Preprints.org

Article

Not peer-reviewed version

An Efficient Reduction of Timer Interrupts for Model Checking of Embedded Assembly Programs

Satoshi Yamane *, Taro Kiriyama *, Yajun Wu

Posted Date: 6 October 2023

doi: 10.20944/preprints202310.0337.v1

Keywords: model checking; embedded assembly program; reduction of interrupt handler executions

Preprints.org is a free multidiscipline platform providing preprint service that is dedicated to making early versions of research outputs permanently available and citable. Preprints posted at Preprints.org appear in Web of Science, Crossref, Google Scholar, Scilit, Europe PMC.

Copyright: This is an open access article distributed under the Creative Commons Attribution License which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 October 2023 doi:10.20944/preprints202310.0337.v1

Disclaimer/Publisher's Note: The statements, opinions, and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions, or products referred to in the content.

Article

An Efficient Reduction of Timer Interrupts for Model Checking of Embedded Assembly Programs

Satoshi Yamane †,‡,*, **Taro Kriyama** † **and Yajun Wu** †

Kanazawa University; smt.liberation2@gmail.com (T.K.); kwu@csl.ec.t.kanazawa-u.ac.jp (Y.W.)
* Correspondence: syamane@is.t.kanazawa-u.ac.jp; Tel.:+81.76.234.4856
† Current address: Kakumamachi, Kanazawa city.
‡ The author contributed to this work.

Abstract: In verifying programs for embedded systems, it is essential to reduce the verification time because state explosion may occur in model checking. One solution is to reduce the number of interrupt handler execution. In particular, when periodic interrupts such as timer interrupts are incorporated, it is necessary to know the physical time. In this paper, we define a control flow automata (CFA) that can handle time and propose an algorithm based on interrupt handler execution reduction (IHER). The proposed method reduces the number of interrupt executions, including timer interrupts. A case study verifies the effectiveness of this algorithm.

Keywords: model checking; embedded assembly program; reduction of interrupt handler executions

1. Introduction

Embedded software is widely used in cell phones, TVs, and cars, and the complexity of both hardware and software has increased over time. Many tests are required for embedded systems because they are subject to many errors. For example, in the automotive industry, safety-critical automotive software requires high reliability, resiliency, and recovery. This affects its design, development, test, and maintenance. In addition, many verifications make them more time-consuming and costly. Not only testing but also strict rules are needed. Formal methods are effective for this purpose. In this paper, we also focus on model checking [1]. The model checker in the existing C language is hardware-independent, so it is intended for ANSI-C verification [2]. To verify an embedded system, it is necessary to construct a verification system that takes into account many hardware-dependent functions. In a program written in C, for some types of Interrupt Service Routine (ISR) the point in time of the execution is unpredictable or hard to predict. This causes problems with the verification of time constraints that embedded and safety-critical software usually has. And since each state in which an instruction is executed represents a state of each block of Control Flow Graph (CFG) or CFA, the state explosion problem occurs. B. Schlich et al. developed an interrupt handling execution reduction (IHER) [3] algorithm to suppress the generation of these interrupts. IHER is an algorithm that prevents the interrupt handler (IH) from processing interrupts that occur at specific locations in the program. The core idea is to reduce execution with no dependencies [3]. We will explain 'dependency' in Section 3. If there is a dependency, it is allowed to occur, but if there is no dependency, it is not allowed to occur. In B. Schlich's study, although he uses timers in the experiments, there is no mention of periodic timer interrupts. However, timer interrupts are often used in the construction of embedded systems. Therefore, the introduction of the concept of time can contribute to reducing not only simple event interrupts but also periodic interrupts, the so-called timer interrupts. In this paper, in order to reduce timer interrupts, we define a Timed CFA that can handle time, and propose Timed IHER.

2. Related Works

In this section, we explain the position of this research in terms of timed models and interrupt processing methods in reducing timer interrupts, based on related research.

© 2023 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 October 2023 doi:10.20944/preprints202310.0337.v1

1. B. Schlich studies a new method for verifying the assembly code of model checking software for microcontrollers [4]. A simulator based on static analysis constructs a state space, abstracts time, handles non-determinism, and over-approximates the behavior exhibited by the actual microcontroller. On the other hand, our previous paper has developed a model checker that uses static and dynamic analysis, such as undefined values [5]. This paper and the reference [4] adopt different approaches as follows. In the reference, when the model checker requests a successor to a state it has not yet generated, the state space generates the successor on the fly using the simulator. In this paper, the verification system completes the state transition system and passes it to the simulator.
   This article generates the entire CFG upfront. The latter is bette [4]r and we will achieve it in the future.
2. B. Schlich et al. have studied IHER to reduce the number of Interrupt Handler executions [3]. B. Schlich et al. also study various abstraction techniques based on static program analysis [6]. In this paper, we extend this IHER.
3. S. Yamane and others conducted a study using IHER to enable verification using SMT solvers [7] together with Assembly Code Block (ACB) [8]. We focus on timer interrupt and does not use SMT solver for verification.
4. Matthew Kuo and others [9] calculate the Worst-Case Response Time (WCRT) using reachability to analyze the assembly code. However, the model is different from our study, Matthew Kuo and others generate TCCFG (Timed Concurrent Control Flow Graph) by static analysis from each assembly code, but we verify microcontrollers that run on a single thread and perform model checking.
5. Wu Yajun [10] defined a timed Kripke structure and verified its real-time nature. Timed Kripke structure is nondeterministic finite automaton but it is not appropriate for our study because its model does not hold the instruction element of the program. We have to control a model by the value of the program counter and the edges should hold the instruction.
6. R. Alur and D. L. Dill studied timed automata [11]. Timed automaton is an extension of a finite state automaton and is a model that describes the system by both discrete event and continuous time-lapse according to state transitions. On the other hand, in this study, we develop timed CFA for the execution time of assembly program. Our study is different from timed automata. The proposed structure deals with discrete time in our study, and we use the execution time of each instruction of the assembly program in each state.
7. The previous study in reference [12] reduces timer interrupts to include real-time performance. On the other hand, this study differs in that it does not consider real-time performance but defines the algorithm more concretely and conducts experiments using multiple timer interrupts in a case study.
8. Recently, L. Lihao et al. proposed an efficient verification of low-level embedded C software with interrupts based on partial-order coding and symbolic execution [13]. On the other hand, this paper verifies an assembly program with an algorithm that also reduces timer interrupts based on the reduction of the number of interrupt handlers by IHER; if the method of Lihao Liang et al. is adopted, nested interrupts can be handled effectively in this paper.

3. IHER

IHER is a method developed by B.Schlich et al. to block the execution of IHs as much as possible. It is an abstraction technique based on partial order reduction [14]. If there is no dependency between the IH and the main program, the IH's execution is reduced by blocking the IH. The core idea of dependency is that one instruction influences the other.

For example, when an assembly instruction of IH and the main program read/write in the same memory location, the overall processing behavior of the program may change if the interrupt is blocked or not. This technique was developed to execute IH only when such behavior changes. IHER consists of the following four steps.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 October 2023 doi:10.20944/preprints202310.0337.v1

1. Detect dependencies between IHs
   Step 1 performs when two or more IHs exist. $i, j \in$ IH depend on each other if either of the following conditions is true. Access here means both writing and reading.

   - one IH enables or disables the other IH,
   - one IH writes to a memory location accessed by the other IH, or
   - one IH writes to the memory location used in an atomic proposition (AP).

   Examples of APs include a variable being equal to a certain value. IHs that manipulate the memory location of an AP manipulate the core of the program. So, It is therefore necessarily associated with all IHs. In other words, only one IH is mentioned, and if one IH writes the memory location used by the AP, then all IHs are dependent on each other.
2. Detect dependencies between program locations and IHs
   step 2 shows the conditions under which two labels, the $execution_i$ label and the $barrier_i$ label, are attached on locations to detect dependencies between the program and IH. An $execution_i$ label allows the execution of an IH executed after the execution of the program instruction. On the other hand, the label $barrier_i$ denotes that there exists a dependency between that program location and an IH, and therefore, this IH needs to be executed before the instruction at that location is executed. In determining the label, we assume that program location k is a direct predecessor of program location l. Let program location k be a direct predecessor of program location l. Formally, for each $i \in IH$, l is labeled with $execution_i$ if one of the following conditions is satisfied:

   - k enables or disables i,
   - k writes a memory location that i accessed, or
   - k writes a memory location that is used in an AP.

   Also, for each $i \in$ IH, a program location l is labeled with $barrier_i$ if one of the following conditions is satisfied.

   - i writes a memory location that l accessed,
   - l enables or disables i,
   - l writes a memory location that i accessed, or
   - l writes a memory location that is used in an AP.
3. Refine results
   Step 3 performs refinement on the $execution_i$ label. Each $execution_i$ label is moved until one of the following conditions is satisfied.

   - a program location labeled with $barrier_i$ is reached,
   - a loop entry is found, or
   - a loop exit is found.
4. Label blocking locations
   In the last step, all program node positions are labeled IH. At this point $i \in$ IH, we assign a $blocking_i$ label to any location without an $execution_i$ label.

IHER can reduce the number of program locations where an interrupt handler (IH) may execute. In this paper, we use IHER to reduce the state space. The paper by B.Schlich proves that the model checking of CTL*-X is valid even though IHER reduces the number of program locations where the execution of an interrupt handler (IH) must be considered. Here, CTL*-X is defined as prohibiting the nexttime operator X in the CTL* formulas [15]. In this paper, since safety is verified for assembly code using a subclass of CTL*-X, the verification is valid even if the IHER reduces the location of the assembly program.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 October 2023 doi:10.20944/preprints202310.0337.v1

4. Formal Model of an Assembly Program

In this paper, the assembly program is applied to CFA (Control Flow Automata) [16] because IH can occur at every assembly instruction. In Shlich's study [3], IHER was explained using CFG [17,18], which is different from CFA. Nodes in CFA are program locations, and directed edges between nodes are instructions that execute when a control moves from the source to the destination. In this paper, since time must be calculated when dealing with timer interrupts in the static analysis, we use the execution time of instructions to calculate the time. Therefore, we define Timed CFA, which extends CFA and introduces a concept of time. While Toth et al.'s TCFA represents real number time, our time CFA represents natural number time to handle program execution time [19]. We deal with exact program execution times, while Ermedahl et al. deal with worst-case and best-case response times [20]. The paper by G. Hou et al. treats interrupts as randomly occurring [21]. Our paper covers all possible interruptions.

The Timed CFA of the assembly program is defined as $N = (L, l_0, O, I, \rightarrow, T)$ where

- L : a set of program locations,
- l_0: $l_0 \subset L$, and the initial location of the program,
- O : a finite set of instructions in main (non-interrupt),
- I : a finite set of executinos of interrupt handlers,
- T : execution times such as OT and IT of each instruction in O and I at each program location,
- $\rightarrow$: $\rightarrow \subseteq L \times O \times OT \times I \times IT \times L$, the set of transition relations.

"execution time of each instruction in O and I" means a single natural number. Also it is a point in time since program execution started. There's O and I within the same transition - which both O and I executed in the transition.

We consider only the fact that the execution time of interrupt handler has to be always constant. We represent branching based on data by timed CFA with branched structures.

In addition, each node is shown as follows.
$n_j \in \rightarrow$, where $n_j = (l_j, op_j, tm_op_j, ih_i, tm_ih_i, l_{j+1})$. The assumption is that i is the number of each interrupt and j is the number of each node. l_j is the current location of the Timed CFA, op_j is the instruction of the program, tm_op_j is the execution time of the instruction in the program, ih_i is the interrupt handler, tm_ih_i is the execution time of the interrupt handler executions, l_{j+1} is the next location of the Timed CFA.

An example of Timed CFA is shown in Figure 1. The right side of the figure represents a timer interrupt, and the $period_ih_i$ represents the timer period. The time can be accurately calculated by having each node hold the execution time of instruction. Instructions are microcontroller instructions, such as SUB.L and MOV.L. Instructions are associated with transitions.

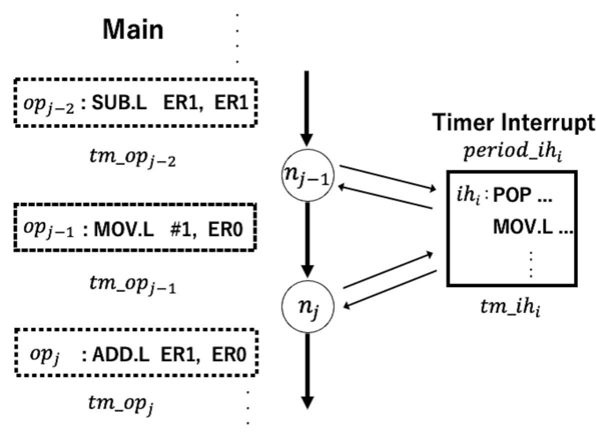

Figure 1. An example of Timed CFA of assembly code.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 October 2023 doi:10.20944/preprints202310.0337.v1

5. Proposed Method of timed IHER

5.1. Verification system

This subsection describes the verification system used in this paper. As a premise, this study uses the assembly code corresponding to the H8/3687F microcontroller [22] manufactured by Electronics Corporation as the verification target and conducts implementation and experiments. The details are explained in the experiment chapter. First, the assembly code is lexical analysis and syntax analysis by the lexer and parser, respectively. We use JFlex [23], and BYACC/J [24] for Java to develop Lexer and Parser. Next, Timed CFA is constructed from the memory map and the parsed code. Then, a static analysis called Timed IHER, the proposed method, is performed to output a Timed CFA with reduced interrupt executable locations. The Timed CFA is then input to the model checker along with the verification properties to confirm that the state space can be reduced.

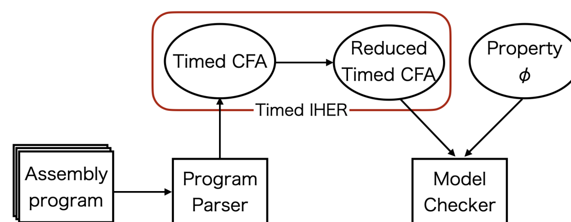

Figure 2. Verification system.

5.2. Algorithm

The Timed IHER algorithm is described in Algorithm 1. The central idea is to suppress execution at the node of occurrence by strictly calculating the execution time not only for interrupts due to events but also for timer interrupts.

In step 1, the dependencies between interrupts are detected. This is detected under the same conditions as in existing research [3]. If the timer interrupt is ready to be executed, it must be executed in the same way as when the dependent interrupt is executed.

In step 2, the execution time is calculated and the label is determined. For this purpose, the labels are determined for each interrupt in order of execution priority, and the cases are divided into timer interrupts and other interrupts. In the latter case, the conventional $execution_i$ and $barrier_i$ are used to determine the label. In the former case, it first determines whether the total instruction execution time TM_i exceeds the $period_ih_i$, which is the timer period. Then, if the same conditions as for the $execution_i$ label are satisfied, the $timer_execution_i$ is labeled with the node, and the execution time of the interrupt process is added to all TM_i, and TM_i is assigned to it divided by $period_ih_i$. If TM_i has not exceeded $period_ih_i$, the current timer time is stored in $pre_time[j][i]$ for use in step 3, TM_i is assigned by subtracting it by $period_ih_i$, and $pre_timer_execution_i$ is labeled. Also, if the same conditions for the $barrier_i$ label are satisfied, then the node should be labeled $timer_barrier_i$. Finally, the instruction time of the main program of the current node is added, and the next node is checked.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 October 2023 doi:10.20944/preprints202310.0337.v1

Algorithm 1 Timed IHER

Require: Timed CFA
Ensure: Reduced Timed CFA

```
1:  n_j = (l_j, op_j, tm_op_j, ih_i, tm_ih_i, l_{j+1}) ∈→
2:  ε(n_j) //the conditions to execution_i in step 2 of IHER
3:  ψ(n_j) //the conditions to barrier_i in step 2 of IHER
4:  period_ih_i //timer period of each timer interrupts
5:  L(n_j) //sets of labels at the node
6:  pre_time[j][i] //timer value at pre_timer_execution_i
7:  TM_i //total of instruction execution time
8:  x //the node number after transition in step 3
9:
10: //step 1
11: Detect dependencies between ih_i
12:
13: //step 2
14: for op_j of n_j do
15:     for ih_i ∈ Interrupts do
16:         if ih_i is timer interrupts then
17:             if period_ih_i ≤ TM_i && ε(n_{j-1}) == true then
18:                 L(n_j) := L(n_j) ∪ timer_execution_i
19:                 ∀i.TM_i += tm_ih_i
20:                 TM_i %= period_ih_i
21:             else if period_ih_i > TM_i && ε(n_{j-1}) == true then
22:                 L(n_j) := L(n_j) ∪ pre_timer_execution_i
23:                 pre_time[j][i] = TM_i
24:                 TM_i -= period_ih_i
25:             end if
26:             if ψ(n_j) == true then
27:                 L(n_j) := L(n_j) ∪ timer_barrier_i
28:             end if
29:         else
30:             if ε(n_{j-1}) == true then
31:                 L(n_j) := L(n_j) ∪ execution_i
32:                 ∀i.TM_i += tm_ih_i
33:             end if
34:             if ψ(n_j) == true then
35:                 L(n_j) := L(n_j) ∪ barrier_i
36:             end if
37:         end if
38:     end for
39:     ∀i.TM_i += tm_op_j
40: end for
```

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 October 2023 doi:10.20944/preprints202310.0337.v1


```
1:  //step 3
2:  for op_j of n_j do
3:      for ih_i ∈ Interrupts do
4:          if timer_execution_i ∈ L(n_j) then
5:              Transition to a node with a timer_barrier_i
                ,loop entry or loop exit
6:              L(n_x) := L(n_x) ∪ timer_execution_i
7:          else if pre_timer_execution_i ∈ L(n_j) then
8:              Transition to a node with a timer_barrier_i
                ,loop entry or loop exit
9:              pre_time[j][i] += total time of passed nodes
10:             if period_ih_i ≤ pre_time[j][i] then
11:                 L(n_x) := L(n_x) ∪ timer_execution_i
12:                 for all j such that x ≤ j do
13:                     ∀i.pre_time[j][i] += tm_ih_i
14:                 end for
15:             else
16:                 L(n_x) := L(n_x) ∪ blocking_i
17:             end if
18:         end if
19:         if execution_i ∈ L(n_j) then
20:             Transition to a node with a barrier_i
                ,loop entry or loop exit
21:             L(n_x) := L(n_x) ∪ execution_i
22:         end if
23:     end for
24:
25:     //step 4
26:     for n_j do
27:         if (timer_execution_i || execution_i) ∉ L(n_j) then
28:             L(n_j) := L(n_j) ∪ blocking_i
29:         end if
30:     end for
31: end for
```

In step 3, refinements are made to the $timer_execution_i$ and $execution_i$ labels. The $timer_execution_i$ label is moved until reaching the $timer_barrier_i$ label, the entrance or exit of the loop. The $execution_i$ label is moved until reaching the $barrier_i$ label, the entrance or exit of the loop. If one of the following is satisfied, we label the node with the $timer_execution_i$ or $execution_i$ and x is the node number after the transition. On the other hand, in case of $pre_timer_execution_i$ label, we add the execution time of the instruction of the node which has passed to $pre_time[j][i]$ including instruction of interrupt when it reaches a node satisfied with the same conditions as the $timer_execution_i$ label in the transition. If this $pre_time[j][i]$ exceeds the $period_ih_i$, the $timer_barrier_i$ is labeled, and we add the execution time of the interrupt to $pre_time[j][i]$ after the transitioned node destination. In other words, even if the timer period has not been elapsed in step 2, if it is elapsed at the destination, the execution of the timer interrupt is permitted. We can accurately calculate the time by adding the execution time to the timer

of $pre_timer_execution_i$ label and other timer interrupt after the node after the transition, to the extent that we did not add them in step 2.

In step 4, $blocking_i$ is labeled to block interrupts at node positions other than those where no $execution_i$ or $timer_execution_i$ are labeled. This is the algorithm of proposed method.

The reason for the subtraction at line 24 is to distinguish $pre_timer_execution_i$ from the $timer_execution_i$ label and to prevent execution even though permission to execute has not been granted. For example, if both nodes before and after the $pre_timer_execution_i$ label are converted to the $timer_execution_i$ label at step 3, and the node is actually executed, the timer at the later node may not have overflowed.

5.3. Example

To help you understand the algorithm, we will explain a simple example. We apply Timed IHER on the simple code written in Figure 3. To simplify the time, we have one timer interrupt with a period of $3ms$ and put the execution time of each instruction as shown in Figure 4. Then, $pre_timer_execution_0$ label is green, $timer_execution_0$ label is blue, and $timer_barrier_0$ label is hexagonal.

```
Main
0: SUB.L  ER1, ER1
1: MOV.L  #1, ER0
2: ADD.L  ER1, ER0
3: ADD.L  #1, ER1
4: RTS

Timer Interrupt
(period_ih_0 = 3ms)
0: MOV.L  #1, ER1
1: RTE
```

Figure 3. Test code.

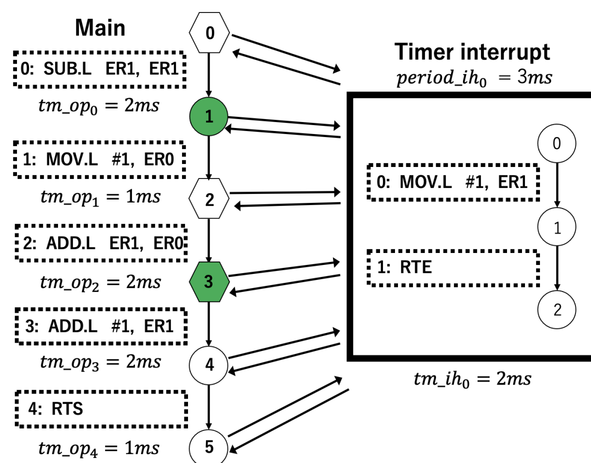

Figure 4. After step 2.

First, we perform labeling in step 2. At n_0, the condition of $timer_barrier_i$ is satisfied. At n_1, since TM_0 is $2ms$, it is less than the period and the condition of $pre_timer_execution_i$ is satisfied. Then, after TM_0 is assigned to $pre_time[1][0]$, $period_ih_0$ is subtracted from TM_0. At n_2, condition of $timer_barrier_i$ is satisfied and TM_0 is $0ms$. At n_3, TM_0 is $2ms$, so the condition of $pre_timer_execution_i$ is satisfied as with n_1. Then, after assigning TM_0 to $pre_time[3][0]$, $period_ih_0$ is subtracted from TM_0. The condition

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 October 2023 doi:10.20944/preprints202310.0337.v1

of $timer_barrier_0$ is also satisfied, so it is labeled. Since n_4 and n_5 don't satisfy any of the conditions, step 2 becomes like Figure 4.

Next, we perform refinement in step 3. First, $pre_timer_execution_0$ of n_1 is transitioned to n_2 labeled $timer_barrier_0$. At this time, the $1ms$ of the MOV instruction is added to the $2ms$ stored in $pre_time[1][0]$, resulting in $3ms$, which is equal to $3ms$ of the timer period, allowing the execution. Therefore, n_2 is labeled $timer_execution_0$. So, the execution time of the timer interrupt must be added to $pre_time[j][i]$ after n_2. Therefore, since $2ms$ of timer interrupt is added to $2ms$ stored in $pre_time[3][0]$ to get $4ms$ and n_3 is labeled $timer_barrier_0$, $timer_execution_0$ is labeled. Step 3 is shown in Figure 5.

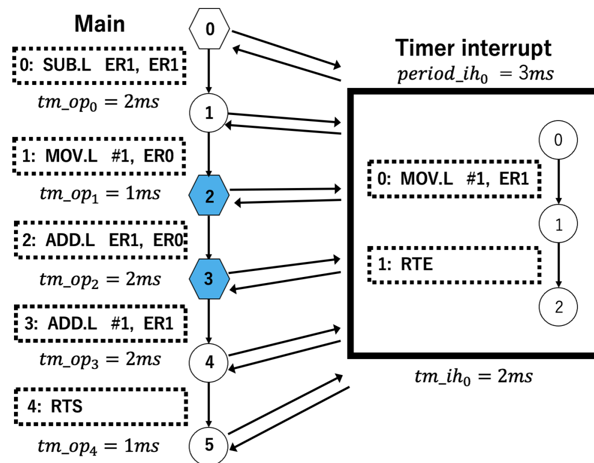

Figure 5. After step 3.

Finally, in step 4, the execution of the timer interrupt is suppressed by labeling $blocking_0$ for the nodes that are not labeled with $timer_execution_0$. The result is as shown in Figure 6. $blocking_0$ is labeled as multiple locations 0, 1, 4, 5 with no interruptions.

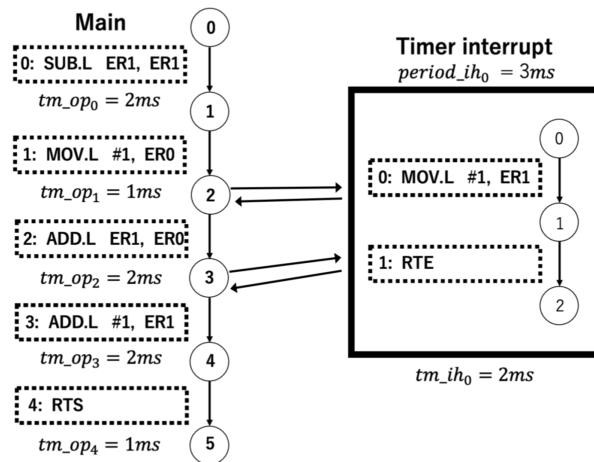

Figure 6. After step 4.

5.4. The correctness of timed IHER

According to the reference [3], we can show the correctness of the time interrupt reduction by defining the semantics of a timed CFA by a labeled transition system. Only a summary of the correctness is given below. We define the labeled transition system $T(G)$ of a timed CFA G,and the labeled transition system $T(G_{reduced})$ of a reduced timed CFA $G_{reduced}$ by timed IHER. We can establish the correctness of our timed IHER by showing that the original timed CFA G and the reduced timed CFA $G_{reduced}$ are equivalent. More concretely we can show that the original timed CFA G and the reduced timed CFA $G_{reduced}$ are related by divergence-sensitive stutter bisimulation [15,25].

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 October 2023 doi:10.20944/preprints202310.0337.v1

6. Experiment

We developed our own model checker. We performed our validation in the environment shown in Table 1.

Table 1. Environment of experiment.

OS	macOS Monterey 12.4
CPU	Intel(R) Core(TM) i5-8257U CPU @ 1.40GHz
Memory	8GB
Java	11.0.4
Program	12000lines

In this experiment, the assembly code for the Renesas Electronics H8/3687F microcontroller [22] is used as the verification target, and implementation and experiments are conducted. All general-purpose registers are 16 bits wide, and 16 registers ($E0$-$E7$ and $R0$-$R7$) are installed. When general-purpose registers are used as data registers, they can be accessed as 8, 16, or 32-bit registers. When 32-bit registers are accessed, E and R are integrated and handled as ER registers, of which ER7 functions as the stack pointer (SP). On the other hand, the control register consists of one 24-bit wide program counter register (PC) and one 8-bit wide condition code register (CCR). The total address space for the program and data area is 64 Kbytes.

We examine three cases. In each case, we conduct experiments for several combinations of the number of timer interrupts (TI) and software interrupts (SI).The case 1 is used for transfer by the serial communication interface (SCI3) with two timer interrupts. The case 2 uses two LEDs. Two timer interrupts are used, one to light one LED at a prime number and the other to light the other LED at an even number, adding 1 every 0.5 seconds until the number goes from 0 to 10. The case 3 is a PID control program. One timer interrupt acquires the sensor value and sets a new target current value when a timer overflow occurs. When the other timer overflows, the timer interrupt performs PID control for a fixed cycle based on the current target value and the measured current value, and outputs the value to the motor. All three cases are implemented in e-nuvo WHEEL [26], and the verification property is $AG\neg error$. e-nuvo WHEEL is a microcomputer-controlled robot. The reference language is Japan. Only a manual in Japanese exists.

Table 2 shows the required memory, execution time, and state space reduction rate for cases 1 through 3. The required memory is computed by Activity Monitor. We confirmed a reduction of about 90% for case 1 and case 3, and about 60% for case 2, but the reduction rate and execution time vary greatly depending on the number of programs and interrupts.

Table 2. The number of state stored by the verification.

case	TI	SI	Without Timed IHER (required memory(kb))	Without Timed IHER (execution time(mus))	Timed IHER (required memory(kb))	Timed IHER (execution time(μs))	Reduction
1	2	0	117448	1889.6	7134	997.2	94%
	2	1	158747	2125.8	8049	1075.2	95%
2	1	1	20260	304.3	8624	280.4	57%
	2	0	26324	253.8	10520	131	60%
3	2	0	2160677	45032.7	204433	25032.7	91%
	2	1	2528550	52042	573202	32051.3	77%

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 October 2023 doi:10.20944/preprints202310.0337.v1

7. Conclusions

In this paper, we proposed the formal model and algorithm based on IHER that can reduce timer interrupts. 12000 lines of java code were used to develop the entire system, including model building and model checking, and its effectiveness was demonstrated. However, it does not strictly take into account priority and multiple interrupts and is limited to verification when the program is executed in a possible position. Also, the optimal refinement of timers when loops exist in the program is unknown. Therefore, handling loops, including strict timer understanding, must be studied. Currently, we are considering algorithms that can reduce the number of states more, and expanding the scope of application of the proposed method.

References

1. Edmund M. Clarke Jr., Orna Grumberg, Doron Peled. Model Checking. The MIT Press, 1999.
2. Schlich, B.; Kowalewski, S. Model checking C source code for embedded systems. Int. J. Softw. Tools Technol. Transf. 2009, 11, 187?202.
3. Bastian Schlich, Thomas Noll, Jorg Brauer, and Lucas Brutschy. Reduction of interrupt handler executions for model checking embedded software. In Haifa Verification Conference, pp. 5?20. Springer, 2009.
4. Schlich, B. Model Checking of Software for Microcontrollers. ACM Trans. Embed. Comput. Syst. 2010, 9, 1?27.
5. Yamane, S.; Konoshita, R.; Kato, T. Model checking of embedded assembly program based on simulation. IEICE Trans. Inf. Syst. 2017, 100, 1819?1826.
6. Schlich, B.; Brauer, J.; Kowalewski, S. Application of static analyses for state-space reduction to the microcontroller binary code. Sci. Comput. Program 2011, 76, 100?118.
7. Armando, A.; Mantovani, J.; Platania, L. Bounded model checking of software using SMT solvers instead of SAT solvers. Int. J. Softw. Tools Technol. Transf. 2009, 11, 69?83.
8. Yamane, Satoshi; Kobashi, Junpei; Uemura, Kosuke. Verification Method of Safety Properties of Embedded Assembly Program by Combining SMT-Based Bounded Model Checking and Reduction of Interrupt Handler Executions. Electronics , 2020, 9.7: 1060.
9. Matthew Kuo, Roopak Sinha, and Partha Roop. 2011. Efficient WCRT analysis of synchronous programs using reachability. In Proceedings of the 48th Design Automation Conference (DAC '11). Association for Computing Machinery, New York, NY, USA, 480?485.
10. Wu, Yajun, and Satoshi Yamane. "Model Checking of Real-Time Properties for Embedded Assembly Program Using Real-Time Temporal Logic RTCTL and Its Application to Real Microcontroller Software." IEICE TRANSACTIONS on Information and Systems 103.4 (2020): 800-812.
11. Alur, Rajeev, and David Dill. "The theory of timed automata." Real-Time: Theory in Practice: REX Workshop Mook, The Netherlands, June 3?7, 1991 Proceedings. Springer Berlin Heidelberg, 1992.
12. KIRIYAMA, Taro; WU, Yajun; YAMANE, Satoshi. Reduction of Timer Interrupts for Embedded Assembly Programs Based on Reduction of Interrupt Handler Executions. In: 2021 IEEE 10th Global Conference on Consumer Electronics (GCCE). IEEE, 2021. p. 464-466.
13. Liang, L.; Melham, T.; Kroening, D.; Schrammel, P.; Tautschnig, M. Effective Verification for Low-Level Software with Competing Interrupts. ACM Trans. Embed. Comput. Syst. 2018, 17, 36:1-36:26.
14. Peled, D.: Ten years of partial order reduction. In: Y. Vardi, M. (ed.) CAV 1998. LNCS, vol. 1427, pp. 17?28. Springer, Heidelberg (1998)
15. Browne, M.C.; Clarke, E.M.; Grumberg, O. Characterizing Finite Kripke Structures in Propositional Temporal Logic. Theor. Comput. Sci. 1988, 59, 115-131.
16. Dirk Beyer, Thomas A Henzinger, Ranjit Jhala, and Rupak Majumdar, The software model checker blast. International Journal on Software Tools for Technology Transfer, Vol. 9, No. 5-6, pp. 505 525, 2007.
17. Muchnick, S. Advanced Compiler Design and Implementation; Morgan Kaufmann Publishers Inc.: Burlington, MA, USA, 1997.
18. Aho, A.V.; Sethi, R.; Ullman, J.D. Compilers: Principles, Techniques and Tools, 2 Revised ed of International ed.; Pearson: London, UK, 2006.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 October 2023
doi:10.20944/preprints202310.0337.v1

19. Toth, T.; and Majzik, I. : Formal Modeling of Real-Time Systems with Data Processing. In Pataki, B., editor(s), Proceedings of the 23rd PhD Mini-Symposium, pages 46-49, 2016.
20. Andreas Ermedahl, Jakob Engblom: Execution Time Analysis for Embedded Real-Time Systems. Handbook of Real-Time and Embedded Systems 2007
21. Gang Hou, Weiqiang Kong, Kuanjiu Zhou, Jie Wang, Xun Cao, Akira Fukuda: Analysis of Interrupt Behavior Based on Probabilistic Model Checking. IIAI-AAI 2018: 86-91
22. Corporation, R. E. : Renesas Electronics, Renesas Electronics Corporation (online), http://japan.renesas.com/
23. JFlex-The Fast Scanner Generator for Java. 2015. Available online: http://jflex.de/ (accessed on 3 September 2022).
24. BYACC/J Java Extension. 2013. Available online: http://byaccj.sourceforge.net/ (accessed on 3 September 2022).
25. van Glabbeek, R., Weijland, W.: Branching time and abstraction in bisimulation semantics. Journal of the ACM 43(3), 555-600 (1996)
26. ZMP Inc. Nuvo R WHEEL. 2010. Available online: https://robot.watch.impress.co.jp/cda/news/2006/07/12/81.html (accessed on 3 September 2022).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.