Title

Design for synthesis, transform for simulation : automatic transformation of threading structures in high-level system models

Authors

Savoiu, Nick
Shukla, Sandeep K.
Gupta, Rajesh K.

Publication Date

2001-12-18

Peer reviewed

Design for Synthesis, Transform for Simulation: Automatic Transformation of Threading Structures in High-Level System Models

Nick Savoiu
Sandeep K. Shukla
Rajesh K. Gupta

ICS

TECHNICAL REPORT

Technical Report # 01-57
(December 18, 2001)

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425

Information and Computer Science

University of California, Irvine

Design For Synthesis, Transform for Simulation: Automatic Transformation of Threading Structures in High-Level System Models

UCI-ICS TR # 01-57

Nick Savoiu
savoiu@ics.uci.edu

Sandeep K. Shukla
skshukla@ics.uci.edu

Rajesh K. Gupta
rgupta@ics.uci.edu

December 18, 2001

Center for Embedded Computer Systems
Dept. of Information and Computer Science
http://www.ics.uci.edu/~cecs/
University of California, Irvine
Irvine, CA

TABLE OF CONTENTS

UCI-ICS TR # 01-57 ..... *1*

1. INTRODUCTION *3*

2. BACKGROUND *5*

3. SYSTEMC CONCURRENCY MODELING *7*

4. ALGORITHM FLOW *8*

4.1. SystemC Analysis **10**

4.2. Process Analysis **11**

4.3. APU Generator **12**

4.4. Scheduler **14**

5. CONCLUSION *19*

6. REFERENCES *20*

TABLE OF FIGURES

Figure 1. SystemC Scheduler Flow 8

Figure 2. Algorithm Flow 9

Figure 3. Algorithm Pseudocode 10

Figure 4. Example of IF and FOR HTG Nodes 10

Figure 5. DMA SystemC Example 12

Figure 6. Controller APU Decomposition 14

Figure 7. Example of Process Slicing on a Control Signal 15

Figure 8. Handling of a Signal Hold/Drop 16

Figure 9. Determining Cycle Boundaries 17

Figure 10. DMA Example Final APU Flow 19

ABSTRACT

This paper presents a static transformation algorithm, for C++-based hardware models such as SystemC models. This algorithmic transformation changes the threading structure of the models and generates efficient C++ code with pre-emptive threading for efficient simulation on multi-processor systems. Efficient modeling for simulation and modeling for synthesis seems to be competing goals in the context of C++ based modeling paradigms, because for synthesis, the design should be partitioned according to the targeted hardware units and their interconnections, and concurrency is aligned along unit boundaries. However, for simulation performance, such a model may contain many concurrent modules, implemented by many user-level threads, which is an artifact of the C++-based concurrency modeling mechanism. It has been shown in previous work, however, that multi-threading is not necessarily a simulation performance hindrance. In fact, a proper choice of multithreading, especially in huge simulation models with I/O bound computations, is necessary for simulation efficiency. However, since the existing C++-based modeling frameworks employ user level thread packages, even this necessary threading cannot take advantage of the multi-processors availability, because user-level threads are transparent to operating system kernel. We solve this problem by accepting synthesis targeted SystemC models, and compiling them into multi-threaded simulation models, with kernel-level threads, resulting in faster simulation on multi-processors, as well as on single processors. Concurrency alignment in our resulting code is usually along the dataflow through the model. In our past work we have shown that simulation is faster after such concurrency re-assignment and here we give a foundation for implementing such an algorithm. This work has similarity with Quasi-Static Scheduling work in the literature, however, the aim and context are different, and so is the basis for the algorithm design.

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. Moreover, as designers progress more towards using system level models, for architectural exploration, they seek behavioral synthesis tools [16], which would allow them to synthesize hardware from the

same models. When considering simulation efficiency, it is common to rely on concurrency enhancements, and concurrency management techniques (using multiprocessor hardware, and multithreading 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] and others [14][15], model concurrency using either function calls or thread packages. The thread library employed can be cooperative or preemptive [8]. However, most existing system level simulation and modeling frameworks have implemented their kernels using application-level threading libraries (cooperative threads) [6]. These user-space threads are transparent to operating system kernels and hence cannot take advantage of multiprocessor systems for performance improvement.

The other dimension to the simulation performance problem is that hardware concurrency, most naturally, is aligned along units or modules. Especially if the system level model is to be used for synthesis, it is important to model the hardware units and their interconnections as planned for the real hardware, since behavioral synthesis tools are not sophisticated enough to create optimized hardware from any functional model. In other words, system designers, due to natural design intuition, as well as for synthesizability, and considering the language constructs provided in the existing language frameworks, build concurrency into the model by providing threads for the modules or units representing concurrent hardware modules/units in the planned design. Trade-offs between function calls, and threading, and how to cluster modules together, to obtain an efficient simulation model are discussed in our previous work [2]. In our experiments, we found that, due to threading packages used in the simulation kernels, as well, as designers' choice of mapping functionalities into threads, a manual design trade-off is not easy to achieve. We therefore sought automatic transformations of models that are created for synthesis purpose, into efficient C++ code, with pre-emptive threading, so that they can be efficiently simulated on multi-processor systems. In [1] we showed by manual transformations of SystemC models, that such code generation

indeed benefits the simulation performance, in some cases up to 100%. In this paper we describe automatic code generation for efficient simulation via model transformation based on conversion of threading structure.

There is an extensive body of literature [17][18][19][20] on software synthesis from concurrent system description. However, most of these are aimed at synthesizing embedded software that has small code size, which is completely or partially sequentialized and then scheduled. The idea there is to synthesize software for embedded devices from high-level specification. Such synthesis usually would result in defining tasks, and their schedule, or at least, quasi-schedule, such that data dependent choices in schedule is done in run-time, and rest of the tasks are scheduled at code generation time.

However, our context is different in few ways. First, we are not aiming at software synthesis for embedded systems, with real-time constraints, and hence we are not scheduling in the same sense. Although, re-architecting the threading in the software, keeping the same functionality and semantics, is similar to software synthesis, but it is closer to code transformation than synthesis. Second, our models are already in C++ code, and hence, we know the sequential order of the code in each thread, and we do not reorder any of the sequential code in a single thread. We just map the sequential activities in different threads onto a lesser number of threads, some times a single thread, which is equivalent to complete sequentialization. Third, these previous approaches modeled the specification as Petri-net of some kind, and sequentialized the tasks by unrolling the Petri-net the required number of times, and then generate C-code corresponding to the unrolled net. We parse and generate a Hierarchical Task Graph [4] models, from the given model, and use traversal techniques to generate an efficient threading structure in the generated code.

2. BACKGROUND

As mentioned earlier, hardware systems are inherently concurrent. The system designers are used to describing their systems as a collection of concurrent modules, especially when targeting synthesis from the model. This allows for an easy intuitive, design, which facilitates synthesizability, but as shown in [2] using this paradigm to

directly 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.

A multiprocessor simulator poses different issues than a single processor one. 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. Also, 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 active as possible, we should be able to obtain better simulation performance in a multiprocessor environment.

In [9] it was shown that a multi-threaded implementation of a communication protocol stack can be threaded in two different ways: task-based or message-based. The first binds threads to one or more tasks in the system and the tasks pass the necessary messages between them. The second binds a thread to one message and takes it through the whole set of computation to be performed on a single message. If we think, for example, of SystemC dataflow through signals as messages, we can view SystemC description of a system with concurrency aligned along the modules (i.e. 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 imposing 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.

3. SYSTEMC CONCURRENCY MODELING

For hardware systems, a 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 [7]. For modeling concurrency SystemC provides two types 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 processes are more general form of the synchronous processes that can be used to model any kind of circuit.

In SystemC, asynchronous blocks are implemented using function calls. The synchronous and asynchronous processes are implemented using Quick thread library in Unix[6] and fibers (i.e. light-weight self-scheduled threads) in Windows NT[21]. Both provide a cooperative thread framework, where a thread yields control to another thread at will. A thread cannot preempt 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.

To better understand why the current SystemC[12] implementation does not scale to multiprocessor simulation consider the simulation kernel flow in Figure 1.

In the *Initialization* phase all processes (except SC_CTHREAD processes) are executed in an unspecified order. In the *Evaluate* phase as long as processes ready to run are available they are selected and resumed. This can cause additional processes to become ready to run in this phase due to immediate event notifications. When no such ready to run processes exist, the simulator enters the *Update* phase were signal values are updated to the values computed in the evaluate phase. At this point, if there are pending delayed notifications, the simulator determines which processes are ready to run due to the delayed notifications and returns to the *Evaluate* phase.

Otherwise, if there are no timed notifications either, simulation is finished. If there are timed notifications the simulator advances the current simulation time to the earliest pending timed notification and reenters the *Evaluate* phase.

While this approach works for single processor simulation the fact that process evaluation (i.e. execution) is relegated to the Evaluate phase has the following repercussions. First, even if preemptive threads are assigned to each process each process executed in the *Evaluate* phase will require a context switch and potentially a significant overhead. Furthermore the amount of code run in a cycle is typically small. Also if the number of processes ready to run in the *Evaluate* phase is less than the number of processors available to the scheduler poor utilization of the available processors results.

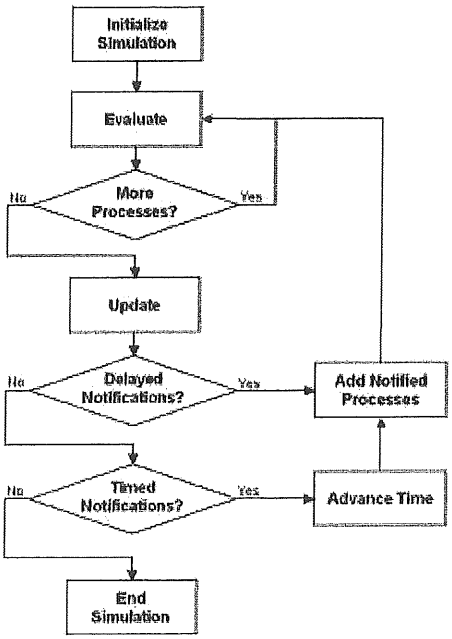

Figure 1. SystemC Scheduler Flow

This leads us to the strategy of concurrency reassignment to help lower the number of threads and also context switches. 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 a methodology of concurrency remapping or reassignment. In it 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 [9]..

4. ALGORITHM FLOW

In Figure 2 we present the overall flow of the algorithm and some of the data structures created while running it. The SystemC description is analyzed and several data structures are built for later consumption. The SystemC processes present in the description are identified along with information about their type and interconnections. These processes are then divided into indivisible pieces (i.e. from the SystemC scheduler's point of view) called atomic process units (APUs). The APUs are connected using the inherent C++ sequencing and the sequencing imposed by control signals present in the description. Scheduling and sequentializing this APU graph we define execution paths through the system that can be then be mapped to threads. As a final step information collected initially about the SystemC description is used now to generate a new, SystemC-like description that executes using the new threads just created. Figure 3 contains pseudocode that outlines the steps we just presented. Although our algorithm is tailored for the way SystemC is used to describe systems, it could easily be adapted for other C++-based high-level modeling frameworks.

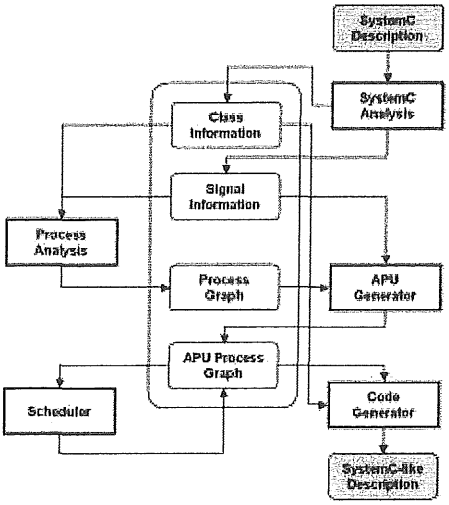

Figure 2. Algorithm Flow

In following subsections we will discuss the algorithm steps in more detail with the use of a SystemC example. In our previous work we have shown results for a simplified RISC processor and an MP3 decoder. However these examples are too complex to serve as a good example in the context of this paper. Therefore we present a simple Direct Memory Access (DMA) system to help exemplify the various transformations in our algorithm. The SystemC implementation of the DMA example has 4 modules implemented as SC_CTHREAD processes. Figure 5(a) shows the high-level interaction between the modules. The controller provides the DMA module with the transfer parameters and awaits acknowledgement that the transfer is complete. The DMA module receives these parameters and then proceeds to transfer data from one memory module to the other by performing alternative read and write operations on the two memory modules. If we chose to implement this

example by simply creating a thread for each process and then assigning it to a processor, the controller process thread will be mostly idle resulting in poor utilization of that processor.

```
for each SystemC class
 for all functions in class
  create HTG graph for all class functions
 identify processes and determine types
 determine input/output ports and identify port operations

extract netlist information
connect HTGs through signals

repeat until no changes
 for each HTG
  do constant propagation using test bench information
  remove dead code
 propagate across signals and function calls

for each process
 identify atomic process units

generate and schedule APU graph

regenerate SystemC object source code
sequentialise schedule and generate C++ code for it
generate thread management code
```

Figure 3. Algorithm Pseudocode

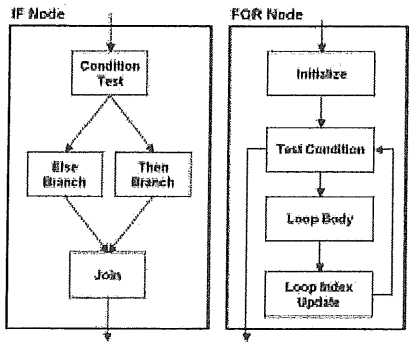

Figure 4. Example of IF and FOR HTG Nodes

While the information in the high-level view is enough for the human understanding of the interaction between these modules further system analysis is needed if we want to automatically extract the system execution path for concurrency reassignment.

4.1. SystemC Analysis

To be able to restructure the SystemC code describing a system we must first understand the structure and interaction that occurs within each such system. As the first step in our algorithm we parse the SystemC description and build several data structures. These data structures serve a dual purpose. First, they enable the system analysis required to do the source-level transformations required for concurrency reassignment and if that is not feasible to suggest possible code changes that will allow the automation of these transformations. Second, once the analysis and transformations are complete, they should allow for code to be generated that resembles the input code as closely as possible. This gives us design closure and thus allows a designer to easily identify his code in the output code and be able to make changes to the source code that would have a meaningful impact on the output code.

To that end we chose as an intermediate representation a class hierarchy underlying Hierarchical Task Graphs (HTGs) [6] representing class functionality. The class hierarchy is needed because some of our transformations (e.g. signal connectivity, constant propagation) require whole-program analysis while the underlying HTGs guarantee that sufficient source information is retained for the code generation step. For example, the source statements making up an *if-then-else* conditional construct or a *for* loop are aggregated into compound HTG nodes as shown in Figure 4. The structural nature of the HTG nodes means that we retain information about both the *then* and *else* branches and also make a clear distinction of the parts that make up a loop. The latter is beneficial for isolating loop body statements (e.g. functionality in an SC_[C]THREAD process is often encapsulated in an infinite loop) while the former is useful in slicing operations.

An important part of the SystemC description is the signal definition and connection. It defines the signals present in a system and the processes that they interconnect. SystemC signals do not have a direction attached to them. We will have to determine that later once we identify how processes use each signal.

For our example the class hierarchy is straightforward: one class for each component in the high-level system view and each class contains a single clocked thread process. The signals present in the system are those identified in Figure 5(a) and, as mentioned above, they are initially undirected.

4.2. Process Analysis

Once the class and signal information was generated we can analyze the source code to identify the processes present in the SystemC description.

One important characteristic of a process is its type. To identify that we locate the calls in the HTGs that register these functions with the SystemC simulation kernel (i.e. calls to SC_METHOD(), SC_THREAD(), and SC_CTHREAD()). Type information (i.e. method, thread or clocked thread) is used to determine how a process is divided into atomic units and also when cycle boundaries are being crossed.

Also at this time we determine how processes communicate with each other. The signal information collected previously is further refined into the process graph in Figure 5(b). Signals are analyzed for direction of communication (through the identification of port read and writes in the process HTGs) and the process source is used to determine if a signal is merely a *data* signal or a *control* signal. A *data* signal is a signal that only provides a way for processes to exchange data whereas a *control* signal is used to determine when a process that depends on another process can resume its execution. Control signals can also optionally provide data to the target process but SystemC 2.0 discourages that through the introduction of *events*. They allow inter-process control but no data exchange. Events are functionally similar to our control signals.

In Figure 5(b) data signals are represented by normal lines and control signals represented by bold lines. For example, the controller process uses the *Start* signal to notify the DMA process that it should start a new transfer and also waits on the *Done* signal for acknowledgement that the transfer is complete. What remains to be determined is in which order are these signals activated. Intuitively, if we follow this flow of control signals, we get an *execution path* through the system.

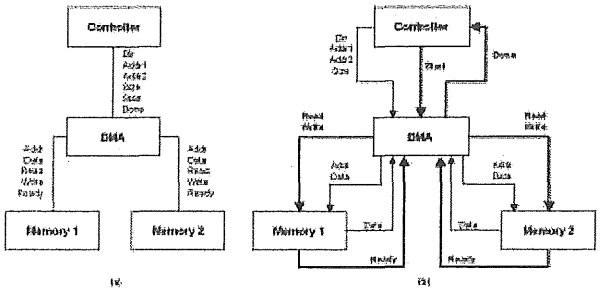

Figure 5. DMA SystemC Example

But the process graph still has too coarse a granularity to allow us to effectively determine and restructure the graph for better concurrency. Therefore we need to break up the processes into atomic process units.

4.3. APU Generator

An APU represents the part of a process that will execute within one invocation in the *Execute* phase of the SystemC simulation kernel. It can be thought as the smallest piece of a process that the scheduler kernel is aware of, hence the *atomic* quantifier.

To determine the APUs of a process we need to take into account all the synchronization points of a process. A synchronization point is in general marked by the presence of a *wait()* statement in the process source code. However the process type affects how it is divided into APUs. Since a method process is executed until it finishes for each invocation, it cannot contain any calls to *wait()* and thus the entire process is a single APU. For thread and clocked thread processes we have to locate all calls to *wait()* statements and divide the process according to those. We do this by first slicing the process once for each incoming control signal. We then take the resulting process code and divide it into APUs using the remaining synchronization points as the dividing elements. The synchronization calls are removed from the code but the APU is associated with that enabling signal or signals. If the process source contains loops, we generate APUs only from the loop body code.

Currently we assume that synchronization points are not present in the branches of *if* statements. If such points are allowed inside if statements the problem of dividing processes into APUs can become intractable because of the potential for deadlock. There are however some *if* statements that can be currently handled by our algorithm. Consider the code in Figure 7(a). It shows the implementation of one of the memory units. The code checks if control signal *read* is active. If so, the *then* branch is taken which contains the code needed to perform a read operation. Otherwise the *write* signal is checked and if true the actions needed to perform a write are taken. This seems like an example that would require restructuring since it contains synchronization points within the branches of an *if* statement. However, to determine the statements that are needed for a particular APU we must first slice [7] the code based on each enabling control signal that reaches the process (this might be a subset of the list of signals a process is sensitive to). Let us assume, for example, that the enabling signal is *read*. After slicing the memory unit's process will only consist of the section highlighted in Figure 7(b). As we can see, that code segment has no *if* statements and it can therefore can be easily divided into APUs at the synchronization points. Similar results are obtained if slicing is done on the *write* signal. Therefore our restriction regarding the placement of synchronizing events applies to the process slices rather than the source process code.

Since the multiple slicings of a single process can result in duplicate APUs we only retain unique APUs. We can finally create the APU graph for a process while taking into account the control flow imposed by the original C++ code. Consider the example in Figure 6(a) showing pseudo-SystemC code for the controller unit and the resulting APU graph in Figure 6(b). It was created after the code was divided into 3 APUs and they were connected by inherent sequencing provided in the C++ code. As a final pruning of the APU graphs any APUs that are not connected to the graph are removed.

The last step in our augmentation of the process graph is to do something similar to data dependence analysis in a regular software compiler. However, instead of statements we have APUs and instead of variables we have control signals. This analysis allows us to determine the inter-APU dependences based on such signals. We use the signal information we collected in the first step and add control signal dependence arcs to the APU graphs.

It seems that the use of global and module-global variables would imply that we should also account for inter-APU dependences introduced by these variables. However, the SystemC functional description[2] indicates that the simulation kernel neither enforces nor guarantees an order of execution for ready-to-run processes. Thus, if processes depend on global or module-global variables for synchronization, the results can be undetermined and as such we can ignore such dependences.

4.4. Scheduler

At this point in the algorithm we have the fully APU graphs. The next step is to schedule the APUs in the graphs according to these dependences. But before we can do that we need to do several preliminary steps.

The first one concerns synchronous thread processes (i.e. SC_THREAD and SC_CTHREAD processes). These

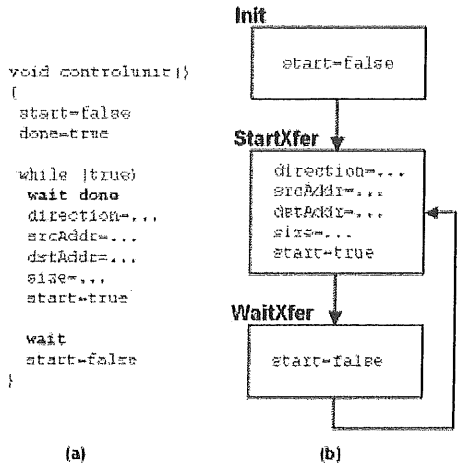

Figure 6. Controller APU Decomposition

are generally modeled as an infinite loop encompassing the actual functionality of the process. We can use the structural nature of our HTG information captured in the first step to identify the loop bodies of such processes thus isolating the process functionality. Only APUs contained in this loop body will be used during scheduling.

Once this is done we apply an As-Soon-As-Possible (ASAP) scheduling algorithm to the APU graph. The result of this will be an APU graph scheduled according to the dependences among APUs. Note that if the system description contains several independent subsystems then the actual result of scheduling will be several independent APU graphs. The graphs now represent parallelized versions of the schedules. However since we want to finally generate a thread of execution (i.e. sequential) we have to sequentialize the tree.

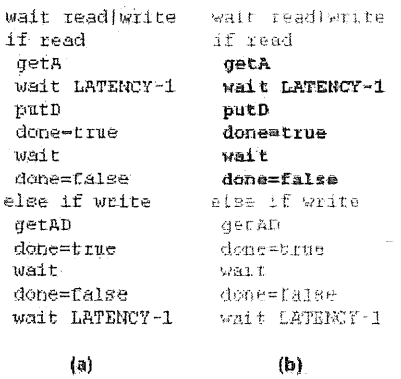

Figure 7. Example of Process Slicing on a Control Signal

In general sequentializing is a straightforward process that traverses the graph using a breadth-first search. However there are some things that have to be taken into consideration to make sure the resulting code is correct and corresponds to the SystemC simulation semantics.

The diagram in Figure 8(b) represents a fragment of the APU flow diagram in Figure 10 and it is the result of the code in Figure 8(a). To make Figure 10 more readable certain APUs (i.e. those deemed not critical in showing the final path of execution through the system) were not included. However one such APU is an example of a common occurrence when modeling hardware: at a certain point in the execution of a process a signal is held (high or low) to indicate the occurrence of a specific event (e.g. such as the availability of data on the memory data bus) and then it is dropped to indicate the end of the event. Such is the case with the *Done* signal for the memory modules. Once the data is available on the data bus a *Done* signal is raised and held for one cycle as in Figure 8(b). To understand how scheduling would handle this we have to consider the dependencies that are present.

DropDone has a flow dependence (i.e. resulting from the original C++ code) on *MemDone* while *GetD1* has a control signal dependence on *MemDone*. Therefore both will be scheduled after *MemDone* and possibly in the same scheduling step. However, considering the SystemC scheduling semantics (i.e. all processes in a delta cycle execute and then update the signals they have changed) correct behavior is obtained only if the sequentializing step favors control signal dependences over flow dependences that are present at the same level in the graph. In this case a sequence of *MemDone*, *GetD1*, *DropDone* leads to correct simulation results.

After sequentializing we have obtained the actual flow through the system's APUs that allows a sequential execution of the simulation. Figure 10 shows the flow of execution after scheduling and sequentializing our DMA example. The graph nodes represent the APUs that processes have been divided into (some APUs have been omitted for brevity and clarity). The solid arcs between APUs reflect sequencing imposed by the input description (i.e. flow dependences). The dotted lines reflect sequencing imposed by the control signal analysis performed by our algorithm (i.e. control signal dependences) and are labeled with the signal that generates them. Finally the numerically labeled lines represent the final sequence of APUs that make up the path of execution through the system. Also notice that the *Init* APUs in each process could actually be executed in any order since the SystemC simulation kernel does not impose any order on processes that are simultaneously ready-to-run. A randomization step could be introduced to parallel the randomization option available in SystemC 2.0.

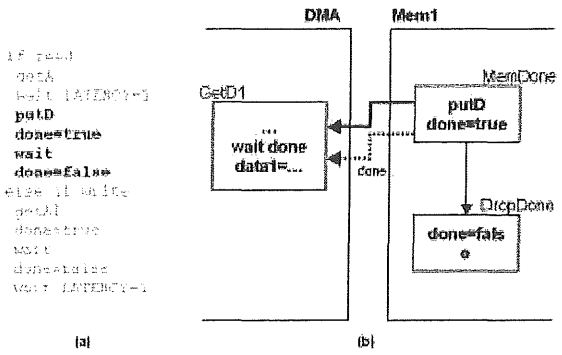

Figure 8. Handling of a Signal Hold/Drop

If the system has a global clock and cycle information is necessary we can compute cycle boundaries as follows. We have two classes of APUs based on the process type they originated from. If the process type was method or thread such an APUs is on a clock if one of the control signals that it depends is *clock*. However is the APU

originated from a clocked thread process it will always be on a cycle boundary since clocked threads are only sensitive to the *clock* signal. Figure 9 shows how such boundaries would be determined in a scheduled APU graph.

Code Generator

The final step generates output code that is similar to the input SystemC description. However some preprocessing of the scheduled APU graph is required. Recall that in SystemC thread and clocked thread process functionality is enclosed in an infinite loop (e.g. for example the dotted line from *MemDone* to *WaitStart* in Figure 10 shows us the back arc of the infinite loop in the implementation of memory module 1). This loop is needed so that the memory module is ready to service another request after finishing the previous one. However since we have restructured the system flow to explicitly call the *WaitStart* APU when its controlling signal is activated the infinite loop construct is no longer need. Similarly we can remove the infinite loop construct from the other memory module and the DMA module. However we cannot always do that. For example, for the Controller module we have to leave the loop in place since it lacks any incoming control signal (the process gets its signal values from a test bench, for example a file, rather than incoming signals).

We regenerate the SystemC class descriptions needed to build the simulator objects (from the information we retained in the first step). We also circumvent the SystemC simulator kernel: once the simulator has setup the objects and passes control to the *sc_main()* function we only relinquish control once the simulation is completed. That means that the code generator is responsible for generating the code that defines the main execution routine. It also has to generate code for managing thread execution and synchronization in the

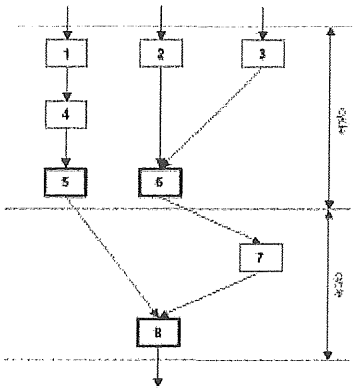

Figure 9. Determining Cycle Boundaries

newly redesigned system.

As the code generator has to generate code for the newly defined execution path the question arises of how much of it can be "packaged" into a thread that can then be multiply-spawned. In our DMA example the whole path cannot be used because we were unable to remove the infinite loop encompassing the controller functionality (due to signals having their values obtained from a test bench). Therefore one or more smaller sections of the execution path could be candidates for threads. As in traditional software compilation we give special attention to any loops present in the description. Since applications spend most of their execution time in loops so improvements to loop runtimes translate in overall runtime improvements. The DMA has such a loop (i.e. the loop that iterates until all the data has been transferred). We can try to "pipeline" the execution path that lies inside the loop (bold line segments 7 through 16 in Figure 10). Since multiple instances of the same thread will be executing simultaneously we need to preserve the proper order of execution. We can do this by reanalyzing the subprocess dependences from a *loop-carried* point of view (because the thread code is in effect the body55 of a loop) and determine if any such *loop-carried* dependences exist on control signals. If they do, we need to enforce them by inserting a thread synchronization event between any APUs involved in dependences to ensure that the threads execute those APUs in the proper order. In our example there are no such control signal dependences so the threads can run fully parallel. This will result in almost linear improvement in the loop execution time but in general that will not be the case since some synchronization will be necessary.

6. CONCLUSION

In this paper we detail our algorithm for automatic transformation of C++ based system level models, which are designed for synthesis, with hardware structure in designers mind. This transformation results in a semantically equivalent C++ code, which shows better simulation performance than the original SystemC code. The generated C++ code is threaded with preemptive threads so that they can simulate even faster on multi-processor systems. Our algorithm, is based on constructing Hierarchical Task Graphs from SystemC code and traversing the resulting

graphs. Unlike the software synthesis based on scheduling or quasi-scheduling, we can generate a multi-threaded program since we want to exploit multiprocessor simulators. The generated program is validated against the original code by extensive testing. However, future work include an analytical proof of their semantic equivalence, applying theories of serializability of concurrent transactions.

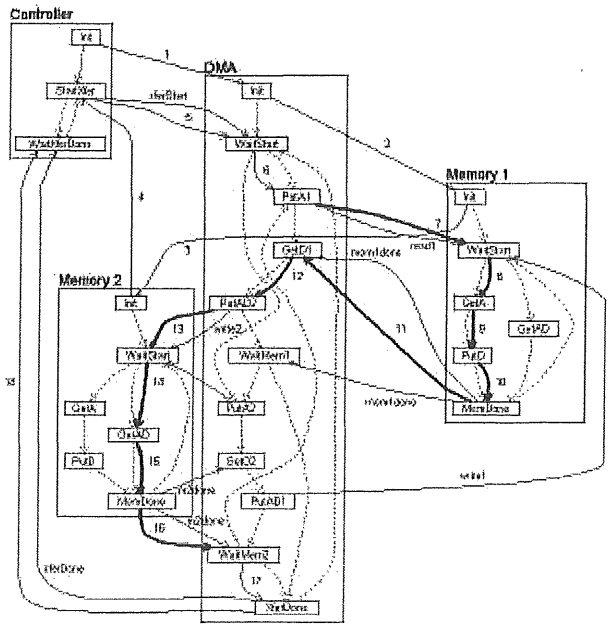

Figure 10. DMA Example Final APU Flow

6. REFERENCES

[1] N. Saviou, S. Shukla, R. Gupta, "Automated Concurrency Re-Assignment in High Level System Models for Efficient System Level Simulation", UCI-ICS Tech. Report No. 01-51, September 2001, Accepted for presentation at the Design Automation and Test Conference (DATE 2002), March 2002.

[2] P. Garg, S. Shukla, R. Gupta, "Efficient Usage of Concurrency Models in an Object Oriented Co-Design Framework", In the Proceedings of the Design Automation and Test in Europe (DATE'01), Designers Forum, Munich, Germany, March 2001, IEEE Computer Society Press.

[3] CONCUR PROJECT-UCI-ICS, http://www.ics.uci.edu/~skshukla/concur/concur.html.

[4] 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.

[5] 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.

[6] D. Keppel. Tools and techniques for building fast portable thread packages. Technical Report UWCSE93-05-06, Computer Science and Engineering, Univ. of Washington, May 1993.

[7] 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.

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

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

[10] M. Sgroi, L. Lavagno, Y. Watanabe, and A. Sangiovanni-Vincentelli. Quasi-static scheduling of embedded software using equal conflict nets. In Proceedings of 20th International Conference on Application and Theory of Petri Nets. ICATPN '99, June 1999.

[11] Synopsys Inc., http://www.systemc.org. SystemC 2.0 Functional Specification

[12] Synopsys Inc., http://www.systemc.org. SystemC 2.0 User's Guide

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

[14] CynApps Inc. http://www.cynapps.com.

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

[16] G. Economakos, P. Oikonomakos, and I. Panagopoulos. Behavioral Synthesis with SystemC. http://www.systemC.org

[17] Bill Lin. Software Synthesis of Process Based Concurrent Programs, In Proceedings of Design Automation Conference, 1998.

[18] S. A. Edwards. Compiling Esterel into Sequential Code, In Proceedings of Design Automation Conference, 2000.

[19] J. Cortadella, A. Kondratyev, L. Lavagno, M. Massot, S. Moral, C. Passerone, Y. Watanabe, and A. Sangiovanni-Vincentelli. Task Generation and Compile-time Scheduling for Mixed Data-Control Embedded Software, In Proceedings of Design Automation Conference, 2000.

[20] M. Sgroi, L. Lavagno, Y. Watanabe, and A. Sangiovanni-Vincentelli. Quasi-Static Scheduling of Embedded Software Using Equal Conflict Nets. In Proceedings Application and Theory of Petri Nets 1999.

[21] Microsoft Developer Network. http://msdn.microsoft.com/