Digital Force White Paper

Date: 26th February 2005
(Last Build: February 26, 2005)

Benchmarking the Distributed SystemC™ Synchronization Library Rev. 1.1.0

Mario Trams

Mario.Trams@digital-force.net

Thanks go to foobar GmbH as well as Chemnitz University of Technology, Chair of Computer Architecture for providing access to various hardware equipment for practical testing.

Digital Force / Mario Trams

http://www.digital-force.net

Benchmarking the Distributed SystemC™ Synchronization Library Rev. 1.1.0

Mario Trams

Mario.Trams@digital-force.net

Thanks go to foobar GmbH as well as Chemnitz University of Technology, Chair of Computer Architecture for providing access to various hardware equipment for practical testing.

Abstract

This paper presents an analysis in terms of performance characteristics of the distributed SystemC synchronization library in its revision 1.1.0. There are discussed simulation runs on different machines with various interconnects. Two kinds of applications are considered in more detail. The first one is a synthetic model and the second one is a practical dual–processor model based on the OpenRISC design.

1 Introduction

The basic operability of a library for distributed simulation of RTL–based SystemC models has been already demonstrated in [1] and [2]. Apart from the general purpose of the use of such library there is one important question: What does the synchronization library cost in terms of simulation run time. Or in other words: How can the synchronization library help towards reduction of simulation run time?

Of course, for the majority of cases the second interrogation will be the important one. But there are also cases where simulation speed is not the primary concern and the feature of having a distributed simulation model as such is more important.

In order to get a rough idea about the impact of the synchronization library on simulation run time, there are some exemplary simulation models needed which can be analyzed in terms of runtime behavior.

This paper discusses two kinds of exemplary applications or benchmarks. The first makes use of a rather simple and well understood synthetic model. Though, synthetic models do usually not resemble practical simulation models. But as they have a well–defined behavior they are a very nice tool for an objective analysis of various characteristics.

Apart from some formal characteristics it is also important to get a feeling about the behavior of a practical simulation model. So the second model discussed herein is a dual–processor system, which is of immediate practical use.

At the beginning of the paper, the setup of the test bed is shortly described. This does also include a brief overview about the communication latencies for various network connections which are used for later testing. The next and major part of the paper discusses the synthetic benchmark and its results. In section 4 the mentioned dual–processor simulation model becomes shortly introduced and its speedup results are presented and discussed. The paper concludes with some closing remarks.

2 Test Bed Setup

The following subsections describe various general aspects about the test bed that has been used for practical experiments.

2.1 Time Measurement Method

All times were measured by making use of `gettimeofday()`. This is not the most precise method, but the precision proved to be sufficient for this application.

The better way would be the use of the time stamp counter via the `RDTSC` instruction. However, the problem is that the use of the time stamp counter is dangerous in multiprocessor environments. This is because there is no guarantee that at a certain time the counters of all processors share the same value.

Practical tests have shown that the difference between the `gettimeofday()` and `RDTSC` are marginal for the investigations discussed herein.

In order to rule out accidental mismeasurements caused by unexpected events on the testing machines or whatsoever, every test run has been done a couple of times (5–10, typically). From these runs, the one with the smallest total simulation run time has been used.

2.2 Test Machine Configurations

Three types of test machines have been used for gathering practical measurements:

kermit/earnie (by courtesy of foobar GmbH):

- 2 × AMD Athlon MP 2600+ (2.13GHz clock)

- 256kB Cache per CPU
- AMD–760MP Chipset
- 512MB RAM
- Available TCP/IP Interconnects:
  Loopback Device (LO)
  Fast Ethernet (FE)
  Gigabit Ethernet (GE)
  Dolphin Sockets (SCI)
- gcc-3.2.3
- SystemC-2.1beta11

jack9/jack10 (by courtesy of TU Chemnitz):

- 2 × Intel P4 Xeon 2.4GHz (HT enabled)
- 512kB Cache per CPU
- E7500 Chipset
- 2GB RAM
- Available TCP/IP Interconnects:
  Loopback Device (LO)
  Fast Ethernet (FE)
  Gigabit Ethernet (GE)
- gcc-3.3.3
- SystemC-2.1beta11

nocona1/nocona2 (by courtesy of TU Chemnitz):

- 2 × Intel P4 EMT64 Xeon 3.4GHz (HT enabled)
- 1024kB Cache per CPU
- E7520 Chipset
- 512MB RAM
- Available TCP/IP Interconnects:
  Loopback Device (LO)
  Fast Ethernet (FE)
  InfiniBand Sockets (IB)
- gcc-3.3.3
- SystemC-2.1beta11
- **Note:** No 64Bit application code has been used for these machines.

If not stated otherwise, the parameters of the synchronization library were not changed (i.e. the default settings were used).

Whenever there are different results combined, it is assumed that all involved machine types are identical. That is, when the reference model has been benchmarked on a machine of type X, the corresponding distributed model has been benchmarked on two machines of type X. Alternatively, the corresponding distributed model has been benchmarked on the same machine of type X using the loopback TCP/IP connection. This ensures that the symmetry is always kept and there is made no comparison between apples and pies.

2.3 Connection Speed

When it's coming to communication between processes, the performance of the connection is important.

In order to see how the different communication subsystems perform for the communication pattern that is driven by the synchronization library, there has been developed a special benchmark. This benchmark is basically a send/receive benchmark and communicates just as the synchronization library does. That is, it is sending and receiving data in a non–blocking manner. If needed and/or necessary, sending and receiving data can be carried out interleaved. The latencies that are achieved with this benchmark are higher than what a simple ping–pong would achieve, but they are more close to reality.

Figure 1 shows a summary of the communication latencies for all involved machines and associated communication media. The maximal block size is 8kB, what is more than enough for the simulations considered in this paper. Notice that the latency axis has a logarithmic scale and starts at 6μs!

The measurements where obtained by communicating a number of packets while taking the time for all packets and dividing it by the number of packets. Each run was made multiple times and the smallest per–packet time has been chosen.

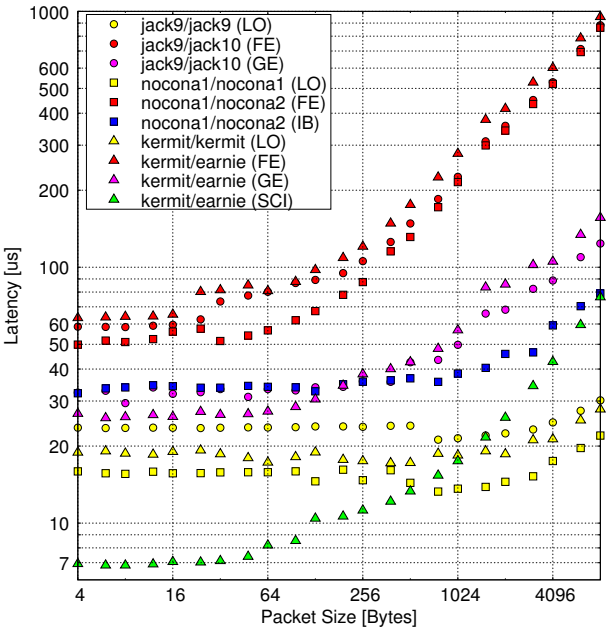

Fig. 1: Latencies for various Connections

The graphs sometimes show an interesting behavior. However, discussing them is outside the scope of this paper. These graphs are just shown as reference in order to demonstrate the quantitative differences of the different communication media.

3 Synthetic Benchmark

3.1 Basic Reference Model

Figure 2 shows the basic simulation model that is used for performance evaluation.

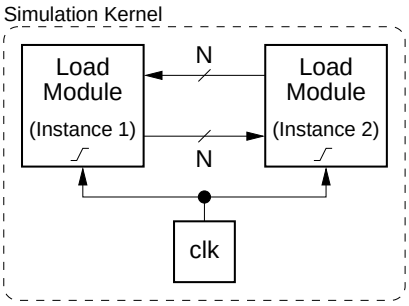

Fig. 2: Basic Model

There are two instances of a so–called load module. These modules contain a single `SC_METHOD` process that is triggered with each rising clock edge.

Both modules are connected by a customizable number of signals. These signals are of intrinsic type `bool`. Notice that these are individual SystemC signals and are not enclosed within a single bit vector! The only thing the modules are doing with the signals is to assign each output port the inverted value of the according input port (one by one for each port). Initially, all signals are set to zero by the main function.

So from a functional point of view, a load module represents a simple register with variable width and inverted output. The reason for the inverted output port assignment rather than a non–inverted assignment is to have the signals changed and not kept at fixed values.

Apart from the register–functionality, the load module contains a customizable busy–waiting loop (in fact, a simple for–loop). This loop resembles the actual load of the module. I.e. it fakes some real work that is not directly associated with SystemC interactions. In the following text, the *load parameter*, or in short *load* or L refers to the number of loops.

3.2 Distributed Model

The distributed model does not differ very much from the basic reference model that has been just discussed. Of course, from a functional point of view the distributed model is exactly the same as the reference model. Figure 3 shows the structure of the distributed model.

Both load modules are located in separate simulation kernels. The signals of both simulation kernels are appropriately tied together by making use of the synchronization library. `In` and `Out` marks inbound resp. outbound sync modules.

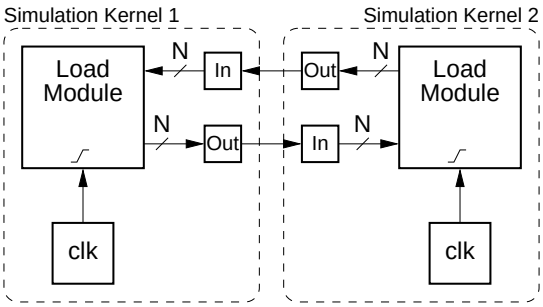

Fig. 3: Distributed Model

Notice that each simulation kernel incorporates its own clock. Both clocks need to be identical, of course. Actually, we could also select one simulation kernel for clock generation and feed it to the other kernel through the synchronization library. However, there are two reasons why this is not done. The first reason is symmetry. Especially for this more formal analysis both simulation kernels should be as identical as possible (although the additional load caused by the clock generation can very likely be neglected). The second reason is overall simulation speed. When the clock signal is to be synchronized as well, it has to be synchronized at twice the clock rate. Effectively, this doubles the number of synchronization cycles and hurts the overall performance. So when speed is a concern, clocks should be duplicated rather than synchronized. Remember that the two simulation kernels anyway need to be aware about the actual clock rate, as they have to specify according update cycles for outbound signals. So the hope for the elimination of potential misconfigurations by making use of clock distribution cannot be fulfilled anyway.

3.3 Denotation of Basic Time Parameters

Timing parameters that are subscripted by ref refer exclusively to the reference model, while timing parameters subscripted by $dist$ refer exclusively to the distributed model. In case neither ref nor $dist$ is present, this refers to a general timing parameter or a timing parameter that has not been specialized for either the reference or the distributed model.

There are two primary timing parameters that will be measured directly: The *total simulation run time* and the *load module time*.

The total simulation run time denoted t_{run} is the time spent inside the `sc_start()` call. Notice that t_{run} does not include the time of the elaboration phase of the simulation kernels.

The load module time is denoted t_{mod_1} resp. t_{mod_2} — one for each load module, or t_{mod} for a general designation. This time includes the duration of the for–loop as well as the loop that reads the input ports and writes the negated values into the output ports. t_{mod} is not measured as a whole. Instead, the run

time is measured for each individual clock cycle and becomes accumulated for the whole simulation run.

The speed–up is simply calculated by dividing the run time of the reference model by the runtime of the according distributed model:

$$Speedup = \frac{t_{run_{ref}}}{t_{run_{dist}}}$$

3.4 More formal Maths?

Well, initially it has been planned to draw more formal dependencies from the simulation model discussed herein. One special goal was to develop a formula that represents the runtime of the distributed model as a function of N, L, the runtime of the reference model, as well as the overhead of the synchronization library. However, as it turned out and will be shown later, the parameters of reference and distributed model cannot be easily merged together. But this would be actually needed in order to determine the synchronization overhead.

At the time of this writing it is believed that the synchronization overhead can only be reliably determined by implementing according timing measurements directly into the synchronization library. This has not been done yet.

3.5 Measurement Conditions

All single measurements for the individual runs have been made by using scripts. As previously stated, each measurement has been taken 5 to 10 times and the smallest recorded time has been used for further analysis. In some rare cases individual measurements differed significantly from the values they should have in view of a smooth graph. In those cases the individual measurements have been manually repeated until sensible times were measured. This was necessary, because sometimes even the multiple–run/minimum selection as described above did not yield usable values due to some reason. Sometimes there appeared highly scattered measurements where it was almost impossible to retrieve smooth results by repeating the measurement. In those cases the graphs were kept despite of the scattering.

All simulation runs used a clock of 10MHz (100ns clock cycle) and a total simulation time of 100000ns. This results in 1000 simulated clock cycles in total. Furthermore, all given times refer to totally accumulated times — i.e. not the times that accrue for a single simulated clock cycle, but for the whole simulation.

3.6 Reference Model Behavior

The simulation run time as a function of L (load) is as expected a clean linear function in all cases. There is no need to discuss details about that.

More interesting is the behavior of the simulation run time as a function of N (signal pairs). Well, formally we would expect a linear behavior as well. But practice is looking different.

Figure 4 shows the measured dependency of the total simulation run time as well as the module run time components as a function of N (for kermit and jack9). The diagram is cropped at the y–axis in order to keep the focus on the important aspects.

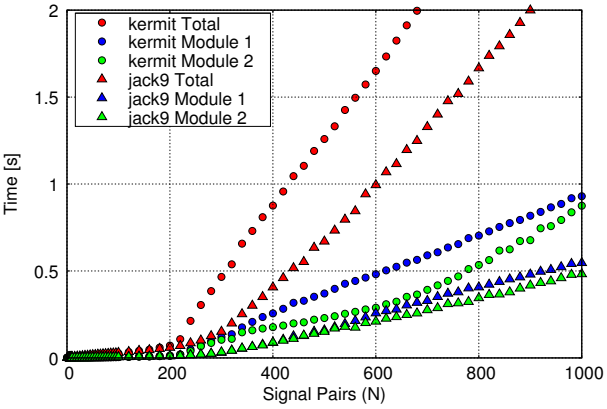

Fig. 4: kermit and jack9: Run Time depending on Signal Pairs (no Load)

There are two primary odds visible.

The first one is a major discontinuity at around $N = 200$ for kermit. This could be caused by caching effects as the amount of data SystemC has to deal with increases with increasing signal counts. On the other hand, 200 signal pairs (meaning 400 signals in total) are not that much. In case of jack9 the break appears at larger N. This might be due to the larger cache of the Xeons.

The second odd is the significant discrepancy between the time spent in the two load modules. This is especially extreme for kermit. Beginning at around $N = 300$ the gap between both modules is increasing. At around $N = 600$ module 1 needs almost twice as much time as module 2. With larger N, the gap is slowly closing again. This seems to be caused by some caching effect as well. An additional test has unveiled that this module that becomes instantiated first (normally module 1) needs more time than the other one. When signal info structures are brought into the cache by processing the first module, they are quickly available for the processing of the second module. The fact that the gap between both modules closes for large signal numbers is consistent with the caching–theory. This is because the larger N becomes, the higher is the probability that data brought into the cache during module 1 processing becomes overwritten by module 1 itself (and hence

resides not anymore in the cache).

Figure 5 shows the behavior on nocona1

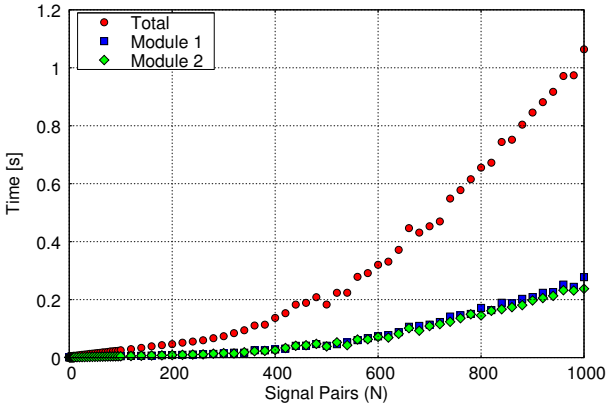

Fig. 5: nocona1: Run Time depending on Signal Pairs (no Load)

In comparison with kermit and jack9 we can see that the point where the runtime is increasing faster is shifted even more upwards. But in general, there is not such a hard break at all. This can be attributed to the significantly larger cache of nocona1 (twice as much as for jack9 and four times the size as for kermit). The difference between the two modules is visible, but is by far not that extreme as for kermit.

Figure 6 shows a summary of the total simulation run time of all tested machines on logarithmic scales.

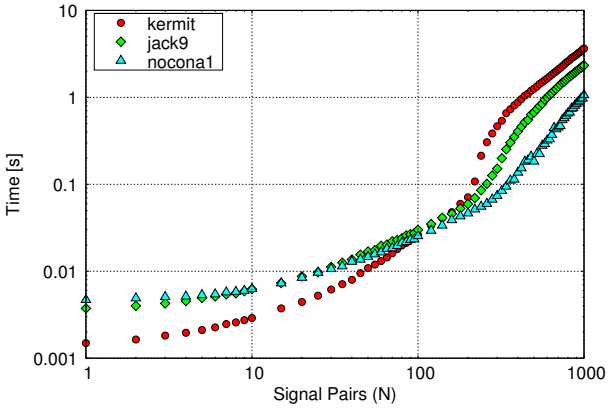

Fig. 6: Summarized t_{run} for Reference Model (no Load)

It is interesting to see that the run time for the Xeons (jack9, nocona1) is significantly larger than this for the Athlon (kermit) for smaller N. This reverses for larger N.

The (by specification) fastest machine — nocona1 — clearly shows its muscles for large N. However, it is the slowest machine for small N.

All these facts substantiate the suspicion that the strange effects are indeed caused by the caches. The Athlon (kermit) seems to have a faster (first level?) cache than the Xeons.

Sidenote: Just for the sake of interest, the memory consumption of the reference model has been observed for different N (up to 50000). The behavior is somewhat strange, and sometimes the the memory consumption decreases with increasing N. Overall, the memory consumption per signal is in the range of 1–6kB. The author is not very firm with the internals of the SystemC reference implementation, but this appears to be quite a lot. Perhaps it is just C++ as it lives...

3.7 Comparison of Reference and Distributed Model

3.7.1 Speedup Depending on Signal Pairs

In order to get an idea about absolute worst–case scenarios for the speed-up (or slow–down), we can have a look at the speedup as a function of N while $L = 0$. This is a worst–case scenario, because with $L = 0$ we have no additional computation time apart from the signal inversion.

Figure 7 shows the behavior for runs on various SMP machines using the loopback TCP/IP connection.

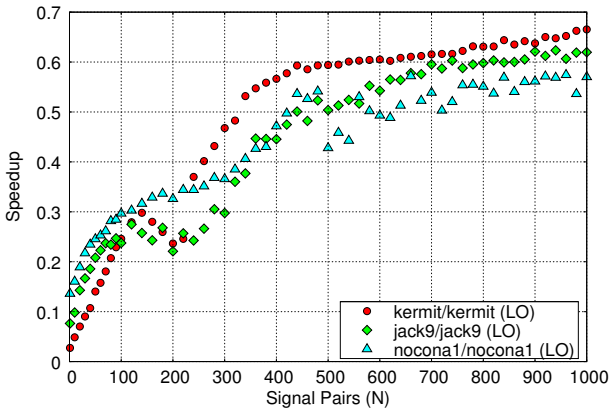

Fig. 7: Speedup for SMP runs ($L = 0$)

The results are quite interesting, indeed. We can see that the speedup is increasing with increasing N. At $N = 1000$ the speedup has almost reached its maximal value. Obviously, the savings due to the parallel execution of both load modules increase at a higher rate as the synchronization overhead does. An observation of the behavior for $N > 1000$ (graphs not presented) has shown that the speedup is decreasing very very slowly again.

Very interesting is the local minimum visible in the graphs. This minimum appears to match exactly with the number of signals were we see a change in the runtime behavior of the reference model ($N = 200$). Nocona1 is an exception as it does not show a local minimum. Nethertheless, there is a similar S–feature visible in that area. The graphs are a little bit scattered for larger N, especially in case of nocona1. It is not clear where this does come from.

Figure 8 shows the speedup development on jack9

and jack10 using Fast Ethernet (FE) and Gigabit Ethernet (GE). The loopback–curve (LO) for jack9 is shown again as reference.

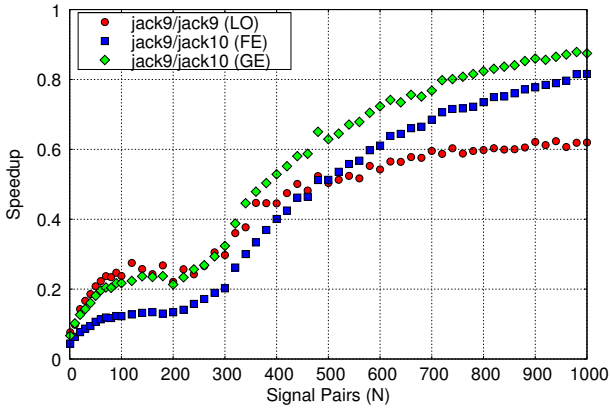

Fig. 8: Speedup on jack9/jack10 ($L = 0$)

It is a little bit surprising that for larger N the slow Fast Ethernet connection (FE) yields a better performance than the fast loopback connection. The Gigabit Ethernet connection (GE) is even more impressive when compared to the loopback connection. Even for smaller N the performances almost match each other before the Gigabit Ethernet connection starts gaining a significant advantage at around $N = 350$.

Obviously, the processors of the SMP system disturb each other too much when accessing main memory. This is yet another sign that SystemC appears to be quite cache–hungry even for moderate signal counts.

A look at the development of the average load module times ($\frac{t_{mod_1}+t_{mod_2}}{2}$) gives a clear evidence for this theory. Figure 9 clearly shows how the computation times decrease when separate machines are involved. Notice that the times for the reference graph are slightly smaller than the graphs where two machines were involved. This is because of the caching effect that we have already seen in figure 4.

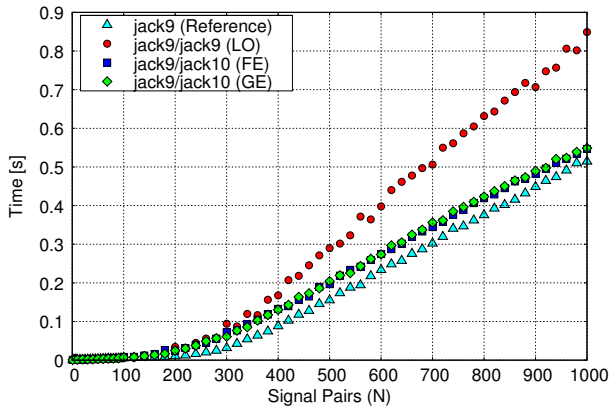

Fig. 9: t_{mod} (average) on jack9/jack10 ($L = 0$)

Ok, let's have a look at the behavior on other machine types. Figure 10 shows the curves for kermit and earnie for different networks. The loopback curve (LO) is again shown for reference.

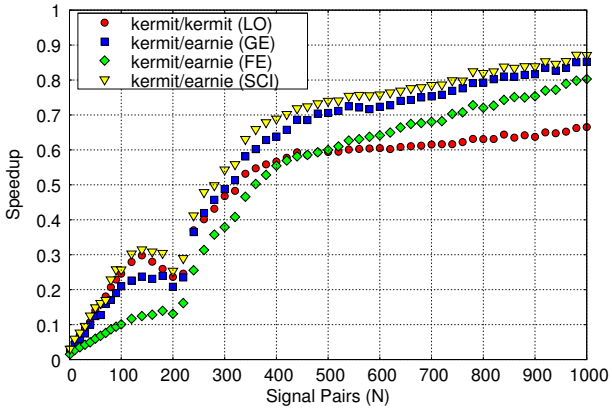

Fig. 10: Speedup on kermit/earnie ($L = 0$)

Basically, the behavior is much as we have seen it for jack9/jack10 (figure 8). Nevertheless, it is worth for some more words. The point is that the Dolphin Sockets (SCI) connection has a significantly lower latency than the Gigabit Ethernet connection (two to four times — depending on packet size) and is even faster than the loopback TCP/IP connection for smaller packet sizes. However, as it can be seen, the difference between Gigabit Ethernet and Dolphin Sockets is marginal. Only in the area from 100 to 200 signal pairs there is a significant advantage of the Dolphin Socket connection. This is a little bit disappointing. Intuitively it is clear that the speedup is better for interconnects with lower latencies. This becomes visible when comparing the curves for Gigabit Ethernet and Fast Ethernet.

One plausible explanation is that there exists some additional significant timing component that is independent from the raw communication latencies. So far it has been believed that the synchronization overhead is dominated by the communication times. These issues need to be analyzed in more detail later.

Finally, figure 11 shows the curves for nocona1 and nocona2.

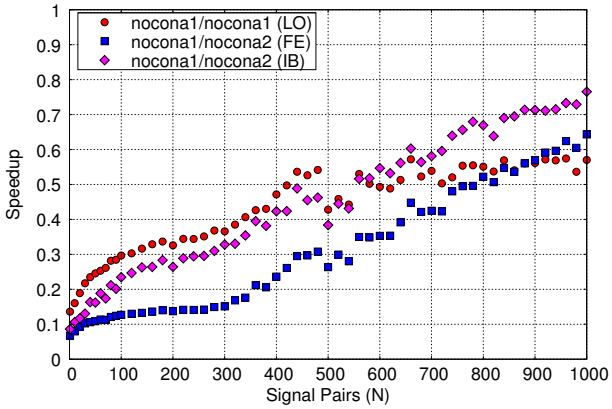

Fig. 11: Speedup on nocona1/nocona2 ($L = 0$)

It is not clear why the graphs become suddenly somewhat irregular for $N > 500$. This problem is already

visible in case of the reference model (see figure 5). The SMP–run using the loopback connection stays the most performing choice for quite a long time. This can be attributed to the larger caches.

3.7.2 Speedup Depending on Load and Signal Pairs

The following pictures show the speedup that has been measured for different L and N as well as for different machine/network configurations.

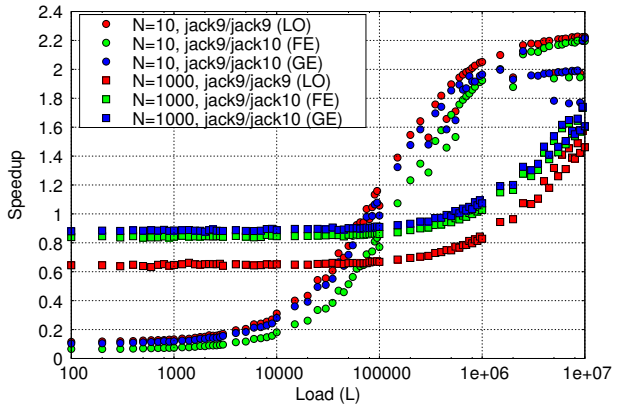

Fig. 12: Speedup on jack9/jack10

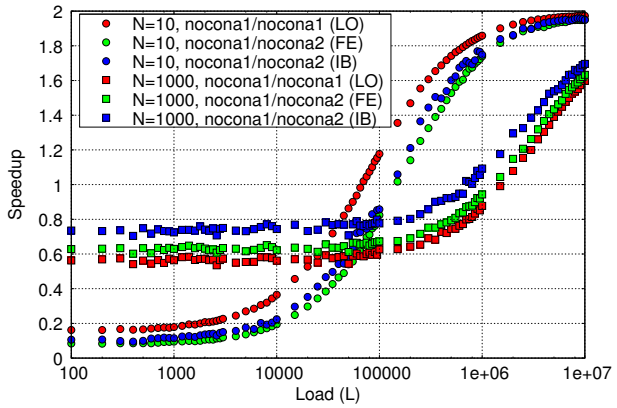

Fig. 13: Speedup on nocona1/nocona2

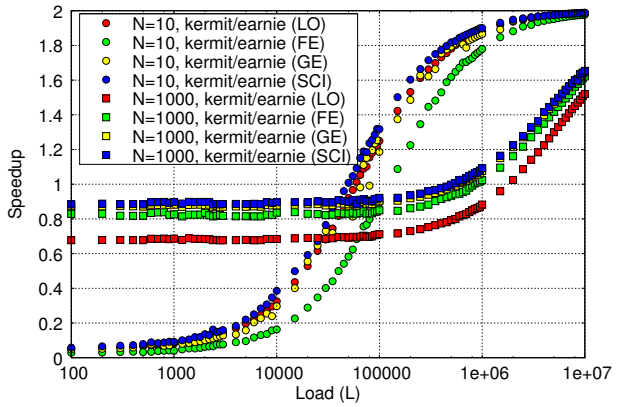

Fig. 14: Speedup on kermit/earnie

In general, the speedup development depending on load behaves as expected. That is, the larger the load, the higher the speedup. This is because the overhead caused by the synchronization library becomes less significant when the computation time increases. We can also see that the speedup is clearly exceeding the crucial value of 1.

The behavior of the speedup depending on the number of signal pairs is as expected. In section 3.7.1 we have already seen that the speedup increases when N increases. In addition, the larger N, the more load is needed in order to increase the speedup.

The curves for jack9/jack10 are somewhat strange. Notice that for large L the speedup is sometimes greater than 2 which means we see a real hyper–speedup here. In fact, a hyper–speedup is not that unrealistic — but not for this kind of model. The point is that the for–loop used for faking the load fits easily into the cache. That is, distributing the loops across two processors does maximally achieve a speedup of 2. Things would be different, when the load includes operations on signals that are local to each load module.

Back to the graphs for jack9/jack10. Especially suspicious is the detailed behavior of the shown graphs as they are a little bit scattered. A more careful analysis shows two trends for these graphs: One that is approaching 2 as an upper limit, and one that is exceeding 2. This behavior is introduced by $t_{run_{ref}}$, while $t_{run_{dist}}$ shows an almost perfect curve (graphs not shown). So apart form the hyper–speedup theory, it is more likely that the speedup values > 2 are simply caused by too bad runs of the reference model. This strange behavior of the reference model seems to be unique to these Xeon–Systems and becomes visible when the total simulation run times increase. An increase of the number of tests for each configuration did not help here. Perhaps this effect is caused by the HyperThreading. On the other hand, the noconas which feature HyperThreading as well do not show such an effect.

Overall, the results for all three test configurations are very comparable. That is, for $N = 10$ the speedup surpasses the 1–mark for around $L = 100000$ and for $N = 1000$ this point is at around $L = 1000000$. The impact of different communication media is nicely visible as well and is consistent with what we have seen in section 3.7.1. Notice that the Gigabit Ethernet curve (GE) for kermit/earnie and $N = 1000$ (figure 14) is almost exactly overlaid by the Dolphin Sockets curve (SCI).

3.7.3 Influence of the Send Buffer Size

In order to optimize communication operations, the synchronization library contains buffers intended for collecting signal change notifications. This makes it possible that a bunch of notifications can be sent within a single transaction. Every outbound sync module has such a send buffer whose size is config-

urable by the application (the default size is 4096 bytes). The size of the according receive buffer in the inbound sync module becomes automatically adjusted depending on the remote send buffer size.

To interpret the runtime behavior as it depends on the buffer size, it is important to know how much data is to be transferred at all for a single synchronization cycle. All the signals the model is dealing with are of type `bool`, which is represented by a single byte (`sizeof(bool) = 1`). Signal designators are of type `unsigned int`, or 32 bits wide. Hence, every signal notification occupies 5 bytes.

Figure 15 shows the speedup depending on the buffer size as it has been measured for nocona1 and nocona2 (the load L is zero for all measurements).

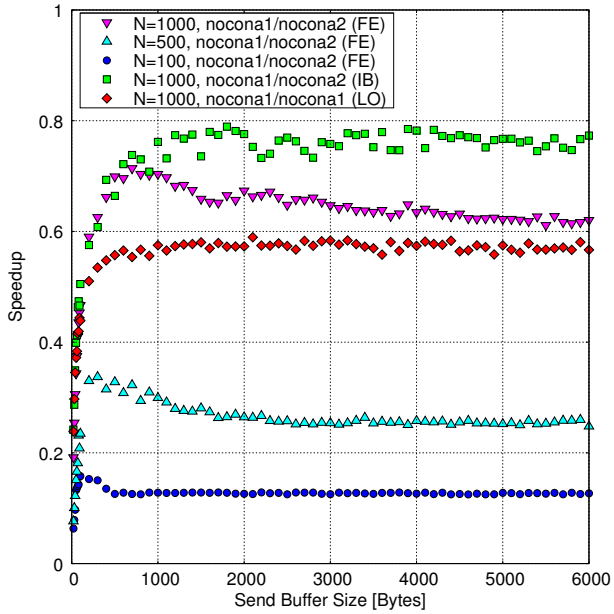

Fig. 15: noconas: Speedup depending on Send Buffer Size ($L = 0$)

All graphs show a bad behavior for very small buffer sizes. This is trivial because the communication overhead is very high due to lots of small packets that are to be transferred.

As shortly described in [3], large buffer sizes can be contra–productive under certain circumstances. The measurements shown above prove and quantify this theory.

Clearly visible, for the Fast Ethernet connection the default buffer size of 4096 bytes is not the best selection. For $N = 1000$ the optimal buffer size is between 500 and 1000 bytes. For smaller N the optimal buffer size shifts slightly downwards.

The performance does not change anymore when the buffer size becomes increased beyond the total amount of data that is to be transferred per synchronization cycle. Extra buffer space won't be touched at all. For $N = 1000$ it is 5000 bytes, for $N = 500$ it is 2500 bytes, and for $N = 100$ 500 bytes. The graphs prove this, basically (slight deviations are caused by typical dispersion of real–world measurements). It can also be seen that by selecting an optimal buffer size the achievable net gain decreases with decreasing N. For instance, for $N = 10$ (graph not shown) there is no visible gain at all. This is because there is just too less data to be transferred. In this case, spreading the data across multiple packets does only hurt the performance because of the significant communication overhead.

The loopback (LO) connection (only shown for $N = 1000$), does show up almost no special features. But a closer inspection (not good visible in figure 15) shows basically the same effects, but at a smaller magnitude. The InfiniBand (IB) graph that is shown for $N = 1000$ also has an interesting behavior. But the changes in performance are not that large as it is the case for the Fast Ethernet connection.

Figure 16 shows the impact of the buffer size for jack9 and jack10. Here it is especially interesting that for

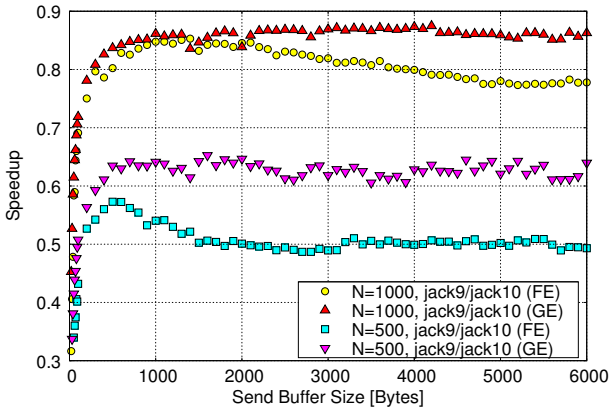

Fig. 16: jack9/10: Speedup depending on Send Buffer Size ($L = 0$)

$N = 1000$ the best performance with Gigabit Ethernet (buffer size of around 3000 – 4000 bytes) is just slightly higher than for the cheaper Fast Ethernet for a buffer size of around 1000 bytes. For smaller N the gap between Gigabit and Fast Ethernet slightly opens as it can be seen in the figure for $N = 500$. Here the lower latency of the Gigabit Ethernet pays out and cannot be compensated for Fast Ethernet by overlapping the communication. Notice that the graph for $N = 500$ and Gigabit Ethernet is showing an unsteady behavior even for buffer sizes > 2500 bytes, although it should be mostly constant from there. These disturbances cannot be a cause of the buffer size at all, actually.

Figure 16 does not show curves for loopback communication as there is nothing interesting visible (just like in case of nocona1, figure 15).

In case of kermit and earnie the behavior is not much different as we have seen it for the other machines. Some graphs for the Fast Ethernet connection have been shown nevertheless (figure 17) because of the interesting incursions in the area of 2000 and 3000—3600 bytes for $N = 1000$. Those drops in perfor-

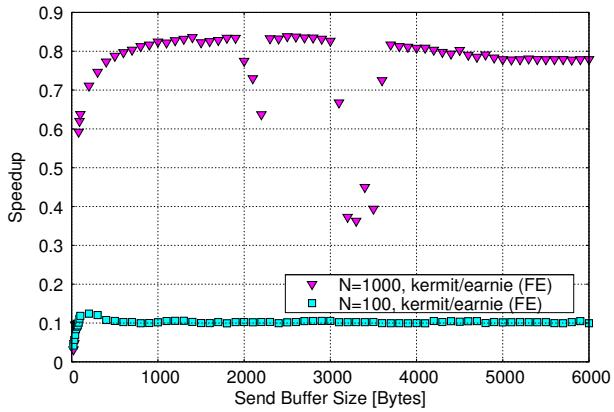

Fig. 17: kermit/earnie: Speedup depending on Send Buffer Size ($L = 0$)

mance were somehow symptomatic. Probably these critical buffer sizes cause some strange interferences leading into significant loss of performance.

Performance Gain Explanation

Why exactly do we see an improvement of performance for smaller buffer sizes? Both simulation kernels are perfectly symmetric. That is, for each clock cycle both kernels equally process the load modules and enter the synchronization library at almost the same time.

First let's assume that the transmission buffer is large enough to take up all data for the current synchronization cycle. It will take some time until all data has been placed into the buffer. This time is also formally equal for both kernels. Then, the buffer will be sent across the network. Depending on the network technology, this takes more or less time. During transmission, the kernels do nothing because the synchronization library cannot proceed until there has been received some data. The higher the communication latencies, the longer is spent without doing anything worthwhile.

When the buffer sizes are decreased so that they cannot take up all data for a single synchronization cycle, the communication can be done in background and in parallel with the actual signal processing by the synchronization library.

Clearly, the positive effect of overlapped processing becomes less effective for lower–latency networks.

The exact dependency for the optimal buffer size is a function of N, the CPU performance, and the network characteristics. The latter one mostly includes the latency. So far, there have been made no attempts to formalize this dependency.

3.8 Conclusions for the Synthetic Model

We have seen that the synchronization library can significantly speedup the simulation of a SystemC model.

Overall, the selected model represents a worst–case scenario in various aspects. Therefore the results can be considered as worst–case results accordingly. This does also apply to the results discussed in section 3.7.2 (speedup depending on load). Remember that the load consists of a simple for–loop that is very cache–friendly. So the load modules do not access any memory location, except for the signal inversion. The load of practical simulation models will involve more or less heavy operation on other signals. There, a distribution of the model will pay out much better. Therefore the synthetic model should be perhaps changed for future analysis so that a number of local signals are changed instead of doing just a simple busy–waiting delay.

But one can also imagine to consider generally different models, especially such ones including more than two modules and various different connection topologies. Some that are coming in mind first are

- Several modules connected in a ringed–fashion.
- Several fully connected modules.
- Several modules connected in a star–fashion (one central module where the others are connected to).

So there is plenty of stuff left to be done for the future in this area.

4 Benchmarking a Real Model

As we have seen for the synthetic benchmark, the synchronization library is capable of delivering a significant speedup. Though, one can hardly imagine what these results mean for the situation in case of practical simulation models.

So in order to get an idea about the relations for simulation models of practical use, an exemplary simulation model is needed first.

It has been decided to use the OpenRISC processor [4] as basic element (OpenRISC 1200 or OR1200 in particular). This processor model cannot be considered to be very huge. However, with around 40 modules and more than 10000 lines of code it cannot be considered to be very small as well. The OR1200 is freely available in form of synthesizable Verilog code. This is a problem, as a SystemC model is needed. So the whole Verilog code as been (manually) ported 1:1 into SystemC, which was a quite annoying task...

An alternative for the Verilog→SystemC conversion would be the use of "Verilator" [5]. However, this has not been done because the use of Verilator appeared a little bit complicated for the first view, and more importantly, because the SystemC model should be a synthesizable one. Synthesizable (i.e. RTL) models are more close to reality and simu-

lation run times are a much larger problem there as it is the case for more behavioral models. According documentation, Verilator generates special code for intra–module communication without using SystemC mechanisms. While it is claimed that this speeds up the simulation significantly (which is certainly true), the SystemC model is not synthesizable. In the end, Verilator's primary purpose is just a fast simulation of Verilog models. Of course, in general the "verilated" Verilog code should be suitable as well for a distributed SystemC simulation.

As the Verilog code has been really ported 1:1, the result can be considered as almost synthesizable. Only "almost", because some parts, especially RAMs, are not synthesizable. There have been also made some small C++–technical changes of the description in contrast to commonly published templates for synthesizable code. In particular, the module constructor code has been moved from the header file into the implementation file. Wilson Snyder (the primary author of Verilator) has also criticized in [6] the commonly propagated bad practice of placing module constructor code into the header files. Besides the problem that Wilson mentioned, this causes large compile times for upper hierarchy level modules because the compiler needs to work through the code of all included header files every time. Anyways, it has not been checked whether common synthesis tools can deal with such a code layout (the author does not have access to such tools).

The OR1200 is highly customizable. The used model has been configured as following (major things):

- no DMMU, no IMMU
- 1way/512 bytes code cache
- 1way/4kB data cache
- 32 registers
- no debug–unit
- no PIC
- Tick–Timer implemented

One possible option for demonstrating the simulation of a larger model would be to split a single processor into several parts and distribute these. However, it has been decided to construct a simple two-processor system and place the single processors into separate simulation kernels for the distributed simulation.

Figure 18 shows the reference model of the small dual–processor system.

Every processor has got its own instruction RAM while both are sharing a single data RAM. For the sake of simplicity, the data RAM has been laid out as dual–ported RAM. There is also no cache–coherency preserved.

Figure 19 shows the according distributed model version.

So the second CPU has been moved together with its

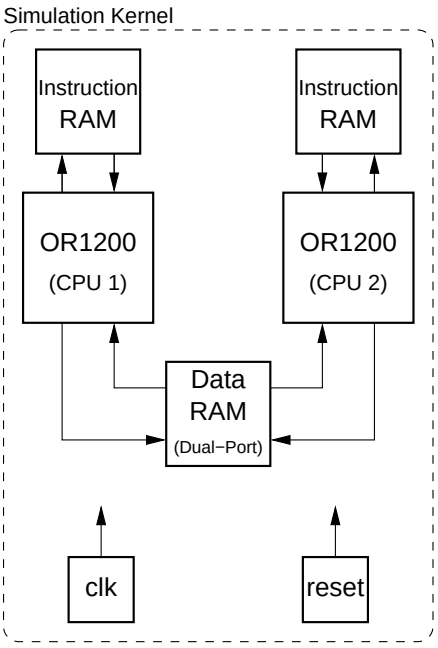

Fig. 18: Reference Model

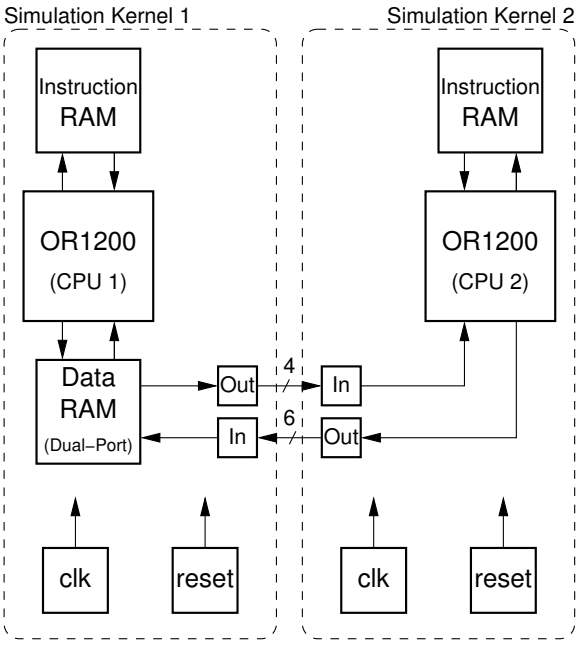

Fig. 19: Distributed Model

instruction RAM into a separate simulation kernel. Both clock and reset circuitries have been duplicated. The reasons for the duplication have been already described in the context of the synthetic model (section 3.2). The common data RAM remains with the first CPU in one and the same simulation kernel. This causes a slight imbalance, but the RAM model is almost nothing compared to the OR1200 model.

The number of signals that are to be synchronized is rather small. There are just 4 signals that are feed out of simulation kernel 1 and 6 signals that are feed in.

The CPUs itself run a small program that enables

the caches and then fills some memory block with some data in an endless loop.

Note: It has been observed that the simulation run times significantly depend on the code that is being executed by the simulated CPUs. There has been made no in-depth analysis here, but once a test has shown a tripled runtime when the CPUs process the code that is used here or just NOPs. The reason for this behavior is that SystemC is event-driven and needs to process only those signals that have been actually changed. Less program activity means less signal changes and hence less simulation run time.

Also notice that the used synchronization library anyway synchronizes all signals — regardless whether they have changed or not. For the small amount of involved signals there would be very likely almost no advantage when signals are only synchronized when it is really needed.

4.1 Results

The simulation run times for all following measurements have been obtained for 100000 simulated clock cycles. Like in case of the previously discussed synthetic benchmark, the shown times include only this time that is spent for the actual simulation (i.e. for `sc_start()`). That is, initialization and synchronization library connection setup is not included. However, practice has shown that this extra overhead just takes a fraction of a second here.

Every simulation run has been carried out a couple of times and the smallest measured time has been used for further analysis.

Figures 20 to 22 illustrate the results for various machines and various communication networks.

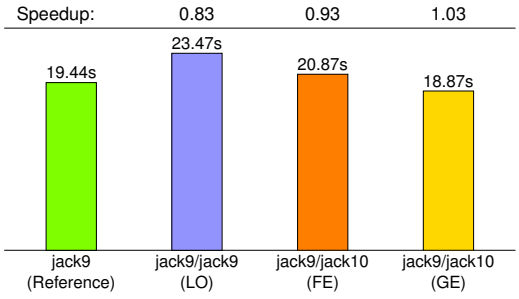

Fig. 20: Results for jack9/jack10

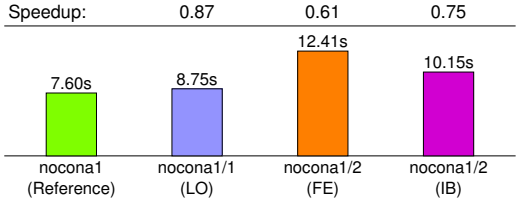

Fig. 21: Results for nocona1/nocona2

These results are not an exact mirror of the behavior

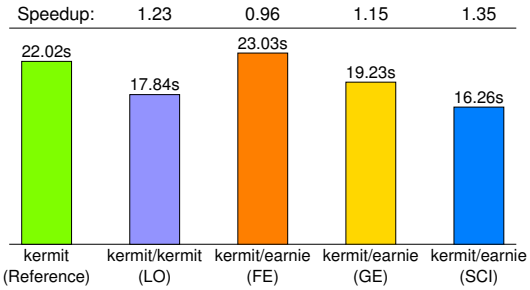

Fig. 22: Results for kermit/earnie

of the synthetic model. The SMP–runs on jack9 and nocona1 are slightly slower than the according reference run. Interestingly, the run on kermit shows with 1.23 a quite significant speedup. This is especially interesting, because the Athlons are those processors with the smallest cache.

The total simulation times for the Fast and Gigabit Ethernet connections are comparable between jack9/jack10 and kermit/earnie. The Dolphin Sockets connection (SCI) shows with 1.35 the best speedup of all tests. There is also a quite big advantage over the Gigabit Ethernet connection. This is a little bit surprising, as the synthetic benchmark showed almost no difference between both interconnects.

In case of nocona1/nocona2 there is no valuable speedup at all. As the noconas are the fastest machines, the speedup results are generally worser than those for the other machines with comparable interconnects. This is natural, as the synchronization overhead is more significant here. Or expressed in other words: The simulated model is just too small.

5 Discovered Problems

Despite the aspect of performance analysis, the tests that have been made were also a good stress test for the synchronization library. Basically, almost everything went smooth as expected. In case of the simulation of the dual–CPU model not a single ambiguity has been logged.

However, the synthetic benchmark unveiled a bug of the synchronization library rev. 1.1.0. Though, it occurred only under extreme and mostly unrealistic conditions.

The problem is that the distributed simulation freezes under certain conditions and the processes do not consume any CPU time anymore. The condition depends on the machine type, the communication network, the buffer size, and the number of signal pairs (N). For instance, the problem occurred on jack9/jack10 and Fast Ethernet with $N = 19000$ and the standard buffer size of 4096 bytes. For faster connections (i.e. Gigabit Ethernet or loopback device) the problem occurred for $N = 21000$ resp.

$N = 36000$. In case of kermit/earnie the problem has been already logged for $N = 100$ when using the Dolphin Sockets, but with an unrealisticly small buffer size of 20 bytes. In case of nocona1/nocona2 the problem did not occur at all.

Finally, the problem could be pinned down quickly and it occurs due to a slight mishandling of non-blocking send/receive operations. The bug has been fixed and an updated synchronization library will be made publically available soon.

6 Closing Remarks and Conclusions

What are the lessons learned?

First of all and most importantly we have seen that the distributed simulation can yield an effective gain in performance. Of course, there was basically no doubt about it, but this paper has given a more quantitative insight into the question. This is especially true for the discussed dual–processor model. There we have seen that its complexity is almost high enough for achieving a break–even or a moderate speedup — depending on the simulation hosts and networks.

The general procedure for distributing a certain model should be a partitioning of the design into as equal as possible parts that are connected by as less as possible signals. When the number of signals that are to be synchronized is rather large (hundreads to thousands) and the simulation hosts are rather fast compared to the network connection, it might be worth to play with the buffer size. As we have seen in section 3.7.3, the achievable gain can be quite significant.

One aspect in the road map of synchronization library enhancements is the support for SystemV shared memory as communication medium. This is important, as it cuts down the communication latencies significantly. However, almost all measurements basically show that the use of SMP systems does not unveil the best results. The processors just hinder each other too much. The only exception are the Athlons, where at least the SMP–run of the distributed processor model has shown a rather good result. Of course, we have to admit that the SMPs used here are merely low–cost bus–based models. That is, they do not feature multiple memory–banks and an associated crossbar. In this context it would be interesting to see the behavior on multi–processor AMD Opteron systems. As these processors feature a built–in memory controller, they have a per–processor RAM and different processors do not hinder each other at all while accessing private memory. Unfortunately, there were no Opteron systems accessible for testing at the time this paper has been written.

When somebody is concerned with acquiring new hardware equippment for his distributed SystemC simulation tasks, the following rules of thumb can be given so far:

- Of course, the actual machines should be as fast as possible and should incorporate caches as large as possible.
- As for the question whether to go with SMP machines or not, the tendency is more in direction of single processor machines. As described above, the provisions against the SMPs count mostly for those rather simple bus–based systems. Nethertheless, dual processor systems might be useful — but not with the intention of running two simulation kernels on one system. The reason is rather to have one CPU exclusively available for one simulation kernel while the other one can deal with other concurrent tasks. Though, as we have seen in case of the Athlons and the CPU model, there can be achieved a speedup as well (figure 22).
- When there are many (thousands) signals to be synchronized, not too much emphasis should be put on fast and expensive networks. Standard networking technology in junction with well (experimentally) selected buffer sizes can do the job evenly good.
- Ultra–fast (and expensive) networking technologies become interesting when there are only tens to hundreads of signals to be synchronized. Notice that “ultra–fast” means **low latencies** rather than high bandwidths!

References

[1] Mario Trams: *Conservative Distributed Discrete Event Simulation with SystemC using Explicit Lookahead.* Digital Force White Paper, February 2004. Available from http://www.digital-force.net/publications

[2] Mario Trams: *A First Mature Revision of a Synchronization Library for Distributed RTL Simulation in SystemC™.* Digital Force White Paper, November 2004. Available from http://www.digital-force.net/publications

[3] *User Manual for Distributed SystemC™ Synchronization Library Rev. 1.1.0* Digital Force Public Documentation, November 2004. Available from http://www.digital-force.net/projects/dist_systemc

[4] OpenRISC project website: http://www.opencores.org/projects.cgi/web/or1k

[5] Verilator project website: http://www.veripool.com/verilator.html

[6] Wilson Snyder: *Verilator and SystemPerl.* Presentation at NASCUG, June 2004. Slides available via http://www.veripool.com/verilator.html