Optimizing Communication in Embedded System Co-simulation

Ken Hines and Gaetano Borriello
Department of Computer Science & Engineering, Box 352350
University of Washington, Seattle, WA 98195-2350
{hineskj,gaetano}@cs.washington.edu

Abstract

The Pia hardware-software co-simulator provides substantial speedups over traditional co-simulation methods by permitting dynamic changes in the level of detail when simulating communication channels between system components. However, it places a burden on the designer to develop several communication routines, at different levels of abstraction, for each communication operation. This often requires an intimate understanding of both the simulator and the design being simulated. This paper presents and demonstrates a way to use communication transaction annotations to provide a platform independent language for describing fast communication primitives. Additionally, we show a tool for automatically generating some of these annotations, so that the designer does not even require an intimate understanding of the design under simulation. This can be important when simulating systems where the design itself is synthesized by automatic tools, and is liable to change frequently.

1 Introduction

The Pia hardware-software co-simulator allows a designer to specify several communication methods - each at different levels of detail - for any communication function. The Pia co-simulator will choose an appropriate method at runtime based on criteria given by the designer. Previous work has shown this to be an effective way to substantially reduce simulation time while retaining many of the the benefits of detailed co-simulation[Hin96].

Detailed co-simulation can be a good way to validate a design as well as to evaluate some of its interesting features. For example, through detailed co-simulation a designer can gain better insight into resource utilization and the frequency of shared resource conflicts in a system. Unfortunately, using a uniform level of detail in co-simulation can be extremely time consuming and in many cases the designer really only needs the details for small parts of the system. The rest of the system is only being simulated to provide a wordload for the portions of interest. Static mixed-level co-simulation can eliminate some of these problems, but it forces the designer to choose the mix of detail at the start of the simulation run. Once the mix is chosen the designer can't alter it without stopping the simulation and starting over.

The substantial speedup observed in Pia comes from allowing a designer to select different detail levels for different paths *and* to dynamically change this selection. This means that the designer can tell the simulator things like "use a low detail method for this path until some specific event occurs, then switch to the full detail method."

This leaves us with the problem of generating the optimized communication methods. In our previous papers on Pia, we have assumed that there is a library of standard interfaces, and that a designer can usually find a library interface that is similar enough to any new interface that the alternative methods can be inherited. If such an interface is not available or if not all of the optimized routines make sense in the new interface then the designer will be forced to develop the routines in addition to the design. There may be cases where a high level transaction causes state changes in several lower level interfaces, and it may be important to continue this even when running at a lower level of detail. An example of this may be a network interface where we are tracking utilization at the lowest level. When we step up to a higher level of abstraction, we still need to keep track of network utilization, even though we are no longer performing any work at the detail level where the logging occurs. It can be difficult for the designer to write fast communication primitives that take advantage of every optimization possible, and yet are still as accurate. In general, the writing of accurate, efficient fast modes requires a good understanding of:

- The high level behavior of the system.
- The interaction between the system and the architecture to which it is mapped.
- The operational details of the simulator itself.

This means that a designer could spend a considerable amount of time generating good fast modes for a particular architecture, and then find them useless when the target architecture changes even slightly.

2 The Pia hardware-software co-simulator

The Pia co-simulator has two significant parts: a language pre-processor, and a simulator core. The simulator core in the current version is built on Berkeley's Ptolemy[ea95], but we are in the process of writing a stand alone core which has integrated Java support.

Most portions of a system simulated with Pia are written in the Pia language, which is a language designed to explicitly support dynamic communication modes. Systems described in Pia consist of four different types of objects:

0-8186-7895-X/97 $10.00

- *component* A collection of interfaces and behavior.
- *interface* A collection of ports, driver routines and simple event handlers. These usually belong to a component.
- *port* A sender and/or receiver of data events. These usually belong to an interface.
- *net* An interconnection of ports with some arbitration scheme.

The behavior of a component can be specified with C programs from separate files. In fact, processors are usually modeled with Pia components whose interfaces and pins match those on a physical processor and whose behavior is provided by the actual software that will run on the physical processor.

A C program does not require any alteration to run as part of a Pia component. To run with time accuracy, however, it must include timing annotations which can be placed in comment fields.

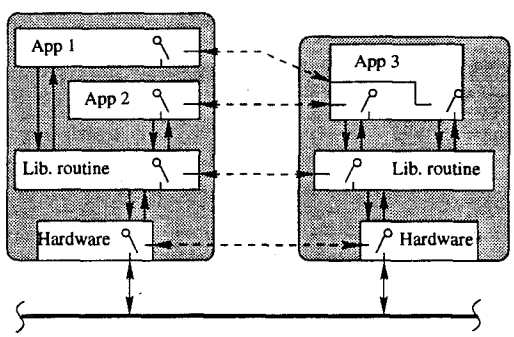

Fig. 1: Two communicating components

There may be several versions of each communication routine for Pia to choose from. This can allow the designer to essentially install switches at each abstraction level (as shown in Fig. 1) and to dynamically throw these when simulation efficiency and detail shift in importance.

To simplify timed simulation and to minimize inter-component synchronization, we make some assumptions concerning the types of systems for which Pia will be most commonly used. We assume that

- Components aren't interrupted often.
- Components can usually finish all of their work between interrupts.
- Interrupt handlers will be limited in the extent of internal state which they can alter. *i.e.* they will have only a few addresses in a processors local memory to which they can write.

The simulator core schedules all communication and component run time so there is no dependence on operating system communication primitives as in[TC95, BST92]. Several versions of time are maintained during a simulation run. For example, each component keeps a version of its own *local time*, and the main scheduler keeps a version of time called *system time*. There is also *real time* which functions in Pia as a meta-time. Essentially, we can strobe all of the components' versions of time at any particular real time and find them to be different. There are three rules about how the versions of time must relate to each other:

1. System time is always less than or equal to all local times.
2. System time is always monotonically increasing in real time.
3. Before a component can read a value from outside, that component's local time must equal system time.

It's not hard to show that if interrupts are not allowed, these rules will keep the system coherent. For example, we know that if we follow the rules: whenever we read a value that was sent to us, every other component will have a local time which is greater than or equal to our local time, so any components that would change the value already have. If we consider interrupts, it gets a bit more complex but as long as our assumptions hold we can deal with them without much increased cost. For example, if we can statically determine which addresses in a component's local memory are either written to or read from by interrupt handlers, we can have the Pia compiler mark those locations as *synchronous* which means that the component will have to ensure that its *local time* matches *system time* when it reads or writes to any of these locations.

In many cases, it may be overly conservative to mark all such locations as synchronous. To ensure that it isn't necessary to do so, Pia includes a checkpoint and restore mechanism for implementing optimistic techniques where there is loosely bound synchronization space. For example, suppose there is a pointer in one of the interrupt handlers, and we do not have tight bounds on the range of addresses to which this pointer may point. Instead of marking all addresses as synchronous, we can dynamically determine which ones really count as we run our simulation. To do this, we start off with no addresses marked as synchronous. If an interrupt handler attempts to read from or write to an address that isn't marked, it marks the address and causes an exception which forces the system to backtrack.

3 Optimizing communication

The systems for which the Pia simulator was designed have these characteristics in common:

- Many of the lowest level communications routines are provided by a protocol library.

- The systems are synthesized from some higher level specification by a tool such as Chinook[COB95].

In both cases, there are many opportunities to short circuit detailed communication and provide better simulation performance where high detail levels aren't required. For example, it is usually possible to provide at least one optimized routine for every library function. (*e.g.* by increasing the grain-size of the data transfers)

3.1 Transaction annotations

A designer can specify the behavior of a transaction through *transaction annotations*. A transaction annotation is a small statement which describes the behavior of a transaction. It specifies conditions on the system state that may be required for the transaction to occur, as well as the alterations in state caused by the transaction.

A transaction annotation consists of a precondition-action-postcondition triplet. The precondition and postcondition list the parts of system state that are relevant to the transaction. For example, suppose there is a routine named `write_page` which transfers a specified page from component X to component Y. The transaction annotation for the entire transaction may look something like this:

{**free**: *buf, page*
 {X{*buf*{*page*}:**word**[256]}, Y{z{_}:**word**[256]}}
  X.write_page(Y.address, *page*:**word**[256])
   {X{*buf*{*page*}}, Y{z{*page*}} } }

This annotation tells us that before a successful transaction occurs, Y has some 256 word buffer named z which can be overwritten and that at the end of the transaction, the buffer z must contain the value *page* - which is the value passed as a parameter by X. The **free** keyword in the annotation indicates variables that are to be considered free variables. It is important to note that the precondition should contain *only* the state required to set up a transaction.

Unfortunately, this particular annotation requires components X and Y to be aware of each other - violating the principles of modularity. Also, the annotation above is a not a property of either component, so it cannot be inherited. To allow better modularity and to facilitate inheritance, we allow descriptions of transaction behavior to be split between a number of annotations. This is important, for example, for describing all the transactions that can occur between a number of different components on a bus. For each component we can specify a *partial transaction* annotation which describes its share of an action. The transaction represented by the annotation above is shown below represented by two separate annotations:

{**free**: *buf, page, any*
 {X{*buf*{*page*}:**word**[256]}}
  X.write_page(*any*.address, *page*:**word**[256])
   {X{*buf*{*page*}}} }

{**free**: *page, any*
 {Y{z{_}:**word**[256]}}
  any.write_page(Y.address, *page*:**word**[256])
   {Y{z{*page*}}} }

These mean that as far as X is concerned, a page write behaves the same, regardless of the target address and as far as Y is concerned, a page write from anywhere with it's own address always triggers the same response.

3.1.1 Timing information in Transaction Annotations

The Pia co-simulator requires timing annotations for time accurate simulation. It is important to ensure that each of the optimized versions of any driver routine take the same amount of simulation time. Transaction timing can be included in an annotation as follows:

{**free**: *buf, page, any*
 time: 400*us*
 {X{*buf*{*page*}:**word**[256]}}
  X.write_page(*any*.address, *page*:**word**[256])
   {X{*buf*{*page*}}} }

This states that the entire transaction takes 400 microseconds.

It is possible to have a timing mismatch between two halves of an operation. For example, the Y half of the above transaction may only require 300 us to receive a page. In these cases, the maximum time of both transactions is always chosen.

3.1.2 Combining transaction annotations

Transactions which mainly consist of concatenations of other transactions can usually be easily generated by combining the annotations as follows: Let $D.pre$ refer to the precondition portion of annotation D, $D.operation$ refer to the operation portion of D and $D.post$ refer to D's postcondition. A pair of concatenated operations can be combined for a transaction annotation: $Comb(A, B)$ where

$$Comb(A,B).pre = A.pre \cup (B.pre - A.post)$$
$$Comb(A,B).operation = A.operation : B.operation$$
$$Comb(A,B).post = B.post \cup (A.post - B.pre)$$

This capability is important in enabling further optimization of collections of communications, possibly generated and inserted automatically, into larger units.

3.2 Pia FaST

The Pia FaST (Fastmode Synthesis Tool) translates transaction annotations into optimized communication method pairs (*i.e.* a Pia driver routine, and a Pia handler) whose action is described by the given annotation. Ideally, the only difference between these methods and the original methods

is that the intermediate states may not be the same. Practically, this will probably not always be the case - otherwise, there would not be a reason to ever ask for the higher levels of detail. Some of these differences may occur when the intermediate states are used in a complex arbitration scheme, which may be difficult to capture at a higher level of abstraction.

3.3 Automatic generation of transaction annotations

Although transaction annotations allow a designer to generate fast modes without an understanding of the internal workings of the simulator, the designer still needs to understand both the specification and the design of the system under simulation. This can cause problems if the design is the output of synthesis tools, because requiring such a high level of understanding before a system can even be simulated is counter-productive.

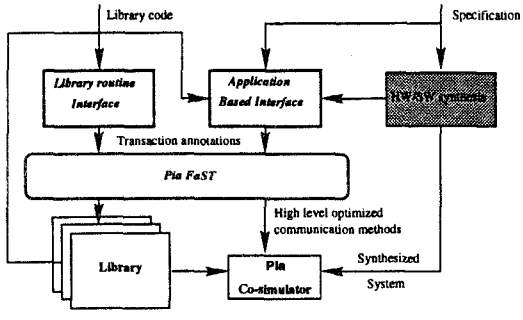

Fig. 2: The Pia FaST can synthesize fast modes from transaction annotations. These can be written by hand or generated by various pre-processors.

3.3.1 Library writer's interface

The library writer should usually generate the annotation for all of the library routines. There is a combination tool which combines concatenated transactions as in Section 3.1.2, and this can be used as an aid in generating large parts of library annotations, however the output of this tool will usually not be sufficient.

3.3.2 Application writer's interface

The application writer will usually have a very good understanding of the system *specification* but since the *design* may be the product of a synthesis tool, it may not be so well understood. The application writer's annotation synthesis tool tries to include as much information as possible in the annotation. For example, in a system's high level specification, there may be a communication function `send_frame` that causes different state changes depending on how the system is implemented. For example, if somewhere down the line the frame passes through a network, we may want to log it at the entry node (even if our lower detailed communication method doesn't pass the page through the network at all). In fact, this is an example of where it may not be possible to include specific information about all changes in system state in the annotation. For example, if we're using a network with non-deterministic routing and each intermediate node needs to log every message that passes through it, it is not possible to determine in advance which nodes will have logged any particular transaction. In many of these cases, it is probably best to leave the state unchanged and to simulate at a higher detail level to gain this sort of information.

The application writer's tool is oblivious to all changes of state that are not included in either the top level specification or in annotations for lower level transactions. This means that if detail is left out of a library annotation, the same detail will also be left out of any application routine that calls the function it represents.

4 A Test Run

This section demonstrates the use of both pre-processing tools and the Pia FaST on a particular application. The application itself is a video circuit for some limited bandwidth path. This system uses a camera to generate 128 X 128 frames of data (although this size is arbitrary) and passes them through a link with an available bandwidth of about 100k bits per second. To achieve a frame rate of 14 frames per second, we need to reduce the bandwidth from nearly 2Mbps to less than 100k so that it fits (nearly a 20 times reduction).

4.1 High Level Specification

The system contains several compression layers, as shown in Fig. 3, which is also an example of a Pia *drawing*. There are *attributes* associated with each channel in the drawing. For example, the $(Camera, WaveletEncoder)$ channel is identified as a 16 k byte message channel, so that each message can contain a full frame from the camera.

It is important to note that in this picture the *camera* and the *LCD* do not yet represent physical devices. Instead they are camera and LCD abstractions with arbitrary protocols. It is expected that a synthesis tool will set up the physical communication protocol when the appropriate physical device is chosen.

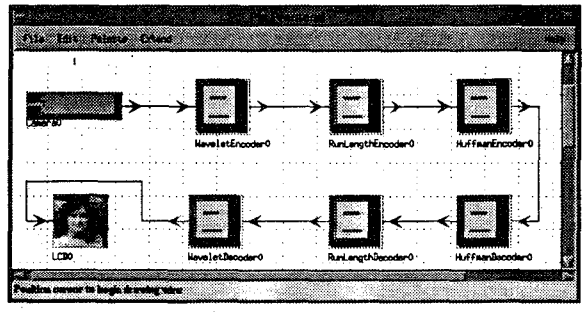

Fig. 3: A high level description of a video circuit

4.2 Mapped version

Fig. 4 shows the system mapped to a particular architecture. The "Compress" and "Decompress" components are specified as ASICs that include fast microprocessor cores. In

this drawing the camera and LCD do, in fact, represent physical devices. In this, we import a library protocol for communicating with these devices.

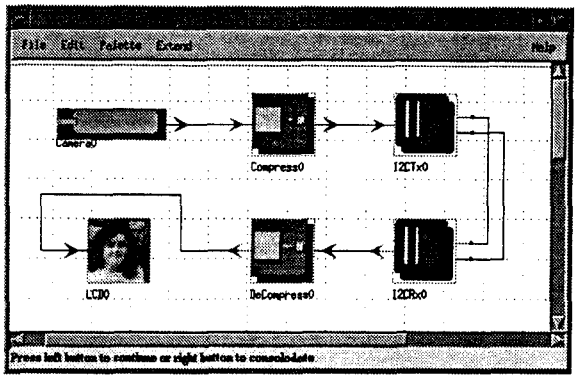

Fig. 4: The video circuit mapped to an I^2C architecture

4.3 Library specification

We use a simplified I^2C library and only specify a few communication primitives:

- send start and slave address
- send byte
- send ack
- send nack

The transaction annotations for each of these operations are written by the library writer.

Our library extension consists of one driver routine (`send_pkg`) for the master component and one handler (`handle_pkg`) for the slave component.

4.4 Results

We define 4 distinct run levels for this system:

1. Hardware level : Detailed wires of I^2C.
2. Hardware abstraction level : I^2C byte level transfers.
3. First library level : Packages of 256 bytes.
4. Application level : Unencoded packages.

All of the optimized run levels (2 through 4 above) are generated by the Pia FaST, but the *hardware abstraction level* and the *first library level* are stored in the I^2C library, while the *Application level* is stored in the *Compress* and *Decompress* component specifications. The *transaction annotations* for the *hardware abstraction level* are manually generated by a library writer and the annotations for the *first library level* and the *application level* are generated by the combination tool and the application tool respectively.

Table 1 shows the time to transmit and display 14 frames under each of the run levels. These tests were performed on a

Run Level	Run Time (s)
Hardware	375.2
Hardware abstraction	32.2
Library abstraction	22.4
Application level	13.72

Table 1: The time for transmitting 14 128 x 128 frames at various run levels

70MHz Pentium workstation running *Linux*. It is interesting to note that for this example, the largest part of the speedup comes from the methods synthesized from handwritten transaction annotations. This is actually because the compression portion of the circuit reduces the communication bandwidth sufficiently to allow computation to contribute significantly to communication time when we use byte wide communication.

5 Conclusion

If used properly, multiple communication methods which perform the same transactions can improve performance of embedded system co-simulation, while retaining detail where required. In this paper, we presented some tools that generate optimized communication methods and minimize the amount of lost accuracy for various speedups. In our experiments, these tools have shown significant speedups over unoptimized communication and have demonstrated that it is possible to automatically generate and include these optimizations.

References

[BST92] D. Becker, R.K. Singh, and S. G. Tell. An engineering environment for hardware/software co-simulation. In *29th ACM/IEEE Design Automation Conference*, pages 129–134, 1992.

[COB95] Pai Chou, Ross B. Ortega, and Gaetano Borriello. Interface co-synthesis techniques for embedded systems. In *ICCAD Proceedings*, 1995.

[ea95] Joseph T. Buck et. al. *Almagest, Ptolemy manual version 0.5.2*. UCB, 1995.

[Hin96] Ken Hines. Pia: A framework for embedded system co-simulation with dynamic communication support. Technical Report UW-CSE-96-11-04, University of Washington, November 1996.

[TC95] D. E. Thomas and S. L. Coumeri. A simulation environment for hardware-software codesign. In *Proceedings, International Conference on Computer Design*. IEEE CS Press, October 1995.