A Scheduling and Pipelining Algorithm for Hardware/Software Systems

Smita Bakshi†
Dept. of Electrical & Computer Engineering
University of California, Davis
Davis, CA 95616
bakshi@ece.ucdavis.edu

Daniel D. Gajski
Dept. of Information & Computer Science
University of California, Irvine
Irvine, CA 92697-3425
gajski@ics.uci.edu

Abstract

Given a hardware/software partitioned specification and an allocation (number and type) of processors, we present an algorithm to (1) map each of the software behaviors (or tasks) to processors, (2) pipeline the system specification, and (3) schedule the behaviors in each pipe stage, amongst selected hardware components and processors, so as to satisfy a throughput constraint at minimal hardware cost. Thus, to achieve high performance, not only are critical tasks implemented as pipelined hardware architectures, but the system is also divided into concurrently executing stages. Furthermore, to offset the cost of this increased concurrency, non-critical sections are implemented on processors or as cheaper hardware blocks. Our experiments demonstrate the feasibility of our approach and the necessity of system pipelining in high performance design.

1 Introduction

Digital system design, especially within signal and image processing, is an immensely complex task and requires the fine-tuning and balancing of a number of different parameters. Some of the design parameters that have a large impact on the final design are the number and types of components (such as ASICs, general purpose processors, FPGAs) used in the design, the interconnection and communication mechanisms for these components, and the manner in which the different tasks within the system are partitioned, scheduled and pipelined amongst these components. Partitioning refers to dividing tasks amongst components such that critical tasks are implemented on faster components, while the less critical tasks are implemented on slower and cheaper components. Pipelining refers to dividing the tasks into concurrently executing stages, to increase the effective parallelism, and hence performance of the design. Finally, scheduling refers to deciding the sequentiality or the order of tasks within a pipe stage and on a component.

In this paper, we present an algorithm for scheduling and pipelining a specification amongst selected hardware and software components. The hardware components refer to RTL components such as adders, multipliers, ALUs, registers, and so on, while the software components refer to general-purpose processors such as the Pentium and PowerPC. This algorithm is part of a larger design flow in which the specification is first spatially partitioned into hardware and software tasks, and a hardware implementation is then determined for each of the hardware tasks. After the spatial partitioning, our algorithm performs temporal partitioning, that is, it divides the specification into pipe stages and time slots within the pipe stages.

The spatial partitioning allows the user to take advantage of the unique features offered by hardware and software components. Thus, critical sections may be placed in hardware and the less critical sections (or the ones that require programmability) may be implemented as software. Temporal partitioning, on the other hand, determines the level of concurrency in the design required to achieve a desired throughput constraint. Temporal partitioning is most needed for high performance applications, when the fastest implementation of critical behaviors still violates constraints. In this paper, we present an algorithm for temporal partitioning, given a spatially partitioned specification.

The next section describes related work in pipelining and hardware/software partitioning. Section 3 defines the problem and illustrates it with an example. We present the algorithm in Section 4, and experimental results along with conclusions in Sections 5 and 6, respectively.

2 Previous Work

Research in hardware/software partitioning has mainly concentrated on spatial partitioning [3] [6] [10] [14] wherein a specification is divided amongst multiple ASICs and processors to best satisfy constraints on area and/or performance. However, these tools have not considered the possibility of pipelining the system, at any level of granular-
ity, and hence fail to explore the design space within which most large DSP systems lie. Our work uses a very simple approach for partitioning, but by pipelining the system, it is able to explore this high-performance design space.

Several algorithms for scheduling and pipelining a data flow graph [9] or a control-data flow graph [8] [12] have been proposed in the past. These algorithms certainly form a basis for our pipelining and scheduling algorithm. An important difference, however, is that along with pipelining, we also determine a processor selection and binding for each of the software tasks. This places additional constraints on the algorithm, thereby increasing its complexity. Other differences are minor and arise mainly because of the different levels of granularity of pipelining. Our level of granularity is a task (a sequence of RTL operations), whereas for most algorithms it is an RTL operation.

We would also like to mention that the concurrency achieved due to pipelining is similar to the paradigm of concurrent tasks in synchronous data flow machines [11]. By pipelining and scheduling we are simply trying to balance the level of concurrency and sequentiality in a system, so as to minimize cost for a throughput constraint.

3. Problem definition

As mentioned in the introduction, our algorithm for scheduling and pipelining is part of a larger design flow [2]. To motivate our problem definition, we briefly describe this design flow. Given a system-level specification, hardware and processor libraries, and a throughput constraint, we first estimate the performance (number of clock cycles) of each task (or behavior) within the specification, on each of the available processors. Based on these estimates and the throughput constraint, we determine the hardware/software partition, that is, we associate a hardware or software type with each behavior. For all the hardware behaviors we then synthesize an implementation using components from the hardware library. This implementation may, in turn, be pipelined to satisfy constraints. We then obtain a processor allocation, and finally schedule and pipeline the hardware/software partitioned specification into pipe stages and amongst the selected processors. In this paper, we focus on this problem of scheduling and pipelining a partitioned specification.

The input specification is represented as a control flow graph, CFG(V,E), where vertices (V) represent tasks, and edges (E) represent control dependencies. Each task is represented by a sequence of VHDL statements which, in turn, are represented as a CDFG (control data flow graph).

For the given CFG, our aim is to determine the schedule and pipeline that will satisfy a throughput constraint, T. If we think of time as a grid $T ime(x, y)$, where x is a continuum from $\{0 \cdots T\}$ representing time in ns, and $\{y = 1 \cdots \infty \}$ represents the number of pipe stages, then the problem may be defined as follows:

**Given:**
1. a partitioned control flow graph CFG(V,E) in which each $v \in V$ has been designated a hardware or software type
2. a processor allocation $P$ (number and type of processors) selected from the processor library.
3. an execution time, $T_v$ for every hardware node, $v \in V$
4. an execution time, $T_{vp}$ for every software node, on each of the available processors, $p \in P$.
5. the throughput constraint, $T$.

**Determine:**
1. For every behavior $v \in V$, a start and finish point in the grid, $(x_v, y_v)$ and $(x'_v, y'_v)$, where $x'_v > x_v$.
2. For every software behavior $v, v \in V$, a processor $p \in P$, which will be used to implement it.
3. For every processor $p \in P$, a utilization list containing pairs $(x_p^1, x_p^2)$, indicating time intervals when the processor is utilized.

**Such that:**
1. $x'_v - x_v = T_v$, where $T_v$ is the execution time in hardware or on a selected processor $p_v$, if $v$ is a software behavior.
2. $x'_v < T$, $\forall v \in V$.
3. If $w \in Predecessor(v)$ is mapped to the start and finish grid points $(x_w, y_w)$ and $(x'_w, y'_w)$, then $y_w \leq y_v$. Furthermore, if $y_w = y_v$, then $x_v > x'_w$.
4. If $v$ is a software behavior mapped to processor $p_v$, then the time interval, $(x_v, x'_v)$, does not overlap with existing time intervals in the utilization list of processor $p_v$.

In simpler words, we assign each behavior, $v$, to a pipe stage and to a time slot within a pipe stage such that predecessor behaviors of $v$ finish their execution either in a previous stage, or in the same stage before the behavior $v$ begins its execution. Furthermore, if $v$ is a software behavior, then we have to make sure that the processor we select to execute it, is not used by any other behavior during the time interval that it is executing behavior $v$. This assignment of behaviors to pipe stages is done so as to (1) satisfy the throughput constraint, $T$, using the given processor allocation, $P$, and (2) to reduce the number of pipe stages (and therefore, the latency of the design and also the memory required to store data between pipe stages).
4 Algorithm overview

Our algorithm, outlined in Figure 2, is an extension of the well known list-scheduling algorithm [7]. We start by determining the longest completion time from each node till all output nodes, assuming that the fastest processor is used to execute a software node. This completion time gives the priority one node has over another during scheduling. The completion time of a node is a direct indication of its criticality, and hence, the higher the completion time, the higher its priority. In the example CFG in Figure 1 the completion times of nodes A, B, C and D are 17, 15, 10 and 2 respectively. Hence, the priority from highest to lowest is A, B, C and D.

As mentioned in the previous section, with each processor, we associate a utilization list within which we store the durations in which the processor is being utilized to execute any of the software behaviors. In step 2, we reset the utilization list of all processors to empty. We then form a list of ready nodes, that is nodes whose predecessors have already been scheduled. These nodes are prioritized using the completion time priority function.

After forming the ready nodes’ list and the utilization list for the processors, we find the “best time slot” for every node in the ready list, starting with the first (or highest priority) node. For nodes that are labeled as software, the “best time slot”, of course depends on the processor selected to implement that node. Hence, a by-product of determining the best slot time is a processor selection for every software behavior. After a time slot (and processor) has been assigned for a node, we mark it as “scheduled”, and then remove it from the ready list. Once removed, we update the ready list by checking to see if any of its successor’s is now ready to be scheduled. If so, we add the successor(s) to the ready list.

Steps 5 to 17 are repeated till all nodes in the CFG are scheduled.

1. For every node in CFG, determine longest completion time from that node till any output node. Assign node priorities.
2. Initialize the utilization list of all processors.
3. Form a new ready list.
4. Loop
5. current_node = first node in ready list
6. If (current_node is type software)
7. Find processor and corresponding time slot that gives earliest completion time.
8. If (no available processor)
9. Exit (No feasible solution)
10. Else
11. Assign current_node to processor & time slot.
12. Update utilization list of processor.
13. End if
14. Else if (current_node is type hardware)
15. Assign current_node to earliest feasible time slot.
16. End if
17. Mark current_node as scheduled, remove from ready list, and update ready list.
18. Until (all nodes in CFG are scheduled).

Figure 2. Algorithm overview.
4.1 Determining the best time slot

Every node should ideally execute immediately after all its predecessors have completed execution, in the same pipe stage as its last predecessor\(^1\). However, this may not be feasible for two reasons. Firstly, by scheduling it immediately after its predecessors, the throughput constraint may be violated, that is, the pipe stage delay might exceed the constraint. Secondly, in the case of a software node, a processor for the desired duration may not be available. In both these situations it will then be necessary to schedule the node in the next pipe stage.

The procedure for determining the best time slot is outlined in Figure 3. The inputs to this procedure are the partially scheduled and pipelined CFG, the utilization list of all processors, and the node to be scheduled (current_node). We first select the maximum of all the pipe stages in which predecessors of current_node have been pipelined. We call this max_pipestage. We then determine the latest time at which predecessors in max_pipestage finish execution. This is given by the variable max_preddlalay. These two quantities give us the earliest pipe stage and earliest time within that pipe stage that current_node may be scheduled in. Next, we check to see if current_node is in hardware or software. If it is a hardware node, we assign it to max_pipestage if it can complete its execution within the throughput constraint, else we assign it to max_pipestage + 1.

If it is a software node, we check the utilization lists of all processors and find the earliest completion time of current_node on all processors. At this point, we check two possibilities:

- the execution can begin after max_preddlalay in max_pipestage
- the execution can begin at time 0 in max_pipestage + 1

From these two possibilities, we select the one which gives the earliest completion time. This may result in current_node being scheduled in max_pipestage or in max_pipestage + 1.

We explain the procedure for determining the best time slot by using the example in Figure 1. The algorithm starts by finding the longest delay from each node till any output node, assuming execution on the fastest processor. Thus, starting from the bottom of the CFG, node D has a delay of 2 ns (we’ll drop the ns from now on) assuming execution on processor P1, node C has a delay of 10 (8+2), node B of 15 (10+5), and node A of 17 (10+7), giving node A the highest priority and node D the lowest. The initial prioritized ready list, then, consists of nodes A and B in that order.

We start by finding the best time slot for node A. We have a choice of three processors and corresponding three time slots: from 0 to 7 on processor P1, 0 to 9 on processor P2, and 0 to 11 on processor P3. This is indicated in the Completion Time Table in Figure 4(a). Each entry in the table gives the completion time and the pipe stage for a specific behavior on a specific processor. Note that the completion time table is not built before we start scheduling, but an entry for a node is appended to the list when the node is at the top of the ready list, next in line to be scheduled.

Of the three choices for node A we select processor P1 since that gives us the earliest completion time. Node A is thus scheduled in the first pipe stage, starting at time 0 and ending at time 7. The schedule as well as the utilization list of processor P1 is updated as shown in Figure 4(c) and (b), respectively. Next, the completion times for node B on all the processors is calculated, and processor P2 is selected

\(^1\)The last predecessor is the predecessor that finishes its execution last in the last pipe stage amongst all predecessors. For instance, if Predecessor 1 finishes execution in stage 1, time 100 ns, and Predecessor 2 finishes execution in stage 2, time 50, then Predecessor 2 is the last one.
5 Experiments

We have implemented the scheduling and pipelining algorithm within the design flow outlined in Section 3. This work has been incorporated into SpecSyn [4], a system-level synthesis tool. Our experiments indicate the performance gains obtained by pipelining at the system level and at the behavioral level. They also demonstrate the extent and nature of design space explored. In an attempt to gauge the quality of our algorithm, we also compare the algorithmic design exploration against a manual one.

The experiments are conducted for the MPEG decoder II system and for the Volume system, an example of a medical instrument. Both, the MPEG and the Volume specification contain 14 behaviors or tasks (hence, 14 nodes in the CFG), though the MPEG specification is the larger of the two, with 1085 lines of code as opposed to 238 for the Volume system. For both examples we use a clock of 10 ns, a software library containing 6 processors (Intel’s 8086, 8088, Pentium, Sun’s SPARC, Motorola’s 68000 and PowerPC), and a hardware library containing multiple implementations of RTL components such as adders, multipliers, and comparators. The processor and hardware execution time tables are obtained by using software [5] and hardware [1] estimators implemented within SpecSyn.

In order to explore the design space of both examples, we run our algorithm with a range of throughput constraints, from low to high. Our algorithm returns a set of designs ranging from all hardware, highly pipelined solutions, to ones containing processors and possibly custom hardware, and finally to the slowest ones that are implemented solely on a processor. Results for this algorithmic exploration are shown in Tables 1 and 3, in Figure 5. For the MPEG example, the exploration was also performed manually [13] (Table 2) For both examples, the tables indicate the number of behavioral level. They also demonstrate the extent and nature of design space explored. In an attempt to gauge the quality of our algorithm, we also compare the algorithmic design exploration against a manual one.

The experiments are conducted for the MPEG decoder II system and for the Volume system, an example of a medical instrument. Both, the MPEG and the Volume specification contain 14 behaviors or tasks (hence, 14 nodes in the CFG), though the MPEG specification is the larger of the two, with 1085 lines of code as opposed to 238 for the Volume system. For both examples we use a clock of 10 ns, a software library containing 6 processors (Intel’s 8086, 8088, Pentium, Sun’s SPARC, Motorola’s 68000 and PowerPC), and a hardware library containing multiple implementations of RTL components such as adders, multipliers, and comparators. The processor and hardware execution time tables are obtained by using software [5] and hardware [1] estimators implemented within SpecSyn.

In order to explore the design space of both examples, we run our algorithm with a range of throughput constraints, from low to high. Our algorithm returns a set of designs ranging from all hardware, highly pipelined solutions, to ones containing processors and possibly custom hardware, and finally to the slowest ones that are implemented solely on a processor. Results for this algorithmic exploration are shown in Tables 1 and 3, in Figure 5. For the MPEG example, the exploration was also performed manually [13] (Table 2) For both examples, the tables indicate the number of system-level pipe stages and the processors and/or area required for different throughput values. Recall that system-level pipelining divides the CFG of tasks into pipe stages. This is as opposed to behavioral-level pipelining, in which each task in the CFG is further pipelined.

We make the following observations from the results in Tables 1 and 3:

- For the MPEG example, the fastest throughput attainable by our algorithm is 2980 ns. This design contained 12 pipe stages at the system-level. In addition, about 5 of the 14 behaviors were further pipelined into stages ranging in number from about 3 to 6. If we do not pipeline the design at the system-level, but still allow pipelining within behaviors, the fastest design we can obtain has a throughput of about 30000 ns, that is approximately 10 times that of the design obtained with system pipelining.

- For the Volume system, the fastest throughput attainable by our algorithm is 420 ns. However, without system-level pipelining, the throughput is approximately 2500 ns, which is about 6 times the highest throughput.

Both these observations indicate that in order to obtain high throughputs it is not sufficient to pipeline individual
behaviors. The pipelining needs to be extended to the system level such that the control flow graph of behaviors is further divided into concurrent stages, thereby reducing the throughputs.

From these results, we can also obtain a comparison of the algorithmic and manual design process for the MPEG example (Tables 1 and 2). Note that in the manual designs, the hardware area is the area of the entire datapath and controller, while our algorithm just estimates the functional unit and the memory area. Hence, in general, the area of the manual design is higher than the estimated area. The results indicate that the design exploration conducted by our algorithm closely matches the manual exploration. Though the accuracy of our hardware implementations is not very high, its fidelity is extremely high. Also note that our algorithm obtained a faster design, than a designer could obtain manually. This is attributed to hierarchical pipelining (within a system, behavior, loop, and operation) which an algorithm performs easily, but is complex for designers to perform manually.

6 Conclusions

We have presented an algorithm to schedule and pipeline a hardware/software partitioned specification, given a processor allocation and a throughput constraint. Results have indicated the feasibility of our approach as well as the necessity of system and behavioral pipelining in order to obtain high data rates. However, our approach has several drawbacks, some of which we are currently addressing. Principal amongst these is the exclusion of communication overheads (in terms of area and delay) in our model and results. After this and other improvements, such as estimation of controller area, we expect our approach to offer a viable and a much faster alternative to manual design and exploration.

References