Efficient Flash-based Virtual Memory for Sensor Networks

Andreas Lachenmann, Pedro José Marrón, Kurt Rothermel

Universität Stuttgart, Germany
{lachenmann,marron,rothermel}@ipvs.uni-stuttgart.de

Abstract. In this paper we present a virtual memory system for sensor networks that uses flash-memory to extend the amount of RAM available on each node. By analyzing access traces from simulation tools it creates an efficient memory layout that makes virtual memory viable despite the resource constraints typical for sensor networks.

1 Introduction

Most sensor nodes are equipped with just a few kilobytes of RAM. Therefore, main memory is usually a very scarce resource when developing sensor network applications. In fact, several applications already require more memory than available on current sensor nodes. For instance, TinyDB [1] requires the user to select at compile-time which functionality to include in the code image.

As applications for sensor networks increase in complexity, RAM limitations will continue to cause difficulties for developers. In traditional computing systems, virtual memory [2, 3] 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 (e.g., [4, 5]) do not include support virtual memory.

Sensor nodes are equipped with flash memory as secondary storage, which is much larger than main memory (between 512 kB and 1 MB). 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 energy expensive. Nevertheless, this type of memory is appropriate for the implementation of virtual memory on sensor nodes.

In this paper we present *ViMem*, a virtual memory system for TinyOS-based sensor networks that 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 flash accesses is necessary.

2 Design Overview

ViMem consists of two main parts: a compiler extension and a runtime component, which are described in this section.

Application 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 [6], the programming language used by TinyOS [4]. We have selected nesC and TinyOS because of their active research community that has developed a large number of application and system components.

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. The pre-compiler executes the memory layout algorithm described in Section 3 to place variables on pages in virtual memory.

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 (writing modified pages is more expensive than just reading).

This algorithm was not the main focus of our research. Therefore, for *ViMem*'s replacement policy we have adapted the Enhanced Second-Chance Algorithm [3], which approximates a least-recently used (LRU) page replacement strategy.

3 Memory Layout Algorithm

This section describes our memory layout algorithm 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 algorithm has two main goals: First, it aims to reduce the overall number of page replacements. Second, it puts special effort in decreasing the number of write accesses to flash memory.

3.1 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. 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, there are typically patterns that recur. Our algorithm uses simulation traces to determine such patterns for variables stored in virtual memory. Gathering information about variable accesses using simulation does not introduce any overhead at runtime and, thus, does not alter the behavior of the application itself.

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.

The pre-compiler splits up complex variables, such as 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".

3.2 Grouping of Data Elements

Having gathered information about accesses to data elements, the memory layout algorithm forms groups of those that are 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 $f : V \to \mathbb{R}$, indicates how often the corresponding data element has been accessed, and the weight of an edge, defined by $g : E \to \mathbb{R}$, gives information about the proximity of two data elements.

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 being present. The sum of the sizes of all these elements may not exceed the size of a flash page. Thus, these elements represent those that should be preferably in RAM when the new element is accessed. When the *ViMem* pre-compiler adds a data access from the trace, it increments both the access count in the data element's node and the proximity value of all data elements accessed previously.

Fig. 1 shows an example of 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. The figure shows the simple variables "a", "b", and "c" as well as struct "s". 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 following changes. 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

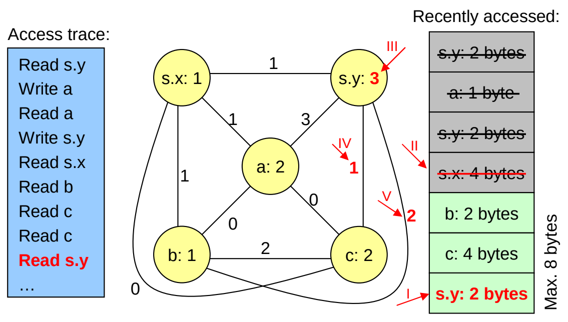

Fig. 1. Example for processing an access trace

removes the oldest element ("s.x", II). Then it increments the weights of "s.y" (III) and of all its edges to elements in the list (IV and V).

After reading the complete access trace, *ViMem*'s pre-compiler traverses the graph G using a breadth-first search that takes into account the proximity values. It begins with one of the nodes from which the edge with the maximum proximity value starts. Being at node v, the search follows an edge e only if $g(e)/f(v) > k$, where k is a threshold that indicates which data elements should be definitely put on one page of flash memory. The search is aborted if the elements reached no longer fit on a single page. All data elements that are reached using this search mechanism are grouped in order to be placed on the same memory page.

3.3 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 algorithm processes the elements in the order of their access frequencies and places them with a first-fit strategy. 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.

The algorithm uses two sets of pages: one with elements that are mostly read and one with those that are modified more often. If a "mostly-read" page has to be removed from RAM, this approach makes it more likely that it does not have to be written back to flash memory.

4 Simulation Results

We modified TinyDB to make use of *ViMem* and obtained simulation results with the Mica2 simulator Avrora [7]. In Fig. 2 we show the number of page faults, i.e., the number of read accesses from secondary storage. In many cases *ViMem*'s memory layout algorithm greatly reduces the number of accesses to flash memory. Using only the data references from the source code it is already able to decrease the percentage of accesses leading to page faults by more than 60%. Nevertheless, memory layouts that have been optimized for a given scenario can reduce the percentage of page faults even further by additional 80%.

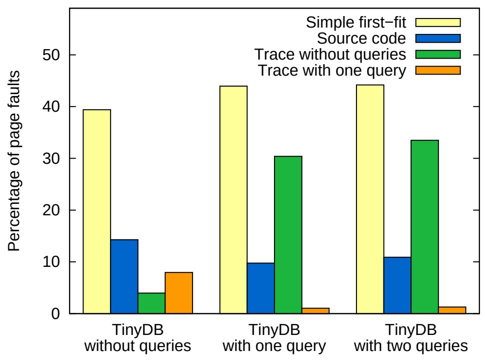

Fig. 2. Number of page faults

5 Summary

In this paper we have described *ViMem*, our virtual memory system for TinyOS-based sensor nodes. It uses a compile-time approach to create an efficient memory layout based on data access traces obtained from simulation. We have presented results that show that this algorithm reduces the overhead of virtual memory significantly. The remaining overhead does not hinder the implementation of complex applications for sensor networks. Therefore, using *ViMem* the memory limitations of sensor nodes are not as strict as before.

References

1. Madden, S., Franklin, M.J., Hellerstein, J.M., Hong, W.: The design of an acquisitional query processor for sensor networks. In: Proc. of the Int'l Conf. on Management of Data. (2003) 491–502
2. Denning, P.J.: Virtual memory. ACM Comput. Surv. **2**(3) (1970) 153–189
3. Silberschatz, A., Galvin, P.B., Gagne, G.: Operating System Concepts. 6th edn. John Wiley & Sons (2002)
4. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K.: System architecture directions for networked sensors. In: Proc. of the 9th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems. (2000) 93–104
5. Dunkels, A., Grönvall, B., Voigt, T.: Contiki – a lightweight and flexible operating system for tiny networked sensors. In: Proc. of the First Workshop on Embedded Networked Sensors. (2004)
6. Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., Culler, D.: The nesC language: A holistic approach to networked embedded systems. In: Proc. of the Conf. on Programming Language Design and Implementation. (2003) 1–11
7. Titzer, B., Lee, D., Palsberg, J.: Avrora: Scalable sensor network simulation with precise timing. In: Proc. of the Fourth Int'l Conf. on Information Processing in Sensor Networks. (2005) 477–482