An Efficient Time Annotation Technique in Abstract RTOS Simulations for Multiprocessor Task Migration

Henning Zabel and Wolfgang Müller

Abstract Complex control oriented embedded systems with hard real-time constraints require real-time operation system (RTOS) for predictable timing behavior. To support the evaluation of different scheduling strategies and task priorities, we use an abstract RTOS model based on SystemC. In this article, we present an annotation method for time estimation that supports flexible simulation and validation of real-time-constraints for task migration between different target processors without loss of simulation performance and less memory overhead.

1 Introduction

Complex control oriented embedded systems with hard real-time constraints require real-time operation system (RTOS) for predictable timing behavior [1]. Different scheduling strategies are applied and evaluated to guarantee deadlines for a given task set. If accurate execution times for tasks are known, a schedulability analysis can validate if the selected trategy leads to feasible schedules for a given task set.

Interrupts complicate the predictability of deadlines as they do not rely on the RTOS scheduling decisions. Accurate timing analysis in consideration of interrupts are currently executed by means of instruction set simulators (ISS), which implement a complete model of the target processor including I/Os, interrupts, pipelines and memories. The use of ISS requires the embedded software to be fully implemented and is therefore only applicable in late development phases. Schedulability tests and response time analysis helps to evaluate different scheduling strategies and task priorities in early design phases. For this, timing information for execution times of atomic blocks of a task has to be available. Those timing information can be achieved by worst case execution time analysis (WCET) or empirical studies.

Henning Zabel · Wolfgang Müller
University Paderborn, e-mail: henning, wolfgang@c-lab.de

In combination with an abstract RTOS model library, SystemC allows functional simulations of task scheduling with timing. This approach has proven to be adequate for early HW/SW co-design decisions and delivers good approximations for timing analysis with small errors compared to complex instruction set simulations and gives a simulation speedup up to 1000x [2].

In our approach, we simulate a given task set based on our abstract SystemC RTOS model. For schedulability and interrupt analysis, tasks are divided into atomic blocks and each block is annotated by its execution time on a specific target processor. To validate real-time constraints for task migration in multi processor environments, the annotated execution times have to be flexibly adapted to the target processor for each migration. For analysis of the execution time and implementation of the annotation it has some advantages to annotate the start and not the end of an atomic block, which is explained later. Thus, the execution time of an atomic block is simulated at the beginning of the next atomic block. Typically, a block can have different predecessors and thus the previous simulated block must be identified to simulate the correct execution time. This can be realized by:
(i) hard-coded switch-statements, which are very efficient but do not support task migration and flexible adaptation and
(ii) a look-up table, where the column index identifies the previous atomic block and the row the actual atomic block. These tables are of quadratic size in the number of atomic blocks.
Our approach uses processor-specific look-up tables to store execution times. For each target processor one look-up table, which is of linear size in the number of atomic blocks, is generated. Task migration can be simply performed by the exchange of tables. We compare our approach with the two above mentioned solutions for time annotation. Our evaluation shows, that our approach compares to the fast simulation of alternative solutions but due to the linear table size our tables support more complex applications and are easily applicable to different processor platforms.

The paper is structured as follows. Section 2 gives an overview about existing RTOS models. In Section 3 we present our main approach, which is evaluated in Section 4. The article closes with a conclusion in Section 5.

2 Related Work

Today, the functional analysis of embedded SW is mostly executed on an Instruction Set Simulator (ISS). ISS simulations can give accurate timing analysis for a specific target processor if the software code is already available. However, such simulations are considerably slow and thus can have only limited use for early design stages. Early design steps typically apply a static Worst case Execution Time (WCET) analysis [3]. WCET analysis takes the static program in higher-level programming language or machine code and typically extracts graph representations, e.g., control and/or data flow graphs, for worst case runtime estimation computation.

For this approach we use a representation similar to T-Graphs [7] as input. Currently available professional WCET tools for static analysis like aiT (AbsInt) and SymTA/S (SymtaVision) support static WCET execution and response time analysis. Advanced processor behavior like pipelining, caching, and branch prediction are considered. ISS based simulation is usually very accurate since it executes the real SW on a virtual platform. However, it also comes with a very slow execution (i.e., 0,5-500MHz) so that no detailed evaluations and analysis like the evaluation of different scheduling strategies can be efficiently performed. Due to this drawback several research groups have developed abstract canonical RTOS models implemented in SpecC and SystemC give simulation speeds 500-1000 faster than the comparable ISS execution [2, 5, 4, 6]. Whereas those models lacked precision in the beginning, most recent reports indicate an accurate simulation and a well coverage of task and interrupt scheduling behavior with a fast simulation speed at the same time.

Gerstlauer et al. present a methodology based on SpecC/C in [2] for transaction level based refinement. They introduce a canonical abstract RTOS model for scheduling analysis of tasks which covers basic operations for process state transitions, context switching, and semaphores. Tasks are annotated by additional control statements. Synchronization between the scheduler and tasks and between tasks is realized by events. Their approach covers task and interrupt scheduling.

Huss and Klaus present a similar RTOS model in SystemC [5]. They introduce a scheduler class with basic RTOS functions where individual schedulers can be inherited from. Their model covers task scheduling but lacks interrupt management.

Posadas et al. [6] considers tasks divided into different basic blocks. A separate time manager monitors interrupts and segment execution times where non-predictable and predictable (i.e., timer and timeouts) are distinguished. The simulation is based on a implementation of the POSIX API in SystemC. They report an 8% worst case deviation with respect to ISS. This work estimates execution times during simulation by replacing C++ operators, which comes at costs of longer simulation times.

We have developed a canonical SystemC library based on the concepts of [2] for simulation at PV-T (programmers view with timing) transaction level which also overcomes the drawback of non-preemptive tasks for accurate interrupt management including nested and prioritized interrupts. In contrast to other works, our approach comes with separated management for tasks and interrupts to support the analysis of different interrupt and task scheduling strategies.

All those approaches are based on the insertion of timing estimation information of the target platform. Timing information, which defines the consumed CPU time of a particular SW block, is typically directly inserted into the SW code by back annotation. In this article, we present an approach to include timing information into SystemC by means of a table. Thus, we can easily exchange the timing information without the need of recompilation of the complete model. We introduce an approach with lookup tables of size 2*n where n is the number of annotated atomic blocks. Our experimental results demonstrate that our lookup tables are a flexible approach and have no impact on the simulation time.

3 Automated Runtime Estimation

RTOS analysis focuses on the time points when a specific basic block is executed. This supports tracing of simulation results with respect to their execution time. Especially for timing analysis in combination with interrupts, this is of great help. For analysis of execution times, the code is separated into atomic blocks and annotated by its execution time. By means of our SystemC RTOS model we can simulate tasks and allocate them to different virtual CPUs to analyze their timing behavior. To define these blocks the designer marks individual locations in the source code. The automated runtime estimation is then performed in two phases: (1) the execution time from one mark to the next mark is evaluated by disassembling the firmware for target processor and (2) the source code is back-annotated by extending marks with time labels of the estimated times. This code can be compiled for simulation on common PC which allows a runtime estimation of the software with high performance. We present an annotation technique based on a small sized table with for fast simulation. The possibility to exchange those tables during runtime supports the efficient simulation of task migration.

3.1 Code Estimation

For dividing the source code of functions into atomic blocks, the designer marks specific points in the source code by, e.g., special C-macros like *Mark_CC*(). To keep changes to the source code as small as possible we use assembler labels to mark specific points in the source. Therefore C-macros are mapped to assembler labels for cost estimation. Labels are used to mark entry points followed by some lines of code.

```
short rc = 0;
if (a > b)
{
  rc = b;
  if (a > 0)
  {
    rc = -b;
    MARK_CC(A);
  }
  MARK_CC(B);
}
MARK_CC(end)
```

```
short rc = 0;
if (a > b)
{
  MARK_CC(B);
  rc = b;
  if (a > 0)
  {
    MARK_CC(A);
    rc = -b;
  }
}
MARK_CC(end);
```

Fig. 1 Different Annotations: end of an if-statements is marked with a label (left) and beginning of each if-statement is marked with a label (right)

There are two possibilities to mark blocks by labels, like depicted Figure 1:

(1) the end of a block is marked. Because the marks are replaced by back annotations later on, this location refers to the annotation of a atomic block after its execution.

(2)the beginning of each branch of a conditional control flow is marked. Therefore the end of a block is implicit defined at the start of the next one. Here the previous mark has to be considered to annotate an execution time.

In the first approach, it is most likely that the optimization made by the compiler will remove label "A", because the entry-point of "A" and "B" are the same. This eliminates the separation between the two if-statements so that the mark becomes useless for timing estimation. As the second approach does not have this problem, we apply marking at the beginning of branches.

To give an example, we evaluate the cost estimation for the Atmel AT90CAN128 RISC processor. The CPU uses a 2 stage pipeline with no cache. The above mentioned marks have to be added into each function of the program. If the source code is not available, the estimation tool can also follow calls to subroutines. However, subroutines must have no loops. When the function has conditional branches with different exertion times, the estimated costs can be inaccurate. For estimation the labels of marks are mapped to assembler labels like shown in Figure 2 by the example of the Euclidean algorithm.

```
short euklid (short a,short b)
{
  MARK_CC(euklid_start);
  while (b != 0){
    MARK_CC(euklid_loop);
    short h = mod (a,b);
    a = b;
    b = h;
  }
  MARK_CC(euklid_end);
  return a;
}
```

```
000000e2 <euklid>:
  e2: cf 93        push r28
  e4: df 93        push r29
  e6: 9c 01        movw r18, r24
  e8: eb 01        movw r28, r22

000000ea <euklid_start>:
  ea: 67 2b        or r22, r23
  ec: 11 f4        brne .+4       ; 0xf2
  ee: 09 c0        rjmp
.+18      ; 0x102
  f0: ec 01        movw r28, r24

000000f2 <euklid_loop>:
  f2: be 01        movw r22, r28
  f4: c9 01        movw r24, r18
  f6: 0e 94 51 00  call 0xa2 <mod>
  fa: 9e 01        movw r18, r28
  fc: 00 97        sbiw r24, 0x00 ; 0
  fe: c1 f7        brne .-16      ; 0xf0
 100: 9e 01        movw r18, r28

00000102 <euklid_end>:
 102: c9 01        movw r24, r18
 104: df 91        pop r29
 106: cf 91        pop r28
 108: 08 95        ret
```

Fig. 2 Euclidean algorithm in C and the corresponding assembler for the Atmel AT90CAN128 processor

The compiler links these labels to unique addresses in program memory space. Thereafter, an estimation tool can generate a time graph $G = (V, E)$ from those addresses. Each address defines a node $N \in V$ in the graph and an edge $e \in E$ denotes the costs, i.e., estimated execution times. For the Euclidean example nodes are N_0="euklid", N_1="euklid_start" (start), N_2="euklid_loop" (loop), N_3="euklid_end" (end) and N_4="ret" where the latter is given by the return-instruction. We can now

compute the cost $C_{i,j}$ from node N_i to N_j by disassembling the firmware and evaluating all possible execution paths through the control flow by depth-first search (DFS). The DFS assigns values for each pair of $N_i, N_j \in \{1, .., \#N\}$ and to the return-instruction. The DFS terminates when a label in $N \setminus \{N_j\}$ is reached. The case $i = j$ is important for loop estimation. If at least one direct path from N_i to N_j is detected an edge is added to the graph with the estimated worst case execution time. For the above example, this estimation leads to a graph like it is shown in Figure 3.

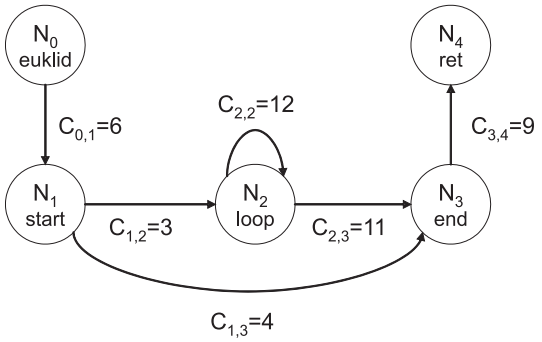

Fig. 3 Annotation Graph: Nodes are identified by annotated marks and edges denote costs given in numbers of CPU cycles.

When there is more than one path between two nodes in the assembler, costs can be an interval and the annotation is only an upper bound.

3.2 Back-annotation

For simulation the marks in the source code are replaced by function calls to simulate the execution time of the previous block. The time consumption is simulated by means of the SystemC. For annotating node N_i all edges leading to N_i must be onsidered. The fastest and most obvious way doing this, is to implement the graph by a C++ **switch-statement**, which store the successor-predecessor relationships of nodes by their index. Alternatively, a **look-up table** of size $(\#N)^2$ can be used to store the cost of $e_{i,j}$. Our approach uses a table presentation with a direct access through the node index and a reduced table size of $2 \cdot \#N$. The key idea in our table representation is to assign relative input costs C_i^{IN} and static output costs of C_i^{OUT} to each node N_i like depicted in Figure 4.

When leaving mark N_i a variable $\_cc\_cost$ is initialized with output cost C_i^{OUT} and when reaching mark N_j the relative cost C_j^{IN} is added to $\_cc\_cost$ and consumed by a function call. Additionally, it has to be ensured, that $C_i^{OUT} + C_j^{IN}$ is exactly the cost $C_{i,j}$ of the edge from N_i to N_j. Figure 4 gives a possible solu-

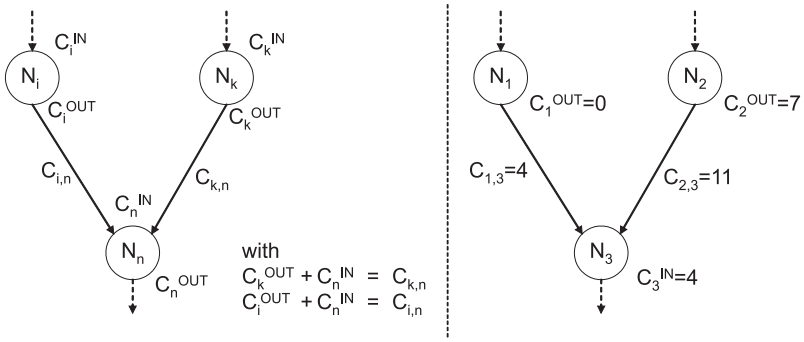

Fig. 4 general graph with divided input and output costs (left), timing graph for the Euclidean algorithm example with nodes N_1-N_3

tion for N_1-N_3 of the Euclidean example. The equations $C_i^{OUT} + C_j^{IN} = C_{i,j}$ forms a system of linear equations and can be solved by the Gauss algorithm as follows. Let $A \in M(\#E, 2 * \#N, \mathcal{N})$ be a matrix ($\#N$ number of nodes and $\#E$ number of edges in the time graph) and $b \in \mathcal{N}^{\#E}$ a vector with $\forall e_l = (N_i, N_j) \in E, l \in \{1..\#E\} : A_{l,2*i+1} = 1, A_{l,2*j} = 1, b_l =$ cost from N_i to N_j. Then solving $Ax = b$ with $x = (C_1^{IN}, C_1^{OUT}, .., C_{\#N}^{IN}, C_{\#N}^{OUT}) \in \mathcal{N}^{\#N}$ delivers the wanted values for the in- and out-cost, if and only if $Ax = b$ is solvable. Since we only need one solution it is not important if this solution is unique or not. If the linear equations are unsolvable, this annotation technique is not applicable at the moment.

The result x is finally stored in an integer-array with exactly $2 \cdot \#N$ elements. The relative incoming cost C_i^{IN} of N_i is stored at index $2 \cdot i$ and the static outgoing cost is stored at index $2 \cdot i + 1$. Therefore the table look-up can be implemented easily without complex access function to retrieve the corresponding costs for node N_i.

4 Evaluation

We evaluated our approach by four examples for the Atmel AT90CAN128 processor. The examples are at first simulated with the AVR-Studio 4.12 from ATMEL to get reference values t_{ref} for the execution times in CPU cycles. Thereafter, we analyzed the binaries with our execution time estimation tool and annotated the source code as mentioned above with a table with the solutions of our linear equations, namely "table $2N$" next. We finally compared simulation speed of "table $2N$" with hard-coded switch-statements and the table with $(\#N)^2$ entries and direct access, i.e., "table N^2".

Our estimation tool generates header files for each annotation, which redefines the previous introduced marks. Thus, the annotation can be performed without changing the original source code. The annotated code is compiled on a standard PC (with Core2 Duo 6600) and simulated to achieve the required cycles t_{sim} and the

simulation speed. The following condition should be true: $|t_{ref} - t_{sim}| < \varepsilon$ with a very small ε. In our example, ε was always zero.

Our benchmarks implemented the following examples:

primf implements a factorization of 4 byte unsigned integer into its primal factors. To avoid the addition of optimized assembler code for 32bit integer operation to the firmware by avr-compiler, we use our own implementations to evaluate the quotient and remainder.

sort implements an array sorting with recursion and nested loops. The array is sorted by quick-sort and then a copy of this array is sorted by bubble sort. In the end these arrays are compared in order to validate both results.

chk implements a small checksum check with bit-wise operations. The checksum is combined by an evaluation of a linear function.

fib implements the computation of a Fibonacci number as a final example for recursion.

All functions are invoked once for cost estimation and 10^6 times during simulation. Figure 5 shows the simulation results with not optimized code and Figure 6 with optimized code. The optimization corresponds to the compilation of the firmware, for simulation the embedded software is always compiled with optimization.

	Cycles	Switch		Array N^2		Array $2N$	
primf	310421(*)	33.5us	2692	32.17us	3608	34.4us	2916
sort	288130	19.78us	3243	20.01us	3259	22.4us	2991
chk	30007	12.54us	1898	14.77us	2220	12.5us	2068
fib	233815	30.56us	1474	27.56us	1598	32.3us	1566

(*) 308111 for Array $2N$, since some edges are estimated via two pathes

Fig. 5 Evaluation results for non-optimized firmware (-O0)

	Cycles	Switch		Array N^2		Array $2N$	
primf	76682	33.46us	2676	31.45us	3452	32.6us	2740
sort	97888	21.11us	3243	19.42us	3259	22.5us	2991
chk	12655	12.43us	1896	14.78us	2168	12.2us	2060
fib	119373	30.55us	1474	31.54us	1598	32.2us	1566

Fig. 6 Evaluation results for optimized firmware (-O2)

The tables show the estimated cycles, measured execution times and the code size of the simulation. The estimated cycles are almost the same for all solution and match the reference values from the AVR-Studio. Only for *primf* the value for the array $2N$ differs by about 0.7%. The execution times for the simulation remain similar for the different annotations. This demonstrates that our approach can be applied without loss of performance. The smaller object size for Array $2N$ (compared

to Array N^2) are due to the smaller table size with our approach. This is mainly due to the smaller data sections, which includes the tables. The advantage of the table approach is that the implementation of the marks for simulation is independent from the generation of the lookup-tables, means the accessed data. When using switch statements, as mentioned above, implementation and annotation is one part. Additionally, the table can be replaced during runtime to simulate a task migration without the loose of performance.

We also measured the execution speed of the AVR-Studio with 220 thousand instructions per second. Our backannotated simulation was executed with around $1 - 10 \cdot 10^9$ instructions per second, which finally is a speed-up of more than 4000x. Here, simulation times for optimized and unoptimized code were almost the same, because during simulation only the cost for the edges changes, but not the simulated code (except some constant optimization made by the compiler). Because the optimized code for the AT90CAN128 is considerably smaller than the unoptimized one, that leads to significant differences in performance during an instruction set simulation.

5 Conclusion

In this article, we present a method for time estimation and back-annotation based on an abstract RTOS in SystemC. It supports the flexible simulation and validation of real-time-constraints for task migration between different target processors without loss of simulation performance and less memory overhead.

For our approach we use prepared source code as input, which contains marks at the beginning of each branch. For timing estimation the marks are mapped to assembler and we evaluate a timing graph, where the edges denote the cost from one mark to another. We separate the cost of each edge as static output and relative input cost for each node (mark) by solving a system of linear equations. The solution is stored as an array of size $2 \cdot n$, where n is the number of nodes (marks). Then, the back-annotation can be efficiently implemented by table lookups, since the table-indices are static at compile time.

We demonstrated our approach by four examples for the Atmel AT90CA128 processor. At first, our back-annotated simulations deliver the same cycle counts like simulations with the AVR-Studio. At second, this annotation approach achieves similar simulation performance in comparison to hard-coded switch statements and uncompressed tables, but needs less space and allows easy simulation of task migration by replacing the tables.

Acknowledgments

The work described herein is partly funded by the DFG through the Sonderforschungsbereich 614, the German Ministry for Education and Research (BMBF) through the ITEA2 project TIMMO (01IS07002), and by the EU through COCONUT (FP7-ICT-3217069).

References

1. Giorgio C. Buttazzo and Giorgio Buttanzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.
2. A. Gerstlauer, H. Yu, and D. Gajski. Rtos modeling for system level design. In *Proceedings of Design, Automation and Test in Europe, March 2003.*, 2003.
3. Kopetz H. *Real-Time Systems. Design Principles for Distributed Embedded Applications*. Springer Verlag, Dordrecht, Netherlands, 1997.
4. M. AbdElSalam Hassan, Keishi Sakanushi, Yoshinori Takeuchi, and Masaharu Imai. Rtk-spec tron: A simulation model of an itron based rtos kernel in systemc. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 554–559, Washington, DC, USA, 2005. IEEE Computer Society.
5. Sorin A. Huss and Stephan Klaus. Assessment of real-time operating systems characteristics in embedded systems design b systemc models of rtos services. In *DVCon 07: Design and Verification Conference and Exhibitation*, San Jose, CA, 2007.
6. Hector Posadas, Jesús Ádamez, Pablo Sánchez, Eugenio Villar, and Francisco Blasco. Posix modeling in systemc. In *ASP-DAC '06: Proceedings of the 2006 conference on Asia South Pacific design automation*, pages 485–490, New York, NY, USA, 2006. ACM Press.
7. Peter P. Puschner and Anton V. Schedl. Computing maximum task execution times - a graph-based approach. *Real-Time Systems*, 13(1):67–91, 1997.