory flushed from cache is assumed to result in cache misses and is added as corresponding preemption penalty to the measured time for each preemption possible during runtime.

In real world environments the preemptions which can happen to a single task are assumed to be limited. If this is not true, the overestimation introduced by this technique increases, but it's still safe to execute the prototype. So it can be expected that for most applications the usage of cache exceeds by far the penalty introduced by preemption. This method of handling preemption requires that inside the tasks no dynamic memory allocation takes place. This restriction in combination with the absence of `goto`s and recursions is commonplace in real-time programming and produces no loss of generality.

5 Case Studies

To test our approach and examine the behavior of our target architecture we have made measurements on example applications. In a first test, we applied the measurement technique to a small application, which fits into the first-level cache of the Pentium II processor operating at 233 MHz. The second processor was switched off during the measurement. The application consisted of a loop, with two `sttrace` calls in the loop-body. Without cache flushes the measurement was reproducible to the clock cycle. As we introduced the cache flushes, the measurements showed statistic effects. The execution times deviated between 648 and 767 cycles. These effects have their source in uncontrollable system behavior (statistic execution interference SEI) like e.g. the memory refresh cycles of the DRAM.

As a larger application we have implemented an FFT-Algorithm which computes the FFT on 4096 complex data elements. Since the number of loops and the access of the

[3] The single word access avoids expensive explicit mutual exclusion when writing the CIOP register.

	Input-data	Max. cycles	Min. cycles	Diff.
1	0 vector	333,771	327,827	5,944
	1 vector	334,883	328,306	6,577
	sinus	339,458	331,942	7,516
	noise	332,430	325,510	6,920
	extremes	339,458	325,510	13,948
2	0 vector	2,783,799	2,780,739	3,060
	1 vector	2,786,938	2,783,550	3,388
	sinus	2,783,400	2,781,297	2,103
	noise	2,776,855	2,771,535	5,320
	extremes	2,786,938	2,771,535	15,403

Table 1. Results of FFT Measurement

data inside the loops are invariant, it was possible to instrument the loops as one block before the loop-heads and after the loop tails. Thus the FFT was divided into two parts. The results in Table 1 show that statistic influence of the SEI decrease with the increase of the runtime of the application. To test the influence of data on execution time of data invariant paths different kinds of input data have been investigated.

As can easily be seen, the deviation in execution times introduced by the type of data is limited to a few percent and doesn't increase in absolute values for larger application parts. This deviation is dependent on the application.

To get a grip on the overhead introduced by the flushing of caches inside the application, we have tested a filter application. The CFG of this application is given in Figure 5.

The filter application filters a vector of 4096 double values. The execution of alternatives in loops and the number of loop iterations were data invariant.

Blocks	Max. cycles	Min. cycles
1-4	155,436	155,051
5-8	141,709	125,384
5,9-12	148,795	125,880
5,9,13,14	254,983	254,913
15-19	116,476	116,188
20-31	203,826	203,680
Worst Case Sum	730,721	
One-run	597,085	596,749

Table 2. Measurements of Example Filter Application

The data in Table 2 shows the overestimation introduced by the cache flushes and the code of the instrumentation.

A hamming bit correction algorithm was used to examine this effect on a very small application. The algorithm was separated into parity generation and bit correction. In

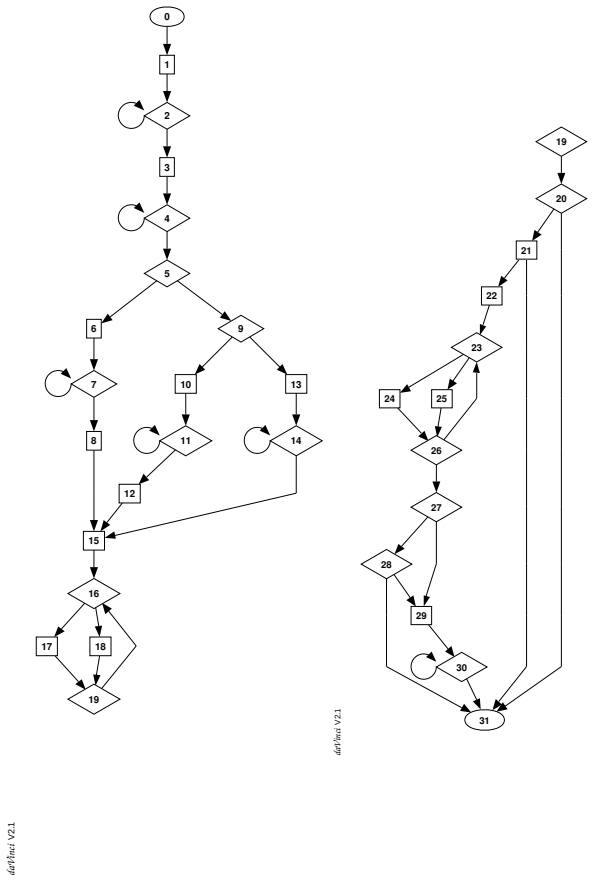

Figure 5. CFG of Example Filter Application

this case the execution time was nearly doubled by the instrumentation between these parts:

- 1989 cycles with additional instrumentation
- 1086 cycles without additional instrumentation

This case opens a view to the possible analysis of small loops with data dependencies.

6 Conclusion and Future Work

In this paper we have presented an approach for a feasible worst case execution time estimation on state of the art processors as integral part of a design environment. For the analysis of fully optimized code the control flow graph generated and used by the compiler is utilized. The analysis of automatically generated code has shown that a severe reduction of the control flow graph is often possible. We presented a method for automated instrumentation and measurement and have shown how the complexity the problem can be reduced by partitioning of the control flow graph.

The impact of interaction with the real-time operating system is covered in our approach.

As next steps in the near future the degree of automation of the methods described will be increased. This includes the automated reduction of the control flow graph and the automation of the second stage of instrumentation. As mentioned before the search for mutual exclusive paths and automatically generated annotations on the number of loop iterations will increase the accuracy of the approach significantly. Also detection of loops with a fixed number of iterations is desirable. Additional test applications of real world programs with more demanding complexity have to be investigated.

References

[1] R. Chapman, A. Burns, and A. Wellings. Integrated program proof and worst–case timing of SPARC Ada. In *Proceedings of the ACM SIGPLAN Language, Compiler, and Tool Support for Real-Time Systems (LCTS) workshop*, Orlando, Florida, June 1994. ACM Press.

[2] J. Engblohm, P. Altenbernd, and A. Ermedahl. Facilitating worst–case execution time analysis for optimized code. In *Proceedings of the 10th Euromicro Workshop on Real-Time Systems*, Berlin, Germany, June 1997.

[3] A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. In *Euro-Par'97, 20th Workshop on Real-Time Systems and Constraints*, Passau, Germany, August 1997.

[4] Christian Ferdinand and Reinhard Wilhelm. On predicting data cache behavior for real–time systems. In Müller et al. [11], pages 16–30.

[5] Franz Fischer, Thomas Kolloch, Annette Muth, and Georg Färber. A configurable target architecture for rapid prototyping high performance control systems. In Hamid R. Arabnia et al., editors, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, volume 3, pages 1382–1390, Las Vegas, Nevada, USA, June 30 – July 3 1997.

[6] Georg Färber, Franz Fischer, Thomas Kolloch, and Annette Muth. Improving processor utilization with a task classification model based application specific hard real–time architecture. In *Proceedings of the 1997 International Workshop on Real–Time Computing Systems and Applications (RTCSA'97)*, Academia Sinica, Taipei, Taiwan, ROC, October 27–29 1997.

[7] ITU–T. *ITU–T Recommendation Z.100: CCITT Specification and Description Language (SDL)*, June 1994.

[8] C.–G. Lee, J. Hahn, Y.–M. Seo, S.L. Min, R. Ha, S. Hong, C.Y. Park, M. Lee, and C.S. Kim. Bounding cache-related preemption delay for real–time systems. In *18th IEEE Real–Time Systems Symposium*, San Francisco USA, December 3–5 1997. IEEE, IEEE Computer Society Press.

[9] Y.–T.S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, pages 456–461. ACM, June 1995.

[10] Y. A. Liu and G. Gomez. Automatic accurate time–bound analysis for high–level languages. In Müller et al. [11], pages 31–40.

[11] Frank Müller, Azer Bestravros, et al., editors. *Languages, Compilers and Tools for Embedded Systems*, Lecture Notes in Computer Science, Montreal Canada, June 19–20 1998. ACM SIGPlAN, Springer–Verlag.

[12] C.Y. Park and A.P. Shaw. Experiments with a program timing tool based on source–level timing schema. *IEEE Transactions on Computers*, 24(5):48–57, May 1991.

[13] Stefan Petters, Annette Muth, Thomas Kolloch, Thomas Hopfner, Franz Fischer, and Georg Färber. The REAR framework for emulation and analysis of embedded hard real–time systems. In *Proceedings of the 10th IEEE International Workshop on Rapid Systems Prototyping (RSP'99)*, pages 100–107, Clearwater, Florida, USA, June 16–18 1999. IEEE Computer Society Press.

[14] P. Puschner and A. v. Schedl. Computing maximum task execution times — a graph–based approach. *Journal of Realtime Systems*, pages 67–91, July 1995.

[15] F. Stappert and P. Altenbernd. Complete worst–case execution time analysis of straight–line hard real–time programs. Technical Report 27/97, C–Lab, Fürstenallee 11, Paderborn, Germany, December 1997.

[16] R. White, F. Mueller, C. Healy, D. Whalley, and M. G. Harmon. Timing analysis of data caches and set–associative caches. In *3rd IEEE Real–Time Technology and Applications Symposium*, Montreal Canada, June 9–11 1997. IEEE, IEEE Computer Society Press.