FAST TRANSPARENT VIRTUAL MEMORY FOR COMPLEX DATA PROCESSING IN SENSOR NETWORKS

Nan Lin[1], Yabo Dong[1,2] and Dongming Lu[1,2]
[1]*College of Computer Science and Technology, Zhejiang University, Hangzhou, China*
[2]*Cyrus Tang Center for Sensor Materials and Applications, Zhejiang University, Hangzhou, China*

Keywords: Virtual Memory, Memory Cache, Code Transformation, Sensor Networks.

Abstract: Sensor networks has been used in numerous fields, some of which require complex processing of large amount of data, such as visual surveillance. Low-power MCUs on sensor nodes have kilobytes of RAM which is not enough for mass data processing. In this paper, we present a transparent software-based virtual memory for mass complex data processing in sensor networks which can be much larger than the physical memory. Traditional optimizations have been employed and tuned to best suit the virtual memory. Address translations were found to be a major overhead in virtual memory for most applications. With respect to assembly conversion commonly used in other software-based virtual memory systems, we use flexible C code transformation which generates cache index buffers to greatly reduce the address translation overhead. The usability of the virtual memory is further verified by a number of common algorithms using the virtual memory.

1 INTRODUCTION

As sensor nodes continue to evolve, sensor networks are used in more and more regions. Sensor nodes now are able to collect much more data from the environment. For example, visual sensor nodes acquire kilobytes or megabytes of image/video data. In-situ data processing might be required to reduce network traffic or to provide real-time response. Complex mass data processing would require a large amount of memory space for data storage and processing, however traditional sensor nodes are generally highly energy-constrained with only kilobytes of RAM.

The straightforward solution is to use a sensor node platform powered by an MCU with a large SDRAM. Unfortunately, SDRAM has been known to be more expensive and energy-consuming per bit compared to other memory mediums (NOR FLASH, NAND FLASH, et al.). So it would be extremely difficult for sensor networks to provide sufficient RAM space for mass data processing while still be within acceptable range of power consumption and hardware cost. For example, Intel Mote 2(Crossbow Technology, 2007) is a sensor platform with increased CPU performance and improved radio bandwidth to acquire, process and transmit large data streams. The platform provides 32MB SDRAM on-board, however the current in deep sleep mode is about 390μA which is much larger than that of traditional motes (about 10μA) (Crossbow Technology, 2009a; Crossbow Technology, 2009b). This would greatly reduce the lifetime of the sensor nodes. So, sensor platforms with large RAM space are not suitable for long-time surveillance.

Virtual memory has been used in traditional operating systems to extend usable memory for applications(Denning, 1970). With MMU (memory management unit), virtual memory can provide each application a flat memory space which can be much larger than the actual physical memory size. Virtual memory however can not be applied to sensor networks directly. Sensor node platforms usually use non-MMU processors to reduce power consumption and hardware cost, thus hardware aided virtual memory is normally not an option for sensor networks. Software based virtualizations are usually used to provide virtual memory support for sensor networks(Gu and Stankovic, 2006; Lachenmann et al., 2007).

Software based virtual memory may involves calculations or external storage accesses which may slow down node code execution considerably. In traditional sensor networks where nodes are featured by burst code execution, this problem can be neglected as long as the node interactivity performance is not affected significantly(Gu and Stankovic, 2006). However, when mass data is acquired and processed, vir-

Lin N., Dong Y. and Lu D..
FAST TRANSPARENT VIRTUAL MEMORY FOR COMPLEX DATA PROCESSING IN SENSOR NETWORKS.
DOI: 10.5220/0003804900240034
In *Proceedings of the 1st International Conference on Sensor Networks* (SENSORNETS-2012), pages 24-34
ISBN: 978-989-8565-01-3

tual memory overhead will be significant and takes a great proportion of execution time. Since processing power and energy supply of sensor nodes are limited, sensor nodes can not afford long-time high intensity processing or external storage accessing. So, virtual memory for data-intensive sensor networks must be highly efficient in code execution and external storage accessing.

Virtual memory for mass data processing in sensor networks should be transparent for programmers. Existing algorithm implementations should be usable immediately or can be used with minor modifications. This can greatly reduce the software development cost, especially for some sophisticated data processing algorithms (e.g. image compression & decompression). Complex data processing algorithms may employ a good deal of pointer manipulation and de-referencing. Some software-based virtual memory however may not allow accessing virtual memory using pointers or virtual memory pointer calculation(Lachenmann et al., 2007). Transparency of virtual memory pointers manipulations and de-referencing makes programmers much easier to port or implement complex data processing algorithms. Virtual memory should be able to tell pointers of virtual memory from pointers of physical memory(RAM) and handle them accordingly. It will be a significant burden if programmers have to be aware of the destination memory type of pointers.

We have developed a virtual memory system named FaTVM (Fast Transparent Virtual Memory) for sensor networks on a platform with ARM Cortex-M3 MCU(Ltd, 2010). FaTVM aims to provide sensor network data processing programs an efficient memory space which is much larger than the physical memory. We explored among design choices to search for the most suitable configurations for data processing programs in different parts of virtual memory, including cache management, code transformation, etc. Our contribution is two-fold: we have developed an efficient virtual memory system for ARM Cortex-M3 MCUs, and we show how C code transformation can be used to realize more flexible optimizations to reduce virtual memory overhead.

The rest of this paper is organized as follows. Section 2 gives an overview of FaTVM. Section 3, 4, 5 describes important components of FaTVM including cache management, assembly virtualization and C code virtualization. Some implementation issues we encountered are illustrated in Section 6. Evaluation of FaTVM performance is stated and described in Section 7. Section 8 discuss related work and possible improvements and future work of FaTVM are discussed in Section 9.

2 OVERVIEW

We describe the overview of FaTVM in several aspects. Subsection 2.1 describes our developing hardware platform of FaTVM. Subsection 2.2 explains how source code is virtualized and compiled to final executive. Subsection 2.3 describes how virtual memory are accessed in programs.

2.1 Platform Description

We have been developing FaTVM on our image sensor nodes of an image sensor network(Soro and Heinzelman, 2009; Soro and Heinzelman, 2009). Image sensor networks acquire image data by camera, and the sensor nodes are relatively more powerful than common motes such as MicaZ(Crossbow Technology, 2009a), TelosB(Crossbow Technology, 2009b), etc. The image sensor nodes are equipped with 32-bit ARM Cortex MCUs(STMicroelectronics, 2011) from STMicroelectronics. The microcontroller has 72MHz 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 still be straightforward to port FaTVM to other platforms, since FaTVM does not employ any advanced MCU features.

The sensor nodes are also equipped with Micro-SD card for saving sensed pictures. The size of Micro-SD card can be up to gigabytes. FaTVM uses a dedicated SWAP partition of Micro-SD as the secondary storage. Micro-SD card is more convenient to use than NAND flash since it has a built-in FTL(Flash Translation Layer)(Chung et al., 2009) which handles flash block erasing and wear-leveling.

STM32 MCU has a 4GB memory map, in which memory address ranges of different sizes are allocated for Code, SRAM, Peripherals, External RAM, etc. 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 sub-range of 1GB External RAM addresses to be used by virtual memory, making it possible to determine address type to be virtual or physical by checking if the address is within the dedicated address space.

2.2 Code Virtualization

FaTVM are developed without any assumption of the underlying operating system. Program code written in C or assembly are transformed to access virtual memory. This progress is called as code virtualization.

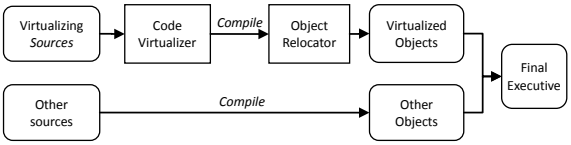


Figure 1: Program building process with FaTVM.

Figure 1 shows how source files are virtualized and compiled to objects, which are relocated and linked to the final executive. The programmer is responsible for specifying which source files to be virtualized thus it is possible to wrap only a selected set of modules. Code virtualizer transform source code written by programmers who may not be aware of the virtual memory to work with FaTVM. Object relocater moves all variables in physical data section to VM data section. It is still possible for the programmers to manually specify variables to be allocated in physical memory. This may bring significant performance improvement if the variables are accessed frequently in the program.

Assembly virtualization are commonly used in other software-based virtual memory mechanisms. Memory load or store instructions 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.

- The overhead of assembly virtualization is high;
- The replacements for memory load/store instructions are difficult to craft;
- It is difficult to manually write complex assembly routine;
- Assembly virtualization may brought issues related to MCU instruction set specifics. More details about this can be found in section 6.

The current execution context(e.g. status register) needs to be saved before accessing VM and restored afterwards if 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. STM32 MCUs supports many load/store instructions and flexible addressing, so it is a challenge to write assembly VM accessing routines for all different memory load or store instructions. Further more, it is extremely difficult to implement sophisticated optimizations directly in assembly, avoiding function invocation.

FaTVM introduced C code virtualization. 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 this to construct more complex routines for accessing VM with further optimizations. Section 5 describes details of the C code virtualization. Yet, FaTVM still supports assembly virtualization, which can be a comparison for C code virtualization, and is able to virtualize compiled libraries when the source code is missing.

2.3 Accessing Virtual Memory

FaTVM uses caches to boost virtual memory performance. When accessing virtual memory, the Micro-SD block containing the accessing bytes are 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 to find a usable cache. The cache organization is described at length in section 3. According to our evaluations, address translation rather than secondary storage accessing is responsible for the 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.

3 CACHE MANAGEMENT

A configurable number of block sized caches are used to store block data during multiple accesses. Since Micro-SD card is read or written on a block basis, each cache is of the size of a Micro-SD block (typically 512 bytes), which we call it a cache block, to facilitate cache fetching and writing back. The cache data can be fetched from or written back to Micro-SD card by reading or writing one block. The number of cache blocks is limited by available free memory size. It is apparent that allocating more cache blocks leads to less cache misses.

3.1 Address Translation

In general, application code accessing virtual memory is translated to reading or writing of Micro-SD. Micro-SD must be read block by block, 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 are shown in figure 2. Each cache

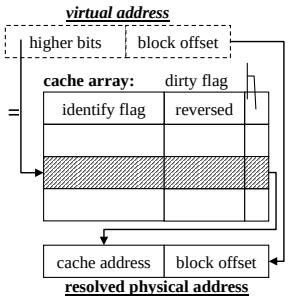


Figure 2: Address translation in FaTVM.

block has a 4-byte field for storing per-cache control information. The structure of per-cache control field is also shown in figure 2. 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 the lower bits of zero. The dirty bit indicates whether the cache block is written and needs to be written back to the secondary storage.

3.2 Cache Searching and Replacement

The algorithms for cache searching and replacement have a great influence on the performance of virtual memory.

Rather than using set associative caches(Hennessy et al., 2003), we use fully associative caches in which any cache can be mapped to any Micro-SD block. Figure 3 shows the data structures for managing cache blocks. All cache blocks are linked in the universal double linked list. The order of cache blocks in the universal double linked list is determined by the adopted cache replacement algorithm. When resolving a virtual address, all cache blocks are traveled in specific order to find the matched cache. Searching among fully associative caches are generally slower than searching among set associative caches because set associative caches can be implemented using efficient cache array data structure. Nevertheless, fully associative cache are known for its best (lowest) miss rates(Hennessy et al., 2003), and so it performs best in FaTVM because the miss penalty is high.

Despite the fully associative caches, FaTVM uses set associative lists to accelerate cache searching. Cache blocks within the same set are hashed to one of the cache set double linked lists accordingly to enable faster searching for a specified cache. The lists are also shown in figure 3.

We use LRU algorithm for cache replacement. Cache blocks in the universal double linked list are ordered by the their recent used times. CFLRU(Park et al., 2006) can also be implemented easily using the universal double linked list, however it does not bring performance improvement for many programs according to our evaluation.

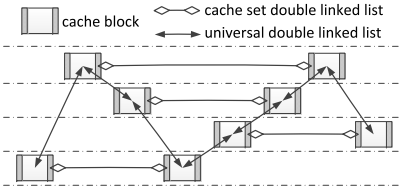


Figure 3: Cache management data structures for searching.

3.3 Optimizing Micro-SD Accessing

We have employed a number of effective optimizations for Micro-SD accessing.

Micro-SD card supports asynchronous operations. Asynchronous accessing saves secondary storage accessing time by reading or writing Micro-SD simultaneously while the program is doing other computation.

SD card is read and written on block basis, and writing multiple conjoint blocks in one call can be much more efficient than writing blocks one by one. FaTVM write multiple conjoint dirty blocks when possible. When writing a dirty block back (the block is being replaced), FaTVM travel through subsequent blocks to find contiguous dirty blocks, whose both cache index and virtual memory address are contiguous. Evaluation shows that this can be a significant cost reduce in SD card operations.

4 ASSEMBLY VIRTUALIZATION

Assembly virtualization is a straightforward method for implementing software-based virtual memory. In STM32 MCU, memory access instructions include the LDR and STR instructions and their numerous variants. In this section, we describe the wrapping of LDR/STR instructions and some optimizations for reducing address translation overhead.

FaTVM assembly virtualizer replaces LDR/STR instructions with customized routines which access the virtual memory at specified addresses. Figure 4 shows an simplified assembly wrapping example. The replacing assembly routine is responsible for the follow tasks.

1. Save and restore APSR (application status register);
2. Determine if the address is in virtual memory, and branch if accessing physical memory;
3. Save and restore registers used in this routine;

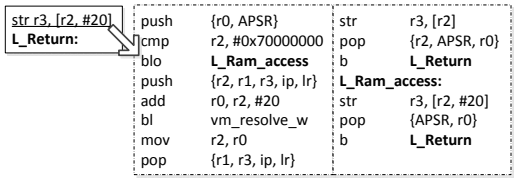


Figure 4: Wrapping an STR instruction.

4. Invoke the address translation routine written in C;
5. Execute memory operation using the translated physical address.

Cortex-M3 MCUs support many different LDR/STR instructions and flexible addressing. Currently, FaTVM assembly virtualizer generates different assembly wrapping routines for different instructions. The wrapping routine generator is carefully crafted to handle all LDR/STR instructions correctly. This is highly correlated with the MCU instruction set, and we leave out the details in this paper.

Load or store instructions in assembly includes accessing of both virtual memory and physical memory. However in assembly, the semantics of C source can not be accessed, making it not straightforward to determine if virtualization is needed. We have developed a simple intraprocedural data flow analysis to analyze to derive register values in instructions within procedures. We maintain a register set description(RSD) for each instruction in all assembly functions, in which the possible values for each register (in STM32 register table) are saved. Using the derivation rules of instructions, RSD of one instruction can be derived from RSD of its preceding instructions. For each function in assembly, the algorithm starts at the function entry by setting up RSD for the sentry instruction and analyze instructions until RSDs not change anymore. For a load/store instruction, the possible values of the source register can be checked to determine the destination memory type. We leave out the implementation details of this analyzation algorithm in this paper.

To further take the advantages of the high locality of data-intensive sensor node program behavior, we save the most recent address resolution to aid subsequent accesses to the same block. We use two fixed registers to save source address and destination address of the last address resolution. Upon a address resolution, the resolving virtual address is saved in one fixed register and the resulting address is saved in the other fixed register. In subsequent address resolving , a fast match computation is carried out, and if the current resolving virtual address is within the same block with that of the last address resolution, the physical address in cache can be easily computed using the last resolution result.

5 C CODE VIRTUALIZATION

We use CIL (C Intermediate Language)(Necula et al., 2002) 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 space in the form of lvalue, that is memory is accessed when any lvalue is read or written. C compilers normally translate lvalue access to assembly code that load or save 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 fail. 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 of the lvalue and the 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 facilitate C code virtualization, we create local variables in functions for lvalues, 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 same as the type of the lvalue. Generally, there are two different forms of lvalue virtualization listed as bellows.

- Reading lvalue in expression: read the lvalue to its local image and replace the lvalue with its local image in the expression;
- Assigning lvalue in 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 5. Note that evaluating one expression may reads two or more lvalues resulting a number of 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 lvalue), virtualization is not applied. The current implementation simply decide 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

```
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 5: C lvalue virtualization.

physical memory and accessed at native speed. Lvalues involving pointer de-referencing (e.g lvalue "*p") are always virtualized conservatively, because the target memory types of pointers are unknown.

5.2 Lvalue Synchronization Algorithms

Local images needs to be synchronized to their lvalues when it is read or written so as not to destroy the original logic of program. Synchronization of an lvalue and its local image includes reading value of lvalue to local image and writing 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 figure 6 and figure 7. A local image is read from the lvalue when it may have been unsynchronized in any execution path to the current statement. As in figure 6, the a local image is updated when there exists a statement which may be executed prior to the current statement and may desynchronize 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 affect any subsequent execution. As in figure 7, 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, which becomes significant if 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 bellows.

- the statement synchronizes the lvalue;
- the statement desynchronizes the lvalue;
- the statement depends on the synchronization of lvalue and its local image;

```
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
```

Figure 6: 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
```

Figure 7: Algorithm to determine if writing local images back to lvalues is necessary.

- the statement changes the lvalue.

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

- the statement changes the lvalue;
- the statement read 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.

- the statement changes some lvalue which is referenced in the lvalue;
- 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. (Atkinson and Griswold, 1998; Horwitz et al., 1989). We currently take a conservative strategy that assumes pointers may point to arbitrary positions in memory. Statements using pointer de-referencing lvalues depend on all lvalues and statements always depend on pointer de-referencing lvalues.

5.3 Cache Index Buffering

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 spatial locality, most successive address translations refers to the same cache block. A large portion of address translation overhead can be eliminated if redundant address translations can be suppressed.

The flexibility of C code virtualization enables us to take further advantage of this feature. Rather than saving the last resolved address as in assembly virtualization, we use multiple cache index buffers to save the cache indexes of the resolving physical addresses while reading or writing lvalues. Cache index buffers are variables of byte type which are used to hold the cache indexes. Different cache index buffers are used for different lvalues. Upon an lvalue access, the corresponding cache index buffer are passed to the virtual address translation routine and the cache index in the buffer is checked before the original address translation progress. If the cache block indexed by the cache index buffer still matches the resolving virtual address, the address translation progress is highly boosted because the physical address to caches can be computed from cache index directly.

We do not assign each lvalue different cache index buffers. Different lvalues may be within the same cache block, so it would be reasonable and preferable to let two or more lvalues share one cache index buffer. FaTVM assign cache index buffers to lvalues according to their ”lvalue bases”. The lvalue base of an lvalue can be derived using rules in table 1. The main idea is that two lvalue of same lvalue base should in high possibility be close to each other in memory.

Low level functions are usually called by upper level functions for multiple times during the execution. For example, image convertion routine may call DCT transformation function for each continuous data block. So, it is obvious that lvalues accessed in different DCT transformation function invocations are contiguous in memory. We make cache index buffers to be global so that cached indexes will not be lost crossing function calls. Since the size of a cache index buffer is just one byte, the RAM cost of cache index buffers is acceptable.

Table 1: 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

Using cache index buffers is able to eliminate much unnecessary address translations thus reduce cost of address translations which is responsible for a great portion of virtual memory overhead. More detailed evaluation of cost reduction benefit by cache index buffering is stated in section 7.

5.4 Virtualization Inlining

Part of lvalue virtualization code is simple enough to inline them, including the checking of the destination address to determine if it is a virtual address, and the lvalue accessing routine using cache index buffer. The inlining can eliminate function invocations whenever block index is hit. Supporting more complex inlining shows superiority of C code virtualization compared to assembly virtualization.

6 IMPLEMENTATION ISSUES

In this section, we discuss some subtle implementation issues that we came across during the development of FaTVM. The issues may relate to the underlying micro-processor or instruction set.

Variables in virtual memory need to be initialized just like that in physical memory. FaTVM initializes data sections in virtual memory when booted. Although this may take considerable time if data sections are large, it is not a severe problem because it is carried out only when the node is booting.

Cortex-M3 supports unaligned memory access to achieve flexible memory operations. However, this brings extra difficulty to the development of FaTVM. In the presence of unaligned memory access, one load or store instruction may access a range of memory crossing boundary of two successive blocks(cross-block accessing). We have addressed this issue by read the adjacent Micro-SD block of the accessing

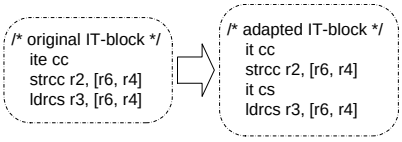


Figure 8: Dividing an IT-block into two IT-blocks.

Micro-SD block to the adjacent cache before executing the final load or store instruction. This may increase cache misses slightly since the adjacent cache has to be re-associated. Fortunately this condition is rare in program execution according to our observations and can be eliminated by careful data structure organization. The overhead of handling cross-block accessing is negligible.

If-Then condition(IT) instruction[1] in Cortex-M3 thumb2 instruction set had brought difficulties in implementing assembly virtualization. LDR/STR instructions in IT block can not be translated if it is not the last instruction inside the IT block because a branch or any instruction that modifies the PC must either be outside an IT block or must be the last instruction inside the IT block. To address this problem, one IT-block containing LDR/STR instructions will be divided into two or more IT-blocks if necessary to make sure that LDR/STR instructions are always the last instructions in IT-blocks. An instance of transformation is shown in figure 8. The STR instruction is not the last instruction of the original IT-block, requiring the IT-block be transformed. This is an issue specific to ARM instruction sets with IT-block.

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, etc to verify our design choices.

To evaluate the performance of real data-intensive applications, we have ported several well-known data processing algorithms to use FaTVM, and the execution result are shown and analyzed.

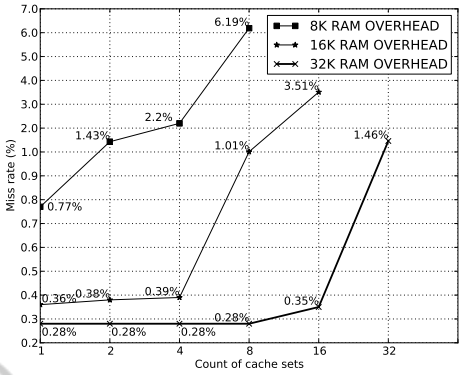
7.1 Caching Evaluation

Performance metrics of caching includes:

- cache searching overhead,
- cache miss rate,
- secondary storage accessing overhead.

[1]The IT (If-Then) instruction makes up to four following instructions (the IT block) conditional. The conditions can be all the same, or some of them can be the logical inverse of the others. (cited from ARM Compiler toolchain Assembler Reference).

Table 2: Cache miss rates of different configurations.



fully associative cache requires searching in a cache list 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 cache searching overhead is minor compared to secondary storage accessing overhead, and in C code virtualization, cache index buffering is able to eliminate most of cache searching overhead.

7.1.1 Cache Miss Rate Evaluation

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 2. The result is acquired by emulating a memory access trace which is acquired by running JPEG to BITMAP program on FaTVM. Obviously miss rate raise as the memory footprint reduce. For each fixed RAM overhead, we examined different cache way settings to find the optimal setting. When the count of cache sets increase, the way of cache decrease. Count of cache set being 1 is equivalent to fully associative caches. The miss rates increase as the count of cache sets increase, and fully associative caches has achieved the least miss rates.

7.1.2 Basic Memory Operation Evaluation

The overhead of FaTVM accessing different memory regions is shown in table 3.

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. Memory access (without swap) using assembly virtualization takes 207 cycles, which is the total overhead of assembly wrapping, address translation and RAM ac-

Table 3: FaTVM basic operation overheads.

Access Type	Count	Time	Cycles
Native	10M	417 ms	3.0
AV	10M	28787 ms	207.3
AV-swap	1	3.3 ms	240637.5
CV-CIBHit	10M	16968 ms	54.1
CV-CIBMiss	10M	74269 ms	127.2
CV-swap	1	3.3 ms	240637.5

AV: Virtual Memory using assembly virtualization.
CV: Virtual Memory using C code virtualization.
CIBHit: Cache index buffer hit.
CIBMiss: Cache index buffer miss.
swap: Swap due to cache missing.

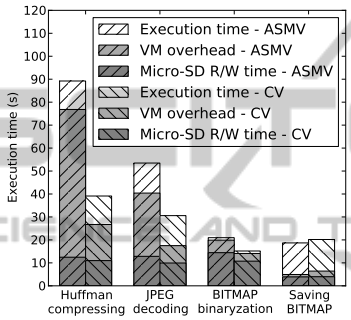


Figure 9: Algorithm for local image write back.

cessing. Memory access using C code virtualization only 54 cycles to access memory when the cache index buffer hit. Memory access using C code virtualization takes less cycles than memory access using assembly virtualization even when the cache index buffer misses because C code virtualization does not have the overhead assembly wrapping.

7.1.3 Program Performance

We evaluate FaTVM performances using a set of typical programs. Figure 9 shows the execution time of programs.

The Micro-SD access overhead is small compared to the VM overhead or the total execution times. The VM overhead consists of Micro-SD access overhead and overhead of address translation. For the first two programs, the overhead of address translation takes most of the execution time when using assembly virtualization and C code virtualization brings much less address translation overhead compared to assembly virtualization. Table 4 shows the CIB(cache index buffer) hit rates of the programs. The CIB hit rates are all above 90 percent and for simple data processing (i.e. BITMAP binaryzation and BITMAP saving), the CIB hit rates can even be more than 99 percent. So high CIB hit rates have eliminated the majority of the address translation overhead.

Table 4: Cache index buffer hit rates of programs.

Program	CIB hit rate
Huffman compressing	94.75%
JPEG decoding	90.64%
BITMAP binaryzation	99.85%
Saving BITMAP	99.71%

The topmost portion of execution time is caused by the algorithm calculations in program, which would be the total execution time if the physical memory is large enough to hold all program data making virtual memory unnecessary. BITMAP binaryzation algorithm is very simple and has minor calculation time. Saving BITMAP takes most time writing BITMAP data to a file on Micro-SD card, so the VM overhead is small compared to program execution time.

For complex data processing algorithms(i.e. huffman compressing and JPEG decoding), the VM overhead of FaTVM using C code virtualization is tolerable since these complex data processing should not be carried out frequently.

8 RELATED WORK

Virtual memory(Denning, 1970) has been an important research subject in traditional operating system research for a long time. It provides applications with isolated large flat address spaces which greatly simplified development of reliable programs, and prevent programs from ruining each other's address space. However, traditional virtual memory researches are generally based on MMUs, so they are not usable in MMU-less embedded systems.

Softvm(Jacob and Mudge, 2001) implements software-managed address translation without TLBs, however it is still designed for conventional computer systems.

MEMMU(Bai et al., 2009) proposed an automated compile-time and runtime 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 (e.g. TinyDB(Madden et al., 2005)) on sensor nodes. The virtual memory was generally assumed to be relatively small and node interactivity was the major consideration among application performance metrics.

Our work differs from previous work largely in that we were trying to provide a large efficient virtual memory for mass data processing. Most variables accessed in data processing are assumed to be in virtual memory. So, our work mainly focused on reducing the overhead of address translations and the miss rates of programs.

t-kernel(Gu and Stankovic, 2006) provides software-based virtual memory called DVM which provides to the user application a flat virtual memory space which can be much larger than the physical memory space of the host node. t-kernel uses assembly virtulization to access virtual memory. 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 is not applicable for sensor network programs with complex data processing because of the high overhead of secondary storage access. We didn't evaluate t-kernel's performance in this paper because t-kernel is not ported to STM32 MCU.

ViMem(Lachenmann et al., 2007) brings virtual memory support to TinyOS(2.x Working Group, 2005). The developer adds tags to nesC(Gay et al., 2003) source code to place variables to the 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 network programs, however it would be useless when the program is accessing a large variable sequentially or randomly, which is common in complex data processing. Further more, 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 believe that virtual memory transparency and flexible pointer operation is vital for porting or implementing complex data processing algorithms.

Recently, Enix(Chen et al., 2010) supplies software segmented virtual memory for code memory by code modification and also uses a Micro-SD card as secondary storage. However Enix does not provide virtual memory for data segments because of high run-time overhead.

9 CONCLUSIONS AND FUTURE WORK

We have designed and implemented a virtual memory for data-intensive applications in sensor networks, making it possible for sensor nodes to carry out complex computations with heavy memory footprints without using energy-hungry MCUs equipped with much larger RAM. We had been focusing on reducing virtual memory overheads in different aspects including cache management, address translation, Micro-SD accessing, etc to make achieve acceptable performance for sensor network applications. We showed that C code virtualization enables more flexible optimizations to take advantages of the high locality of memory access pattern of data processing programs in sensor networks.

Either the overhead of address translation, or the overhead of secondary storage accessing is considerable and worth more works to further improve the efficiency of virtual memory. We have been using Micro-SD as secondary storage since it is more convenient to use. However, we expect performance promotion if NAND flash with an FTL tailored for virtual memory is introduced to FaTVM.

Our evaluation demonstrated that common data processing algorithms generate memory footprints with high spatial and temporal locality. Currently, the address translation overhead is still remarkable within the total execution time. More sophisticated code analysis and transformation may further reduce virtual memory overhead.

Sensor nodes are normally flashed with programs for specialized applications, that is to say, the programs reside on sensor nodes and execute for long periods. This has made it feasible to optimize virtual memory specifically for certain applications or configure virtual memory parameters accordingly to better serve the applications. Different applications have different memory access patterns, which can be further analyzed to improve virtual memory efficiency.

ACKNOWLEDGEMENTS

This work was supported by the project "Advanced Sensor Network Platform Research" in Zhejiang Uni-

versity (No. 2010X88X002-11).

REFERENCES

2.x Working Group, T. T. (2005). Tinyos 2.0. In *Proceedings of the 3rd international conference on Embedded networked sensor systems*, SenSys '05, pages 320–320, New York, NY, USA. ACM.

Atkinson, D. C. and Griswold, W. G. (1998). 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, pages 46–55, New York, NY, USA. ACM.

Bai, L. S., Yang, L., and Dick, R. P. (2009). Memmu: Memory expansion for mmu-less embedded systems. *ACM Trans. Embed. Comput. Syst.*, 8:23:1–23:33.

Chen, Y.-T., Chien, T.-C., and Chou, P. H. (2010). Enix: a lightweight dynamic operating system for tightly constrained wireless sensor platforms. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, SenSys '10, pages 183–196, New York, NY, USA. ACM.

Chung, T.-S., Park, D.-J., Park, S., Lee, D.-H., Lee, S.-W., and Song, H.-J. (2009). A survey of flash translation layer. *Journal of Systems Architecture*, 55(5-6):332 – 343.

Crossbow Technology, I. (2007). Imote2 datasheet. Available at http://wsn.cse.wustl.edu/images/e/e3/Imote2_Datasheet.pdf.

Crossbow Technology, I. (2009a). Micaz specs. Available at http://www.xbow.com/Products/Product_pdf_fles/Wireless_pdf/MICAz_Datasheet.pdf.

Crossbow Technology, I. (2009b). Telosb specs. Available at http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/TelosB_Datasheet.pdf.

Denning, P. J. (1970). Virtual memory. *ACM Comput. Surv.*, 2:153–189.

Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., and Culler, D. (2003). The nesc language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 1–11, New York, NY, USA. ACM.

Gu, L. and Stankovic, J. A. (2006). ¡i¿t-kernel¡/i¿: providing reliable os support to wireless sensor networks. 1182809 1-14.

Hennessy, J., Patterson, D., and Goldberg, D. (2003). *Computer architecture: a quantitative approach*. Morgan Kaufmann.

Horwitz, S., Pfeiffer, P., and Reps, T. (1989). Dependence analysis for pointer variables. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, PLDI '89, pages 28–40, New York, NY, USA. ACM.

Jacob, B. and Mudge, T. (2001). Uniprocessor virtual memory without tlbs. *IEEE Trans. Comput.*, 50:482–499.

Lachenmann, A., Marrón, P. J., Gauger, M., Minder, D., Saukh, O., and Rothermel, K. (2007). Removing the memory limitations of sensor networks with flash-based virtual memory. *SIGOPS Oper. Syst. Rev.*, 41:131–144.

Ltd, A. (2010). Cortex-m3 technical reference manual. Available at http://infocenter.arm.com/help/topic/com.arm.doc.ddi0337i/DDI0337I_cortexm3_r2p1_trm.pdf.

Madden, S. R., Franklin, M. J., Hellerstein, J. M., and Hong, W. (2005). Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173.

Necula, G., McPeak, S., Rahul, S., and Weimer, W. (2002). Cil: Intermediate language and tools for analysis and transformation of c programs. In Horspool, R., editor, *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 209–265. Springer Berlin / Heidelberg.

Park, S.-y., Jung, D., Kang, J.-u., Kim, J.-s., and Lee, J. (2006). Cflru: a replacement algorithm for flash memory. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, CASES '06, pages 234–241, New York, NY, USA. ACM.

Soro, S. and Heinzelman, W. (2009). A survey of visual sensor networks. *Advances in Multimedia*, 2009(640386):1–21.

STMicroelectronics (2011). Stm32f103ze datasheet. Available at http://www.st.com/internet/com/TECHNICAL_RESOURCES/TECHNICAL_LITERATURE/DATASHEET/CD00191185.pdf.