Performance Evaluation Tool for Rapid Prototyping of Hardware-Software Codesigns *

Karam S. Chatha and Ranga Vemuri,
Department of ECECS, University of Cincinnati, Cincinnati, Ohio 45221-0030.
Email:{kchatha,ranga}@ececs.uc.edu

Abstract

Performance evaluation is essential for tradeoff analysis during rapid prototyping. Existing performance evaluation strategies based on co-simulation and static analysis are either too slow or error prone. We therefore present an intermediate approach based on profiling and scheduling for rapid prototyping of hardware-software codesigns. Our performance evaluation tool obtains representative task timings by profiling which is done simultaneously with system specification. During design space exploration the tool obtains performance estimates by using well known scheduling and novel retiming heuristics. It is capable of obtaining both non-pipelined and pipelined schedules. The tool includes an area estimator which calculates the amount of hardware area required by the design by taking resource sharing between different hardware tasks in to account. The tool also allows the user to evaluate the performance of a particular schedule with different task timings. In contrast to co-simulation and static analysis, the tool is able to provide fast and accurate performance estimates. The effectiveness of the tool in a rapid-prototyping environment is demonstrated by a case study.

1. Introduction

Performance evaluation is the corner stone for effective design space exploration. The performance evaluation tool should satisfy two important requirements. It should be fast so that the design cycle time is short; and it should give fairly accurate estimates to be a useful tool. In this paper we discuss a performance evaluation tool which satisfies these two important requirements.

The hardware-software codesign flow for rapid prototyping is shown in Figure 1. The inputs are denoted in dashed boxes and the design steps are enclosed in solid boxes. The codesign architecture contains a single general purpose microprocessor, a single hardware (HW) processor and a block of shared memory. The shared memory is exclusive read and exclusive write. It is used for communication between tasks mapped to the software (SW) processor and the HW processor and also between two tasks mapped to the HW processor. We assume that no two HW tasks can execute in parallel on the HW processor. The advantage is that such an implementation strategy allows us to bind more tasks to HW. The trade-off is that we reduce the amount of concurrency that can be implemented in HW.

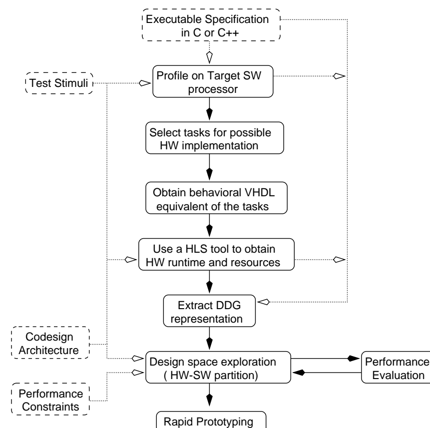

Figure 1. Rapid Prototyping Design Flow

The input specification is profiled by executing it with typical input stimuli and the runtimes of the various tasks is obtained on the target SW processor. Runtimes of certain tasks might constitute a large percentage of the overall runtime of the design. Such tasks might be suitable for HW implementation. The behavioral VHDL equivalent of these tasks is passed through a high level synthesis (HLS) system. The HLS system helps in obtaining an estimate about the runtime of the tasks in HW and the amount resources required by the tasks in HW. The specification is then captured in a data dependency graph (DDG) representation and the nodes of the graph are partitioned to either SW or HW. After a partition is obtained it is essential to evaluate its performance. This task is done by the performance evaluation tool. The tool gives the partitioner feedback in terms of the runtime, shared memory re-

*This work was partially supported by the ARPA RASSP program and monitored by the Wright Lab, US-AF under contract number F33615-93-C-1316.

quirements and amount of HW resources required by the design. Once a constraint satisfying design is obtained by the partitioner the design is implemented by rapid-prototyping.

In the design flow mentioned above performance evaluation is done as a two step process. In the first step, the runtimes of the tasks in SW and HW are obtained. After the input specification is written, the designer verifies its functionality by running it on test stimuli. The SW runtimes of the tasks can be obtained at this stage itself, thereby saving design time. The runtimes and the resources required by the tasks in HW are obtained by using a HLS tool and this does not take a long time. During the second step of performance evaluation the performance estimates are obtained by scheduling the tasks on the codesign architecture. Scheduling also does not take much time and hence our performance evaluation strategy gives shorter design cycle time as compared to co-simulation [2][11][14][16] or prototyping. Performance evaluation techniques based on static analysis [6] of the code are faster but they have high error margins (over 30%) [8]. This is because static analysis techniques do not fully capture the memory caching and pipelining effects of the SW processor. Since we obtain SW runtime estimates by actual execution of the code our strategy has lower error margins.

The tool uses a profiling and scheduling to obtain runtime and shared memory estimates of the design alternatives. Other researchers consider scheduling integrated with HW-SW partitioning [3][4]. Profiling has been used as an aid in design space exploration in [7]. We obtain pipelined designs by iterative retiming transformation [5] and scheduling. Our retiming heuristic RECOD [5] differs from other existing techniques [9][13][17] since it does retiming in two steps: in the first step it tries it tries to minimize the runtime of the pipelined design and in the second step it tries to minimize the shared memory requirement of the HW-SW codesign. Pipelined scheduling techniques based on loop unfolding [12] have the disadvantage of large memory requirements proportional to the unfolding factor of the loop. List scheduling based techniques [1] do scheduling and assigning tasks to different pipe stages at the same time. This results in limited design space exploration.

The paper is organized as follows: in Section 2 we discuss our graph representation, Section 3 presents the performance evaluation tool, the case study is in Section 4 and finally Section 5 concludes the paper.

2. Graph Representation

The internal graph representation used by our system is a *data dependency graph (DDG)* format. The graph is specified by a four tuple $G(V, E, \lambda, \delta)$ [13] where V is the set of tasks, E is the set of directed edges representing the data dependencies between tasks , λ is a mapping $\lambda : V \rightarrow \mathbb{N}$ which denotes the *iteration index* of a task and δ is a mapping $\delta : E \rightarrow \mathbb{N}$ which denotes the *dependence distance* of a data dependency ($\mathbb{N}$ is the set of integers). Each vertex contains information about binding of the task (to HW or SW), the task's runtime on the SW processor, its runtime on the HW processor and a list of the number and type of HW resources required by the task. Each edge contains information about the number of variables transfered across the corresponding data dependence. Iteration index (λ) and dependence distance (δ) are required to implement a pipelined schedule of the graph. In order to explain these terms we consider the DDG shown on the left hand side in Figure 2. We assume that the DDG is executed a number of times inside a loop. The pipelined schedule of the design is shown below the DDG. The big rectangles correspond to the various tasks and the small rectangles indicate shared memory read and write. The pipelined schedule consists of the *prologue, steady state* and *epilogue*. In any iteration i of the steady state instance of task A belonging to the $(i+1)$ iteration of the original loop is executed. Hence we say that the iteration index of A, $\lambda(A) = 1$. Similarly $\lambda(B) = 0$, $\lambda(C) = 1$ and $\lambda(D) = 0$. Now for the data dependency $e = AB$, data produced by task A in iteration i of the steady state is consumed by task B in the iteration $i+1$ of the steady state. Therefore the dependence distance of edge $e = AB$, $\delta(AB) = 1$. Similarly $\delta(AC) = 0$, $\delta(BD) = 0$ and $\delta(CD) = 1$. The DDG corresponding to the steady state is shown on the right hand side of the figure.

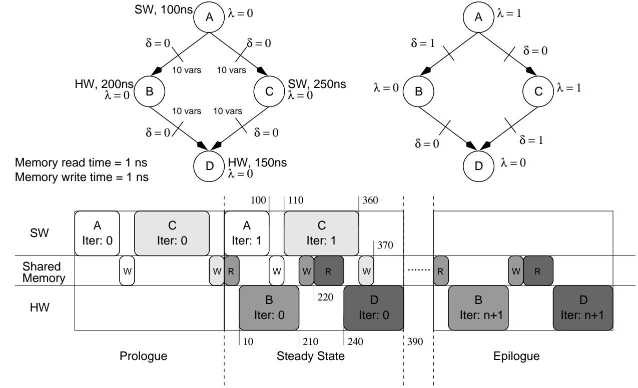

Figure 2. DDG and Pipelined Schedule

3. Performance Evaluation Tool

The performance evaluation tool with its various subblocks is shown in Figure 3. The inputs of the tool are the partitioned DDG, the codesign architecture and

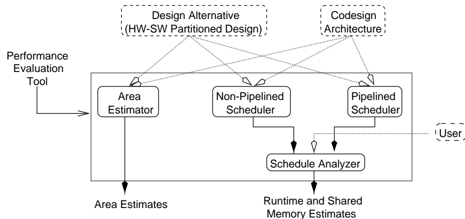

Figure 3. Performance Evaluation Tool

user specified data. A brief overview of the tool flow is as follows. Given a partitioned DDG the area estimator calculates the HW resources required by the design. Depending on the user the tool generates a non-pipelined or pipelined schedule of the design. The schedule tries to minimize the runtime of the design in the presence of resource conflicts and communication delays. After a schedule has been obtained the tool can estimate the runtime and shared memory requirements of the design. A particular schedule is based on representative task timings obtained during profiling. The user can examine the performance of a particular non-pipelined or pipelined schedule under different task timings by using the schedule analyzer. In the following sections we discuss each of these sub-blocks in detail.

3.1. Area Estimator

The area estimator calculates the total area occupied by the tasks which are bound to the HW processor. Since we assume that only one task can execute on the HW processor at a given time, the HW tasks can share resources. The resources of the task are classified into two types: those that can be shared and those that cannot be shared. The resources which can be shared are functional units (for example adders, subtracters, ALU) and registers. The resources which cannot be shared are the controller and interconnect. The area estimator takes the union of the shared resources associated with all the tasks and then calculates the area due to them. The area associated with resources which are not shared is estimated by addition. The sum of the areas due shared and non-shared resources is the total area occupied by the tasks bound to the HW processor.

3.2. Non-pipelined Scheduler

A non-pipelined schedule of a $DDG = G(V, E, \lambda, \delta)$ is an assignment of start times S to the tasks $S : V \rightarrow IN$ such that all the data dependencies are honored, that is $\forall (u, v) \in E \ such \ that \ \delta(u, v) = 0, S(v) \geq S(u) + u_{exec}, u, v \in V$. u_{exec} is the total execution time of the task. It is the sum of the task read time, runtime of the task on the processor that it has been bound to and task write time.

We use a *list based scheduler* to estimate the performance of a non-pipelined design. The scheduler takes resource conflicts due the HW processor, SW processor and the shared memory into account. It also takes communication delays into account. The list based scheduler maintains *three ready lists*: a *HW ready list*, a *SW ready list* and a *memory ready list*. The list based scheduler also maintains a variable called *time* to keep track of passage of time as various tasks are scheduled. A task is added to the HW or SW ready list when all its predecessor tasks have been scheduled and completed their execution. A task is selected to be scheduled from the HW or SW ready list when the corresponding resource is free. The task is then considered to be in the read state and it is added to the memory ready list. After it has been selected from the memory ready list the task does its read operation and runs on the assigned resource. After the task has completed its run state, it goes in to its write state and is again added to the memory ready list. A task finally finishes its execution when it is selected from the memory ready list and scheduled to do its write operation. After all the tasks have been scheduled the tool can estimate the total runtime of the design.

Estimation of Shared Memory Requirement: The shared memory required to implement the design is the maximum amount of memory occupied during one complete execution of the design. We assume that a task during its execution needs shared memory space for both its read and write variables. When a task completes execution it frees the shared memory occupied by its read variables.

Heuristic Select Function: The scheduler uses the same heuristic priority function to select a task from the three ready lists. The priority of a task to be selected depends on the following three properties in descending order : *mobility, variable difference* and *number of successors*. A task with lower mobility is selected to be scheduled before a task with higher mobility. The variable difference for a task u is given by $vardiff(u) = rdvar(u) - wrvar(u)$, where $rdvar(u)$ ($wrvar(u)$) is the number of variables read (written) by the task u from (to) the shared memory. The memory requirement of a schedule is given by the maximum memory occupied by the data items during one iteration of the steady state. A task which reads more variables than it writes is likely to reduce the number of variables present in the memory. Hence it should be scheduled near its ASAP time. Alternatively a task which writes more variables than it reads should be scheduled near its ALAP time. Hence a task with lesser variable dif-

ference is selected to be scheduled before a task with greater variable difference. A list scheduling algorithm performs better when it has more choice in the ready list [13]. Hence a task whose completion adds more tasks to the ready list is selected.

3.3. Pipelined Scheduler

The pipelined schedule refers to the schedule of the steady state of the pipeline. The pipelined schedule of a DDG is characterized by its *Initiation Interval (II)* which is the time difference between the start of two consecutive iterations of the steady state. The pipelined schedule with an initiation interval II is an assignment of start times to tasks, $S(v)$, such that for all tasks v in the graph $0 \leq S(v) \leq II$ [13]. For a dependency $e = (u, v)$, the schedule time of u and v must honor the data dependence, that is $S(v) + \delta(e) \times II \geq S(u) + u_{exec} \Rightarrow S(v) \geq S(u) + u_{exec} - \delta(e) \times II$. We obtain a pipelined schedule by scheduling and retiming in an iterative manner as shown in Figure 4. Given a DDG there exists a theoretical lower bound on the initiation interval of it pipelined schedule called the *Minimum Initiation Interval (MII)* [13]. The MII is determined by two factors: the number of resources in the codesign architecture and recurrences (or cycles) in the graph. We calculate the MII, and try scheduling the DDG in MII time. We use the list based scheduler described in Section 3.2. Due to the schedule constraining dependencies and communication conflicts we may not be able to schedule the DDG in MII. If we can't, we try selecting a dependency to be retimed. In the case that we do find a dependency, we retime it and try scheduling the DDG again. *The objective of retiming is to reduce the number of schedule constraining dependencies.* If we cannot find a dependency to be retimed we increase the objective initiation interval and try scheduling again. The increment factor is set to maximum of one percent of MII or one time unit. After a successful schedule of the steady state has been obtained we generate the corresponding prologue and epilogue graphs by using loop unrolling transformation [13]. We then schedule the prologue and epilogue graphs to estimate the performance of the complete pipelined implementation.

Schedule constraining dependencies: Depending upon its dependence distance a dependency may or may not constrain a pipelined schedule. A dependency (u, v) with $\delta(u, v) = 0$ implies that the data produced by the predecessor task u is consumed by the successor task v in the same iteration of the steady state and hence it constrains the schedule. Such a dependency is called a *intra loop dependency (ILD)*. We assume

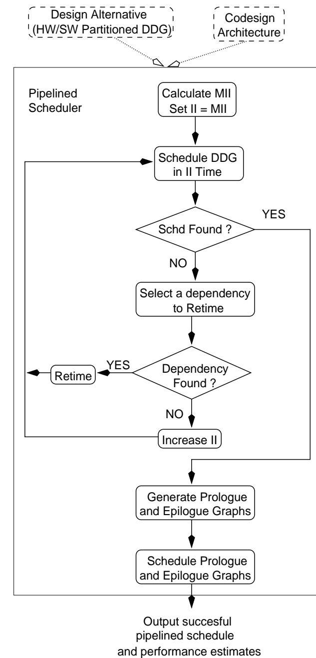

Figure 4. Pipelined Scheduling Design Flow

that the all the tasks belonging to one iteration of the steady state are executed before any task belonging to the next iteration. Also the HW and SW resources are themselves non-pipelined with respect to task execution. Then a dependency (u, v) with $\delta(u, v) > 0$ does not constrain the pipelined schedule. Such a dependency is called a *loop carried dependency (LCD)*. A LCD represents a data dependence between tasks belonging to different iterations of the steady state.

Shared memory requirement of the Steady State: $LCDs$ represent data dependencies between tasks belonging to different iterations of the steady state. Hence before an iteration of the steady state can begin there is already some memory occupied by the LCD data which is given by $Mem_{LCD} = \sum_{e \in LCD} \delta(e) \times var(e)$, where $var(e)$ is the number of variables transferred across the data dependence e. The memory required during one iteration of the steady state is the maximum amount of memory occupied by the data items during execution, Mem_{exec}. This memory is both due to $ILDs$ and $LCDs$. The memory requirement of the steady state, $MemReq$ is then given by: $MemReq = \max(Mem_{LCD}, Mem_{exec})$.

RECOD, A retiming heuristic: We do retiming when we are unable to schedule a DDG in the desired initiation interval. Retiming changes the iteration indices of nodes and dependence distances of edges to

produce an equivalent DDG with tasks belonging to different iterations of the original loop. In other words retiming helps in producing a pipelined DDG. The objective of retiming is to decrease the number of schedule constraining dependencies with the least increase in shared memory requirements. The performance evaluation tool uses RECOD [5] which optimizes the resource and memory utilization of HW/SW codesigns. A brief explanation of the heuristics is as follows. RECOD does retiming in two steps.

RECOD Step 1: In the first step RECOD selects a dependency which on retiming gives maximum freedom to the scheduler. Such a dependency is an ILD between tasks bound to heterogeneous resources.

RECOD Step 2: We retime a dependency by increasing its dependence distance thereby converting it into a LCD. However in order to apply a legal retiming transformation which results in an equivalent graph there are other dependencies that may also need to be retimed. In step 2, RECOD selects a set of dependencies which on retiming give the least increase in shared memory requirements. Given a dependency (u, v) to be retimed we can define the following three sets with respect to u:

$V_c = \{\textit{connected component to which u belongs}\}$

$P = \{w \in V_c | \textit{there is a path from w to u}\} \cup \{u\}$

$S = \{w \in V_c | \textit{there is a path from u to w}\}$

$R = V_c - \{P \cup S\}$

In the second step RECOD partitions the set R into sets P and S such that the number of variables transfered across the cut are minimized. RECOD retimes the DDG by using the following two equations:

$\forall u \in P, \lambda(u) = \lambda(u) + 1$

$\forall (u, v) \in E, u \in P, v \notin P, \delta(u, v) = \delta(u, v) + 1$

Generation of Prologue and Epilogue Graphs: Consider the retimed data dependency graph, $DDG_{retimed}$ shown in Figure 5. The retimed graph represents the steady state of the pipelined schedule. At the ith iteration of the steady state, a task u represents the execution of the same task belonging to iteration $i + \lambda(u)$ of the original un-retimed DDG. For example at the first (0th) iteration of the steady state task B represents the execution of the same task belonging to the second $(0 + \lambda(B) = 1)$ iteration of the original DDG. Since the maximum iteration index is $\lambda_{max} = \lambda(A) = 2$, the prologue must contain instances of task A belonging to iterations 0 to $\lambda_{max} - 1$ of the original DDG. In order to obtain the prologue we unroll the original DDG $(\lambda_{max} - 1)$ times and assign tasks to prologue. In the figure we unroll the original DDG one time. We assign tasks $A, BandC$ belonging to iteration zero; and task A belonging to iteration one to the prologue. Now assume that the steady state is executed zero times. Since

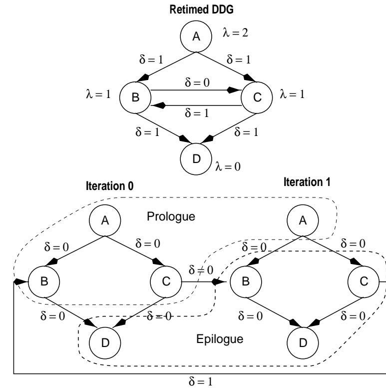

Figure 5. **Prologue and Epilogue Graphs**

the prologue has some tasks belonging to iterations 0 to $\lambda_{max} - 1$ of the original DDG, the epilogue must have the remaining tasks of the graph belonging to the same iterations. For every incomplete iteration i of the unrolled DDG in the prologue, the epilogue must have the remaining tasks which complete that iteration. We therefore assign task D belonging to iteration zero; and tasks $B, CandD$ of iteration one to the epilogue.

Performance Estimation of the pipelined design: After the prologue and epilogue graphs have been obtained the performance evaluation tool estimates the memory requirement and runtime of the prologue and epilogue graphs by using the list based scheduler. Let the runtime of the prologue and epilogue graphs be given by $runtime_{pro}$ and $runtime_{epi}$ respectively. Then the execution time of the pipelined design for n iterations of the steady state is given by: $ExecutionTime_{pipeline} = runtime_{pro} + n \cdot II + runtime_{epi}$. The memory requirement of the pipelined design is the maximum of the memory requirement of the steady state, prologue and epilogue.

3.4. Schedule Analyzer

The task runtimes which are stored in the DDG are representative task runtimes which may change depending on the inputs. Therefore the performance of the obtained schedule will also vary. The designer may need to analyze the performance of a given schedule for different task timings. This is done by the schedule analyzer. The schedule analyzer constructs another graph called the *estimate graph* which embeds the schedule information inside the DDG. Consider the example shown in Figure 6. The DDG contains 7 tasks labeled A to G. The schedule time of each task is shown in the figure. The execution order of the tasks according to the schedule on the SW processor is $[A, B, C, D]$. Similarly the execution order on the HW processor is $[F, E, G]$. Since

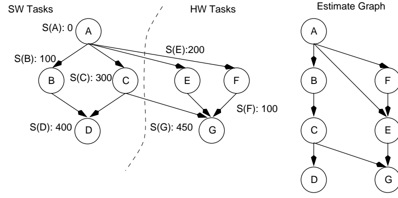

Figure 6. **Construction of Estimate Graph**

the schedule of the *DDG* honors the data dependencies between the tasks, the execution order of the tasks for a resource honors the data dependencies between tasks bound to that resource. Hence we can replace the data dependencies between tasks bound to the same resource, and add new directed edges between successive tasks in the order to obtain the estimate graph. In the figure all data dependencies between tasks bound to the same resource have been deleted, for example between tasks A and C. Extra edges are introduced between successive tasks in the execution ordering ,for example tasks F and E. Note we do not remove dependencies between tasks bound to different resources.

The user can determine the performance of the schedule by a breadth first traversal of the estimate graph. Also he can change the runtime of any task and obtain estimates about the performance of the schedule. Performance estimation using estimate graph gives the correct estimates for only the run time of a particular schedule. It does not take in to account any change in memory requirements that might occur.

4. Case Study

We discuss the HW-SW codesign of JPEG algorithm for rapid-prototyping [10]. The codesign architecture consists of a Pentium based PC operating at 100 Mhz and the protozone [15] coprocessor. The protozone board consists of two banks of 8 bit, 32K memory and a Xilinx 4010 FPGA. The memory access time of the protozone board is 125 ns. As a result the maximum operating frequency of the HW is limited to 8 MHz. The various tasks that are performed during JPEG compression and decompression are shown in Figure 7. Our implementation of JPEG differs from the standard since we process a 4x4 block of pixels instead of a 8x8 block. We do this because we are constrained by the size of the shared memory on the protozone board.

The SW code of the JPEG algorithm was profiled using commercial tools and it was noticed that the discrete cosine transform (DCT) task takes over 75% of the time both during compression and decompression [10]. The DCT is composed of smaller sub-tasks as shown in Figure 8. The profiling results indicated that scalar to scalar multiplication (ssprod) takes over 60%

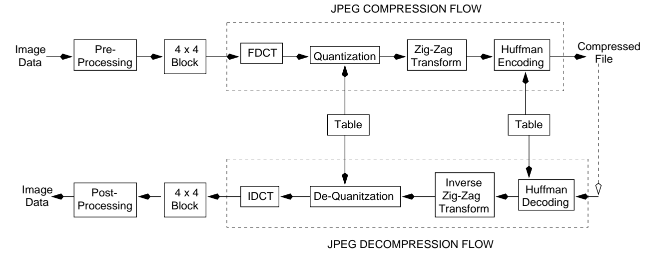

Figure 7. **JPEG Algorithm**

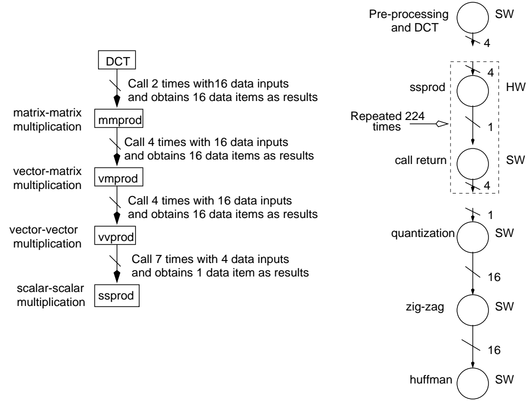

Figure 8. **Call Graph for ssprod and corresponding DDG representation**

of the total execution time. Hence as a first cut it was decided that ssprod be implemented in HW. The DDG representation of the design is shown in Figure 8 along with the task bindings. The JPEG algorithm was fully implemented and the achieved execution times for various designs were compared with the runtime estimates obtained by using the performance evaluation tool. The results of the comparison are in Table 1. As can be seen from the table we were able to obtain tight runtime estimates for both the compression and decompression algorithm. The tool required less than 2 seconds to generate each of these estimates.

It was observed that the JPEG implementation with ssprod in HW actually runs slower than the SW implementation. This was because of the communication overheads of the design. As can be seen from the *DDG* shown in Figure 9 for a single block of 16 pixels ssprod is called 224 times leading to the high communication times. Also it was noticed that with ssprod in HW there was an under-utilization of the available HW resources. Therefore it was decided to implement the entire DCT in HW. The *DDG* for the design was similar to the JPEG algorithm flow shown in Figure 7. It was a linear graph consisting of the four JPEG tasks. Again the design was implemented [10] and the achieved runtimes were compared with those obtained by our tool. The results of this comparison are in Table 2. We are able

Input File	Compression			Decompression		
	Run time (s)		%	Run time (s)		%
	Act.	Est.	Err.	Act.	Est.	Err.
Scenery	293	292.6	0.14	288	287.9	0.03
Portrait	284	284.4	0.14	279	297.8	6.7
Parrots	146	145.6	0.27	143	143.2	0.14
Turbo	34.85	34.5	1	34.12	33.94	0.5
Group	28.98	27.84	3.9	27.68	27.39	1.04
XV_sym	27.26	27.1	0.6	26.87	26.66	0.7

Table 1. Comparison of times for Design 1

Input File	Compression			Decompression		
	Run time (s)		%	Run time (s)		%
	Act.	Est.	Err.	Act.	Est.	Err.
Scenery	19.68	19.52	0.81	18.04	17.85	1.05
Portrait	18.69	18.97	1.49	16.85	17.35	2.96
Parrots	9.54	9.71	1.78	8.76	8.88	1.36
Turbo	2.4	2.3	4.16	2.26	2.11	6.63
Group	1.97	1.85	6.09	1.85	1.7	8.10
XV_sym	1.88	1.81	3.72	1.76	1.65	6.25

Table 2. Comparison of times for Design 2

to estimate the runtimes of the designs within 8% error margins. The time required to generate the estimates was about 1 second.

In both design implementations discussed so far the SW processor is idle when the HW executes and vice versa. JPEG is a loop oriented algorithm and therefore ideal for pipelined implementation. Pipelining of the design will allow the SW and HW to execute in parallel and increase the resource utilization of the design. The pipelined implementation of the JPEG algorithm overlapped the execution of preprocessing and DCT belonging to one iteration of the loop with quantization, zigzag transform and huffman encoding of the previous iteration of the loop [10]. The pipelined design required 4 seconds to compress a 10000 pixel file. The tool estimated the execution time for the same file as 4.8 seconds. The decompression of the same file required 4.2 seconds, while the performance estimate by the tool was for 4.14 seconds. The tool required less than 2 seconds to generate the estimates.

5. Conclusion

In this paper we presented a performance evaluation tool for rapid prototyping of pipelined and non-pipelined HW-SW codesigns. The case study of the JPEG algorithm showed that the tool is able to generate accurate estimates within a short time period (less than 2 seconds). The limitation of the tool is its rather simplistic data dependency graph representation. Another limitation is that the performance estimates are only as good as the representative task timings gathered during profiling. It is up to the designer to use a good test set to obtain these timings.

References

[1] S. Bakshi and D.D. Gajski, "Hardware/Software Partitioning and Pipelining," *Proceedings of* 34th *Design Automation Conference*, Anaheim, CA, June 1997.

[2] J. Buck, S. Ha, E.A. Lee and D.G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems," *International Journal of Computer Simulation*, VOL C-36, NO. 1, pp 24-35, January 1987.

[3] P. Bjorn-Jorgensen and J.Madsen, "Critical Path Driven Cosynthesis for Heterogeneous Target Architectures,", *Proceedings of Fifth International Workshop on Hardware/Software Codesign*, Braunschweig, Germany, March 1997.

[4] T. Benner and R. Ernst, "A combined Partitioning and Scheduling Algorithm for heterogeneous Multiprocessor systems," Technical Report CY-96-2, Technical University of Braunschweig, Germany.

[5] K.S. Chatha and R. Vemuri, "RECOD: A Retiming Heuristic To Optimize Resource And Memory Utilization in HW/SW Codesigns," *Proceedings of Sixth International Workshop on Hardware/Software Codesign (CODES-CASHE'98)*, Seattle, March 1998.

[6] R.K. Gupta and Giovanni De Micheli, "Hardware-Software Cosynthesis for Digital Systems", *IEEE Design and Test of Computers*, pp. 29-41, September 1993.

[7] R.W. Hartenstein, J. Becker and R. Kress, "A Profiling-driven Hardware/Software Partitioning of High-Level Language Specifications," *IFIP International Workshop on Logic and Architecture Synthesis*, Grenoble, France, 1995.

[8] S-Soo Lim, Y.H. Bae, G.T. Jang, B-Do Rhee, S.L. Min, C.Y. Park, H. Shin, K. Park, S-Mook Moon, C.S. Kim, "An Accurate Worst Case Timing Analysis for RISC Processors," *IEEE Transactions on Software Engineering*, Vol 21, No. 7, July 1995.

[9] R. Lauwereins, M. Engels, M. Adé and J.A. Peperstraete, "Grape-II: Graphical Rapid Prototyping Environment for Digital Signal Processing Systems,", *IEEE Transactions on Computers*, pp 35-43, February 1995.

[10] N. Narasimhan, V. Srinivasan, M. Vootukuru, J. Walrath, S. Govindrajan and R. Vemuri, "Rapid Prototyping of Reconfigurable Coprocessors," *Proceedings of the 1996 International Conference on Application-Specific Systems, Architectures and Processors,* IEEE Press, August 1996.

[11] C. Passerone, M. Chiodo, W. Gosti, L. Lavagno and A. Sangiovanni-Vincentelli, " Evaluation of trade-offs in the design of embedded systems via co-simulation,", Technical Report Version (UCB/ERL M96/12), University of California at Berkeley, Berkeley, CA.

[12] K.K. Parhi and D.G. Messershmitt, "Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding," *IEEE Transactions on Computers*, 178-195,1991.

[13] F. Sánchez, *Loop Pipelining With Resource And Timing Constraints*, Ph.D. Dissertation, UPC Universitat Politèchnica de Catalunya, Barcelona, Spain, October 1995.

[14] M.B. Srivastava and R.W. Broderson, "SIERA: A Unified Framework for Rapid-Prototyping of System-Level Hardware and Software," *IEEE transactions on Computer-Aided Design of Integrated Circuits and Systems*, VOL. 14, NO. 6, June 1995.

[15] Stanford University, *Protozone User's Guide*.

[16] C.A. Valderrama, A. Changuel, P.V. Raghavan, M. Abid, T. Ben Ismail and A.A. Jerraya, "A Unified Model For Co-Simulation And Co-Synthesis Of Mixed Hardware/Software Systems," *Proceedings of European Design and Test Conference*, March 1995, Paris, France.

[17] V. Živojnović, H, Koerner and H. Meyr, "Multiprocessor Scheduling with A Priori Node Assignment," *Proceedings of VLSI'94*, La Jolla, October 1994.