Impact of DRAM Refresh on the Execution Time of Real-Time Tasks

Pavel Atanassov, Peter Puschner
Institut fuer Technische Informatik
Technische Universitaet Wien, Vienna, Austria
{pavel,peter}@vmars.tuwien.ac.at

Abstract *In this paper we discuss how DRAM refreshes influence the execution times and the Worst-Case Execution-Time (WCET) analysis of real-time programs executed on platforms that use DRAM. To this end we identify the parameters and the circumstances that affect the wait states caused by a DRAM refresh, and demonstrate that DRAM refreshes can invalidate the worst-case execution path determined by static WCET analysis for a program.*

Keywords: *DRAM refresh*, *WCET analysis*, *embedded systems*

1. Introduction

Dynamic random-access memory (DRAM) needs to be periodically refreshed because its data-storing capacitors slowly discharge. A *DRAM matrix* contains cells, each cell holding one bit. The duration for which the cells can keep their bit state without being refreshed is called *data retention time*. Reading a row of the matrix simultaneously refreshes all cells in the row. Each row has to be refreshed periodically, at least once in a certain time window, called *refresh period* (t_{REF}), which must be shorter than the retention time of the worst cell in the matrix. In this paper we only address the most common refresh implementation – *distributed DRAM refresh.* Using this implementation, single rows of the DRAM matrix are refreshed periodically. *Refresh interval* (t_{Rint}) is the time between refreshing two subsequent rows, and is given as the division of t_{REF} by the number of rows.

If the processor tries to access memory while a DRAM refresh is in progress, the execution of the current instruction stalls. The wait states caused by this interference lead to an increase of the execution times of programs. In general-purpose computer systems this slowdown is hardly of significant importance. In real-time systems, however, DRAM refresh time has to be accounted for in the timing analysis to guarantee the timely operation of the system.

1.1 Related Work

The additional wait-states introduced by DRAM refreshes lead to a relatively small increase (say, 2%) of the overall execution time of programs. However, although the increase is small, it is important that it is accounted for in real-time systems, where real-time tasks have timing deadlines. The common problem of performance degradation in computer systems due to DRAM refreshes is well known. Still, the impact of refreshes on real-time tasks has not received enough coverage in literature.

Park and Shaw in [4] first described the impact of DRAM refresh on execution-time analysis. They tried to experimentally validate their static worst-case execution time (WCET) estimations for a 10MHz MC68010 controller. This occurred to be surprisingly difficult, mainly due to DRAM refreshes. The effects of refreshes were larger than expected, mainly due to bus arbitration. The worst-case increase of program execution time caused by refreshes was measured to be 6.7%. To safely account for refreshes, the computed WCET bounds were increased by 7%. We will show in Section 3 that this approach has the drawback of eventually invalidating the worst-case execution path found by static analysis, which was not considered in [4].

In his master's thesis, Vrchoticky [6] also investigated the overhead of DRAM refreshes on task execution time. Oscilloscope measurements showed that the delay caused by a DRAM refresh was 2 CPU cycles longer than the 3 cycles specified in the documentation.

Panda *et al*. [3] presented several compiler-level optimizations for utilizing of efficient memory access features of DRAM memories, such as DRAM Page Mode Read/Write and

Read-Modify-Write. According to the authors, as a post-processing step of the compiler they incorporate DRAM refresh cycles in the memory access schedule of a particular program, ensuring that refreshes do not overlap in time with any other memory access. We consider such an approach as impractical, because it assumes that the knowledge of when a refresh would occur is available at the compiler-level. This is impossible because the execution of a task cannot be synchronized with DRAM refreshes.

Ohsawa *et al*. [2] proposed hardware solutions to the problem of performance degradation caused by DRAM refreshes. The *Selective Refresh Architecture (SRA)* allows selective refreshes of DRAM rows. The idea is that rows holding data that will not be used anymore in the program need not be refreshed. The *Variable Refresh Period Architecture (VRA)* has different refresh periods for each row, depending on the row's specific data retention characteristics. To our knowledge, such architectures are still not present on the market, and therefore, not considered in our work.

1.2 Contributions of the Paper

In this paper we discuss how DRAM refreshes impact the execution times and the WCET analysis of real-time tasks. The contributions of the paper can be summarized as follows:

1) We identify the hardware parameters that influence the delay caused by a single DRAM refresh. Our observations on a MC68360 controller from Motorola show that the delay caused by a refresh depends on the types of the memory accesses between which the refresh occurs (DRAM vs. SRAM accesses). Measurements reveal that the actual maximum possible delay caused by a DRAM refresh is longer than documented in the hardware manual.

2) Beside a WCET bound, static WCET analysis yields information about the worst-case path of a program. We demonstrate that the worst-case execution path determined by static WCET analysis not considering DRAM refresh may no longer be the worst-case path in the presence of refreshes. This means that by adjusting statically computed WCET bounds to account for the influence of refreshes, the information about the worst-case execution path may no longer be valid.

The paper is structured as follows. Section 2 presents the hardware features that influence the wait states caused by a refresh. It is shown that the delay caused by a refresh depends on the type of the memory accesses between which it occurs. In the same section a comparison is presented between the documented refresh duration and the measured duration. In Section 3 we demonstrate that DRAM refreshes can invalidate the worst-case execution path determined by static WCET analysis for a program. Section 4 is the conclusion of the paper.

2. Features Influencing the Delay Caused by a Single Refresh

A safe estimation of the impact of DRAM refreshes on program execution times requires the knowledge of the maximum possible delay a single refresh can cause. In this section, different architectural features are described that can influence this delay.

Our measurements and observations were carried out on a VME IP-360 board from Microsys with a MC68360 Quad Integrated Communications Controller (QUICC) from Motorola using 25MHz CPU32 core [1]. There were 256 Kbytes of 16-bit SRAM and 1 Mbytes of 32-bit DRAM. No cache memories were present. We measured execution times of programs by sending start/stop messages to an external timer.

The timer was a HP53131A universal counter from Hewlett Packard. It allowed us to measure execution times with the precision of a single CPU clock tick. The target CPU clock-out signal was input on the timer's channel 1. The memory-mapped Port B of the Communications Processor Module (CPM) of the QUICC was used as the start/stop line to the external gate of the timer. Execution times were measured as the number of clock ticks on channel 1 between a start and a stop message on the external gate.

The DRAM controller of the MC68360 uses $\overline{CAS}$-*Before*-$\overline{RAS}$ (*CBR*) refreshes, *i.e.*, refreshes are started by asserting $\overline{\text{CAS}}$ before $\overline{\text{RAS}}$ (DRAM reads/writes, in contrast, start by asserting $\overline{\text{RAS}}$ before $\overline{\text{CAS}}$). This signal order activates an internal refresh counter of the DRAM chip that generates the address of the row to be refreshed. Figure 1 illustrates a *CBR* refresh on the DRAM chips used. The refresh begins with the assertion of $\overline{\text{CAS}}$ followed by assertion of $\overline{\text{RAS}}$. After a signal is negated, it cannot be asserted again for some time, called *precharge time*. Hence, after the negation of $\overline{\text{RAS}}$ at the end of the t_{RAS} interval, the DRAM cannot be accessed until $\overline{\text{RAS}}$ has been precharged again.

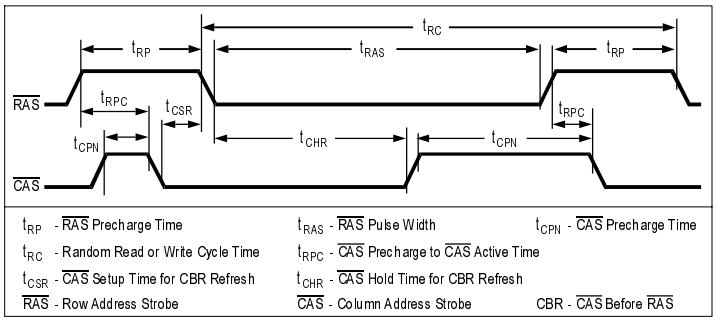

Figure 1. CAS -Before- RAS refresh on the MCM514256A-J70 DRAM chip from IBM

2.1 RAS Hold and Precharge Times

The RAS hold and RAS precharge times for DRAM *refresh* can be different from the hold and precharge times for DRAM memory *accesses*. For example, on the MC68360 controller it is possible to configure the RAS hold and RAS precharge times for DRAM refreshes separate from those for DRAM accesses.

2.2 Type of Memory Accesses between which a DRAM Refresh Occurs

DRAM refreshes can delay accesses to *other memory types than DRAM* (e.g. SRAM).

On the board used, DRAM refreshes do not only delay accesses to DRAM but also SRAM accesses, because the DRAM controller holds the bus during a refresh. Hardware specifications usually provide information about the duration of a refresh only for the case that it occurs between two DRAM accesses. Our observations show that when a refresh occurs between two memory accesses that are not both to DRAM, the refresh duration is formed in a different way than documented. This fact is important because in such situations the duration of a refresh can be longer than documented.

We measured the duration of a refresh for four cases, depending on the type of memory accesses between which the refresh occurs. Table 1 presents the results from the measurements compared to the documented refresh duration for the particular hardware configuration used. The longest delay caused by a refresh was measured when a refresh occurred between a SRAM and a DRAM access.

The measurements revealed that a refresh can cause a longer delay than documented. Hence it is necessary to perform real measurements in order to safely assess the maximum possible delay that a refresh can cause.

Oscilloscope observations also revealed that the way a refresh is performed on the tested hardware is different from the way explained in the documentation. The hardware manual [1] states that the RAS precharge time for a refresh should be *before* the RAS hold time for a refresh. Our observations clearly showed that the RAS precharge time is held *after* the RAS hold time. We contacted the Support Center of the Motorola Semiconductor Department to discuss the issue. It was confirmed that our observations were correct and demonstrate a case of real operation differing from the expected one.

2.3 Internal Instruction Processing

Besides reading and writing memory operands, the execution of an instruction involves internal processing. Nowadays, processors use different resources concurrently. For example, memory accesses and internal instruction processing can be performed in parallel. There can be, however, internal processing cycles during program execution that do not overlap with memory reads/writes. The delay of a DRAM refresh overlapping with such internal processing cycles may be smaller or may even be nullified if the whole refresh overlaps with internal processing.

Figure 2a) shows the execution of two cycles of internal processing between two memory accesses, each 4 cycles long. In Figure 2b) a DRAM refresh occurs between the two memory accesses. The refresh is 6 cycles long, but it overlaps with the two cycles of internal processing, so the effective delay caused by the refresh is 6 – 2 = 4 cycles.

	Value documented in [1]	Measurements (refresh occurs between the following types of memory accesses)			
		DRAM–DRAM	DRAM–SRAM	SRAM–DRAM	SRAM–SRAM
Refresh duration	7	7	7	**8**	6

Table 1. Documented vs. measured duration of DRAM refresh (in CPU cycles)

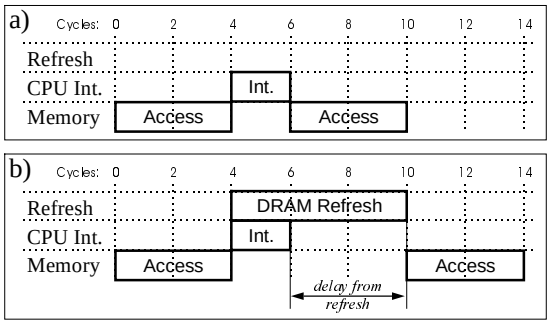

Figure 2. Overlapping with internal processing

2.3.1 A Side Effect from DRAM Refresh

As a side effect, DRAM refresh can cause reordering of the normal instruction execution schedule. We illustrate this by an example observed on the target architecture.

Figure 3a) depicts the normal execution of a CLR instruction that clears Port B of the CPM of the MC68360. The instruction sends a stop message to the external timer, as described at the beginning of this section. First, the high and low word of the memory address of Port B are fetched. After that, the next instruction is fetched. Parallel to this, the internal processing of CLR is performed. At last, Port B is cleared and the stop message is sent in the 68th cycle.

Figure 3b) illustrates the execution of the same CLR instruction when a refresh occurs after Port B's address is fetched. As in case a), at the beginning of the instruction the high and low words of the address of Port B are fetched. At this moment (cycle 62), a DRAM refresh starts and occupies the bus. Consequently, the prefetch of the next instruction is delayed. The internal processing of the CLR is done in parallel to the refresh and at the end of this processing (cycle 66), the request for writing to Port B is submitted to the bus controller. After the refresh ends at cycle 68, Port B is written prior to the pre-fetch, because operand accesses in the MC68360 always have priority over instruction pre-fetches [1]. Thus, though the execution time of CLR increases by 6 cycles (the refresh duration), the stop message is only delayed by 2 cycles (6 cycles refresh – 4 cycles internal processing) because the execution order was changed by the refresh. Such reordering of instruction execution can slightly affect CPU cycle-based measurements.

2.4 Bus Arbitration

On the MC68360 there was no bus-arbitration penalty when performing a DRAM refresh. Nevertheless, additional cycles for bus arbitration before and after a refresh may be present, as reported in [4]. This was also confirmed by the Motorola Semiconductor Support Center.

2.5 Conclusions

In real-time systems that use DRAM it is important to estimate the system's performance degradation due to DRAM refresh cycles. A safe estimation requires the knowledge of the maximum possible delay a single refresh can cause to program execution time. Our experiments revealed that hardware documentations can be incomplete on the issue of DRAM refresh. Reliable knowledge about the actual DRAM refresh timing and performance can be gathered by measurements on the real target hardware.

To make a tight estimate of the impact of DRAM refreshes on program execution time, different features must be considered – hold and precharge time for a refresh, the types of memory accesses between which a refresh occurs (DRAM vs. SRAM accesses), eventual bus arbitration before/after a refresh, and the overlap of refreshes with internal processing.

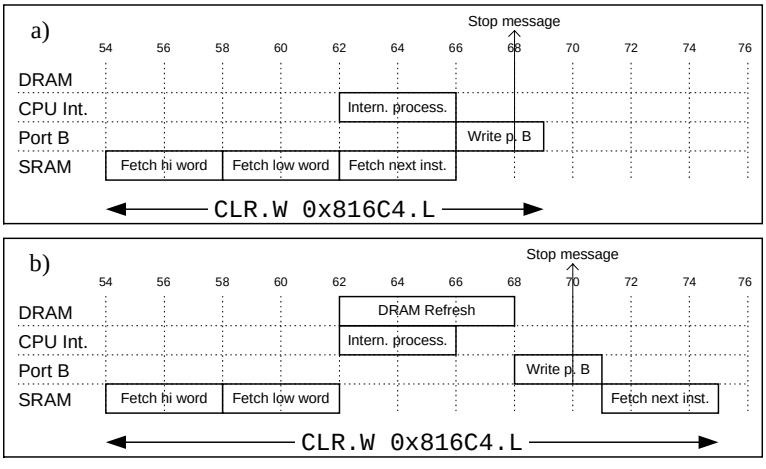

Figure 3. Reordering of the normal instruction execution by a refresh

3. DRAM Refresh and Execution Time Analysis

If a real-time system uses DRAM, the impact of refreshes on the execution time of tasks must be properly accounted for in the timing analysis. Though the increase caused by refreshes is relatively small, it must not be underestimated.

DRAM refreshes occur asynchronously to program execution time. This means, repeated execution of a program with identical input data in the presence of DRAM refreshes results in different interference of the refreshes with the program's memory access pattern. The measured execution times therefore vary in some interval rather than being constant for all executions. DRAM refreshes can be regarded as a periodic highest-priority interrupt with unknown offset to program execution. The periodicity of DRAM refreshes can be used to calculate an upper bound on the maximum increase of execution time they can cause. This will normally be an overestimation of the actual maximum increase.

If the target architecture has cache memories, the overestimation will be bigger, because cache hits do not access conventional memory and can be regarded to as internal processing.

3.1 Adjusting Static WCET Bounds to Account for Refreshes

Static WCET analysis calculates a safe upper bound of the Worst-Case Execution Time (WCET) of a real-time program [5]. The WCET bound is calculated statically without ever executing the program on the target hardware. If the target uses DRAM, this bound can be adjusted to account for the delays caused by refreshes.

A safe estimation of the WCET of a program in the presence of DRAM refreshes is:

$$T_{WCET}^{refr} = T_{WCET} + \left\lceil \frac{T_{WCET}}{t_{Rint} - t_{delay}^{max}} \right\rceil \cdot t_{delay}^{max} \qquad (1)$$

The time T_{WCET} is the WCET bound calculated by static analysis. The symbol $\lceil\ \rceil$ is the integer ceiling function and t_{Rint} is the refresh interval. Note, that although t_{Rint} is named "refresh interval", it is actually the period of the refreshes, whereas the "refresh period" t_{REF} is the period of refreshing one and the same row of the DRAM matrix. t_{delay}^{max} is the maximum possible delay that a single refresh can cause.

For the particular target configuration, the maximum possible increase of the execution time due to refreshes is about 2.13%. Though the impact of refreshes is relatively small, for reasons of safety it has to be accounted for in embedded real-time systems that use DRAM.

3.2 A Problem with the Adjustment

Beside the WCET bound, static analysis also yields information about the path that produces the WCET and about the timing of the program blocks on that path. The information about the worst-case execution path can be used to determine the worst-case input data for a program. Moreover, the timing information about the blocks on that path can be used to further optimize the source code of the program.

We will now show that the worst-case path determined by static analysis (not considering DRAM refresh) may no longer be the worst-case path in the presence of DRAM refreshes.

We constructed a small piece of code, *Catch*, listed in Figure 4. The main function, `catch()`, receives an integer parameter `var`. If `var` is greater than zero, an empty loop is executed. If `var` is equal to zero, the function `loop()` is called 1,000 times. In this function, there are multiply operations with long internal processing time – more than 30 cycles per operation. Thus, if `var` = 0 there are long time intervals of internal processing that can overlap DRAM refreshes, which is not the case if `var` > 0.

```
void catch (int var) {
  int i;
  if ( var > 0 ) for (i = 0; i < 19210; i++);
  else  for (i = 0; i < 1000; i++) loop(1);
}

long loop (int bound) {
  int i; long a, b, c, d, e, f;
  for (i = 0; i < bound; i++);
  a = 346723; b = 782389; c = 982312; d = 123721;
  e = a * b; f = c * d; return (e * f);
}
```

Figure 4. Code of Catch

The MC68360 allows disabling of DRAM refreshes. *Catch*'s code and data were placed entirely in SRAM. The code was executed for `var` = 0 and `var` = 1 when DRAM refreshes were disabled. Then, DRAM refreshes were enabled and the code was executed 10,000 times for each of the two values of `var`. The results from the experiments for the particular hardware configuration are shown on Table 2.

As can be seen from Table 2, the worst-case path without refreshes is not the worst-case path with refreshes enabled. The reason is that when `var` = 0, there are 3,000 multiplications that have long internal processing which does not overlap with any memory access. Consequently, many of the DRAM refreshes occurring on that path overlap with this internal processing and their effect to the execution time is thus nullified. On the other hand, during the execution of the path for `var` = 1, there is no "free" internal processing.

Catch	DRAM refreshes disabled		DRAM refreshes enabled			
			`var = 0`		`var = 1`	
	`var = 0`	`var = 1`	min	max	min	max
Ex. time	**500,179**	499,648	505,055	505,087	505,434	**505,453**

Table 2. Execution times of Catch with and without DRAM refreshes (in CPU cycles)

Therefore, each refresh occurring on that path causes an increase of the execution time. As a result, the worst-case path with refreshes enabled is executed for `var` = 1, while without refreshes, the worst-case path is for `var` = 0.

Formula (1) produces a safe estimate because it considers the worst-case delay by refreshes and because its estimate are proportional to the execution time. Still, the path considered by static WCET analysis is no longer the valid worst-case execution path. As a consequence, it cannot be generally guaranteed that the actual worst-case execution path in the presence of refreshes is the one found by static WCET analysis. This is a secondary effect of DRAM refreshes, which was not reported in literature by now, and was not considered in works that adjust static WCET estimates to account for DRAM refreshes, like in [4].

The probability that a situation like the one described above would occur in real-life is relatively small. It can, however, occur if a task has several alternative main paths of similar execution times but with different instruction and memory access characteristics.

4. Conclusion

In this paper we discussed the impact of DRAM refreshes on the execution time and execution-time analysis of real-time tasks. In general, DRAM refreshes lead to increase of the execution time, which must be properly accounted for in real-time systems.

We identified the hardware parameters that can influence the duration of a refresh. This is important in order to estimate the maximum possible delay a single refresh can cause. The parameters were identified by performing experiments on a real target architecture based on the MC68360 controller from Motorola. We showed that the delay a refresh can cause depends on the types of the memory accesses (DRAM vs. SRAM accesses) between which the refresh occurs. Our measurements revealed a longer delay caused by a refresh than the documented value. This demonstrates that the actual delay caused by a refresh can be greater than the refresh length provided in the hardware documentation. Therefore, it is necessary to validate the documented values by measurements.

We demonstrated that the worst-case execution path determined by static WCET analysis (not considering DRAM refresh) may no longer be the worst-case path in the presence of refreshes. Therefore, adjusting static WCET estimates to account for DRAM refreshes may invalidate the worst-case execution path determined by static analysis. The occurrence of such situations appears to be hard to predict.

Though the impact of refreshes on program execution times is relatively small, for reasons of safety it must always be considered for real-time systems that use DRAM. An upper bound for the impact can be calculated but a precise estimate appears to be difficult.

Acknowledgements

Grants by the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) under contract P12476-INF, as well as by the IST research project "Systems Engineering for Time-Triggered Architectures (SETTA)" of the EC under contract IST-10043.

References

[1] Motorola Inc., "*MC68360 Quad Integrated Communications Controller. User's Manual*". Motorola Literature Distribution, USA, 1995.

[2] Ohsawa, T., Kai, K., Murakami, K. "Optimizing the DRAM Refresh Count for Merged DRAM/Logic LSIs". In *Proc. of the International Symposium on Low Power Electronics and Design*, pp. 82-87, 1998.

[3] Panda, P., Dutt, D., Nicolau, A. "Exploiting Off-Chip Memory Access Modes in High-Level Synthesis". *Digest of Technical Papers, 1997 IEEE/ACM International Conference on Computer-Aided Design*, pp. 333-340, 1997.

[4] Park, C. Y., Shaw, A. C. "Experiments with a Program Timing Tool Based on Source-Level Timing Schema". *Computer*, 24(5):48-57, May 1991.

[5] Puschner, P., and Burns, A. "A Review of Worst-Case Execution-Time Analysis". Guest Editorial. *International Journal of Time-Critical Computing Systems*, 18(2/3), 115-128 (2000), Kluwer Academic Publishers, Boston, May 2000.

[6] Vrchoticky, A. "*The Basis for Static Execution Time Prediction*". Master's Thesis, Technische Universitaet Wien, Institut fuer Tech. Informatik, Vienna, Austria, April 1994.