Automated Concurrency Re-assignment in High Level System Models for Efficient System-Level Simulation

Nick Savoiu Sandeep K. Shukla Rajesh K. Gupta
Center for Embedded Computer Systems,
Department of Information and Computer Science,
University of California at Irvine,
Irvine, CA 92697
E-mail: {savoiu,skshukla,rgupta}@ics.uci.edu

Abstract

Efficient modeling of concurrency and reactivity along with their efficient implementation in the simulation kernel are crucial to the overall utility of system level models using the C++ based modeling frameworks. However, the concurrency alignment in most modeling frameworks are naturally along hardware units, which is supported by the various language constructs, and the system designers express concurrency in their system models by providing threads for some modules/units of the model. Our experimental analysis show that this concurrency model leads to inefficient simulation performance, and a concurrency alignment along dataflow gives much better simulation performance, but changes the conceptual model of hardware structures. As a result, we propose an algorithmic transformation of designs written in these C++ based environments with concurrency alignment along units/modules. This transformation, provided as a compiler front-end, will re-assign the concurrency along the dataflow, as opposed to threading along concurrent hardware/software modules, keeping the functionality of the model unchanged. Such a front-end transformation strategy will relieve hardware system designers from concerns about software engineering issues such as, threading architecture, and simulation performance, while allowing them to design in the most natural manner, whereas, the simulation performance can be enhanced upto almost two times as shown in our experiments.

1. Introduction

The advent of System-on-Chip (SoC) solutions has posed a need for efficient system level models for early design stage tradeoffs. Accuracy and speed of simulation are important criteria for employing a model for design exploration. When considering efficieny of simulation, it is common to rely on concurrency enhancements, and concurrency management techniques (using multi-processor hardware, and multi-threading etc.). However,actual simulation performance may vary significantly depending on the specific choices of concurrency support and mechanisms in the framework, and how they relate to application level programming. Further more, explicit modeling of concurrency is an important aspect of any hardware language framework. The recently introduced C++ based high level modeling frameworks such as SystemC [11], Cynlib [1], OCAPI [8], etc., model concurrency using either function calls or thread packages. The thread library employed can be cooperative or pre-emptive [7]. However, since most existing system level simulation and modeling frameworks have implemented their kernels based on application level threading libraries (co-operative threads) [5], and these user-space threads are transparent to the kernel, these cannot take advantage of multi-processor systems, for performance improvement.

The other dimension to the simulation performance problem is that hardware concurrency, most naturally, is aligned along the units/modules. In other words, system designers, due to natural design intuition, and due to language constructs provided in the existing language frameworks, build concurrency into the model by providing threads for some modules/units of the model. Garg et al., in [2], discuss trade-offs between function calls, and threading, and how to cluster modules together, to obtain an efficient simulation model. However, in our experiments, we find that, due to threading packages used in the simulation kernels, as well, as designers choice of mapping functionalities into threads, such manual design trade-off is not easy to achieve. In this work, we experimentally study the efficiency of the concurrency constructs in the context of SystemC. We specifically address two questions:

1. What is the best threading mechanism to be used in the simulation kernels for efficiency of the simulation models?

2. Given a specific threading mechanism what is the best mapping from hardware modeling constructs, and libraries to the underlying threading package?

In connection with the first question, we present results of our experiments with various threading models, for SystemC simulation kernel. Towards answering the second question, we show that the task based concurrency [10], aligned along system units/modules is much less efficient for simulation than if threads were assigned to dataflows in the models. But designing module based concurrency is more intuitive for designers. Hence, we propose an algorithmic transformation of designs written in these C++ based environments. This transformation, will *re-map* the concurrency along the dataflow of the computation of the model, as opposed to threading along concurrent hardware/software modules, keeping the functionality of the model unchanged. This re-maping can also map the computation of the model onto kernel level threads, whereby, exploiting multi-processor systems as well. The simulation performance benefits of such re-mapping is shown through experimental results, and the strategies for such transformation is outlined in the paper. Such a front-end transformation strategy will relieve hardware system designers from worries of threading, simulation performance issues, and allow them to design in the most natural manner, whereas, the simulation performance will not be sacrificed.

2. Multi-Threading in SystemC

For modeling of the hardware systems, the modeling framework should have constructs for modeling reactivity and concurrency. For example, SystemC provides efficient constructs for modeling reactivity in form of *watching* and *waiting*[6]. For modeling concurrency SystemC provides two type of processes: *synchronous* and *asynchronous*. A *synchronous* process is a process that communicates with other processes only at specific instances of time determined by the clock edge to which the process is sensitive. On the other hand, *Asynchronous* objects are more general form of the synchronous processes that can be used to model any kind of circuit. SystemC provides two types of asynchronous objects: an *asynchronous process* and an *asynchronous block* [1].

In SystemC, the asynchronous block is implemented using function calls. These are evaluated by the *main* SystemC kernel thread. The synchronous and asynchronous processes are implemented using *Quick* thread library[5]. The Quick thread provides a *co-operative* thread library, where a thread yields the control to another thread at its will. A thread cannot pre-empt another on the basis of priority or blocking. So at a time only one thread can be in *running* or *blocked* state while all the other threads are in *ready* state waiting for the running thread to yield the control. The SystemC[11] kernel follows the following simulation semantics:

1. All asynchronous objects with sensitive input signals that have changed are executed.
2. All the signals that have been changed are updated.
3. Step 1 and 2 are repeated until no signal changes its value.
4. All synchronous processes sensitive to the clock edge that has changed are executed.
5. Simulation time is advanced to next clock edge.

Recognizing that co-operative threads are not known to operating system kernel, and hence cannot be scheduled on multi-processor, we embarked on the following experiment. We replaced the usage of Quick threads in SystemC with the *POSIX* threads[12] to have a pre-emptive version of the SystemC kernel. With POSIX threads, if the running thread is blocked then the control is passed to another ready thread by the system. As the result a blocked thread would not occupy the CPU and a ready thread can be switched to utilized the CPU. The simulation semantics still remain the same. The asynchronous blocks are still executed sequentially, but the asynchronous processes can go in parallel with each other. Similarly all the synchronous processes can be executed in parallel with each other. An asynchronous process and a synchronous process cannot go in parallel to maintain the SystemC simulation semantics. The mutual exclusion for the shared data structures has been implemented using *semaphores*. The SystemC implementation of a reasonably sized example, namely AMRM prototype [9] has been simulated for various configurations. We vary the number of threads by altering the balance between asynchronous blocks and processes in the model. Initially all the components are modeled using asynchronous blocks. One-by-one, we moved the asynchronous blocks to asynchronous processes to see the effect of increasing threading on the simulation speed.

A Sun Ultra 5 station is used for running simulation in single processor enviornment. We used a Sun Ultra 4 workstation with 4 processors to run simulations in a multiprocessor environment. The different SystemC simulation kernels used are one with *co-operative Quick threads* and other with *pre-emptive POSIX threads*. As the components are moved to asynchronous processes, the simulation times in minutes are shown in table 1. The column 1 in the table represents the total number of threads in the system when

[1] Although our experiments were done with SystemC version 1.0.2, we believe, the same discussions hold for SystemC 2.x, as well.

the component is modeled with asynchronous process. Column 2 shows the simulation time for simulating in a single processor environment with co-operative threading. Column 3 corresponds to multi-processor environment with co-operative threading. Column 4 represents single processor environment with pre-emptive threading. Column 5 corresponds to multi-processor environment with pre-emptive threading.

Number of Threads	Co-oper Single	Co-oper Multi	Pre-empt Single	Pre-empt Multi
0	9.43	9.11	12.59	9.21
12	9.33	9.00	17.13	48.45
19	9.49	9.14	19.27	68.00
41	9.50	10.12	28.35	85.00
54	9.54	10.60	28.07	120.0

Table 1. Simulation time with co-operative and pre-emptive threads in single and multi processor environment

As can be seen from the table, the co-operative threading implementation has out-perfomed the pre-emptive threading implementation. Although marginally, the simulation time first decreases and then increases with co-operative threading in both the multi- and single processor environments. Of course, in case of co-operative threads, difference in simulation time between single CPU, and multi-CPU machines, is not significant, because in either case, the threads are never scheduled on the extra processors. With pre-emptive threading, simulation time increases as the number of threads in the model increases. This is because, for the AMRM model, level of description is the register-transfer(RT) level with a predominantly FSM models. There is little computation done in each state, that leads to higher overheads due to synchronization than the gains from parallelization of code using pre-emptive threads. This fact is evident from the simulation time for the multi-processor environment which are even worse than for the single processor environment due to higher synchronization overheads. In examples where the components of the system model are computationally intensive then the gain from parallelization of code using threads will be higher compared to the synchronization overheads imposed. Modeling systems at higher abstraction, for example at the level of tasks and functions, may be efficient with the pre-emptive thread implementation. Clearly, the SystemC implementation with co-operative threads is optimized for models where the amount of computation per cycle is relatively low compared to the cost of synchronization across, as is commonly the case with RT-level models.

This leads us to the strategy of concurrency **re-assignment**. Since, we do not want the designers to be concerned with the amount of computation per thread, and threading architecture of the simulation kernel, we suggest, this methodology of concurrency re-mapping or re-assignment. In this strategy, the computation in a thread will be enough to justify the creation of threads, and to overcome the synchronization overheads. This is very similar to the design of network protocol stack implementation using multi-threaded architecture as in [10].

3. Rationale for Concurreny Re-Assignment

As mentioned earlier, hardware systems are inherently concurrent. The system designers are used to describing their systems as a collection of concurrent modules. This allows for an easy intuitive, design, which facilitates synthesizability, but as shown in the previous section using this paradigm directly to drive simulation does not scale well to multiprocessor systems. Preemptive threads are required if we are to use multiprocessor systems to improve simulation time but they can impose a prohibitive context switching overhead and thus have to be judiciously used if they are to be efficient.

When considering a multiprocessor simulator we take the following into account:

- Consider a system described using concurrent processes (e.g. SC_[C]THREAD processes in SystemC) and assume that we associate a thread with each process. Since all threads must synchronize with the clock edge (so that modified signals are updated), threads have to wait for each other at the clock edge. This means that some threads will be idle while the slowest thread in the system finishes its workload for that cycle. This can result in a significant amount of CPU cycles not being utilized.

- Complex systems may have a large number of processes. Assigning a thread to each process (especially if the number of available processors is much smaller than the number of threads) means that there will be a significant amount of context switching. And considering that most designs are simulated for a large number of cycles, this may result in a significant overhead due to context switches.

So, intuitively, if we can first reduce the number of threads and then try to keep those threads as fully utilized as possible, we should be able to obtain better simulation performance in a multiprocessor environment.

Generally speaking, multithreaded systems fall generally in two categories: task-based or message-based [10]. The first binds processing elements (i.e. processors) to one or more tasks in the system and the tasks pass the necessary messages between them. The second binds a task to one message and *takes it through* the whole system. If we think,

for example, of SystemC signals dataflow as messages, we can view SystemC description of a system with concurrency aligned along the modules, as a task-based approach. Each SystemC process is assigned to a thread and signals are used to communicate between them (e.g. each thread processes incoming signals and updates outgoing signals). Rather than requiring a new modeling style by the designers, we seek to automatically convert a given task-based system description into a message-based description such that simulating it can be scalable to multiprocessor platforms. Scalability will be achieved if we can reduce the number of threads in the system and also expose any parallelism implicitly available in the description such that multiprocessor systems can take advantage of it.

4. Concurrency Re-Assignment Strategy

The strategy we propose attempts to do just that: extract a logical sequence for the processes describing a task-based system such that it can be restructured into a message-based system and then determine how these new threads can be executed relative to each other and themselves.

4.1 The Re-assignment Algorithm

Our Algorithm works as follows: Starting from a program dependence graph (similar to [4]), the algorithm has four steps.

Step 1. Divide processes into subprocesses slicing at synchronization points

Step 2. Analyze subprocess dependencies

Step 3. Schedule subprocess graphs

Step 4. Insert Synchronization events as necessary

The first step can be thought of as a slicing of the processes based on the current cycle and the signals updated. It has two sub-steps. First processes are subdivided at cycle boundaries (we need this to maintain cycle information based on *wait*() and *wait_until*() statement in SystemC) and the implicit sequencing between them is made explicit. Each resulting subprocess that is activated by more than one signal is then sliced based on each such signal (or signals if they are compounded by logical AND). We do this because once a signal has been updated we need to know which dependent subprocesses can potentially be activated but also want to isolate the effects of incoming signals.

At this point we have determined all the subprocesses in the system together with what signals each subprocess reads or writes and are ready for the next step. We use information from the system description (e.g. sensitivity lists, *wait_until*() statements, regular port read/writes and process variables in SystemC) to determine how subprocesses interact with each other. This is similar to doing a data flow analysis and will generate a graph similar to a PDG [4, 3]. If the graph is actually a collection of independent graphs then each such graph will later on be mapped to one thread of execution. We can now perform step 3: schedule (e.g. ASAP) each graph and thus get the sequence of subprocesses in the threads of execution.

The final step is needed to account for having multiple instances of the same thread executing simultaneously (i.e. multiprocessor simulator). We need to preserve the proper order of execution and can do this by reanalyzing the subprocess dependencies from a loop-carried point of view (e.g. each thread is viewed as the body of a loop). Any loop-carried dependencies need to be enforced by inserting a synchronization event to ensure that the threads execute those subprocesses in the proper order.

5. Experimental Results

We envision the above algorithm being implemented as a source-to-source transformation as a part of a compiler front-end. The implementation should accept the source code (e.g. SystemC) and generate C/C++ as its output retaining the functional correctness and original semantics. In our experiments we have verified the functional correctness by manual tests, however, how to verify the semantic equivalence of the newer version and the original model, is a research direction that we also intend to follow in the future. The newly threaded output can be compiled with any standard optimizing C/C++ compiler.

To illustrate our algorithm we have looked at two examples included in the SystemC 1.0.2. distribution: a pipeline example and a simple RICS-like processor. The examples were simulated on a dual Pentium II 300MHz machine. Since we wanted to use the same platform for both single and dual threaded runs we ran single-threaded scenarios by specifying that the thread be only bound to one processor.

The pipeline example is a straightforward example with five independent stages (i.e. each stage only depends on signals from the one before it). We manually applied our algorithm and then simulated 1,000,000 cycles in three scenarios: stages implemented as SC_METHOD processes, SC_CTHREAD processes, and, finally, using our reconstructed thread and present the results in Table 2. Data flow analysis has revealed that each pipeline pass was independent and such there was no need for synchronization (other than to make sure that the right number of cycles has been executed). This accounts for the difference between 8.20 sec and 7.67 sec (ideal 50% of the SC_METHOD run time). As we can see in this particular example we are very close to optimal speed up from going to a multiprocessor simulator.

	Wall Clock (sec)	User Time (sec)	System Time (sec)
sc_method	15.34	15.02	0.00
sc_thread	16.53	16.52	0.00
threaded	8.20	16.23	0.00

Table 2. Simulation time with Concurrency Re-assignment

However this is a rather atypical example where there is no cross-cycle interaction between processes and thus minimum perfomance hit from synchronization. Next we consider a more complex example of a RISC-like processor. Its description in SystemC consists of eight SC_CTHREAD processes and 30 connecting signals. In Figure 1, we show only some of those signals to help illustrate some of the modifications that were made to the original description.

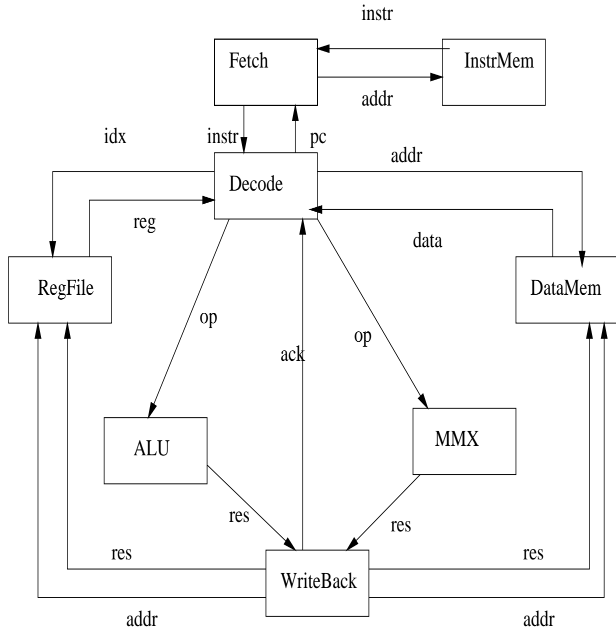

Figure 1. processes and connecting signals in a RISC-like processor

Figure 2 shows the threading structure after the concurrency reassignment. Note that DataMem1 and DataMem2 are both passes of the DataMem process but one is for data reading when invoked by the Decode process while the other is for data writing invoked by the WriteBack process. Similarly other processes have been *sliced* but we are only presenting here important ones.

Analysis of the data flow through the signals connecting the processes has also led to the necessity of two synchronization events and these are marked with dashed lines in the Figure 2. One is to prevent subsequent threads from fetching the next instruction until the next program counter has been resolved and the other to prevent register and data argument values to be read before previous values have been committed.

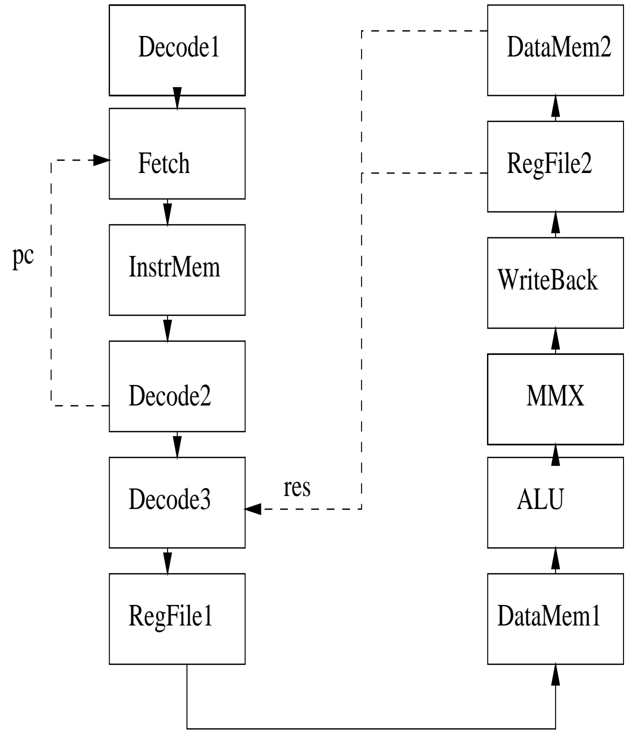

Figure 2. Re-assigned Process Flow

Table 3 shows the results from simulation run of the original and restructured models.

Simulation Results	Total Time (sec)	User Time (sec)	System Time (sec)
Original 1-threaded	12.75	12.68	0.02
Modified 1-threaded	13.22	12.83	0.25
Modified 2-threaded	9.87	13.25	2.53

Table 3. Simulation time with Concurrency Re-assignment in the RISC processor Example

We can see that the modified single threaded version is already faster than the original SystemC version. This is due to a couple of factors: first, fewer context switches (cooperative threads still have context switching overhead) as a result of the subprocesses being spliced together into a single thread. The other reason is that as opposed to our first example, we are not using most of the SystemC simulator runtime. In particular signaling was simplified in the

modified version: when a signal is written to the subprocesses dependent on it will be scheduled automatically instead of being polled each clock edge to see if their signal was activated. The results for the dual-threaded run show that thread execution could be overlapped (i.e. some parallelism was extracted) given the new thread structure and that resulted in better simulation performance but also that synchronization costs can be an issue.

Our last example is an MP3 decoder, which has been modeled in three different ways, as was the previous example. We just mention the data in Table 4, to show how concurrency re-assignment helped improve the simulation speed.

	Wall Clock (sec)	User Time (sec)	Kernel Time (sec)
Original 1-threaded	20.02	19.81	0.09
Modified 1-threaded	21.42	20.51	0.78
Modified 2-threaded	13.15	20.57	1.35

Table 4. Simulation time with Concurrency Re-assignment in MP3 decoder Example

6. Summary and Future Work

In this paper, we first showed that even when we replace the co-operative threading model used in most C++ library based hardware design environments, with pre-emptive threading model, one cannot exploit multi-processor based speed up. The reason is that the amount of computation per cycle in a typical hardware modeling style is too small to overcome the kernel level synchronization costs. As a result, one might ask of the designers to change their natural hardware modeling styles, to be able to speed up simulation. However, we do not want to burden a hardware designer with software engineering issues. Hence, we have presented an algorithm that takes a SystemC description of a hardware design (which is typically modeled using a task-based paradigm) and automatically restructure it into a dataflow paradigm that is more suitable for fast simulation on both single and multiprocessor architectures.

We have seen that improvements in simulation performance (often upto 2x) are possible for both the single processor and multiprocessor cases. We believe we can improve this even further. Further research is needed to better understand how to control synchronization costs, experiment with larger numbers of processors and explore ways of trading off cycle accuracy for simulation speed.

References

[1] CynApps Inc., http://www.cynlib.com. *Cynlib Reference Manual*.

[2] P. Garg, S. Shukla, and R. Gupta. Efficient usage of concurrency models in an object-oriented co-design framework. In *Design Automation and Test in Europe, Designers Forum*, 2001.

[3] D. Jackson and E. J. Rollins. A New Model of Program Dependencies for Reverse Engineering. In *Proceedings of the SIGSOFT conference on Foundations of Software Engineering*, New Orleans, December 1994.

[4] D. Jackson and E. J. Rollins. Chopping: A generalization of slicing. Technical Report CMU-CS-94-169, School of Computer Science, Carnegie Mellon University, 1994.

[5] D. Keppel. Tools and techniques for building fast portable thread packages. Technical Report UWCSE-93-05-06, Computer Science and Engineering, Univ. of Washington, May 1993.

[6] S. Liao, S. Tjiang, and R. Gupta. An Efficient Implementation of Reactivity for Modeling Hardware in Scenic Design Environment. In *Proceedings of the 34th Design Automation Conference*, pages 70–75, Anaheim, CA, June 1997.

[7] C. J. Northrup. *Programming with UNIX Threads*. John Wiley, 1996.

[8] OCAPI Website, http://www.imec.be/ocapi.

[9] Omitted for Blind Refering Process.

[10] D. C. Schmidt and T. Suda. The performance of alternative threading architectures for parallel communication subsystems. *Journal of Parallel and Distributed Computing*, submitted 1996.

[11] Synopsys Inc., http://www.systemc.org. *System C Reference Manual*.

[12] T. Wagner and D. Towsley. Getting started with posix threads. Technical report, Computer Science Department, Univ. of Massachusetts, Amherst, July 1995.