A Scalable Hardware Algorithm for Demanding Timer Management in Network Systems

K. Septinus[1], S. Dragone[2], M. Langner[1], H. Blume[1], and P. Pirsch[1]

[1] Institute of Microelectronic Systems, Appelstr. 4, D-30167 Hannover, Germany
{Septinus, Langner, Blume, Pirsch}@ims.uni-hannover.de
[2] IBM Research GmbH, Zurich Research Laboratory, CH-8803 Rüschlikon, Switzerland {SID}@zurich.ibm.com

Abstract. This paper presents a dedicated parallel hardware architecture for timer management in demanding network processing systems. The proposed architecture is based on three main concepts: (1) usage of a hierarchical, scalable non-redundant ordering scheme for timer entries, (2) pipelined processing approach and (3) a particular cache for optimization of off-chip memory transfers. In contrast to existing algorithms, the proposed timer manager provides a very high performance and a high precision over a range of applications.
Based on a cycle-true SystemC model, timer entry insert rates of up to a new event every few cycles were simulated referring to highest packet rates in a network system. Thereby, timer expiration delays of less than tens of cycles could be measured for a system with 4000 entries. In summary, this contribution proposes a dedicated high-performance architecture extending state-of-the-art parallel timer management algorithms.

1 Introduction

Network applications have continuously advanced. Huge server machines automatically synchronize data over networks and provide a high performance in terms of data processing, data throughput and data storage. In this general context, several connection-oriented protocols include specifications of time-outs in order to automatically invoke specific actions and efficiently use the network bandwidth. The most common examples are re-transmit timers that trigger repeated transmitting of non-acknowledged data. However, if the data rates increase and more data streams are transmitted in parallel the requirements for handling time-outs also increase. This seems to be particularly true, in e.g. future servers or clustering applications in high-performance environments. Consequently, in such environments SW-based timer management becomes insufficient and a central hardware unit only for this purpose is a promising alternative.

As each connection refers to particular timer functions these function can be handled as *timer entries* in the system. Here, relevant system parameters include the number N of timer entries that are concurrently active at a time and the time-out period and unique identifier associated with a particular timer function of a particular connection. Formally, such a timer entry can be specified

as a tuple $t = [ID, T]$, whereas ID is a unique identifier and T the time-out period. Communication protocols specify the maximal acceptable time difference ΔT between the actual expire-time and the invoked action, e.g. the re-transmit. This delay time ΔT typically ranges from $-0\,\%$ to $100\,\%$, relative to the time-out period T.

Other application-specific parameters are the timer entry insert rate, which shall be denoted by i, and the relatively number c of timers deleted before the expire-time. In most systems, c can be assumed to be above $80\,\%$ or $90\,\%$. However, as T and i are high enough, the value of N must also increase. This case can also be illustrated by the assumption multiple thousand concurrent connections, whereas an update of a timer entry is required for most incoming packets. Then, *deleting*, *inserting* and generally *storing* of timer entries practically becomes an issue. As a consequence, a architecture concept is required that provides a sufficient performance for a scalable number of timers. Without loss of generality, N is assumed around 4000 in this paper.

Since typically timer entries are concatenated together and imprecisely handled by software and this is sufficient for today's systems. So state-of-the-art network processor units do not necessarily require processing thousands of concurrent timer functions. Instead they rather rely on straight-forward schemes. However, in order to address the general issue of handling of thousands of timer entries in parallel at demanding precision constraints - our contribution is the presentation of a dedicated high-performance timer manager algorithm extending previous works. Thereby, an efficient hardware design is proposed as depicted in Fig. 1 that provides a performance beyond current requirements, using on-chip and off-chip memory resources.

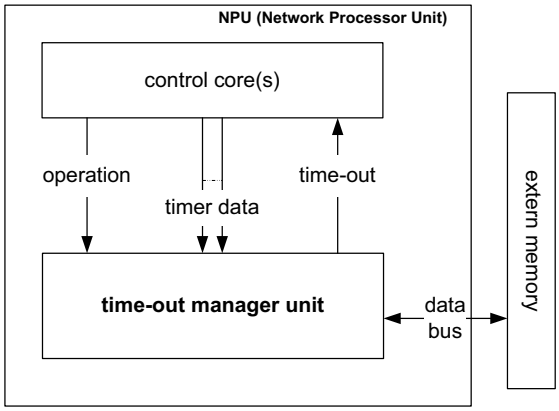

Fig. 1. Timer Manager Unit. *The interface particularly provides 2 types of operations: insert and delete of timer entries. Furthermore, expired timers are signaled to the cores by using an ID associated with the expired timer event.*

This paper is organized as follows: Section 2 summarizes existing approaches including straight-forward schemes to handle timer functions. Section 3 introduces the proposal. Results and design evaluation are presented in Section 4. Finally, conclusions from this work are drawn in Section 5.

2 Related Work

Since most of the today's programmable systems do not have a dedicated hardware timer manager, the implementation is done in pure software. Varghese presented an efficient data structure for a software implementation in [1], which is widely-used e.g. in Linux systems. The data structure is based on a hashed and hierarchical list (*timing wheel*). Such a wheel "rotates" with a fixed frequency. To support several different speed granularities, multiple of such wheels are maintained. Experiences in high-performance network endpoints have shown that typically the timer manager requires a dedicated processor from the multiprocessor pool [2]. Therefore, providers of network-endpoint solutions came up with dedicated hardware coprocessors [2].

A straightforward solution for a hardware timer manager would be an array of programmable decrementers. A decrementer is a register of width w that decrements at the same rate that the time base increments. When a non-zero time value is written to the decrementer, it begins to decrement. A timeout is signaled when a decrement occurs on a decrementer count of 1. This is a very expensive implementation area-wise for a large scale timer facility. Therefore, it is not an applicable solution.

Heddes described in his PhD thesis [3] a hardware solution based on a hierarchical list. The hierarchy is done by rounding the timing interval of each new started timer to a fixed duration. In a CAM each entry represents then the head of the list for the timers with the same duration. Thus, every new timer just has to be linked to the end of such a duration list. The entries of the CAM are then continuously compared with the time base, and a match marks the expiration of a specific timer. The disadvantage of this approach is that if the duration of the timers for a certain application is for all about the same, silicon area is wasted because of the unused CAM entries, independent of the number of allocated timers.

Linked lists also have the drawback that the complexity of the insert and the *delete*-operation increase linearly by the number of entries resulting in a complexity of $O(N)$. Thus, other approaches used different data structures to sort their entries. The most popular data structure is the heap structure [4–6]. The operation complexity on the heap structure is logarithmic $O(\log N)$. Furthermore, the heap structure also has the advantage that it is very efficient in the allocated memory, as it does not need any pointers and only a limited number of comparators are needed. Björkman organized all timer entries in a single heap structure [4], whereas Dragone et al. structured the heap in multiple levels which they split into on-chip memory hierarchies and external memory [5]. Essentially, the problem of sorting the heap with L levels whose nodes have C

children each, is broken down to sorting one of the $C^{L-2} \cdot (1 + C)$ heaps present in each level L.

3 Design Approach

This section presents the basic concepts for the timer manager hardware algorithm. Thereby, it is worth pointing out that a **parallel processing** of timers becomes feasible because timers are additionally grouped and stored using independent layers. Finally, design realization concepts, simulation set-up and some trade-offs are sketched. Because the idea of behind the algorithm was the development of a dedicated hardware unit, we rather emphasize on practical implementation instead of formal specifications.

3.1 Timer Management Algorithm

Like previous works [4, 5], the timer entries are supposed to be organized using a d-heap DATA STRUCTURE. The structure is prioritized by the minimum, thus each item is at least as large as its parent. In contrast to a standard d-heap structure where each root key holds d children, the number of the next higher layer is not fixed, but configurable. In this work we refer to different horizontal layers L1, L2 and L3 and additionally introduce vertical layers. We refer the L2 nodes as *areas* consisting of v entries. The L2 nodes are called *blocks* and h *blocks*, which belong to the same L2 node, are again outlined as *area*. Thus, the L3 level consists of $v \times h$ *blocks*. Each *block* is assumed to consist of e timer entries. The total number of addressable timer entries is $N = v \times h \times e$.

The allocated memories for the first and the second level are on-chip and accessible in parallel. For timer entries in the last level, a particular on-chip CACHE will be introduced later in this paper. This cache is supposed to be organized using *blocks* of timer entries. The rest of the L3 entries are stored off-chip. In the corresponding memory, timer entries are stored on a *block* basis. Figure 2 shows a visualization of the proposed organization.

As every timer entry should be found in short time (referring to a probably high deletion rate c as described in Sect. 1), the ID for the timer entry is associated with a particular *area*. This issue was solved in the work of Dragone et al. [5] by introducing a so-called "hardware ID" that corresponds to the physical storage location of a timer entry in the external memory. In this work, the first $\lceil \log_2 v \rceil$ bits of the ID always indicate in which of the *areas* the timer entry is stored. Then, an internal table is used store the *block* location and no other external handling of ID is required.

The concept of further parallalizing the time-out processing (additional to the de-coupling of the horizontal layers) lead to a PIPELINE. This is also motivated by the fact that processing of timers requires equal types of operation, regardless of the *area* the timer is mapped to.

Opposed to [5], each timer entry is only stored once in the whole system. This indicates that every time a new entry is stored in the unit, the proper location is

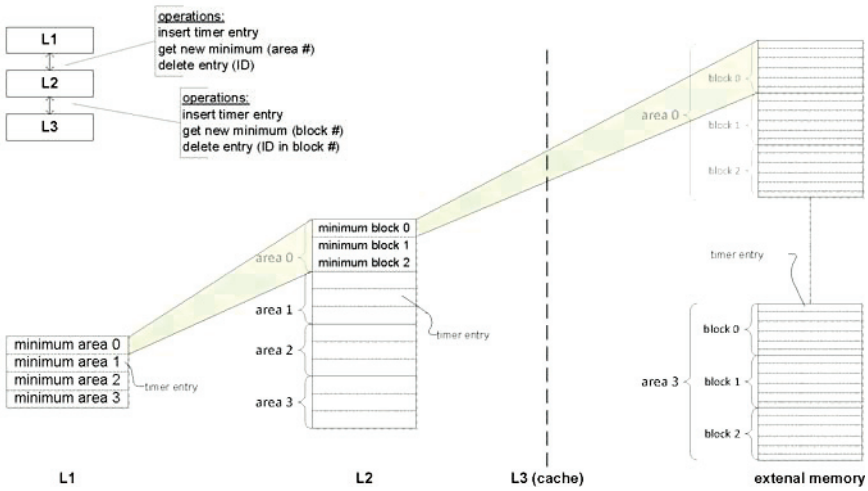

Fig. 2. Timer Entry Organization. *Without loss of generality, this diagram shows a configuration with* 3 *layer L1, L2 and L3, whereas the entries are mapped into* $v = 4$ areas *and* $h = 3$ blocks *per* area, *whereas* $e = 6$ *timer entries are stored in a single* block. *Here, the maximum number of addressable entries* N *would be* $N = 4{\times}3{\times}6 = 72$, *which are stored* non-redundant *(only once) in the system.*

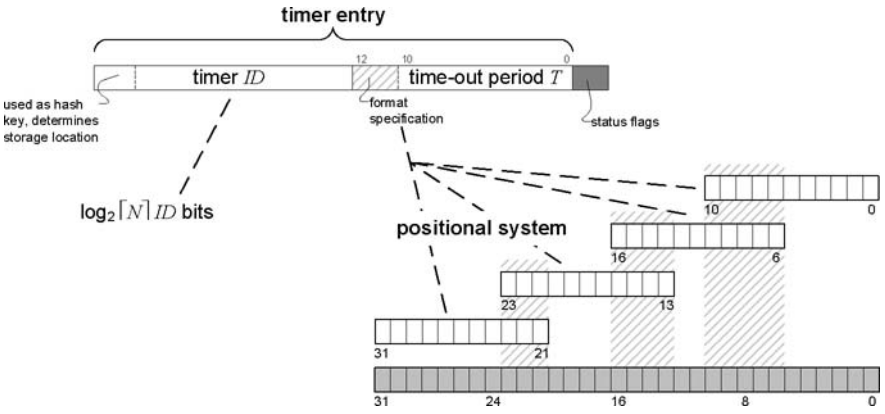

Fig. 3. Storing of timer Entries.

determined based on the time-out period T, i.e. timer entries generally are sorted based on the respective values for T. A search can then start from L1 going to L2 and so on. This also reflects that a timer does not have to be inserted into the external memory in the first place and any operation can possibly end in L1. In other words, the pipeline structure also allows streamline-processing of time-out events, going from the top to the bottom layer. So if the total number of timers is relatively small, again, less off-chip memory transfers are required. This allows additional speed-up.

Assume an operation requires approximately 100 cycles as cited in previous works, then, using the pipeline-based organization and parallel comparators indicated that a new operation can start much earlier and an effective cycle count can be significantly decreased. In general, the *expire*-operation profits the most of the parallelization. As in every layer L the minimum can be determined in constant time $t_{min} = 1$ and simultaneous in all the layers, the complexity of the *expire*-operation becomes $O(1)$.

The overall performance of the unit is most likely limited by the achievable bandwidth to the external memories, which is the motivation for minimizing the number of bits associated with one timer entry. What in turn also leads to a decrease of the chip area requirements, because a high number of these entries are stored locally. This work also assumes that the timer manager unit uses a central counter (like a time-of-the-day clock in a CPU) that is continuously incremented and cannot be paused. Consequently, the difference between a specified period T and the counter has to be evaluated as well as negative values (for overrun reasons). As shown in Fig. 3, for unit capable of processing N time-outs, at least $\lceil \log_2 N \rceil$ bits are needed in order to identify different timer entries by their ID.

In order to minimize the bits that specify the timer period T, different bit portions can be used, as similarly introduced by Heddes in [3]. Figure 3 e.g. shows 4 formats, whereas 2 bits of the entry identify the specific format. Some additional bits are required for status flags (overrun etc.), the actual timer runtime depends on the clock period and the value of T. Note that the loss of accuracy at this point can be estimated in before-hand. The cores that utilize the timer manager unit are supposed to provide all timer entry information or there is another unit deployed for these pre-processing steps.

3.2 Hardware Design Considerations

L1 and L2 are consequently organized as multi-staged pipelines. Thereby, a scheduler in front of each pipeline assures that parallel operations refer to different *area*. In order to achieve a maximum of performance, each level introduces multiple comparators in order to identify the first and second minimum of the stored timer entries in the particular layer. The second minimum is required if the first minimum is deleted or pushed to the higher layer, respectively. In the hardware implementation, L2 and L3 are tightly coupled, because the (*block* read and *block* write) cache and address table functions. A central FSM controls cache updates and schedules external data transfers optimizing also the scheduling order. Figure 4 sketches the hardware architecture. Thereby, two tables are

required in order to implement the timer management algorithm: the *storage manager* and the *link table*. The first table tracks free storage location in available memory blocks. The second table includes the mapping of IDs into *blocks* such that the unit can track where to load a particular entry from. Figure 5 shows the basic pipeline operation concept.

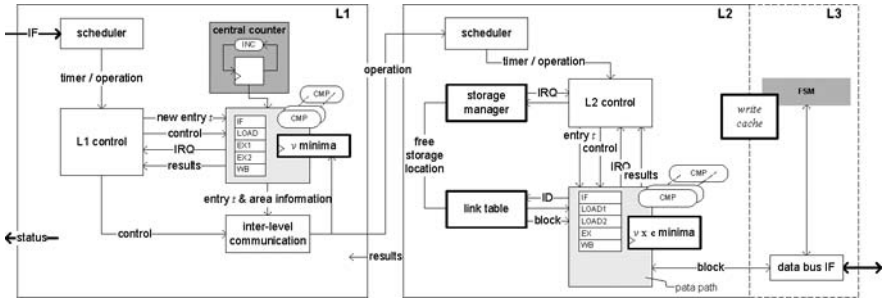

Fig. 4. Internal Organization. *The architecture basically works like a pipeline, inserting pushing new entries from top to bottom and let pop up potential expiring timers from bottom to top in parallel. According to the commonly used notation in microprocessor design, the different stages are labeled with "IF" (instruction fetch), " LOAD" (instruction load), "EX" (execute) and "WB" (write back).*

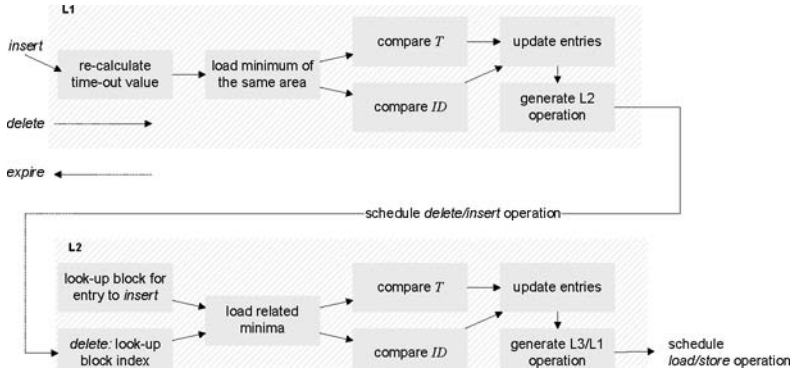

Fig. 5. Parallel Pipeline Operation Concept.

Mainly, all these features were investigated in order to maximize the performance and extend the range of applications, i.e. for the purpose to have a system that can handle a small number of timers entries meeting strict precision

requirements as like as a large number of entries using the external memory under less strict precision requirements. This focus is obviously different from works like [5], in which the authors proposed the heap structure mainly in order to optimize the timer searches with a minimal comparator network for specific Internet protocol applications.

Based on the cited concepts, finally, a cycle-true SystemC model and accurate simulation test-bench have been investigated. Except from the cache that had been implemented using high-level simulation code, all other modules were realized down to register-transfer level based interfaces in order to analyze potential hardware modules parallel to the simulation phase. Otherwise, one of the main issues was the development of a monitor system that could observe any delays in the signaling of expired timer events. The input data for the model consists of a text file with the timer operations *insert* and *delete*. Mainly synthetic trace have been used as basis for the studies, with respect to worst case scenarios.

Table 1 shows the estimation of hardware costs. Here, n refers to the total number of bits required per timer entry (e.g. 48 bits). Note that the number of bits per timer entry can be reduced as indicated. The hardware costs are significantly high, particularly if the *link table* is fully stored on-chip. However, the mapping from an ID into a hardware storage location would be handled by the unit, what simplifies the overall processing of timer functions. Another general trade-off is the cache size that also dominates the total hardware costs of the design.

		Resource
L1	1	central counter (time-of-the-day clock)
	v	n-bit timer entry register for the minima of each *area*
	$v+1$	comparators
	5	n-bit pipeline registers
L2	$v \times h$	n-bit registers for the minima of each *block*
	2	comparator networks in order to find the minima
	$N^* \times \lceil \log_2 h \rceil$	SRAM cells in the *link table*
	$m \times e \times n$	SRAM cells in the cache with m cached *blocks*
	1	memory interface logic and *storage manager*
	6	n-bit pipeline registers
L3	$N \times n$	RAM cells in the *external memory*

4 Results

Figure 6 shows the maximum delay ΔT_{max} for expired timers, measured in clock cycles. The x-axis refers to the average number of clock cycles between subsequent *insert*-operations, e.g. 65 means that a new timer entry is inserted every 65th cycle, here corresponding to a insert rate of $i = 1/65$.

Moreover, it was assumed that $c = 0.95$ holds, i.e. 95 % of the timers are deleted before the expiration time by *delete*-operations. The unit was configured for $N \approx 4000$ using $v = 8$ areas and $h = 16$ blocks per area, $e = 32$ timer entries per block. Different the lines in the diagram correspond to different sizes of the cache, ranging from 4 to 128 cached blocks. Respectively, the later case requires a much smaller number of accesses to the external memory, which generally is the system bottleneck. Using Fig. 6, one can see that there exists a particular insert period that cannot be decreased any further. This period depends on the system parameters and the maximum number of timer entries N that can be processed in parallel.

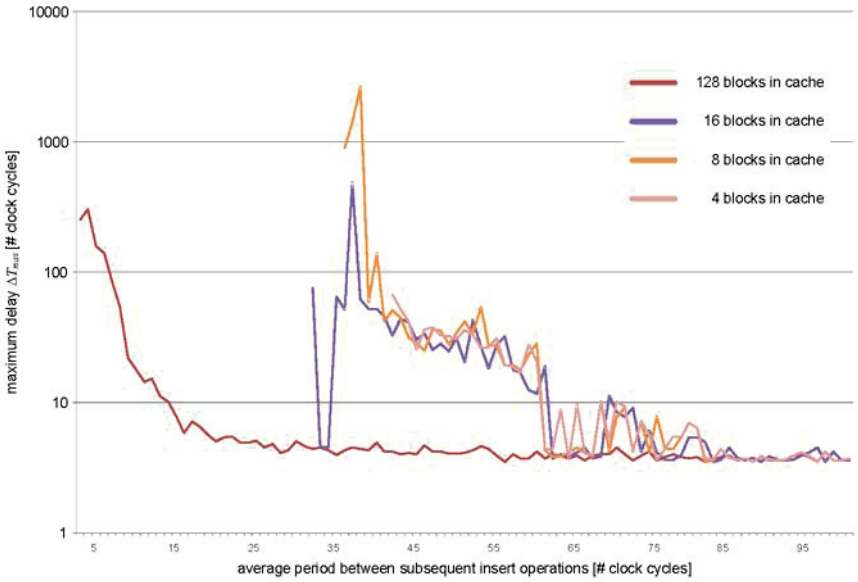

Fig. 6. Measured Delay Times.

Otherwise, these results show that the design is capable of processing timer entries at extremely high precision requirements (up to delays below tens to hundreds of clock cycles. The achievable delay times ΔT become obviously shorter for smaller insert rates i.

Based on our studies, rather general results can also be summarized: The introduced system allows around 5 to 10 times higher insert rates and a much higher precision than the previous work. Future research includes the analysis of how the number of vertical and horizontal layers (i.e. v, h and e) can be optimized for a given N. This design space exploration should be performed in respect to the chip area costs and potential application constraints and parameters.

5 Conclusion

We have presented a dedicated algorithm and parallel design for timer management for demanding network processing systems that enables timer entry processing within a few cycles, whereas thousands of timers can be handled in parallel. The key concepts applied can be summarized as follows: usage of a hierarchical, scalable non-redundant ordering scheme for the timer entries, pipelined processing approach and a particular cache for optimization of off-chip memory transfers. The proposed structure particularly supports to process time-out events in a pipelined manner. Compared to the e.g. proposed straight heap-based algorithms using a limited number of comparators, the processed architectures results in higher costs in terms of logic, registers and on-chip memory cells. This work elaborates the maximum time-out processing rates.

References

1. Varghese, G., Lauck, A.: " Hashed and Hierarchical Timing Wheels: Efficient Data Structures for Implementing a Timer Facility". In: IEEE/ACM Trans. Networking. (December 1997) 824–834
2. Chelsio Communications White Paper: The Terminator Architecture: The Case for a VLIW Processor vs. Multi-processor SOC to Terminate TCP and Process L5-L7 Protocols (2004)
3. Heddes, M.: A Hardware/Software Codesign Strategy for the Implementation of High-Speed Protocols. PhD thesis, Technical University of Eindhoven, The Netherlands (1995)
4. Björkman, M.: Designing hierarchical hardware for efficient timer handling. In: Proc. of the Second IEEE Workshop on Future Trends of Distributed Computing Systems. (1990) 149–152
5. Dragone, S., Döring, A., Haugenau, R.: A Large-Scale Hardware Timer Manager. In: Proc. of the ANCHOR Workshop. (June 2004) 24–29
6. Tsaimos, D.: UMTS Gi Interfacing and Measuring System. Master's thesis, Royal Institute of Technology (2006)