RG-EDF: An I/O Scheduling Policy for Flash Equipped Sensor Devices

Adam Ji Dou and Vana Kalogeraki

Dept. of Computer Science and Engineering
University of California, Riverside, CA 92521
{ jdou, vana }@cs.ucr.edu

Abstract. Flash equipped sensor devices are becoming increasingly complex and are now capable of supporting real-time multiple applications on a single sensor, rich sensing of visual and audio data, and storage of large amounts of data. With this increase in complexity, it is no longer sufficient to provide first in first out (FIFO) type capture of data into more persistent memories. In this paper we propose RG-EDF, a new scheduling policy for flash equipped sensor devices. RG-EDF aims at providing QoS support to multimedia tasks by considering the unique characteristics of flash-based devices. We have implemented our scheme on a CC1010 sensor node with a SD flash card attached and compared our technique to other popular scheduling policies. Our experimental results show the working and benefit of our system.

1 Introduction

Wireless Sensor Networks (WSNs), composed of small, low cost and low power sensor devices, have found popular applications in many situations including environmental monitoring, military surveillance, inventory tracking and seismic control. Typical sensor devices feature a low-frequency, low-power processor (≈4-58MHz), limited memory (4-10KB), a wireless radio for communication, on-chip sensors, and an energy source such as a set of AA batteries or solar panels.

The introduction of flash equipped sensors (RISE [4], PRESTO [11]) has significantly enhanced the storage capacity of sensors; it allows storing large amounts of data locally. Since the power consumed by transmitting data is a magnitude higher [17] than storing it locally, the sensors can now store and process data, sending only relevant information to the sink in response to preset conditions or queries. Such in-network data storage provides a significant reduction in energy usage and a corresponding increase in the lifetime of the WSN.

As flash-based devices become increasingly popular, they are required to perform real-time tasks, such as storage and retrieval of multimedia data. Today, visual (Cyclops [15], CMUcam [2]) and audio (EnviroMic [13]) sensors enable the sensing of high bandwidth, rich data; providing much more information than simple scalar measurements of temperature, humidity, etc [10]. This rich data can supplement the simple measurements with more complete context information. Multimedia data are different from traditional file systems, as they require large storage capacity and must support intensive real-time I/O traffic. It is common for data generated by the tasks to be sequentially positioned on the flash, while data generated concurrently by different tasks is multiplexed. However, such sequential storage can lead to significant latencies (as we show in the paper), thus limiting the ability to meet satisfy the real-time requirements of the tasks.

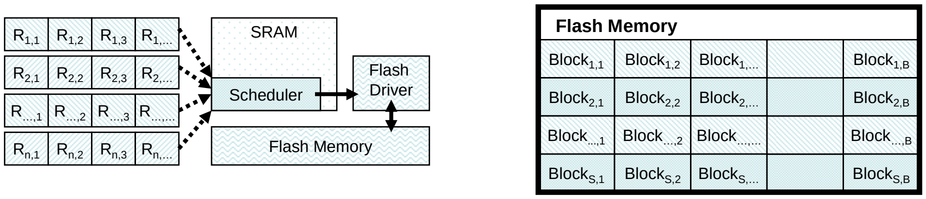
Although there have been proposed several quality of service (QoS) methods for traditional hard drives (HDD) in the literature [6, 9, 12], they do not readily lend themselves for implementation on sensors or for flash memories. Most of the traditional HDD based schedulers exploit the physical characteristics of hard drives to provide improved performance: access time affected by the positioning of the data access arm. This particular limitation is not applicable for flash memories, but flash memories do have other constraints and characteristics which must be considered. File systems specific for flash memories (ELF [8], JFFS [16], YAFFS [3]) work around the constraints of flash memories using log based structures, but these are more concerned with providing structures for storing files and efficient garbage collection for freeing space than QoS and I/O efficiency, which is the focus of our work.

In this paper we propose Reordering Grouped Earliest Deadline First (RG-EDF), a scheduling policy for flash-based sensor devices. Our policy aims at by combining multiple requests from the same task and selectively reordering the requests so that several requests can be written on the flash together. Our method provides much better performance than current method of data storage by taking advantage of the unique characteristics of flash memories. This is the first work, that we know of, which focuses on providing quality of service for storage in flash memories on sensor devices. We have implemented our scheme on a CC1010 sensor node with a SD flash card attached. We have experimentally compared our scheduler with FIFO and regular EDF schedulers. Our experimental results show the working and benefits of our proposed system.

2 Background

Traditional schedulers designed for improved efficiency in hard drive based systems ([14] [12] [7]) try to optimize the storage I/O by rearranging the order of operation in such a way to minimize drive arm movement. Although flash memories do not have the physical limitations of a drive arm, we are also trying to organize our data in a way to take advantage of I/O characteristics.

Anticipatory scheduling [9] is a proposed solution to combat prevents deceptive idleness by waiting for a while after requests for new requests from the process which has just been serviced. While we do not have the problem of deceptive idleness, we are inspired by the concept of waiting for additional requests in order to increase the overall throughput for flash storage.


$R_{1,1}$ $R_{1,2}$ $R_{1,3}$ $R_{1,...}$
$R_{2,1}$ $R_{2,2}$ $R_{2,3}$ $R_{2,...}$
$R_{...,1}$ $R_{...,2}$ $R_{...,3}$ $R_{...,...}$
$R_{n,1}$ $R_{n,2}$ $R_{n,3}$ $R_{n,...}$
SRAM
Scheduler
Flash Driver
Flash Memory
Flash Memory
$Block_{1,1}$ $Block_{1,2}$ $Block_{1,...}$ $Block_{1,B}$
$Block_{2,1}$ $Block_{2,2}$ $Block_{2,...}$ $Block_{2,B}$
$Block_{...,1}$ $Block_{...,2}$ $Block_{...,...}$ $Block_{...,B}$
$Block_{S,1}$ $Block_{S,2}$ $Block_{S,...}$ $Block_{S,B}$

(a) The sensor system: each process generates a series of requests, which are placed into the scheduler residing in *SRAM*. The scheduler then interacts with the flash driver to complete the requests.

(b) The structure of a flash memory with *S* sectors, each containing *B* blocks. Blocks are labeled as $\text{Block}_{\text{sector,block}}$

Fig. 1. System and Flash Structure

Bisson et al , in [6] and [5], reduce the energy consumption and increase the efficiency of hard drive spin down algorithms through the addition of a flash memory cache. Although they do make use of flash memories, it is used as a temporary cache for increased hard disk performance and they are not concerned with the actual write efficiency to the flash memory itself.

Flash memories differ from traditional hard drive scheduling optimizations in several critical ways, the ones we are concerned about are the I/O characteristics. There is no disk arm on a flash device and the physical location of data on a flash device does not affect its access or read time, however, I/O operations in flash memory are performed in block sized units. If we wish to write or read less than a block of data, we incur the same time and energy penalties as if we were operating on an entire block.

Recently techniques have been proposed for data storage and indexing in sensor networks. A few flash-based file systems have been proposed, including RISE [4] and PRESTO [11]. File systems specific for flash memories (ELF [8], JFFS [16], YAFFS [3]) implement log-like file structure designed for wear-leveling. These are more concerned with providing structures for storing files and efficient garbage collection for freeing space. In contrast, our work focuses on scheduling for improving QoS and I/O efficiency, targeting real-time multimedia tasks.

3 System Design

In this section, we first present our system settings, assumptions, and highlight the design principles. We then present an overview of the core components of the system and how they interact with each other, we then detail the design of the our scheduler.

3.1 System Settings

To control the order of execution and improve the quality of service, we insert a scheduler which intercepts requests and reorders them (see Figure 1).

We focus on the I/O processes on a single sensor device. We assume that a sensor runs n processes $P_1...P_n$, each process P_i produces a series of requests $R_{i,1}, R_{i,2}, Rs_{i,3}, ...$ (refer to Figure 1(a)). Each request $R_{i,k}$ can be represented by a tuple $< t_{i,k}, d_{i,k}, size_{i,k} >$ where $t_{i,k}$ is the time the request is issued, $d_{i,k}$ is a deadline (time that the request must complete), and $size_{i,k}$ is the size of the request. We assume that requests arrive sequentially. A request $R_{i,k}$ is considered to be written successfully if it is stored to nonvolatile memory (flash memory, in this case) before time $t_{i,k} + d_{i,k}$, otherwise, it is considered to be a miss.

The flash memory is divided into S sectors with each sector containing B blocks of size B_{size} (this is shown visually in Figure 1(b). As mentioned earlier, due to the constraints of flash memory: Each write to flash memory takes $write_{time}$, must be exactly B_{size} and must occupy an entire block. Similarly, each read from flash Memory takes $read_{time}$, can read at most B_{size} and cannot cross block boundaries. When a block $b_{i,k}$ in sector s_i is deleted or modified, all other blocks $b_{i,l}$ in s_i must also be deleted (and possibly re-written).

3.2 Design Principle

Our work is based on the main observation of the read and write property of flash memories: all native operations are performed at a block level. When data is written or read from flash memory, requests for data smaller than a block still takes the same amount of time and energy as if an entire block is read or written. Consequently, we want our scheduler to avoid wasted capacity and maximize utility by trying to read and write only full blocks of data.

We exploit this property in our scheduler by grouping requests and completing them together instead of separately. Grouping requests together improves performance in two ways: by combining multiple requests together, we increase I/O utilization and are essential writing the extra requests for "free". Additionally, by writing multiple requests, we clear out the scheduler faster and reduce the number of drops due to scheduler saturation. In situations where request injection is bursty, grouping allows many of the requests from the bursts to be together, making it especially effective in these cases when compared to the simpler, non-grouping schedulers.

3.3 Requests

When a request is generated, space is allocated for data in the $SRAM$, and the request object is sent to the scheduler. The request object contains: process id, sequence number, the deadline of the request, its data size and a pointer to its actual data. We chose to use a pointer to the data rather than the storing the data in the request object because we wish to maintain a constant sized scheduler in memory even when requests differ in size.

When the I/O component becomes idle and the application is ready to perform the next I/O operation, it will retrieve a request object from the scheduler. Then, a check will be made to determine if the deadline of the request can be met and the operation is performed depending on the system's policies (e.g. drop request on miss).

3.4 Reordering Grouped EDF Scheduler

We introduce a new scheme, called Reordering Grouped-EDF (RG-EDF). RG-EDF attempts to avoid performing partial block requests by grouping consecutive tasks together each time the request with the smallest deadline will be served. Performance is further improved by allowing for additional flexibility in scheduling the order of writes.

We initially consider a Grouped EDF (G-EDF) scheduler. G-EDF (and RG-EDF) uses the same heap structure as the regular EDF scheduler. When a request needs to be retrieved, instead of just returning the top item in the heap, the G-EDF scheduler will search through the heap and try to find sequential requests from the same process to combine.

The request $R_{i,k}$, is at the top of the heap. The scheduler will scan the heap searching for requests $R_{i,k+1}, R_{i,k+2}, ...$ and combine the requests until

$$\sum_{j=0}^{n} size_{i,k+j} > B_{size}$$

or all the items have been searched. The scheduler will then combine and return the requests $R_{i,k}...R_{i,k+n-1}$.

The RG-EDF scheduler improves upon the base G-EDF scheduler by selectively reordering the requests and waiting for new requests before returning a set of grouped requests. RG-EDF checks if better I/O utilization can be achieved by allowing another request with a later deadline proceed before the current earliest deadline request in cases where the deadline of the requests permit us.

Suppose $R_{i,k}$ is the earliest deadline request, we can allow a request $R_{j,l}$ to proceed ahead if we can accurately predict how long $R_{j,l}$ will take to service:

$$t_{i,k} > current_{time} + write_{time} + buffer_{time}$$

$$\sum_{a=0}^{n} size_{i,k+a} < \sum_{b=0}^{m} size_{j,l+b}$$

where

$$\sum_{a=0}^{n} size_{i,k+a} \leq B_{size} and \sum_{b=0}^{m} size_{j,l+b} \leq B_{size}$$

This allows for the group of requests $R_{j,l}...R_{j,l+m}$ which occupies a large portion of a block to proceed before $R_{i,k}...R_{i,k+n}$. This also provides the opportunity for more requests from process i to arrive while the I/O operation for process j is underway.

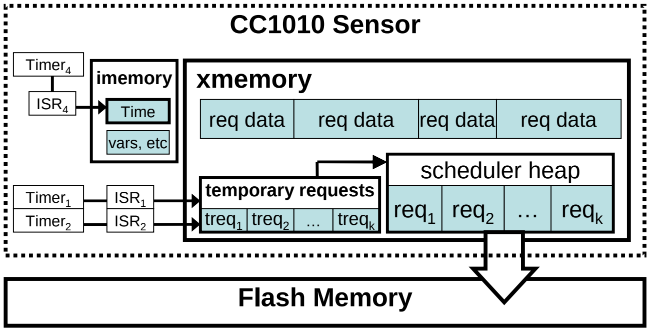


Fig. 2. implementation structure of the schedulers

4 Implementation

We implemented the system on a CC1010 sensor with a SD flash card attached through the Serial Peripheral Interface (SPI) bus. This section discusses the main components of the CC1010 sensor, the interface and characteristics of the SD flash card and gives implementation details on our different schedulers. An implementation system diagram is shown in Figure 2.

4.1 CC1010 Sensor

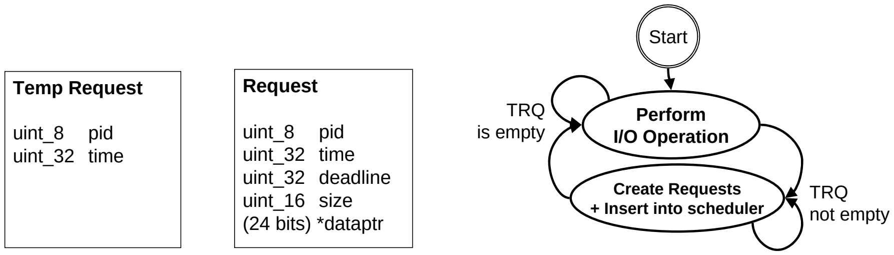
We are using is a Chipcon CC1010 sensor with an 8051 Enhanced Microcontroller [1]. The main features we are concerned about in the CC1010 are its memory model and its interrupt timers.

The memory is divided into two main sections: an internal memory (imemory) of 128 bytes and an external memory (xmemory) of 2024 bytes. Since the imemory is faster than the xmemory, we place frequently used and time sensitive data items (such as loops and the time counter) in imemory. We use xmemory for data structures which require large amounts of memory such as the queue and scheduler heap.

4.2 SD Flash Card

We used a 128 MiB Sandisk card with 512 byte blocks arranged in 256 block sectors. There are two basic operations supported by the flash SD card: reading a block and writing a block. Any erasing and recopying of information in a sector are handled internally by hardware on the flash card itself. Our custom flash driver implements the two basic read and write operations.

We connect to the flash card using a SPI bus. While this does limit the speed of reading and writing to the card, it also greatly simplify the hardware interface. Both of the basic operations mentioned above must be completed in blocks of exactly 512 bytes. To perform an operation, we first place the command followed



(a) Data Structure of Request in temporary request queue

(b) Data Structure of Request in schedulers

(c) Main Application States

Fig. 3. Data structures and application states.

by the address of the block onto the SPI bus. We then poll the serial interface to either read or write data to and from the card. In addition, after a write completes on the sensor, we must wait for an additional time while the card finalizes the operation.

Write time for a block to flash requires 9 ms, but reading from xmemory and writing the data typically takes up to 13 ms for a full block of 512 bytes. For the first write into a new sector, an extended write time ranging from 60 to 140 ms is required, this is a characteristic of the internal hardware of the SD flash card. This extra time is used to reorganize the data internally and speeds up subsequent writes. Reading a block takes slightly more time than writing ranging from 11 ms to 15 ms.

4.3 Timing and Request Injection

The CC1010 sensor provides 4 interrupt timers ($timer_1$ - $timer_4$), 3 of which we can use. We set $timer_4$ as the highest priority interrupt for keeping a counter to measure elapsed time in 1 ms increments. We simulate multiple processes issuing requests by using the ISRs associated with $timer_1$ and $timer_3$.

To keep the interrupt handlers as compact as possible; a full request object is not created at the time of an interrupt. Instead, a temporary request (treq) object is placed into the temporary request queue (TRQ), and a request object is created between I/O operations. The treq structure is shown in Figure 3(a) and contains the timestamp at the time the request is injected. Between I/O requests, complete request objects (shown in Figure 3(b)) are created for any treqs in the TRQ and are inserted into the scheduler.

Request injection is handled by the ISRs invoked by $timer_1$ and $timer_3$. The period that each timer fires its interrupt can be adjusted and this is what we use to vary the request injection frequency. When a ISR is invoked, a treq is created for each process and placed into the TRQ. $timer_1$ is always used to control process 1 and $timer_3$ is used to control processes $2...n$ where there are n total processes. We can then adjust $timer_1$ independently of $timer_3$ when we wish to perform experiments where the frequency of only one process changes.

Bursty request injection is modeled by having the ISR inject multiple requests instead of a single request. For example, on every 10th run of the ISR for $timer_1$, 5 requests are injected for process 1 instead of a single request.

4.4 Scheduler Implementation

The application has two main states (shown in Figure 3(c)): one state where I/O operations are being performed and another when requests are generated and insert into the scheduler from the TRQ. Two classes of schedulers are implemented: a FIFO queue and a EDF scheduler. Our RG-EDF scheduler is built on top of the EDF scheduler base. Our schedulers also have the ability to drop requests which are determined to miss their deadlines.

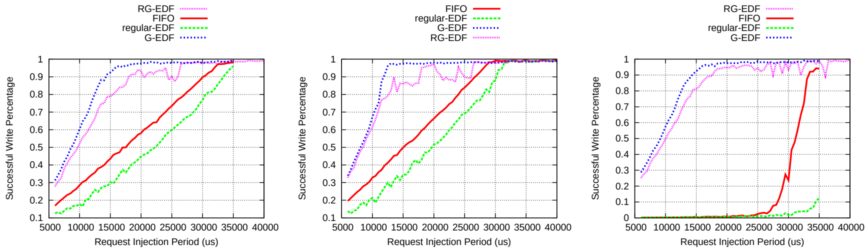
FIFO Queue We simply used the TRQ as a FIFO queue. The TRQ is implemented as a circular queue in an array of treq objects. When this queue becomes full, no further requests are accepted and these requests are counted as dropped requests. In the I/O phase, we simply retrieve the first treq object from the queue, generate a full request object and perform the I/O operation.

EDF Schedulers The base EDF scheduler is a priority queue implemented as a binary heap. The data structure of each request in the heap can be seen in figure 3(b). We only keep track of the time a request is generated for statistical purposes and can be optimized out.

By keeping the size of each request object constant, we are able to use a simple array-based heap structure. The top item in the heap will always be at index 0 and the child nodes for an item at index i would be at $i * 2 + 1$ and $i * 2 + 2$. In addition to being able to remove items from the top of the heap, we can also specify which heap index we wish to remove an item from.

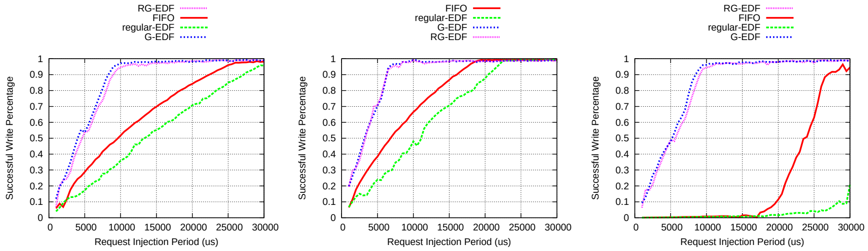
Building on top of this base EDF scheduler heap, functions are introduced to group and reorder requests. When a request is made, a list of indices from the heap is returned containing requests which can be grouped together. Once the application receives the list of indices, it then proceeds to retrieve, and remove from the heap, each request, starting with the last index (this ensures that the remaining heap indices remain valid). After retrieving all the requests in the group, a single I/O operation can be performed.

Knowing the typical time required to write a request to flash, the scheduler can allow requests which have later deadlines proceed ahead of requests which have an earlier deadlines, if the earlier deadline is not violated. The scheduler compares the grouped size of the top request with the grouped size of the next request from a different process. If the second group of requests has a larger total size, the second group of will be allowed to proceed first. This improves the



(a) Bursty requests with dropping misses (b) Non Bursty requests with dropping misses (c) Bursty requests without dropping misses

Fig. 4. Varying period of request injection for all 3 processes with default deadline delay ($d_{i,k} = 70$ ms) and request size ($size_{i,k} = 64$ bytes).



(a) Bursty requests with dropping misses (b) Non Bursty requests with dropping misses (c) Bursty requests without dropping misses

Fig. 5. Varying period of request injection for 1 of 3 processes with default deadline delay ($d_{i,k} = 70$ ms) and request size ($size_{i,k} = 64$ bytes).

I/O utilization and also allows more requests from the first process to enter the scheduler while the I/O operation is underway.

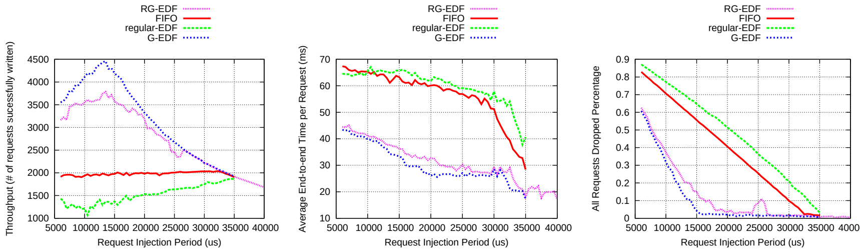
5 Experimental Evaluation

5.1 Experiment Setup

We compare the performance of our systems: RG-EDF, with the traditional FIFO approach and an EDF implementation. We run the experiments with three data flows. Each data flow (i) is defined by the period of the request injections, the request sizes ($size_{i,k}$) and the deadline delay ($d_{i,k}$) for each request.

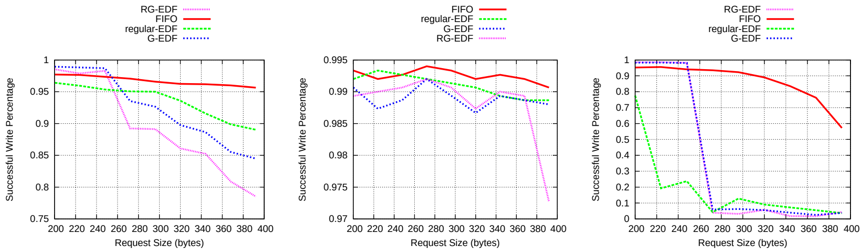
We evaluate the schedulers by varying single parameters over a range of values. We also choose some conservative default values such that each of the schedulers perform well at these values:

- injection period: 40 ms
- request size: 64 bytes



(a) Throughput in number of requests successfully written (b) End-to-end time each request spends in the system (c) Total # of dropped requests due to full scheduler and dropping misses

Fig. 6. Varying period of request injection for all 3 processes with default deadline delay ($d_{i,k} = 70$ ms) and request size ($size_{i,k} = 64$ bytes).



(a) Bursty requests with dropping misses (b) Non Bursty requests with dropping misses (c) Bursty requests without dropping misses

Fig. 7. Varying request size for all 3 processes with default injection period 40 ms and deadline delay ($d_{i,k} = 70$ ms).

– deadline delay: 70 ms

In addition to varying a parameter, we examined the effects of allowing the schedulers to drop requests when it is determined that their deadlines cannot be met. We also look at the performance in the presence of **bursty** traffic: instead of a single request, 5 requests are injected for process 1 every 10 injection cycles.

For each experiment, 10 sets of 10 second runs were completed and the results averaged. The results are also aggregated across all three processes running.

5.2 Varying Frequency

We vary the frequency for all three process at the same time. Since the deadlines for all the request are the same and the periodic requests are injected at the same time, the EDF scheduler is acting exactly like the FIFO scheduler, except the EDF scheduler has the additional overhead associated with the scheduler. The G-EDF and RG-EDF schedulers share the same overhead as the EDF scheduler but is able to compensate by grouping requests together, leading to much better performance. Figure 4 shows that G-EDF performs better than RG-EDF because there are not many opportunities for reordering and the extra overhead of reordering logic is demonstrated.

Figure 4 (a) and (b) show the difference in performance between bursty and non bursty traffic, all schedulers perform better when the traffic is non bursty, due to there being less requests overall, but G-EDF and RG-EDF handle bursty traffic better due to their ability to group these requests together.

When we disallow dropping misses (compare Figure 4 (a) and (c)), by allowing requests to complete even when they are going to miss their deadlines, we see that FIFO and EDF performance drop off rapidly due to missed deadlines causing more misses. When drops are enabled, FIFO and EDF perform much better since they are available to only perform on-time requests. Further, figure 6(c) shows that the majority of the drops for G-EDF and RG-EDF are due to scheduler saturation rather than being dropped due to misses.

Figure 5 shows the results when only varying injection period for a single process. FIFO and EDF perform better because there are less results in total to deal with: while the injection period of the single process is increased, the other two processes are injecting in 40 ms periods. G-EDF shows a modest amount of performance gain while RG-EDF has many more reordering opportunities and shows improvement over Figure 4. As the injection rate for the single process increases, more reordering opportunities are presenting themselves.

Figure 6(a) shows the throughput of each scheduler as the injection rates are increase. The FIFO scheduler has a constant throughput; it can write a constant number of request independent of how many are injected. EDF suffers from the scheduler over head is unable to keep up. G-EDF and RG-EDF both perform well and peak when scheduler saturation causes drops. Figure 6(b) shows that requests spend much less time in the G-EDF and RG-EDF schedulers.

5.3 Varying Request Size

In this set of experiments, we vary the request sizes. From the results (Fig 7), we see a drop in performance from our schedulers. This occurs when the request size increases above 256 bytes; beyond this point, the scheduler can no longer group multiple requests together because our block size is only 512 bytes.

G-EDF and RG-EDF rely on grouping multiple requests together to improve performance. These schedulers introduce extra sophistication and overhead; if requests are large and grouping not possible, they act like the EDF scheduler and the extra overhead will cause the grouping schedulers to perform poorly.

6 Conclusion

Sensor applications are become increasingly complex: requiring multiple streams of data storage and taking rich measurements such as visual and audio data which are frequently bursty. In this paper we have proposed the RG-EDF scheduling policy, which shows excellent performance in many cases where simple schedulers are insufficient. RG-EDF achieves that by taking into consideration the unique characteristics of flash-based devices when storing real-time multimedia data. RG-EDF is easy to implement, and thus makes it suitable for resource-constrained sensor nodes.

References

1. Chipcon cc1010, http://www.keil.com/dd/chip/3506.htm.
2. Cmucam, http://cmucam.org/.
3. Yaffs (yet another flash file system), http://www.yaffs.net/.
4. A. Banerjee, A. Mitra, W. Najjar, D. Zeinalipour-Yazti, V. Kalogeraki, and D. Gunopulos. Rise- co-s : high performance sensor storage and co-processing architecture. In *IEEE Sensor and Ad Hoc Communications and Networks*, Santa Clara, CA, Sept. 2005.
5. T. Bisson and S. Brandt. Reducing energy consumption with a non-volatile storage cache. In *Proc. of International Workshop on Software Support for Portable Storage (IWSSPS), held with RTAS 2005*, San Fransisco, CA, March 2005.
6. T. Bisson, S. A. Brandt, and D. D. E. Long. Nvcache: Increasing the effectiveness of disk spin-down algorithms with caching. In *MASCOTS*, pages 422–432, Monterey, CA, September 2006.
7. E. Carrera and R. Bianchini. Improving disk throughput in data-intensive servers. In *HPCA 2004*, Madrid, Spain, Feb 2004.
8. H. Dai, M. Neufeld, and R. Han. Elf: an efficient log-structured flash file system for micro sensor nodes. In *SenSys '04*, pages 176–187, Baltimore, MD, USA, 2004.
9. S. Iyer and P. Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *18th ACM Symposium on Operating Systems Principles*, Chateau Lake Louise, Banff, Canada, Oct. 2001.
10. P. Kulkarni, D. Ganesan, and P. Shenoy. The case for multi–tier camera sensor networks. In *NOSSDAV '05*, pages 141–146, Stevenson, Washington, USA, 2005.
11. M. Li, D. Ganesan, and P. Shenoy. Presto: feedback-driven data management in sensor networks. In *NSDI'06*, pages 23–23, San Jose, CA, 2006. USENIX.
12. C. Lumb, J. Schindler, G. R. Ganger, E. Riedel, and D. F. Nagle. Towards higher disk head utilization: Extracting “free” bandwidth from busy disk drives. In *OSDI 2000*, pages 87–102, San Diego, CA.
13. L. Luo, Q. Cao, C. Huang, T. Abdelzaher, J. A. Stankovic, and M. Ward. Enviromic: Towards cooperative storage and retrieval in audio sensor networks. In *ICDCS '07*, page 34, Washington, DC, USA, 2007. IEEE Computer Society.
14. E. Mumolo. Prediction of disk arm movements in anticipation of future requests. In *MASCOTS 1999*, College Park, Maryland, Oct 1999.
15. M. Rahimi, R. Baer, O. I. Iroezi, J. C. Garcia, J. Warrior, D. Estrin, and M. Srivastava. Cyclops: in situ image sensing and interpretation in wireless sensor networks. In *SenSys '05*, pages 192–204, San Diego, California, USA, 2005.
16. D. Woodhouse. Jffs: The journalling flash file system. In *Proc. Ottawa Linux Symp.*, http://sourceware.org/jffs2/, 2001.
17. D. Zeinalipour-Yazti, S. Lin, V. Kalogeraki, D. Gunopulos, and W. A. Najjar. Microhash: An efficient index structure for flash-based sensor devices. In *FAST 2005*, San Fransisco, CA, Dec 2005.