A Software-Hardware Cosynthesis Approach to Digital System Simulation

Kunle Olukotun, Rachid Helaihel, Jeremy Levitt and Ricardo Ramirez
Computer Systems Laboratory
Stanford University, CA 94305

Abstract

This paper presents a software-hardware based simulation approach to digital system simulation. Our approach provides better performance than a software only approach and also works with high-level system models. The approach uses an optimizing simulation compiler to transform a hardware description language system model into a high performance simulator. The target architecture for the simulation compiler is a tightly coupled processor and field-programmable gate array (FPGA). The components of the simulation compiler are a high performance compiled-code software simulator, an automatic partitioner that partitions the system model between the processor and FPGA, and a scheduler that maximizes concurrent execution within the FPGA and between the FPGA and processor. We describe these components and show how they can be used to improve the performance of synchronous digital system simulation by up to a factor of two when compared to a high performance all software simulator.

1.0 Introduction

Simulation is used extensively to evaluate performance and to verify correctness of digital systems. As system designs become more complex, the level at which the system is initially specified is made more abstract in order to manage the complexity. Simulation techniques must keep up with these upward shifts in the abstraction level to ensure that systems designers can efficiently evaluate the performance and correctness of their ideas as early as possible in the design process. Recently, there has been a trend towards specifying systems using hardware description languages (HDLs). This trend is partly due to the widespread acceptance of logic synthesis tools and, to a lesser extent, high-level synthesis tools. This trend, coupled with the much greater demand for simulation performance resulting from the increase in system complexity, motivates the need for new simulation techniques that are optimized to simulate high-level system models as efficiently and as economically as possible.

In this paper we describe and evaluate a simulation approach that converts an HDL model into a high-performance simulator consisting of tightly coupled software and hardware components that execute on processor and FPGA architecture. Our approach uses compiled-code software simulation, accurate performance estimation, logic synthesis, software-hardware partitioning, and software-hardware scheduling to generate these

components. The remainder of the paper is organized as follows. In the next section we provide an overview of our simulation approach. Section 3.0 presents related work. Section 4.0 describes the compiled-code software simulator. Section 5.0 describes the hardware compilation process. Section 6.0 describes algorithms for scheduling and partitioning the software and hardware components. Section 7.0 presents performance results, and Section 8.0 presents conclusions and areas of future research.

2.0 A Software-Hardware Simulator

Software-hardware simulation of HDL models has two components: a system architecture and a simulation compiler. The system architecture must be a good target for compiled HDL models and the simulation compiler must be optimized to make the HDL models simulate efficiently on the target architecture. In this section we give an overview of our simulation architecture and compiler.

2.1 Simulation Architecture

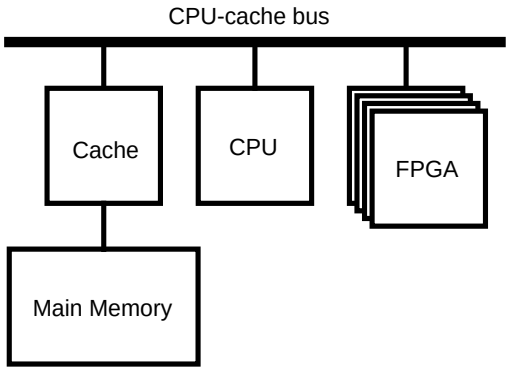

Figure 1. The simulation architecture.

The target architecture is shown in Figure 1. The processor has one or more FPGA chips connected by the same bus as the processor cache. This implementation detail is important because it means that the communication latency and bandwidth between the processor and cache and between the processor and FPGA chips are comparable. The FPGA chips accelerate HDL simulation in two ways. Firstly, the FPGAs can accelerate parts of the HDL model that take less time to execute in an FPGA than in a general purpose CPU. Control sections of HDL models typically fall into this class. Secondly, the parallel execution among the CPU-based and FPGA-based parts of the HDL model accelerates simulation even further.

The primary purpose of the FPGAs are to accelerate simulation. However, the simulation architecture could also be used as the control and data manipulation processor in an embedded system. In such a system, the FPGAs might also serve as the interface logic to sensors and actuators. For certain applications, this system might provide sufficient perfor-

mance such that the simulator itself could be used in place of a custom hardware implementation.

2.2 Simulation Compiler

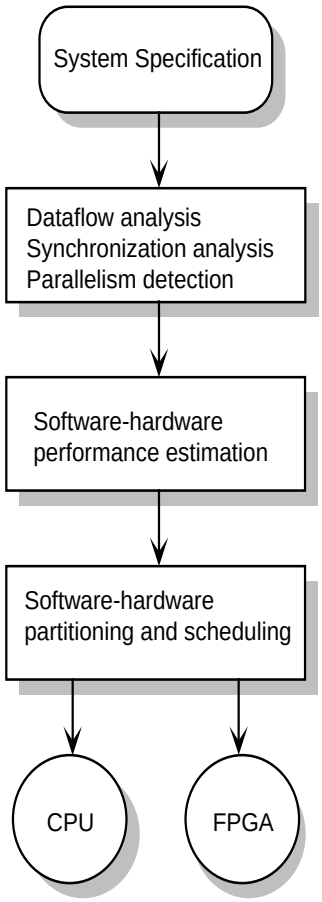

Figure 2. The simulation compiler.

The overall goal of the simulation compiler is to produce software and hardware components that execute efficiently on the target architecture. Figure 1 shows an overview of the simulation compiler. The input to the compiler is an HDL system specification. This specification is converted into an intermediate form represented by a hybrid abstract-syntax tree and a dataflow graph. Analysis of the synchronization and data dependencies in the intermediate form is used to extract the concurrency in the specification. This information enables us to generate software-only and mostly hardware versions of the simulator. By accurately estimating the performance of these two versions of the simulator, we determine the execution time of a particular part of the HDL model in hardware or in software. This data provides the basis for partitioning the model between the CPU and the FPGA. The partitioning and scheduling algorithms attempt to maximize performance with a given number of FPGA chips. Parts assigned to the FPGA should result in an execution schedule with the shortest execution time while not exceeding FPGA chip limitations.

One of the key features of the simulation compiler is that it is capable of producing efficient simulators for a single processor, or for a processor and FPGA; furthermore, the compiler will optimize the simulator for the number of gates in the FPGA chips. This provides a flexible way to change the cost/performance of the simulator by varying the number of FPGA chips.

3.0 Related Work

Previous work in the area of software/hardware co-synthesis has focused on the design of embedded systems. The following discussion of previous work will be limited to those co-synthesis approaches that start with a single program specification of the system and perform the partitioning between software and hardware automatically.

In the Cosyma system [7] the functionality of the system to be synthesized is described in a superset of C called C^X. C^X extends C with timing constraints and tasks. The target architecture of the Cosyma system is a micro-controller and co-processor that communicate via a shared memory. Cosyma maps a C^X description onto the target architecture by assigning basic blocks of the description to run in software on processor or in hardware on the co-processor. The partitioning is based on a cost-function that includes the estimated benefits of moving a particular block from software to hardware and the estimated communication cost. The actual partitioning is performed using a simulated annealing algorithm.

The advantage of the Cosyma approach is that the use of a general purpose programming language makes it easy to describe complex systems. However, because the approach does not currently overlap the execution of the micro-controller and the co-processor, it does not exploit one of the main performance enhancements possible with a co-processor. Furthermore, the use of estimates of software performance does not accurately account for the effects of an optimizing compiler on the performance of the software. This results in software-hardware systems with poor speedup and sometimes a slowdown over the software-only system.

The approach presented by Gupta and De Micheli [9] is similar to the approach presented in this paper. The approach uses a hardware description language, HardwareC, to specify the system description. Hardware C includes program constructs for specifying delay and execution rate constraints as well as constructs for explicitly expressing concurrency. The goal of the system is to reduce the cost of implementing a system using an ASIC by combining an off-the-shelf microprocessor and an ASIC. The automatic partitioning is accomplished by starting with an initial partition in which all program constructs with unspecified delay are in software and the rest are placed in hardware. An iterative improvement algorithm moves operations from hardware to software to reduce the cost of the system while meeting the delay and rate constraints. As in the Cosyma system, the performance of the software is only estimated without actually executing the code. This sort of software performance estimation makes it difficult to get an accurate measure of software performance and does not include the effects of optimizing compiler technology.

4.0 Software Compilation and Performance Estimation

This section describes the software simulation compiler, the starting point of our co-synthesis approach. We first discuss the algorithms used by the compiler and then follow with an example showing the compilation of a very simple HDL model.

Software generation is accomplished with a Verilog to C compiler (VCC) that compiles a Verilog HDL model into a C program. Verilog has been designed for fast event-driven simulation, and contains constructs that have no clear analogues in hardware or procedural programming languages. By restricting the Verilog programs that VCC will accept to those that describe synchronous digital systems, we guarantee that the Verilog program can be compiled by VCC. The output of VCC is a statically scheduled C program that has the same behavior as the Verilog model running on a event-driven Verilog simulator but achieves much higher simulation performance. There has been previous work in compiled code simulation [4, 10, 12] which was based on input descriptions that were gate-level netlist representations of circuits [4, 12] or RTL models in a simple description language [10]. VCC advances the work in compiled code simulation because it generates a compiled code simulator from Verilog. Compiling Verilog programs requires a significant amount of analysis that is not required for input descriptions with simple execution semantics. VCC must perform this analysis even though it cannot compile the complete Verilog language.

The analysis performed during the VCC compilation process has three main steps:

1. Dataflow graph construction parses the Verilog program and produces a dataflow graph that is combined with an abstract syntax tree.
2. Static scheduling analyzes the dataflow graph and generates an execution schedule for the graph that preserves the semantics of the Verilog program
3. C code generation produces C code.

We describe these steps in more detail in the following sections.

4.1 Dataflow Graph Construction

The input to VCC is a synchronous Verilog description. Verilog programs have a syntax that is similar to C, but semantics which are very different. Verilog programs are composed of *modules*. These modules may be instantiated inside of other modules to create a hierarchy that represents the structure of the hardware system. In our Verilog programs, modules contain two types of concurrent process statements or *concurrent blocks*; these are *always-blocks* and *continuous-assignments*. An always-block is a group of statements with sequential semantics that executes whenever an event occurs that is in the block's *activation list*. A continuous-assignment is an assignment whose left hand side continuously reflects the current state of the variables on the right hand side.

The dataflow graph is created by parsing the Verilog program, flattening the hierarchy, and creating a directed graph. The vertices of the graph represent variables, always-blocks or

continuous-assignments. The edges of the graph represent uses or definitions of the variables. There is a directed edge between a variable vertex and a concurrent block vertex for each variable that appears on the right hand side of a continuous assignment or in the activation list of an always-block. There is also a directed edge between each concurrent block vertex and each variable vertex that is assigned in a concurrent block.

4.2 Static Scheduling

To make static scheduling of the concurrent blocks possible, the potentially cyclic dataflow graph must be converted into a directed acyclic graph (DAG). A DAG is created by breaking the feedback loops in the dataflow graph at the clock signals. Clock signals are identified as variables that appear in a special always block named “clock.” This clock block also defines the clock signal transitions and is the only block that can contain delay statements. Every always-block in the program is dependent upon at least one clock signal transition. The always-block vertices and continuous-assignment vertices associated with a transition must form a DAG in the dataflow graph. If they do not form a DAG then no static schedule of blocks is feasible; VCC will reject the Verilog program.

A schedule for the process statements is generated by selecting the first clock transition in the clock-block. The effect of this transition on other signals is propagated by constant propagation of the transition to all the dependent blocks. in the dataflow graph [2]. The always-blocks and continuous assignments that are fired directly or indirectly by the transition are then scheduled by topologically sorting the dataflow graph [6]. This process is repeated for all the clock transitions in the clock block.

4.3 C Code Generation

VCC attempts to generate C code that has as little run time as possible. To achieve this, C code is generated so that it can be optimized by the C compiler. VCC performs optimizations such as packing multi-bit signals into a single machine word and eliminating bit field selection that require specific knowledge of Verilog semantics. Standard C compiler optimizations like constant propagation, dead-code elimination and common sub-expression evaluation are left for the C compiler. However, VCC does perform a substantial amount of copy propagation to eliminate redundant assignments resulting from the structural hierarchy in the model. C code is generated by emitting the code for a clock transition and for each of concurrent blocks appearing in the static schedule for that clock transition. This process is repeated until code has been generated for all the clock transitions.

4.4 A Simple Example

```
module top();
reg [1:0] state;
reg phi1;

initial
 state = 2'b00;

always
begin clock
 phi1 = 0;
 #5
 phi1 = 1;
 #5
end

always @(posedge phi1)
begin state_machine
 case (state)
   2'b00: state = 2'b01;
   2'b01: state = 2'b10;
   2'b10: state = 2'b11;
   2'b11: state = 2'b00;
 endcase
end

endmodule
```

Figure 3. A simple Verilog description of a clocked state machine.

Figure 3 shows a simple Verilog description of a clocked state machine. It has a single module called `top` which contains an initial block and two always-blocks. The initial-block initializes the variable `state`. The special `clock` always-block defines the clock signal `phi1`, and the `state_machine` always-block is activated on the positive edge of `phi1`.

Figure 4 shows the dataflow graph that corresponds to the state machine description. The `clock` always-block defines the `phi1` variable which is used by the `state_machine` always-block. The `state_machine` always-block defines the `state` variable. This dataflow graph is already a DAG; thus, scheduling the evaluation of the vertices is straightforward. The C code that results from the scheduling and code generation phases is shown in Figure 5.

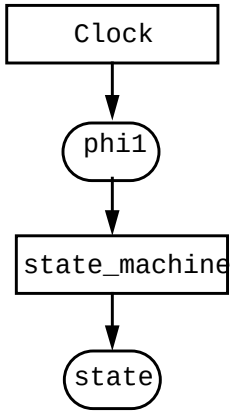

Figure 4. The dataflow graph for the sate machine description.

```
#include <stdio.h>

struct top_s {
 unsigned long phi1;
 unsigned long state;
};


struct top_s top;

main() {

 initial ();
 simulate ();

}


void initial () {

 top.state = 0;
}
```

```
void simulate () {

 while (1) {
  top.phi1 = 0;
  top.phi1 = 1;
  switch (top.state) {
   case 3:
     top.state = 0;
     break;
   case 2:
     top.state = 3;
     break;
   case 1:
     top.state = 2;
     break;
   case 0:
     top.state = 1;
     break;
  }
 }
}
```

Figure 5. The C code for the state machine description.

4.5 Software Performance Estimation

The performance of the all-software based simulator is impressive. On the Verilog models we have experimented with we are able to achieve speedups of between 150–300 times

faster than Verilog-XL1.6 [5]. Experience with this simulator indicates there are software optimizations that will further improve Verilog simulation performance; however, the focus here is on the additional performance improvements that are possible using software-hardware co-synthesis.

The granularity for partitioning into software or hardware is a block of Verilog code that does not contain any event-control. This ensures that a block assigned to hardware will execute to completion without requiring any intermediate communication with blocks running in software. Event-control-free blocks include continuous assignment blocks and simple always blocks. Even though our system cannot directly handle complex always blocks with nested event control, these types of blocks can be handled by splitting them into event-control free components. In the rest of the paper, *blocks will be used to refer to event-control free components*.

The purpose of software estimation is to provide the average execution time and the shortest execution time of the C code associated with each block. This requires a careful analysis of the object code produced by the C compiler for the C statements associated with each Verilog block. To make this feasible, interleaving of statements from different blocks is disallowed, and compiler optimizations are restricted in scope. In practice, most compiler optimizations are unaffected, and the resulting code is almost as fast.

Profiling of the software simulator is used to estimate the average execution time of the statements associated with each Verilog block. Profiling is more accurate than static estimation because it captures the dynamic frequencies with which branches are taken in the code. To ensure that profiling provides accurate performance estimates, the behavior of the model should accurately reflect the real behavior of the model. This requires the use of large and varied input data sets during profiling. If efficient profiling techniques are not used the collecting this information could take a very long time.

The profiling technique used to obtain the average software execution time is the object code annotation tool *pixie* [13]. This tool captures the dynamic execution frequencies for each basic block in the object code. By analyzing the execution time of each basic block on the target processor architecture and multiplying this time by the execution frequency of the block, an exact count of the number cycles executed by each basic block can be found. This profiling technique is very efficient; it slows down program execution time by only a factor of four. This technique also accurately accounts for pipeline stalls; however, it does not account for the performance of the memory hierarchy. The software execution time of each block can be calculated by adding up the execution cycles for all the basic blocks in the object code that are associated with the block.

To estimate memory-hierarchy performance, pixie is used to generate an address trace. This address trace is used to drive a cache simulator that models the processor memory-hierarchy performance. However, unlike CPU performance, it is almost impossible to assign the time spent in the memory hierarchy to individual blocks. Fortunately, in our experiments, performance losses in the memory hierarchy were small. We suspect that with much larger models the memory hierarchy could have a significant effect on the software performance.

To determine the shortest execution time of a block, the object code produced for the block is analyzed in detail. The shortest path-length through the block is determined by assuming that all instructions execute in one cycle and by finding the sequence of taken or untaken branches that minimizes the number of instructions executed in the block.

5.0 Hardware Synthesis and Performance Estimation

The hardware portion of the software-hardware simulator is implemented with Field Programmable Gate Arrays (FPGA). FPGA's are an ideal implementation medium for simulation because they allow the design to be changed easily. The Xilinx FPGAs used in our experiments consist of an array of configurable logic blocks (CLBs) that are interconnected by a hierarchy of routing channels, and surrounded by a perimeter of programmable Input/Output Blocks (IOB) [15]. In the remainder of this section, we describe the procedure for implementing a Verilog HDL model with FPGAs. We begin by explaining the interface between the processor and the FPGA chips and the internal organization of each of the FPGA chips.

5.1 The Processor/FPGA Interface

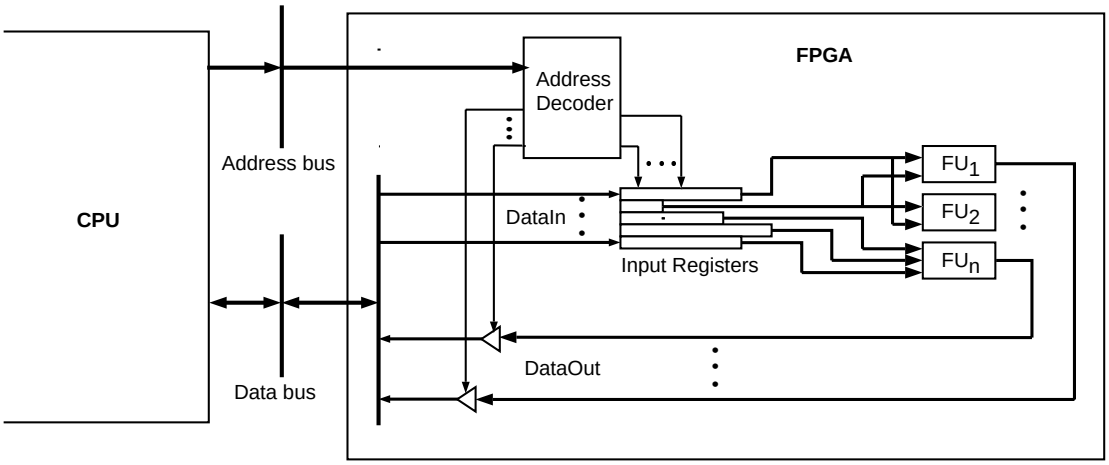

Figure 6. The internal structure of the FPGA and the interface between the FPGA and the processor.

With current FPGA densities it is possible to emulate complex logic designs on a single FPGA; however, many of these implementations cannot fully utilize the FPGA gates because there are insufficient I/O pins [3]. To overcome this problem, we have designed a memory-style interface in which the FPGA chips communicate with the CPU over common address and data busses. Figure 6 shows a block diagram of the interface between the functional units in the FPGA and the CPU. The FPGA is organized as a register file which feeds independent functional units that implement the operation of a particular block. The registers contain the operands for the functional units. Because the functional units are

independent, once the operands have been loaded all functional units can operate in parallel. Even though the data bus is 32 bits wide, the number of bits in each register corresponds to the width of the input operand it contains. In addition, functional units with common input operands can share registers. The variable register bit widths and register sharing among functional units reduces the FPGA resources needed to implement this organization.

To activate a particular functional unit, the CPU places the data on the data bus, then places the address of the register on the address bus. The decoder in the FPGA will turn the internal data bus into an input bus, and enable the addressed register to latch the data on the bus. After the functional unit has completed execution, the CPU can read the output by applying the address of the appropriate output. The decoder will enable the addressed tri-state bus and drive the output from the internal bus onto the data bus. Since the FPGA is based on SRAM technology, we assume that the time it takes to read or write one of the registers in the FPGA is on the same order as an SRAM chip access time. With a 50 MHz CPU, we allot one clock cycle for each CPU-to-FPGA access.

5.2 Hardware Synthesis and Performance Estimation

Logic synthesis tools from Synopsys are used to convert Verilog blocks into FPGA hardware [14]. The input description to the synthesizer for each block includes the interface logic that will be connected to it on the FPGA. This ensures that the area and critical path delay estimates reported by the synthesizer represent the true costs of placing the block in an FPGA chip. The synthesizer reports the area cost in terms of CLBs and the critical path delay in nanoseconds for each synthesized block. To minimize the synthesizers run time, no timing or area constraints are used. Without these constraints, the FPGA synthesizer does not optimize the blocks for speed or area. To account for the delay of routing wires, which represents a large fraction of the delay in an FPGA, before the CLBs have been placed and routed, the FPGA synthesizer uses a statistical approximation method based on a simple wire load model.

The synthesis tools are not able to synthesize all blocks. The synthesizer is designed to translate register transfer level descriptions into gate level descriptions and it is unable to handle certain behavioral Verilog statements. Our verilog models are intended for simulation and consist mainly of behavioral code; inevitably there are certain blocks for which the synthesizer is unable to find a gate level description. Although a high-level synthesizer would be less restrictive in the types of Verilog statements it could translate, it too would not be able to completely translate all Verilog programs to hardware since Verilog can be used to describe systems that do not have any reasonable all-hardware representation.

6.0 Software-Hardware Cosynthesis

Once the execution times for all blocks in both software and hardware and the area cost in hardware for all blocks have been determined, each block must be placed in either hardware or software so that the overall execution time of the simulation is minimized. This is a difficult problem because it requires the combined solution of partitioning and scheduling sub-problems.

The partitioning of blocks between software and hardware affects scheduling in two ways. Firstly, the time required for a block to execute depends on whether the block is placed in software or hardware. Secondly, the execution overlap among hardware blocks, and between software and hardware blocks depends on the placement of all blocks. This makes it difficult to evaluate the performance of a particular partition without considering scheduling at the same time. Yet, it is impossible to produce an execution schedule without first selecting a partition.

Our solution to this aspect of the cosynthesis problem is to efficiently find a near optimal solution to the scheduling sub-problem and then to use this scheduling algorithm in the inner loop of an iterative partitioning algorithm. The algorithms for scheduling and partitioning are described in the following sections.

6.1 Software-Hardware Scheduling

Software-hardware scheduling creates an execution schedule that minimizes execution time. The algorithm starts with a predetermined partition of blocks between hardware and software and uses the dataflow-graph generated by VCC to produce a schedule for the CPU. At any point during a simulation, the CPU can be in one of four states: executing a software block, writing arguments to a hardware block, reading results from a hardware block, or waiting for a hardware block to finish. Our software-hardware scheduling algorithm attempts to find an execution order for hardware and software blocks that minimizes the time the CPU spends in the waiting-for-hardware-blocks-to-finish state. Since the amount of CPU time spent in the other three states is fixed, minimizing the time spent in the waiting-for-hardware state also minimizes the total execution time.

The dataflow constraints between blocks restrict the order in which blocks can be executed. The CPU imposes an order on the execution of blocks which satisfies these restrictions, since each block is either executed entirely by the CPU or has it arguments written by the CPU and its results read back by the CPU. Software blocks have their execution serialized by the CPU and thus are easily ordered. Hardware blocks, however, execute in parallel with each other and asynchronously. The CPU enforces a correct order of execution for hardware blocks by serializing *communication* with the FPGA. Time spent by the CPU in the waiting-for-hardware state is minimized by maximizing the parallel *execution* among hardware blocks and between software and hardware blocks.

Finding the optimal solution to the software-hardware scheduling problem is intractable; however, for the problems encountered in practice our algorithm, using simple heuristics, quickly finds near-optimal approximate solutions. Since the optimal solution can be no

better than a solution in which the CPU is never in the waiting-for-hardware state, it is possible to bound the error between our approximate solution and the optimal solution.

Our scheduling algorithm makes the following assumptions:

1. Once a software block has begun execution, it is allowed to finish without interruption. The execution of a software block cannot be interrupted in order to read the results from the FPGA or to write arguments to the FPGA.
2. Where necessary, NOPs are used to ensure that the CPU never reads results from the FPGA prematurely. These NOPs are executed regardless of the dynamic execution behavior of software blocks.
3. CPU to FPGA reads and writes always take two cycles. The two cycles is a result of assuming that in addition to the one cycle transfer time between the CPU and FPGA, the CPU also has to load the argument from or store the result in memory. While there are cases where another instruction must be executed to calculate the source or destination address, there are also cases where a load or store between the CPU and memory is not necessary because the value is already in the register file or is immediately used by the CPU.
4. A 50MHz CPU clock frequency is used to calculate the number of cycles hardware blocks take to execute.

6.1.1 Scheduling Algorithm

The basis of the simple algorithm used to schedule the software and hardware blocks is list scheduling [1]. List scheduling is computationally cheap and an effective technique for a large class of problems [8]. If at a any point it is possible to communicate the arguments for more than one hardware block, then priority is given to the hardware block with the longest execution time. The rationale for this heuristic is to allow the slowest hardware blocks to execute while the CPU is communicating arguments to the FPGA and reading back results for the faster hardware blocks. For software blocks, which execute serially, priority is given to those with more descendents in hardware. This is done to promote maximum parallelism between hardware blocks.

Our algorithm maintains three sets: a set of unscheduled blocks ($\mathbf{S}_{un}$), a set of executing blocks ($\mathbf{S}_{ex}$) and a set of scheduled blocks ($\mathbf{S}_{sch}$). Listed below is the pseudo-code:

```
S_all = {set of all blocks};
S_un = S_all;
S_ex = ∅;
S_sch = ∅;
Pred(x) ::= {y ∈ S_all | block y is a predecessor of block x};
t = 0;

while (S_sch ⊂ S_all) {
   forall x ∈ S_un ordered by execution time {
      if (Pred(x) ⊆ S_sch && x is a hardware block) {
         schedule CPU to write arguments for x;
```

```
            t = t + (time to write arguments for x);
            S_un = S_un − {x};
            S_ex = S_ex ∪ {x};
            x.time when execution finishes = t + (time for x to execute);
        }
    }
    if (∃(x ∈ S_ex) where (x.time when execution finishes ≤ t) {
        schedule CPU to read results from x;
        t = t + (time to read results from x);
        S_ex = S_ex − {x};
        S_sch = S_sch ∪ {x};
    }
    else if (∃(x ∈ S_un) where Pred(x) ⊆ S_sch && x is a software block) {
        schedule CPU to execute x;
        t = t + (shortest time to execute x);
        S_un = S_un − {x};
        S_sch = S_sch ∪ {x};
    }
    else {
        t_idle = min{x.time when execution finishes | x ∈ S_ex} - t;
        schedule CPU to wait t_idle seconds for hardware;
        t = t + t_idle;
    }
}
```

Software-blocks require a variable number of cycles to execute; their execution time is dependent on dynamic arguments. As discussed in Section 4.5, the shortest execution time in software and the average execution time in software are determined for each block. To guarantee that the results from hardware blocks are never read before they are available, the scheduler assumes that software blocks always execute in the shortest possible time. While this pessimistic assumption may result in the CPU spending more time waiting for hardware, in practise we found it has little effect. The average schedule length can be calculated by using the average execution times for software-blocks that appear in the final schedule.

6.1.2 Scheduling Results

Despite the relative simplicity of our scheduling algorithm, it produced close to optimal schedules for the examples we considered. Table 1 is a breakdown of processor time for the state machine, the unpipelined CPU, and the pipelined CPU models scheduled assuming unlimited hardware resources. (These models are described in detail in Section 7.0.)

	Software Execution	Software-Hardware Communication	Waiting for Hardware
State machine	83.3%	16.7%	0.0%
Unpipelined CPU	29.0%	70.1%	0.9%
Pipelined CPU	52.5%	46.8%	0.7%

Table 1. A breakdown of best execution times with unlimited hardware

Table 1 shows that the CPU spends almost no time waiting for hardware blocks to finish executing and thus the schedules produced are very close to optimal. The success of our scheduling algorithm is due in part to the nature of the Verilog models we experimented with. They have plenty of explicit parallelism and very few data-dependencies between blocks.

6.2 Software-Hardware Partitioning

The software-hardware partitioning algorithm assigns each block to either software or hardware so that overall execution time is minimized. The algorithm has two phases: initial partition construction and partition improvement. These phases are described in detail below.

To guide the construction of a partition the average software execution time (t_{SW}), hardware execution time (t_{HW}), and software-hardware communication time (t_{com}) for each block are used to divide the blocks into three sets. The members of each set are determined by the following three inequalities:

1. $t_{\mathrm{com}} > t_{SW}$
2. $t_{SW} > t_{\mathrm{HW}} + t_{\mathrm{com}}$
3. $t_{\mathrm{HW}} + t_{\mathrm{com}} > t_{SW} > t_{\mathrm{com}}$

Blocks that satisfy the first inequality are assigned to the software set ($\mathbf{S}_{sw}$). These blocks can never benefit from being placed in hardware. Blocks that cannot be synthesized into hardware also belong to this set. Blocks that satisfy the second inequality are assigned to the hardware set ($\mathbf{S}_{hw}$). The execution time will always improve if these blocks are implemented in hardware. Blocks that satisfy the third inequality are assigned to the software-hardware set ($\mathbf{S}_{hw \vee sw}$). The effect on the execution time due to the placement of these blocks can be either beneficial or detrimental depending on the placement of other blocks.

The first phase of the partitioning algorithm constructs an initial partition using the three sets of blocks and constrained by the amount of FPGA resources available. The software set blocks are immediately assigned to software. Initially the hardware and software-hardware blocks are also placed in software. The partitioning algorithm then tries moving each of the hardware and software-hardware blocks into hardware and measures the improvement in execution time after each move. The block that resulted in the greatest speedup when it was the only block in hardware is assigned to hardware. The process then repeats; each of the remaining unassigned hardware and software-hardware blocks are again moved into hardware. The block that resulted in the greatest speed up when it was moved into hardware along with the block already fixed there is itself fixed in hardware. This process iterates until either no moves result in an improved execution time or until no more blocks can fit in the FPGA. The pseudo-code for this phase of the algorithm is given below:

$\mathbf{S}_{fixed-in-sw} = \mathbf{S}_{sw}$;
$\mathbf{S}_{fixed-in-hw} = \varnothing$;
$\mathbf{S}_{placed-in-sw} = \mathbf{S}_{hw} \cup \mathbf{S}_{hw \vee sw}$;

```
Schedule(HW_blocks, SW_blocks)::= the execution time for the partition;
Measure(execution time, fpga area) ::= best time - execution time;
available fpga area = Size of FPGA in CLBs;
best area = 0;
best measure = Measure(Schedule(∅, S_sw ∪ S_hw ∪ S_hw∨sw), 0);

repeat {
   best move = ∅;
   forall X ∈ S_placed-in-sw where (X.fpga area < available fpga area){
      current time = Schedule(S_fixed-in-hw ∪ {X}, (S_fixed-in-sw ∪ S_placed-in-sw) - {X});
      if (Measure(current time, X.fpga area)> best measure){
         best move = {X};
         best area = X.fpga area;
         best measure = Measure(current time, X.fpga area);
      }
   }
   S_fixed-in-hw = S_fixed-in-hw ∪ bestmove;
   S_placed-in-sw = S_placed-in-sw - bestmove;
   best time = Schedule(S_fixed-in-hw ∪ bestmove, (S_fixed-in-sw ∪ S_placed-in-sw) - bestmove);
   best measure = 0;
} until (best move == ∅)
```

The second phase of the partitioning algorithm iteratively improves upon the initial partition. Each block from the *fixed-in-hardware* set is moved back into the *placed-in-software* set. The moved block is kept in software as the algorithm from the first phase tries moving other blocks from the *placed-in-software* set into hardware to fill the space just vacated. The partition that results in the greatest speedup is used as the new initial partition, and the procedure is repeated. This second phase finishes when a partition is found where no single element can be removed from the *fixed-in-hardware* set without degrading performance. Since a shorter execution schedule is being found on each iteration, the algorithm is guaranteed to eventually converge on a locally optimal solution. While theoretically this convergence could be slow, it was quite rapid for our example models.

To determine the sensitivity of the final solution to the criterion used to select blocks for movement into the *fixed-in-hardware* set, we experimented with several different heuristics. In addition to using the “best-improvement” measure given in the pseudo-code above, we also experimented with “smallest-block” and “best-improvement-per-CLB” measures. The pseudo-code for these measures is given below.

“smallest-block” measure:

```
Measure(execution time, fpga area) = 1 / fpga area;
```

“best-improvement-per-CLB” measure:

```
Measure(execution time, fpga area) = (best time - execution time)/ fpga area;
```

As we will show in the results section, these heuristics do not significantly affect the partition that the iterative improvement algorithm converges on.

7.0 Performance Results

The performance results from experiments with our software-hardware simulation compiler are modest. The Verilog example programs we have experimented with range in complexity from a simple state machine, similar to the one shown in Figure 3, to a pipelined processor that executes a subset of the MIPS instruction set architecture (ISA) [11]. Table 2 shows the key characteristics of the three Verilog example programs. The first program models a simple state machine. The second program is an unpipelined CPU that executes a subset of the MIPS ISA. The last program is a pipelined version of the same CPU.

Speedups were calculated by comparing the execution times for the all-software simulations compiled with full compiler optimizations to the execution times for the software-hardware simulations as estimated using the profiling data gathered in Section 4.5.

	Lines of Verilog code	Number of blocks
State machine	56	4
CPU	1417	37
Pipelined CPU	1652	48

Table 2. Characteristics of Verilog test programs.

	Number of CLBs	Speedup over software
State machine	21	1.07
Unpipelined CPU	1013	2.76
Pipelined CPU	1174	2.04

Table 3. Software-hardware simulation performance with unlimited hardware.

Assuming unlimited FPGA resources, our co-synthesis approach achieves speedups over the all-software simulator ranging from 1.07 for the trivial state machine to 2.04 and 2.76 for the CPUs. For the unpipelined model the results of our algorithm were compared with the optimal result found through exhaustive search; the two results were the same. These speedups represent the maximum speedup possible with our proposed approach. The speedups are somewhat low because the all-software implementation was able to benefit from compiler optimizations that we did not incorporate into the software-hardware implementations. The software-hardware implementations are up to four times faster than an unoptimized all-software implementation.

For the limited FPGA resources case, we present the results of experiments that vary the amount of FPGA resources and the heuristic used to generate the partition. These results show how the speedup of the software-hardware approach is related to the amount of FPGA resources available and how the speedup achieved by the iterative improvement partitioning algorithm is affected by the heuristic used.

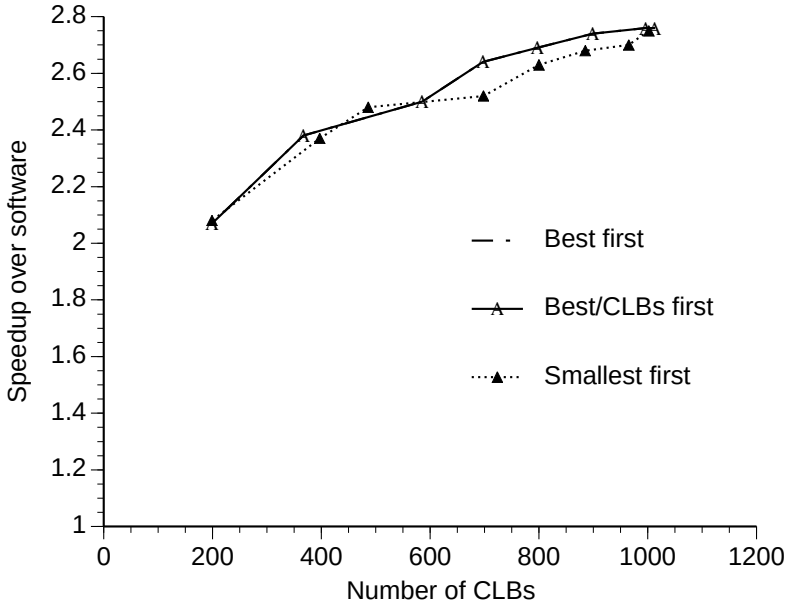

Figure 7. Speedup versus number of CLBs for the unpipelined CPU model using different heuristics for partitioning.

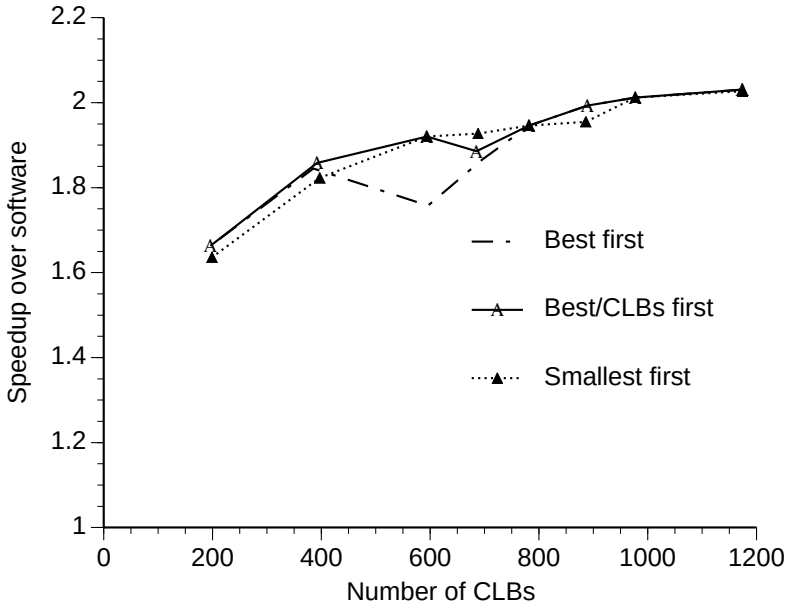

Figure 8. Speedup versus number of CLBs for the pipelined CPU model using different heuristics for partitioning.

Figure 7 and Figure 8 show the speedup versus number of CLBs of the unpipelined and pipelined CPU models using the three heuristics described in Section 6.2. Most of the speedup of the software-hardware simulator is achieved with a small number of CLBs. For the unpipelined CPU model a factor of 2.07 speedup is achieved with 200 CLBs. Using

six times as many CLBs only increases the speedup to 2.76. Similarly for the pipelined model a factor of 1.663 speedup is achieved with 196 CLBs. Using 1174 CLBs only increases the speedup to 2.031. Today, medium size FPGAs contain 250 CLBs. The largest FPGAs currently available have 600 CLBs.

Figure 7 also compares the performance of the different heuristics used by the iterative improvement partitioning algorithm. “Best-improvement-per-CLB” and “best-improvement” have nearly identical results, while “smallest-fit” performs slightly worse. This indicates that the iterative improvement algorithm is relatively insensitive to the heuristic used, but that a performance directed heuristic is preferable to one that conserves FPGA resources.

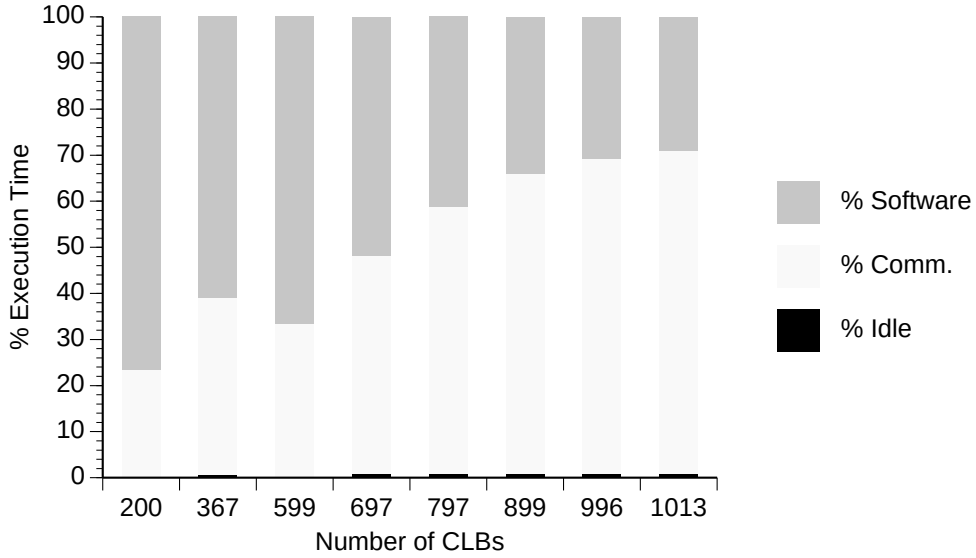

Figure 9. CPU execution time broken down into categories for the unpipelined CPU using the “best-improvement” heuristic.

Figure 9 and Figure 10 provide a breakdown of CPU execution time for the unpipelined and pipelined CPU models. As the number of CLBs used increases, the ratio of time spent communicating between the CPU and the FPGAs increases, limiting the speedup gained by placing more blocks in hardware. The ratio of CPU time devoted to communication increases as more blocks are added to hardware both because more communication is required and because fewer blocks are being executed in software.

For the unpipelined model the ratio of time spent communicating increases from 23.4% with 200 CLBs to 70.1% with 1013 CLBs. This sharp increase in communication time negates most of the benefits of the extra hardware. For the pipelined model the ratio increases from 12.3% with 196 CLBs to 46.7% with 1174 CLBs.

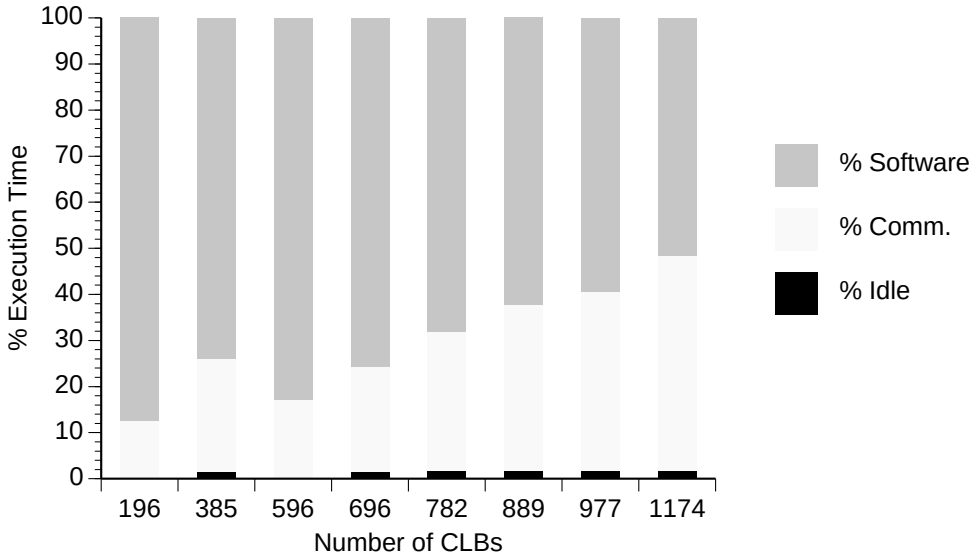

Figure 10. CPU execution time broken down into categories for the pipelined CPU using the “best-improvement” heuristic.

8.0 Conclusions and Future Work

This paper has described a fully automated software-hardware cosynthesis approach to the simulation of Verilog HDL models. Our approach is based on accurate measurements of execution times in software and hardware. This allows the performance benefits of placing a block in hardware or software to be accurately evaluated. By combining partitioning with an efficient scheduling algorithm we are able to place blocks in hardware or software so that performance is maximized. For the case of unlimited FPGA hardware our partitioning algorithm produced an optimal partition for all the models that we experimented with. Our results show that using these techniques we can achieve modest, but significant speedups over all-software simulation. Furthermore, in our experiments most of the performance benefits of a software-hardware simulator were achieved with a single FPGA chip.

There are areas in which our approach requires further refinement. The results of Section 6.1.2 show that communication severely limited the speedup possible with more FPGA resources; communication costs significantly degraded the performance of our software-hardware simulator. Thus we see reducing communication overhead as an important goal.

To reduce communication overhead, the partitioning algorithm could be enhanced by using the dataflow analysis from VCC to partition blocks between software and hardware such that communication is considered and minimized where beneficial. Ideally, once the CPU has written an argument to the FPGA we would like that argument to reused by as many hardware blocks as possible. Furthermore, we would like to place blocks in the FPGA so that the results of one hardware block can be passed as arguments to another hardware block without requiring intervention from the CPU.

Acknowledgments

We would like to thank the reviewers for their insightful comments on earlier drafts of this paper.

This research was supported by a grant from the Powell Foundation.

References

[1] T. L. Adam, M. K. Chandy, and J. R. Dickson, “A comparison of list schedules for parallel processing systems,” *Communications of the ACM*, vol. 17, no. (12), pp. 685–690, December 1974.

[2] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers-- Principles, Techniques, and tools*, 1987, Reading, Massachusetts: Addison-Wesley Publishing Company. 1987.

[3] J. Babb, R. Tessier, and A. Agarwal, “Virtual Wires: Overcoming Pin Limitations in FPGA-based Logic Emulators,” in *IEEE Workshop on FPGA-based Custom Computing Machines*, 1993.

[4] Z. Barzilai, J. L. Carter, B. K. Rosen, and J. D. Rutledge, “HSS– A high speed simulator,” *IEEE Transactions on Computer-Aided Design*, vol. CAD-6, no. (6), pp. 601–617, July 1987.

[5] Cadence, “Verilog-XL, Version 1.6,” March 1991.

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, 1990, MIT Press and McGraw-Hill. 1990.

[7] R. Ernst, J. Henkel, and T. Benner, “Hardware-Software Cosynthesis for MIcrocontrollers,” *IEEE Design & Test of Computers*, vol. 10, no. (4), pp. 64–75, December 1993.

[8] J. A. Fisher, “The optimization of horizontal microcode within and beyond basic blocks: An application of processor scheduling with resources,” Ph. D. Thesis, New York University, New York, 1979.

[9] R. K. Gupta and G. D. Micheli, “Hardware-Software Cosynthesis for Digital Systems,” *IEEE Design & Test of Computers*, vol. 10, no. (3), pp. 29–41, September 1993.

[10] C. Hansen, “Hardware logic simulation by compilation,” in *25th ACM/IEEE Design Automation Conference*, 712-715, 1988.

[11] G. Kane and J. Heinrich, *MIPS RISC Architecture*, 1992, Englewood Cliffs, New Jersey: Prentice Hall. 1992.

[12] D. M. Lewis, “A hierarchical compiled code even-driven logic simulator,” *IEEE Transactions on Computer-Aided Design*, vol. 10, no. (6), pp. 726–737, June 1991.

[13] M. D. Smith, “Tracing with Pixie,” Technical CSL-TR-91-497, Stanford University, Computer Systems Laboratory, Nov. 1991.

[14] Synopsys, “Design Compiler Reference Manual, Version 3.0,” Synopys, December 1992.

[15] S. M. Trimberger, ed., “Field-Programmable GAte Array Technology,” 1994, Kluwer Academic Publishers: Boston.