Noname manuscript No.
(will be inserted by the editor)

Making DRAM Refresh Predictable

Balasubramanya Bhat and Frank Mueller

Received: Oct 31, 2010

Abstract Embedded control systems with hard real-time constraints require that deadlines are met at all times or the system may malfunction with potentially catastrophic consequences. Schedulability theory can assure deadlines for a given task set when periods and worst-case execution times (WCETs) of tasks are known. While periods are generally derived from the problem specification, a task's code needs to be statically analyzed to derive safe and tight bounds on its WCET. Such static timing analysis abstracts from program input and considers loop bounds and architectural features, such as pipelining and caching. However, unpredictability due to dynamic memory (DRAM) refresh cannot be accounted for by such analysis, which limits its applicability to systems with static memory (SRAM).

In this paper, we assess the impact of DRAM refresh on task execution times and demonstrate how predictability is adversely affected leading to unsafe hard real-time system design. We subsequently contribute a novel and effective approach to overcome this problem through software-initiated DRAM refresh. We develop (1) a pure software and (2) a hybrid hardware/software refresh scheme. Both schemes provide predictable timings and fully replace the classical hardware auto-refresh. We discuss implementation details based on this design for multiple concrete embedded platforms and experimentally assess the benefits of different schemes on these platforms. We further formalize the integration of variable latency memory references into a data-flow framework suitable for static timing analysis to bound a task's memory latencies with regard to their WCET. The resulting predictable execution behavior in the presence of DRAM refresh combined with the additional benefit of reduced access delays is unprecedented, to the best of our knowledge.

Keywords Real-Time Systems · DRAM · Worst-Case Execution Time · Timing Analysis · DRAM Refresh · Timing Predictability

1 Introduction

Dynamic Random Access Memory (DRAM) has been the memory of choice in most computer systems for many years for commodity and embedded systems ranging from 8-32 bit microprocessor platforms. DRAMs owe their success to their low cost

This work is partly supported by NSF grants CNS-0905181, CNS-0720496 and EEC-0812121.

Department of Computer Science,North Carolina State University,Raleigh, NC 27695-8206
E-mail: mueller@cs.ncsu.edu, phone: +1.919.515.7889, fax: +1.919.515.7925

combined with large capacity, albeit at the expense of volatility. There are many variants of DRAMs, such as Asynchronous DRAM, Synchronous DRAM (SDRAM), Double Data Rate (DDR) SDRAM etc. Each bit in a DRAM is stored in just one capacitor within the silicon. Due to the structural simplicity (one transistor and one capacitor per bit) of DRAMs, they can reach very high density resulting in large capacities. On the downside, just as commodity capacitors, these DRAM capacitors lose charge over a period of time. Thus, the data stored in DRAM is gradually lost unless the capacitors are recharged periodically. In contrast, Static RAM (SRAM) is more complex (uses four transistors and two load elements per data bit) and takes more space but does not require periodic refresh to retain the data.

DRAMs are typically organized as a small set of banks that maintain their states independently of each other. Each bank consists of multiple rows and columns. Reading data from a given DRAM row requires the old row to be first closed (precharge) at a cost of Row Precharge (tRP) delay and the new row to be opened (activate) at a cost of Row Access Strobe (tRAS) delay. Once the given row is opened, any column within that row can be accessed within a Column Access Strobe (tCAS) delay. An auto-refresh operation of a given row involves closing (precharge) the currently opened row (with a tRP latency) and opening (activate) the row being refreshed (with a tRAS latency). Thus, the total time taken by an auto-refresh is tRAS+tRP.

In a typical DRAM, refresh operations are triggered by an external control circuit that periodically sends refresh commands to the DRAM over the command bus. This method of refreshing DRAMs is called auto-refresh. If a CPU or a peripheral tries to access the DRAM when an internal refresh is in progress, such memory references will stall until the refresh operation is complete. A typical DRAM requires one refresh cycle every 15.6 μs [13]. During the refresh operation, the last opened row is closed before the refresh row is opened. The processor accesses the DRAM memory for fetching data and instructions in the event of a cache miss, but this fetch is stalled while DRAM auto-refresh is in progress. Thus, the response time of a DRAM access depends on the point of time memory is accessed by the processor relative to a DRAM refresh.

In general-purpose computer systems, the delay due to DRAM refresh has no impact on program correctness and little impact on performance. Embedded control systems deployed in safety-critical environments or process plants, on the other hand, generally impose more stringent timing predictability requirements that are prone to be violated due to DRAM refresh delays. From avionics over industrial (chemical or power) plants to automotive subsystems such as ABS, system correctness extends from the traditional input/output relationships to deadlines. In such systems, a missed deadline may result in system malfunction with potentially hazardous implications or even loss of life. Such systems are typically referred to as hard real-time systems as opposed to soft real-time systems where deadlines can occasionally be missed. Deadlines together with release times or task periods are then combined with an execution budget to assess the schedulability, *e.g.*, through utilization-based tests under rate-monotonic (RM) or earliest-deadline-first (EDF) scheduling [12].

Determining the execution budget by bounding the worst-case execution time (WCET) of a task's code is key to assuring correctness under schedulability analysis, and only static timing analysis methods can provide safe bounds on the WCET (in

the sense that no execution may exceed the WCET bound) [27]. Various methods and tools for static timing analysis have been developed for a variety of embedded platforms ranging from 8-bit to 32-bit microprocessors [28]. However, none of these techniques consider the effect of DRAM refreshes on WCET bounds. Hence, a statically derived WCET bound is only safe if augmented pessimistically with the cost of refresh delays, which is inherently difficult to calculate or even to tightly bound due to the asynchronous nature of refreshes combined with task preemption. In fact, unpredictability in execution times can be observed in tasks of hard real-time systems with DRAM. Atanassov and Puschner [2] discuss the impact of DRAM refresh on the execution time of real-time tasks. For their target configuration, they calculate the maximum possible increase of execution time due to refreshes to be 2.13%. Using their analytical method, we calculated the worst-case execution time for the target configurations we are using in our experiments to be about 2%. This method assumes that in the worst case every DRAM refresh is overlapped with a memory access. Hence, the refresh overhead is bounded by the number of refresh intervals that can occur during a job's execution multiplied by the refresh cost. However, we observe that this assumption is not valid in the presence of hardware or software preemptions due to preemptive scheduling. We show that in the presence of preemptions, the number of refreshes encountered by a task increases with the number of preemptions of that task. The objective of this work is to develop novel methods for real-time system design that eliminate unpredictability due to DRAM refreshes and to thereby eliminate the need to consider DRAM refreshes in WCET analysis. We further show that these methods reduce the power consumption in the DRAM by reducing the number of precharges without affecting data retention.

Contributions:

1. This paper gives a detailed analysis of the impact of DRAM refresh delays on the predictability for embedded systems with timing constraints and in particular hard real-time systems. It identifies the sources that affect response times when the accesses to DRAM by the processor are not synchronized with DRAM controller activity.
2. We show that most commonly used analytical method to estimate the delay due to DRAM refreshes on WCET of tasks is insufficient in the presence of preemptions.
3. Two novel approaches to mitigate the impact of DRAM refresh are developed. The basic idea behind both approaches is to remove the asynchronous nature of hardware DRAM refreshes. By modeling and realizing DRAM refresh as a periodic task in software and performing the refresh operations in burst mode, delays due to refreshes can be isolated from application execution. (i) The first method disables hardware auto-refresh in favor of a purely software-based refresh task. (ii) The second method combines hardware and software based approaches in a hybrid scheme. Here, the software initiates hardware refresh in burst mode at regularly scheduled and well-defined intervals for a fixed duration of time.
4. We show that these new methods also result reduced DRAM power consumption by about 5% due to a lower number of row precharges.

5. Both approaches to mitigate refresh unpredictability have been implemented on multiple hardware platforms, and the pros and cons of each in terms of performance, overhead and predictability are discussed and experimentally evaluated.
6. We develop a methodology for incorporating variable latency memory references into static timing analysis to bound a task's WCET. We formalize our methodology in a data-flow framework, provide same algorithms for their incorporation into static timing analysis at the intra-task level and derive the cost of memory-related preemption delays (MRPD) for schedulability analysis at the inter-task level.

These methods effectively *eliminate* DRAM auto-refresh unpredictability and have the additional benefit to actually *reduce* subsequent memory access delays that otherwise would be incurred under hardware auto-refresh.

Naïvely, disabling auto-refresh seems dangerous as DRAM-stored values would be lost if a software refresh is missed. In a hard (or mixed criticality) real-time system, however, any deadline miss of a hard real-time task also renders the control system faulty. It is thus paramount to *ensure* that deadlines are met for all hard real-time tasks, including but not limited to the DRAM refresh task, and we demonstrate that this objective can be met in practice.

The cost of a single asynchronous DRAM refresh is small relative to typical task periods. Accounting for this cost at the schedulability analysis level offline, however, turns out to be a daunting task. After all, variations in execution times of task make it notoriously difficult to constrain the point in execution where a refresh occurs dynamically. Our approach combines the advantages of improving DRAM performance, lowering its power consumption and providing flexible scheduling, notably not precluding future extensions for non-preemptive scheduling or scheduling with limited preemption points.

Variable latencies for memory references as a result of DRAM refresh are a problem not only in hard real-time systems. For example, Predator [1], a hardware approach to make SDRAM memory controllers predictable for refreshes, was originally motivated by a need for highly predictable memory latencies during high-definition television decoding, which falls into the domain of soft real-time multi-media. We argue that our methods are universally applicable and, in contrast to Predator, do not require costly hardware modifications.

This paper is structured as follows. Section 1 provides an overview and motivation for this work. Section 2 describes different methods for performing DRAM refreshes and their trade-offs. Section 3 presents the approach used in our paper. Section 4 elaborates on implementation details and the experimental framework. Section 5 presents and interprets experimental results. Section 7 contrasts our work approach with prior work. Section 8 summarizes our contributions.

2 DRAM Refresh Modes

Every row in a DRAM should be periodically refreshed in order to retain the data. Refresh may be accomplished for a given row of DRAM cells by presenting the corresponding row address in combination with asserting the row address strobe (RAS)

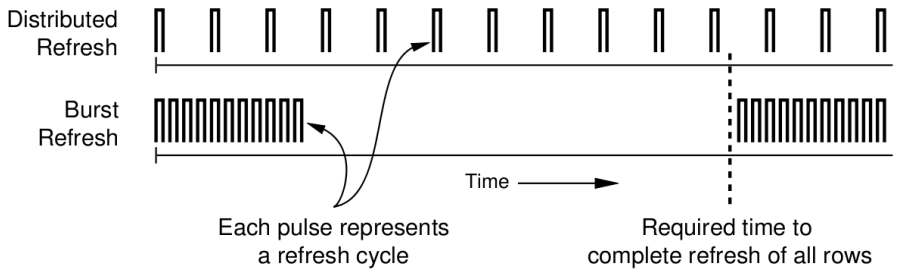

Fig. 1: Different DRAM Refresh Methods. See [13].

line. This method of refresh is called RAS Only Refresh (ROR). In this method, it is necessary for the hardware/software performing the refresh to keep track of which DRAM rows are refreshed and ensure that all rows of DRAMs are accessed within the specified refresh interval. Many modern DRAMs provide an alternative method to perform refreshes using a special command cycle. One example of such a special cycle is where the column address strobe (CAS) line is asserted prior to the row address strobe (RAS) line, commonly referred to as CAS-Before-RAS (CBR) refresh. When utilizing the CBR method to refresh DRAMs, it is not necessary for the hardware/software performing the refresh to track row addresses. Instead, we program the DRAM controller by sending a sufficient number (4096 in our case) of CBR cycles within the specified DRAM retention interval (64 ms in our case). The internal circuitry in the DRAM controller maintains a refresh counter per bank and refreshes successive rows for every refresh command until all rows within the bank have been refreshed. Depending on when refresh commands to successive rows are sent, we can classify a DRAM refresh scheme as either a distributed refresh or a burst refresh.

2.1 Distributed Refresh in Hardware

In this method of refreshing DRAMs, a single refresh operation is performed periodically, as illustrated in Figure 1. Once a full cycle of refresh is complete, it is repeated again starting from the first row. This is currently the most common method for refreshing DRAMs. Most memory controllers use this method to perform auto-refresh in hardware. However, this method causes the DRAM response time to vary depending on the relative time and the row numbers of memory accesses by the processor and the DRAM refresh.

It is well known that, whenever a DRAM reference by the processor is blocked by an auto-refresh operation, the processor has to wait for a delay bounded by tRAS+tRP. However, we make another key observation here, which has not been considered by related works and show that its omission leads to WCET bound violations that render hard real-time systems incorrect: *A DRAM refresh closes a previously opened row by the processor and opens up the new row being refreshed.* Hence, the next memory access by the processor, likely to the same row, now requires the refresh row to be closed and the old row to be reopened *at an additional cost of tRAS+tRP*.

Table 1: Delays due to Interfering Refresh Cycles

Old Row	Ref. Row	Next Row	Normal Delay	Delay w/ Refresh
n	n	n	tCAS	tRP+tRAS+tCAS
m	n	n	tRP+tRAS+tCAS	tRP+tRAS+tCAS
m	n	m	tCAS	tRP+tRAS+tCAS

Table 1 shows the additional delay suffered by the processor due to an intervening refresh in addition to the latency of the refresh operation itself. Depending on which rows are accessed by the processor and refresh operation, the memory exhibits different latencies. The "Old Row" column indicates the previously opened row in the DRAM. "Ref. Row" indicates the row being refreshed. "Next Row" indicates the next row being accessed by the processor following the refresh operation. "Normal Delay" is the delay suffered by the processor during the next memory access in the absence of any interim refresh operations. "Delay with Refresh" is the delay for next memory access due to an interim refresh operation, excluding the latency of the refresh operation itself. With an interim refresh operation, next memory access always takes tRP+tRAS+tCAS as the DRAM controller is not aware of which internal row is actually refreshed.

Thus, we see that a refresh operation not only delays by making the processor wait for a memory access for the duration of the refresh operation itself, but also causes additional delays for future accesses as rows need to be reopened.

Existing methods estimate the increase in WCET for tasks due to DRAM refresh using the formula given by Atanassov and Puschner [2], expressed as:

$$T_{WCET}^{refr} = T_{WCET} + \lceil \frac{T_{WCET}}{t_{Rint} - t_{delay}^{max}} \rceil \times t_{delay}^{max} \quad (1)$$

This formula computes the maximum number of refresh operations that can occur during a task's WCET and multiplies it with cost of each refresh. For example, if we consider a task with a WCET of 1000 μs, at the rate of one refresh for every 15.6 μs and 200 ns maximum refresh delay, there can be a maximum of 65 refresh intervals during the entire task execution, which increases the task's WCET to 1013 μs. Now consider that this task gets preempted by higher frequency tasks / interrupts every 100 μs, each running for 20 μs. This means that our 1000 μs task now runs in chunks of 80 μs each. There can be a maximum of 6 refresh intervals during this 80 μs period. Thus, during the total task execution time of 1000 μs, there can be a total of 76 refresh intervals. This increases the WCET of task to 1015.2 μs, which exceeds the 1013 μs bound from Eq. 1.

This example illustrates that the actual delay caused by refreshes in the presence of preemptions is dependent on the number of preemptions and the duration of each time slot during the task execution. Predicting the maximum number of preemptions due to interrupts, higher priority tasks and the interval between the preemptions is not a straight forward problem. In systems where DRAM and other peripherals share the same bus, it is necessary that all unrelated bus traffic cease during the entire period of a refresh operation to avoid contention, which is hard to model [2]. The time required

for a DRAM refresh in these systems thus degrades system performance not only from a memory availability standpoint, but also because of the time that the bus is unavailable during DRAM refresh, precluding other non-memory access bus traffic during that time.

As an example, the DSP platform we used in our experiments has 16MB DRAM space split across 4 banks that can be accessed and refreshed in parallel. Each bank (4 MB) has 4096 rows of 1KB requiring a total of 4096 refreshes. In distributed refresh mode, the refresh rate of a typical (and also this) DRAM is 15.6 μs with a duration of 150 ns for refreshing one row. Thus, the entire DRAM is refreshed once every 64 ms with a total total overhead of 614 μs for 4096 refreshes. This ratio of 1-2% overhead is typical for DRAM technology, but it does not yet take into account the overhead of additional tRP/tRAS delays for individual DRAM references that interfere with refreshes occurring in the background.

2.2 Burst Refresh in Hardware

An alternative method for performing DRAM refresh is called burst mode in which a series of explicit refresh commands are sent, one right after the other, until all rows have been refreshed. Once a sufficient number of refresh commands have been sent to refresh the entire DRAM, no more commands are sent for some time until the beginning of the next refresh period as illustrated in the Figure 1. The majority of the DRAM controllers do not readily support this mode of refresh in hardware. Even though we can configure them to send refresh commands one right after the other, they often lack the ability to stop and set up rates for subsequent bursts. Even when this mode is supported in hardware, refresh still interferes with task execution making task timings unpredictable as described earlier.

3 Our Approach

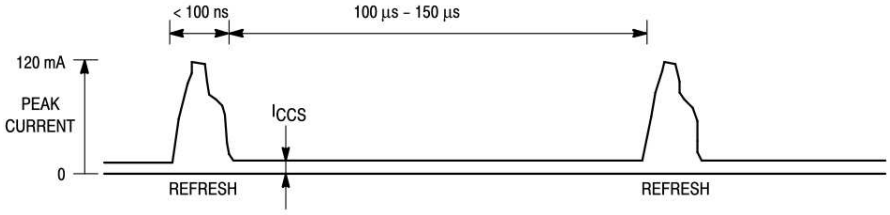

Fig. 2: Power supply current of a typical DRAM during refresh. See [15].

The basic problem with the hardware-controlled DRAM refresh is that the periodic refresh events generated by the DRAM controller and the memory access events generated by the processor are not synchronized with each other. Whichever event

comes later will have to wait for the former to complete. Also, an interleaving refresh operation delays the next memory access. The central idea of our approach is to remove the asynchronous nature of the two events and schedule the two events at predetermined time intervals so that they do not interfere with each other. Below, we describe two methods to perform DRAM refresh using this approach.

3.1 Software-Assisted Predictable Refresh

In this method, a new periodic task with a reserved time slot is created for performing the DRAM refresh in software. This task issues precharge operations of different rows in a back-to-back manner, *i.e.*, after one refresh completes, the next is started instantly. It makes use of RAS Only Refresh (ROR). To refresh one row of the DRAM memory using RAS Only Refresh, the following steps must occur:

First, the row address of the row to be refreshed must be applied at the address lines. The RAS line must switch from high to low while the CAS line remains high. Then, at the end of the required amount of time, the RAS line must return to high. The ROR refresh is not commonly supported in DRAM controllers. However, simply reading any word within some other DRAM row would have the same effect as refreshing the currently open row, except at a slightly higher cost of putting the column address (CAS latency) and fetching the word into the register. The DRAM controller tracks the currently open row in each of the DRAM banks and automatically issues the Activate (ACTV) command before a read or write to a new row of the DRAM. On systems with caches enabled, we have to ensure that our accesses reach the DRAM row by using cache bypass instructions (*e.g.*, atomics).

3.2 Hybrid Software-Initiated Hardware Refresh

The second method is a hybrid hardware/software method in which the period of the refresh interval is overwritten by reprogramming the DRAM control registers. Most DRAM controllers allow the refresh interval to be configured. Instead of requiring a fixed delay (15.6 μs in our example) between the two refresh cycles, we configure them to occur one right after the previous refresh cycle. Once this refresh period is configured and hardware refreshes are enabled, the refresh task just waits for a predetermined amount of time to let the entire DRAM be refreshed and then disables the refresh altogether. When the refresh task is waiting, the processor can be forced into a reduced power state or it can perform calculations not involving the DRAM. At the beginning of the next refresh period, the refreshes are re-enabled with zero delay between the refresh cycles and the same pattern is repeated. This behavior is similar to the hardware burst refresh method in terms of its timing diagram (see Figure 1). In contrast to the hardware-induced burst, our scheme starts and stops each burst of refreshes through software control at pre-scheduled intervals. No other tasks are executed when a refresh task is running. This can be ensured in a real-time system by assigning the highest priority to the refresh task or by disabling and re-enabling refreshes each time the refresh task is preempted and resumed respectively.

This method requires special handling of interrupts during a refresh burst in order to avoid interference. One option is to completely disable the interrupts during the entire duration of the refresh burst. Another option is to disable and re-enable refreshes upon entering and leaving the ISRs. Yet another option is to run the whole ISR from internal SRAM. Further, instead of sending one long burst, it is possible to send multiple smaller bursts more frequently as long as enough refresh commands are sent within the refresh threshold. This reduces the latency suffered by other tasks due to the refresh bursts. Once all rows have been refreshed, no refresh is going to occur until the next invocation (after the next release) of the refresh task. As seen earlier, the duration of one burst of refreshes (614 μs) is about 2 orders of magnitude smaller than the refresh period (64 ms). In between invocations of the refresh task, real-time tasks in the system execute with guaranteed absence of any interference from DRAM refreshes. This method makes use of the CAS before RAS (CBR) Refresh. The main difference between ROR and CBR refresh is the method for keeping track of the row address to be refreshed. With ROR, the refresh task must provide the row address to be refreshed. With CBR, the DRAM memory keeps track of the addresses using an internal counter.

3.3 Impact on DRAM Power Consumption

As we have seen earlier, performing a refresh operation on DRAM involves closing (precharge) the currently open row and activating the row being refreshed in every bank. The major source of power usage in a DRAM comes from these precharge operations. Figure 2 shows the power supply current during the refresh operation (essentially a row precharge) for a typical DRAM in self-refresh mode [15]. We observe that we can reduce the power consumed by the DRAM by reducing the number of precharge operations wherever possible. As we have seen, an intervening refresh operation closes the open row that is currently being accessed by the processor. When the processor tries to read this row again, it has to close (precharge) the new row and activate the old row that it wants to access. This costs an additional precharge operation and, hence, a small amount of additional power. In our approach, since all refreshes occur in bursts at pre-scheduled intervals, they do not interfere with other tasks in the system. This realization led us to conduct experiments on the power consumption of DRAMs under different methods explained in this paper. Section 5.4 discusses these results.

4 Implementation

We implemented our methods on three different embedded hardware platforms. The first platform is a TMS320C6713 DSP Starter Kit (DSK) module from Spectrum Digital. This board has a Texas Instruments TMS320C6713 DSP chip running at 225 MHz. This is a 32-bit processor with an advanced Very Long Instruction Word (VLIW) architecture, eight independent functional units that can execute up to 8 instructions per cycle, fixed and floating point arithmetic, 2 levels of caching, up to

256KB of on-chip SRAM, 512KB of flash memory, an on-chip DRAM controller and 16MB of SDRAM memory. We utilized the programming environment and the compiler supplied though the Code Composer Studio v3.1 from Texas Instruments.

The DSK uses a 128 megabit synchronous DRAM (SDRAM) on the 32-bit External Memory Interface (EMIF). Total available memory is 16 megabytes. The integrated SDRAM controller is part of the EMIF and must be configured in software for proper operation. The EMIF clock is configured in software at 90MHz. This number is based on an internal clock of 450MHz required to achieve 225 MHz operation with a divisor of 2 and a 90MHz EMIF clock with a divisor of 5. When using SDRAM, the controller must be set up to refresh one row of the memory array every 15.6 microseconds to maintain data integrity. With a 90MHz EMIF clock, this period is 1400 bus cycles. The second embedded platform features a Samsung AX4510 microcontroller board. The Samsung S3C4510B is a 32-bit ARM7 TDMI RISC processor design running at 50MHz clock speed. It also has 8KB of configurable on-chip SRAM/unified cache, an on-chip DRAM controller, 4MB of external Flash and 16MB of external SDRAM. We used the Keil Embedded Development Tools for development and testing on this platform. The S3C4510B provides a fully programmable external DRAM interface with four DRAM banks and auto-refresh mode for SDRAMs. It also provides control registers to configure the DRAM refresh mode, refresh timings, and refresh intervals. The third platform is an IBM PowerPC 405LP evaluation board used exclusively for the SDRAM power measurement experiments. This supports Dynamic Voltage and Frequency Scaling (DVFS) where voltage and frequency can be scaled in software *via* user-defined operation points ranging from 266 MHz at 1.8V to 33 MHz at 1V. There are four PC-133 compatible SDRAM memory modules (1M x 32b x 4 internal banks, 128Mb, non-ECC). The four 16-MB modules are arranged to provide 64MB of total SDRAM memory. More details about this platform are given later in the section 5.4.

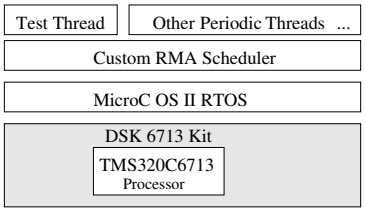

Fig. 3: System Architecture

Figure 3 depicts the layered system software architecture utilized in our experiments on the embedded platform. Our implementation effort includes porting a commonly used real-time operating system, Micro C/OS-II [9], which has a fixed priority preemptive scheduling. We then implemented a rate monotonic (RM) scheduler [11] on top of Micro C/OS-II with novel support for creating and running periodic threads of arbitrary periods imposing strict execution-time control within each period. If a thread does not complete execution by its deadline, it is preempted and rescheduled

during the next period by the scheduler. This scheduler is also capable of monitoring the total execution time of each task with a precision of 4 CPU clock cycles excluding the time spent inside interrupt service routines and the scheduler overheads. We utilized the same system software architecture on all three hardware platforms discussed earlier. Micro C/OS-II and our custom RM scheduler are ported to these platforms and provide the same APIs to applications on all platforms. Hence, the test application can be run on different platforms with minimal modifications. In addition, we also run experiments on Linux on the PowerPC Platform (more details can be found in later sections). All programs were written in C and assembly language on these platforms.

5 Results

We performed several experiments on the embedded platforms discussed in the previous section. This section describes these experiments and the results.

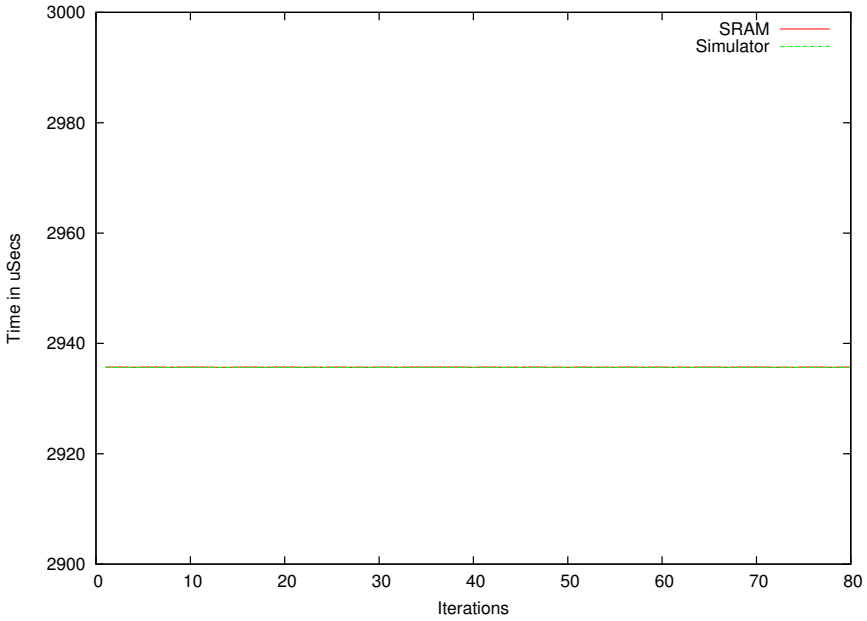

Fig. 4: Memory Latencies on DSP Platform using SRAM / Simulator. Time Range [2900-3000] μs.

5.1 Unpredictable Timings under Hardware DRAM Refresh

First of all, we wanted to assess the effect of DRAM refresh cycles on the timing of application tasks, in a controlled environment with a real workload and selectively

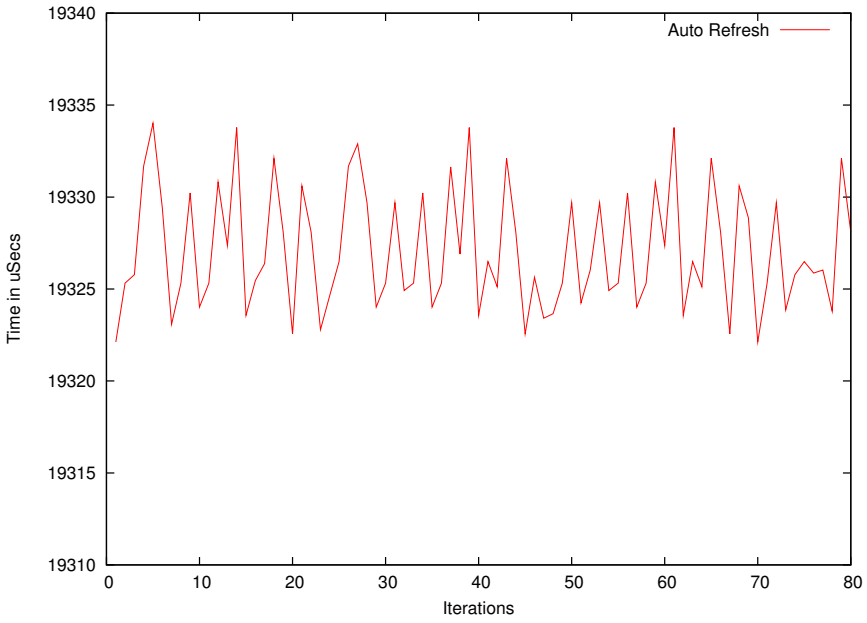

Fig. 5: Memory Latencies on DSP Platform using SDRAM. Time Range [19310-19340] μs. Software and Hybrid Values are so close that the Lines cover each other in the Graph.

disabled caches. We used a bubblesort algorithm to sort 100 integer elements as a test load. We disabled the instruction and data cache in all these experiments in order to force memory accesses for *every* instruction, which provides better control over the experiment. The test load function is run inside a periodic task of 400 ms period.

We executed this workload in three different scenarios. First, we ran the load using SRAM memory. For this purpose, we modified the linker script to place all code, data and stack segments in the on-chip SRAM. The on-chip SRAM has a very low latency and, unlike DRAM, does not require periodic refreshing. Figure 4 shows the result of execution using SRAM. We can see that the workload takes exactly the same amount of time in every iteration. This is because the on-chip SRAM does not need periodic refreshing and exhibits uniform latency every time it is accessed.

Next, we executed the same load on a TI cycle-accurate device simulator for the TMS320C6713 processor. The measured times matched those for SRAM-based execution as seen in Figure 4.

Finally, we ran the same workload on SDRAM memory, which requires refresh. The SDRAM controller is initialized using the default configuration script supplied along with the 6713DSK kit. This causes the SDRAM controller to send one auto-refresh command every 15.6 μs. Every time an auto-refresh command is received, the SDRAM internally refreshes one row in every bank. The 6713DSK SDRAM has four banks, 4096 rows per bank and 1024 bytes per row. Thus, the SDRAM requires 4096 refresh commands to refresh the entire memory. The entire SDRAM is refreshed approximately every 64ms. With the SDRAM controller correctly configured to retain

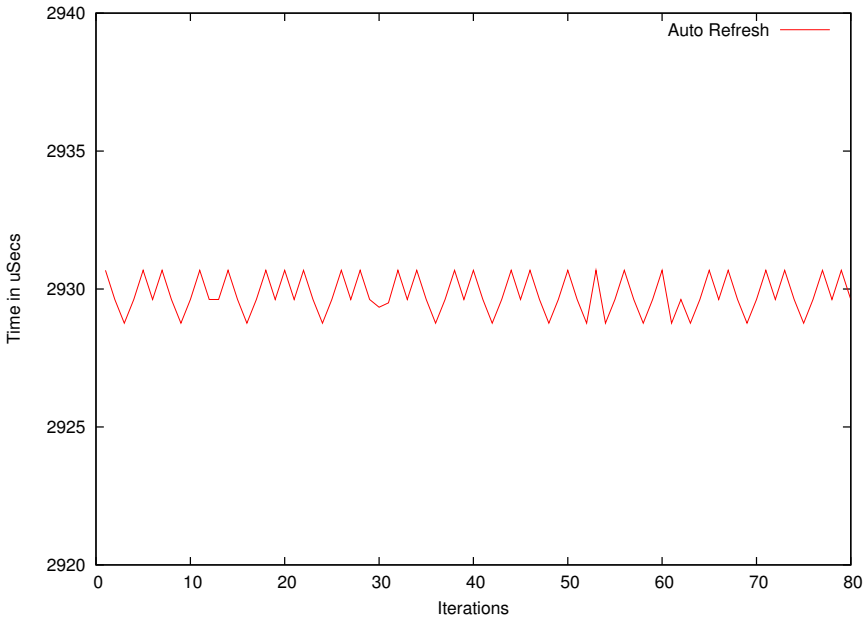

Fig. 6: Memory Latencies on ARM Platform using SDRAM. Time Range [2920-2940] μs. Software and Hybrid Values are so close that the Lines cover each other in the Graph.

the data, we modified the linker script to place all code, data and stack in SDRAM. The earlier workload is run again with this setup and the resulting times are shown in Figures 5 and 6 for the DSP and ARM platforms, respectively.

We make the following observations from the graphs. First of all, the measured times for SDRAM are much higher than for SRAM. This is caused by the higher latency of SDRAM compared to the on-chip SRAM. (Of course, SRAM is much more costly and significantly smaller than DRAM so that many embedded systems utilize DRAM in practice.) Secondly, the measured values are very jittery in nature. There are mainly two reasons for variations in DRAM response times. First of all, every time a new row is accessed within a bank, the SDRAM needs to close the current row and open the new row, which requires tRP + tRAS time. However, since we access the same set of memory addresses in every iteration of the workload, this cannot be the reason for variations in the graph. The same conclusion is is also true for the CAS latency (tCAS) that is incurred every time we access different columns of the same row.

Secondly, the SDRAM auto-refresh cycles occurring during the workload execution also cause variations in the time taken. As we saw earlier, one auto refresh occurs every 15.6 μs and each refresh takes time for one Row Access Strobe (tRAS) and Row Precharge (tRP). The refresh happens asynchronously from the point of view of program execution. The processor needs to access memory when it has to fetch a new instruction or data. The worst-case delay occurs when there is a refresh in progress every time the processor accesses the memory. The best-case behavior

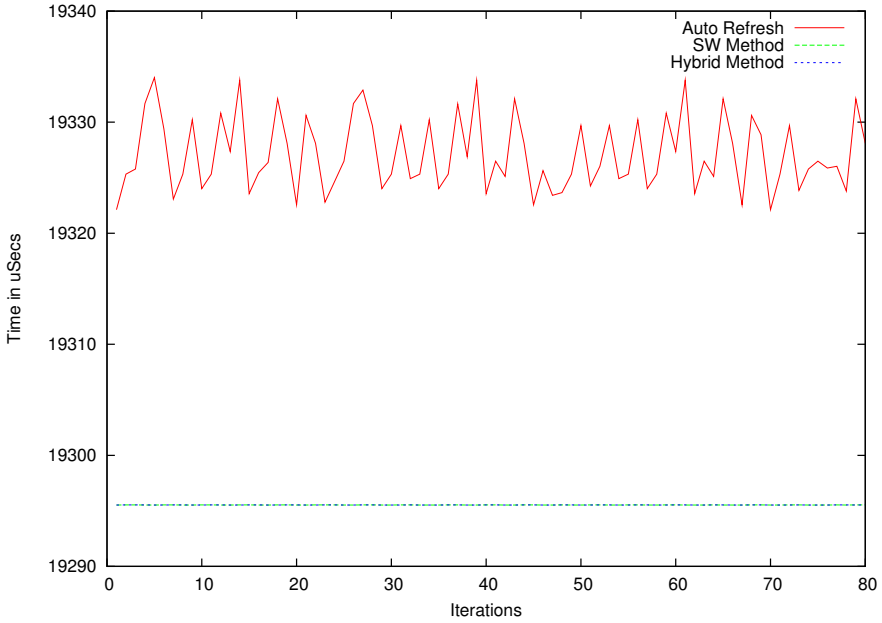

Fig. 7: Memory Latencies for DRAM Refresh Techniques on DSP Platform in Time Range [19,290-19,340] μs

occurs when the processor is not accessing memory whenever there is a refresh in progress. Further, an intervening refresh cycle could close the currently opened row by the processor causing the next memory access by the processor to take longer. This causes the observed degree of unpredictability in the measured time as shown in Figures 5 and 6.

5.2 Software-Assisted Predictable Refresh

Next, we modified the code to disable SDRAM auto-refresh and created a separate periodic task to refresh all the SDRAM rows in software, as described in Section 3. The refresh task was created with a periodicity of 10ms. It refreshes a subset of rows in each period, such that the entire SDRAM is refreshed within 60ms. Thus, when the workload thread runs, it never has to wait due to an auto-refresh in progress. Figure 7 shows the results for this configuration. As can be seen from this graph, the measured times with software-assisted refresh (second line from the top) in all intervals are *uniform*. Also, the average time is *less* than that with hardware-based SDRAM auto-refresh. This is because task execution is never interrupted by an asynchronous auto-refresh. This graph illustrates that, by delegating the SDRAM refresh responsibility into a dedicated periodic task, other real-time tasks in the system become isolated from the erratic latency response of the SDRAM due to auto-refresh.

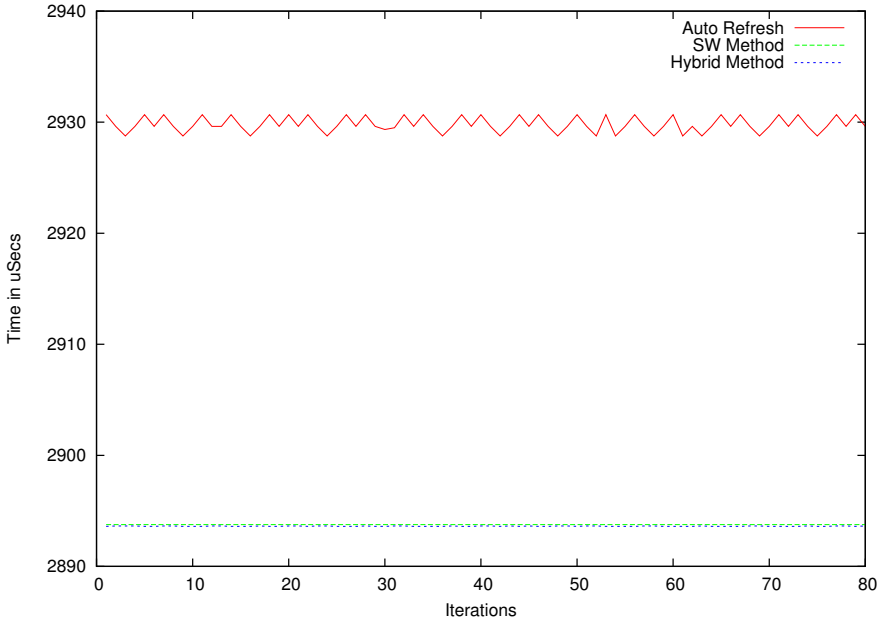

Fig. 8: Memory Latencies for DRAM Refresh Techniques on ARM Platform in Time Range [2,890-2,940] μs

Performing the SDRAM refresh in software instead of hardware has some additional overheads. First of all, in hardware, for every auto-refresh command sent by the controller, one row is refreshed in all the banks in parallel. Since we have to explicitly read one word from every row during software refresh, we cannot take advantage of this bank parallelism. Thus, instead of sending 4096 auto-refresh commands, we now need to access 4×4096 rows to entirely refresh the SDRAM. Also, sending a hardware auto-refresh command does not utilize the address and data buses, nor does it bring any data into cache. However, in a software refresh, since we explicitly read one word from every bank, we potentially evict data from cache. This can be avoided in many embedded processors that support cache-bypass load instructions. In the absence of bypass support, such evictions can be modeled in static cache analysis to bound WCETs and cache-related preemption delays [10, 16, 21, 23, 24]. Nonetheless, any memory access requires bus bandwidth. Because of these overheads, the total time spent for a software refresh is larger than that of a hardware auto-refresh. In our experiments on the DSP platform, we measured that the software refresh task, as described above, takes about 16% of the processor time compared to a maximum of 4-5% overhead in hardware auto-refresh (due to blocked DRAM accesses, see Section 2.1). Yet, we not only increase predictability but also performance of all other tasks in the system by a moderate 2.8% to 0.16% (for smaller and larger workloads, respectively, where the latter is shown in Figure 7) due to absence refresh-incurred delays on memory references.

We also ran the same experiments on our ARM platform using bubble sort on 50 elements. The results are shown in the Figure 8. Since this platform has a much slower processor frequency (50MHz), the difference in the speed of DRAM and the processor is less significant compared to the TI DSP platform. Hence, we only observe minor variations in the measured times due to background DRAM refreshes. Again, the software-based refresh technique has succeeded in producing uniform timing values.

In an effort to improve the performance of software refresh, we implemented a set of optimizations. One optimization is that instead of accessing every successive row one after the other linearly, we access one row from each of the four banks and then move to the next row. It is also important to make the four load instructions from four banks independent of each other. This enables the SDRAM to pipeline these loads. A read in progress on one bank will not block the read in other banks. With these optimizations in place, we measured the overhead of the refresh task to be about 12%, which is lower than in the earlier case, but still more than the hardware auto-refresh overhead.

5.3 Hybrid Software-Initiated Hardware Refresh

Our second method improves the performance further by utilizing the hardware refresh capabilities in a different manner. In this approach, we initially disable the hardware auto-refresh. A periodic refresh task is again created with a 10ms period. During each period, this task first enables the hardware auto refresh when invoked. But instead of configuring for one refresh for every 15.6 μs, the SDRAM controller is configured to send successive refresh cycles back-to-back without any delay. The refresh task waits for a calculated amount of time after which it disables the auto-refresh. The refresh task is allowed to run for a predetermined amount of time in each period, such that the *entire* SDRAM is safely refreshed, *e.g.*, within 60ms in case of the TI DSP. Since this method uses the hardware auto-refresh, all banks can be refreshed *in parallel*, which implies that no data is actually transferred between memory and processor, *i.e.*, caches remain completely unaffected. Figures 7 and 8 show the measured times for this hybrid approach. In Figure 7, the lines for the hybrid and software approaches coincide with each other as measured cycle times are identical for both approaches. As can be seen from the graphs, the measured times are constant for both the DSP and ARM platforms. We measured the processor overhead of the refresh task to be about 9%, which is significantly lower compared to the non-optimized software refresh approach.

There is one problem with the current implementation of this approach. Most of the SDRAM controllers do not track how many auto-refresh cycles they have sent. Because of this, the refresh task cannot determine exactly when it has completed sending the required number of refresh cycles. The only method to mitigate this problem is to allocate time for the refresh task in excess of worst-case refresh time. We also plan to prototype a modification to the SDRAM controller on an FGPA to track the exact number of auto-refresh cycles it has sent. The refresh task can use this information to more accurately time the auto-refresh. The SDRAM controller can be

made more intelligent to support this method. A similar approach is also possible for DRAMs that support monitoring of the leakage discharge on a per-row basis.

5.4 Reduction in DRAM Power Consumption

Our experiments to assess the power saving potential of the new refresh method is performed on an IBM PowerPC 405LP evaluation board [17]. This board is customized for conducting power-related experiments with support for Dynamic Voltage and Frequency Scaling (DVFS). Voltage and frequency can be scaled in software *via* user-defined operation points ranging from 266 MHz at 1.8V to 33 MHz at 1V. There are four PC-133 compatible SDRAM memory modules (1M x 32b x 4 internal banks, 128Mb, non-ECC). The four 16-MB modules are arranged to provide 64MB of total SDRAM memory. Individual SDRAM memory modules are arranged on the evaluation board in "banks" to improve throughput. In this board design, four modules with four banks are installed such that each module containing four internal banks. This board has several probe points that enable us to measure the voltage and current of all SDRAM modules — independently of the remaining components, such as processor and I/O. We used an analog data acquisition board to measure the voltage and current supplied to the SDRAM modules. A real-time earliest deadline first (EDF) scheduling policy was implemented as part of a user-level threads package under the Linux operating system running on the board. A suite of task sets with synthetic CPU workloads was utilized, similar to the task set of pattern one in DVSleak [29] with aggressive dynamic frequency and voltage scaling (DVFS) enabled based on feedback. Each task set comprises ten independent periodic tasks whose WCET is in the range of 1ms to 100ms. The task set is designed to allow user control of the CPU/memory load so that we can study the SDRAM power consumption at various load points. On this platform, we implemented the refresh task within a Linux kernel module. When this module is loaded, it disables auto-refresh by programming the DRAM Controller and starts a task with a period of 10 ms responsible for DRAM refreshes. At every invocation, this task refreshes a subset of rows using our hybrid refresh method. Figure 9 compares the power consumption of the SDRAM at different load points (from 10% to 90%) between our approach and hardware auto-refresh. As can be seen from the graph, our approach always consumes less power than the hardware auto-refresh method. For the memory subsystem, we obtained about 5% power savings on the average for the same amount of work in a fixed period of time. Power savings are linear to execution time here as we do not exploit the DVFS of the board. We suspect that operating system noise caused the power to be the same for 30% and 50% utilization (for the hybrid method). This prompted us to verify power savings on the board by replacing Linux with Micro C/OS-II.

We ported Micro C/OS-II and the RMA scheduler described earlier onto this platform to conduct DRAM power experiments in a tightly controlled RTOS environment. A periodic refresh task is created with a 10 ms period to perform DRAM refreshes using our hybrid scheme. The processor is configured to run at a frequency of 266.6 MHz/1.8V without DVFS. The instruction and data caches were disabled as in our earlier experiments. The idle loop is configured to execute out of internal

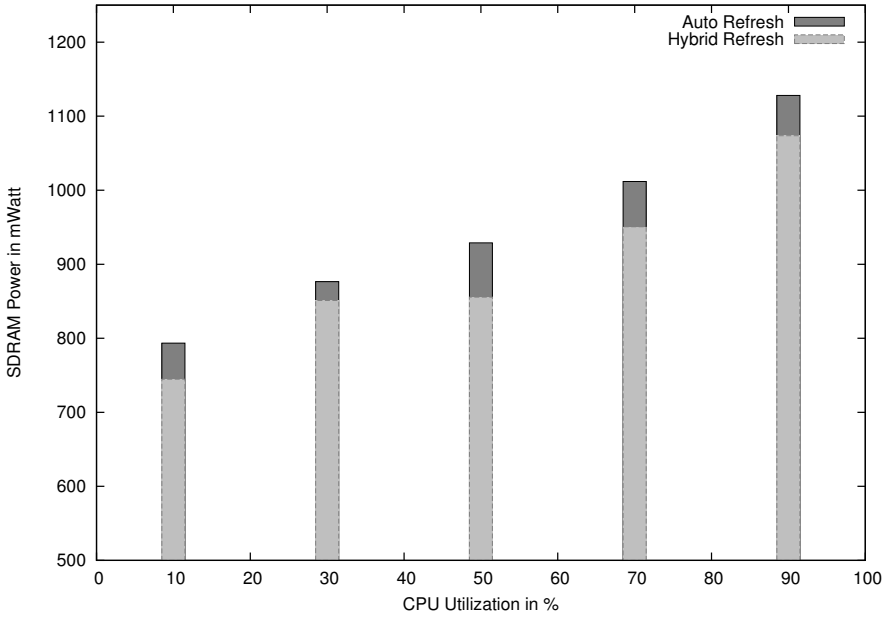

Fig. 9: Power comparison between different refresh methods under Linux. Range [500-1200] mWatt

SRAM in order to avoid DRAM accesses when the CPU is idle. The CPU load is controlled by changing the period and execution time of the test task. We assume that the number of DRAM accesses is proportional to the CPU load when cache is not enabled. We used the same bubblesort algorithm to sort 100 integer elements as a test load. We then measured the SDRAM power consumption under both hardware auto-refresh and hybrid refresh schemes, the results of which are shown in Figure 10. These results indicate that the hybrid refresh scheme reduces the SDRAM power consumption for the memory subsystem by up to 2.5% over different CPU utilizations. Notice that this experiment did not exploit aggressive DVFS with feedback (DVSleak [29]) as DVFS capabilities are not available under Micro C/OS-II. Instead, the processor was running at full clock speed (266 MHz) all the time. Hence, power consumption in Figure 10 is expected to be higher than that with DVFS in Figure 9.

6 WCET Analysis for Variable Memory Latencies

Our approach to handle DRAM refresh in software or as a hybrid approach provides a clean separation between refresh-induced latencies and the executing real-time application tasks. Only the dedicated software refresh task is subject to refresh delays, which are incorporated into the real-time schedule and are subject to the same schedulability tests as any other task (albeit at higher priority as interrupts remain disabled during a refresh, see Section 3.2).

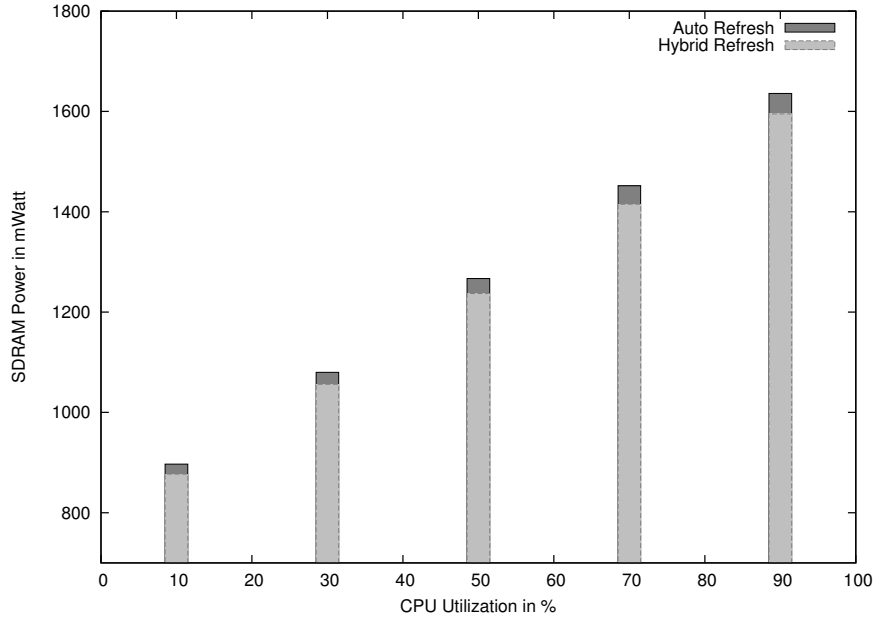

Fig. 10: Power comparison between different refresh methods under Micro C/OS-II. Range [700-1800] mWatt

All remaining real-time tasks now experience predictable access times when referencing memory as discussed in the following. A memory reference is unconditionally subject to a tCAS latency. Conditionally, if the reference refers to a different row than was previously accessed on a given memory bank, a tRAS latency is imposed. Yet, applications no longer suffer from precharge (tRP) delays or unpredictable tRAS overheads due to interrupt-triggered hardware refreshes as reported in Table 1.

These constraints can be expressed as data-flow equations of the control-flow structure of a function within the code of a task. Let $CFG(f) = (s, V, E)$ be the control-flow graph of function f, where V is the set of basic blocks (vertices), E is the set of control-flow transitions (edges) $v \rightarrow w$ for each branch from $v \in V$ to $w \in V$ and $s \in V$ is the start vertex of f. Let B be the set of memory banks of the DRAM. We then define the set of defined, def, and killed, $kill$, rows per memory bank $b \in B$ based on the mapping of a memory reference m to a row $r(m)$ and the corresponding bank $b(r(m))$. (We say m is a memory reference if a corresponding instruction or data reference in the code results in cache misses at all levels — or *may* result in cache misses according to may analysis [3, 16]).

Definition 1 The set of rows defined in a basic block $def(v)$ for $v \in V$ is $r(m)$ for the last memory reference in v for any bank $b(r(m))$ that row $r(m)$ maps to.

Definition 2 Conversely, the set of rows killed in a basic block $kill(v)$ is the inverse piece-wise set of all rows that map into same bank as the corresponding row in $def(v)$ or simply

$$kill(v) = \bigcup_{r \neq d} b(r) = b(d) \texttt{ where } d \in def(v)$$

Definition 3 We can then define the data-flow sets incoming to $in(v)$ and outgoing from $out(v)$ a block v relative to its predecessor blocks $preds(v)$ where $preds(v) = \bigcup_{p \in V} p \rightarrow v \in E$ and $in(s) = \emptyset$.

$$in(v) = \bigcap_{p \in preds(v)} out(p)$$
$$out(v) = in(v) \cup def(v) \setminus kill(v)$$

This set of data-flow equations can then be solved iteratively over the control-flow graph until it converges (reaches a fixpoint). An outline of an algorithmic framework is depicted below in Algorithm 1.

Algorithm 1 Iterative Data-flow Algorithm

```
FOREACH block v
  FOREACH bank b
    in[v,b] = φ

REPEAT
  converged = true
  FOREACH block v
    oldin[v]=in[v]
    LET p0 ∈ preds(v), in[v] = out[p0]
    FOREACH p ∈ preds(v) \ p0
      FOREACH bank b
        in[v,b] = in[v,b] ∩ out[p,b]
    IF oldin[v] ≠ in[v]
      converged = false
    out[v]=def[v] ∪ (in[v] ∩ kill[v])
UNTIL converged
```

The resulting sets are then suitable for determining the memory latencies of individual memory references within a basic block in a sequential manner. It can thus readily be incorporated into static timing analysis to determine the WCET of individual instructions, basic blocks, functions and eventually entire tasks (see Algorithm 2). In fact, it can be performed after static cache analysis once it can be determined which instruction and data references result in cache misses through all cache levels.

This approach allows the derivation of safe but tight bounds on the WCET of a task. This follows an intra-task model where WCET is first bounded separately for each task before inter-task effects, such as cache-related preemption delay (CRPD),

Algorithm 2 WCET Calculation of Access Strobes

```
FOREACH block v
  openrows=in[v]
  FOREACH mem ref m (instr/data cache miss at all levels)
    wcet[v] = wcet[v] + tCAS
    IF openrows[b(r(m))] ≠ r(m)
      openrows[b(r(m))] = r(m)
      wcet[v] = wcet[v] + tRAS
```

are taken into account at a higher level. These inter-task costs are considered at the level of schedulability analysis. For DRAM memories, there is also a memory-related preemption delay (MRPD). This delay stems from the fact that preemption may result in opening another row on a memory bank before resuming a preempted task at a point where an open row prior to preemption was modeled to be reused by a subsequent reference in the absence of preemption (intra-task analysis). Multiple approaches can be utilized to bound the MRPD.

At a coarser granularity, one may assume that there is no row reuse for any banks across preemptions. Alternatively, in a fine-grained approach, one models the intersection between row usage of different tasks to determine if a subset of rows outside this intersection can be retained across preemptions. These options are equivalent to CRPD modeling of an entire cache invalidation vs. invalidation of only those lines in the intersection between two tasks [20, 23, 24]. Since caches have a large number of lines, the fine-grained approach can give significantly better results, particularly when the points of preemption are constrained as well [21, 22].

The DRAM model differs significantly from caches in that the number of memory banks tends to be small (in the order of 1-8) while the number of rows is large (in thousands). Hence, a preemption statistically results in new rows to be opened on all banks with a very high probability. This indicates that the coarse-grained model is sufficient to model the MRPD effect. In other words, a preemption adds an overhead of $\gamma = |rows| \times t_{RAS}$ to a task's execution time.

7 Related Work

Past work on DRAM refresh focuses on hardware principles, such as refresh methods (mostly in patents), power enhancements, fault tolerance support or discharge monitoring [4–8, 18, 25]. One exception is the work by Moshnyaga *et al.* that utilizes operating system facilities to trade off DRAM vs. flash storage to mitigate current differences in access latencies, bandwidth and power consumption [14]. In contrast to our work, theirs does not address refresh side-effects on predictability. Our work is rather in the spirit of prior work on increasing the predictability of hardware peripherals for real-time software, such as bus-level I/O transaction control [19]. Another related work called RAPID [26] proposes retention-aware placement in DRAM, a novel software method that can exploit off-the-shelf DRAMs to reduce refresh power to vanishingly small levels approaching non-volatile memory. However, this method is not designed to address the predictability problem of real-time systems. Preda-

tor [1] is a predictable SDRAM memory controller using a hardware-based approach to achieve a guaranteed lower bound on efficiency and an upper bound on the latency in the presence of SDRAM refreshes and multiple users sharing the same SDRAM. In contrast, our approach is fully software based and succeeds in completely eliminating the unpredictability due to DRAM refreshes. Predator was motivated by a lack of predictability of a soft real-time application area in the multi-media domain, namely real-time processing for high-definition television during decoding but is applicable to hard real-time as well, just as are our methods.

8 Conclusion

In this paper, we examined the effect of DRAM refreshes on the predictability of real-time tasks. We proposed two novel methods to increase predictability of hard real-time systems in the presence of DRAM refreshes, namely (1) a software-assisted refresh and (2) a hybrid software-initiated hardware refresh. Both methods were implemented and evaluated on two embedded platforms. Experimental results confirmed that both methods result in predictable DRAM accesses without additional refresh delays. We further formalize the integration of variable latency memory references into a data-flow framework suitable for static timing analysis to bound a task's memory latencies with regard to their WCET. We further discussed the cause of overheads for DRAM accesses with respect to our methods. In the future, additional optimizations could be applied to these methods. We are pursuing FPGA-based modifications to a DRAM controller to add native support for burst refreshes in hardware. The burst refresh time can be overlaid with non-memory based activities, such as in-core computation, I/O operations, or memory accesses from other on-chip memory devices. Overall, our new methods alleviate the unpredictability of DRAMs due to refreshes, which facilitates the design of hard real-time systems with DRAMs in an unprecedented manner.

References

1. B. Akesson, K. Goossens, and M. Ringhofer. Predator: a predictable sdram memory controller. In *CODES+ISSS '07: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 251–256, New York, NY, USA, 2007. ACM.
2. P. Atanassov and P. Puschner. Impact of dram refresh on the execution time of real-time tasks. In *Proc. IEEE International Workshop on Application of Reliable Computing and Communication*, pages 29–34, Dec. 2001.
3. C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2/3):131–181, Nov. 1999.
4. M. Ghosh and H.-H. S. Lee. Smart refresh: An enhanced memory controller design for reducing energy in conventional and 3d die-stacked drams. In *MICRO*, pages 134–145, 2007.
5. S. Hellebrand, H.-J. Wunderlich, A. A. Ivaniuk, Y. V. Klimets, and V. N. Yarmolik. Error detecting refreshment for embedded drams. In *VTS*, pages 384–390, 1999.
6. Y. Katayama, E. J. Stuckey, S. Morioka, and Z. Wu. Fault-tolerant refresh power reduction of drams for quasi-nonvolatile data retention. In *DFT*, pages 311–318, 1999.
7. C.-K. Kim, B.-S. Kong, C.-G. Lee, and Y.-H. Jun. Cmos temperature sensor with ring oscillator for mobile dram self-refresh control. In *ISCAS*, pages 3094–3097, 2008.

8. C.-K. Kim, J.-G. Lee, Y.-H. Jun, C.-G. Lee, and B.-S. Kong. Cmos temperature sensor with ring oscillator for mobile dram self-refresh control. *Microelectronics Journal*, 38(10-11):1042–1049, 2007.
9. J. Labrosse. *Micro C/OS-II*. R & D Books, 1998.
10. C. Lee, J. Hahn, Y. Seo, S. Min, R. Ha, S. Hong, C. Park, M. Lee, and C. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. In *IEEE Real-Time Systems Symposium*, Dec. 1996.
11. C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. of the Association for Computing Machinery*, 20(1):46–61, Jan. 1973.
12. J. Liu. *Real-Time Systems*. Prentice Hall, 2000.
13. Micron Technology, Inc. *Various Methods of DRAM Refresh. Technical Note TN-04-30.*, 1999.
14. V. G. Moshnyaga, H. Vo, G. Reinman, and M. Potkonjak. Reducing energy of dram/flash memory system by os-controlled data refresh. In *ISCAS*, pages 2108–2111, 2007.
15. Motorola. *DRAM Refresh Modes (Order No. AN987/D)*, 2004. http://examenesutn.awardspace.com/examenes/tecdigi2/apuntes //Memorias/refresh.pdf.
16. F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2/3):209–239, May 2000.
17. K. Nowka, G. Carpenter, and B. Brock. The design and application of the powerpc 405lp energy-efficient system on chip. *IBM Journal of Research and Development*, 47(5/6), September/November 2003.
18. K. Patel, E. Macii, M. Poncino, and L. Benini. Energy-efficient value based selective refresh for embedded drams. *J. Low Power Electronics*, 2(1):70–79, 2006.
19. R. Pellizzoni, B. D. Bui, M. Caccamo, and L. Sha. Coscheduling of cpu and i/o transactions in cots-based embedded systems. In *IEEE Real-Time Systems Symposium*, pages 221–231, 2008.
20. H. Ramaprasad and F. Mueller. Bounding worst-case data cache behavior by analytically deriving cache reference patterns. In *IEEE Real-Time Embedded Technology and Applications Symposium*, pages 148–157, Mar. 2005.
21. H. Ramaprasad and F. Mueller. Bounding preemption delay within data cache reference patterns for real-time tasks. In *IEEE Real-Time Embedded Technology and Applications Symposium*, pages 71–80, Apr. 2006.
22. H. Ramaprasad and F. Mueller. Tightening the bounds on feasible preemption points. In *IEEE Real-Time Systems Symposium*, pages 212–222, Dec. 2006.
23. J. Staschulat and R. Ernst. Multiple process execution in cache related preemption delay analysis. In *International Conference on Embedded Sofware*, 2004.
24. J. Staschulat, S. Schliecker, and R. Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *Euromicro Conference on Real-Time Systems*, 2005.
25. T.-H. Tsai, C.-L. Chen, C.-L. Lee, and C.-C. Wang. Power-saving nano-scale drams with an adaptive refreshing clock generator. In *ISCAS*, pages 612–615, 2008.
26. R. Venkatesan, S. Herr, and E. Rotenberg. Retention-aware placement in dram (rapid): software methods for quasi-non-volatile dram. In *International Symposium on High Performance Computer Architecture*, pages 155 – 165, Feb. 2006.
27. J. Wegener and F. Mueller. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-Time Systems*, 21(3):241–268, Nov. 2001.
28. R. Wilhelm, J. Engblohm, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenstrom. The worst-case execution time problem — overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):1–53, Apr. 2008.
29. Y. Zhu and F. Mueller. Dvsleak: Combining leakage reduction and voltage scaling in feedback edf scheduling. In *ACM SIGPLAN Conference on Language, Compiler, and Tool Support for Embedded Systems*, pages 31–40, June 2007.