I. J. Communications, Network and System Sciences, **2008, 3, 207-283**
Published Online August 2008 in SciRes (http://www.SciRP.org/journal/ijcns/).

Hardware/Compiler Memory Protection in Sensor Nodes

Lanfranco LOPRIORE
Dipartimento di Ingegneria dell'Informazione: Elettronica, Informatica, Telecomunicazioni
Università di Pisa, via G. Caruso 16, 56122 Pisa, Italy
E-mail: l.lopriore@iet.unipi.it
Received on June 9, 2008; revised and accepted on August 25, 2008

Abstract

With reference to sensor node architectures, we consider the problem of supporting forms of memory protection through a hardware/compiler approach that takes advantage of a low-cost protection circuitry inside the microcontroller, interposed between the processor and the storage devices. Our design effort complies with the stringent limitations existing in these architectures in terms of hardware complexity, available storage and energy consumption. Rather that precluding deliberately harmful programs from producing their effects, our solution is aimed at limiting the spread of programming errors outside the memory scope of the running program. The discussion evaluates the resulting protection environment from a number of salient viewpoints that include the implementation of common protection paradigms, efficiency in the distribution and revocation of access privileges, and the lack of a privileged (kernel) mode.

Keywords: Access Control, Protection Domain, Protection System, Sensor Node

1. Introduction

In sensor node architectures, stringent limitations in terms of hardware complexity and energy consumption [1] prevent utilization an intrinsically complex device such as a memory management unit for virtual to physical address translation [2]. This situation is not likely to change in the near future. Rather than incrementing the hardware power of the single sensor node, system designers are likely to take advantage of progress in integration technologies to reduce the node size and cost, so as to support new applications using sensor networks connecting an always increasing number of nodes [3,4].

In the absence of a memory mapping device, a single address space is shared by all processes, and the form of protection enforced by address space separation between processes is lacking. The code and data areas of all applications are exposed to the risk of corruption by an erroneous process that can even crash the system kernel [5]. This problem is exacerbated by the fact that the writing of application software for sensor nodes is an especially challenging activity, owing to the limitations in terms of available memory and processing power, event-driven concurrency, requirements of real-time response, dynamic application update, and the need to comply with a variety of different sensors. Even worse, programmers may usually rely on very limited support for debugging [2]. These considerations suggest that the presence of protection mechanisms between processes, which is a common feature in general-purpose systems, is highly desirable even in sensor node environments.

By taking the salient characteristics of an environment of this type into considerations, we shall propose a form of fine-grained memory protection [6] as a solution to the protection problem, outlined above. Our solution takes advantage of a form of synergy between the hardware and the compiler. The interface of the protection hardware consists of a set of primitives, the *protection operations*. The compiler inserts the calls to these operations at appropriate points of the object code to enforce separation of memory privileges between tasks while preventing the application programmer from calling these operations explicitly. These are easy compiler tasks, which can be made largely transparent to the programmer. Placing new burdens on the compiler is a tendency now exploited in the solution of several architectural problems, e.g., data prefetching, cache control, translation lookaside buffer management, and instruction scheduling at compile time.

The rest of this paper is organized as follows. Section 2 introduces a simple, low-cost addition to the hardware

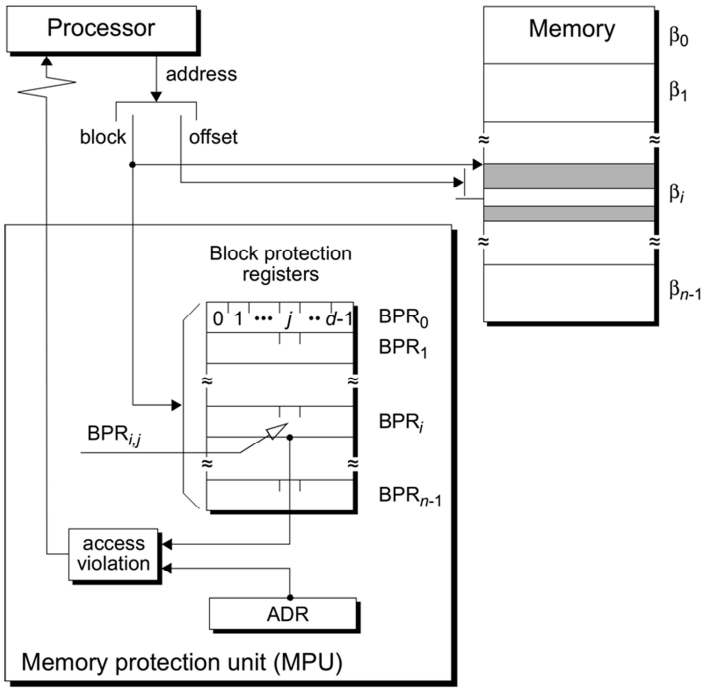

Figure 1. Hardware configuration featuring a memory protection unit interposed between the processor and the memory devices.

inside the microcontroller, and the software primitives to control this hardware and enforce memory protection. Section 3 evaluates the resulting *hardware/compiler* memory protection scheme from a number of salient viewpoints that include the implementation of common protection paradigms, efficiency in the distribution and revocation of access permissions, and the lack of a privileged (kernel) mode. For each protection problem, we devise a solution that demonstrates the flexibility of the proposed approach to memory protection.

2. The Protection System

We shall refer to a classical sensor node configuration in which a microcontroller includes a processor that interfaces both volatile (RAM) and non-volatile reprogrammable (Flash/ROM) storage devices. The memory space is logically partitioned into 2^n *blocks* of a fixed size. Blocks are the passive entities to be protected from tasks. By the term *task* we mean any active entity capable of generating memory accesses; thus, a task may be a scheduled computation [5], or, in an event-driven paradigm, the activity produced by a function activated by a hardware interrupt [3,4].

A *protection domain* is a collection of *access permissions* for memory blocks which can be randomly scattered throughout the whole memory. When a task is running, it is associated with a domain, called the *active* domain. When the task performs an access attempt to a given information item in memory, the access terminates successfully only if the active domain includes access permission for the block storing that information item.

2.1. Hardware Support for Memory Protection

At the hardware level, protection is supported by a circuitry inside the microcontroller, the *memory protection unit* (MPU), interposed between the processor and the memory devices (Figure 1). For each given memory block β_i, $i = 0, 1, \ldots, n - 1$, MPU contains a *block protection register* BPR_i associated with this block. The size of BPR_i is d bits, where d is the number of the *basic* domains $\Delta_0, \Delta_1, \ldots, \Delta_{d-1}$ which are supported by the protection system (as will be made clear later, more domains can be defined in terms of unions of the basic domains). Let $BPR_{i,j}$ denotes the j–th bit of BPR_i. If set, this bit specifies that domain Δ_j holds access permission for block β_i. This means that a task running in Δ_j can successfully access the memory locations in β_i for both read and write.

At any given time, a d-bit register of MPU, the *active domain register* (ADR), contains a quantity with a single bit set, the j-th bit corresponding to the name Δ_j of the domain that is active at that time. An address generated by the processor is partitioned into the index i of a block β_i and an offset within this block. Quantity i is sent to the array of block protection registers to select the register BPR_i associated with β_i. If the result of the bitwise AND of the contents of ADR and BPR_i is 0, then domain Δ_j has no access permission for β_i, and an exception of

protection violation is raised to the processor. If this is not the case, the memory address is delivered to the storage devices and the access to memory is accomplished successfully.

When a given task is running, ADR contains the bit configuration corresponding to the name Δ_j of the domain of this task. As a result, the task can freely access (and even corrupt) all the blocks in this domain; whereas it cannot read or modify the contents of the other blocks. In this way, we implement a form of error confinement. Corrupting the memory areas in the task's own domain is less serious than corrupting information items outside the boundaries of this domain.

Of course, more bits of BPR_i can be set at the same given time, to indicate block sharing between domains. If both bits $BPR_{i,j}$ and $BPR_{i,k}$ are set, then block β_i is shared by domains Δ_j and Δ_k, for instance.

Table 1. Protection operations.

Operation	Effect
setDomain(Δ)	Activates domain Δ.
grantAP(β, Δ)	Grants access permission for block β to domain Δ. Fails if the active domain does not include this access permission.
revokeAP(β, Δ)	Revokes access permission for block β from domain Δ. Fails if the active domain does not include this access permission.

2.2. Protection Operations

A set of primitives, the *protection operations*, makes it possible to access the active domain register and the block protection registers and modify their contents (Table 1). Let *bi*, *dj* and *dk* denote bit configurations featuring a single bit set, i.e. the *i*-th, the *j*-th and the *k*-th bit, respectively. Operation *setDomain*(*dj*) writes quantity *dj* into ADR, thereby activating domain Δ_j. Operation *grantAP*(*bi*, *dk*) sets bit $BPR_{i,k}$, thereby granting access permission for block β_i to domain Δ_k. Execution terminates successfully only if the active domain Δ_j, as specified by the contents of ADR, includes the access privilege to be granted, i.e. bit $BPR_{i,j}$ is asserted. Finally, operation *revokeAP*(*bi*, *dk*) clears bit $BPR_{i,k}$, thereby revoking the access permission for block β_i from domain Δ_k. Execution terminates successfully only if the active domain Δ_j includes the access privilege to be revoked, i.e. bit $BPR_{i,j}$ is asserted. The protection operations are idempotent; each of these operations yields the same result after applying it multiple times.

It should be clear that a harmful task could well use the protection operations unfairly, to change the active domain and gain control of the blocks in a different domain, for instance. We rely on the compiler to prevent the programmer from inserting explicit calls to the protection operations into application programs; whereas these calls will be inserted by the compiler at appropriate points of the program code. Of course, it would be easy for the programmer to circumvent a loose protection of this type. Rather that precluding deliberately harmful programs from producing their effects, our protection environment is aimed at confining the consequences of programming errors within the memory scope of the running program.

3. Discussion

3.1. The Protection Model

In a traditional paradigm, a protection system is modeled by using an *access matrix* with one row for each protected object β_0, β_1, …, β_{n-1} and one column for each protection domain Δ_0, Δ_1, …, Δ_{d-1} (Figure 2). The matrix element corresponding to a given object and a given domain specifies the access rights held by this domain on the object. In a representation by rows, the access matrix takes the form of a set of *access control lists*, one list for each protected object; the access control list of a given object specifies the access rights held by each domain on this object. In a representation by columns, the access matrix takes the form of a set of *capability lists*, one list for each domain; the capability list of a given domain specifies the access rights held by this domain on each protected object [7].

Access control lists make it easy to manage the access rights held by all domains for a given object. However, determining the access rights that form a given domain is a costly action that implies inspection of all access control lists. Capability lists allow straightforward administration of the access rights in a given domain and facilitate actions of access right transmission between domains. However, access rights tend to propagate throughout the system. This makes it hard if not impossible to determine the domains that hold access rights for a given object, as is required to revoke access permissions, for instance [8].

In our protection environment, by reserving a bit for each domain, register BPR_i implements the concept of an access control list for block β_i (Figure 3). Furthermore, the bits in position *j* of all block protection registers, considered as a whole, form the capability list of domain Δ_j. In facts, the array of block protection registers is our hardware implementation of the access matrix, which allows us to take advantage of both methods of access right representation, access control lists and capability lists.

For instance, in the capability list approach, access right transmission between domains corresponds to execution of the *grantAP*() operation that copies an access permission from the active domain into a given domain. This can be useful to pass ownership of a buffer

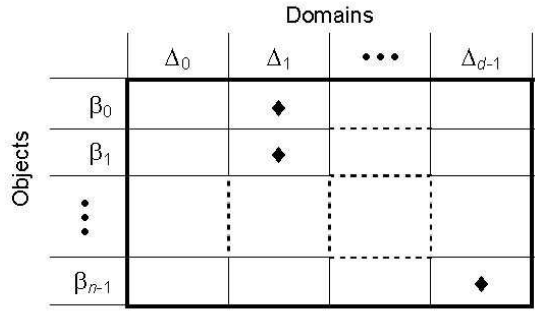

Figure 2. Configuration of the access matrix. The ♦ symbol in a given element of the matrix indicates that the domain of the corresponding column holds access permission for the object of the corresponding row.

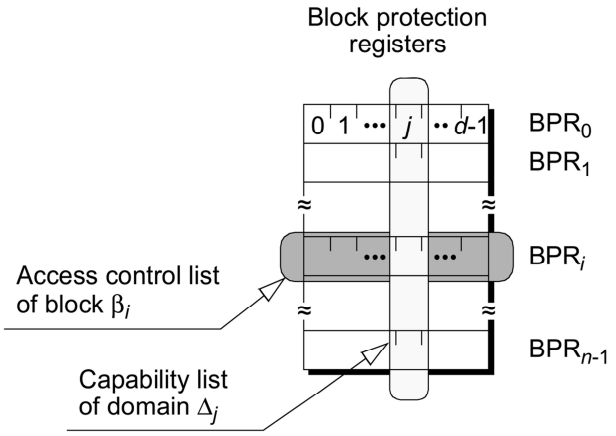

Figure 3. Hardware implementation of the access matrix.

between tasks, for instance. In the access control list approach, access privilege revocation is obtained by executing the *revokeAP*() operation and eliminating the access permission for a given object from a given protection domain. Revocation is important when the sharing of a data item is done on a temporary basis, for instance. In spite of its simplicity, this technique allows *selective* revocation of an access privilege from any subset of the domains that hold this privilege [9].

3.2. Memory Protection and the Privileged Mode

At the hardware level, the classical concept of a privileged (kernel) mode corresponds to both a set of privileged instructions and unlimited device access. Increased hardware complexity follows in the implementation of the instruction set as well as in processor interfacing. At the software level, system efficiency is negatively affected by the need to save and then restore the context of the running task at each system call [5]. Furthermore, the privileged mode prevents in-line expansion of the system calls [10]. All these sources of processor time wastage give raise to additional energy costs.

We give no special privilege to the kernel. As a result, we may well expand the protection operations in-line into the object code. In-line expansion will be straightforward and very effective. In fact, at the assembly language level, the protection operations translate into few instructions or even a single instruction (as may be the case for *setDomain*(), for instance). We never disable protection. Instead, we limit the scope of each application and even of the kernel to the smallest extent necessary to carry out its job.

Of course, our protection hardware may well emulate the unrestricted memory access of a traditional privileged mode. A result of this type will be obtained by writing the all-one bit configuration into the active domain register. The negative effects of an approach of this type on overall system stability are well known [11]. A better solution is to have the kernel run in its own, separate domain. In this way, a stable kernel can always guarantee a form of cold restart after a system crash due to application memory corruption [12].

3.3. Protection Domain Switching

Let us first refer to an event-driven environment featuring non-blocking functions activated by hardware interrupts. In an environment of this type, when execution of a function is started up, the active domain must change to reflect the memory scope of the new function. A result of this type will be obtained by reserving a specific domain to that function or to a set of correlated functions sharing a common memory scope. The compiler will use the *setDomain*() operation to produce the necessary domain switch, by inserting a call to this operation at the beginning of the code of each of these functions.

A problem connected with domain switching is that of restoring the previous domain on termination of execution of the activities in the new domain. A common approach relies on a protection stack where to save the name of the old domain. Given the memory restraints of sensor node environments, the cost of a separate stack for each task is usually considered prohibitive [12]. We shall take advantage of the idempotence property of the protection operations. On returning from the new domain, the caller will use *setDomain*() to restore the original domain, independently of possible situations of coincidence of the new domain with the old.

Of course, the approach outlined above to treat asynchronous hardware interrupts can as well be used to deal with synchronous system calls issued by tasks explicitly. In a system featuring no memory protection, system calls may well take the simple form of a library of

software routines [10]; whereas protection usually forces these calls to be implemented as traps. In our environment, we are in a position to take advantage of linked system primitives or even in-line expansion of the system calls in the application code while preserving separation of access privileges between applications and the system kernel.

Let us now consider a task starting up execution of an operation on a given encapsulated object. In a situation of this type, the memory scope of the task should be enlarged to include the memory area reserved for the internal representation of this object. We can implement this form of amplification of access rights at little effort by relaxing the constraint that, at any given time, the active domain register must contain only one bit set. So doing, we can define the active domain in terms of the union of two or more basic domains, by setting the bits of ADR that correspond to these domains. In our example, let Δ_j be the active domain and Δ_k be the domain including the internal representation of the encapsulated object. We shall use *setDomain*() to replace the contents of ADR with the result of the bitwise OR of these contents and a quantity having a single bit set, the *k*-th bit corresponding to Δ_k. So doing, we expand the active domain to be the logic union of the original domain and the domain of the object.

3.4. Hardware Costs

As seen in the Introduction, the overall design of a sensor node must carefully comply with stringent requirements in terms of energy consumption and space efficiency [13,14]. As far as protection is concerned, the processor time required to manage the protection information has an energy cost that must be kept low [5], and the memory space reserved for storage of the protection information should be kept to a minimum.

In our design, the protection hardware is a small fraction of the total. Its overall complexity is much lower than that of a memory management unit supporting address translation besides protection. For eight basic protection domains (not counting the domains defined in terms of unions of the basic domains), the cost of the protection circuitry in term of the memory resources for the block protection registers and the active domain register is $n + 1$ bytes, n being the number of memory blocks. Thus, the memory requirements of the protection information are kept low. Furthermore, the memory protection strategy neither is an inherent source of memory space waste (by implying a separate stack for each process [3,5], for instance) nor produces processor inefficiencies (e.g., by hampering in-line expansion of the calls to the protection operations and the system kernel).

4. Concluding Remarks

A widely-used approach to the construction of sensor node software is to compile and link all applications and the kernel, and then load the resulting system image into the sensor node; the software is now operational as a whole [4]. An alternative is to permit forms of dynamic linking of application programs, to bring a new application into the system or to upgrade an existing application, for instance [15,16]. In both cases, in the absence of a privileged mode and of address space separation between applications, no protection mechanism inhibits application software from corrupting code and data in memory, even within the scope of the kernel.

On the other hand, the ever increasing complexity of sensor node software deserves special attention from the system architect, especially given the possible effects of programming errors, which may spread even outside the node onto the whole sensor network [12].

The costs in terms of both hardware and energy requirements connected with classical forms of memory management and protection are usually considered prohibitive for a sensor node. This is certainly true for a memory management unit supporting address translation and address space separation between processes. On the other hand, we have shown how to take advantage of a synergy between the hardware and the compiler and implement a form of memory protection between application programs and the kernel, at low costs in terms of additional hardware inside the microcontroller, processing time, memory space requirements and energy consumption.

We have been aimed at safety rather than security [10]. The software installed on a sensor node is usually considered as reliable, but, of course, not bug free. Our purpose has been to limit the spread of programming errors outside the memory scope of the running program [17]. This means that we hypothesize a set of protection mechanisms at the network level, preventing delivery to nodes of deliberately harmful programs.

In our opinion, in a sensor node environment, the advantages that ensue from a hardware/compiler approach to protection may well compensate the disadvantages connected with the lack of a privilege mode and of address space separation between the processes. We hope that our design effort will be a significant contribution in this direction.

5. References

[1] T. Liu, C. M. Sadler, P. Zhang, and M. Martonosi, “Implementing software on resource-constrained mobile sensors: experiences with Impala and ZebraNet,” Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services, Boston, Massachusetts, USA, pp. 256–269, June 2004.

[2] R. Kumar, A. Singhania, A. Castner, E. Kohler, and M. Srivastava, “A system for coarse grained memory

protection in tiny embedded processors," Proceedings of the 44th Annual Conference on Design Automation, San Diego, California, USA, pp. 218–223, June 2007.

[3] A. Dunkels, B. Grönvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," Proceedings of the First IEEE Workshop on Embedded Networked Sensors, Tampa, Florida, USA, pp. 455–462, November 2004.

[4] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, "TinyOS: an operating system for wireless sensor networks," in Ambient Intelligence, New York: Springer-Verlag, pp. 115–148, 2005.

[5] H. Cha, S. Choi, I. Jung, H. Kim, H. Shin, J. Yoo, and C. Yoon, "RETOS: resilient, expandable, and threaded operating system for wireless sensor networks," Proceedings of the 6th International Conference on Information Processing in Sensor Networks, Cambridge, Massachusetts, USA, pp. 148–157, April 2007.

[6] J. Shen, G. Venkataramani, and M. Prvulovic, "Tradeoffs in fine-grained heap memory protection," Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability, San Jose, California, USA, pp. 52–57, October 2006.

[7] L. Lopriore, "Access control mechanisms in a distributed, persistent memory system," IEEE Transactions on Parallel and Distributed Systems, Vol. 13, No. 10, pp. 1066–1083, October 2002.

[8] L. Lopriore, "Access privilege management in protection systems," Information and Software Technology, Vol. 44, No. 9, pp. 541–549, June 2002.

[9] V. D. Gligor, "Review and revocation of access privileges distributed through capabilities," IEEE Transactions on Software Engineering, Vol. SE-5, No. 6, pp. 575–586, November 1979.

[10] D. Lohmann, J. Streicher, W. Hofer, O. Spinczyk, and W. Schröder-Preikschat, "Configurable memory protection by aspects," Proceedings of the 4th Workshop on Programming Languages and Operating Systems, Stevenson, Washington, USA, October 2007.

[11] A. S. Tanenbaum, J. N. Herder, and H. Bos, "Can we make operating systems reliable and secure," Computer, Vol. 39, No. 5, pp. 44–51, May 2006.

[12] R. Kumar, E. Kohler, and M. Srivastava, "Harbor: software-based memory protection for sensor nodes," Proceedings of the 6th International Conference on Information Processing in Sensor Networks, Cambridge, Massachusetts, USA, pp. 340–349, April 2007.

[13] A. Eswaran, A. Rowe, and R. Rajkumar, "Nano-RK: an energy-aware resource-centric RTOS for sensor networks," Proceedings of the 26th IEEE International Real-Time Systems Symposium, Miami, Florida, USA, pp. 256–265, December 2005.

[14] S. Yi, H. Min, J. Heo, B. Gu, Y. Cho, J. Hong, H. Oh, and B. Song, "XMAS: An eXtraordinary Memory Allocation Scheme for resource-constrained sensor operating systems," in: Mobile Ad-hoc and Sensor Networks, Lecture Notes in Computer Science, Vol. 4325, pp. 760–769, 2006.

[15] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt, "Run-time dynamic linking for reprogramming wireless sensor networks," Proceedings of the 4th International Conference on Embedded Networked Sensor Systems, Boulder, Colorado, USA, pp. 15–28, October 2006.

[16] C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A dynamic operating system for sensor nodes," Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services, Seattle, Washington, USA, pp. 163–176, June 2005.

[17] N. K. Jha, S. Ravi, A. Raghunathan, and D. Arora, "Architectural support for safe software execution on embedded processors," Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis, Seoul, Korea, pp. 106–111, 2006.

