# Removing the Memory Limitations of Sensor Networks with Flash-Based Virtual Memory

Andreas Lachenmann, Pedro José Marrón, Matthias Gauger,
Daniel Minder, Olga Saukh, Kurt Rothermel
IPVS, Universität Stuttgart, Germany
{lachenmann, marron, gauger, minder, saukh, rothermel}@ipvs.uni-stuttgart.de

## ABSTRACT

Virtual memory has been successfully used in different domains to extend the amount of memory available to applications. We have adapted this mechanism to sensor networks, where, traditionally, RAM is a severely constrained resource. In this paper we show that the overhead of virtual memory can be significantly reduced with compile-time optimizations to make it usable in practice, even with the resource limitations present in sensor networks.

Our approach, *ViMem*, creates an efficient memory layout based on variable access traces obtained from simulation tools. This layout is optimized to the memory access patterns of the application and to the specific properties of the sensor network hardware.

Our implementation is based on TinyOS. It includes a pre-compiler for nesC code that translates virtual memory accesses into calls of *ViMem*'s runtime component. *ViMem* uses flash memory as secondary storage. In order to evaluate our system we have modified nontrivial existing applications to make use of virtual memory. We show that its runtime overhead is small even for large data sizes.

## Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management—*Virtual Memory*; D.3.4 [**Programming Languages**]: Processors—*Preprocessors*

## General Terms

Design, Algorithms, Performance

## Keywords

Virtual memory, wireless sensor networks, flash memory, memory layout

## 1. INTRODUCTION

Traditionally, main memory has been a very scarce resource in sensor networks, and most sensor nodes are equipped with just a few kilobytes of RAM (for example, the Mica2 nodes from Crossbow only have 4 kB). However, several sensor network applications already require more memory than available on current sensor nodes. For instance, TinyDB [22], which provides query processing capabilities with an SQL-like language, requires the user to select at compile-time which functionality should be included in the code image. This decision can later only be changed by installing a new code image on each sensor node. Likewise, the maximum size of an application in the Maté virtual machine [21] is strictly limited by the amount of main memory available.

As applications for sensor networks increase in complexity, RAM limitations will continue to cause difficulties for developers. For example, if applications perform more complex analysis than just the simple aggregation of most applications today, even nodes with more memory will not be able to satisfy future needs. As the experience from other domains shows, even with comparatively large random access memories there is always a shortage of main memory.

In traditional computing systems virtual memory has been widely used to address this problem. With virtual memory, parts of the contents of RAM are written to secondary storage when they are not needed. This mechanism is easy to use since the system takes care of managing the memory pages. However, current operating systems for sensor networks [1, 7, 14, 17] do not support virtual memory. Only recently t-kernel [12] has emerged as the first attempt to demonstrate that virtual memory can be useful in sensor networks as well.

Sensor nodes are equipped with flash memory as secondary storage, which is much larger than main memory (between 512 kB and 1 MB). In most cases sensor network applications use flash memory to log sensor readings for later analysis. It is organized in pages of several hundred bytes that have to be written *en bloc*. Accessing it is much more expensive than accessing RAM: it takes several milliseconds to read or write a flash page whereas variables in RAM can be accessed in a few processor cycles. In addition, accesses to flash memory are comparatively expensive regarding energy consumption. Nevertheless, as we show in this paper, this type of memory is appropriate for the implementation of virtual memory on sensor nodes.

The system model used by operating systems such as TinyOS [17] is well suited to the use of virtual memory. First, variables and addresses are known at compile-time because all memory – except for the stack – is allocated statically. Secondly, since there is only one application running, the locality of reference [6] is increased compared to multitasking systems and accesses are more predictable. Finally, since sensor nodes usually execute code directly from program memory, which is separate from RAM, in sensor networks virtual memory will only be used for data. Therefore, the additional overhead introduced by virtual memory does not arise with every single instruction but only with data accesses.

In this paper we present *ViMem*, a system that provides a virtual memory abstraction for TinyOS-based sensor networks and uses flash memory to extend the size of RAM available to the application. Since energy is a limited resource in sensor networks, *ViMem* tries to minimize the number of flash memory operations. It uses variable access traces obtained from simulations to rearrange variables in virtual memory at compile-time so that only a small number of accesses is necessary. As we show in the evaluation, this algorithm helps to reduce the overhead of our virtual memory solution.

Using simulation data, the algorithm determines which variables are accessed frequently. It splits up complex data structures such as arrays or structs to examine each element individually. The algorithm then groups such parts of variables that are often accessed together. Likewise, it tries to put variables that are accessed frequently on the same memory page so that it is likely to remain in RAM most of the time.

The memory layout is determined offline by a pre-compiler that modifies the code to redirect variable accesses to virtual memory. *ViMem* manages the flash pages in memory so that the developers do not have to deal with these low-level details and access variables simply as if they were stored in RAM. The only difference for the application developers is that they have to tag all variables that they want to place in virtual memory with a special attribute. This way they maintain full control of which variables are stored in virtual memory. For sensor networks this is important to allow the developers to keep variables permanently in RAM if they are used in time-critical functions.

Our solution has several benefits. First, it makes it possible to develop complex sensor network applications without having to restrict the functionality due to memory constraints. Secondly, although the developers control which variables are placed in virtual memory, accesses to those variables are transparent. Finally, through an intelligent placement of variables that takes into account the properties specific to sensor network applications, to hardware platforms, and to the development process *ViMem* is able to provide its functionality with minimal runtime overhead. We argue that such optimizations are essential to make virtual memory usable for the resource-constrained devices of sensor networks.

The rest of this paper is organized as follows: Section 2 gives an overview of related work. Section 3 describes relevant properties of sensor networks as well as *ViMem*'s design. Section 4 gives details on our memory layout algorithm. In Section 5 we describe the implementation of both the development tools and the runtime system. Section 6 presents and discusses evaluation results. Finally, Section 7 gives an outlook on future work and concludes this paper.

## 2. RELATED WORK

In this section we present work related to *ViMem* that deals with virtual memory and data placement, uses flash memory for other purposes in sensor networks, or provides other solutions to the lack of memory on sensor nodes.

Virtual memory has a long history in operating systems research [6] and is now a standard technique in modern operating systems [26]. It allows the developer to use more RAM than physically available in the system by swapping out data to secondary storage. The system takes care of selecting the memory pages stored in RAM and translating addresses. Therefore, virtual memory is completely transparent to developers. Recently, t-kernel [12] has applied this widely-used mechanism to sensor networks. However, this system does not modify the memory layout to minimize accesses to secondary storage, which is an important property of our approach. In addition, there the overhead of a page fault can occur with every variable access (even in time-critical functions) since the developer cannot select the variables that are to be placed in virtual memory. Finally, because it is based on load-time modifications of the code, this further increases the overhead at runtime.

For traditional computing systems some compiler techniques to optimize the memory layout for the properties of the memory hierarchy have been proposed. Most of them either use hardware support, simulation traces, or simply the source code to determine the memory layout. In many cases the main focus is on restructuring code, which is not needed for our scenario where the code itself is separate from (virtual) data memory. For example, Hatfield and Gerald [16] reorder code sectors to have parts close to each other if they are used together. Similarly, Hartley [15] duplicates code blocks to have functions always near their caller. However, these optimizations do not modify the arrangement of variables in memory.

Stamos [27] classifies different approaches to create a memory layout including both code and data. In his classification our approach would be a graph-theoretic algorithm that uses information from actual execution traces. However, he regards the detailed analysis done by our memory layout algorithm as infeasible for his own scenario, the placement of Smalltalk objects on virtual memory pages.

In addition, there are some techniques that target other parts in the memory hierarchy. For example, compile-time optimizations can be used to create an efficient memory layout concerning CPU cache misses. Although similar in spirit, the problem for CPU caches is different from the one addressed by our approach. Instead of placing data used together on the same memory page, Calder *et al.* [4] try to reduce cache misses by placing such entries in memory locations that are mapped on non-conflicting cache lines.

Muchnick [25] gives an overview of techniques of compile-time optimizations for the memory hierarchy. In contrast to our approach, these techniques often involve modifications to the code such as loop transformations. Regarding the layout of data in memory, Gupta [13] proves that the problem of finding an optimal memory layout is NP-complete and describes a heuristic that uses information from the source code to arrange data in memory.

All of these optimization approaches are targeted to environments different from sensor networks. Therefore, they do not specifically address the properties of this domain (e.g., access characteristics of flash memory, energy considerations, etc.).

Sensor network applications and system components already use flash memory for different purposes. For instance, it is used to store sensor values directly on the sensor node [22, 31]. Storing the data in flash memory can be more energy-efficient than sending it to a base station [8]. In addition, code update mechanisms [18, 19, 23] use flash memory to store and process code updates before transferring them into program memory. Finally, ELF [5] is a file system for flash memory that allows application developers to store data without having to deal with the low-level properties of flash memory. As can be seen from this description, there is a variety of uses for flash memory in sensor networks.

Instead of using virtual memory, one could also build a sensor node which is already equipped with more RAM. For example, unlike most sensor nodes the BTnodes [3] have 240 kB of additional RAM. However, there are currently no nodes available that are equipped both with more RAM and flash memory. If the advantages of flash memory such as non-volatile storage of sensor readings and its even larger size are needed, alternative solutions such as virtual memory have to be used. Since one of the central assumptions of sensor network research is that nodes are cheap, consume little energy, and have a small form factor, equipping them with more RAM in addition to flash memory or replacing flash memory with other, more expensive types of memory is not an option. In fact, in the future the class of inexpensive, energy-efficient nodes will continue to be equipped with very limited hardware resources [12]. In addition, as described above, the experience from other domains shows that even large amounts of RAM do not lessen the need for virtual memory. Typically, more complex applications emerge if more memory is available.

## 3. SYSTEM PROPERTIES AND DESIGN

This section first presents relevant characteristics of sensor networks, then lists our design goals, and finally gives an overview of the system design.

### 3.1 Sensor Network Characteristics

A number of characteristics of sensor networks have influenced the design of our virtual memory system. First of all, the hardware platforms used in typical sensor networks do not include any support for virtual memory. Therefore, the whole system – including address translation, the detection of page faults, and the page replacement policy – has to be implemented in software.

In addition, the behavior of flash memory vastly differs from other types of memory like magnetic disks and RAM [9]. For example, there is a large difference in the access speed for reading and writing. This difference also becomes apparent with the energy consumption of flash memory. Therefore, the number of write accesses has to be minimized. Table 1 shows the properties of the flash memory chip used in the Mica family of sensor nodes. For this hardware, writing a page to flash typically takes 4.5 times as long as transferring one to RAM. Even more important, we have measured that writing requires about 23 times the energy of reading. Furthermore, there is a limit on how often each flash page can be written. Thus some kind of wear leveling [9] has to be used in order to make sure that in the long run each page is written a similar number of times.

Since sensor networks consist of a large number of devices that are embedded in possibly inaccessible locations and offer only limited user feedback capabilities, simulation has become an important part of the sensor network development process [29]. Therefore, simulation can be used as a tool for optimizations. Such optimizations are important in order to meet the application's lifetime goals.

Table 1: Properties of the Atmel AT45DB041B flash memory chip [2]

| **Property** | **Value** |
|---|---|
| Page size | 264 bytes |
| Number of pages | 2048 |
| Number of internal SRAM buffers for pages | 2 |
| Max. standby current | 10 $\mu$A |
| Max. page read current | 10 mA |
| Max. page write current | 35 mA |
| Typical page read delay (measured) | 3.6 ms |
| Typical page write delay (measured) | 16.3 ms |

### 3.2 Design Goals

Our main objective is *to provide a virtual memory abstraction on sensor nodes that minimizes energy consumption and the number of accesses to flash memory.* Taking into account the properties of sensor networks described above we have identified the following design goals for *ViMem* in order to achieve this general objective:

- *ViMem* should not require hardware support for address translation, etc.
- *ViMem* should minimize the number of write accesses to flash memory for energy and efficiency reasons.
- It should be efficient for frequently used variables.
- The developer should be able to control which variables are placed in virtual memory.
- Accesses to variables in virtual memory should be transparent to the developer.
- *ViMem* should allow for the reuse of existing application and system components without requiring major modifications.

### 3.3 Design Overview

*ViMem* consists of two main parts: a compiler extension and a runtime component. The compiler extension redirects variable accesses to *ViMem*'s runtime system and determines the placement of variables on the memory pages. The runtime component takes care of loading and storing flash memory pages when they are needed.

#### *3.3.1 Compiler Extension*

Developers should be able to use variables in virtual memory just like those in RAM. However, since sensor network hardware does not directly support virtual memory, all access to data in virtual memory must be redirected to *ViMem*'s runtime component. Our system accomplishes this by using a pre-compiler that modifies all such variable accesses. This pre-compiler changes source code written in nesC [11], the programming language used by TinyOS [17]. We have selected nesC and TinyOS because of their active research community that has developed a large number of application and system components. Many of them can potentially benefit from *ViMem*. Although this approach is no real virtual memory system, it offers similar benefits to applications.

The developer maintains full control of which variables are kept in RAM and which ones are stored in virtual memory. Only those tagged with a special attribute are put into virtual memory. This way, variables that are used in interrupt handlers and other performance-critical functions can always be kept in RAM. In TinyOS's execution model these functions are called “asynchronous events”. All other functions are executed in the task context which is less performance-critical, as interrupt event handlers can suspend such tasks [11, 17].

The pre-compiler executes a memory layout algorithm to place variables on pages in virtual memory. Our software design allows for easy use of different memory layout algorithms. The default algorithm, which is described in Section 4, uses access traces obtained from simulation tools to get information about the frequency of memory accesses. Doing all the processing offline minimizes the effort at runtime on the sensor nodes.

### 3.3.2 Runtime Component

*ViMem*'s runtime system is responsible for the management of memory pages kept in RAM and for the provisioning of data to the application. The challenge here is to determine which memory page has to be replaced when another one is loaded from flash memory. Therefore, the algorithm has to predict which pages are most likely used again in the future. In addition, it has to consider the costs for replacing a page. If a page has not been modified, replacing this page is less expensive than selecting a page that has to be written back to flash memory.

This algorithm was not the main focus of our research. Therefore, for *ViMem*'s replacement policy we employ the Enhanced Second-Chance Algorithm [26], which approximates a least-recently used (LRU) page replacement strategy. This algorithm stores two bits for each page in RAM: a recently used bit and a modified bit. It uses these bits to group each page into one of the four classes described below. Then it selects the first page in the lowest nonempty class to be replaced.

The lowest class consists of pages that have neither been modified nor recently used. These pages are the best ones to replace. If a page has been modified but not recently used, it is part of the second class. Likewise, the third class comprises all pages that are clean but have been recently used. Finally, a page that has been recently used and modified belongs to the last class. It will not only have to be written back to flash memory but is also likely to be used again soon.

The page replacement algorithm uses a circular list to examine each page in memory. It starts looking for a page in the lowest category. If it finds none, it looks for one of the second class and then continues with the other ones.

In addition, the runtime system makes use of the SRAM buffers, which are part of the flash memory chip used in our implementation platform, the Mica2 nodes. Pages have to be transferred to one of these buffers before they can be written to the flash. In addition, as Fig. 1 shows, the buffers are used as a second level of caching: expensive write accesses are not performed immediately but the page is just stored in the buffer. If it is needed again, it does not have to be written to flash memory and can be retrieved from the buffer. Likewise, if it is not modified again while in RAM, it does not even have to be written back to the buffer again. If, however, a page is read that is currently not stored in one of the buffers, our system reads it directly from flash memory to RAM. This is possible without introducing additional overhead and leaves the SRAM buffers unchanged so that the number of actual write accesses to the flash can be further reduced. Since persistence is not required for a virtual memory system, it does not matter if a page in the buffer is lost when the batteries of the device are depleted, for example.

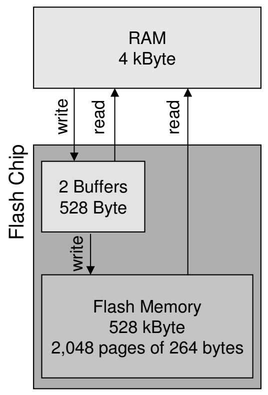


Figure 1: Memory hierarchy of *ViMem*

Each flash memory page can only be written a fixed number of times. Therefore, our runtime system tries to distribute write accesses across several physical pages in order to avoid wearing out pages that are written more often than others. For this purpose, it reserves a larger pool of flash memory pages than it actually needs (e.g., 1.5 times this number). Since there is usually enough free space available in flash memory and since we expect the space needed for virtual memory to be comparatively small, this does not severely reduce the flash memory usable by the application and other system components. If a page is written back to flash, *ViMem* cycles through a list of all reserved flash pages and selects the first one currently not being used. Following this approach, write accesses to frequently used pages will be spread across a larger number of physical pages. The data structures containing the information about available pages can be kept in RAM, as they are relatively small. In addition, since old contents of virtual memory do not have to be accessed after restarting a node, losing these data structures does not lead to problems. Of course, this aspect of our system is only needed if the flash memory chip does not take care of wear leveling itself.

## 4. MEMORY LAYOUT ALGORITHM

This section describes our memory layout heuristic that determines the placement of variables in virtual memory. It is the core part of our approach to reduce the number of flash accesses and, thus, improve on efficiency. The heuristic has two main goals: First, it aims to exploit locality of reference in order to reduce the overall number of page replacements. Second, it puts special effort in decreasing the number of write accesses to flash memory.

*ViMem*'s pre-compiler runs this algorithm when building the application. As we describe in the following subsections, it uses simulation traces to determine an efficient memory layout by grouping and reordering variables.

## 4.1 Use of Variable Access Traces

In general, finding an optimal memory layout is not possible since the exact order in which variables are accessed at runtime depends on many factors. For example, in sensor networks data gathering requests from users as well as sensory input and packets received from other nodes may influence the application flow. Even if the exact order of accesses were known at compile-time, finding an optimal memory layout would be an NP-complete problem [13]. Therefore, our memory layout algorithm can only provide a heuristic that does not necessarily find the best solution for each execution path.

Although the specific order of data accesses is not predictable, usually there are patterns that recur. For example, some variables are often accessed following each other and some of them are accessed more frequently than others. Our heuristic uses simulation traces to determine such patterns for variables stored in virtual memory. Even if the same sequence of accesses is not repeated when running the application later, we argue that these traces can provide valuable hints for data placement. However, the simulation scenario has to resemble the actual operation of the sensor network. Since simulation is a technique often used when developing sensor network applications and since simulation scenarios have to be realistic to evaluate the functionality and performance of the application anyway, this step does not impose excessive additional burden on the application developer. Furthermore, as these results can be obtained by extending the simulator rather than modifying the application, gathering information about variable accesses does not alter the behavior of the application itself. Therefore, a single simulation run can be used both to test the application and to obtain a data access trace.

The *ViMem* pre-compiler can be configured to use for variable placement either the access traces of all nodes of the network, those of a group of nodes, or just the ones of a single node. This allows the system to optimize the memory layout for nodes with different tasks, although they may execute the same code image. For example, a node at the edge of the network can have different variable access patterns than a node at the center where more packets have to be forwarded. We expect that the performance of *ViMem* is best if a separate code image with an optimized memory layout is installed for each such group of nodes. Otherwise, the system remains fully functional, but at increased memory access costs.

If no simulation data is available (e.g., when building a new application), the *ViMem* pre-compiler uses the variable references in the source code to estimate the number of accesses. Obviously, this information can be inaccurate because it is unclear how often a function is called or which branch of an if-statement is selected at runtime, for example.

We have verified each design decision taken for our heuristic using existing sensor network applications and always selected the alternative that offered the best performance. As Section 6 shows, the results of the heuristic are promising.

The pre-compiler splits up complex variables, such as large arrays and structs, and examines each part individually. For example, the first elements of an array might be accessed more frequently than the last ones. Therefore, instead of recording the access just for complex variables as a whole, all data accesses are associated with individual data elements. We define as such a data element an atomic part of a complex variable with a simple data type like "int". This approach allows the memory layout to more closely resemble the actual access patterns. The only exception to this rule are small arrays: our algorithm always treats accesses to one of their elements as accesses to the complete array. In our experience with existing applications the elements of such arrays have a tight coupling and, therefore, should be regarded as a single entity.

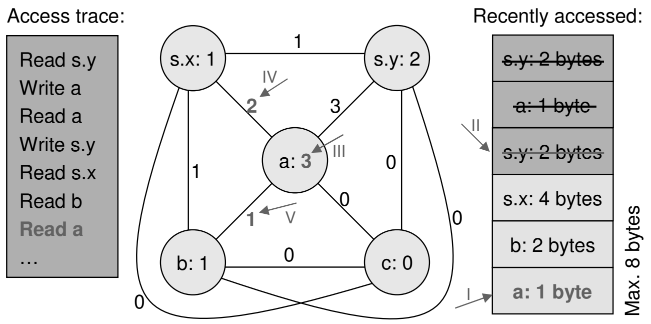


Figure 2: Example for processing an access trace

## 4.2 Grouping of Data Elements

Having gathered information about accesses to data elements, the memory layout algorithm groups those data items often accessed together. When reading an access trace the pre-compiler calculates the weights of a fully-connected graph $G = (V, E, f, g)$, where the nodes $V$ are the data elements and the edges $E$ represent the relationship between the data elements. In this graph both the nodes and edges are weighted: The weight of a node, given by $f : V \rightarrow \mathbb{R}$, indicates how often the corresponding data element has been accessed, and the weight of an edge, defined by $g : E \rightarrow \mathbb{R}$, gives information about the proximity of the data elements it connects.

For each sensor node in the network, the pre-compiler maintains an ordered list of data elements that have been accessed recently. Each data element appears in this list at most once with only its most recent access. The sum of the sizes of all elements may not exceed a parameterizable constant. These elements represent those that should be preferably in RAM when the new element is accessed. If one of them is not stored in RAM, a page fault can occur and another page would have to be loaded to RAM. When the *ViMem* pre-compiler adds a data access from the trace, it increases both the access count in the data element's node and the proximity to all data elements in the list.

Proximity of two data elements deliberately does not take into account the temporal distance between accesses to data elements: To determine if they should be placed on the same memory page, it does not matter whether or not there is some delay between accesses – as long as no other variables are accessed in between.

Fig. 2 shows an example how an access trace is processed. The figure displays parts of an access trace for one node, the graph $G$, and the list of recently accessed elements. For simplicity, it assumes that the size of a memory page is just 8 bytes and that the same maximum size is used for the elements in the list. The figure shows the simple variables "a" (size: 1 byte), "b" (2 bytes), and "c" (4 bytes) as well as struct "s" with its fields "x" (4 bytes) and "y" (2 bytes). As described above, the algorithm splits up the parts of the struct and examines each field individually. In the example the last line of the access trace has been processed, which leads to the changes highlighted with arrows. First, the element is added to the list of recently accessed data elements (I). Since the total size of the elements in this list is greater than the page size, the algorithm removes the oldest element ("s.y", crossed-out in the figure, II). Then it increments the weights of "a" (III) and of all its edges to elements in the list (IV and V).

After reading the complete access trace, *ViMem*'s precompiler tries to group the elements that are often used together. To achieve this goal it merges nodes in the graph $G$ that have high proximity values. This step is inspired by the procedure sorting algorithm [25] that performs similar operations on the call graph to place the code of a procedure always near their caller.

In this step the algorithm does not use the raw proximity value $g(e)$ but normalizes them using the access counts of the data elements ($f(v_1)$ and $f(v_2)$) it connects: $p = g(e)/(f(v_1) + f(v_2))$. This way, the algorithm can form groups both of elements accessed frequently and of those only used seldom. It repeatedly selects the edge $e_{max}$ from the graph $G$ with the highest normalized proximity value $p_{max}$. The algorithm then merges the nodes connected by $e_{max}$ and coalesces their edges. It sets the node weight of the group to the average of its elements' weights, which helps to treat merged nodes and original ones as equal when computing $p$. The weight of coalesced edges is set to the maximum of the original edge weights. This preserves close proximity between elements even if they have become parts of merged nodes. The elements forming one such new node are grouped and always placed on a single memory page later.

The algorithm repeats this process until a given percentage of data elements is grouped. If, however, the size of the group exceeds a given limit (e.g., one eighth of the size of a memory page), no more elements are added to it. The reason for this is that we want to avoid very large groups which are less flexible when creating the memory layout.

### 4.3 Data Placement

After determining the groups of data elements used together, the memory layout algorithm places them on actual memory pages. This part of the heuristic processes the elements in the order of their access frequencies, as proximity has already been exploited when forming groups. This way data elements that are accessed often are placed on the same memory page, which can probably stay in RAM for most of the time, while elements that are used only seldom do not occupy space on pages in RAM.

The algorithm places the data using a first-fit strategy with two sets of pages: one with elements that are modified often and one with those that are written only seldom. When placing an element, it checks if the number of write accesses is above a threshold. In this case the element is placed on a page that contains other elements that are written frequently whereas all elements that are mostly read are placed on different pages. If a page has to be removed from RAM, this approach makes it more likely that the page does not have to be written back to flash memory. Similarly, if a modified page is removed from RAM, it is probably written because of multiple changes. Therefore, this scheme takes

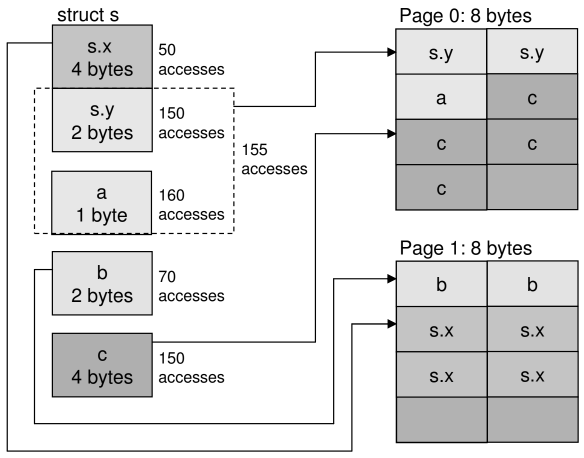


Figure 3: Example for the memory layout algorithm

into account the differences between read and write accesses mentioned above. In some cases this part of the heuristic helped to reduce the number of write accesses by up to 70 %.

Fig. 3 takes up the example from Fig. 2 after the complete access trace has been processed. For simplicity, the example does not distinguish between read and write accesses. In the situation shown just "s.y" and "a" are grouped. Then all groups and single data elements are placed on flash pages in the order of their access counts. The variable "b" is put completely on the second page, as elements may not span several pages. This way at most one flash memory operation is necessary when accessing a variable. As the example shows, the elements on page 0 are accessed more frequently than those on page 1. Therefore, this page will probably be kept in RAM most of the time whereas page 1 can be replaced more often with other pages.

## 5. IMPLEMENTATION

In this section we describe in more detail how *ViMem* has been implemented and how it can be used. We give this description for its two parts, the compiler extension and the runtime component, in the following subsections.

Our current implementation of *ViMem* is based on TinyOS and nesC. It has been optimized to the hardware properties of the flash memory chip used in the Mica family of sensor nodes. However, many of the concepts could also be applied in other operating environments and on other hardware platforms.

### 5.1 Compiler Extension

#### *5.1.1 Overview of the Compilation Process*

Fig. 4 gives an overview of the compilation process of an application that uses *ViMem*. First, the nesC compiler checks the syntax of the source code and generates an XML file with information about components, variables, functions, etc. This file is then used by the *ViMem* precompiler to determine which components contain accesses to virtual memory and, therefore, need to be rewritten. The XML file also includes information about the data types of the variables that are to be placed in virtual memory. Exporting information about the program structure to an XML file has been introduced in version 1.2 of the nesC compiler [10]. Using information from the nesC compiler avoids

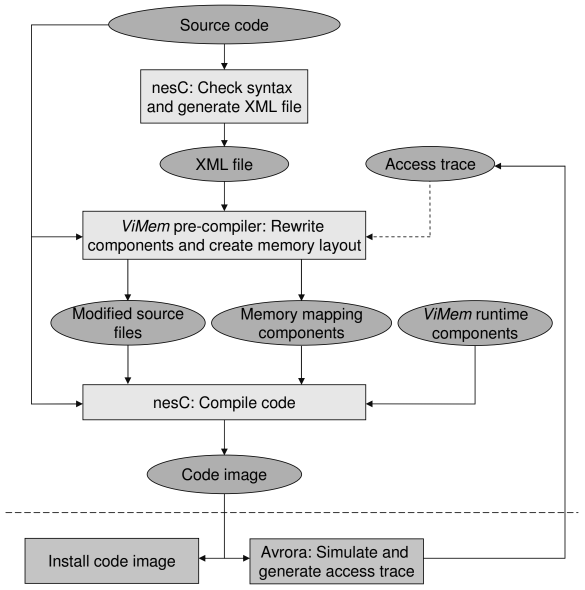

Source code
nesC: Check syntax and generate XML file
XML file
Access trace
ViMem pre-compiler: Rewrite components and create memory layout
Modified source files
Memory mapping components
ViMem runtime components
nesC: Compile code
Code image
Install code image
Avrora: Simulate and generate access trace

Figure 4: *ViMem* compilation process

duplicating existing functionality in our pre-compiler. Besides the source code, another input for the pre-compiler is a data access trace obtained from simulation. If this trace is available, it improves the performance of *ViMem* at runtime. However, it is not essential for building a running application.

Using all these inputs, the *ViMem* pre-compiler modifies the source files that access virtual memory and creates an optimized memory layout as well as components that make this layout available at runtime. These results, the *ViMem* runtime components, and other unmodified source files are the input for the actual compilation step, which is performed by the nesC compiler. As usual, this step results in a code image that can be installed on sensor nodes or simulated using appropriate tools. These two alternatives are shown in Fig. 4 below the dashed line because they are not part of the compilation process itself but are invoked separately.

If the developer chooses to simulate the application, the output of the simulator, in our case the Mica2 simulator Avrora [29], can be used to generate an access trace. Avrora provides fine-grained instrumentation capabilities which allow to get detailed information about the execution without changing the timing of the program [30]. Using this instrumentation interface, we have created probes that record all accesses to data elements in virtual memory. The result given by these probes forms the access trace for the *ViMem* pre-compiler. If the application is compiled again, a memory layout optimized for the given access trace will be generated.

### 5.1.2 Implementation Details

The pre-compiler has been implemented in Java using JavaCC as a parser generator. It modifies all components that access variables in virtual memory, creates the memory layout, and generates components that map variables to their actual position in virtual memory.

As the code example in Fig. 5 shows, all variables that are to be placed in virtual memory have to be tagged with the attribute “@vm”. The ability to tag variables, parameters, and functions with user-defined attributes is another feature

```
uint16_t varInVM @vm();
uint16_t* pointer @vmptr();
uint16_t varInRAM;

uint16_t* testFunction(
    uint16_t* value @vmptr()) @vmptr() {
  *value = 54;
  return value;
}

command result_t StdControl.init() {
  varInRAM = 123;
  varInVM = varInRAM;
  pointer = &varInVM;
  varInVM = *testFunction(pointer);
  return SUCCESS;
}
```

Figure 5: nesC code that accesses variables stored in virtual memory

introduced in nesC 1.2. Only variables which are declared globally within a component or marked as “static” can be placed in virtual memory. Local variables of functions, in contrast, are always allocated in RAM on the stack.

It is not possible to reference a variable in virtual memory using a normal pointer variable, since its actual position in RAM may change after it has been swapped out to flash memory. In this case the runtime system could not associate the pointer value with a variable in virtual memory. Therefore, pointer variables as well as parameters and return values of functions that refer to a variable in virtual memory have to be tagged with the attribute “@vmptr”. All variables and parameters that are tagged with this attribute are modified by the pre-compiler as well: They no longer refer to a location in RAM but contain an ID that denotes the corresponding element in virtual memory.

The only restriction for the developer is that casting variables to types of a different size is not allowed. The reason for this is that data elements in virtual memory are not necessarily contiguous. Without this restriction it would be possible that several flash operations are necessary for such an access.

Adding the attributes “@vm” and “@vmptr” is the only change to the code that has to be performed by the application developer when using *ViMem*. As the example in Fig. 5 shows, accesses to such variables look exactly like accesses to normal variables. In the example there is one variable stored in RAM (“varInRAM”), one stored in virtual memory (“varInVM”), and a pointer to a variable in virtual memory. The sample code shows that *ViMem* uses pure nesC code. If the pre-compiler is not run, nesC could still compile the same code and store variables in RAM.

The *ViMem* pre-compiler takes this code and removes the declaration of all variables placed in virtual memory. Instead it assigns them an ID that refers to a position in virtual memory. Therefore, it replaces all references to such variables as well as accesses via “@vmptr” variables with calls to the *ViMem* runtime component that refer to the element’s ID. Like most functions in nesC, calls to the components of the *ViMem* runtime system are usually inlined. This reduces the overhead associated with accesses to variables in virtual memory. Furthermore, as described in Section 4, *ViMem*’s pre-compiler places variables intelligently in virtual memory in order to reduce accesses to flash memory.

For efficiency reasons, the pre-compiler uses two different ways to translate an ID of a data element in virtual memory to the right variable. If the ID is known at compile-time, the pre-compiler inserts a direct call to the runtime system with the right page and offset in that page. This is possible for all variables except pointers and accesses to arrays when the element index is stored in another variable. For these two types of accesses the exact ID of the element is not known at compile-time. Therefore, the runtime system has to look up the page and offset of the data element whenever such a variable is accessed. This information is stored in an array in program memory where it can be read efficiently without occupying space in RAM.

If possible, the pre-compiler uses the first type of access because the performance of the second variant is slightly worse (see Section 6.1).

## 5.2 Runtime Component

The runtime component is responsible for checking if a page needed currently resides in RAM and to move pages from and to flash memory. It has been optimized regarding its fast path, i.e., the overhead for the most common case (accesses to pages already in RAM) has been reduced. For accesses to pages that have to be loaded first, optimization is not as critical since a much longer delay is imposed by the flash memory operations. Furthermore, we have tried to keep the RAM consumption of the runtime system low. However, where possible, we opted to reduce overhead on the fast path by keeping data structures for efficient accesses in RAM (see Section 6.3 for details on RAM overhead). As with *ViMem* the strict RAM limitations are no longer a severe problem, marginally increasing the RAM consumption of the system does not limit the memory available to the application.

The performance of *ViMem* depends on the number of virtual memory pages that are kept concurrently in RAM. This parameter is determined by the pre-compiler dynamically based on the memory space available. In Section 6.3 we show how this number influences the overall performance.

For accesses to flash memory, *ViMem* uses the standard TinyOS "PageEEPROM" component which has been slightly modified and extended for our purposes. Nevertheless, our version of this component is fully compatible with the old version, which also allows other parts of an application to access the flash. The only drawback of such uses by another component will be a somewhat degraded performance, since the flash chip is blocked if that component reads or writes pages. In addition, *ViMem* has to share the flash buffers with that component, which may lead to an increased number of actual writes to the flash chip.

Our modifications to the flash component are limited to three changes: First, the flash component resets its state immediately after executing a command instead of doing this in a separate nesC task. This modification allows subsequent accesses to flash memory from a single task. Secondly, we have optimized read accesses by transferring pages directly to the CPU without loading them into one of the flash chip's SRAM buffers. Since modified pages would have to be written to actual flash memory if another page was loaded into its buffer, this change reduces the number of write accesses to flash memory. Finally, we added a new command that avoids unnecessary page transfers from RAM if the same (unchanged) page is still stored in one of the buffers. This modification reduces the number of page transfers to the flash memory chip if modified pages have been copied from a buffer to RAM again only for reading.

Table 2: Typical latencies for different kinds of variable accesses

| Type of access | Delay |
|---|---|
| Variable in RAM | 1.09 $\mu$s |
| VM variable in RAM | 18.72 $\mu$s |
| VM var. from buffer without page write | 3.66 ms |
| VM var. from flash without page write | 3.72 ms |
| VM var. from flash with page write to buffer | 7.58 ms |
| VM var. from flash with page write to flash | 19.83 ms |

## 6. EVALUATION

In this section we present evaluation results for *ViMem*. All experiments have been performed using Avrora [29], which accurately simulates Mica2-based sensor networks. It contains an energy model with detailed information about the energy consumption of the hardware components [20].

## 6.1 Isolated Memory Access Performance

Table 2 compares the access speeds of a single variable access depending on where the variable is currently stored.

Variables that are not included in *ViMem*'s virtual memory system are always stored in RAM. The access to such a variable takes only 1.09 $\mu$s.

If a variable is stored in virtual memory, there is some overhead for each access even if the page containing the variable is already available in RAM. The reason for this is that *ViMem*'s runtime system has to check first if the page is currently stored in RAM. Since our hardware platform does not directly support virtual memory, this has to be implemented in software. Therefore, it takes up to 18.72 $\mu$s to access such a variable. This number also includes the address translation of the variable ID to the correct memory page and the offset within this page. However, in many cases this translation can be done offline by the compiler. Then the latency for variable accesses is reduced by 22 % to 14.65 $\mu$s (not shown in the table).

If a page is not stored in RAM but has to be retrieved from either the flash memory chip's buffers or the flash memory itself, the delay increases further. If the replaced page in RAM has not been modified and thus does not have to be written back to flash, it takes about 3.7 ms to read a new page from the flash buffers or the flash memory itself. These numbers also contain the processing overhead of *ViMem*'s runtime system that has to find a page to replace in RAM. However, the dominating factor is the transfer time of the page from the flash memory chip to the CPU via the Serial Peripheral Interface (SPI). This transfer time also prevails if a page from RAM has to be copied to one of the flash buffers. Therefore, in this case the latency approximately doubles.

A somewhat larger delay can be observed if another page has to be written from the flash buffer to flash memory in order to get a free buffer for the new page coming from RAM. The whole cycle of writing that page to flash memory, writing another one from RAM to the flash buffer, and reading the new page takes more than 19 ms.

These numbers seem huge compared to a variable access in RAM. However, it should be noted that *ViMem*'s main

Table 3: Variables moved to virtual memory

| Application | Number | Size | Pages used |
|---|---|---|---|
| TinyDB | 75 | 569 bytes | 2.15 |
| Maté | 1 | 792 bytes | 3.0 |

Table 4: Allocated space in RAM

| Application | Original | *ViMem* |
|---|---|---|
| TinyDB | 2,832 bytes | 2,577 bytes |
| Maté | 3,196 bytes | 2,727 bytes |

goal is to reduce the number of flash accesses, especially page writes. Therefore, in practice only a small number of variable accesses leads to such long delays, and most variables are accessed in RAM (see Section 6.2 and 6.3 for details). In addition, since access to virtual memory is only allowed in non-time-critical functions, even long delays do not affect the operation of the application. Finally, in other domains, where virtual memory has been successfully used for years, similar delays can be observed. For example, the random access speed of typical hard disks for PCs is about 10 ms.

## 6.2 Application Performance

In this subsection we focus on the performance of complex applications using *ViMem*. We have modified two of the most RAM-intensive applications available in the TinyOS CVS repository [28]: TinyDB and Maté.

### 6.2.1 Experiment Setup

TinyDB [22] provides an SQL-like query interface to the sensor network. With almost 30,000 lines of code it is one of the most complex applications available for TinyOS and as a part of the TASK system has been successfully used in real-world deployments [31]. As described in Section 1, TinyDB requires the user to select at compile-time which functionality to include in the code image since the variables of all its components would not fit in RAM.

Maté [21] is a virtual machine that executes applications, which have been compiled to a special byte code format. Although this byte code representation is more compact than native code, RAM still is a limiting factor, since the user's application as well as its variables have to be stored there.

We have modified both applications to make use of *ViMem*. In TinyDB we have only put variables of application and routing components in virtual memory, as they are responsible for large amounts of memory consumption. Pointers to variables in virtual memory are still stored in RAM to avoid two flash accesses for a single data item. In addition, we have not added variables to virtual memory that are used in time-critical functions like "async events" or are cast to another type. TinyDB heavily uses the last kind of variables to implement a dynamic memory allocation mechanism. Although we have left this mechanism untouched, with significant changes to the application it would be possible to replace large parts of it with virtual memory.

Concerning Maté we have focused on the variables storing the application code capsules in RAM and left all other variables unchanged. First, these variables consume large amounts of RAM. Secondly, this part of Maté is essential to build more complex applications. Finally, Maté already provides mechanisms to adapt the size of these variables to the underlying hardware platform. If more memory for the byte code becomes available by using *ViMem*, it is easy to adjust Maté to take advantage of this.

Table 3 summarizes the number and size of variables that are stored in virtual memory and shows the number of memory pages used. All of these modifications were done by simply adding the "@vm" and "@vmptr" attributes to variables or pointers, respectively. With Maté all the code capsules are stored in a single array. Therefore, just one variable has been marked with the "@vm" attribute. We have compiled both applications with their default settings.

Table 4 compares the size of allocated memory in RAM for the original and the *ViMem* versions of the applications. In addition to the size of the page currently kept in RAM the *ViMem* versions also include the RAM overhead of their runtime components as well as the flash memory access components from TinyOS (approximately 50 bytes).

We have created several versions of TinyDB and Maté that use virtual memory and differ only in their respective memory layouts. The first version uses the same runtime components as *ViMem* but places data elements in the order in which they are processed by the compiler using a first-fit strategy. This approach makes use of the observation that variables declared together are often used together and, therefore, exploits the natural locality of the code. The other virtual memory versions use *ViMem*'s heuristic to create an optimized memory layout. One of them just relies on information about variable accesses from the source code whereas the other ones base their layout on simulation traces. For both applications one such trace has been created when no query or no byte code application, respectively, was running. For TinyDB the second trace has been created when one query to report the node ID was being executed. Similarly, for Maté we have run the CntToLeds application from the tutorial [24] for recording the second simulation trace.

All data access traces have been obtained from a deliberately simple setup with just five nodes in a grid topology. In contrast, the actual simulations of the applications have been done in a network of 50 randomly placed nodes. These changes in the simulation setup help to show that some differences between the scenario in which the access traces have been obtained and the actual operation environment do not influence the effectiveness of our algorithm.

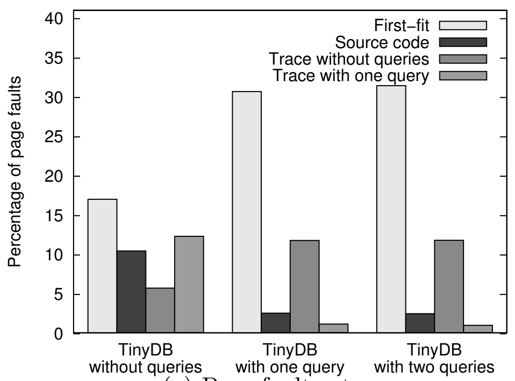
For these experiments we keep just one memory page (264 bytes) in RAM. This is the worst case for a virtual memory system: if a variable from another page is accessed, this page has to be loaded from flash memory first. Having just one page in RAM eliminates side-effects of the page replacement algorithm. Therefore, the performance is just influenced by the memory layout.

We have simulated TinyDB and Maté using different scenarios. Each of them has been simulated for 1,000 seconds.

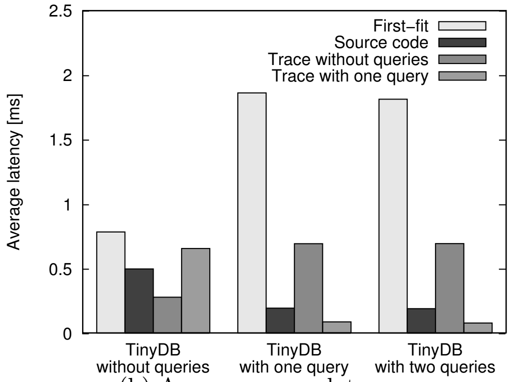
### 6.2.2 Simulation of TinyDB

Fig. 6 shows simulation results for the four variants of TinyDB described above in three different scenarios. The first two scenarios correspond to the ones used to get the data access traces. In the third scenario an additional query is dispatched which reports the values of the light sensors.

In Fig. 6(a) we show the ratio of variable accesses leading to a page fault, i.e., a read access from secondary storage (including reads from the flash chip's buffers). In many cases *ViMem*'s memory layout algorithm greatly reduces the number of accesses to flash memory. Using only the data ref-



(a) Page fault rate



(b) Average access latency

Figure 6: Simulation of TinyDB

erences from the source code it is already able to decrease the percentage of accesses leading to page faults by at least 35 %. Nevertheless, memory layouts that have been optimized for a given scenario can reduce the percentage of page faults even further. For the scenarios executing queries in the best cases the page fault rate is just around 1 % (about 3,000 page reads). This excellent result is possible because only a subset of the variables in virtual memory is frequently used. In these experiments 90 % of all accesses refer to just 20 % of the bytes in virtual memory.

For the scenario when no query is running the page fault rate is slightly greater (approximately 5 % or 1,400 page reads) even if the memory layout has been optimized specifically for this scenario. The reason for this is that here a much smaller number of accesses is distributed over almost the same number of distinct data elements.

As the versions optimized with traces of the respective other situation show, the scenario used to obtain the access traces should not differ too much from the actual execution scenario. In fact, in such cases the version optimized using the source code can be even better than an application optimized for the wrong scenario – especially, if the simulation scenario was too simple for the execution environment. However, if more information about the execution scenario is available, using traces from simulation can reduce the number of page faults by about 60 % compared to just using the source code for optimizations.

With *ViMem* less than 7 % – and in the best case just 0.6 % – of all variable accesses lead to a page transfer to the flash buffer. In addition, at most 0.1 % of all variable accesses result in an actual write access to flash memory. The reason for this is both the efficient memory layout and the use of the flash chip's buffers as a second level of caching. In absolute numbers, in our simulations with *ViMem* each node usually performs just 2 such operations. Our wear leveling heuristic described in Section 3.3.2 ensures that write accesses are evenly spread across all allocated flash memory pages. The first-fit strategy, however, has to transfer a page to the flash buffer in up to 14 % of all variable accesses. Each node writes up to 1,696 pages to flash memory, although it uses the same buffering techniques as *ViMem*. We attribute this difference primarily to the distinction of pages that are mostly read and those that are written often.

As the access latency directly depends on the number of page faults and write accesses, similar results can be reported for these measurements. As Fig. 6(b) shows, *ViMem* reduces the latency for TinyDB up to 95 % compared to the first-fit layout. However, even for the version optimized using the query trace, the average latency is still at least 75 $\mu$s whereas an access of a normal variable in RAM is performed in approximately 1 $\mu$s.

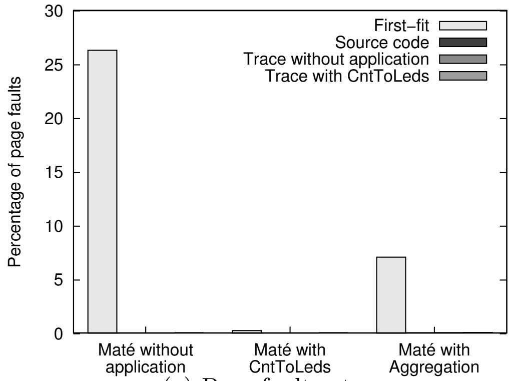
If more than one page is kept in RAM, most of these numbers will be further reduced, since a smaller number of flash accesses will be necessary. For instance, if two memory pages are kept in RAM, for the *ViMem* versions in many cases no page faults at all occur and the average access latency can be as small as 16 $\mu$s. Furthermore, in sensor networks processing power is usually not as constrained as other resources; in typical sensor network applications the CPU is idle most of the time. Therefore, we think that the overhead introduced by *ViMem* is acceptable for many applications.

We do not present detailed results for energy consumption because they are dominated by other hardware components and interactions between nodes. In addition, the timings of the TinyDB versions vary so that, for example, the applications send a different number of packets or turn on their LEDs for differing durations. Therefore, the values oscillate. In Section 6.3 we show results for energy consumption in a more controlled setting.

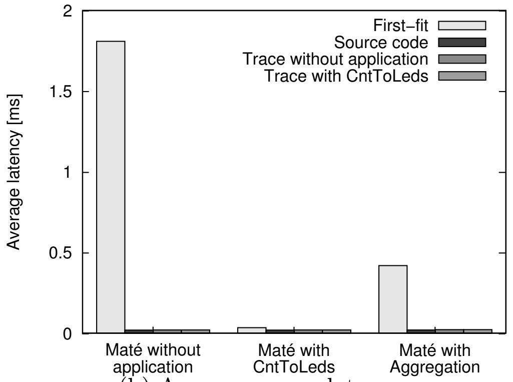
### 6.2.3 Simulation of Maté

Our simulation scenarios for Maté follow the same pattern from TinyDB. First, we have simulated Maté without any (byte code) application running and then with the CntToLeds application, which correspond to the two scenarios used to get the variable access traces for *ViMem*. Then we have run the aggregation application from the Maté tutorial. This is a more complex application that builds a collection tree and calculates the average value of sensor readings in this tree. Like the third scenario for TinyDB this one shows the performance when the operation differs from the traces.

As Fig. 7 shows, for Maté many of the numbers are even smaller than for TinyDB. The reason for this is that in our scenarios only some of Maté's storage spaces for code capsules are used. Therefore, accesses typically refer to just a small set of variables that can be kept in RAM most of



(a) Page fault rate



(b) Average access latency

Figure 7: Simulation of Maté

the time. This behavior, however, is typical for most applications that allocate their memory statically because they have to reserve enough space to deal with the worst case.

When no application is executed, there is only a small number of variable accesses (38 accesses per node, mostly for initialization). Therefore, the high page fault rate for the unoptimized version shown in Fig. 7(a) is somehow misleading as it is just 10 page faults in absolute numbers. The *ViMem* versions, however, do not show any page faults at all for this scenario as well as for the execution of CntToLeds. For this application even the unoptimized version achieves a page fault rate of just 0.23 %. This is because CntToLeds is very small and the code capsules used are (by coincidence) placed on a single memory page.

For the aggregation application the numbers remain excellent for all *ViMem* versions with approximately 6 page faults (less than 0.1 %). This application is still small enough to fit on a single memory page if the memory layout is chosen appropriately. In fact, less than 6 % of the data elements allocated in virtual memory are used in 90 % of all accesses. This is because the total size of statically allocated memory has to be able to accommodate an application consisting of several large code segments. If some code capsules are only filled partially, the memory space in between is not used.

The short access latencies shown in Fig. 7(b) also reflect that a very small number of write accesses is necessary. If there are page faults at all, each node of the *ViMem* versions writes on average less than 4 pages to the flash buffers of which up to 3 are copied to the actual flash memory chip. The unoptimized version, however, has to transfer up to 699 pages to the flash buffers, which increases the latency.

## 6.3 Large Data Storage

In this subsection we evaluate *ViMem* with respect to two parameters: the number of memory pages in RAM and the total size of all data stored in virtual memory.

### *6.3.1 Experiment Setup*

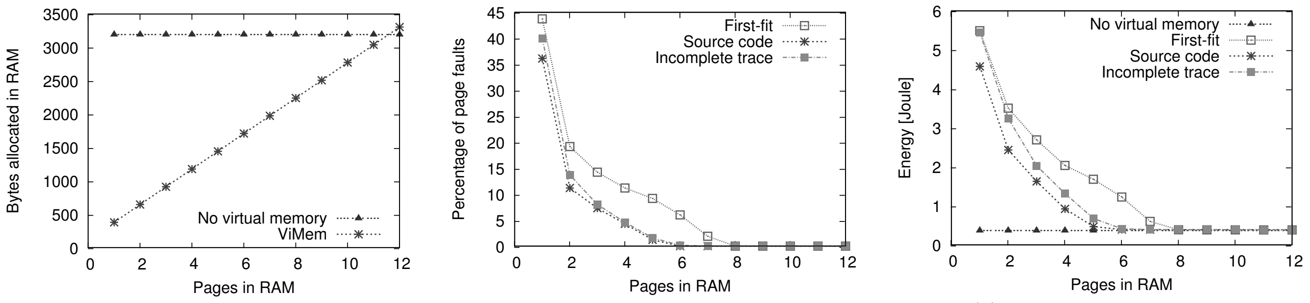
It is not possible to completely evaluate *ViMem* using TinyDB and Maté because in their current implementations these applications do not operate with several kilobytes more RAM than available. It would be interesting to see, however, how *ViMem* performs in this case. Moreover, it is also not possible to evaluate *ViMem* with simple applications, which, for instance, access memory sequentially or randomly. Results from such experiments would not be meaningful, either, because these accesses would not exhibit the patterns found in real applications.

Therefore, in order to evaluate the behavior of *ViMem* with larger data sets we have written a code generator that creates an application with an arbitrary number of data elements whose distribution of memory accesses is based on the accesses of a real application. A difference is, however, that for the generated application each data element has a size of 4 bytes (“uint32_t”). If the number of variables is greater than in the original application, the code generator adds some random jitter to avoid that always the same number of distinct variables is accessed. Therefore, when increasing the total size of data in virtual memory the number of variables actually accessed also grows.

For the experiments described in this subsection we have used this code generator to create applications that perform 5,000 data accesses, which are repeated 10 times. The basis for code generation was a data access trace from TinyDB where no queries were executed.

We have simulated these applications for 1,000 s. After completing the data accesses, the application switches the processor into power down mode, where it consumes less energy. Other hardware devices such as the radio or the LEDs have not been enabled. In addition, no side effects because of other computations done by the application or interactions with neighboring nodes occur. Therefore, using this approach we can measure the pure overhead of the virtual memory system.

We show the results of two sets of experiments. In the first one we have created an application that allocates 3,168 bytes (12 pages) in virtual memory. We have then varied the number of memory pages that are kept in RAM. We intend to evaluate the overall performance of the system here which includes the overhead introduced by the replacement algorithm at runtime. In the second set of experiments we have increased the total size of virtual memory up to 15,840 bytes (60 memory pages) while keeping a constant number of pages in RAM. This is almost four times the size of RAM available on Mica2 sensor nodes. These experiments provide some insight about how *ViMem* performs when varying the



(a) RAM allocation (b) Page fault rate (c) Energy consumption

Figure 8: Varying the number of pages in RAM

data size.

We have simulated the generated applications with different memory layouts. First, where possible, we have created a version that stores all data in RAM without using virtual memory. Obviously, such a variant could not be created for the second set of experiments because the total data size is larger than RAM there. Secondly, we have used a first-fit strategy to place data elements on virtual memory pages in the order in which they have been declared. The third version uses *ViMem*'s memory layout algorithm with information about variable accesses from the source code. Since in our generated application there are no branches or pointers, this variant has a perfect view of all data accesses, which in real applications could only be obtained through simulation. Finally, the last version uses the same trace but omits every fourth access when processing it. The idea of this variant is to see how *ViMem* performs if the operating environment differs from the simulation setup and the exact access trace is not known beforehand.

### 6.3.2 Number of Pages in RAM

Fig. 8(a) shows the amount of RAM statically allocated by the application with and without virtual memory. If *ViMem* keeps just one page in RAM, it saves almost 90 % of RAM and allocates just 378 bytes (including all operating system components and the flash memory component). For each additional page the memory consumption increases by 266 bytes, which is just two bytes more than the size of the memory page itself. Only in the very last measurement *ViMem* allocates more RAM than the variant without virtual memory (overhead: 4 %).

Fig. 8(b) shows the percentage of variable accesses that lead to a page fault when the number of pages in RAM is varied. The results for one page are greater than those presented in Section 6.2.2 because the total data size is larger here. As expected, in these simulations the results using source code optimizations are the best ones. Here the memory layout algorithm can accurately predict the data access patterns.

Of the 3,168 bytes allocated in virtual memory 1,676 bytes are actually accessed. Since this size is much larger than just a few memory pages, conflicts between different pages in RAM occur with any memory layout for 1 and 2 pages in RAM. For these cases the optimized versions do not show much advantage compared to the first-fit approach. In fact, for the version using the incomplete access trace some more write accesses are necessary which further increases energy consumption (see Fig. 8(c)). If, however, with more pages such conflicts no longer occur, *ViMem* is able to create efficient memory layouts. For example, if at least 4 out of 12 memory pages are stored in RAM, the page fault rate for the optimized versions is less than 5 %. From 6 or 7 pages on, respectively, no more pages have to be read from flash. Compared to the unoptimized version *ViMem* avoids up to 3,995 page faults for the 50,000 variable accesses simulated.

Fig. 8(c) presents results for energy consumption. The values shown here just include the energy consumption of the CPU and the flash memory chip. The figure also includes the values for the application that stores all its data in RAM. Of course, this approach performs best with results that cannot be reached by any virtual memory solution that is implemented without additional hardware support. Nevertheless, the results for *ViMem* are encouraging. When no page faults occur, the energy overhead of the virtual memory solutions compared to the RAM-only version is just 0.02 J (5 % overhead). If accesses to flash memory are necessary, roughly half of the total energy is consumed by the CPU and half of it by the flash memory chip.

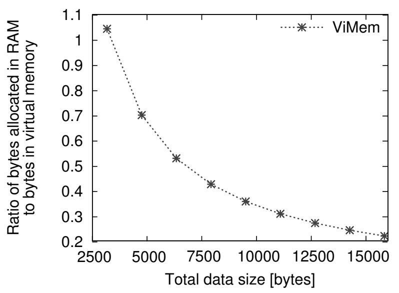
It should be noted that this application represents the worst case for *ViMem* when comparing it to implementations without virtual memory: Energy is only spent to access variables. No other devices such as the radio consume energy and no additional computation is performed by the CPU. For example, if the radio is listening during simulation, energy consumption rises to 39.5 J for the application storing all its data in RAM. Therefore, for real-world applications, the overhead of *ViMem* is less significant, even if some accesses to flash memory are necessary.

Though not shown in the figure, the values for the access latency closely correspond to energy consumption as both of them depend on the number of read and write accesses to the flash memory chip. In the best case its latency is just 4 % of the access latency of the first-fit strategy and only 15 $\mu$s worse than the version storing all its data in RAM.

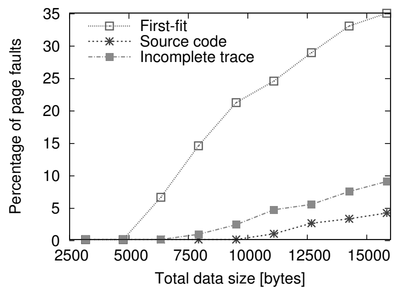
In summary, Fig. 8 suggests that with *ViMem* large amounts of RAM can be saved at justifiable overhead if more than one third of the total number of pages is placed in RAM.

### 6.3.3 Total Data Size

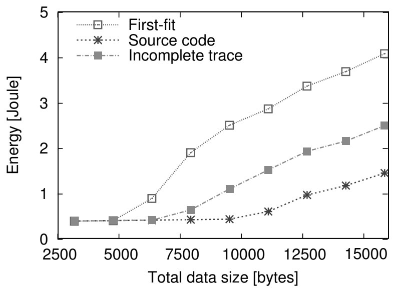
All the previously described applications could be implemented without virtual memory because they allocate less



(a) Relative RAM allocation



(b) Page fault rate



(c) Energy consumption

Figure 9: Varying the size of the data in virtual memory

RAM than the 4 kB available on our implementation platform. With the following experiments we want to find out how *ViMem* performs if the size of the data is increased. Using our code generator we have created applications that allocate up to 15.5 kB of virtual memory. In all simulations the contents of 12 memory pages are kept in RAM.

Fig. 9(a) shows the ratio of bytes allocated in RAM (including all overhead and variables from other components used) to the size of virtual memory. As before, the only case where more memory is allocated in RAM than in virtual memory is the artificial one with all pages in RAM. In all other measurements virtual memory provides much more memory than it allocates in RAM. Each additional page in virtual memory has a RAM overhead of just 3.5 bytes (mostly for the data structures needed because of wear leveling) and provides 264 bytes to the application.

Fig. 9(b) presents the percentage of variable accesses leading to page faults. The numbers for the first-fit version increase to 35 % for a data size of 15,840 bytes (60 pages). For the versions of the applications whose memory layout has been created by *ViMem*'s algorithm, however, a much smaller number of page faults has been measured. For the best version the page fault rate stays smaller than 4.1 % in all cases.

Despite the randomness introduced when increasing the data size only a subset of all variables is accessed. This subset can be kept in RAM all the time if the total data size is less than 11 kB. Only after that the first page faults occur with the best *ViMem* variant. Again, in real applications such good results could only be obtained by using simulation traces for optimizations. If the incomplete data trace is used for optimization, the numbers increase faster but stay well below the first-fit approach. For this version the maximum page fault rate is 9 %.

Fig. 9(c) shows the energy consumed by the different versions of the application. With the best variant the results for the largest data size increase only by 1.1 J compared to the smallest size. For the other *ViMem* version energy consumption increases moderately by 2.2 J whereas the first-fit version with 60 memory pages consumes throughout the simulation 3.7 J more than the 12 page application.

Again, energy consumption is closely related to the average access latency. For this metric the maximum value of the first-fit approach is approximately 1.8 ms. Regarding *ViMem*, the latency increases only to 0.80 ms for the version optimized using the incomplete trace and 0.39 ms for the other one.

In summary, if sufficient information about access patterns is available, *ViMem* is able to significantly enlarge the memory available at only slightly increasing costs.

## 7. CONCLUSIONS AND FUTURE WORK

In this paper we have described and evaluated *ViMem*, our virtual memory system for TinyOS-based sensor nodes. *ViMem* does not require special hardware support for virtual memory and has been implemented for standard Mica2 nodes. We have identified the need for special memory optimizations in the domain of sensor networks and proposed a compiler-based heuristic to create an efficient memory layout. *ViMem* determines the layout based on data access traces obtained from simulation or the source code itself.

Since finding an optimal memory layout is an NP-complete problem, no heuristic can find the best layout for all cases. However, as we have shown in Section 6, our approach can reduce the overhead of virtual memory significantly compared to approaches that just exploit the natural locality of variable declarations. Our simulations show that if the properties of the execution scenario are not known beforehand it is often better to just use the source code for optimizations instead of a simulation scenario that differs too much.

In spite of all optimization efforts, virtual memory introduces some overhead. As we show in the evaluation section, this overhead does not hinder the implementation of non-trivial applications for sensor networks. Compared to other energy consumers on the sensor node, the increase of energy consumption of *ViMem* can be almost neglected. In addition, as sensor network applications are typically inactive for long periods, no virtual memory accesses and, thus, no overhead occur during these sleep times.

From a developer's point of view, using *ViMem* is simple: Virtual memory variables just have to be tagged with a special attribute and can then be used as if they were in RAM. Therefore, the developer does not have to deal with the low-level details of flash memory accesses. In fact, the code expected by our pre-compiler is pure nesC code.

*ViMem* makes it possible to use virtual memory in the domain of sensor networks. Therefore, the memory limitations of sensor nodes are not as strict as before. Even if in future generations of sensor nodes more memory was available, the convenience of an optimizing virtual memory system would help to simplify the development of exciting applications, which are more complex than the ones we know today.

Regarding future work we are going to concentrate on the page replacement algorithm, which has not been the main focus of our research so far. We expect that this way we will be able to reduce *ViMem*'s runtime overhead further.

In addition, since our implementation is currently targeted just on Mica2 nodes, we plan to port it to other hardware platforms. Depending on the properties of these devices there might be some interesting problems to solve. Finally, we will make *ViMem* available to the public to get feedback from a larger community of developers.

## 8. REFERENCES

[1] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han. MANTIS: System support for multimodal networks of in-situ sensors. In *Proc. of the 2nd Int'l Conf. on Wireless Sensor Networks and Applications*, pp. 50–59, 2003.

[2] Atmel Corporation. *4-megabit DataFlash AT45DB041B Datasheet*, 2005.

[3] J. Beutel, O. Kasten, F. Mattern, K. Römer, F. Siegemund, and L. Thiele. Prototyping wireless sensor network applications with BTnodes. In *Proc. of the 1st European Workshop on Sensor Networks*, pp. 323–338, 2004.

[4] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *Proc. of the Eighth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 139–149, 1998.

[5] H. Dai, M. Neufeld, and R. Han. ELF: An efficient log-structured flash file system for micro sensor nodes. In *Proc. of the 2nd Int'l Conf. on Embedded Networked Sensor Systems*, pp. 176–187, 2004.

[6] P. J. Denning. Virtual memory. *ACM Comput. Surv.*, 2(3):153–189, 1970.

[7] A. Dunkels, B. Grönvall, and T. Voigt. Contiki – a lightweight and flexible operating system for tiny networked sensors. In *Proc. of the First Workshop on Embedded Networked Sensors*, 2004.

[8] P. K. Dutta and D. E. Culler. System software techniques for low-power operation in wireless sensor networks. In *Proc. of the Int'l Conf. on Computer-Aided Design*, 2005.

[9] E. Gal and S. Toledo. Algorithms and data structures for flash memories. *ACM Comput. Surv.*, 37(2):138–163, 2005.

[10] D. Gay, P. Levis, D. Culler, and E. Brewer. *nesC 1.2 Language Reference Manual*, 2005.

[11] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. of the Conf. on Programming Language Design and Implementation*, pp. 1–11, 2003.

[12] L. Gu and J. A. Stankovic. t-kernel: Providing reliable OS support to wireless sensor networks. In *Proc. of SenSys 2006*, pp. 1–14, 2006.

[13] R. Gupta. *Compiler Optimization of Data Storage*. PhD thesis, California Institute of Technology, 1991.

[14] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A dynamic operating system for sensor nodes. In *Proc. of the 3rd Int'l Conf. on Mobile Systems, Applications, and Services*, 2005.

[15] S. J. Hartley. Compile-time program restructuring in multiprogrammed virtual memory systems. *IEEE Trans. Softw. Eng.*, 14(11):1640–1644, 1988.

[16] D. J. Hatfield and J. Gerald. Program restructuring for virtual memory. *IBM Systems Journal*, 10(3):168–192, 1971.

[17] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proc. of the 9th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 93–104, 2000.

[18] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proc. of the 2nd Int'l Conf. on Embedded Networked Sensor Systems*, pp. 81–94, 2004.

[19] J. Koshy and R. Pandey. Remote incremental linking for energy-efficient reprogramming of sensor networks. In *Proc. of the 2nd European Workshop on Wireless Sensor Networks*, pp. 354–365, 2005.

[20] O. Landsiedel, K. Wehrle, and S. Götz. Accurate prediction of power consumption in sensor networks. In *Proc. of the Second Workshop on Embedded Networked Sensors*, 2005.

[21] P. Levis, D. Gay, and D. Culler. Active sensor networks. In *Proc. of the 2nd Symp. on Network Systems Design and Implementation*, 2005.

[22] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *Proc. of the Int'l Conf. on Management of Data*, pp. 491–502, 2003.

[23] P. J. Marrón, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel. FlexCup: A flexible and efficient code update mechanism for sensor networks. In *Proc. of the Third European Workshop on Wireless Sensor Networks*, pp. 212–227, 2006.

[24] Maté web page. `http://www.cs.berkeley.edu/~pal/mate-web/`.

[25] S. S. Muchnick. *Advanced Compiler Design & Implementation*. Academic Press, 1997.

[26] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. John Wiley & Sons, 6th edition, 2002.

[27] J. W. Stamos. Static grouping of small objects to enhance performance of a paged virtual memory. *ACM Trans. Comput. Syst.*, 2(2):155–180, 1984.

[28] TinyOS CVS repository. `http://tinyos.cvs.sourceforge.net/tinyos/`.

[29] B. Titzer, D. Lee, and J. Palsberg. Avrora: Scalable sensor network simulation with precise timing. In *Proc. of the Fourth Int'l Conf. on Information Processing in Sensor Networks*, pp. 477–482, 2005.

[30] B. Titzer and J. Palsberg. Nonintrusive precision instrumentation of microcontroller software. In *Proc. of the Conf. on Languages, Compilers, and Tools for Embedded Systems*, pp. 59–68, 2005.

[31] G. Tolle, J. Polastre, R. Szewczyk, D. Culler, N. Turner, K. Tu, S. Burgess, T. Dawson, P. Buonadonna, D. Gay, and W. Hong. A macroscope in the redwoods. In *Proc. of the 3rd Int'l Conf. on Embedded Networked Sensor Systems*, pp. 51–63, 2005.