Hindawi Publishing Corporation
International Journal of Distributed Sensor Networks
Volume 2013, Article ID 324641, 20 pages
http://dx.doi.org/10.1155/2013/324641

Research Article

Providing Virtual Memory Support for Sensor Networks with Mass Data Processing

Nan Lin,[1] Yabo Dong,[1,2] and Dongming Lu[1,2]

[1] *School of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China*
[2] *Cyrus Tang Center for Sensor Materials and Applications, Zhejiang University, Hangzhou 310027, China*

Correspondence should be addressed to Nan Lin; nlin@zju.edu.cn

Received 26 October 2012; Accepted 28 January 2013

Academic Editor: Sheikh I. Ahamed

With the development of sensor networks and emerging of various sensors, sensor networks are capable of acquiring mass data to achieve much more complex monitoring tasks than ever. For example, image sensor nodes take photos using cameras, and images are collected and processed or stored for further processing. So, mass data processing is required for these sensor networks. However, low-power resource-constrained sensor nodes are normally equipped with kilobytes of RAM which might be not enough for storing large data for processing. In this paper, we propose an optimized virtual memory mechanism for large data processing on low-power sensor nodes. We point out the major overhead of virtual memory for large data processing on sensor nodes and introduce efficient solutions to address these issues. Evaluation shows that the overhead of the proposed virtual memory is reduced to an affordable range. We further compare the energy consumption of data processing programs using virtual memory with other means that process or transmit data. Data processing using virtual memory can be significantly more energy efficient than data processing using rich-resource sensor nodes or transmitting data to powerful gateways for central processing.

1. Introduction

Sensor networks are used in various areas nowadays, like environment monitoring, agriculture, video surveillance, and so forth. Sensor nodes now collect not only scalar sensed data from the environment, but also stream of mass data, for example, videos and images. Fast in situ processing of large data might be required to enable fast response or to reduce transmission overhead. Sensor nodes are expected to carry out large data processing with reasonable overhead. However, low-power sensor nodes are known for their limited resources. For instance, motes are equipped with kilobytes of RAM which may easily be insufficient for storing or processing images. Programs for large data processing on sensor nodes may require much more memory than the RAM size of the microcontroller.

There would generally be three different methods to address this problem listed below.

(1) Use more powerful microcontrollers with sufficient RAM for large data processing.

(2) Employ virtual memory to increase the available memory for data processing programs.

(3) Transmit large data to powerful gateway for central processing.

Using more powerful microcontrollers equipped with sufficient RAM for data processing would be a straightforward solution for large data processing. For example, Intel Mote 2 is a sensor platform with increased CPU performance and improved radio bandwidth to acquire, process, and transmit large data streams [1]. The platform provides 32 MB SDRAM on-board, which would be large enough for storing images or videos. Unfortunately, SDRAM has been known to be more expensive and energy consuming per bit compared to other memory mediums (NOR flash, NAND flash, etc.). This has led to the relative high power consumption of Intel Mote 2 in deep sleep mode. The current in deep sleep mode of Intel Mote 2 is about 387 μA which is much larger than that of traditional sensor nodes (about 10 μA) (e.g., MicaZ, TelosB). This would greatly reduce the lifetimes of the sensor nodes since sensor nodes are expected to be in deep sleep mode

most of time. So, sensor platforms with large RAM space may not be suitable for long-term surveillance.

Virtual memory [2] is generally to provide flat isolated memory address spaces for programs on conventional computer architectures. With the support of Memory Management Unit (MMU), page faults can be caught and out-of-bound memory accesses can be protected from destroying other programs or the operating systems. However, virtual memory is generally used on resource-constrained sensor nodes to expand its limited RAM space to provide enough memory for complex programs, for example, TinyDB [3], whose memory footprint outruns the physical RAM provided by low-power microcontrollers. This is different from conventional virtual memory in that isolating memory errors of programs from destroying other parts of system is not a major concern. Without MMU, virtual memory cannot be implemented on low-power sensor nodes easily. Software-aided mechanisms must be used to allow transparent virtual memory access. Generally, assembly instructions accessing memory are replaced by snippets which redirect the memory accessing to read/write virtual memory. Despite the significant benefits of virtual memory, traditional sensor node programs do not employ virtual memory for its high overhead.

Traditionally, sensor nodes collect data and transmit data to central gateway for processing because sensor nodes are generally supposed to be resource constrained. Transmitting large data to gateway can be an option for sensor nodes if data processing is limited due to insufficient RAM. However, transmitting large data to gateway would incur much overhead because network transmission is known to be about 1000 times more energy hungry than data processing. To the best of our knowledge, there is currently no work to compare the energy consumption of network transmission and data processing using virtual memory in sensor networks.

We have been working on building an image sensor network in which sensor nodes capture images of the target object and do in situ image comparing to compress similar images to minimize network traffic. We have developed an image sensor node platform. STM32F103ZE [4] is used as the microcontroller whose deep sleep mode current is about 20 μA, making our image sensor node platform suitable for long-term surveillance. STM32F103ZE is equipped with 64 kilobytes of RAM. This is much larger than that of ATMega128 [5] and MSP430 [6], however, still far from enough for processing images taken from cameras. Without enough memory, algorithms and existing programs must be altered to work with limited memory. For example, image operations can be carried out block by block, intermediate results can be stored on external storage, and the final result can be calculated from the intermediate results. This might bring heavy overhead to the development of image sensor node programs. Our primal motivation for developing virtual memory is to support large data processing and enable long-term operation on sensor nodes at the same time. By developing a virtual memory mechanism for the Cortex-M3 [7] platform, existing algorithm implementations can be used directly without adaptations. We tried to provide megabytes of memory for programs to manipulate images all in memory. The image sensor nodes are equipped with NAND flash of size 2 gigabytes, and we use parallel NAND flash as the secondary storage media for the virtual memory. It is generally a valid assumption that image sensor nodes are equipped with large NAND flashes, mainly for storing sensor data.

The virtual memory mechanism is verified on our image sensor nodes using various general data processing programs. Evaluation shows that the overhead of virtual memory can be reduced dramatically using these optimizations. Our contribution lies in the fact that we prove that virtual memory can be used for sensor nodes to achieve more energy-efficient data processing than using high performance energy-consuming microcontrollers or transmitting large data to resourceful gateway and we have proposed a flash translation layer [8] to use NAND flashes efficiently for secondary storage accessing.

Rest of paper is organized as follows. We describe the challenges we have faced when developing the virtual memory mechanism in Section 2. Section 3 gives an introduction of related work. Section 4 gives an overview of the proposed virtual memory system. Section 5 describes the details of C code virtualization, in which last-cache buffers are used to accelerate address translations. Section 6 describes how NAND flash is used as secondary storage and the design and details of Lavish-FTL. Section 7 evaluates the overhead of the proposed virtual memory mechanism with data processing programs and compares the energy consumption of these programs with other options. Conclusion and future work is drawn in Section 8.

2. Challenges

Two major issues have been met when we were trying to make the virtual memory implementation efficient enough to be affordable for data processing programs. The two issues are discussed in the following two subsections. Section 2.3 gives our solutions for these two issues.

2.1. Code Virtualization. One requirement of the virtual memory is that programmers develop their programs unawares of the underlying virtual memory. Code virtualization is used to make programs work with virtual memory on MMU-less microcontrollers. For example, assembly instructions which access memory (i.e., LDR, STR in the ARM instruction set) can be replaced by routines which access virtual memory. We assume a flat address space provided by the virtual memory, mapped linearly to the secondary storage. It is reasonable when the size of the secondary storage is much larger than the required virtual memory size. Memory operations in programs are executed at virtual address and the memory footprints of programs reside in the secondary storage. Caches are used to eliminate most secondary storage accessing. The replacement routines for memory accessing in code virtualization mainly consist of following steps:

(i) locate the page of accessing the virtual address in caches;

(ii) if the cache is not present, load cache data from secondary storage page at virtual address;

(iii) issue the final read or write in the cache.

The code virtualization employs a progress in which virtual addresses are translated to physical addresses to caches, which is called address translation. The major part of the execution overhead of virtual memory takes place among address translations. The execution overhead of address translation mainly consists of the following parts:

(i) execution overhead of the wrapper snippets,

(ii) execution overhead of cache searching,

(iii) execution overhead of handling cache misses.

The wrapper snippet is mainly responsible for saving the current context and calling the cache searching routine. The implementation and overhead of the wrapper snippet are highly related to the specific instruction set used by the microcontroller.

The overhead for searching cache depends on how the caches are structured. The searching overhead would be minor if the number of caches is small and the caches are structured by cache sets [9], or if the caches can be located at compile time [10]. However, our virtual memory is designed to support complex data processing programs which have large memory footprints, so there must be enough RAM reserved for caches to achieve very low miss rate. We have chosen to organize caches in a fully associative way. It is known that using fully associative caches would achieve the minimal miss rate [11]. However, it is generally slower to search among fully associative caches because more caches need be travelled before hitting the target cache. Although the time of executing cache searching once is just tens to hundreds of MCU cycles, the address translation needs to be executed for every virtual memory access so the total overhead is large.

Overhead of handling cache misses is practically equal to the overhead of secondary storage accessing which is discussed in the next subsection.

2.2. Secondary Storage Accessing. Our work uses large parallel NAND flash with typically 64 sectors in one block as the secondary storage medium. Different flash techniques have been examined in [12] which gave an energy consumption comparison as shown in Table 1, proving that parallel NAND flash is very efficient in per-byte power consumption. To our knowledge, there is currently no work in the area of sensor networks which optimize secondary storage accessing for large NAND flashes. This is probability due to the fact that existing virtual memory mechanisms on sensor nodes are mostly designed for motes using small, low-power serial flashes which can be erased and written on a page basis.

We assume that the virtual address space is mapped to the NAND flash storage space linearly. This is reasonable because the size of NAND flash (gigabytes) is much larger than the required memory space of sensor network applications (megabytes). When accessing a variable at specified virtual address, the virtual address can be mapped to a NAND flash address directly by adding a constant offset. The data at the calculated NAND flash address is then loaded to a RAM cache and is finally served to accomplish the memory operation.

Table 1: Flash energy consumption read, write, and erase.

	Read	Write	Erase	Bulk erase (Page count)
Atmel NOR	0.26	4.3	2.36	n/a
Telos NOR	0.056	0.127	n/a	0.185 (256)
Hitachi MMC	0.06	0.575	0.47	0.0033 (16)
Toshiba 16 MB NAND	0.004	0.009	n/a	0.004 (32)
Micron 512 MB NAND	0.027	0.034	n/a	0.001 (64)

We however have observed some special features in virtual memory secondary storage access. First, the memory footprints of sensor node applications are much smaller than the NAND flash size. So it would be preferable to trade the flash space for reduced access time if possible, since energy is the major concern on sensor nodes. Second, the common knowledge that the access overhead of one NAND sector byte and the access overhead of the whole NAND sector are approximately equal does not hold for sensor nodes since MCUs on sensor nodes are relatively slow and the transmission time of command and data in NAND flash operations is significant and is basically proportional to the access size, although the underlying NAND flash sector reading or programming time is constant for different access sizes. Figure 1 shows the execution time of reading and writing a NAND flash (Model: K9F1G08X0A) on our image sensor node with different sizes for 16000 times. It can be observed that the execution time increases as the size of reading or writing grows and the command time (total write time minus write program time or total read time minus read transfer time) is proportional to the R/W size. The constant execution time for data transfer of reading and programming of writing is irrelevant to the R/W size. The data transfer time and program time takes a minor part of the total execution time which is particular for resource-constrained sensor nodes. One may also notice that the readings are not necessarily faster than writings. So, it is important for VM secondary storage access to reduce extra reading and writing due to their high overhead.

Besides the relative slow access speed, NAND flash has a special access style that pages must be erased before rewriting. NAND flashes are typically read and written on a page basis and erased on a block basis and one block contains multiple pages (typically 64). This requires an intermediate layer called flash translation layer (FTL) to be introduced to adapt NAND flashes to block devices and do wear leveling. State-of-the-art FTLs [13–15] are generally designed and optimized for file systems. Virtual memory secondary storage was not a concern of the designers of these FTLs. FTLs designed for file systems are usually concerned with the file system consistence upon power shutdown, which is irrelevant for

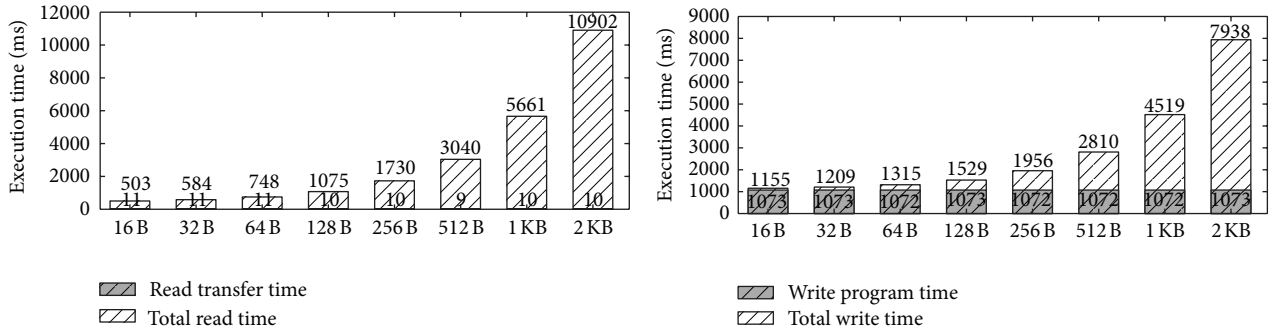
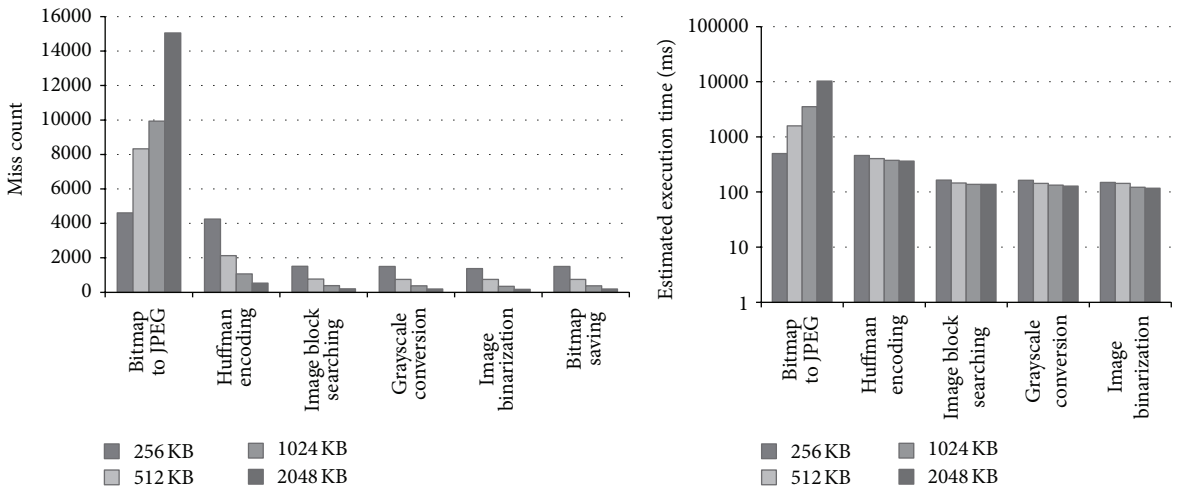


Figure 1: Execution time of reading and writing NAND flash with different sizes.



(a) Cache miss counts of programs

(b) Estimated read overhead for cache misses

Figure 2: Evaluation of cache misses of programs using LRU with cache RAM footprint being 32 KB.

VM secondary storage because data in virtual memory does not need to persist after reboot. NAND flash has an erase limit at about 10000 to 100000 times per block. So, it is critical for virtual memory to reduce block erasing to prolong the lifetime of sensor nodes.

The other issue is the mismatch between the cache size and the NAND sector size. It is apparent that if the two sizes are equal, caches can be read or written back by reading or writing just one sector. However, the best cache size is determined by the memory access pattern of the application and the available RAM space for caches [11]. Figure 2(a) shows the cache miss counts for some programs using LRU cache replacement algorithm with RAM footprint of caches being 32 kilobytes. As the cache size grows, the miss count increases for the JPEG to BITMAP program and decreases for other programs. This is because if the memory access pattern is relatively sequential or the program's working set is small enough, using larger caches would not cause cache misses, otherwise, larger cache size would lead to higher cache miss rate. Read overheads for the cache misses for these programs as shown in Figure 2(b). For the JPEG to BITMAP program, the estimated read overhead grows dramatically with increased cache size. However, for other programs, the estimated read overhead is more or less the same for different cache sizes. The estimated read overhead can be a rough approximation for the secondary storage access overhead. So, it is apparent that cache size smaller than the NAND sector size can have better performance than larger cache sizes; thus the NAND sector size is generally larger than the best cache size. This issue is specific to sensor networks since the page size of virtual memory on conventional computer systems (typically 4 KB) is large and thus this issue does not exist.

2.3. Fast Transparent Virtual Memory for Sensor Nodes. We have developed a fast transparent virtual memory mechanism named FaTVM [16] for our image sensor nodes. FaTVM is implemented on Cortex-M3 microcontroller platform. FaTVM aims to provide sensor network data processing programs with an efficient memory space which can be much larger than the physical memory.

To reduce the overhead of code virtualization, we allocate last-cache buffers (LCBs) for lvalue accessing in C code. LCBs are buffer variables which save the last matched caches for lvalues. When an LCB is hit, the cache searching overhead can

be eliminated. Our evaluation shows the probability of LCB-Hit is generally higher than 90%. So, using LCBs can achieve fast address translation for most virtual memory accessing. LCB scheme is based on C code transformation which does code virtualization on C code level rather than assembly instruction level. C code transformation has further reduced the execution overhead of the wrapper which was described in Section 5.

We have designed and implemented an FTL named Lavish-FTL which is best suited for sensor nodes with large NAND flashes. The main idea of Lavish-FTL is to trade NAND flash space for better performance. The main configuration of Lavish-FTL, which we called the "lavish degree," determines how much NAND flash space is going to be used for a fixed FTL size configuration. For example, if the lavish degree is 2, Lavish-FTL needs 2 megabytes of flash space to provide 1 megabyte of FTL address space. As the lavish degree grows, the execution overhead of Lavish-FTL can be reduced; however other FTLs' overhead cannot be reduced when more flash space is provided. The increased flash space overhead is not an issue for sensor nodes with large NAND flashes. Assuming that the virtual memory size is 8 MB and the lavish degree is 8, Lavish-FTL would need 64 MB flash space served as secondary storage, which is acceptable for large NAND flashes. Lavish-FTL tries to reduce the overhead of erasing and eliminate most extra readings or writings which might be inevitable in block management.

We introduced two schemes to address the mismatch between the cache size and the NAND sector size. One solution is that when writing one cache back, other conjoint dirty caches belonging to the same sector are searched in the cache list so that these caches can be written back by writing NAND sector only once. Evaluation shows that this scheme reveals good performance when there are plenty of caches. The other solution is that an adapting layer is used to reduce the sector size of the FTL. Evaluation shows that this scheme is more efficient than the former scheme if the number of caches is limited.

3. Related Work

3.1. Virtual Memory for Sensor Nodes. Virtual memory has been an important research subject in traditional operating system research for decades. It provides applications with isolated large flat address spaces which greatly simplify development of reliable programs and prevent programs from ruining each others' address spaces. However, traditional virtual memory systems are generally based on MMUs, so they are not usable in MMU-less embedded systems.

Softvm [17] implements software-managed address translation without TLBs however it is still designed for conventional computer systems.

MEMMU [18] proposed an automated compile-time and run-time memory expansion mechanism to increase the amount of usable memory in MMU-less embedded systems by using data compression; however available memory can only be increased by up to 50%.

There have already been several studies on virtual memory in sensor networks. Most previous work was trying to support large complex programs like TinyDB on sensor nodes. The virtual memory was generally assumed to be relatively small and node interactivity was the major consideration among application performance metrics.

t-kernel [9] provides software-based virtual memory called DVM by which user applications can have flat virtual memory spaces much larger than the physical memory space of the host node. Data frames are used to cache virtual memory accessing, which is the same as the caches in FaTVM. t-kernel searches among multiple data frames to find the hitting data frame. t-kernel searches among caches in a round-robin scheme in which all data frames are arranged in a data frame array and it starts searching from the last used data frame. This scheme however could lead to significant searching overhead if there are many data frames. Assembly virtualization is used by t-kernel to make programs access virtual memory. However, assembly virtualization introduces more overhead for context saving and restoring compared with C code virtualization. According to our earlier implementations, even more overhead will be introduced by assembly virtualization on our image sensor nodes due to the flexible addressing modes of the ARM instruction set.

Evaluation of t-kernel showed high efficiency in memory access performance. Accessing heap without swap takes only 15 cycles. We believe it is because that address translation of t-kernel DVM is simpler than that of FaTVM, which is partly due to the simpler address modes of sensor platform MCU. However, t-kernel DVM has a much higher miss rate than FaTVM. The current t-kernel implementation allocates 64 data frames in RAM as buffer for flash access, and the miss rate is about 10% for "slidingwin" application, tens of times of the miss rate of FaTVM. So, we argue that the caching scheme and the assembly virtualization method in t-kernel DVM are not applicable for sensor network programs with complex data processing because of the high overhead of secondary storage access. We did not evaluate t-kernel's performance in this paper because t-kernel is not ported to STM32 MCU.

ViMem [10] brings virtual memory support to TinyOS [19]. Developers add tags to nesC [20] source code to place variables in virtual memory. ViMem creates an efficient memory layout based on variable access traces obtained from simulation tools to reduce virtual memory overhead. The effect would be significant for traditional sensor node programs; however it would be useless when programs access large variables frequently, which is common in complex data processing. Furthermore, it is not possible to reference a variable in virtual memory using a normal pointer variable in ViMem. Pointer variables in virtual memory have to be tagged with the attribute "@vmptr." Since data elements in virtual memory are not necessarily contiguous, casting variables to types of a different size is not allowed, neither is pointer arithmetic. We argue that virtual memory transparency and flexible pointer operation are vital for porting or implementing complex data processing algorithms.

Recently, Enix [21] supplies software segmented virtual memory for code memory by code modification and uses Micro-SD cards as secondary storage. However Enix does not provide virtual memory for data segments because of high run-time overhead, which is the very issue FaTVM trying

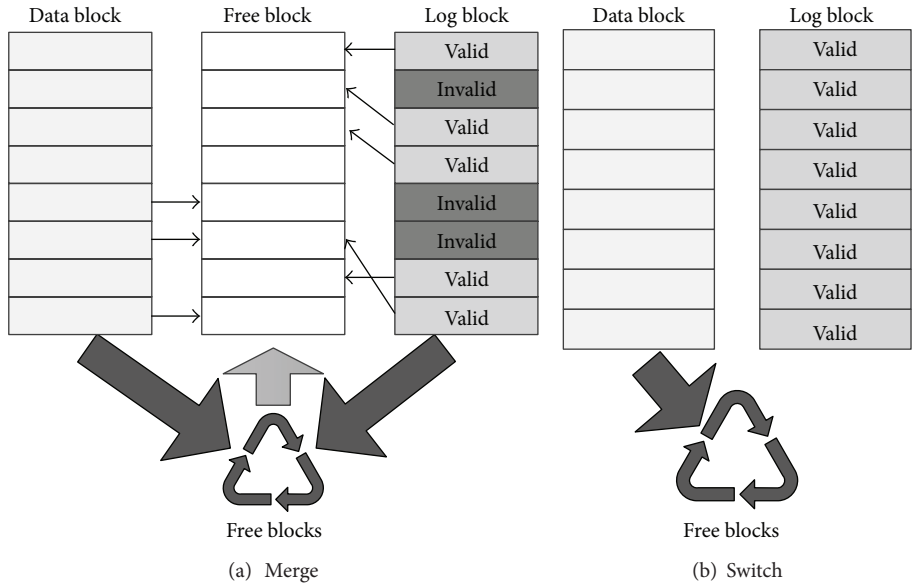


Figure 3: Block merge and switch in log based FTLs.

to address. Our earlier version of FaTVM also used Micro-SD cards as secondary storage; however Micro-SD cards are not as efficient as NAND flashes in energy consumption and access speed according to our evaluations of both implementations.

Our work differs from previous work largely in that we are trying to provide a large efficient virtual memory for mass data processing. The virtual memory mechanism should be able to increase the programs' available memory for tens of times or even hundreds of times. Most memory ranges accessed in data processing programs on sensor nodes are assumed to be in virtual memory. Directly applying previous work would lead to unacceptable overhead. Our work mainly focused on reducing the overhead of address translations and secondary storage accessing.

3.2. Flash Translation Layers. Due to the characteristics of NAND flash memory, flash translation layers are used to hide the inconvenient raw NAND flash interfaces and provide a block device interface. Taxonomy and design decisions of FTLs are not discussed in this paper due to paper size limitation.

Log-based FTLs are known to be efficient in sector writing. Log-based FTLs usually use one or more log blocks for the data block to buffer sector writings. Merge and switch are used to reclaim NAND blocks. Figure 3 shows the basic concepts of block merge and switch in the Log-FTL [13]. In log block merge, valid sectors are copied from the data block and log block to the free block, which brings about copy overhead of valid sectors. One free block is obtained after block merge with two erasings. When the log block contains all the sectors sequentially, block switch can be used which generates one free block with one erase and no copy of valid sectors. Log blocks in Fast FTL [14] can be used for any data block to increase the log utility. A dedicated sequential block is used to handle sequential writes efficiently with block switch. Superblock FTL [15] uses superblocks (a superblock consists of a set of adjacent logical blocks) to further exploit block-level spatial locality. Superblock FTL increases the chances of partial or switch merge operations instead of the expensive full merge operation.

4. FaTVM Overview and Architecture

We describe the overview of FaTVM in several aspects. Section 4.1 describes our developing hardware platform of FaTVM. Section 4.2 explains how source code is virtualized and compiled to final executive. Section 4.3 describes how virtual memory is accessed in programs.

4.1. Platform Description. We have been developing FaTVM on our image sensor nodes of an image sensor network [22]. Image sensor networks acquire image data by camera, and the sensor nodes are relatively more powerful than common motes such as MicaZ [23], TelosB [24], and so forth. The image sensor nodes are equipped with 32-bit ARM Cortex MCUs [4] from STMicroelectronics. The microcontroller has 72 MHz maximum frequency, 256 to 512 kilobytes of flash memory, and up to 64 kilobytes of SRAM. Although FaTVM is developed on our image sensor nodes with relative richer resource, it should be straightforward to port FaTVM to other platforms, since FaTVM is mostly implemented using C programming language and does not employ any advanced MCU features.

The image sensor nodes are equipped with large SLC NAND flashes for storing acquired images. The size of the NAND flash is 2 gigabytes. FaTVM uses a dedicated partition of NAND flash as the secondary storage.

STM32 MCU has a 4 GB memory map, in which memory address ranges of different sizes are allocated for code, SRAM, peripherals, external RAM, and so forth. FaTVM gives the programmer a view of contiguous memory space and a

dedicated range of address are used by application code to access virtual memory. We have specified a subrange of 1 GB external RAM addresses to be used by virtual memory, making it possible to determine address type to be virtual or physical by comparing the address with the boundaries of the dedicated address space.

4.2. Code Virtualization. FaTVM is developed without any assumption of the underlying operating system. Program code written in C programming language is transformed to access virtual memory. This progress is called code virtualization. Figure 4 shows how source files are virtualized and compiled to objects, which are relocated and linked to build the final executive. Programmers are responsible for specifying which source files to be virtualized thus it is possible to virtualize only a selected set of modules. Code virtualizer transforms source code written by programmers, who may not be aware of the virtual memory, to work with FaTVM. Object relocator moves variables in physical data sections to virtual data sections. It is still possible for programmers to manually specify some variables to be allocated in physical memory. This may bring significant performance improvement if variables accessed frequently in the program are allocated in RAM.

Assembly virtualization is commonly used in other software-based virtual memory mechanisms [9, 21]. Instructions of memory load or store are replaced by subtle assembly snippets to handle virtual memory accesses. It is a straightforward software method for implementing virtual memory. However, assembly virtualization has the following drawbacks.

(i) the overhead of assembly virtualization is high;

(ii) the replacements for memory load/store instructions are difficult to craft;

(iii) it is difficult to manually write complex assembly routine;

(iv) assembly virtualization may bring issues related to MCU instruction set specifics.

The current execution context (e.g., status register) needs to be saved before accessing VM and restored afterwards, otherwise the context will be ruined during accessing VM. Complex functionalities like cache searching and management are normally implemented in C, and invoking C routines from assembly brings about function invocation overhead. Cortex-M3 MCUs support many load/store instructions and flexible addressing, so it is a big challenge to write assembly VM accessing routines for all different memory load or store instructions. Furthermore, it is extremely difficult to implement sophisticated optimizations directly in assembly without function invocation.

Through FaTVM supports assembly virtualization, C code virtualization is preferred. C source files are transformed by virtualizer and compiled to objects without assembly transformations. C code virtualization enables more flexible code transformation and we have taken advantage of it to introduce last-cache buffers to fastly locate caches for virtual

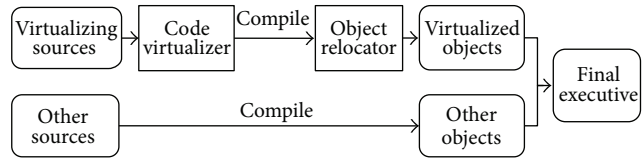


Figure 4: Program building process with FaTVM.

memory accessing. Section 5 describes details of the C code virtualization. Assembly virtualization is neglected in this paper.

4.3. Accessing Virtual Memory. FaTVM uses caches to boost virtual memory performance. A configurable number of block sized caches are used to store block data during multiple accesses. When accessing virtual memory, the page on NAND flash containing the accessing bytes is first read to cache. Then, the virtual address is converted to physical address to the corresponding bytes in cache. This conversion is called address translation. Finally the corresponding bytes in cache are read or written to accomplish the access request. Caches greatly reduce secondary storage accessing cost because FaTVM reads or writes secondary storage only when it failed finding a hitting cache. The number of cache blocks is limited by available free RAM size. It is apparent that allocating more cache blocks leads to less cache misses.

In general, application code accessing virtual memory is translated to reading or writing of the secondary storage. NAND flash must be read page by page, and they are cached in memory to accelerate multiple accessing. Virtual addresses are resolved to physical addresses located in cache blocks. The address translation progress is shown in Figure 5. Each cache block has a 4-byte field for storing per-cache control information. The structure of per-cache control field is also shown in Figure 5. The control field consists of an identify flag and a dirty flag. The identify flag is the visual address of the corresponding virtual memory block without lower bits of cache offset. The dirty bit indicates whether the cache block is written and needs to be written back to the secondary storage. According to our evaluations, address translation can be a major overhead of virtual memory in data-intensive programs. We have employed specialized optimization in C code virtualization which is able to reduce address translation overhead to a great extent.

The algorithms for cache searching and replacement have great influences on the performance of virtual memory. Rather than using set-associative caches [11], we use fully associative caches in which any cache can be mapped to a virtual memory data block. All cache blocks are linked in a universal single linked list (hereafter called cache queue). LRU is used for cache replacement algorithm. When resolving a virtual address, all cache blocks are traveled from the most recent used to the least recent used to find the matching cache. Searching in the cache queue is generally slower than searching in set-associative caches because set-associative caches can be implemented using efficient cache array data structure. Nevertheless, fully associative cache is known for

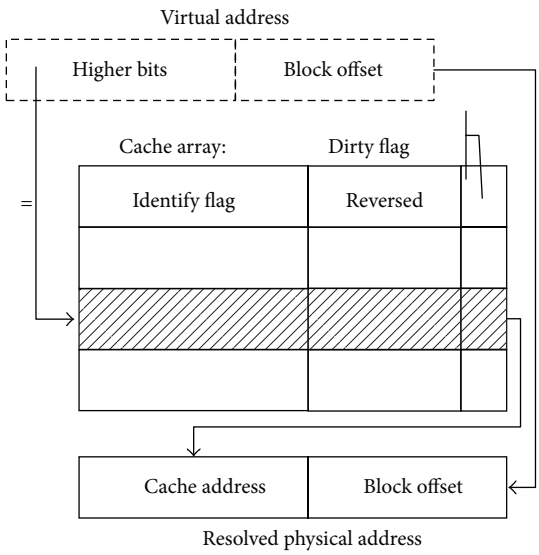


Figure 5: Address translation in FaTVM.

its lowest miss rates [11], and so it performs best in FaTVM due to high miss penalty.

CFLRU [25] and other cache replacement algorithms can also be employed easily to the current implementation; however they do not bring observable performance improvement for data processing programs according to our evaluation.

5. C Code Virtualization

We use CIL (C Intermediate Language) [26] to transform C code. Preprocessed C sources are passed to the C code virtualizer and then compiled by C compiler.

5.1. Image-Based Lvalue Virtualization. C code reads or writes memory in the form of lvalue, which means memory is accessed when any lvalue is read or written. C compilers normally translate lvalue access to assembly code that loads or stores at the start address of the specified lvalue; however in our virtual memory, memory addresses of lvalues in virtual memory are not located in physical memory, making direct load or store invalid. C code virtualization transforms C code so that lvalues in virtual memory are accessed using virtual memory routines. Generally, when reading an lvalue, the start address and size of the lvalue are passed to a predefined virtual memory routine and the corresponding bytes are read from virtual memory. Writing an lvalue is analogous to reading.

To facilitate C code virtualization, we create local variables in functions, which we call them local images. Local images are the copied values of the lvalues. Each different lvalue has its local image if the lvalue access needs to be virtualized and the type of the local image is the same as the type of the lvalue. Generally, there are two different forms of lvalue virtualization listed as belows:

(i) reading lvalue in an expression: read the lvalue to its local image and replace the lvalue with its local image in the expression;

```
vm_var = 10; vm_var is in virtual memory*/
another_var = global_var;
```

```
int local_image; // the local image of vm_var
local_image = 10;
//write back vm_var
vm_write_int(&vm_var, local_image);
//read lvalue to local image
local_image = vm_read_int(&vm_var);
another_var = local_image;
```

Figure 6: C lvalue Virtualization.

(ii) assigning lvalue in an instruction: replace the lvalue with its local image and write the local image back to the lvalue after assigning.

A simple example of C code translation is shown in Figure 6. Note that evaluating one expression may read two or more lvalues resulting in multiple virtual memory accesses.

Not all lvalues need virtualization. For those lvalues which can be inferred to be located in physical memory (e.g., local variable lvalues), virtualization is not applied. The current implementation is simply determined based on lvalue type. Global or static variables are determined to be allocated in virtual memory except those the programmer specified to be in physical memory. Local variables are determined to be in physical memory and accessed at native speed. lvalues involving pointer dereferencing (e.g., lvalue "*p") are always virtualized conservatively, because the target memory types of pointers are unknown.

5.2. Lvalue Synchronization Algorithms. Local images need to be synchronized to their lvalues when they are read or written so as not to destroy the original logic of program. Synchronization of an lvalue and its local image includes reading the value of lvalue to local image and writing the local image back to the lvalue. However, when an lvalue is used in a piece of code for multiple times, it is not necessary to synchronize the local image at all time. FaTVM uses data dependence analysis to determine if local images need to be updated or written back when they are used as substitutes of their corresponding lvalues. The algorithms of data dependence analysis for local image update and writing back are shown in Algorithms 1 and 2. A local image is read from the lvalue when it may have been unsynchronized in any execution path to the current statement. As in Algorithm 1, the local image is updated when there exists a statement which may be executed prior to the current statement and may desynchronizes the local image, and there exists an execution path from the desynchronizing statement and the current statement in which no statement synchronize the local image. The local image needs to be written back if the value of lvalue affects any subsequent execution. As in Algorithm 2, data dependence analysis is used to determine if the subsequent executions depend on the synchronization between the local image and the lvalue. These two algorithms are able to avoid superfluous local image synchronizations,

```
for each execution path to the current statement:
    may_not_initialized = true
    for each statement in execution path in reverse order:
        if the statement synchronizes the local image:
            may_not_initialized = false
            break
        if the statement desynchronizes the local image:
            return true
    if may_not_initialized:
        return true
return false
```

Algorithm 1: Algorithm to determine if reading local images from lvalues is necessary.

```
for each execution path from the current statement:
    reach_end = true
    for each statement in execution path:
        if statement depends on the synchronization of local image:
            return true
        if statement change the lvalue value:
            reach_end = false
            break
    if reach_end:
        return true
return false
```

Algorithm 2: Algorithm to determine if writing local images back to lvalues is necessary.

which become significant if the program references the same lvalue for many times.

There are four relations between code statements and lvalues referenced in the above two algorithms, listed as belows:

(i) the statement synchronizes the lvalue;

(ii) the statement desynchronizes the lvalue;

(iii) the statement depends on the synchronization of lvalue and its local image;

(iv) the statement changes the lvalue.

A statement synchronizes a local image if any one of the followings is true:

(i) the statement changes the lvalue;

(ii) the statement reads the lvalue.

For example, statement *p = val synchronizes both lvalues *p and val and its image since it changes the lvalue "*p" and reads the lvalue "val."

A statement desynchronizes a local image if any one of the followings is true:

(i) the statement changes some lvalue which is referenced in the lvalue;

(ii) the statement changes an lvalue referring to the same object with different but collided offset.

For example statement "p++" desynchronizes lvalue *p since it changes variable "p" which is referenced in lvalue "*p." Also the statement "union_.field1 = 100" desynchronizes lvalue "union_.field2" if "union_" is of union type with two fields named "field1" and "field2."

A statement depends on the synchronization of an lvalue and its local image if the effect of the statement will be incorrect when the local image is not synchronized to the lvalue. A statement changes an lvalue if the lvalue is assigned in the statement. For example, statement "p = 100" changes lvalue "p." These however involve data dependence analysis, which is difficult in the presence of pointers because pointers can cause subtle and complex data dependences. [27, 28]. We currently take a conservative strategy that assumes pointers may point to arbitrary positions in memory. Statements using pointer dereferencing lvalues depend on all lvalues and statements always depend on pointer dereferencing lvalues.

5.3. Last Cache Buffers. Data-intensive programs in sensor networks, like image processing, are generally featured by high locality in memory accessing, which has been made use of by the caches. Caching eliminates most secondary storage access; however the address translation overhead is high when VM is frequently accessed. Due to the rule of spatial locality, most successive address translations refer to the same cache block. A large portion of address translation overhead can be eliminated if redundant address translations can be suppressed.

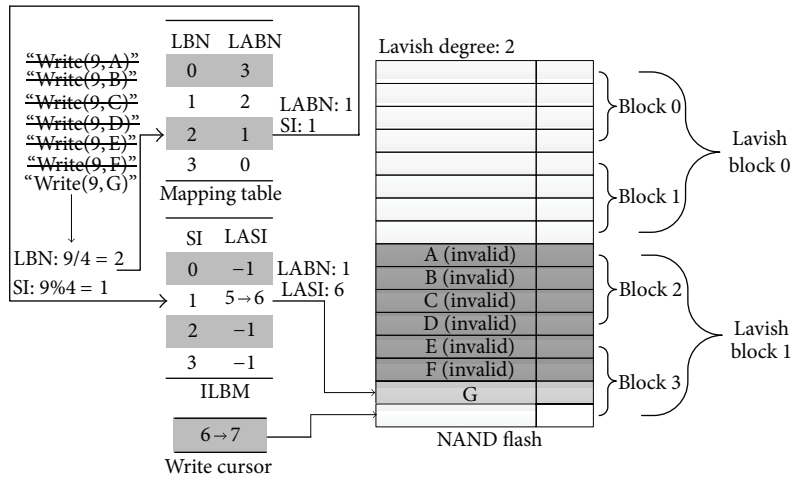


Figure 7: Address translation example of Lavish-FTL.

The flexibility of C code virtualization enables us to take advantage of this feature. Last-cache buffers are used to save the last matched cache of the resolving physical addresses while reading or writing lvalues. Last-cache buffers are variables of pointer type which are used to hold the last caches. Different last-cache buffers are used for different lvalues. Upon an lvalue access, the corresponding last-cache buffer is passed to the virtual address translation routine and the last matched cache is checked before the conventional address translation progress. If the cache block pointed to by the last-cache buffer matches the resolving virtual address, the address translation progress is highly boosted because the cache searching process is eliminated in this case.

We do not assign each lvalue an individual last-cache buffer. Different lvalues may be within the same cache block, so it would be reasonable and preferable to let two or more lvalues share one last-cache buffer. FaTVM assigns last-cache buffers to lvalues according to their "lvalue bases." The lvalue base of an lvalue can be derived using rules in Table 2. The main idea is that two lvalues of the same lvalue base should in high possibility be close to each other in memory.

Using last-cache buffers is able to eliminate most cache searching and thus reduce the cost of address translations which is responsible for a great portion of virtual memory overhead. More detailed evaluation of cost reduction benefiting from last-cache buffers is stated in Section 7.

Table 2: Lvalue base derivation rules.

Lvalue types	Derived lvalue base
variable	VarBase variable
variable.offset	VarBase variable
array (index)	VarBase array
*pointer	MemBase pointer
*(pointer + val)	MemBase pointer
*(pointer).offset	MemBase pointer

6. Secondary Storage Accessing

In this section, we describe the proposed Lavish-FTL and two adaptation schemes to address the aforementioned issues of using NAND flash as VM secondary storage.

6.1. Lavish-FTL. Lavish-FTL was specially designed for sensor nodes, to take advantage of the specific feature of NAND flash as secondary storage. The major motivation of developing Lavish-FTL is to trade NAND flash space for better access performance and less erasing. Lavish-FTL uses multiple NAND blocks to serve as one block, hereafter called a lavish block. The number of physical blocks in one lavish block is defined to be lavish degree. The larger the lavish degree is, the lower the NAND flash space utilization is. Logical block numbers (LBNs) are mapped to lavish block numbers (LABNs) rather than physical block numbers (PBNs). Figure 7 shows an example of Lavish-FTL sector address translation, where the sector with logical sector number (LSN) being 9 is written sequentially for seven times, with each write invalidating the previous write.

Sectors in a lavish block are written sequentially, from the first sector of the first physical block to the last sector of the last block. Each sector in the logical block is mapped to a logical sector of the lavish block. Since lavish blocks are written sequentially, the logical sectors in a lavish block are out of order. So, an intra-lavish-block mapping (ILBM) from logical sectors to physical sectors is required to search efficiently in lavish blocks. The ILBM contains the sector indexes in the lavish block (LASIs) for each logical sector index (SI). Note that there are normally more sectors in a lavish block than in a physical block. The ILBM information is needed for both reading and writing of sectors. The ILBM of a lavish block is saved in a sector of the lavish block itself, and a map index is maintained for each logical block to determine in which sector its latest ILBM is saved at. An example of an ILBM is also shown in Figure 7.

We use a configurable number of mapping caches to reduce the overhead of reading and writing of ILBMs. The caches are replaced using LRU algorithm. When a cache is

evicted from the LRU list, the cached ILBM is then written to the next unused sector of the corresponding lavish block, and the map index of the logical block is updated to index the last saved ILBM.

Same as the Log-FTL, we use log blocks for sector writing. That is, when a lavish block is full, a log lavish block (LLB) is allocated and subsequent writes are redirected to the log lavish block. The first lavish block is called the data lavish block (DLB). For simplicity, current implementation allows only one LLB for each DLB. The LLB increases the possibilities that the sectors in the DLB are invalidated when block erasing is required. Note that, since the ILBMs are maintained for logical blocks, there is only one ILBM for the DLB and LLB.

6.1.1. Reading Lavish Sector. Reading sector data in Lavish-FTL is simple. First, the logical sector number (LSN) is divided into LBN and the sector index. Second, the LBN is translated to LABN and the sector index is mapped to the sector in the lavish block. If the LABN is invalid or the sector was never written, the read buffer is filled with zeros. Otherwise, the corresponding sector in the lavish block is read to the read buffer.

Reading a specified sector in a lavish block is straightforward. If the sector index is smaller than the physical sector count in a lavish block (hereafter called lavish sector count), the sector is read from the DLB, and if the sector index is larger than the lavish sector count, it is read from the LLB. Figure 8 shows the lavish sector indexes for each physical sector in the DLB and its LLB.

6.1.2. Writing Lavish Sector. Writing sector data in Lavish-FTL is more complex than reading sector data. Same as the sector data reading, LSN is divided into LBN and the sector index. Second, the LBN is translated to LABN. If there is no lavish block for the writing logical block, new lavish block is allocated for the logical block and the mapping from LBN to LABN is constructed. The data is then written to the lavish block.

Lavish-FTL maintains a write cursor for each logical block as shown in Figure 7 which points to the next unused sector. The write cursor of a logical block is saved together with the ILBM. When there are unused sectors in the DLB, the data is simply written to the next unused sector. Otherwise if the DLB is full, an LLB is allocated for the DLB. If the LLB is also full, the DLB and the LLB are merged and the data is written after then. The merging details are described in Section 6.1.3.

6.1.3. Lavish Block Switching and Merging. There are two situations where NAND blocks are erased, called lavish switch and lavish merge.

When both the DLB and LLB are full in lavish block writing, a lavish switch progress is carried out so that there are free sectors in the DLB/LLB. Figure 9(a) shows an example of lavish switch. The lavish switch consists of the following:

(i) a free lavish block is allocated;

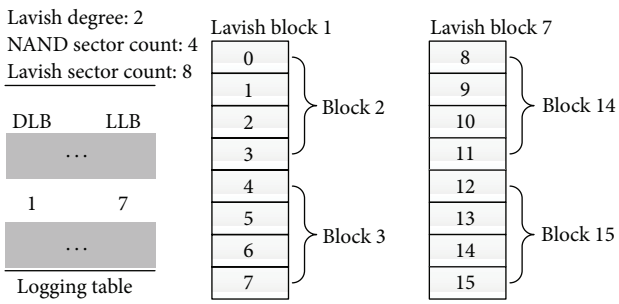


Figure 8: Lavish sector indexes of sectors in data block and log block.

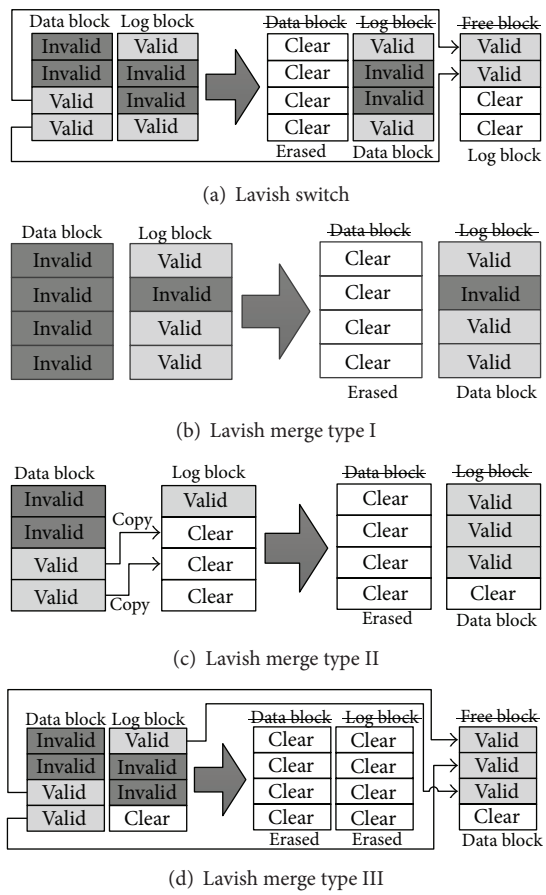


Figure 9: Demonstrations of lavish switch and lavish merge.

(ii) valid sectors in the DLB are copied to the new lavish block;

(iii) erase the DLB;

(iv) make the LLB as the DLB, and the new lavish block as the LLB.

The lavish switch is customized for writing a lavish block. Before lavish switch, the DLB always has valid sectors to copy, otherwise the DLB should have been erased by a lavish merge progress described later in this subsection. Lavish switch copies only the valid sectors in the DLB but not in LLB to

Table 3: Symbol definitions for performance analysis.

Symbol	Definition
LDEGREE	Lavish degree
SECTOR_COUNT	Count of sectors in a NAND block (typically 64)
EE	Erase efficiency of FTLs
CPBW	Number of sectors copied for every SECTOR_COUNT times of FTL writings

reduce sector copies. Since the LLB always has valid sectors before lavish switch, the new LLB always has free sectors after lavish switch even when lavish degree is 1, and thus the writing proceeds. Since the sectors in both DLB and LLB are out of order, the LLB can be used as DLB directly.

Lavish-FTL needs to make sure that there are always free lavish blocks. Lavish merge is used to reclaim free lavish blocks. There are currently three different types of lavish merge for different occasions as shown in Figures 9(b), 9(c), and 9(d).

Lavish merge type I is used when all sectors in the DLB are invalidated. In this occasion, the DLB is erased and the LLB is set as DLB. Valid sector counts of all lavish blocks are maintained in a byte array. Lavish merge type I is carried out whenever an DLB with no valid sector is found. It is similar to the "switch" operation of other FTLs.

When the number of free lavish blocks is low, lavish merge types II and III are used to reclaim free lavish blocks as shown in Figures 9(c) and 9(d). Lavish-FTL currently chooses the logical block among all logical blocks with both DLB and LLB which has the least valid sector count in the DLB. Lavish merge type II and type III differ in whether or not the LLB has enough free sectors to hold valid sectors in the DLB. The progress of lavish merge type II and type III can both be described as follows:

(i) if there are enough free sectors in the LLB to store the valid sectors in the DLB, set the old lavish block as the "writing" lavish block, otherwise a free lavish block is allocated to be the "writing" lavish block;

(ii) copy valid sectors from the DLB to the "writing" lavish block;

(iii) if the "writing" lavish block is not the LLB, copy valid sectors from the LLB to the "writing" lavish block;

(iv) erase the DLB;

(v) erase the LLB if the LLB is not the "writing" lavish block;

(vi) set the "writing" lavish block as the DLB.

6.2. Performance Analysis of Lavish-FTL. In this subsection, we analyze why Lavish-FTL can achieve better performance than other state-of-the-art FTLs. Some symbols are defined in Table 3 for clarification.

It has been stated by other researchers that the major overhead of NAND FTLs is due to the block erasing and valid sectors copying when merging multiple blocks [8, 13–15].

Erasing is generally the most expensive operation for NAND flashes. For block erasing, the best possible performance of FTLs is that each block erasing can get one free block. We define the erase efficiency to be the average number of free blocks to get per erasing. Apparently, the erase efficiency has an upper limit of 1.

Lavish-FTL has a lower bound for erase efficiency. The number of block erasing in lavish switch is the same as the lavish degree. When the lavish switch is complete, only the first NAND block of the new lavish block might be written with copied sectors and other NAND blocks were not touched. So, each lavish switch gets at least LDEGREE − 1 free NAND blocks. So the erase efficiency of Lavish-FTL has a lower limit of (LDEGREE − 1)/LDEGREE. Lavish merging has three different types. Lavish merge type I has the highest possible erase efficiency of 1. In lavish merge type II, where the valid sectors of the DLB is copied to the LLB, the number of erases is equal to the lavish degree and at most one NAND block in the LLB is used to store valid sectors from the DLB. This leads to an erase efficiency of (LDEGREE − 1)/LDEGREE, same as that of the lavish switch. In the third situation where a new lavish block is allocated to store all valid sectors from both the DLB and the LLB, the number of erasing is LDEGREE $*$ 2 and there are at most one NAND block in the LLB wasted and at most one NAND block in the final DLB used for storing valid sectors. So, the erase efficiency has a lower limit of $(2 * \text{LDEGREE} - 2)/(2 * \text{LDEGREE})$, same as the first situation and the lavish switch. So, Lavish-FTL has a lower limit of erase efficiency to be (LDEGREE − 1)/LDEGREE as shown in (1). It is easy to tell that, as the LAVISH DEGREE grows, Lavish-FTL becomes more efficient in block erasing. One has

$$\text{EE}_{\text{Lavish-FTL}} \geq \frac{\text{LDEGREE} - 1}{\text{LDEGREE}}. \quad (1)$$

The other significant overhead of FTLs is due to copying of valid sectors. This is especially the case on sensor nodes since NAND data transmission for reading and writing is time consuming. Evaluation has shown that copying of valid sectors during block merging is the major overhead of FTL implementations on sensor nodes.

Lavish-FTL was designed to reduce the count of sector copying. We define the copy-per-block-write (CPBW) to be the number of sectors copied for every SECTOR_COUNT time of FTL writing. For Lavish-FTL, there are no more than SECTOR_COUNT times of copies during lavish switch or lavish merging. Since for each logical block, the lavish switch or lavish merging are carried out per LDEGREE − 1 times of block write at most and the number of sectors copied is SECTOR_COUNT at most. We can calculate the upper bound of CPBW to be (SECTOR_COUNT)/(LDEGREE − 1), as shown in

$$\text{CPBW}_{\text{Lavish-FTL}} \leq \frac{\text{SECTOR\_COUNT}}{\text{LDEGREE} - 1}. \quad (2)$$

Another advantage of Lavish-FTL is that since a lavish block contains multiple NAND blocks, the possibility of sectors in DLB being invalidated by the time of lavish switch

or lavish merge is high, since LDEGREE ∗ SECTOR_COUNT number of sectors have been written to invalidate sectors in DLB. So the possibility of lavish merge type I being used is higher than that of the "switch" operation being used in other FTLs.

In our current implementation, lavish degree is set to 8 and SECTOR_COUNT is 64. So, the erase efficiency has a lower bound of 7/8 and CPBW has an upper bound of 64/7. In real world execution, Lavish-FTL performs better than the worst-case bounds. So, Lavish-FTL is extremely efficient and evaluation shows that Lavish-FTL is able to eliminate most extra overhead (erasing/copying) of reading and writing sectors.

6.3. Adapting Cache Size and Sector Size. As aforementioned, the best cache size is generally smaller than the sector size. So, the caches cannot be read or written directly. We have developed two schemes for addressing this issue, which are efficient in different situations, respectively.

The first scheme is called the multicaches scheme which enhances the LRU cache replacement algorithm by writing multiple caches to the lavish block at one time when writing back a dirty cache. Whenever the LRU algorithm chooses a cache to write it back, it searches in the cache queue to find dirty caches which belong to the same NAND sector with the writing back cache. When the caches cover the whole NAND sector, which means the number of caches is (sector − size)/(cache − size), these caches are written to the sector at one time. Otherwise, the missing data must be read from the sector before the caches being written to the sector. So, the possibility of caches within the same sector being found when writing back a cache is the key to the performance of this scheme. Evaluation shows that this scheme can be efficient when there is enough RAM space for caches. The evaluation details are left to Section 7.

The second scheme is called the sector-shrink scheme which reduces the sector size of FTL interface. An adapting layer is inserted between the FTL and the underlying NAND flash driver to reduce the sector size. In this way, the sector size of FTL is $1/2^n$ of the NAND flash sector size. For example, sector size of 2048 bytes can be adapted to 512 bytes. In this way, one physical NAND flash sector contains 4 adapted NAND sectors. Scheme II has its advantage and disadvantage compared to scheme I. The advantage is that since the FTL sector size is reduced, the overhead of extra reading in the scheme I can be reduced or eliminated (when the FTL sector size is adapted to be equal to the cache size). The disadvantage is that one NAND sector is written multiple times, cache by cache, which is not as efficient as writing the whole NAND sector. Evaluation shows that the sector-shrink scheme outperforms the multi-caches scheme when the number of caches is not abundant. The details are left to Section 7.

7. Evaluation

In this section, we evaluate the efficiency of FaTVM to verify its usability. We examined the elementary cost of address translation, cache performance, secondary storage accessing cost, and so forth to verify our design choices. We also evaluated the performance of the proposed FTL and sector size adaptation schemes. To evaluate the performance of real data-intensive applications, we have ported several well-known data processing algorithms to use FaTVM, and the execution results are shown and analyzed. Energy consumption of data processing programs using FaTVM is compared with other data handling options.

7.1. Evaluation Platform Description. We evaluate the secondary storage access performance on our image sensor nodes of an image sensor network. Image sensor networks acquire image data by camera, and the sensor nodes are relatively more powerful than common motes such as MicaZ, TelosB, and so forth. The image sensor nodes are equipped with 32-bit ARM Cortex MCUs from STMicroelectronics. The microcontroller has 72 MHz maximum frequency, 256 to 512 kilobytes of flash memory, and up to 64 kilobytes of SRAM. The sensor nodes are also equipped with an SLC NAND flash which is used by Lavish-FTL as the secondary storage.

7.2. Caching Evaluation. Performance metrics of caching include the following:

(i) cache searching overhead,

(ii) cache miss rate,

(iii) secondary storage accessing overhead.

Fully associative cache requires iterating in the cache queue one by one to find the matched cache. It is thus less efficient than set-associative cache. However, we do not evaluate the exact cost of cache searching because last-cache buffers in C code virtualization are able to eliminate most of cache searching overhead.

7.2.1. Cache Miss Rate. Cache miss rate is of critical importance to the performance of virtual memory since secondary access overhead is high. The miss rates for different cache configurations are shown in Figure 10. The result is acquired by emulating a memory access trace which is acquired by running JPEG to BITMAP program on FaTVM. Obviously miss rate raises as the memory footprint reduces. For each fixed RAM overhead, we examined different cache way settings to find the optimal setting. When the count of cache sets increases, the way of cache decreases. Count of cache set being 1 is equivalent to fully associative caches. The miss rates increase as the count of cache sets increases, and fully associative caches have achieved the least miss rates.

The figure shows that, with as much RAM as 32 kilobytes reserved for caches, the miss rates reached a low bound of about 0.28% for cache set counts less than or equal to 8. Set-associative caches can have smaller cache searching time than fully associative caches. So if abundant RAM is reserved for caches, the best cache configuration might not be fully associative caches, but set-associative caches. However, we did not examine cache configurations under different cache footprints. We adopted the fully associative

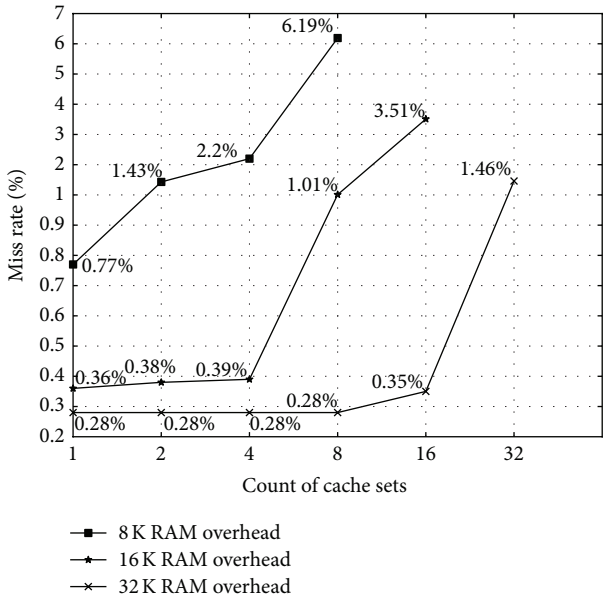


Figure 10: Cache miss rates of different configurations for JPEG to BITMAP.

cache for its simplicity and having lowest miss rates under all circumstances. Cache performance is also highly related to the executing program's work set and memory access pattern. Determining the best cache configurations for sensor node programs can be our future research work.

7.2.2. Basic Virtual Memory Access Overhead. The overhead of FaTVM accessing different memory regions is shown in Table 4. This is acquired by executing the memory operations for 2880000 times and recording the execution time. Note that the MCU frequency is 72 MHz. We also present the overhead of virtual memory access of t-kernel [9] and ViMem [10] for the ease of comparison. The data is copied directly from their published work.

Local variables in functions and static or global variables specified to reside in RAM are accessed at native speed. These variables are accessed frequently without incurring VM overhead. STM32F103ZE MCUs can execute native memory load/store instructions in 2 cycles. The measured 3 cycles include the instruction for loading the RAM address to registers. When last-cache buffers are not used, accessing virtual memory takes about 24 cycles in the best case, which means that the accessing virtual memory block is already cached and the cache is the most recent cache of the cache queue. When the accessing cache is not the most recent, multiple caches have to be travelled before finding the hitting cache, thus increasing the address translation overhead. If last-cache buffers are used, the virtual memory accessing overhead is reduced to 19 cycles when the last-cache buffer hits. Otherwise, the accessing overhead raises to be 33 cycles in the best case.

Recall that the overhead of "LCB-BestDisable" is the overhead of the best case not using last-cache buffers. If the accessing virtual memory block is not recently accessed, the extra overhead for travelling through the cache queue would incur more overhead. Although the overhead of "LCB-Hit" seems not to be much less than the overhead of "LCB-BestDisable" in best case, it eliminates much overhead of cache queue travelling.

Table 4: FaTVM basic operation overheads.

Access type	Loop	Execute time	Avg. cycles
Native	720000	30 ms	3
FaTVM (LCB-BestDisable)	720000	240 ms	24
FaTVM (LCB-Hit)	720000	190 ms	19
FaTVM (LCB-Miss)	2880000	1320 ms	33
t-kernel (AVR)	8388608	26.5 s	16
ViMem (AVR)	1	18.72 μs	17
FaTVM (SWAP)	1	3.3 ms	235714

LCB-BestDisable: best case when last-cache buffers are NOT used.
LCB-Hit: last-cache buffer hit when last-cache buffers are used.
LCB-Miss: last-cache buffer miss when last-cache buffers are used.
SWAP: swap due to cache missing.

Note that the same overhead of cache searching also exists in other virtual memory implementations. We believe this has not been emphasized in any other work because of two reasons shown below.

(i) The number of caches in the virtual memory implementations of this work is not as many as that in our implementation. Since we are expanding the available to megabytes, abundant caches must be provided to reduce the cache miss rate, making secondary storage accessing overhead tolerable.

(ii) This work tries to make the physical addresses of variables fixed, so that the address translation process can be simplified for these variables. That is, only one cache needs to be checked for validation. We argue that this technique does not work well for programs with megabytes of memory footprints. If the caches for some variables are fixed, accessing these variables would replace caches of other memory blocks, leading to higher miss rates. In-depth analysis or simulation is needed to determine the best cache assignments for variables, but it is out of the scope of this paper.

So, the last-cache buffers are used to keep address translation overhead small even when the cache is not the most recent. We evaluate the hit rates of last-cache buffers in real data processing programs, which can significantly affect the effectiveness of last-cache buffers. We also evaluate the effectiveness of last-cache buffers on real programs in later evaluation.

7.2.3. Last-Cache Buffer Hit Rate. Table 5 shows the hit rates of last-cache buffers on various programs. Two programs (i.e., FFT transformation and Red-block searching) reveal lower hit rates of more than 80%. Other programs reveal hit rates of larger than 90%. We believe that the program's hit rate of last-cache buffers is dependent on its access pattern of virtual memory. FFT transformation and Red-block searching operate on a small region of the data in virtual memory, which may frequently cross cache boundaries, invalidating

the last-cache buffer. For sequential accessing programs (i.e., BITMAP grayscaling), the hit rates can be larger than 99%.

7.3. Evaluating Flash Translation Layers. We run a selected set of typical programs using virtual memory and profile the performance of secondary storage access to test different flash translation layers. We have also implemented the Log-FTL and Superblock-FTL on our evaluation platform. 16 kilobytes of RAM are used for caches and the size of virtual memory is 8 megabytes. Table 6 shows the execution result of several programs with different FTLs. The erase count, copy count, and total execution time on secondary storage access are reported in the table. Note that neither adaptation schemes of the aforementioned mismatch issue is used in this evaluation.

Both Lavish-FTL and SuperBlock-FTL outperform Log-FTL to an extent. Although Lavish-FTL and SuperBlock-FTL have similar execution time for these programs, Lavish-FTL achieves the least erase count, guaranteeing the long-term operation of sensor nodes. The reduced ratio of erasing of Lavish-FTL compared with SuperBlock-FTL can be more than 30% for complex algorithms like JPEG decoding or Huffman compressing.

Table 8 shows the RAM footprints of the implementation of different FTLs. Lavish-FTL is most efficient in RAM footprint since run-time information is generally maintained for lavish blocks which are much fewer than the NAND blocks. The saved memory can be used for caches to further reduce the cache miss rate. Table 7 shows the execution result of the same programs using different FTLs where the whole available RAM is used for caches leaving sufficient memory for programs stacks. Apparently, Lavish-FTL is the most efficient in erasing and copying of valid sectors, and Lavish-FTL outperforms SuperBlock-FTL in execution time especially for complex programs like JPEG decoding.

7.4. Evaluating the Adaptation of Cache Size and Sector Size. We have proposed two schemes for the adaptation of cache size and sector size, because the best cache size is generally smaller than the sector size. Figure 11 shows the performance of secondary storage access with respect to different schemes running JPEG decoding. The same image used in Section 7.3 is used for evaluation. The cache size is set to 256 bytes, eighth of the NAND sector size, and Lavish-FTL is used for secondary storage access. When the RAM footprint of the caches is higher than 16 KB, multi-caches scheme outperforms sector-shrink scheme and when the RAM footprint of the caches is lower than 16 KB, the sector-shrink scheme outperforms multi-caches scheme, because multi-caches scheme would cause extra readings raising its overhead.

The secondary storage access takes about 1000 ms for both schemes when RAM footprint of the caches is 32 KB, which is about 16 times faster than the case when adaptation is not used and the cache size is set to the sector size (16382 ms as shown in Table 7).

7.5. Program Performance. We evaluate FaTVM performances using a set of typical image processing programs. The RAM footprint of caches is set to 32 kilobytes to reduce the miss rates. The size configuration of the virtual memory is 16 megabytes. The cache size is set to 512 bytes, which is an efficient setting for the programs. Last-cache buffers are used to reduce virtualization overhead. The total execution time of a program consists of the following three parts:

(i) algorithm time: execution time of the algorithm excluding memory accessing;

(ii) virtualization overhead: overhead introduced by C code virtualization;

Table 5: Last-cache buffer hit rates of programs.

Program	LCB-Hit rate
FFT transformation	80.63%
JPEG decoding	93.39%
Huffman compressing	91.99%
Red-block searching	83.33%
BITMAP grayscaling	99.87%

Table 6: FTL evaluations with 16 kB RAM for caches.

	Log-FTL	SuperBlock-FTL	Lavish-FTl
Erase count			
JPEG decoding	194	105	72
Huffman compressing	68	32	16
BITMAP grayscaling	389	295	264
BITMAP binarization	589	295	288
Copy count			
JPEG decoding	6208	761	172
Huffman compressing	2176	0	0
BITMAP grayscaling	6144	44	0
BITMAP binarization	18816	0	30
Execution time (ms)			
JPEG decoding	39344	32393	31425
Huffman compressing	10818	8076	8054
BITMAP grayscaling	65894	57910	57823
BITMAP binarization	82309	57995	58093

Table 7: FTL evaluations with all available RAM for caches.

	Log-FTL	SuperBlock-FTL	Lavish-Ftl
Cache RAM footprint	26 KB	17 KB	32 KB
Erase count			
JPEG decoding	194	105	32
Huffman compressing	34	15	0
BITMAP grayscaling	193	146	128
BITMAP binarization	294	149	136
Copy count			
JPEG decoding	6208	761	97
Huffman compressing	1088	0	0
BITMAP grayscaling	3072	44	0
BITMAP binarization	9408	0	0
Execution time			
JPEG decoding	35789	32400	16382
Huffman compressing	5456	4038	3990
BITMAP grayscaling	32909	28964	28515
BITMAP binarization	41154	28913	28631

Table 8: FTL RAM footprints.

Log-FTL	SuperBlock-FTL	Lavish-FTL
13532 bytes	23376 bytes	6276 bytes

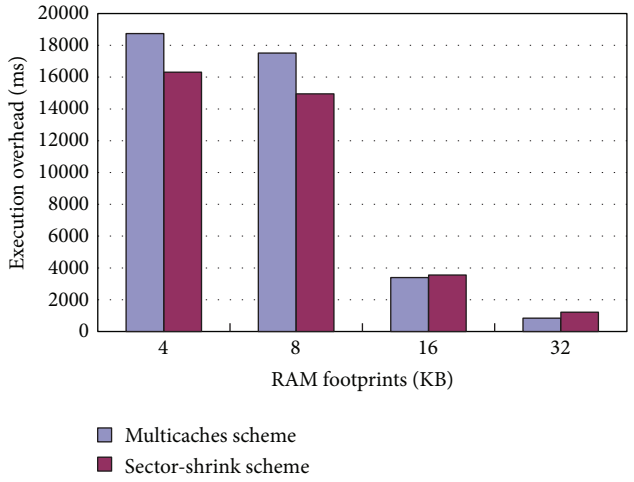


Figure 11: Execution overhead of the two adaptation schemes.

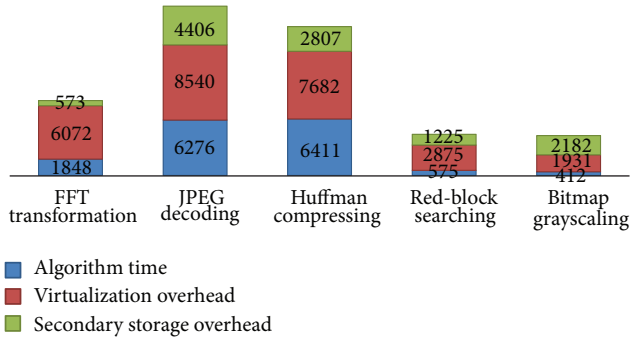


Figure 12: Execution times of typical programs (unit: ms).

Table 9: Operation conditions of the evaluation platform and Intel Mote 2.

	Intel Mote 2	Evaluation platform
MCU frequency	104 MHz	72 MHz
Sleep mode current	387 μA	25 μA
Active mode current	53 mA	37 mA
NAND operating current	N/A	15 mA
NAND stand-by current	N/A	10 μA
Battery voltage	5.5 V	3.3 V

(iii) secondary storage overhead: time spent on reading or writing NAND flash.

We measure the program's execution time using system timer interrupts. A flag is set before invoking virtual memory accessing and cleared after then. The system timer interrupts are fired per millisecond. The interrupt handler checks the flag and increases the corresponding counter by one. In this way, we can measure the execution times of algorithms and virtual memory. Figure 12 shows the execution times of programs.

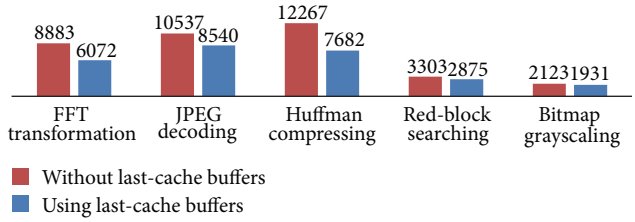


Figure 13: Effectiveness of last-cache buffers (unit: ms).

Table 10: Image sizes in JPEG format.

Image name	Image Size
big.jpg	43,584 bytes
small.jpg	27,504 bytes
tiny.jpg	3,080 bytes

Apparently, the algorithm time is different for each algorithm. JPEG decoding is the most complex among the evaluated algorithms and the algorithm of BITMAP grayscaling is the simplest. The virtualization overhead is relatively large compared with the algorithm time. For simple algorithms, the virtualization overhead is responsible for the major execution time of the programs. For complex algorithms, the virtualization overhead is smaller compared with the algorithm time and comparable to algorithm time for complex algorithms (i.e., JPEG decoding, Huffman compressing). The secondary storage overhead is much smaller than the virtualization overhead. For more complex algorithms including JPEG decoding, Huffman compressing, and FFT transformation, the secondary storage overhead is also much smaller than the algorithm time.

To verify the effectiveness of last-cache buffers, virtualization overhead of these programs without using last-cache buffers in virtual memory is measured. The result is presented in Figure 13. Last-cache buffers are able to reduce up to 40% of virtualization overhead.

7.6. Energy Consumption. It is shown in Figure 12 that the overhead introduced by FaTVM is significant compared with the algorithm time despite many optimizations we have adopted. To verify the availability of FaTVM, we compare the energy consumption of image processing programs using FaTVM with other options that process image or transmit image to powerful gateway listed below:

(1) run image processing algorithms on sensor nodes with sufficient RAM;

(2) send image to powerful gateway without processing.

We calculate and compare the energy consumption for these programs in our evaluation platform with the Intel Mote 2 platform which is equipped with 32 megabytes of SDRAM. The operation conditions of both Intel Mote 2 and our evaluation platform using STM32F103ZE is shown in Table 9. The calculation is based on following assumptions or conditions:

(i) the Intel Mote 2 runs 104/72 times faster than the evaluation platform for all algorithms;

Table 11: Image processing time of algorithms.

Image name	JPEG decoding (NAND)	Grayscale calculation (NAND)
big.jpg	11803 ms (NAND 3096 ms)	1462 ms (NAND 745 ms)
small.jpg	1118 ms (NAND 140 ms)	154 ms (NAND 76 ms)
tiny.jpg	60 ms (NAND 1 ms)	8 ms (NAND 0 ms)

(ii) energy consumption of peripheral devices is not considered;

(iii) energy consumption of other tasks (e.g., communication, storage) of sensor nodes is assumed to be irrelevant;

(iv) sensor nodes enter deep sleep mode most of the time and the extra energy consumption of NAND flash power up/down is neglected;

(v) the data processing algorithm is executed only once in the one duty cycle of the sensor node.

Then, the energy consumption (EC) of data processing and deep sleep mode of the sensor nodes regarding duty cycle (T) can be calculated using (3) and (4). $EC_{IntelMote2}$ is the energy consumption of Intel Mote 2 and EC_{FaTVM} is the energy consumption of the evaluation platform with FaTVM. AT is the algorithm time. SST is the NAND flash operation time which is equal to the secondary storage overhead of the virtual memory. TT is the total execution time using virtual memory. One has

$$\begin{aligned} EC_{IntelMote2} = {} & AT * \frac{72}{104} * 5.5\,V * 53\,mA \\ & + (T - AT) * 5.5\,V * 387\,\mu A, \end{aligned} \tag{3}$$

$$\begin{aligned} EC_{FaTVM} = {} & TT * 3.3\,V * 37\,mA \\ & + SST * 3.3\,V * 15\,mA \\ & + (T - TT) * 3.3\,V * 35\,\mu A. \end{aligned} \tag{4}$$

The duty-cycle energy consumption of previously evaluated data processing algorithms is shown in Figure 14. When the duty cycle is small enough, the Intel Mote 2 platform is more energy efficient than our evaluation platform because the execution time of data processing is significant compared with the duty cycle. However, due to higher sleep-mode current, the energy consumption of Intel Mote 2 increases much faster than that of our evaluation platform as the length of duty cycle grows. Most energy is consumed in sleep mode when the duty cycle is large enough in the Intel Mote 2 platform. The lengths of duty cycles with which two platforms have the same energy consumption are also calculated and shown in the figure, which are approximately between 5 and 20 minutes. This can actually be a threshold of the execution frequency of data processing on sensor nodes to determine if virtual memory should be employed. So, if sensor nodes have short duty cycles and data processing is executed frequently, high performance platforms like Intel Mote 2 with large SDRAM should be preferred. On the other case where data processing is infrequent, using STM32F103ZE MCU with FaTVM is about 3 times more energy efficient than using Intel Mote 2 for 1 hour duty cycles. We are developing an image sensor network to enable long-term monitoring of the surface changing of mural paintings, in which sensor nodes take photos of the mural painting per day. In scenarios like this, using FaTVM to support data processing is dramatically more energy-efficient than using Intel Mote 2 platform for sensor nodes. So, it is verified that the proposed virtual memory mechanism is helpful for constructing long-term sensor networks with long duty cycles and mass data processing.

We further compare the energy consumption of FaTVM with sensor nodes that collect and forward the sensed data to powerful gateways. Assuming an occasion in which a sensor network is constructed and sensor nodes collect images of targets, only the gray scale of the targets is interested and the image format given by cameras on sensor nodes is JPEG. There would be basically two approaches on your evaluation platform to do this task listed below:

(1) decode images in JPEG to BITMAP and calculate the gray scale of the images using FaTVM; then, send scalar gray scale values to gateways;

(2) send images in JPEG to gateways and let gateways do gray scale calculations.

Energy consumption of approach 1 contains energy consumption of image processing using FaTVM and data transmission. Since the data size of scalar gray scale values is neglectable compared with the image size in JPEG, energy consumption of transmitting scalar gray scale values is ignored. The energy consumption of FaTVM can be calculated using aforementioned equation (4) without counting in sleep mode energy consumption. Calculating energy consumption of forwarding sensed data to gateway is more troublesome in approach 2. We use data from others' work directly. It has been mentioned in [12] that the energy consumption (μJ/byte) for CC2420 [29] radio Tx/Rx is 1.8/2.1. However, it is only a theoretical minimum with 250 kbps data rate. According to [30, 31], practical observed data rate is only 40 kbps. Thus energy consumption of Tx/Rx is raised to 11.3 and 13.1 (μJ/byte). This data can be used to calculate the energy consumption of single-hop data transmission. In real sensor networks, reliable multihop transmission, network collision, and so forth can impose significant overhead to network transmission. So the energy consumption of network transmission in real sensor networks is much higher than that of single-hop data transmission.

We consider three different sizes for JPEG image, as shown in Table 10. The image processing time is shown in Table 11. So, the energy consumption for processing or transmitting each image can be calculated. The result is shown

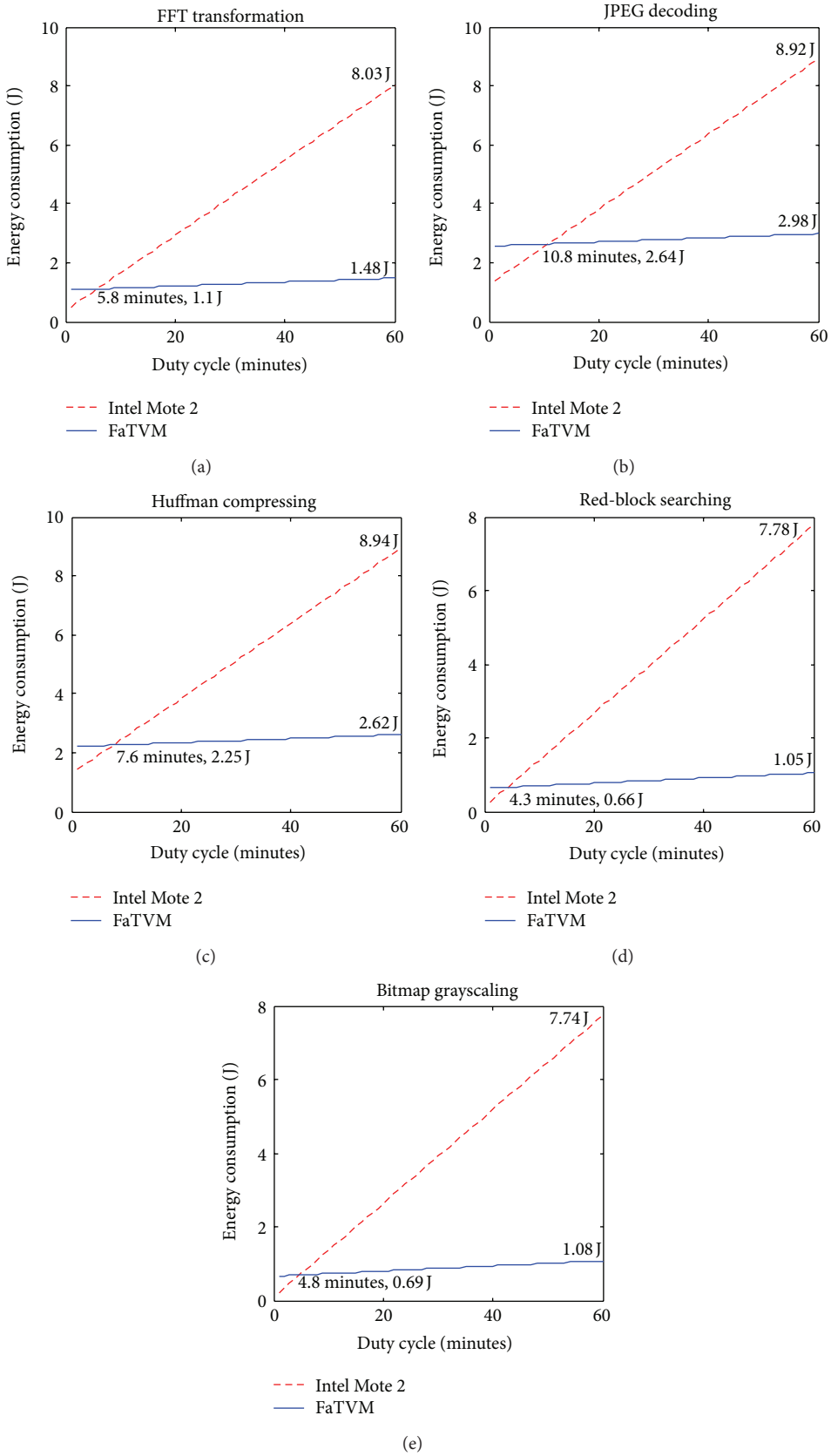


Figure 14: Program energy consumptions of one duty cycle.

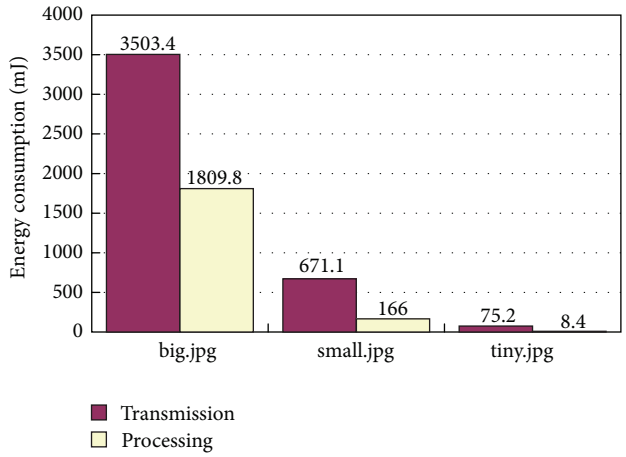


Figure 15: Energy consumption of image transmission or processing.

in Figure 15. Despite the JPEG decoding, data processing is still much more energy efficient than network transmission. Energy consumption of image network transmission is 2 times that of image processing for big images and more than 4 times that of image processing for smaller images. Recall that the calculated energy consumption of network transmission is just for one hop. Energy consumption of image transmission can be multiples of that in real-world sensor networks.

8. Conclusion and Future Work

We have designed and implemented a virtual memory named FaTVM for data-intensive applications in sensor networks, making it possible for sensor nodes to carry out complex computation with heavy memory footprint without using energy-hungry MCUs with large RAM. We have been focusing on reducing virtual memory overheads in different aspects including cache management, address translation, secondary storage accessing, and so forth, to achieve acceptable memory performance for sensor network applications. We use C code virtualization to transform C source code to work with virtual memory. Last-cache buffers are used to accelerate address translations which contributes to a great portion of virtual memory overhead. FaTVM uses NAND flashes as secondary storage. Lavish-FTL is proposed to reduce NAND flash accessing time and block erasing, so as to prolong NAND flash lifetime. Two adaptation schemes were introduced to write caches back to NAND sectors efficiently. Our evaluation shows that the virtual memory overhead is comparable to the algorithm time for typical data processing algorithms on image sensor nodes. The secondary storage overhead has been reduced to be the smaller part of the virtual memory overhead. We compared the energy consumption of the evaluation platform using FaTVM with the Intel Mote 2 platform under reasonable assumptions. The virtual memory solution is proven to be much more energy-efficient for mass data processing programs on sensor nodes with long duty cycles than high performance platforms with large SDRAM. Energy consumption of data processing programs using FaTVM is also proven to be much less than that of large data transmission.

Despite the optimizations we have adopted to reduce virtual memory overhead on various aspects, the overhead introduced by code virtualization is still large compared with the algorithm time. Reducing the overhead introduced by code virtualization without increasing cache misses can be our future research direction. Fat pointers are used to further reduce virtualization overhead by storing cache information for memory accessing. The memory access pattern of sensor nodes is explored to improve cache performance. We have also noticed that it is difficult to determine the best VM configuration for performance and energy efficiency. Finding an automatic solution to find the optimum VM configuration can further improve the utility of FaTVM.

Acknowledgments

This work was supported by the "Advanced Sensor Network Platform Research" Project of Zhejiang University, China (no. 2010X88X002-11), and the National High Technology Research and Development Program of China (Grant no. 2012AA101701).

References

[1] R. M. Kling, "Intel motes: advanced sensor network platforms and applications," in *Proceedings of the IEEE/MTT-S International Microwave Symposium*, pp. 365–368, June 2005.

[2] P. J. Denning, "Virtual memory," *ACM Computing Surveys*, vol. 2, pp. 153–189, 1970.

[3] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "Tinydb: an acquisitional query processing system for sensor networks," *ACM Transactions on Database Systems*, vol. 30, no. 1, pp. 122–173, 2005.

[4] STMicroelectronics, "Stm32f103ze datasheet," 2011, http://www.st.com/st-web-ui/static/active/en/resource/technical/document/datasheet/CD00191185.pdf.

[5] "Atmega128 datasheet," http://www.atmel.com/.

[6] "Texas instruments: Msp430 datasheet," http://focus.ti.com/lit/ds/symlink/msp430f1101a.pdf.

[7] A. Ltd, "Cortex-m3 technical reference manual," 2010, http://infocenter.arm.com/help/topic/com.arm.doc.ddi0337i/DDI0337I_cortexm3_r2p1_trm.pdf.

[8] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song, "A survey of flash translation layer," *Journal of Systems Architecture*, vol. 55, pp. 332–343.

[9] L. Gu and J. A. Stankovic, "*t-kernel*: providing reliable OS support to wireless sensor networks," in *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, pp. 1–14, 2006.

[10] A. Lachenmann, P. J. Marrón, M. Gauger, D. Minder, O. Saukh, and K. Rothermel, "Removing the memory limitations of sensor networks with flash-based virtual memory," *SIGOPS Operating Systems Review*, vol. 41, pp. 131–144, 2007.

[11] J. Hennessy, D. Patterson, and D. Goldberg, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 2003.

[12] G. Mathur, P. Desnoyers, D. Ganesan, and P. Shenoy, "Ultra-low power data storage for sensor networks," in *Proceedings of*

the 5th International Conference on Information Processing in Sensor Networks (IPSN '06), pp. 374–381, ACM, New York, NY, USA, 2006.

[13] C-.H. Wu, T-.W. Kuo, and C-.L. Yang, "A space-efficient caching mechanism for flash-memory address translation," in *Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC '06)*, pp. 64–71, IEEE Computer Society, Washington, DC, USA, 2006.

[14] S. Lee, W. Choi, and D. Park, "Fast: an efficient flash translation layer for flash memory," *Emerging Directions in Embedded and Ubiquitous Computing*, pp. 879–887, 2006.

[15] J-.U. Kang, H. Jo, J-.S. Kim, and J. Lee, "A superblock-based flash translation layer for nand flash memory," in *Proceedings of the 6th ACM & IEEE International Conference on Embedded Software (EMSOFT '06)*, pp. 161–170, ACM, New York, NY, USA2006.

[16] N. Lin, Y. Dong, and D. Lu, "Fast transparent virtual memory for complex data processing in sensor networks," *SENSORNETS*, pp. 24–34, 2012.

[17] B. Jacob and T. Mudge, "Uniprocessor virtual memory without tlbs," *IEEE Transactions on Computers*, vol. 50, pp. 482–499, 2001.

[18] L. S. Bai, L. Yang, and R. P. Dick, "Memmu: Memory expansion for mmu-less embedded systems," *ACM Transactions on Embedded Computing Systems*, vol. 8, no. 3, article 23.

[19] T. T. 2.x Working Group, "Tinyos 2.0," in *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems (SenSys '05)*, p. 320, ACM, New York, NY, USA, 2005.

[20] D. Gay, M. Welsh, P. Levis, E. Brewer, R. Von Behren, and D. Culler, "The nesC language: a holistic approach to networked embedded systems," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '03)*, pp. 1–11, New York, NY, USA, June 2003.

[21] Y. T. Chen, T. C. Chien, and P. H. Chou, "Enix: A lightweight dynamic operating system for tightly constrained wireless sensor platforms," in *Proceedings of the 8th ACM International Conference on Embedded Networked Sensor Systems (SenSys '10)*, pp. 183–196, New York, NY, USA, November 2010.

[22] S. Soro and W. Heinzelman, "A survey of visual sensor networks," *Advances in Multimedia*, vol. 2009, Article ID 640386, 21 pages, 2009.

[23] Crossbow Technology, "Micaz specs," 2009, http://bullseye.xbow.com:81/Products/Product_pdf_files/Wireless_pdf/MICAz_.Datasheet.pdf.

[24] "Telosb specs," 2009, http://www.willow.co.uk/TelosB_Datasheet.pdf.

[25] S.-Y. Park, D. Jung, J.-U. Kang, J.-S. Kim, and J. Lee, "Cflru: a replacement algorithm for flash memory," in *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '06)*, pp. 234–241, ACM, New York, NY, USA, 2006.

[26] G. Necula, S. McPeak, S. Rahul, and W. Weimer, "Cil: intermediate language and tools for analysis and transformation of c programs," in *Compiler Construction*, R. Horspool, Ed., vol. 2304 of *Lecture Notes in Computer Science*, pp. 209–265, Springer, Berlin, Germany, 2002.

[27] D. C. Atkinson and W. G. Griswold, "Effective whole-program analysis in the presence of pointers," in *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '98/FSE-6)*, pp. 46–55, ACM, New York, NY, USA, 1998.

[28] S. Horwitz, P. Pfeiffer, and T. Reps, "Dependence analysis for pointer variables," in *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation (PLDI '89)*, pp. 28–40, ACM, New York, NY, USA, 1989.

[29] T. Instruments, "Cc2420 datasheet," 2007.

[30] J. Polastre, J. Hill, and D. Culler, "Versatile low power media access for wireless sensor networks," in *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys '04)*, pp. 95–107, ACM, New York, NY, USA, 2004.

[31] K. Kumar and P. Kumar, "Tmote Implementation of bmac and smac Protocols," 2011.