Dynamic Communication Models 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

Many co-simulation techniques either suffer from poor performance when simulating communications intensive systems, or they represent communications with a uniformly low level of detail. This paper presents a technique which allows communication to be represented at multiple levels of detail and which gives a designer the ability to dynamically choose the appropriate level for different parts of the system. This paper also presents a tool which uses this technique and experiments which show relative simulation speedups.

1 Introduction

Hardware-software co-simulation is used to validate both the hardware and the software components of embedded systems, as well as the interaction between them. Co-simulation can be used to gain information about an embedded system before a prototype is actually built. Furthermore, co-simulation allows designers to perform experiments that could be impractical in a prototyping environment. For example, designers can perform comparative studies of a single system using a variety of processors, or they can repartition the hardware and software portions of the system, co-simulate and evaluate the results.

To be practical, co-simulation needs both a high degree of detail and good performance. Unfortunately, highly detailed simulation tends to incur a large time cost. This means that designers should simulate at a level of detail which provides only the required information and does not waste time generating unnecessary detail.

Rowson lists several co-simulation techniques which illustrate the tradeoff between performance and detail [10]:

- *The nano-second accurate processor model.* In this technique, the processor is modeled with a great deal of accuracy. Typically, only 1 to 100 instructions can be executed per second. This is probably much more detail than needed in embedded systems.

- *The cycle accurate processor model.* This is likely to be the highest level of detail needed in simulating an embedded processor. The processor model has an internal structure much like the real processor, including pipelines, interlocks and functional units. Typically, between 50 and 1000 instructions can be executed per second.

- *The instruction set accurate processor model.* This model matches the architecture of the target processor, but does not attempt to match timing. Pipelines, caches and such are completely ignored. Performance with this technique can reach 10,000 - 100,000 instructions per second.

- *The model-free synchronizing handshake.* With this technique, the software is compiled for the host processor, and is linked to a hardware simulator with a synchronizing handshake for communication. Performance in this technique is mostly limited by the performance of the hardware simulator.

- *The virtual operating system.* In this method, the hardware and operating system of the embedded system are abstracted away to a *virtual operating system*, which looks the same as the actual system from the the software's perspective. This technique has very good relative performance, but it also provides the lowest level of hardware detail.

- *The bus functional processor model.* In this technique, the entire processor is abstracted to a set of test vectors. This technique is good for validating the hardware, but doesn't help much with the software.

None of these are entirely satisfactory for our purposes. Cycle accurate and nanosecond accurate techniques are not fast enough to give the designer much insight into the complete interaction between hardware and software. The virtual operating system and the bus functional models are unsatisfactory because they each look at only half of the system (either hardware or software). The *synchronizing handshake* method is interesting, because it gives a way to simulate the complete system without a processor model.

In validating embedded systems, we often don't care about the internal details of a processor's operation. We are usually more concerned with *timing*, hardware and software *interfaces* and *functionality*. In other words, any processor model that executes the code properly, and provides the right interface events at the right times will be sufficient.

Wilson describes a co-simulation technique called *selected cycle simulation* [16] which essentially combines the synchronizing handshake and the bus functional models of Rowson. In this technique, the system software is compiled for the host computer. For communication, the software sends a message to a bus functional unit in the hardware simulator. The bus functional unit performs the requested functions and returns any results. Wilson suggests using a Unix pipe for the *software -to- bus functional unit* message link. This technique is useful, but it unfortunately makes synchronization expensive. To avoid invoking this cost unnecessarily, synchronization is performed only when needed for correct communication. This technique is well suited for applications where synchronization and hardware-software communication are minimal.

Many groups have found it worthwhile to compile the system software for the simulator host and run this software in a process of its own using operating system based primitives for communication with a hardware simulator [1, 12, 11, 15, 14, 6]. Techniques

This work was supported by ARPA contract DAAH04-94-G-0272

DAC 97, Anaheim, California
©1997 ACM 0-89791-920-3/97/06..$3.50

of this sort can yield very good performance when simulating computationally intensive systems that don't require much in the way of fine grained communication. However, communication between the hardware simulator and the software is expensive, and this limits the amount of speedup possible. This is especially true when simulating communication or synchronization intensive systems.

This paper presents a technique which permits communication to be described at multiple levels and gives a designer the ability to dynamically choose the appropriate level. Dynamic selection of abstraction level is not unique to our technique. For example, Mentor Graphics' Miami/Seamless CVE allows both address based optimization of processor-memory transactions through a Memory Image Server (MIS) and time optimization, (turning off the simulator clock) [7]. We believe, however, that our technique is unique in that 1) it gives a designer the ability to dynamically speedup any communication between any components and 2) it allows a designer to use a high level understanding of the simulated system to provide optimal communication methods.

2 Our Approach

As we have inferred, it may not be enough to simply speed up the simulation of the embedded processors. In many embedded systems, there is a great deal of communication between the hardware and software portions of the system. This may be true, even if we abstract out all trivial communication such as instruction fetches and local data accesses. In many cases, it may be important to simulate all of the cycle level details of a set of transactions; but it is often true that we just need the data to be transferred and we aren't overly concerned with collecting the details of how it gets done each and every time.

This is similar to the debugging of a hardware prototype using a logic analyzer. A designer working with these tools will not generally set up the bus analyzer to collect the details of all of the transactions on all of the wires. This is because it would generate an unwieldy amount of potentially useless data and there aren't enough storage resources to collect it all at once. Normally, a designer will methodically check the interfaces in a reasonable order, to narrow down suspected bugs. Logic analyzers usually facilitate this explicitly by allowing the designers to program filters, triggers, and trigger windows - and implicitly by allowing the designer to choose which wires to connect to.

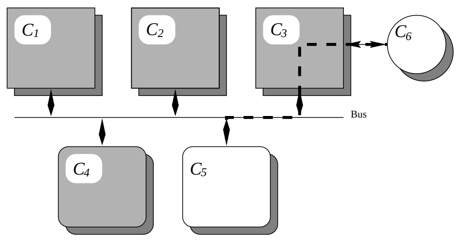

Fig. 1: Several components on a bus

Fig. 1 shows 5 components connected directly to a bus (C_1 through C_5) and an additional component, C_6, is connected through C_3. Suppose there is a bug somewhere in the system that causes the data from C_5 to C_6 to sporadically become corrupted. If we were debugging the system with a hardware prototype, we might first investigate the link between C_3 and C_6 with a data analyzer to determine whether the data is corrupted when it arrives at C_6. In this case, we still need the rest of the system to behave correctly, but we don't need detailed representation of any communication besides that between between C_3 and C_6. If we are debugging with a simulator, we would also like to avoid representing all of the details that cost time, but which we don't use.

We would like to be able to tell the simulator which data transfers it should represent in detail, without having to write separate system specifications for each level of detail. We would also like to be able to dynamically alter this information during a simulation run. This was the primary motive behind the design of the Pia co-simulation tools. The Pia co-simulation tools include a simulator, as well as some other tools for easing the use of the simulator.
The Pia tools

- provide a mechanism for specifying multiple communication models for each interface, and dynamically switching between them during the simulation run,
- include *processor synchronization* in the simulator scheduler so that the observable actions of each processor are synchronized through the scheduler. This eliminates the need for synchronizing through OS primitives, and finally
- provide a mechanism for processors and processor blocks to be described at different levels of detail. This allows: Processor simulators, re-compiled object code (as in [13]) as well as source code compiled for the simulator host to be used to model a processor. In most cases, we expect that source code compiled for the simulator host processor will be sufficient (unlike communication models, processor models cannot be changed during a simulation run.)

To simplify the design of the simulator itself, we chose to implement it as a Ptolemy domain. Ptolemy is a simulation environment which provides a graphical interface, primitives and tools for writing new simulation domains, and primitives for matching communications semantics between domains. In general, Ptolemy domains are distinguished by their simulation semantics.

As part of our simulation environment, we include a language for component and interface specification. This language is designed to allow but not constrain the user to specify communications at multiple levels of detail.

2.1 Driver abstraction and fast communication

Software usually communicates with the hardware as well as software components on other processors through driver. Drivers can be, and often are often hierarchically layered. For example, network applications may call into a *stream* level driver which calls a *packet* level driver, and in turn calls other drivers which actually manipulate hardware state. Each driver essentially hides all of the layers below it. For example, it doesn't matter to the application how the stream layers communicate with each other, as long as they transfer the right data.

Speed of communication is dependent on both *unit of transfer* which is the amount of information transferred on a single transaction and *transaction overhead* which is the cost to perform a transaction. If we move up the hierarchy of drivers, we usually find that the unit of transfer grows with each step up. We can use this information to find reasonable fast communication models for this sort of system.

Fig. 2 shows two devices on a network. The Pia tools allow the designer to define multiple communications models for *each* layer in the driver hierarchy. For example, the stream layers in simulation can communicate with each other directly, or they can communicate through their respective packet layers. Similarly, each of the lower layers can have multiple communications methods.

2.2 Synchronization

In Pia, each component keeps its own local time (t_{c_i} for each component, C_i). The system scheduler keeps track of system time, t_s, which is always less than or equal to all local times and is monotonically increasing. Each component periodically synchronizes its

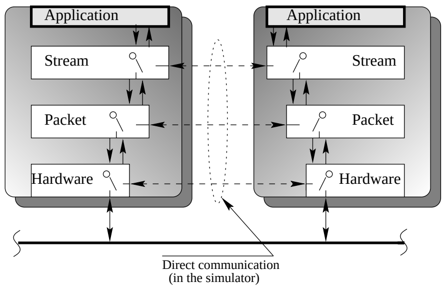

Fig. 2: Direct communication between drivers at different levels

local time with the system time, by essentially blocking until system time catches up. This typically occurs on input events. Fig. 3

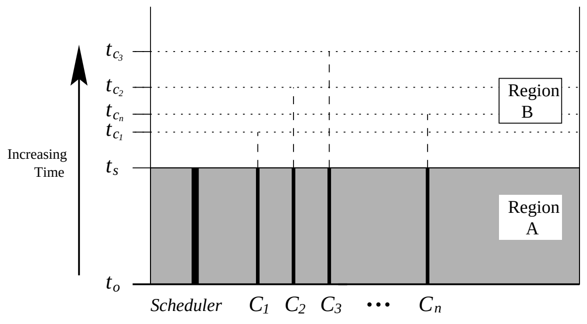

Fig. 3: t_s “pushes” all t_{c_i}s

shows how t_s essentially “pushes” all of the local times forward. What this means is that no component will ever run with a local time less than system time. This is enforced in different ways for *active* and *reactive* components, where reactive components are those that require stimuli to run. Every time an *active* component yields to the scheduler, it also informs the scheduler that it needs to run again when system time equals its local time. This local time will already be greater than or equal to system time, since system time will not have advanced while the component was running. *Reactive* components don't set wakeup calls when they yield, however their local time is set equal to system time if ever system time is greater. This can be performed lazily, *i.e.* if a component has no work to do, system time can be advanced many times without updating the component's local time. However, when the component is activated, all of the increments in system time must be reflected.

Fig. 3 shows that there are two regions in time, labeled “region A” and “region B”. Region A is all time less than t_s and represents time fully passed. Components with local times in region B are allowed to perform $sends$ but $gets$ are prohibited. If a component needs to execute a get while operating in this region, it must block, and let the system time catch up. Whenever a component executes a get, its local time is exactly equal to the system time.

2.3 Pia **objects**

Systems simulated under Pia consist of the following four types of objects:

- *components* - collections of interfaces and behavior (which can include system software), and typically used to represent physical components,
- *interfaces* - collections of ports, driver routines, and simple asynchronous event handlers,
- *ports* - which directly connect an interface to the outside world, and
- *wires* - which are used to connect ports.

2.4 Inheritance

Since the Pia language requires fairly low level specifications, it is important to provide some mechanism for sharing parts of specifications between similar components and interfaces. For example, we may want to use the same stream layer direct communication model for all types of network interfaces. For this reason, Pia provides a mechanism for inheritance.

The semantics of Pia inheritance are rooted in *prototype-based* object-oriented inheritance [2, 8], rather than *class-based.* This means that an object definition creates an instance of an object rather than a class of objects. When we say one object inherits from another, it means that the compiler actually copies parts from the parent object into the child. It is possible to define interfaces which never get imported or used directly. These are called *abstract interfaces* or *interface templates*.

The primary purpose of interface inheritance is to allow libraries of abstract interfaces to be created for groups of similar interfaces. For example, one can define an abstract *memory* interface, which contains only driver routines for fast communications, but leaves the details of bus level communication to the inheriting interface.

3 Pia **Language syntax**

The primary goal of the Pia language is to allow the designer to easily interact with the Pia tools, and to describe components and interfaces at many levels of detail. The language itself is similar to C++, but there are some substantial differences. First of all, Pia is not a general purpose object-oriented language; it has specific types of objects that can only be used in certain ways. Pia is a prototype-based language while C++ is a class-based language. Finally, most Pia objects are implicitly active, while C++ objects are passive.

The Pia runtime system is aware of two different types of methods which may be present in an interface:

- *driver routines (seqs)* - software functions that communicate with the hardware (although possibly through other driver calls). These are important because they allow a component to refer to an operation by name.
- *handlers* - asynchronous event or interrupt handlers.

Example 3.1 shows a Pia language description of a simple memory bus interface with a 32 bit data bus, a 32 bit address bus, and one line each for read and write. This interface would be used with a program that represents the software portion of an embedded system. An example might be a C program that manipulates a frame buffer in memory connected to this bus.

3.1 Explicit Timing and Synchronization

Pia depends on explicit code and action timing annotations to facilitate time accurate simulation. There are two timing primitives provided to the user, *advance(time)* and *sync()*. The *advance* primitive is used to advance the local clock, and the $sync$ primitive is used to put the currently active object to sleep until the system time equals local time. It is expected that in the near future, we will have tools to automatically insert these annotations into the code - so that the designer won't have to worry about them.

Synchronization in Pia (as in a discreet event simulator) is provided purely by simulation time-based event ordering. In this context, *events* include: sends, gets, and syncs.

Example 3.1 A simple interface description in Pia

```
interface memory {
  merge Bus {
    inout [32] DATA;
    inout [32] ADDRESS;
    out WR;
    out RD;
  };
  times { write_pulse_duration:6ns,
    address_setup_time:3ns,
    data_setup_time:3ns,
    data_hold_time:3ns };
  init {
    // Make sure we aren't driving the bus
    Bus.DATA.set_tri();
    Bus.ADDRESS.set_tri(); Bus.send();
  };
  seq Write (int A, int D) {
    Bus.ADDRESS = A; Bus.DATA = D;
    advance(address_setup_time);
    Bus.WR = 0; advance(write_pulse_duration);
    Bus.WR = 1; advance(data_hold_time);
    Bus.ADDRESS.set_tri(); Bus.DATA.set_tri();
  };
  seq Read (int A) {
    Bus.ADDRESS = A;
    advance(address_setup_time);
    Bus.RD = 0; advance(data_hold_time); sync();
    Bus.RD = 1;
    return(DATA);
  };
}
```

3.2 Dynamic "seqs" and driver abstraction

The mechanism for switching between different implementations of driver routines is provided through *dynamic seqs*. A *seq* name can refer to many multiple driver routines, while the choice of the actual routine is deferred until runtime. The criteria for this choice is an integer value which is maintained by each interface. This value describes the *runlevel* or level of communications detail at which the interface is running.

Example 3.2 Dynamically dispatched *Write* seqs

```
defval hardwareLevel 0;
defval sharedMemory 4;
  ...
  seq Write (int A, int D);

  seq [hardwareLevel] Write (int A, int D) {
    Bus.ADDRESS = A; Bus.DATA = D;
    advance(address_setup_time);
    Bus.WR = 0; advance(write_pulse_duration);
    Bus.WR = 1; advance(data_hold_time);
    Bus.ADDRESS.set_tri(); Bus.DATA.set_tri();
  };

  seq [sharedMemory] Write (int A, int D) {
    MemArray[A] = D;
    advance(address_setup_time +
              write_pulse_duration +
              data_hold_time);
  };
```

Example 3.2 shows two *seq*s, for Write. In this example, writes at run level "sharedMemory" (4) write the data directly to the memory array, instead of going through the simulated hardware to do so. Pia provides the mechanism for switching to optimal communications methods, but actually generating these optimal methods is the topic of [5].

3.3 Using inheritance

For a Pia object to inherit the characteristics of another, the **inherits** keyword is used in its definition as follows:

interface DRAM **inherits** memory { ...}

This indicates that the interface for DRAM inherits all of the characteristics of a memory interface, except those which are overridden by the DRAM interface specification.

4 Implementation

This section describes some of the implementation issues involved in building the Pia toolkit. The current implementation includes a Pia language compiler, a Ptolemy simulation domain as well as several simulated peripheral and test devices.

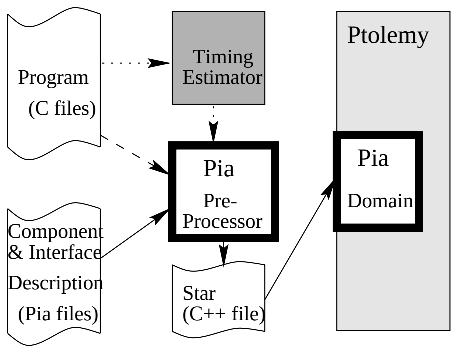

Fig. 4: Where Pia tools fit in

4.1 Pia Pre-processor

The Pia pre-processor translates Pia interface and component descriptions into Ptolemy stars.

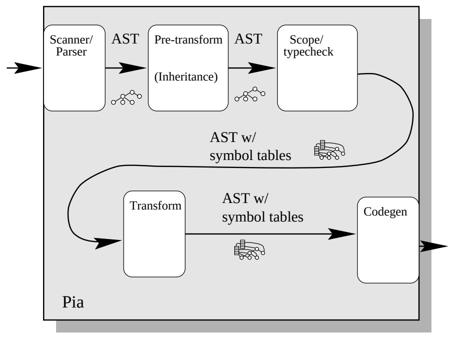

Fig. 5: Pre processor phases

The structure of the pre-processor is fairly standard (see Fig. 5). The scanner and parser read the file and produce an Abstract Syntax Tree (AST). This AST is then sent through several stages which behave as follows:

- the *pre-transform* stage resolves inheritance,
- the *scoping* stage checks each definition into its active scope (this is important, because each variable is later tagged with the appropriate scope, and moved into a Ptolemy star definition, thus preserving the correct behavior for non-reentrant code),

- the *transform* stage linearizes a component description, then breaks it into atomically executable pieces, so that it can be mapped to the Ptolemy fire-and-return model of execution (essentially, the description is broken apart at syncs, and at call sites where the callee contains a sync), and
- the *codegen* stage produces a Ptolemy star.

The resulting star can be loaded into Ptolemy in the Pia domain. The "scoping" and the "transform" stages require a pre-processor that actually understands the program structure, rather than a text replacement tool, such as CPP.

4.2 The Pia simulation domain

The Pia domain is quite similar to Ptolemy's standard DE domain, but with some important differences. Although the Pia domain is a timed domain the scheduler works with untimed components as well.

In the Pia domain, portholes automatically consume particles, and buffer the values so it is possible to check the current value of a wire. Pia Portholes also have a *transition()* method, so that a star can check to see if the current value is the result of a *recent* change in values.

The Pia scheduler contains $checkpoint$ and $restore$ methods for speculative communication. This allows a designer to safely use faster communication modes, even when the methods themselves are not certain to be safe. Essentially, the designer can set up the simulator so that it saves state occasionally, and if it detects a dependency violation, it restores the stars to a known safe state, and decreases the run level.

5 Experiments

This section describes two experiments performed with the Pia tools. In both examples, the software components were compiled for the simulator host, and as such, ran at near workstation speed. These experiments were performed on a Linux/Pentium-70 "workstation" with 36 M of physical memory.

5.1 Experiment 1: Page transfer

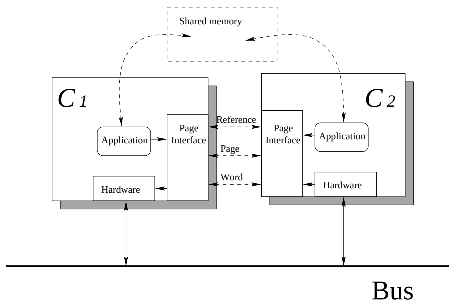

Fig. 6: Two communicating components

In this example, C_1 fills 8KB pages with data, then sends them to C_2 (see Fig. 6). This experiment presents a nearly ideal situation for Pia since there is very little computation, and lots of communication. We look at four separate communication models:

1. hardware - each word (4B) is sent to C_2 through the bus, with the full bus handshake protocol,
2. fast word - each word is directly transmitted to C_2, without going through the bus, but these writes are still synchronized through the event queue,

Mode	Transfer time
1	548.91 seconds
2	21.95 seconds
3	1.80 seconds
4	0.24 seconds

Tab. 1: Simulation times for 64 page writes

3. page transfer - the entire page is copied into a buffer, which is then transmitted to C_2, and
4. reference transfer - only a reference to the page is transmitted to C_2.

Note: There are applications where some modes may cause inter-component data dependency problems. For example, mode 4 might not be safe if C_1 writes to to the page buffer before C_2 can act on the contents.

Table 1 shows the experimental results for 64 page writes (2^{17} 4 byte words, or 2^{19} bytes). It is interesting that in this example, there is a marked speedup between mode 1 and mode 2 (about 25 times). Since the only difference between these two modes is that mode 1 performs all of the bus actions for a memory write, and mode 2 does not, we can infer that about 96$\%$ of the time in mode 1 is spent on this overhead. For a bus, this overhead is actually a function of the number of citizens on the bus, since each citizen may need to monitor all communications to determine which it needs to act on.

5.2 Experiment 2: Robot vision subsystem

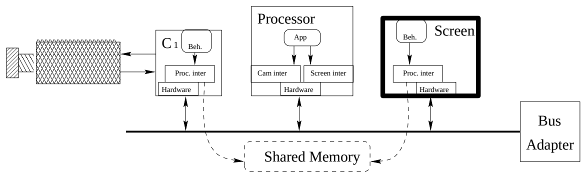

Fig. 7: Robot vision subsystem

This experiment tests the effective speedup in a system with a mixture of dense and sparse communication paths, and some non-trivial computation. Essentially, this is an image processing system which is loosely coupled to a robot's main processor. The main processor can issue commands such as "grab a new frame" and "send me the center of the brightest object" or "send me the beacon offset." where the "beacon offset" would be the horizontal position of a locater beacon, in degrees from image center.

This system includes a camera with a controller, a processor, and an LCD screen (see Fig. 7). The bus adapter generates an interrupt whenever it gets a command from the main processor. On receiving the interrupt, the image processor:

1. reads the command, and any data from the bus adapter,
2. performs the requested action, and
3. writes a return value to the bus adapter.

On each "grab a new frame" command, the processor also copies the new frame to the LCD screen. There are 3 effective paths in this system:

1. processor $\Leftrightarrow$ bus adapter (low density),
2. processor $\Leftrightarrow$ camera (high density), and
3. processor $\Rightarrow$ LCD. (high density).

Configuration	Transfer time
1	228.78 seconds
2	115.64 seconds
3	0.92 seconds

Tab. 2: Sequence times

For our experiments, we used two communication modes for each high density path: Hardware accurate mode and shared memory mode. Shared memory communication should be safe in this application because because both the camera and the LCD are slaved to the processor. This means that neither will read or write the memory unless the processor orders it.

Below are the specific configurations we tested.

1. All communications are through the bus and are interface accurate.
2. The camera and processor communicate through shared memory, and the rest of the communication is interface accurate.
3. The camera and the screen both communicate with the processor through shared memory, the rest of the system uses interface accurate communication.

Table 2 shows the times to complete the following command sequence in each of the configurations:

1. get a new frame, and
2. find the center of the brightest section of the center scan line.

It's interesting to note that by switching only one of the high density paths to shared memory mode we improved performance by less than a half. The main reason for the tremendous speedup between configuration 1 and configuration 3 is that the computation itself is occurring at near native speed, so it doesn't have a large impact on simulation time.

6 Future work

We are looking at ways to automatically generate fast communication models from high level descriptions of a system [5]. These can benefit from an understanding of the high level semantics of communication. This sort of automation should be especially useful when the Pia tools are used to validate the output of automatic synthesis tools such as Chinook [3, 4, 9].

While *speculative communication* (communication which may cause dependency violations, and thus need to be reversed) should allow us to use larger units of transfer than may be predictably safe, it can also cost *more* than if smaller units of transfer were used in the first place. This is because it is expensive to undo a communication once committed. We are looking into ways of statically determining reasonably large units of transfer, that will give us fast communication that will rarely need to be undone.

Pia currently offers only marginal software debugging support. We are looking at ways of modifying a debugger (such as gdb) so that it recognizes Pia structures, and can give better feedback.

7 Conclusion

Hardware-software co-simulation of embedded systems can perform poorly for many reasons, but usually, it is related to an excess of detail, *i.e.* the simulator is producing more detail than is actually required. Processor internal details can be abstracted away in most cases. Communication can present a bigger problem, since we don't always know in advance what details we will need, and which we can neglect. This paper presented techniques that gives a designer the ability to focus in on certain communications, while representing the rest with less detail, but still with the ability to generate a full system workload.

Although the speedup for a specific system will vary depending on the ratio between communication and computation, we expect the techniques embodied in Pia to lead to significant gains when the focus is on a limited part of the system.

References

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

[2] Borning, A. Classes versus prototypes in object-oriented languages. In *Fall Joint Computer Conference* (November 1986).

[3] Borriello, G., Chou, P., and Ortega, R. Embedded system co-design - towards portability and rapid integration. NATO, 1995.

[4] Chou, P., and Borriello, G. Software architecture synthesis for retargetable real-time embedded systems. In *Codes/CASHE '97* (1997).

[5] Hines, K., and Borriello, G. Optimizing communication in hardware-software co-simulation. In *Codes/CASHE '97* (1997), IEEE, ACM.

[6] Kim, K., Kim, Y., Shin, Y., and Choi, K. An integrated hardware-software cosimulation environment with automated interface generation. In *7th International Workshop on Rapid Systems Prototyping* (June 1996).

[7] Klein, R. Miami: a hardware software co-simulation environment. In *Proceedings. Seventh IEEE International Workshop on Rapid System Prototyping. Shortening the path from specification to prototyping* (June 1996).

[8] Lieberman, H. Using prototypical objects to implement shared behavior in object oriented systems. In *OOPSLA `86 Proceedings* (September 1986).

[9] Ortega, R., and Borriello, G. Communication synthesis for embedded systems with global considerations. In *Codes/CASHE '97* (1997).

[10] Rowson, J. Hardware/software co-simulation. In *Proceedings of the Design Automation Conference* (1994), pp. 439–440.

[11] Schnaider, B., and Yogev, E. Software development in a hardware simulation environment. In *33rd Design Automation Conference Proceedings* (1996), pp. 684–689.

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

[13] Živojnović, V., and Meyr, H. Compiled hw/sw co-simulation. In *33rd Design Automation Conference Proceedings* (1996), pp. 690–695.

[14] Valderrama, C. A., Nacabal, F., Paulin, P., and Jerraya, A. A. Automatic generation of interfaces for distributed c-vhdl cosimulation of embedded systems: an industrial experience. In *7th International Workshop on Rapid Systems Prototyping* (June 1996).

[15] Valderramma, C. A., Changuel, A., and Jerraya, A. A. Virtual prototyping for modular and flexible hardware-software systems. *Design Automation For Embedded Systems* (1996).

[16] Wilson, J. Hardware/software selected cycle solution. In *Proceedings of the international International Workshop on Hardware-Software Codesign* (1994).