Performance Modeling and Tradeoff Analysis During Rapid Prototyping

Jeffrey Walrath, Karam S. Chatha, Ranga Vemuri, Naren Narasimhan and Vinoo Srinivasan
Laboratory for Digital Design Environments
Department of ECE & CS, University of Cincinnati
Cincinnati, OH 45221-0030
Email: ranga.vemuri@uc.edu

Abstract

Tradeoff analysis is a central aspect of any design process. Languages and tools to support performance modeling and tradeoff analysis are necessary to facilitate rapid prototyping of designs. An effective modeling and evaluation environment reduces the overall design time of both the prototype and the final product by helping designers in determining which parameters of a design are critical for meeting a set of desired performance goals. This paper describes a case study in performance modeling using a language called PDL (Performance Modeling Language). The PDL system supports tradeoff analysis and performance visualization. This paper also addresses some of the key issues for successful tradeoff analysis during rapid prototyping and explains how many features of PDL make it a suitable choice for this purpose.

1: Introduction

During any design process, many decisions are made which affect the overall performance of a design. Many such decisions result from detailed tradeoff analysis among several related attributes of the design. For example, choice of the input clock frequency depends partly upon the desired upper bound on power consumption and the desired lower bound on the throughput rate. Decisions such as these are made at various levels of the design process from specification to implementation.

In order to make effective design choices, the design environment and supporting tools must be well suited for performing tradeoff analysis throughout the design process. We believe the design environment should have the following features: (1) Modeling at any level of abstraction must be supported since tradeoff analysis occurs at various levels during the design process; (2) The modeling environment must lend itself to reusability. Reuse of models is very critical because it reduces the time spent in writing a model for each new version of the design. (3) The performance evaluation engine should be flexible enough to partially evaluate a model when variations in some parameters are unknown. This facilitates incremental analysis of the model; (4) The modeling environment must be easy
to use so that the development and analysis of performance models can be done quickly and efficiently.

We have developed a performance modeling and tradeoff analysis environment, the PDL System, which meets all the criteria and is well suited for use during rapid prototyping [6]. A PDL program declares design objects (various kinds of modules, nets, and ports) that can possibly appear in a design. It also describes the containment and connectivity relationships among the objects that were declared. In addition, various types of attributes and evaluation rules can be declared which are attached to the design objects. An evaluation rule is a mathematical expression that describes how to calculate a particular performance parameter of a design.

Figure 1 illustrates the PDL System and design process for generating and evaluating performance models. An executable performance model is created when a PDL program is compiled with a specific design net-list. The evaluator can be configured to evaluate and collect data from the model in several different ways. In the simplest case, a model can be completely evaluated with a complete set of input data. A configuration can be specified that allows for the collection of data for graphical analysis or tabulation. Incremental evaluation is also possible with a performance model (the feedback loop in the figure). During evaluation, only some of the input data is supplied with the result being another performance model. Further evaluation on this model can be done with more input data specified as necessary.

2: Hardware/Software Co-design with Coprocessors

Rapid prototyping for hardware-software co-design of embedded processor and coprocessor systems is a current research area. A design is typically broken down in to several smaller, independent tasks or jobs. The specification for a co-design is represented as a task graph where nodes represent the tasks and edges represent communication channels between tasks [4]. For hardware-software co-design, the goal is to determine which tasks should be implemented in hardware or software based upon some performance criterion.

The number of tasks that can be implemented in hardware is determined by the target architecture. In an embedded system, several hardware tasks can be implemented as ASICs, and all of the software tasks are allocated to execute on a single embedded processor. However, in a coprocessor system, only a single task is allocated to hardware, and all other tasks are allocated to software running on the same single processor. A coprocessor system is a configurable plug-in board connected to a main processor such as a workstation or a personal computer. Components of the coprocessor board include a programmable chip, interface memory, and a predefined interface protocol to the main computer [7].

There are several factors which affect the overall performance of a co-design that must be considered during the design process. From a software perspective, tasks have certain
properties that govern their execution sequence. If all tasks are bound to execute in software, then only one task can execute at a time. The next scheduled task can not begin execution until all preceding tasks are finished. Figure 2 is an example of a task graph. Although there are tasks which appear to be independent of each other, an execution order is associated with this task graph since one task can execute at a time. For this example, task 6 can not begin executing until tasks 3, 4, and 5 finish. This execution order is referred to as the task schedule for a particular task graph.

Another feature of hardware software co-designs is the inherent parallelism available between the hardware and software. This is achieved by having one hardware task executing in the coprocessor simultaneous with a software task executing in the main processor. Figure 3 shows a simple task graph where hardware-software parallelism can be exploited. When none of the tasks are bound to hardware, the only task schedule is task 1 followed by task 2 and so forth. If task 2 were bound to hardware, then the task schedule could be pipelined so that task 2 and 3 operate simultaneously. Pipelining occurs with a data buffer between tasks 2 and 3. After task 2 finishes executing the first time, the data moves to the buffer at the input of task 3. Then, the next time task 2 executes, task 3 also executes. The buffer is necessary to ensure task 2 does not write to the same memory being read by task 3 during execution.

When the target architecture is a coprocessor board, another performance parameter is the communication time between tasks when one task is in software and the other is in hardware. The computer can not transmit data directly to the coprocessor. Instead, data is transmitted through memory located on the coprocessor board. Because this memory is located on a board which is connected to a slower bus interface, communication time necessary to read and write data from the coprocessor to the computer's main memory is slower than usual memory transfers within the computer. This will have a noticeable impact on estimating execution time of a particular set of task bindings. In most co-design problems, the communication between the hardware is such that one task writes to the coprocessor memory prior to execution of the hardware task. Once the hardware task...
finishes, the next software task reads the results from the coprocessor memory.

The expression for calculating the total execution time of a task graph is based on a sum of the execution times for each task. However, with the hardware parallelism that can occur due to pipelining, it is not a simple summation of each task’s execution time. The following equations detail how the total execution time for a graph is determined when pipelining is used.

Calculating the execution time for an individual task is given by the equation:

\[
\text{ExecutionTime} = \text{BindingTime} + \sum \text{RdOverHd} + \sum \text{WrOverHd} \tag{1}
\]

\[
\text{RdOverHd} = \text{NumVariables} \times \text{ReadTime} \tag{2}
\]

\[
\text{WrOverHd} = \text{NumVariables} \times \text{WriteTime} \tag{3}
\]

In this equation, BindingTime is the execution time for a particular task depending upon a hardware or software binding. As previously mentioned, tasks which must read or write to the coprocessor memory have an associated communication time related to transferring the data. Recall that a single task can have several edges which are input to the task. For each task on the input which transmits data via coprocessor memory, RdOverHd will be a non-zero value. If coprocessor memory is not involved, then RdOverHd will be zero for that particular input edge. Thus, the ExecutionTime includes adding all the time necessary for reading data from the coprocessor memory. A similar addition is used for writing to coprocessor memory for all the outputs and is accounted for by WrOverHd.

Calculating total execution time is given by the equation:

\[
\text{GlobalTime} = \sum \max(\text{task}_1,'\text{ExecutionTime}',\text{task}_2,'\text{ExecutionTime}',...) \tag{4}
\]

GlobalTime is a sum of the execution times for each task. A particular task is scheduled and the execution time is ExecutionTime. Another task is scheduled and its execution time is added to the previous time. This process continues until all tasks have been scheduled with GlobalTime accumulating the execution times for each task. Because pipelining allows more than one task to execute at a time, the total global time only increases by the maximum ExecutionTime of all tasks which are scheduled to execute simultaneously.

3: Performance Model for Co-designs

In this section, we develop a suitable performance model in PDL for co-design performance estimation. PDL has three basic object types for representing designs: modules, carriers, and ports. A module can be used to represent any type of component typically found in a design. A carrier is commonly used for representing transport components such as connections, wires, buses and communication channels. Ports are objects used primarily for representing the connectivity among various design components. [5]

The first step in developing a PDL program is to declare the different types of objects that may appear in a design net-list. Figure 4 shows the object declarations for the co-design example. The module codesign is declared as the object which represents the entire task graph. The two ports task.in.port and task.out.port in conjunction with the carrier edge represent a directed edge in the graph, and the module task represents a single task.

An important feature in PDL is that various objects can contain references to other objects; this is known as containment. Containment serves two useful purposes. First,
port task.out_port
end port;

module codesign
   ports
      inputs{} : task.out_port;
      outputs{} : task.in_port;
   carriers
      connections{} : edge;
   modules
      tasks{} : task;
   end module;

Figure 4. PDL Object Declarations for a Co-design

type
   hw_sw_bind : enum {hw,sw};
end type;

port task.aut.port
   attributes
      primitive num_var : int;
      t1.bind : hw_sw_bind;
      t2.bind : hw_sw_bind;
      wr_overhd : real;
      rd_overhd : real;
      dynamic done, t2.job : int := 0;
   rules
      wr_overhd =
         wr_comm(t1.bind, t2.bind, num_var);
      rd_overhd =
         rd_comm(t1.bind, t2.bind, num_var);
   end port;

port task.in.port
   attributes
      t2.bind : hw_sw_bind;
      rd_overhd : real;
      dynamic done, t2.job : int := 0;
   end port;

carrier edge
   ports
      input : task.out_port;
      output : task.in_port;
   rules
      input't2.bind = output't2.bind;
      input't2.job = output't2.job;
      output'rd_overhd = input'rd_overhd;
      output'done = input'done;
   end carrier;

Figure 5. Port and Carrier Declarations with Attributes

it allows the parent object access to information within any contained object. Secondly, it allows for declaring the connectivity of two objects. If two different objects, perhaps two modules, contain a reference to the same port, then the two objects are considered connected to each other through that port.

Once all the objects are defined, the next step in developing a PDL program is to declare attributes and attribute evaluation rules in the different objects. Attributes are parameters that are propagated and computed in the PDL model. An evaluation rule describes how to perform the calculation of an attribute in an object. Figure 5 shows the declarations of the port and carrier objects with all their attributes and evaluation rules. Figure 6 show the completed declarations for the task and codesign module objects.

Attributes are defined in the attributes section of an object, and can be one of several different data types. Some of the types available are integer, real, enumerated type, heterogeneous records, lists, and a variety of combinations of these. An attribute is associated with the object where it was declared and not with the object where the attribute is given a value or referenced. Thus, when an attribute is used in an expression where it is not
defined within the object, it is referenced as object 'attribute. For example, in the edge
carrier there is a reference to input 't2.bind. The port input is declared as a contained
port and within the port there is an attribute declaration for t2.bind.

Along with defining the type, the attributes section is used to define an attribute as
primitive or non-primitive. An attribute is non-primitive unless explicitly declared as a
primitive. A primitive attribute is an attribute which will not have an evaluation rule for
defining how to calculate it. Instead, a primitive attribute will have its value set by the
user during the execution of the performance model.

In addition to being primitive, an attribute can also be dynamic or static. An attribute is
considered static unless declared dynamic. During model execution, all static attributes are
calculated once. These are attributes which are not based upon some dynamic stream of
events, but instead are values which need to be calculated once since they are independent
dynamic events. Conversely, dynamic attributes are not single values but streams of
single values. Every dynamic attribute is re-evaluated several times, each time with a new
value from the stream of values. For example, if there are 5 evaluation cycles, then every
dynamic attribute have a stream of 5 distinct values. This is similar to simulating the
performance model based on a stream of events.

Recall the equations for calculating the execution time of a task graph described in
Section 2. All of those equations are used in the co-design performance model. However,
the model is slightly more complicated because global time also depends upon the task
schedule. The remaining attributes in the model are used to determine when a task is
executing based upon the task schedule. Because of the dynamic evaluation of the model,
during each evaluation cycle only one or two tasks can be executing. When a task is not
executing, the execution time is calculated to be zero. Thus, when global time is calculated,
only the maximum time of all executing tasks is used during each evaluation cycle.

4: Tradeoff Analysis Using the PDL System

Once a PDL model is written, the next step is to compile it. As mentioned previously,
a PDL model by itself is not an executable model. It is a template which describes how to
create a performance model for a class of designs. An executable model is only generated
when the PDL program is coupled with a specific design. The PDL compiler takes a PDL
model and a design (a specific task graph in the case of our example) as input and generates
an executable performance model. Figure 7 is a detailed overview of the PDL system and
the flow of a PDL program and net-list through an analysis cycle.

Once a performance model has been generated, the user executes the model with the PDL
evaluator. Model evaluation can be done in several ways depending upon the configuration
and input to the evaluator tool. A performance model can be completely evaluated when all
the primitive attribute values are supplied; this is known as full evaluation. Another option
is to evaluate the performance model with only some of the primitive data specified. This is
known as partial evaluation. In addition, the evaluator can be configured to collect data for
analysis based on ranges of values for primitive attributes instead of single values. Finally,
the evaluator can be linked into an existing CAD/CAE tool to perform data analysis.

Full evaluation of a model begins with an executable performance model. All primitive
attributes that were defined in the PDL program must be defined by the user in an input
data file. When the evaluator is invoked, both the performance model and data file are
read, and all expressions are evaluated with the results written to another performance
module codesign
ports
inputs{} : task.out_port;
outputs{} : task.in_port;
carriers
connections{} : edge;
modules
tasks{} : task;
attributes
primitive num_jobs : int;
primitive dynamic sched_task : int := 0;
dynamic global_time : real := 0.0;
rules
tasks{} num_jobs = num_jobs;
tasks{} sched_task = sched_task;
global_time =
max(foreach t in tasks{t.time});
tasks{} global_time = curr global_time;
end module;

module task
ports
inputs{} : task.in_port;
outputs{} : task.out_port;
attributes
primitive binding : hw_sw_bind;
primitive hw_time : real;
primitive sw_time : real;
primitive dynamic sched_task : int;
dynamic time : real := 0.0;
dynamic job : int := 0;
dynamic job_diff : int := 0;
dynamic done_in : int := 0;
dynamic done_out : int := 0;
dynamic exec : int := 0;
dynamic global_time : real := 0.0;
num_jobs : int;
rd_overhd : real;
wr_overhd : real;
comm_overhd : real;
exectime : real;
rules
inputs{} t2.bind = binding;
inputs{} t2.job = curr job;
outputs{} t1.bind = binding;
exectime = if (binding == hw)
then hw_time
else sw_time
endif;
done.in =
eval(foreach p in inputs{ curr p'done });
job.diff = curr job -
min(foreach p in outputs{ p't2.job });
rd_overhd =
sum(foreach p in inputs{ p'rd_overhd });
wr_overhd =
sum(foreach p in outputs{ p'wr_overhd });
exec =
begin
temp := int;
if ((done.in == 1) and (curr job < num_jobs)
and (job.diff < 1))
then if (binding == hw)
then temp := 1;
else if (sched_no == sched_task)
then temp := 1;
else temp := 0;
endif;
endif;
else temp := 0;
endif;
return temp;
end;
comm_overhd = if (binding == hw)
then 0.0
else wr_overhd + rd_overhd
endif;
time := if (exec == 1)
then global_time + exec_time + comm_overhd
else time
endif;
job = if (exec == 1)
then job + 1
else job
endif;
done.out = if (exec == 1)
then 1
else done.out
endif;
outputs{} done = done.out;
end module;

Figure 6. Codesign and Task Module Declaration
model. Since the model was fully evaluated, all evaluation rules will have been replaced with their corresponding evaluation result. Thus, the resulting performance model will contain nothing but attributes and their evaluated values.

In addition to full evaluation, the user can partially or incrementally evaluate a model. Instead of specifying a complete set of values for all primitive attributes, the user can specify only some of the data for the primitive attributes. When the evaluator is invoked, the performance model and data file are read, all evaluation rules are partially evaluated and a residual performance model is generated. The residual model model will still contain (partially evaluated) evaluation rules for various attributes which have been reduced and simplified with respect to the original expression. The residual model can be further evaluated when more primitive attribute data is available.

In addition to evaluating a model with single data points, the evaluator can be configured to collect data for ranges of primitive values. For example, instead of setting a primitive attribute to one particular value, the user can specify that a primitive attribute can be a range of values. Then during execution, the model is evaluated with the specified attribute set to each value in the range. Any number of primitive attributes can be setup to have ranges of values. In addition, the evaluator can be configured to collect data on any attribute attribute within the model, primitive or non-primitive. Two types of data collection is possible: Data can be collected in a format suitable for two or three dimensional plots, or the evaluator can collect data for any number of attributes and store the results in tabular form.

5: Results of Tradeoff Analysis for a Co-design Example

The co-design model written in PDL and discussed in Section 3 is flexible enough to perform performance modeling for many different types of task graphs. In this section, we discuss results of using this model for a specific co-design example involving a JPEG-like compression/decompression scheme [8, 11]. The target architecture was a coprocessor system using an FPGA connected to a personal computer [3]. Tradeoff analysis was performed using the PDL model to determine which task to implement in hardware. Figure 8 shows the task graph for the compression part of the JPEG algorithm in terms of objects from the PDL program. Arrows in the figure represent the connectivity among the objects.

Before the performance model could be executed, it was necessary to obtain hardware and software execution times for each task. Software execution times were obtained using

---

Figure 7. PDL System Overview
Figure 8. Task Graph for JPEG

Table 1. Measured and Estimated Execution Times

<table>
<thead>
<tr>
<th>Task</th>
<th>Hardware</th>
<th>Software</th>
<th>PDL Estimated Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCT</td>
<td>8.4 µs</td>
<td>311.3 µs</td>
<td>0.23 s</td>
</tr>
<tr>
<td>Quantization</td>
<td>0.6 µs</td>
<td>7.56 µs</td>
<td>1.51 s</td>
</tr>
<tr>
<td>ZigZag</td>
<td>0.4 µs</td>
<td>1.63 µs</td>
<td>1.54 s</td>
</tr>
<tr>
<td>RLE and Huffman Encoding</td>
<td>884 µs</td>
<td>18.48 µs</td>
<td>4.07 s</td>
</tr>
</tbody>
</table>

existing software profiling tools to time each of the tasks in the software version of the JPEG algorithm. In this case, all software times were collected with timing functions on a Pentium system containing a P100 microprocessor, 256 kilobytes of standard cache, and 16 megabytes of main memory. Hardware execution times were estimated using a synthesis tool [2] that generated a register transfer level design for each task. Results of the synthesis tool included estimated execution times for each RTL design. Times were estimated for a 2 micron CMOS technology. Table 1 shows estimated hardware and measured software execution times for the various JPEG tasks. These times are for each task performing its respective job on 16 pixels at a time.

The next step in the analysis process was to use the PDL model to calculate the overall execution time for the task graph with each task bound to hardware. In order to do this, other input information such as a task schedule, the number of variables being transferred, and the size of the picture file were necessary. The final column in Table 1 shows the estimated execution times for an input picture file containing 4080 pixels with a task schedule that included pipelining.

Results of Table 1 show that the DCT (Discrete Cosine Transform) task was the best choice for implementation on the coprocessor hardware. The entire codesign was implemented with the DCT task in hardware on a coprocessor system [7] connected to a Pentium based PC [3]. Once complete, actual execution times for compressing images of different sizes were measured. Accordingly, the PDL performance model was evaluated with primitive attributes set for each of the different input images. Table 2 shows the results of the PDL estimations compared with the coprocessor execution times.

6: Conclusion

We have presented a performance modeling and analysis approach for co-designs using the PDL system. In PDL, it is straight-forward to make several enhancements to the codesign model presented in this paper so that it requires less primitive input information or considers more performance parameters than just execution time. For example, the
model could determine a task schedule based on the hardware software bindings, more
detail could be included as to the target architecture, estimation could be incorporated
for cost, hardware area, and so forth. As the design evolves and requires more detailed
performance analysis, so too can the PDL model evolve and be refined to a more accurate
representation of the system being modeled.

It is important to note that the PDL model is a generic model from which specific,
executable performance models can be generated (using the PDL compiler) given a spe-
cific task graph. Thus, the PDL model applies to any task graph which follows the
object construction scheme specified in the PDL program. This is the essential differ-
ence between performance modeling in PDL versus a procedural hardware description
language such as VHDL. More information on the PDL language and system, includ-
ing the system software, can be obtained through the PDL home page on the WWW
at http://www.ece.uc.edu/~ddel/pdl.html.

References

of Reconfigurable Coprocessors". In Proc. of the 1996 International Conference on