Efficient Simulation of Synthesis-Oriented System Level Designs

Nick Savoiu Sandeep K. Shukla Rajesh K. Gupta
Center for Embedded Computer Systems
University of California, Irvine
{savoiu, skshukla, rgupta}@cecs.uci.edu

ABSTRACT
Modeling for synthesis and modeling for simulation seem to be two competing goals in the context of C++-based modeling frameworks. One of the reasons is while most hardware systems have some inherent parallelism efficiently expressing it depends on whether the target usage is synthesis or simulation. For synthesis, designs are usually described with synthesis tools in mind and are therefore partitioned according to the targeted hardware units. For simulation, runtime efficiency is critical but our previous work has shown that a synthesis-oriented description is not necessarily the most efficient, especially if using multiprocessor simulators. Multiprocessor simulation requires preemptive multithreading but most current C++-based high level system description languages use cooperative multithreading to exploit parallelism to reduce overhead. We have seen that, for synthesis-oriented models, along with adding preemptive threading we need to transform the threading structure for good simulation performance. In this paper we present an algorithm for automatically applying such transformations to C++-based hardware models, ongoing work aimed at proving the equivalence between the original and transformed model, and a 62% to 76% simulation time improvement on a dual processor simulator.

Categories and Subject Descriptors: B.6.3 Design Aids---Simulation
General Terms: Algorithms, Performance, Design.
Keywords: System-level Design, Simulation, SystemC.

1. INTRODUCTION
Simulation efficiency is generally one of the critical aspects of a design model and that can be especially true in design space exploration. With the advent of System-on-Chip designs a need for efficient system level models arose and C++-based high-level modeling frameworks (e.g. SystemC[14] and others[1][9]) were developed to address that need. As designers progress towards system level models, even for architectural exploration, they seek behavioral synthesis tools[2] that allow them to simulate and synthesize hardware from the same models. This requires models that are efficient for both synthesis and simulation. Since hardware designers usually develop models with synthesis (i.e. implementation structure) in mind we want to improve simulation speed without imposing on designers a specialized modeling style just for simulation.

Most hardware designs have some level of concurrency and therefore some inherent parallelism. When considering simulation efficiency, it is common to rely on concurrency enhancements and management techniques (i.e. multithreading, multiprocessing) for improved performance. Cooperative multithreading currently seems to be the choice for the simulation kernels of most modeling frameworks as it provides efficient implementations due to low runtime overhead. However, as we seek to further improve simulation performance (by exploiting the inherent design parallelism) through the use of multiprocessing we must from cooperative to preemptive multithreading in the simulation kernels. Our previous work[4] has shown however that simply switching simulation kernels from cooperative threading to preemptive threading does not improve simulation performance but rather degrades it. This is due to potentially high threading overheads if the threading structure of the model is not judiciously created. Further insight into the structure of models was needed to improve multiprocessor simulation performance. We started from the fact that, if we consider SystemC processes to be tasks and dataflow through signals to be messages passed between them, then such a description can be viewed as a task-based[12] model. It was however shown, in the context of communication protocol implementation, that for multithreaded execution, a message-based model would be more efficient but this requires a structural change of the model as to align it to the new threading structure. As hardware designers are used to the task-based design style, first because it is a natural way of modeling concurrency and second since synthesis tools are also targeted to suit such a modeling style, we do not want to impose a new modeling style that is to be used for efficient simulation only. But can we transparently convert a given task-based SystemC description into a message-based description such that simulating the transformed version can improve multiprocessor simulation performance? In [11] we showed what transformations are needed and that such restructuring indeed benefits the simulation performance. In this paper we describe how to automatically apply those transformations and generate code for efficient multiprocessor simulation while also giving some insight into the ongoing work aimed at proving the semantic equivalence of the transformed models to the original models.

2. ALGORITHM DESCRIPTION
Intuitively the transformations we developed identify new execution paths through a SystemC description of a design such that they are fewer and have longer execution times between synchronization points than the execution paths defined by the SystemC simulation kernel for the same design. This should help

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISSS '02, October 2-4, 2002, Kyoto, Japan.
Copyright 2002 ACM 1-58113-562-9/02/0010…$5.00.

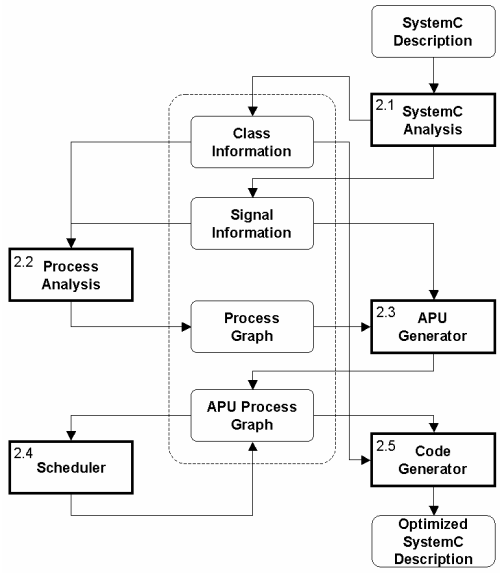

Figure 1. Algorithm Flow

control the runtime overhead due to preemptive threading and thus provide efficient execution of the simulation on a multiprocessor simulator.

In Figure 1 we present an overview of the algorithm. The first step is to analyze the SystemC description and extract the information needed to restructure the design. Processes and signals present in the description are identified along with information about their types and interactions. The processes are then decomposed into indivisible pieces called Atomic Process Units (APUs). The APUs are further structured into APU graphs using the inherent C/C++ sequencing and also the sequencing imposed by control signals present in the design description. We then apply scheduling and sequentializing algorithms to the APU graphs defined above and thus define new execution paths through the system that can be then be mapped onto threads. As a last step we generate a semantically equivalent system description that uses the newly created threads.

Figure 2 outlines in pseudo code the steps we just presented. Although our algorithm is tailored for SystemC, it could easily be adapted for other C++-based high-level modeling frameworks.

We illustrate the steps in our algorithm using a Direct Memory Access (DMA) example model. The SystemC implementation has four modules implemented as SC_CTHREAD processes and Figure 3(a) shows the high-level interaction between the modules.

If we implement this example by simply creating a thread for each process, multiprocessor simulation performance would suffer from high overhead due to frequent thread synchronization (i.e. on each cycle). We therefore need to restructure the threads to better fit a multiprocessor environment. For this we need to refine our high-level view in Figure 3(a) to better understand the interaction between these modules. This will allow us to automatically extract the system execution paths for concurrency reassignment[11].

2.1 SystemC Analysis

For that we must first understand the design's structure and the interactions within it. Therefore we need to represent the

```
for each SystemC class
 for each function
  create HTG graph
 identify processes and determine types
 for each process
  determine input/output signals
  identify signal activations/requests

for each signal
 determine process connectivity
 determine communication direction

for each process
 divide HTGs into APUs

perform source code optimizations
(i.e. constant propagation, dead code removal)

generate and schedule APU graphs
generate SystemC object definitions
sequentialize schedules and generate C++ code
determine thread interactions
generate management code
```

Figure 2. Algorithm Pseudo-Code

information collected with two major tasks in mind: source-level code transformations (needed to restructure and reassign concurrency) and code generation (containing complete system functionality and resembling the input code as closely as possible). The latter is particularly important since a designer should be able to easily identify his code in the output code for design closure. To satisfy these requirements we represent the source SystemC code as a class hierarchy with underlying Hierarchical Task Graphs (HTGs)[5] representing functionality for each class. The class hierarchy allows us to perform whole-program transformations (e.g. signal connectivity) while the underlying HTGs guarantee that sufficient source information is retained for source-level transformations and code generation.

An important part of the SystemC description are signal definitions and port connections as they define processes interconnections. SystemC signals are undirected so we have to determine the direction of communication by analyzing the process code.

For our DMA example, the class hierarchy is straightforward: one class for each component in the system containing a single SC_CTHREAD process. As for the signals present in the system, they are identified in Figure 3(a) as undirected.

2.2 Process Analysis

Once the class hierarchy and HTGs are created we can analyze the source code to identify all the processes and their properties (types in particular). The types can be identified from the callbacks registered by the constructors of each class which define them as SC_METHOD(), SC_THREAD(), or SC_CTHREAD(). The type information later used to divide a process into APUs. We also determine how processes communicate with each other. The signal information previously collected is further refined into a process graph as shown in Figure 3(b). Signals are analyzed for their direction of communication and the process source code is used to determine if a signal is a data signal or a control signal. A data signal allows processes to exchange data whereas a control signal is primarily used to determine when a dependent process can resume its execution. In our example, Figure 3(b) has data signals represented by normal lines and control signals represented by bold lines. By analyzing the source code we can

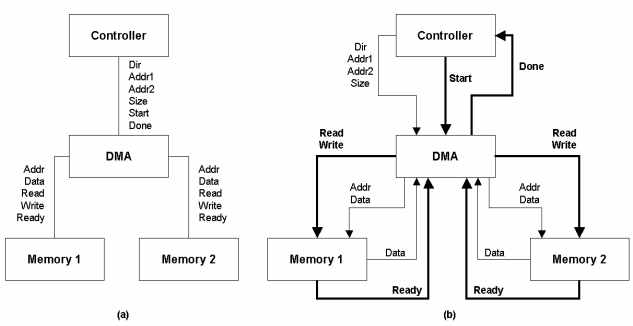

Figure 3. DMA SystemC Example

determine that the Controller process uses control signal *Start* to notify the DMA process that it should start a new transfer and also waits on the control signal *Done* for acknowledgement of transfer completion. What remains to be determined is in which order are these signals activated. Intuitively, if we follow the control signals, we follow the execution paths through the system.

But the process graph is still too coarse to allow us to effectively determine the execution paths and restructure the graph for better concurrency. In the original SystemC description two processes can have a sizeable set of signals to handshake on. We divide processes into APUs to such reduce interprocess dependences. By identifying the APUs for each process we seek to refine the interaction between processes by reformulating it at the APU-level. This has a two-fold benefit: a reduction in the size of handshake signal sets and reduction of the impact of such signals by reducing their control footprint. This leads to more freedom in scheduling APUs relative to each other that would have been allowed by the scheduling the whole processes.

We therefore break up the processes into smaller independent (in respect to control signals) units (i.e. APUs) using the control signal information.

2.3 APU Generator

An APU represents the part of a process that will execute within one invocation of the Execute phase by the SystemC simulation kernel[14]. It is the smallest part of a process that the scheduler kernel is aware of (hence the atomic quantifier) and has a single activating control signal[1].

To determine the APUs of a process we need to consider the process type, synchronization points and incoming control signals. A synchronization point is, in general[2], marked by the presence of a *wait()* statement in the process source code. However, a method process is executed until it finishes for each invocation (i.e. no *wait()* calls) and thus it is a single APU. For thread and clocked thread processes we use calls to *wait()* and control signals to determine APU boundaries. We first slice[7] the process once for each incoming control signal. The resulting slices are then divided into APUs using the synchronization points still present in each slice. Any calls to *wait()* are finally removed from the APUs but control signals that they are waiting on (i.e. CS_{use} - the set of control signals they are activated by) are

[1] A logical composition of signals that can be seen as a single signal

[2] For simplicity we consider here only *wait()* statements

```
wait read|write
if read
 getA
 wait LATENCY-1
 putD
 done=true
 wait
 done=false
else if write
 getAD
 done=true
 wait
 done=false
 wait LATENCY-1
```

(a)

```
wait read|write
if read
 getA
 wait LATENCY-1
 putD
 done=true
 wait
 done=false
else if write
 getAD
 done=true
 wait
 done=false
 wait LATENCY-1
```

(b)

Figure 4. Process Slicing on a Control Signal

associated with the respective APU. We also associate processes with the control signals they update (i.e. CS_{def} - the set of control signals that they activate). For loops containing control signal operations we need only generate APUs for the loop body code and maintain the loop APU flow.

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 and determining the execution paths is complicated by the increased potential for deadlock. There are however some *if* statements that can be handled by our algorithm. Consider the code in Figure 4(a). It shows the implementation of one of the memory units. The code checks if the control signal *read* is active. If so, the *then* branch is taken containing 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 example apparently has 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 the code based on each enabling control signal that reaches the process. Let us assume, for example, that the enabling signal was determined to be control signal *read*. After slicing, the memory unit process will only consist of the section highlighted in Figure 4(b). As we can see, that code segment has no *if* statements and it can therefore be easily divided into APUs using the remaining synchronization points. Similar results are obtained if slicing is done on the *write* control signal. Therefore our restriction regarding *if* statements and synchronizing events applies to the process slices rather than the source process code.

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 5(a) showing pseudo-SystemC code for the controller unit and the resulting APU graph in Figure 5(b). The source code was divided into three APUs that were connected by the sequencing provided in the C++ code. Similarly we added the control signal information to the graph thus connecting the APUs that activate a control signal to the APUs that are activated by that control signal.

While constructing the APU graphs we also have to account for regular data dependences. Intra-process data dependences are accounted for by the C++ sequencing. The possible use of global and module-global variables would imply that they should also account for inter-APU dependences. However, the SystemC functional description[14] indicates that the simulation kernel

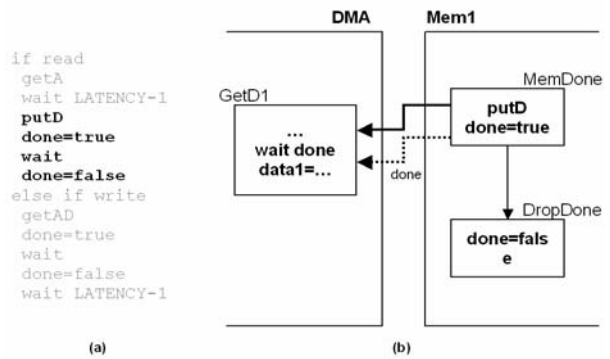

Figure 5. Handling a Signal Pulse

neither enforces nor guarantees an order of execution for the processes. Thus, if processes depend on global or module global variables for synchronization, the results can be undetermined and these dependences can be ignored.

2.4 Scheduler

The APU graph constructed defines a partial ordering on the execution of APUs. Before we can obtain a full ordering a preliminary step is needed that concerns SystemC SC_THREAD and SC_CTHREAD processes. These are generally modeled as infinite loops encompassing the actual functionality of the process. We can use the structural nature of the HTGs to identify the loop bodies of such processes and discard the rest of the loop construct.

To fully order the APU execution we apply a scheduling algorithm (e.g. ASAP) to the APU graph. This results in an APU schedule were several APUs can be executed at each step. Since we want to finally generate a thread of execution we have to sequentialize the scheduled graph. In general sequentializing is a straightforward process that traverses the graph breadth-first. However we need to ensure that the resulting code corresponds to the SystemC simulation semantics.

The diagram in Figure 6(b) represents a fragment of the APU flow diagram in Figure 7 and it is the result of the code in Figure 6(a). To make Figure 7 more readable certain APUs were not included. One such APU is an example of a common occurrence when modeling hardware: a signal is pulsed to indicate the occurrence of a specific event. Such is the case with the *Done* control signal for the memory modules. Once data is available on the data bus the *Done* signal is raised and held for one cycle as shown in the code in Figure 6(a). To understand how scheduling would handle this we have to consider the dependences that are involved. *DropDone* has a flow dependence (due to C++-imposed sequencing) on *MemDone* while *GetD1* has a control signal dependence (due to process communication) on *MemDone*. Therefore both will be scheduled after *MemDone* and possibly in the same scheduling step. However, since we are not using the SystemC signal update semantics (i.e. signals are updated at end of delta cycles) correct behavior is obtained only if sequentializing favors control signal dependences over flow dependences 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 obtain the execution paths through the system's APUs. Figure 7 shows one such possible execution

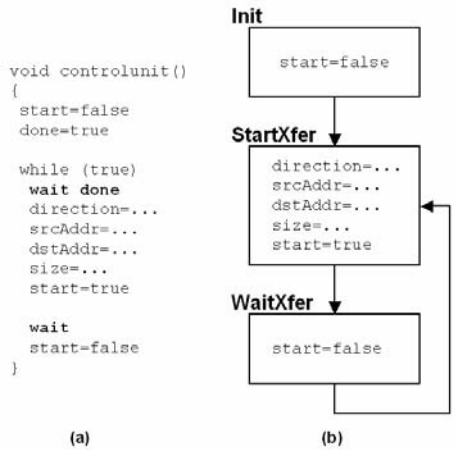

Figure 6. Controller APU Decomposition

path for our DMA example corresponding to a memory transfer from Mem1 to Mem2. There is a similar path corresponding to the reverse transfer: Mem2 to Mem1. The graph nodes represent the APUs that processes have been divided into (again some APUs have been omitted for clarity). The solid arcs between APUs reflect sequencing imposed by the input description (i.e. control flow dependences). The dotted arcs reflect sequencing imposed by the control signal analysis performed by our algorithm (i.e. control signal dependences) and are labeled with the signal that generate them. Finally the numerically labeled arcs represent the final sequence of APUs that make up the path of execution through the system for a transfer from Mem1 to Mem2.

2.5 Code Generator

In the final step we generate output SystemC-like C/C++ code for the transformed description. Some preprocessing is however required. Recall that in SystemC thread and clocked thread process functionality is enclosed in an infinite loop. For example, the dotted line from *MemDone* to *WaitStart* in Figure 7 shows the back arc of the infinite loop in the implementation of Mem1. 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 needed. Other loops ca similarly be removed but there are cases where that is not possible. For example, for the Controller module we have to leave the loop in place since it lacks any incoming control signal (e.g. the process gets its signal values from a test bench).

The code generator is also responsible for generating the code that defines the main execution routine and for managing thread execution and synchronization in the newly restructured system. As code is generated for the new execution paths, the question arises of how much of them can be mapped onto threads that can then be executed simultaneously. In our DMA example we could create a thread that comprises the whole system but each such thread will have to wait for the previous thread to finish before it can execute due to data dependences on the memory array accesses. This will result in the threads actually executing sequentially and therefore would not provide any simulation speedup. We have to therefore try to identify smaller sections of the execution paths that we can map onto threads and execute simultaneously. As in traditional software compilation, we give

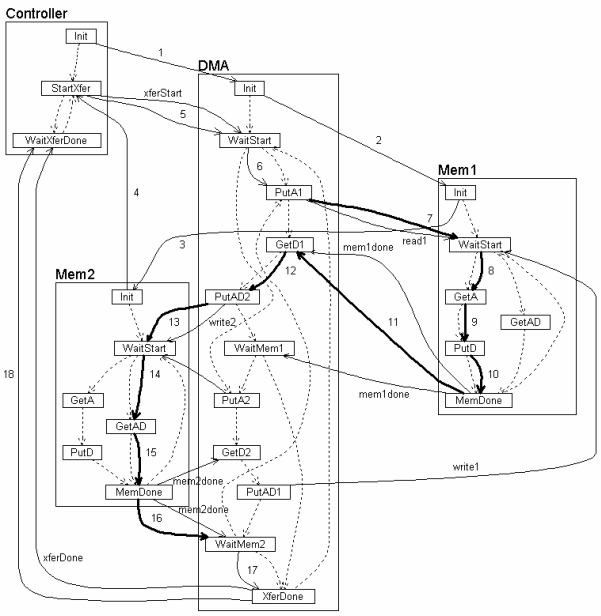

Figure 7. DMA Example Final APU Flow

special attention to any loops present in the description. Since applications spend most of their execution time in loops improvements to loop runtimes translate into good overall runtime improvements. The DMA module has such a loop corresponding to the iteration until all the data has been transferred. Since we would like to issue multiple instances of this thread simultaneously we need to analyze the path being considered (i.e. bold line segments 7 through 16 in Figure 7) for loop-carried dependences (both control and data). If they exist, we need to enforce them by inserting thread synchronization events between any APUs involved in dependences to ensure that the threads execute those APUs in the proper order. In our example there are no dependences so the threads do not have to wait for each other nor does the issuing thread need to wait for threads to finish before issuing more. This will result in almost linear improvement in the loop execution time for this example but in general some synchronization will be necessary.

3. TRANSFORMATION CORRECTNESS

One aspect that needs to be addressed is the correctness of our algorithm or in other words that the resulting model is semantically equivalent to the original SystemC description. We will sketch our strategy of proving such an equivalence using the notions of communicating sequential processes (CSP)[6] and proof-carrying compiler. Note that, once we precisely characterize the classes of SystemC models on which our algorithms will work, we will have the transformed code equivalent (i.e. by construction) to the original. However, since we do not have a precise characterization, currently we manually generate a proof of correctness with each transformation.

3.1 Proof-Carrying Compiler

The idea of proof carrying compiler is that, when a source program P_S is compiled into a target program P_T, the proof that P_S is equivalent to P_T is also generated during the compilation process. In our case the original model P_O and the final model P_F have to be proven equivalent. This means that all possible APU sequences that occur when simulating P_O are also possible in P_F modulo shuffling on non-local actions (i.e. APU order within P_F will maintain the local APU order present in P_O). Any safety property tested on the transformed design P_F would also hold for the original design P_O due to this equivalence. Considering the DMA example we can illustrate how the compiler could generate the communicating sequential processes expressions from the original and the transformed APU graphs and then use a trace equivalence algorithm to prove their equivalence.

3.2 Communicating Sequential Processes

CSP is a process algebra, in which communicating processes can be described using terms over an alphabet of atomic actions and various operators. The operators we use in the following example are: the sequencing operator →, the choice operator |, and the parallel composition operator ||. Further details on the semantics of these and other CSP operators and terms are discussed in [10]. However, given two process terms representing two models, the modulo shuffling trace equivalence between them is the equivalence relation semantically meaning that both models are capable of executing the same (albeit modulo shuffling) sequence of APUs. In case of the DMA example, we construct the CSP description of each module, in terms of their atomic actions, sequencing and choices. A parallel composition of the terms represents the model of the system. The trace set of this model should be equivalent to the trace set of the model obtained by concurrency reassignment. The trace equivalence is decidable, although EXPSPACE-hard[8]. A future version of the compiler will also generate CSP expressions for both the initial and restructured model and generate the equivalence proof or a counter example showing non-equivalence using modulo shuffling trace equivalence. However, details of this methodology are outside the scope of this paper.

3.3 DMA Example

To create CSP processes from our SystemC description we have to look at the source code from a control signal point of view. As we mentioned before, we associate each process with the control signals it activates or it is activated by. Thus we can define the alphabet of each process as the union of all such signals:

$$\alpha P = CS_{use} \cup CS_{def}$$

Referring to Figure 7 showing the DMA modules in terms of their APU graphs and the control dependence, we can obtain the CSP specification of the modules as follows:

Mem1:

$$\alpha Mem1 = \{initMem1, read1, write1, mem1done\}$$
$$Mem1 := initMem1 \rightarrow Mem1Body$$
$$Mem1Body := read1 \rightarrow Mem1Finish \mid write1 \rightarrow Mem1Finish$$
$$Mem1Finish := mem1done \rightarrow Mem1Body$$

Mem2:

$$\alpha Mem2 = \{initMem2, read2, write2, mem2done\}$$
$$Mem2 := initMem2 \rightarrow Mem2Body$$
$$Mem2Body := read2 \rightarrow Mem2Finish \mid write2 \rightarrow Mem2Finish$$
$$Mem2Finish := mem2done \rightarrow Mem2Body$$

Controller:

$\alpha Controller = \{initController, xferStart, xferDone\}$
$Controller := initController \rightarrow ControllerBody$
$ControllerBody := xferStart \rightarrow xferDone \rightarrow ControllerBody$

DMA :

$\alpha DMA = \{initDMA, xferStart, xferDone, read1, write1, mem1done, read2, write2, mem2done\}$
$DMA := initDMA \rightarrow DMABody$
$DMABody := xferStart \rightarrow DMAXfer$
$DMAXfer := read1 \rightarrow M1toM2 \mid read2 \rightarrow M2toM1 \mid xferDone \rightarrow DMABody$
$M1toM2 := mem1done \rightarrow write2 \rightarrow mem2done \rightarrow DMAXfer$
$M2toM1 := mem2done \rightarrow write1 \rightarrow mem1done \rightarrow DMAXfer$

The entire system can be expressed as a CSP process through parallel composition of the 4 component modules which will account for the control dependence shown in Figure 3(b). Therefore the final CSP expression for our DMA example will be given by:

$$DMAControl\,ler = Controller \parallel DMA \parallel Mem1 \parallel Mem2$$

The transformed model can be similarly expressed as CSP terms and it can be verified by inspection to be MS equivalent to the original model traces.

4. IMPLEMENTATION AND RESULTS

We have implemented the algorithm starting from an EDG C++ Front End[3] that allows us to fully support the latest C++ standards and easily generate output C/C++ code. To facilitate our analysis we have layered an HTG representation along with a class hierarchy on top of the EDG intermediate representation, and implemented various passes (e.g. data dependence analysis, constant propagation, dead code elimination, signal analysis, program slicing) however APU selection for thread mapping and CSP generation are still done manually.

In [11] we showed simulation results for a simple RISC processor and an MP3 decoder. For this paper we applied our algorithm to the DMA example and also a few of the samples included with the SystemC 2.0 distribution[14]. We simulated both the original and transformed models on a dual processor machine. Table 1 presents the experimental results and shows improved simulation performance. There is some overhead due to the use of preemptive threading in particular for the DMA example (as a result of not being able to extract a large enough thread of execution).

5. CONCLUSION

In this paper we presented an algorithm for threading structure transformation of C++ based system level models that allows hardware design models (i.e. designed for synthesis with the hardware structure in mind) to also efficiently target simulation and, in particular, multiprocessor simulation. Unlike software synthesis based on scheduling or quasi-scheduling[13], we can generate a coarsely multithreaded program so as to exploit multiprocessor simulators. Also, for now, we manually generate a proof of equivalence or counter example along with the transformation. This methodology is based on CSP trace equivalence but we are yet to characterize what classes of SystemC models will result in equivalent models after the transformation (the examples we have tried produced trace equivalent code). However, our future work is focusing on a necessary and sufficient characterization of the nature of SystemC models for which the transformation will always produce trace equivalent code.

Table 1. Simulation Results on a Dual Processor System

Benchmark	Original Time	Threaded Time	Speedup
DMA	50.674	31.134	62.76%
FIR	17.381	10.190	70.57%
FFT	9.822	5.635	74.30%
PKT_SWITCH	153.234	86.617	76.91%

6. REFERENCES

[1] CynApps Inc, http://www.cynapps.com.

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

[3] Edison Design Group, http://www.edg.com/cpp.html.

[4] P. Garg, S. Shukla, R. Gupta, “Efficient Usage of Concurrency Models in an Object Oriented Co-Design Framework”, In Proceedings of DATE 2001.

[5] M. Girkar, C.D. Polychronopoulos, “The Hierarchical Task Graph as a Universal Intermediate Representation”, International Journal of Parallel Programming, vol. 22, no. 5, October 1994.

[6] C. A. R Hoare. “Communicating Sequential Processes”, Prentice Hall, 1985.

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

[8] A.J. Mayer, L.J. Stockmeyer, "The complexity of word problems - this time with interleaving", Information and Computation, Vol 115, 1994.

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

[10] A. W. Roscoe, “The Theory and Practice of Concurrency”, Prentice Hall, 1998.

[11] Nick Savoiu, Sandeep Shukla, Rajesh Gupta, “Automated Concurrency Re-Assignment in High Level System Models for Efficient System Level Simulation”, In Proceedings of DATE 2002.

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

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

[14] SystemC 2.0, http://www.systemc.org.