2011 Eighth IEEE International Conference on Mobile Ad-Hoc and Sensor Systems

Boreas: Efficient Synchronization for Scalable Emulation of Sensor Networks

Robert Sauter*, Richard Figura*, Olga Saukh†, and Pedro José Marrón*
*Universität Duisburg-Essen, Germany
†ETH Zürich, Switzerland
{robert.sauter,richard.figura,pjmarron}@uni-due.de, olga.saukh@tik.ee.ethz.ch

***Abstract*—Cycle-accurate emulation of sensor networks allows a detailed analysis of platform target code for development and evaluation. However, the high overhead incurred by providing the necessary fidelity limits the size of the emulated networks considerably. The use of multiple cores provided by modern hardware can significantly improve the speed of emulation but requires synchronization algorithms to preserve causality. Based on the well-known multithreaded event-driven emulator Avrora, we investigate a number of synchronization methods including an algorithm that does not require any locks to improve the performance. We show that both the speed and the scalability can be significantly improved without sacrificing correctness. Additionally, we evaluate the impact of modern CPU technologies such as simultaneous multithreading on emulation performance.**

***Keywords*-sensor networks; scalable emulation**

I. Introduction

Due to the high cost of real deployments, simulation and emulation are important tools for research and the development of wireless sensor network software. Cycle-accurate emulation of sensor networks allows the testing of platform target code including unmodified operating systems and device drivers. This high-fidelity simulation enables detailed analysis of the performance (e.g. energy consumption) of algorithms and systems and allows convenient debugging of the final system. However, this fidelity also incurs a significant performance overhead compared to more abstract simulation approaches. With the proliferation of multi-core systems and the slower pace in increasing single-core performance, parallel emulation is becoming more and more important. In this paper, we present the development of new synchronization algorithms for the popular emulator Avrora [1] that often increase the performance on multi-core systems by an order of magnitude.

Challenges The core challenge of parallel and distributed simulation is to preserve causality, i.e. no event impacting another node (e.g., a message) is allowed to arrive in the 'past'. This can be easily guaranteed in sequential emulation as, e.g. used by ATEMU [2], where all the nodes are processed round-robin one cycle at a time. However, when nodes are emulated in multiple threads, the execution order of simulated instructions with respect to global time is non-deterministic because threads are not guaranteed to run at exactly the same speed. Therefore, parallel and distributed simulation require the use of a synchronization algorithm that enforces causality. Before a cycle of a node is executed, the emulator has to check that all other nodes have advanced so far that it is guaranteed that they cannot influence the current cycle of the node.

Since the emulation of sensor nodes requires frequent synchronization and the synchronization algorithm often requires holding a global lock, one core challenge to enable scalable emulation is to reduce the number of synchronizations and the time the algorithm requires, especially the part when holding a lock

Contribution and roadmap In this paper we investigate the impact of synchronization algorithms and present two algorithms that increase the scalability considerably. We discuss the problem of synchronization in detail in Section II and related work in Section III. Our contribution in this paper is threefold.

In Section IV, we present a synchronization algorithm that significantly reduces the time holding the global lock and thereby decreases the synchronization overhead considerably. The algorithm exploits the special properties of the so-called *last node* – the node with the (currently) smallest clock value. We also discuss a number of smaller optimizations for this approach including more accurate computation of the lookahead.

In Section V, we show an algorithm that does not rely on locks at all but instead exploits the consistency guarantees of the Java virtual machine, which mirrors the guarantees of modern CPUs, and the properties of the synchronization problem to preserve causality.

Third in Section VI, we study the impact of the synchronization algorithm on the emulator performance and scalability. We investigate the influence with four applications that represent a wide spectrum of sensor network types and put different stress on the synchronization algorithm. We consider network sizes between 16 and 256 nodes and host environments between 1 and 8 cores. We show that the synchronization algorithm has a significant impact on both performance and scalability and that our approaches can increase speed by an order of magnitude.

978-0-7695-4469-4/11 $26.00 © 2011 IEEE
DOI 10.1109/MASS.2011.53

II. The Synchronization Problem

The core of an emulator consists of the execution engine interpreting the individual instructions of the software-under-test and an event simulation system that is responsible for the timely interruption of the normal program flow. Events usually stem from simulated interrupts of the microcontroller model, e.g. when a timer fires. While these node-local events are known in advance, external events, i.e. the reception of a packet from another emulated node, can occur at any time and depend on the emulated states of the other nodes. The task of the synchronization algorithm is to preserve causality. The execution must not advance until it is guaranteed that no external event can be generated any more that could influence the program flow of the past. This requires frequent synchronization with the other emulated nodes and examining their clocks.

One core parameter of this algorithm is the lookahead that is derived from to the minimum time difference (in simulated cycles) between the point in time when the simulator 'knows' that a node will influence another node and the point in time the influence takes effect. This time difference results from physical properties of the emulated hardware. For sensor network emulation, messages are the only way that one node can influence another. The lookahead depends, therefore, mostly on the state of the emulated radio. There is a hardware-induced delay between switching the radio to send mode and the time when the sending begins and, therefore, a receive event can occur on other nodes. There are a few more hardware modes that influence the lookahead. Foremost, while a node is sending it cannot be influenced by another node and, therefore, the emulation of this node can proceed until the sending is stopped. Additionally, the lookahead can increase when a microcontroller sleeps, which is especially relevant in sensor networks with their focus on low power operation and frequent uses of power conserving operation modes. When the emulation of a node calls the synchronization algorithm, it uses the times of the other nodes and the computed lookahead to determine if the emulation can proceed and how many cycles can be processed before the emulation of the node must synchronize again. A synchronization algorithm usually consists of the following steps.

Time Update: In this step, a node informs the synchronization algorithm about its current time to enable a global view over all the nodes. While the times of the other nodes may already be out-of-date, it is guaranteed that all nodes have progressed at least as far as currently known to the synchronization algorithm.

Wake Up Waiting Threads: In the second step, the synchronization algorithms uses the new time to check if any waiting node that has been blocked before can be woken up.

Block or Continue: In the final step, the algorithm determines if the emulation of the current node can proceed immediately or if the thread has to block until the emulation of other nodes advances.

A. Avrora Synchronizer

Avrora uses two fixed time offsets to decide if the emulation of a node can progress immediately. It computes a threshold time by subtracting the first offset from the current clock value. If the clocks of all other nodes have advanced beyond this value, the emulation of the node can continue immediately. If no message is scheduled to arrive, the next synchronization is scheduled using the second offset. The sum of these offsets equals the hardware-induced delay between switching the radio to send mode and the time the first byte is delivered from the receiving radio to the microcontroller to ensure causality. If a message transmission is underway, the synchronization algorithm computes the reception time and schedules the next synchronization accordingly.

To manage the blocking nodes, the algorithm uses a list of so-called wait links. This structure consists of a timestamp and a counter containing the number of nodes that have already progressed beyond this timestamp. If the emulation thread of a node has to block in the third step, it inserts a wait link with the computed threshold and the number of nodes that are already beyond this value and blocks. If a wait link with the same threshold already exists, it is reused. In the time update step, the synchronization algorithm uses the previous clock value and the new clock value to traverse this list and update the counters for the wait links. If the counter of a wait link reaches the total number of nodes, all its blocking threads are woken up. The complete algorithm requires the use of a global lock to guarantee consistency which, as we show later, limits the scalability considerably.

III. Related Work

ATEMU [2] was one of the first cycle-accurate emulators for wireless sensor networks. However, ATEMU does not support parallel emulation. VMNet [3] enhances ATEMU with support for host-networking but does not address the scalability issue.

There are three major approaches to increase simulation speed. First, parallel emulation uses a multithreaded approach to exploit multiple core or CPUs in one system. Our work is based on Avrora [1][4], a popular cycle-accurate emulator that supports parallel simulation using threads and contains a large number of functions to support debugging and evaluation of sensor node software. We only replace the synchronization mechanism that ensures causality without removing or constraining any functionality. PolarLite enhances the Avrora emulator with an optimization that takes the sleep times into account [5] and with support for using software state from the TinyOS 1.x MAC protocol at runtime to compute bigger lookaheads [6]. These mechanisms are

orthogonal to our approach and could be combined to achieve higher scalability.

A second approach uses distributed emulation to increase the performance. DiSenS [7] uses a cluster of hosts to emulate a network. However, the performance is strongly dependent on the topology as explained in the paper. LazySync [8] aims at reducing the number of clock updates in a distributed emulation. WorldSens [9] is another distributed emulator that uses an optimistic approach to increase efficiency.

A third approach to improve the speed of emulation is to increase the abstraction level. TOSSIM [10] provides source level simulation for TinyOS by replacing hardware dependent parts with simulation components. This allows simulating the final application code of TinyOS applications but does not provide the fidelity, analysis details and operating system independence of cycle-accurate emulation. TimeTossim [11] extends TOSSIM with accurate timing by instrumenting the code based on debugging information gained from compiling for the target platform. This approach is suitable if the replacement of part of the platform code by simulation code and the provided accuracy is adequate for a scenario. COOJA [12] provides cross-level simulation, i.e. the simulation of a network of nodes where the abstraction levels of the nodes can range from emulation, over source code simulation to nodes developed in Java. Emulation is accomplished by using MSPsim [13] and source code simulation is coupled to the Contiki operating system. This approach allows for example emulating only a part of a network while the other nodes are simulated. This increases the speed but also decreases the fidelity for the overall simulation. SenQ [14] and [15] focus on the realism of simulation by combining a TOSSIM-like approach with an established simulation platform to benefit from established physical layer models, battery models and clock drift.

IV. Boreas Synchronization

The goal of the Boreas Synchronization Algorithm is to reduce the amount of time holding the global lock. The primary concept of the algorithm is to reduce the problem of verifying that all nodes have progressed beyond a certain threshold to just comparing the threshold to the so-called *last node*, i.e. the node whose clock has the smallest value. This node, which of course changes constantly during the simulation, has two important properties. First, all blocking nodes wait at least for this node (possibly for more). Therefore, only if the time of the last node changes, any waiting node may be woken up. Second, a node has only to check its waiting threshold with the clock of the last node to decide if it has to block or not.

We use these two properties to reduce the time in the critical section. When a node synchronizes, we distinguish two cases. If the node is not the last node, it compares its waiting threshold with the clock of the last node. If the waiting threshold is higher, then the node has to block, otherwise it continues. This check consists of just two comparisons (to see if the node is the last one and to compare the threshold and the clock) that have to be made while holding the global lock. In the second case where the synchronizing node is the current last node, we have to perform the majority of the work. First, the synchronization algorithm computes the new last node and its time. Then, it compares the threshold of all waiting nodes with the new comparison time and wakes up all nodes where the threshold is below. Third, the algorithm compares the threshold of the calling node with the clock value of the newly selected last node to check if the node has to block or may continue.

A. Further Optimizations

In addition to this new concept of the synchronization algorithm, we investigate a number of changes with regard to their impact on scalability.

Exact Lookahead When computing the lookahead, Avrora distinguishes neighboring nodes based on if they are currently sending or not. When a node is sending, Avrora uses the current clock of this node to compute the lookahead and thus the next time for synchronization. However, in the common case that no neighboring node is currently sending, the lookahead is only statically set to a minimum guaranteed time instead of taking the exact clock values of the neighbors into account. When using exact lookahead computation, we take the time of the last node into account. This requires a slight overhead for keeping this up-to-date but potentially allows increasing the time till the next synchronization.

LPL Optimization As explained above, Avrora distinguishes only two modes of the radio (sending and receiving). However, the emulated hardware features for example a power off mode used by low power listening MAC protocols. This power mode requires a considerably longer time before a message can be sent. The gain by considering this mode depends heavily on the application, however. In numerous sensor network applications the CPU load is very low and, therefore, the microcontroller is usually powered down almost the same time as the radio. Avrora already exploits these sleep times to improve scalability so that only the emulation of applications with higher CPU load, e.g. due to signal processing, are expected to experience a performance improvement.

Clustered Synchronization We also investigate a hierarchical approach to synchronization where we divide the topology into a number of clusters. The synchronization algorithm is used as described above within a cluster. Additionally, the clusters share their clock – as defined by the last node within a cluster – with all neighboring clusters, to take them into account during synchronization. This eliminates the need for a global lock and instead introduces cluster-global locks.

V. Lockless Synchronization

Because the time spent in locks is the main limiting factor for the scalability of the emulator, we investigated how to minimize this time. In this section we present a synchronization algorithm that does not require any locks at all but instead uses the guarantees provided by the Java virtual machine specification [16] that are a perfect fit for the causality requirements of the emulator.

The Java VM specification guarantees that reads and writes to variables are atomic, e.g. it is not possible that only some bytes of a multi-byte integer variable are updated before another thread reads the value. This mirrors the hardware guarantees of current micro-processors, e.g. as detailed in [17].

However, the VM specification does not provide any guarantees with respect to the global order of accesses among threads to the whole variable. Thus, it is possible for a thread to read an out-of-date value if the write from another thread has not been propagated to main memory yet.

Our lockless synchronization algorithm uses these guarantees to allow directly accessing the clock variable of each thread, which contains the cycle count of the respective emulated node. This variable is updated by the corresponding thread after the emulation of each instruction. When a thread has to synchronize, it compares the values of these variables with its own. The following properties and requirements for causality combined with the guarantees are responsible for the correctness and performance of this solution. First, there is only one writer (combined with multiple readers) per variable. This renders locks between writers unnecessary. Second, because the emulation clock is monotonically increasing also the value of the variable is monotonically increasing. This guarantees that when an out-of-date value is read, it is always smaller than the actual value. Because smaller values can lead to unnecessary waits, i.e. the synchronizing thread assumes another thread is "too far behind", these can impact the performance but always preserve causality.

The algorithm itself is then very simple. If any of the clocks of the other nodes is smaller than the threshold, the thread calls *yield* to give other threads the possibility to run and, thereby, advance the clocks of their emulated nodes. When the thread gains control again, this loop continues – possibly yielding again – until all the other nodes have advanced beyond the synchronization threshold. Therefore, the algorithm does not require explicitly blocking or waking up threads.

VI. Evaluation

In this section, we present the evaluation of the different algorithms with respect to a number of varying factors. We run all tests on servers with two 2.8 GHz quad-core CPUs. The CPUs are capable of simultaneous multithreading presenting a total of 16 virtual cores to the operating system. The systems are equipped with 24GB of RAM and run a 64-bit Linux and a 64-Bit Java VM using 8 GB of RAM. Unless otherwise noted, we use the taskset utility to bind the Java process to a certain number of cores while avoiding to use two virtual cores served by the same physical one. We run each application for 180 simulated seconds and repeat each test 4 times.

A. Factors

We vary the following factors in our evaluation.

Algorithm We distinguish among 6 synchronization algorithms: the algorithm used by Avrora, the lockless algorithm, and 4 variants of the Last Node algorithm as detailed Section IV. For evaluating the clustered algorithm we divide the topology into 8 clusters, which equals the maximum number of physical cores.

Number of Cores We vary the number of cores among 1, 2, 4 and 8. The program is bound to one CPU when four or fewer cores are used. Additionally, we show the difference between using 4 cores of one physical CPU and 2+2 cores on two physical CPUs that, depending on the algorithm, is quite significant. We also evaluate the use of simultaneous multithreading by using all 16 virtual cores.

Applications We evaluate four different applications that represent a significant part of the software spectrum of wireless sensor applications with respect to the influence on emulation speed. As the foundation, we use the TinyOS example application MViz that periodically senses data and uses CTP [18] to forward them to a sink and, therefore, is a good representative for a typical sensor network application. MViz does not use low power listening by default. We evaluate a second variant that enables LPL ('MVizLPL') and one variant that enables LPL and additionally generates a high CPU load with computations lasting several seconds ('MVizLPLFFT'). Furthermore, we evaluate a tiny application that just turns the radio on ('RadioOn'). This is the worst case scenario for the synchronization algorithm because the lookahead is always minimal and thus synchronization and blocking of threads occurs very frequently.

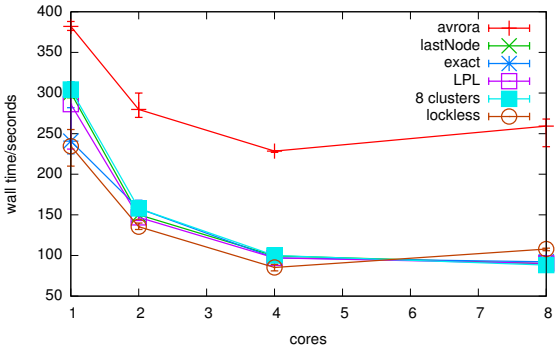
Number of simulated nodes We evaluate regular quadratic grid topologies with 16, 49, 144, 196 and 256 nodes. Additionally, we evaluate topologies with the same number of nodes where all nodes have the same position ('point').

B. Metrics

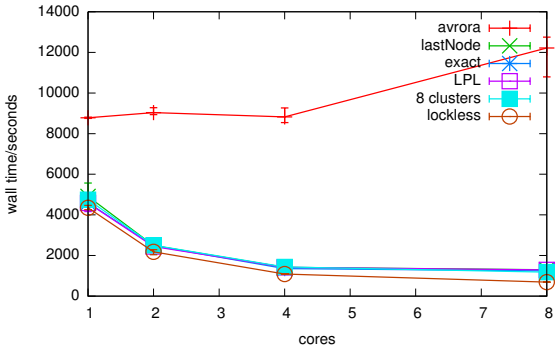
We collected the following core metrics for evaluating the performance of the different algorithms.

Wall Time: The wall time is the time the whole emulation requires and the most important metric to evaluate the algorithm for a real simulation problem.

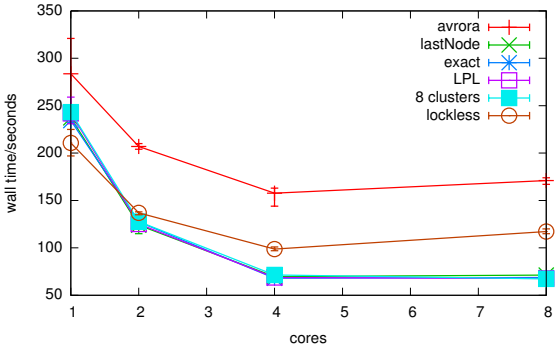
CPU Usage: This value is the average CPU usage during the simulation. Generally, higher values are a good indicator for high scalability and lower values indicate that the threads spend a lot of time waiting.



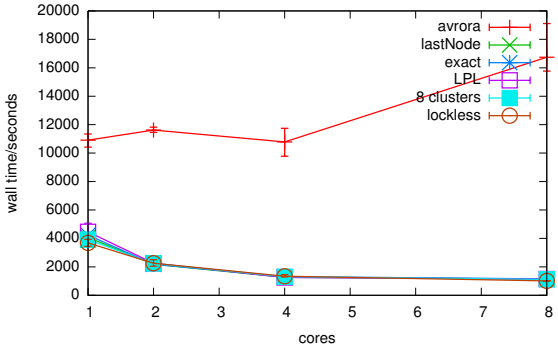
(a) MViz, 16 nodes



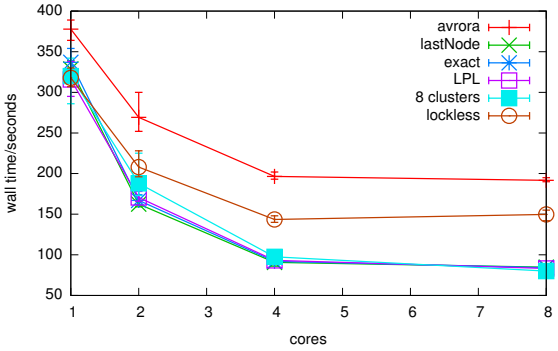
(b) MViz, 256 nodes



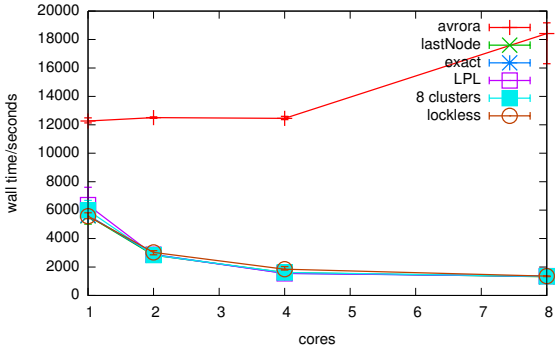
(c) MVizLPL, 16 nodes



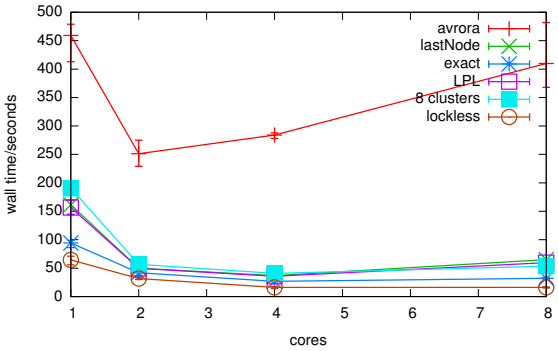
(d) MVizLPL, 256 nodes



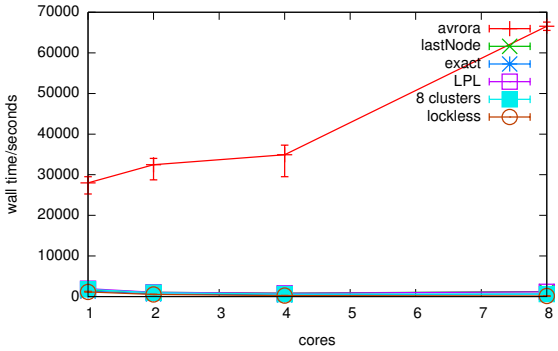
(e) MVizLPLFFT, 16 nodes



(f) MVizLPLFFT, 256 nodes

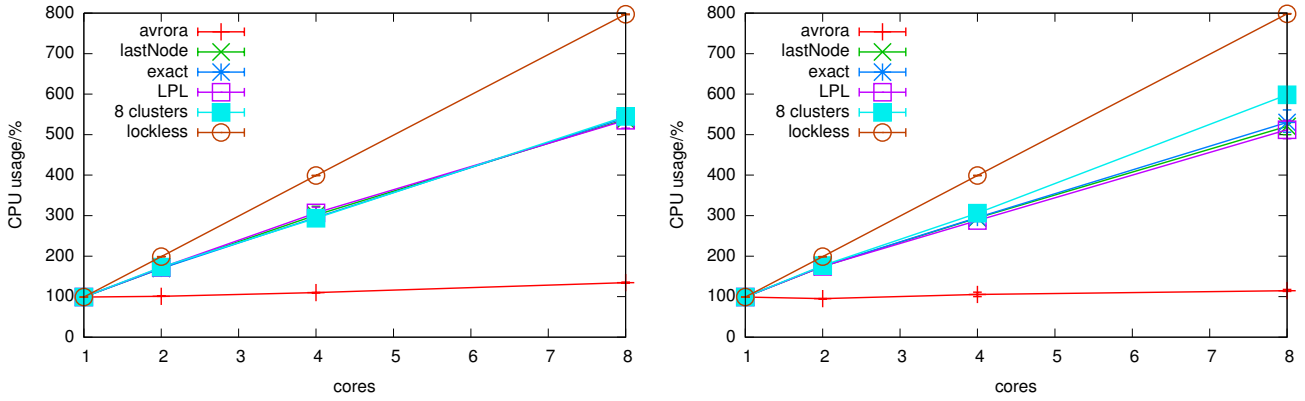


(g) RadioOn, 16 nodes



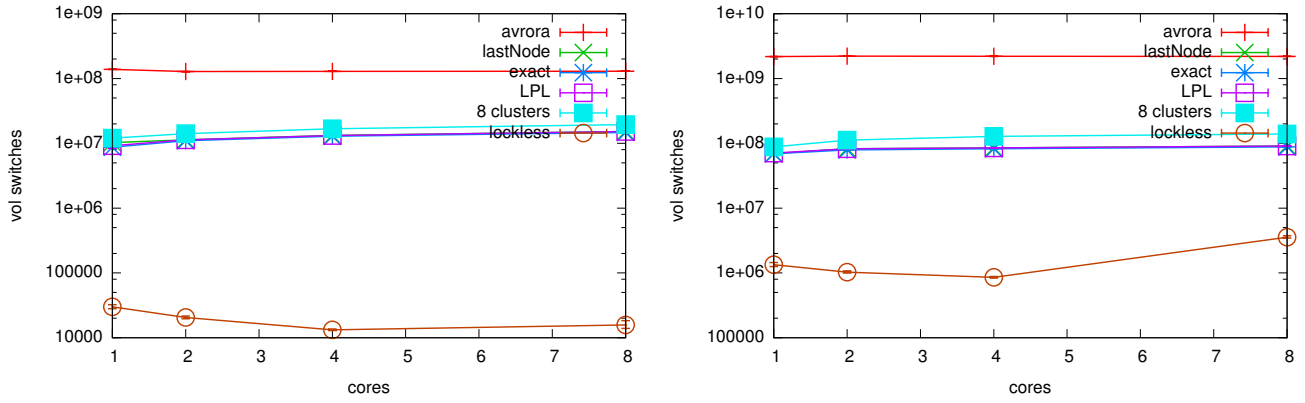
(h) RadioOn, 256 nodes

Figure 1. Wall times for different applications, 16 and 256 nodes



(a) 49 nodes (b) 256 nodes

Figure 2. CPU usage for MVizLPL



(a) 49 nodes (b) 256 nodes

Figure 3. Number of voluntary context switches for MVizLPL (logarithmic scale)

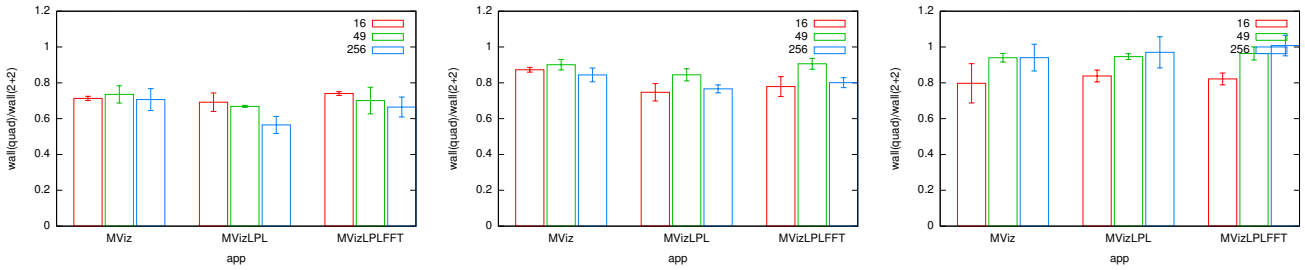
Speedup: Speedup is a metric for the parallelism of an algorithm and defined as $S_p : T_s/T_p$ where T_s is the time used by the sequential algorithm, p is the number of processors used and T_p is the time required for the parallel execution on p CPUs. Ideal or linear speedup is defined by $S_p = p$.

Involuntary/Voluntary Switches: We record the total number of context switches during the runtime of an emulation and differentiate between involuntary switches that occur when a thread uses its time slice and voluntary switches that occur for example when a thread blocks to wait for a lock or sleeps. A high ratio of involuntary switches to voluntary switches is a good indicator for a scalable program. However, this metric is less meaningful for evaluating the lockless algorithm because the operating system does not include the number of switches initiated by calling *yield* in either of these two counters.

C. Avrora Performance Comparison

In Fig. 1, we present the wall times of the algorithms in the different scenarios. We show the average, minimum and maximum wall times for 1, 2, 4 and 8 cores emulating 16 and 256 nodes.

First, the results show that the synchronization algorithm of Avrora performs worse than our approaches especially in scenarios with a large number of nodes or cores. Depending on the emulated application and the size of the network, Avrora performs even worse with increasing number of cores. While the emulation of networks with 16 nodes generally benefits from using 2 or 4 cores, 8 cores do not improve the performance for any test. The negative impact correlates strongly with the synchronization requirements of the setup: as described above, RadioOn followed by MViz have the highest frequencies of synchronization compared with the time spent for actually emulating instructions. The



(a) Avrora (b) Last Node (c) Lockless

Figure 4. Ratio of wall times for 4 cores on 1 CPU compared to 2+2 cores on 2 CPUs

synchronization overhead increases also with the number of nodes. Similar behavior of Avrora has been found in [6] and has been explained by the increase of CPU speed in the last years that reduces the time spent on emulating instructions. Meanwhile communication overhead between cores has not decreased correspondingly. Additionally, emulating larger networks has gained importance, which also puts more stress on the synchronization algorithm. The test of MVizLPLFFT with 16 nodes where the computational load of the nodes is relatively high and the on-time of the radio is relatively small supports these findings as this test shows the highest scalability of Avrora.

We present two indicators for the performance differences in Fig. 2 and 3. The average CPU usage presented in Fig. 2 show that the CPU usage of Avrora increases very slowly when adding cores to the emulation and thus Avrora does not benefit from the additional resources. It is important to note that the CPU usage generated by the lockless algorithm that almost fully uses all possible cores does not directly translate to a higher performance than the Last Node algorithm, because the lockless algorithm does not block at all. The use of yield is also the reason for the comparably low value of voluntary context switches depicted in Fig. 3 since the operating system does not include these calls in the counter. Avrora shows an order of magnitude higher number of voluntary context switches (i.e. incurred by blocking) than the Last Node variants. The reduced time holding a global lock of our algorithms reduces the synchronization overhead considerably.

D. Overhead of multiple CPUs

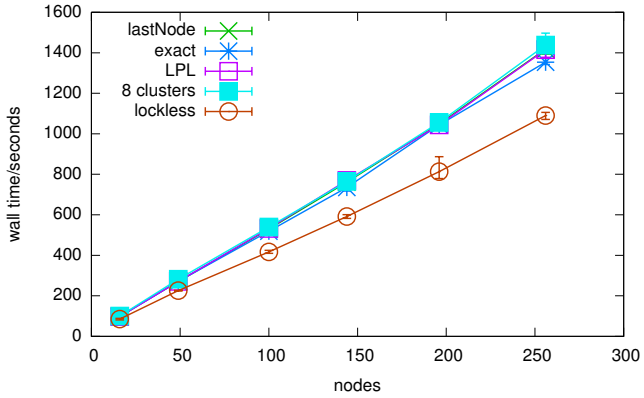
As shown in Fig. 1, depending on the scenario the results of using 8 cores are worse than using 4 cores. Therefore, we evaluate the overhead of using 2 CPUs by testing the use of 4 cores. In Fig. 4, we show the relative performance of running an emulation on 2+2 cores on 2 CPUs compared to using 4 cores of 1 CPU. We show the average ratios and their estimated standard deviations for Avrora, the basic Last Node algorithm and for the lockless algorithm. The overhead incurred by using 2 CPUs is significant for both Avrora and the Last Node algorithm. Avrora looses around 30% of its performance when two CPUs are used. This issue combined with the small gain by using more cores for Avrora as indicated by the CPU usage, is a major reason for the limited scalability and the bad performance when using 8 cores. While the Last Node algorithm also suffers from a noticeable overhead, this only leads to a worse performance in small networks with 16 nodes whereas the gain of using more cores outweighs the overhead in larger networks as shown in Fig. 1.

E. Boreas Algorithms

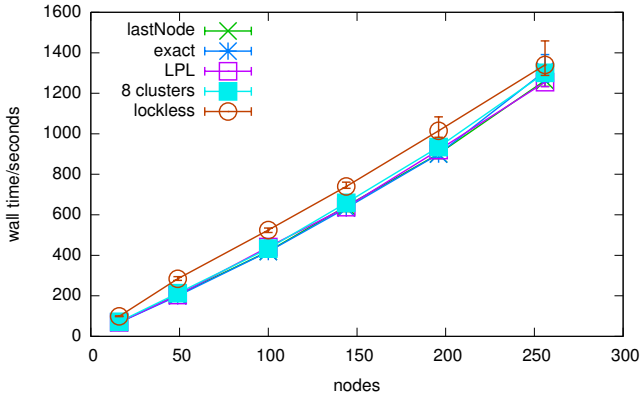
To compare the performance and scalability of the different Boreas Algorithms, we show in Fig. 5 the wall times when using 4 cores and when using 8 cores. There is a relatively clear distinction between the lockless algorithm and the variants of the Last Node algorithm. In the more synchronization heavy MViz test, the lockless algorithm consistently outperforms the Last Node algorithm with a more noticeable gain in larger networks. While there is no clear winner within the Last Node group when using 4 cores, the clustered algorithm has an advantage when using 8 cores. This highlights once more the overhead of using global locks. The results show that for MVizLPL the lockless algorithm generally performs worse that the Last Node algorithm. One exception is the largest network with 8 cores where the smaller overhead when using multiple CPUs leads to a small advantage for the lockless algorithm.

F. Point Topologies

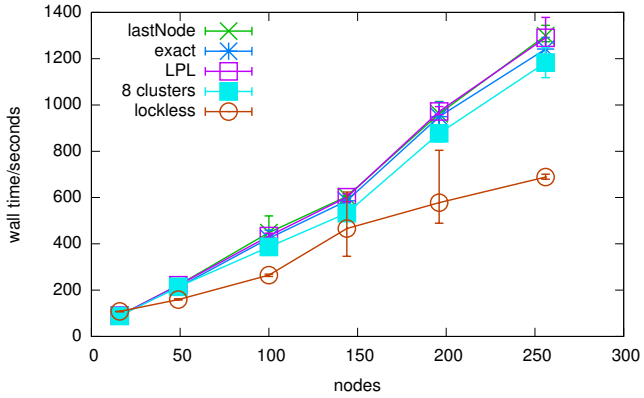
In Fig. 6, we show the results of the RadioOn application and the MVizLPLFFT application for point topologies where all 256 nodes are on the same position. While the results of the algorithms are very similar for MVizLPLFFT (and also MViz and MVizLPL), the RadioOn test highlights the suitability of the clustered algorithm and the lockless algorithm for synchronization-intensive applications. The results show a similar behavior as the results of the grid topologies.



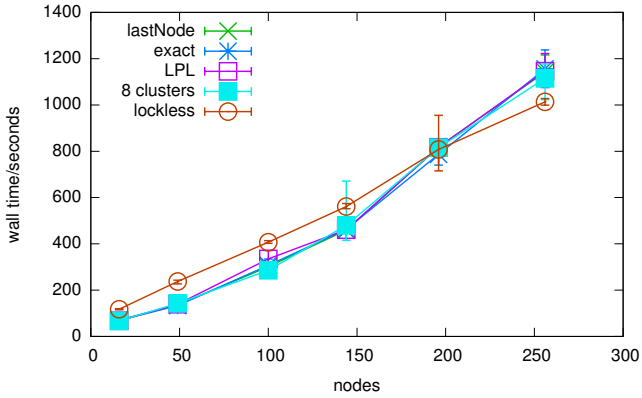
(a) MViz, 4 cores



(b) MVizLPL, 4 cores

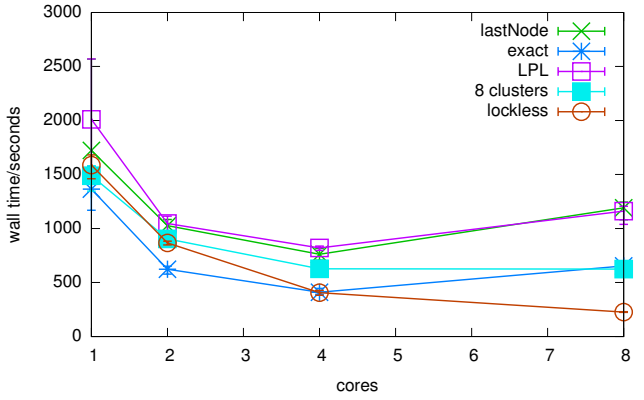


(c) MViz, 8 cores

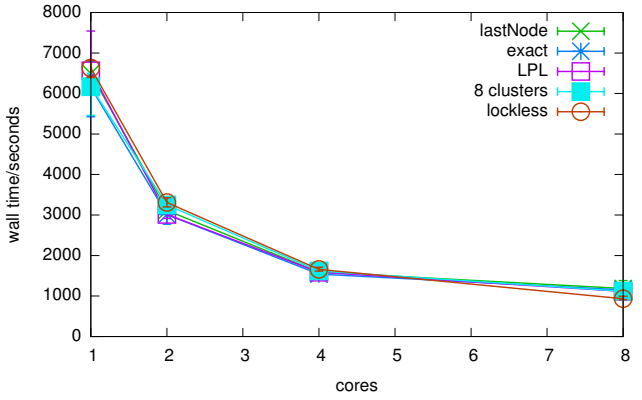


(d) MVizLPL, 8 cores

Figure 5. Wall times of Boreas algorithms on 4 and 8 cores



(a) RadioOn

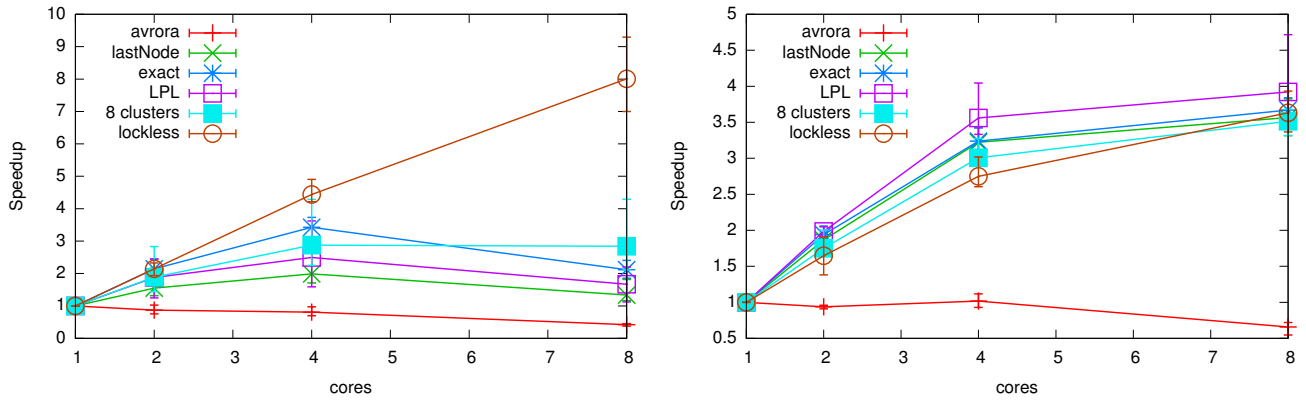


(b) MVizLPLFFT

Figure 6. Wall times for point topologies with 256 nodes

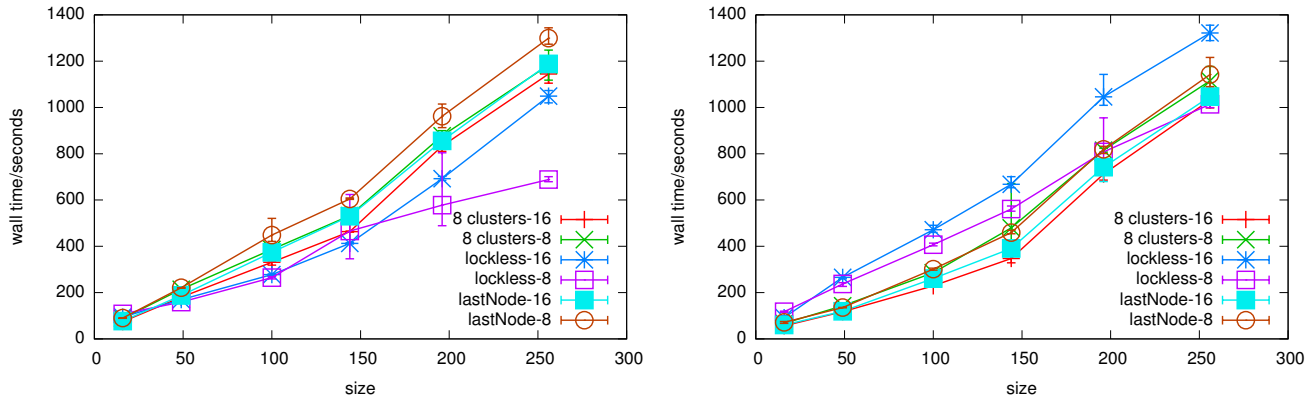
G. Speedup

The speedup is an important metric for evaluating the scalability of an algorithm. We show the speedup for the RadioOn and the MVizLPL applications emulated on 256 nodes in Fig. 7. The lockless algorithm is best suited for the synchronization intensive RadioOn application. However, the speedup is also slightly skewed due to the weakness of the algorithm on one core. The other algorithms show



(a) RadioOn (b) MVizLPL

Figure 7. Speedup for 256 nodes



(a) MViz (b) MVizLPL

Figure 8. Wall times for simultaneous multithreading (-8: 8 cores, -16: 16 cores)

the effect on using multiple CPUs with frequent synchronizations and the clustered algorithm performs once again slightly better than the rest of the Last Node algorithms. The MVizLPL application shows a different pattern with the LPL optimized version outperforming the other algorithms as expected. Because all threads are always eligible to run but may have to busy-wait, the lockless algorithm performs slightly worse than the others although gaining again when using 8 cores due to the lower overhead when multiple CPUs are involved.

H. Simultaneous Multithreading

Finally, we evaluate the impact of simultaneous multithreading (SMT). This technology aims at increasing the degree of capacity utilization of super-scalar CPU architectures where each core includes multiple instruction execution units, e.g., for integer calculations, that can function in parallel. When consecutive instructions of one thread cannot be parallelized due to interdependencies among them, SMT uses these free units for other threads. We show a sample of our results in Fig. 8. We use the basic Last Node algorithm, the clustered algorithm and the lockless algorithm both with 8 physical cores and with all 16 virtual cores. The lockless algorithm performs worse with 16 cores both in the MViz and the MVizLPL application. The performance of the algorithm is heavily impacted by the delay incurred for 'finding' the logically eligible thread to run by calling yield by the others. When, however, this thread is scheduled by the operating system on a virtual core, the delay may increase when another thread sharing the same physical core is using the necessary execution units. The Last Node variants, however, gain from using SMT. Since only logically eligible threads are in the running state, no congestion with the blocked threads occurs.

VII. Conclusions and Future Work

We show in the evaluation that the synchronization algorithm has an enormous impact on the performance of emulation. Our algorithms consistently outperform the stock algorithm of Avrora. In the most extreme case, RadioOn with 256 nodes on 8 cores, Avrora takes 459 times as long as our best algorithm – 193 times if we compare the best results for this network independently of the number of used cores. But also in small networks and with more realistic applications, our algorithms are faster and most importantly scale significantly better both with increasing number of cores and larger network sizes where the gain is usually an order of magnitude.

However, within the group of our algorithms there is no clear winner. The lockless algorithm performs best when the application requires a lot of synchronization as the RadioOn and the MViz test. The group of Last Node algorithms performs better with the LPL applications. The clustered algorithm provides a slight advantage when used with synchronization intensive tests. Additionally, these algorithms benefit from simultaneous multithreading. As a rough guideline, it seems beneficial to default to the Last Node algorithm with 4 cores or less or when SMT is available and use the lockless algorithm for more cores and especially multiple CPUs.

For the future, we are planning to evaluate the effect of orthogonal optimization techniques such as increasing the lookahead in combination with our synchronization algorithms.

Acknowledgments

This work has been partially supported by CONET, the Cooperating Objects Network of Excellence, and NOBEL (www.ict-nobel.eu) funded by the European Commission under FP7 with contract numbers FP7-2007-2-224053 and FP7-247926-ICT-2009-4.

References

[1] B. L. Titzer, D. K. Lee, and J. Palsberg, "Avrora: scalable sensor network simulation with precise timing," in *Proc. of the 4th Intl. symposium on Information processing in sensor networks*. IEEE Press, 2005.

[2] J. Polley, D. Blazakis, J. McGee, D. Rusk, and J. S. Baras, "ATEMU: a fine-grained sensor network simulator," in *Proc. of the 1st Annual IEEE Comm. Society Conference on Sensor and Ad Hoc Communications and Networks (SECON 2004)*, October 2004.

[3] H. Wu, Q. Luo, P. Zheng, and L. M. Ni, "VMNet: Realistic Emulation of Wireless Sensor Networks." *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 2, 2007.

[4] B. L. Titzer and J. Palsberg, "Nonintrusive precision instrumentation of microcontroller software," in *Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, ser. LCTES '05. ACM, 2005.

[5] Z.-Y. Jin and R. Gupta, "Improved distributed simulation of sensor networks based on sensor node sleep time," in *Proc. of the 4th IEEE Intl. Conf. on Distributed Computing in Sensor Systems*. Springer-Verlag, 2008.

[6] Z.-Y. Jin and R. Gupta, "Improving the speed and scalability of distributed simulations of sensor networks," in *Proc. of the 2009 Intl. Conf. on Information Processing in Sensor Networks*. IEEE, 2009.

[7] Y. Wen, R. Wolski, and G. Moore, "Disens: scalable distributed sensor network simulation," in *Proc. of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2007.

[8] Z.-Y. Jin and R. Gupta, "LazySync: A New Synchronization Scheme for Distributed Simulation of Sensor Networks." in *Proc. of the 5th IEEE Intl. Conf. on Distributed Computing in Sensor Systems*. Springer, 2009.

[9] A. Fraboulet, G. Chelius, and E. Fleury, "Worldsens: development and prototyping tools for application specific wireless sensors networks," in *Proc. of the 6th Intl. Conf. on Information processing in sensor networks*. ACM, 2007.

[10] P. Levis, N. Lee, M. Welsh, and D. Culler, "TOSSIM: accurate and scalable simulation of entire TinyOS applications," in *Proc of the 1st Intl. Conf. on Embedded networked sensor systems*. ACM, 2003.

[11] O. Landsiedel, H. Alizai, and K. Wehrle, "When timing matters: Enabling time accurate and scalable simulation of sensor network applications," in *Proc. of the 7th Intl. Conf. on Information processing in sensor networks*. IEEE, 2008.

[12] F. Österlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt, "Cross-Level Sensor Network Simulation with COOJA." in *Proc. of the 31st Annual IEEE Conf. on Local Computer Networks*. IEEE, 2006.

[13] J. Eriksson, A. Dunkels, N. Finne, F. sterlind, and T. Voigt, "Mspsim – an extensible simulator for msp430-equipped sensor boards," in *Proc. of the European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session*, Jan. 2007.

[14] M. Varshney, D. Xu, M. Srivastava, and R. Bagrodia, "Senq: a scalable simulation and emulation environment for sensor networks," in *Proc. of the 6th Intl. Conf. on Information processing in sensor networks*. ACM, 2007.

[15] Y.-T. Wang and R. Bagrodia, "Scalable emulation of tinyos applications in heterogeneous network scenarios," in *Proc. of the 6th Intl. Conf. on Mobile Adhoc and Sensor Systems*. IEEE, 2009.

[16] T. Lindholm and F. Yellin, *The Java(TM) Virtual Machine Specification (2nd Edition)*. Prentice Hall PTR, April 1999.

[17] Intel Corporation, *Intel(R) 64 and IA-32 Architectures Software Developer's Manual: Volume 3A: System Programming Guide, Part 1*, June 2010.

[18] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis, "Collection tree protocol," in *Proc of the 7th ACM Conf. on Embedded Networked Sensor Systems (SenSys'09)*. ACM, 2009.