Using Preemption in Event Driven Systems with a Single Stack

Karsten Walther, Reinhardt Karnapke, Andre Sieber and Jörg Nolte
Distributes Systems/Operating Systems group
Brandenburg University of Technology
Cottbus, Germany
Email:{kwalther, karnapke, as, jon}@informatik.tu-cottbus.de

Abstract

Event driven programming is widely used in sensor networks because of the high resource efficiency and the ease of synchronization in such systems. Also, the application logic itself can often be expressed naturally with an event handling system. However, handling of long running tasks is a severe problem, because most systems imply non-preemptive run-to-completion semantics of tasks. Therefore, a long running task must be split in several parts to prevent it from blocking other ones. In this paper we present a preemptive stack sharing approach which preserves the benefits of event driven programming, while solving the long running task problem.

1 Introduction

Sensor networks are an important research topic and currently crossing the border to be used by non experts. Because of the severe resource constraints in energy and memory, first operating systems like TinyOS [9] were tailored to a minimum of functionality, but also to a minimum of programming convenience. With the expieriences made with these systems, it became clear that event driven programming is a good idea because of the inherent efficiency and because it solves many problems of threaded systems like the precedence problem [16].

However, there are also drawbacks. In event driven systems tasks cannot perform blocking wait operations like threads and long running tasks might delay the execution of other, probably more urgent tasks. While the former problem has been addressed by approaches like Protothreads [7], the latter is still an open problem in the area of wireless sensor networks. However, in the context of real time systems Baker has proposed a solution that is based on preemptive scheduling using a single stack only [1]. The prerequisites for this approach are both run-to-completion semantics for the tasks and a priority based scheduling scheme. We have adopted Baker's approach in the REFLEX [18] system to tackle the problem of long running tasks, while the problem of blocking tasks can be solved with concepts like protothreads, which are conceptually orthogonal to our approach.

The reminder of the paper is structured as follows. Section 2 provides a short overview of the evolution of sensor network operating systems. Section 3 introduces REFLEX and discusses the implementation of preemptive scheduling schemes in single threaded systems. In section 4 different aspects of this approach are evaluated in terms of memory consumption and runtime overhead. Finally we provide a conclusion in section 5.

2 Related Work

There is a long history of publications on the advantages or disadvantages of the event driven or the thread driven design of systems. Lauer and Needham have stated that both methologies are duals [14], but each has advantages over the other in special contexts. In the sensor network community event driven programming is widely used. There are several reasons for that, the most important is maybe the inherent efficiency [16] and the ease of synchronization [17]. Last but not least sensor networks are working reactively and therefore it is natural to work by means of events.

The most widely used operating system for sensor network nodes seems to be TinyOS [9], including a number of variations and extensions like TinyGALS [4]. One of the restrictions of the system is that it is not preemptive. This is problematic for complex tasks, which block others and lead to timing problems in real world applications [13].

There have been several proposals of mechanisms to integrate threads into TinyOS. In [8] the authors run TinyOS on top of the threaded Mantis operating system [2]. The blocking tasks are executed by the Mantis scheduler and the non-blocking ones by the TinyOS scheduler. Unfortunately, the solution has several drawbacks. Each application contains two runtime systems, this wastes memory. Furthermore, the source of an event must be aware of the type of the receiver (blocking or non-blocking task) and use the matching event post mechanism. Last but not least the application must be carefully synchronized by the user.

In [5] the authors present an approach which allows multiple blocking tasks in TinyOS. The concept allows also to mark a task as non interruptable. One drawback is the problem that the stack for each blocking task has to be allocated manually, which is a non trivial problem. The authors also mention that all the synchronization has to be taken care of by the programmer too.

Contiki [6] is also an event driven system, which can be used for sensor network applications. A major contribution are the so called Protothreads [7], that allow the programmer to write blocking code sequentially like in threaded systems. This is achieved by macros which store the state of the task and returns to the system. The next time the task is invoked, the execution continues at the stored location. Thus protothreads help dealing with writing state machine code. But the handling of long running tasks is not adressed by Protothreads.

There are also some approaches of threaded operating systems for nodes in the sensor network class like AVRx [10], FreeRTOS [11], Mantis [2] or RETOS [3]. They all have in common the drawbacks of threaded programming, like stack allocation and manual synchronization. Furthermore, except for Mantis, they do not provide suitable power management facilities. RETOS is proposed to solve the problem of long running tasks by so called event aware thread scheduling. But as mentioned already, the benefits of events are dropped in the system. Moreover, the kernel already consumes more than 20KB of memory.

3 Motivation

A typical application for a sensor network is habitat monitoring as described in [15]. Consider an example, where sample values are taken by each node every minute. Because the lifetime of the network should be maximised and transmitting values requires a lot of energy, the sampled values are stored inside each node. Every night, the nodes aggregate and compress the sampled values and transmit the compressed information to a base station. This transmission involves a routing layer for multihop forwarding and a TDMA based medium access control for collision avoidance (figure 1).

There are multiple timing problems arising from this scenario. The sampling needs to be done with a constant rate. A timer should be used, which initiates sampling every

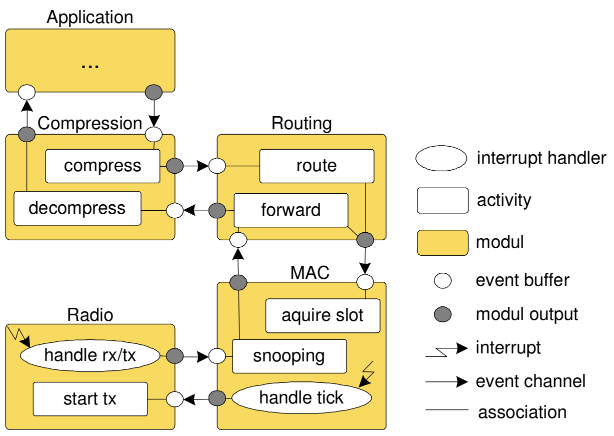

Figure 1. Network part of a typical sensor network application

minute. Each day at midnight, the compression is started. As there are 1440 values to aggregate and prepocess, this task easily needs several ten-thousand CPU cycles. Therefore, the compression can seriously affect timing of other tasks in the system, if tasks are non-preemptive.

In the application the MAC component is critical. In a dense network, a TDMA based MAC should be used because a contention based one would lead to a lousy throughput. In a TDMA based MAC each node acquires a slot in which it may transmit. This may be done a priori or on demand. Either way, the node is assigned a time slot. When that slot arrives, the node must transmit waiting messages at once. It can not wait for the compression to finish. Moreover, there may be incoming messages that have to be processed while the compression is still running. The MAC layer needs to read information about used slots from the incoming packets and may have to synchronize the clock. The routing needs to forward packets and to repair broken routes.

All these problems arise because of the long running compressing task. Of course this problem could be solved with a tailor-made compression stage, which splits the compression in several shorter runs. This has two drawbacks. First the runtime overhead is increased by unnecessary scheduling operations. Second the granularity of splitting would have to be adapted every time the behaviour of the application was changed, this reduces the re-usability of the component.

Our approach allows the usage of a general purpose compression stage, configured with a low priority. The routing and MAC stages would receive a higher priority, allowing them to interrupt the compression when nessesary.

4 Concept Discussion

We want to integrate long running tasks into an event driven system without using multiple threads. As operating system base we use REFLEX, whose event model is explained below. Afterwards we discuss the implementation of preemptive scheduling schemes with a single stack.

4.1 The Event Model of REFLEX

REFLEX (**R**eal-time **E**vent **FL**ow **EX**ecutive) is an object oriented operating system for deeply embedded control systems and sensor nodes. Applications are programmed according to the so called event flow model. In that model the schedulable entities are activities, that are triggered when events are posted to associated event buffers. Triggered activities are then invoked by the scheduler according to the chosen scheduling strategy. All activities have run-to-completion semantics like in other event driven systems. However, since activities are objects (instances of C++ classes) rather than functions, they can easily preserve important state information across multiple activations without the need for a private stack.

Events can be associated with data but also pure events are allowed. They are raised either by interrupt handlers or other activities. The initial source of any activity in the system is always an interrupt. Figure 2 shows a sense and send application for Reflex.

Usually several activities and interrupt handlers are combined to form functional components, such as device drivers or network protocol engines. The communication between the components is always asynchronous, while inside the components the activities and handlers can share state information. This concept is similar to TinyGALS (Globally Asynchronous Locally Synchronous) [4]. The event buffers needed at the inputs of the components can be of any type: event counters (for dataless events), queues, fifos or simple value buffers are already supplied. Since all buffers count the posted events, the related activities can be scheduled exactly once for each post.

Note, that the access to buffers is atomic for readers and writers. Most applications are therefore implicitly synchronized [12]. Furthermore, the base model makes no assumption about the applied scheduling scheme. Details of the scheduling schemes are hidden in the `Activity` base class that is chosen at system configuration time. The choice of a given scheme is transparent for the inner implementation of the components. The reusability of software components is therefore significantly eased and, thus, applications can often be composed of prefabricated bricks.

The overall scheduling framework was already presented in [18]. So far we implemented FCFS- (First Come First Served), FP- (Fixed Priority), EDF- (Earliest Deadline First) and TT-scheduling (Time Triggered). The FP- and EDF-scheduler exist as preemptive and non-preemptive versions.

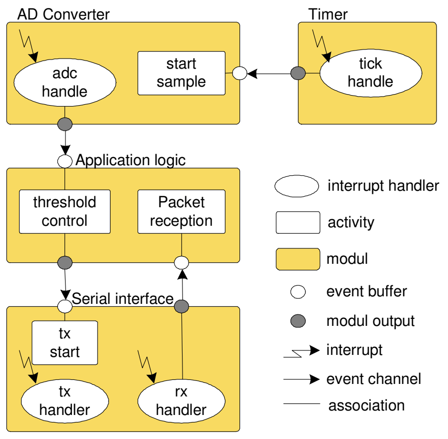

Figure 2. Event Flow Scheme of REFLEX

4.2 Preemptive Stack Sharing

Baker proposed in [1] the concept of preemptive stack sharing for run-to-completion tasks in real time systems. The key idea was that a task only consumes stack space if it is running, because it returns to the system after execution. If even a LIFO style scheduling policy is used, the tasks can share a single stack. LIFO means in this context that the last started task finishes first. Baker noted also that the number of priority levels determines the stacking depth. This is an important feature for severely memory constrained systems.

REFLEX uses stack sharing for preemptive priority based scheduling. There is an order of disruptions where an activity is able to interrupt another. The interrupted activity cannot interrupt the interrupting activity again. On top of the stack is always the activity with the highest priority at a time. This is the LIFO principle mentioned by Baker. Figure 3 shows a snapshot of a running REFLEX application with the mapping of scheduled activities (ReadyList) to the related stack frames.

The running activity can be interrupted either by an interrupt or by a software scheduling request. If a scheduled activity is of a higher priority than the running one, the running activity is interrupted and the scheduled one is started. If the scheduled activity has a priority lower than or equal to that of the running one, it is only inserted into the ready list. To activate these objects when they are in the front of the ready list, a schedule frame is needed for each started activity. Since an activity cannot be running twice at a time and the count of activities is typically low in the applications, it is possible to determine an upper bound for stack memory

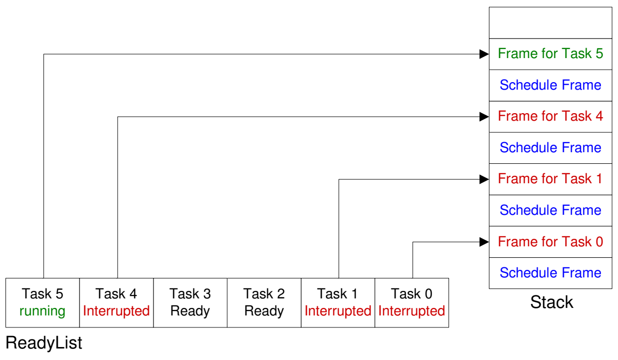

Figure 3. Preemption with one stack

consumption. Furthermore stack space is only required by already started activities.

Figure 3 shows also where the schedule frames are located on the stack. Those frames are created by the first interrupt in a row or by a scheduling request. In such a scheduling frame the call frame for the dispatch method of the scheduler is stored. If scheduling was initiated by an interrupt handler, volatile registers are also stored in the scheduling frame.

Figure 4 compares the stack usage in multi threaded and single threaded systems. In the single threaded approach only one stack needs to be allocated, and redundant allocations e.g. for interrupt handling frames can be avoided. Furthermore, the overall stack depth depends heavily on the applied number of priority levels. In most cases only a few will be needed, thus the overall stack consumption is significantly lower than in threaded systems.

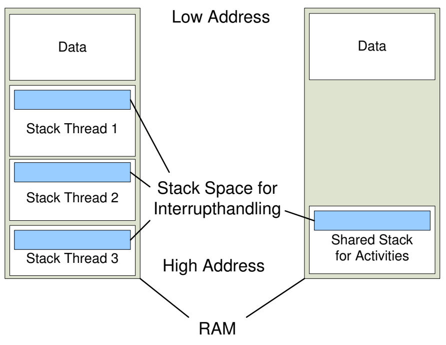

Figure 4. Stack consumption in multi- and single-threaded systems

For non-preemptive systems the same single-stack approach is used. But in this case only one task occupies the stack at a time and there is only one schedule frame at the bottom.

The single-stack approach saves much RAM, since stacks do not need to be preallocated for activities. They only occupy stack space during execution.

5 Evaluation

There are several aspects of a system, which are usually affected when introducing preemption. In this section we evaluate the effects of stack sharing to event driven processing. We also compare the results with those of the threaded solutions.

5.1 Building Applications

One of the predestined fields of application for preemptive stack sharing is compression and aggregation of network packets. In figure 5 an example application is shown. It is based on a network stack, which aggregates and compresses packets before these are given to the medium access (MAC) layer. The application and network activities typically have a short runtime, while compression and aggregation take a long time. Note that especially in the network component activities exist which must meet real time constraints, e.g. a time based medium access mechanism.

In our experiments we evaluated several implementations of the compression module, e.g. run length encoding (RLE), Lev Zimpel encoding by Welch (LZW) and Huffman encoding. These greatly differ in their complexity and therefore in their runtime. Nevertheless the encoding code is unaware of its environment and the application code is unaware of the currently used compression scheme. The only place in the code where the information is needed is the global configuration, which is the `NodeConfiguration` C++ object in REFLEX. Listing 1 shows the `NodeConfiguration` implementation for the Huffman encoding version.

The top level sensor node application configuration is a class, whose member variables are the components. These are connected in the constructor of the

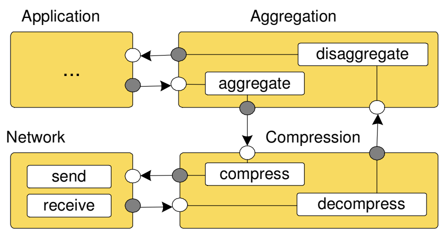

Figure 5. Example application which uses packet compression

NodeConfiguration, where also the minimum priority is assigned to the long running compression component.

Listing 1. "Top Level Application Configuration"

```
class NodeConfiguration {
public:
 NodeConfiguration()
 {
  //set connections to network
  app.lowerOut.connectTo(agg.upperIn);
  agg.lowerOut.connectTo(henc.upperIn);
  henc.lowerOut.connectTo(net.upperIn);

  //set connections from network
  net.upperOut.connectTo(henc.lowerIn);
  henc.upperOut.connectTo(agg.lowerIn);
  agg.upperOut.connectTo(app.lowerIn);

  //set priorities
  henc.setPriority(MIN_PRIORITY);
 }

 Application app;
 PacketAggregator agg;
 HuffmanEncoder henc;
 Network net;
};
```

All other activities get a standard priority which is in the middle of all possible priorities. Note that it is trivial to use a non-preemptive scheduling scheme without any modifications to the components, only the NodeConfiguration needs to be updated. This is for example a good option for RLE encoding, because its runtime is much shorter than that of Huffmann encoding. With a simpler scheduling scheme the system runtime is reduced and therefore power consumption is lower.

5.2 Concurrency

One could argue that in the proposed scheme concurrency of long running activities is limited. Activities cannot run fully interleaved like in threaded systems, but due to the run-to-completion semantics this is not needed.

Suppose a traditional threaded system, which has to execute two long running processes. Since the processes feature an infinite loop in that they operate on available data, the system allocates each process a timeslice and would then preempt it at the end of the slot. In event driven systems activities resume after processing a data set. Therefore interleaving is not needed, the long running activity which has the tightest deadline of all long running activities in an application is given the highest priority among them. This activity can now only be preempted by short running activities. A positive side effect is, that unnecessary context switches are avoided implicitly.

Table 1. Memory Consumption of Scheduling Schemes in Reflex on Texas Instruments MSP430 in Bytes

	System		Activity
	Rom	RAM	RAM
FCFS	2214	78	7
Time Triggered	2124	80	7+usedSlots*6
FP - non-preemptive	2252	78	8
FP - preemptive, 1 list	2380	78	8
FP - preemptive, 2 lists	2512	80	8
EDF - non-preemptive	2310	78	16
EDF - preemptive, 1 list	2428	78	16
EDF - preemptive, 2 lists	2568	80	16

5.3 Memory Consumption

Preemption is not for free in terms of memory consumption. However, table 1 shows that the memory overhead is nearly negligible in REFLEX. The table shows the code size, and the RAM consumption of the base system for the provided scheduling schemes of REFLEX. Additionally, the corresponding RAM overhead per activity is shown.

The overall system size (including scheduling, interrupt handling and power management) is even for the most complex earliest deadline first (EDF) implementation below 3KB. This is a comparable memory consumption to that of TinyOS and comparable to Contiki's memory footprint. Therefore REFLEX is well suited for use in sensor networks.

Interestingly, preemptive schemes have no significant impact with respect to RAM consumption per activity. The overhead is caused by the scheduling principle (Fixed Priority (FP) or Earliest Deadline First), if it is implemented preemptively does not matter.

5.4 Runtime Overhead

The runtime overhead for the preemptive scheduling scheme is not negligible like the memory overhead. Table 2 shows the ticks needed for the scheduling of an activity in different schemes on a Freescale HC(S)12. The microcontroller uses a CISC architecture. We used this platform for some real time applications.

The non-preemptive scheduler versions only sort the activities into their ready-list. Therefore these are faster

Table 2. Clock ticks needed for scheduling of an activity on Freescale HC(S)12

FCFS	48
Time Triggered	11
FP - non-preemptive	89
FP - preemptive, 1 list	213
FP - preemptive, 2 lists	346
EDF - non-preemptive	247
EDF - preemptive, 1 list	445
EDF - preemptive, 2 lists	571

than the preemptive ones. The overhead of the preemptive schemes is mainly caused by initializing and calling a dispatch routine at the end of scheduling, i.e. the instantiation of the schedule frames on the stack. The most complex versions of FP and EDF scheduling (2 lists versions) guarantee constant interrupt blocking times.

Additionally, the values for the other implemented non-preemptive scheduling schemes are shown. These are faster because there is no expensive sorting of activities. For TT-scheduling the schedule is done a priori.

Overall the runtime overhead cannot be neglected since the typical runtime of a task is relatively short. We measured activity runtimes from 70 to 30000 ticks in our applications so far. Thus the scheduling overhead is significantly higher than in PDAs, PCs or Highend Servers and must be considered when choosing a scheduling scheme.

Nevertheless, the time a high priority activity is blocked by lower priority activities can be significantly lowered with preemptive scheduling. It is even trivial to switch between preemptive and non-preemptive scheduling since activity code is not affected. This makes it possible to choose the best scheduling scheme at at system configuration.

5.5 Synchronization

The possibility of implicit synchronization is one of the strengths of event driven programming. In the event flow model an application is implicitely synchronized by the atomic but non blocking access to the event flow buffers. Therefore the communication between components is already synchronized even in the case of preemption.

Inside a component the preemptive operation could lead to synchronization problems. This can be solved by handling a component as a monitor. The interrupts for a component and higher priority activities of this component can be locked, when an activity of the component is executed. Note, that this form of locking affects only the scope of a single component and does not raise inter component dependencies. Thus also the violation of the monitor semantics of a component can be accepted if this is needed for performance reasons. Due to the synchronization scope the component programmer knows all the code, which affects synchronization and is able to synchronize the component by hand.

Precedence constraints are also a synchronization problem in threaded systems. The scheduler cannot decide, if a thread runs into a blocking condition immediately after it is started. In our event driven system an activity is only executed, when its inputs are valid. This behaviour is not affected by preemption.

5.6 Energy Consumption

An event driven system is only active, when an event occurs. Since all activities are triggered by events, the system can switch to a sleep mode if no activity is pending for execution. With the presented approach this scheme is not violated because long running tasks have run-to-completion semantics like all other tasks. Thus the stack sharing solution allows to implement an implicit power management scheme.

6 Conclusion

This paper introduced the stack sharing mechanism as solution to one of the remaining problems with event driven programming, namely handling long running tasks. The presented approach is different compared to multi threaded approaches, since it keeps all the benefits of event driven programming. It introduces no precedence problems or similar to scheduling. Furthermore, the approach is more lightweight than a thread based system due to the use of only one stack.

In the future we will do some comparisons with other schemes by implementing applications on the same plattform. This will yield a quantitative comparison in means of runtime and memory consumption.

References

[1] T. P. Baker. Stack-based scheduling for realtime processes. *Real-Time Syst.*, 3(1):67–99, 1991.

[2] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han. Mantis os: an embedded multithreaded operating system for wireless micro sensor platforms. *Mob. Netw. Appl.*, 10(4):563–579, 2005.

[3] 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. In *In Proc. of the 6th intl. conf.on Information processing in sensor networks (IPSN '07)*, Cambridge (USA), 2007.

[4] E. Cheong, J. Liebman, J. Liu, and F. Zhao. Tinygals: A programming model for event-driven embedded systems. In *SAC*, pages 698–704, 2003.

[5] C. Duffy, U. Roedig, J. Herbert, and C. J. Sreenan. Adding preemption to tinyos. In *EmNets '07: Proceedings of the 4th workshop on Embedded networked sensors*, pages 88–92, New York, NY, USA, 2007. ACM.

[6] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *In Proceedings of the First IEEE Workshop on Embedded Networked Sensors*, 2004.

[7] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, Boulder, 2006.

[8] R. H. E. Trumpler. A systematic framework for evolving tinyos. In *In Proc. of IEEE Workshop on Embedded Networked Sensors (EmNets) 2006*, 2006.

[9] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System Architecture Directions for Networked Sensors. In *the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 93–104, Cambridge, MA, November 12–15 2000.

[10] http://www.barello.net/avrx.

[11] http://www.freertos.org/.

[12] K.Walther and J.Nolte. Event-flow and synchronization in single threaded systems. In *First GI/ITG Workshop on Non-Functional Properties of Embedded Systems (NFPES)*, 2006.

[13] K. Langendoen, A. Baggio, and O. Visser. Murphy loves potatoes: Experiences from a pilot sensor network deployment in precision agriculture. In *14th Int. Workshop on Parallel and Distributed Real-Time Systems (WPDRTS)*, apr 2006.

[14] H. C. Lauer and R. M. Needham. On the duality of operating system structures. In *In Proc. of 2nd International Symposium on Operating Systems*, 1978.

[15] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless sensor networks for habitat monitoring. In *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 88–97. ACM Press, 2002.

[16] J. Ousterhout. Why threads are a bad idea (for most purposes), January 1996.

[17] D. B. Stewart, D. E. Schmitz, and P. Khosla. The chimera ii real-time operating system for advanced sensor-based control applications. *IEEE Trans. on Systems, Man, and Cybernetics*, 22(6):1282–1295, December 1992.

[18] K. Walther and J. Nolte. A flexible scheduling framework for deeply embedded systems. In *In Proc. of 4th IEEE International Symposium on Embedded Computing*, 2007.