Hardware–Software Partitioning and Pipelined Scheduling of Transformative Applications

Karam S. Chatha, *Member, IEEE*, and Ranga Vemuri, *Senior Member, IEEE*

***Abstract*—Transformative applications are computation intensive applications characterized by iterative dataflow behavior. Typical examples are image processing applications like JPEG, MPEG, etc. The performance of embedded hardware–software systems that implement transformative applications can be maximized by obtaining a pipelined design. We present a tool for hardware–software partitioning and pipelined scheduling of transformative applications. The tool uses iterative partitioning and pipelined scheduling to obtain optimal partitions that satisfy the timing and area constraints. The partitioner uses a branch and bound approach with a unique objective function that minimizes the initiation interval of the final design. We present techniques for generation of good initial solution and search-space limitation for the branch and bound algorithm. A candidate partition is evaluated by generating its pipelined schedule. The scheduler uses a novel retiming heuristic that optimizes the initiation interval, number of pipeline stages, and memory requirements of the particular design alternative. We evaluate the performance of the retiming heuristic by comparing it with an existing technique. The effectiveness of the entire tool is demonstrated by a case study of the JPEG image compression algorithm. We also evaluate the run time and design quality of the tool by experimentation with synthetic graphs.**

***Index Terms*—Image processing, partitioning, performance tradeoffs, pipelining, scheduling, system-level design.**

I. Introduction

TRANSFORMATIVE applications are multimedia and digital signal processing (DSP) applications. Typical examples are JPEG and MPEG (1 and 2) image compression-decompression algorithms, Viterbi decoding etc. Embedded system implementations of transformative applications require them to be cost effective, high performance, and flexible. As a result most of them are implemented by heterogeneous architectures that utilize off the shelf software (SW) processor cores and custom hardware (HW) coprocessors. The SW processors reduce the cost of the system and provide flexibility. The custom HW coprocessors implement the computation intensive components of the application and enhance the performance of the system. For example, Eijndhoven *et al.* in [1] discuss the Philips TM1000 series very large instruction word (VLIW) processor core that might be combined with different on-chip coprocessors to obtain the desired performance/silicon area ratio. The coprocessors communicate with each other and the processor core through an on-chip bus and share access to a single off-chip memory. Takata *et al.*in [2] present the D30V/MPEG multimedia processor that consists of a processor core and dedicated HW to perform MPEG-2 video and audio decoding. Ikeda *et al.* [3] discuss the SuperEnc MPEG-2 video encoder chip that consists of a reduced instruction set computing (RISC) processor, a single instruction multiple date (SIMD) processor and dedicated HW. HW–SW codesign transforms an application specification into communicating HW and SW components of an embedded system that exhibit the desired behavior and satisfy the performance constraints. HW–SW codesign consists of two basic design stages: HW–SW partitioning partitions the application specification into HW and SW components and scheduling specifies the execution order of these components. The authors in [3] also present the pipelined schedule for MPEG-2 encoding on their architecture. This paper presents a tool for HW–SW partitioning and pipelined scheduling of transformative applications.

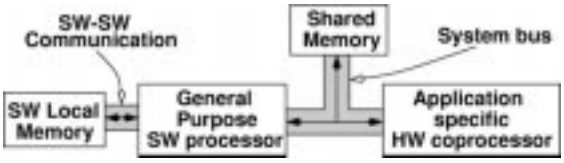

Fig. 1. Implementation architecture.

A. Characteristics of the Application Domain

Transformative applications are dominated by dataflow behavior with few control-flow constructs. Further, they can be easily broken down into distinct tasks at a coarse level of granularity. The task are computation intensive and internally strongly interconnected with sparse external communication. The previous observations imply that transformative applications can be specified by a data dependence based task-graph format. Finally, these applications are iterative in nature. They execute repeatedly over different sets of input data. Hence, they naturally lend themselves to pipelined implementations. Similar observations were also made by De Man *et al.* in [4].

B. Implementation Architecture

We implement the applications as a codesign architecture that consists of a single SW processor, a HW coprocessor, a shared memory for HW–SW communication and SW local memory for SW–SW communication (see Fig. 1). The SW processor is

Manuscript received August 15, 1999; revised March 10, 2000. This work was supported in part by the ARPA RASSP Program and USAF, Wright Lab, under contracts F33615-93-C-1316 and F33615-97-C-1043.

K. S. Chatha was with the Department of Electrical and Computer Engineering, University of Cincinnati, OH 45221-0030 USA. He is now with the Department of Computer Science and Engineering, Arizona State University, Tempe, AZ 85287-5406 USA (email: karam.chatha@asu.edu).

R. Vemuri is with the Department of Electrical and Computer Engineering, University of Cincinnati, OH 45221-0030 USA (e-mail: ranga.vemuri@uc.edu).

Publisher Item Identifier S 1063-8210(02)00794-1.

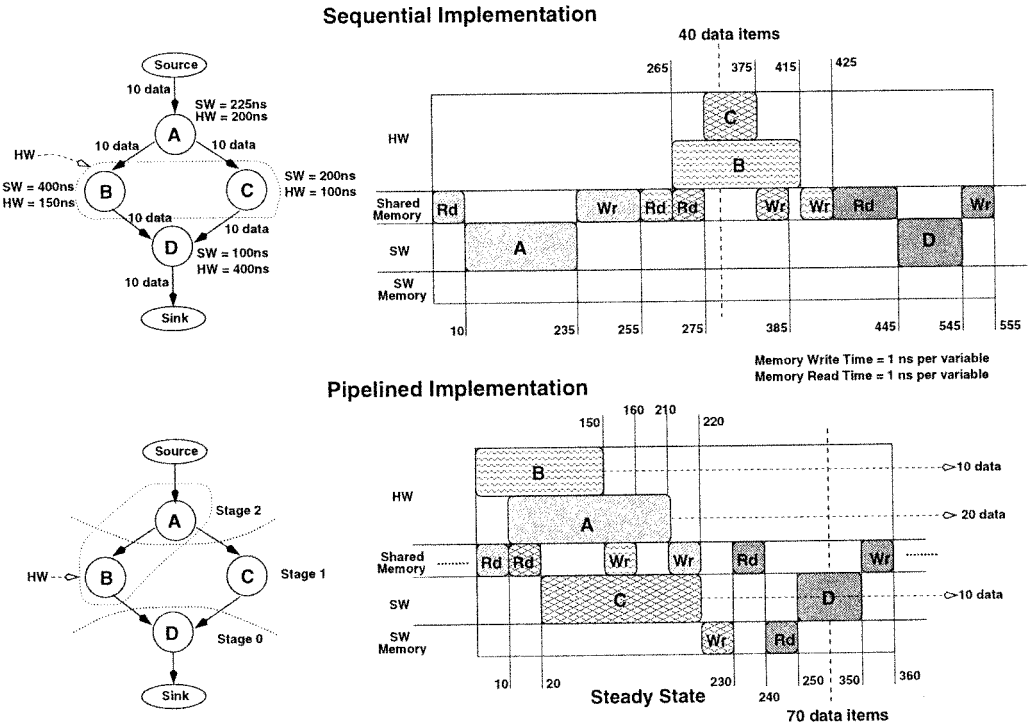

Fig. 2. Sequential versus pipelined design.

a uniprocessing system. The HW coprocessor supports concurrent execution of multiple HW tasks. The shared memory and SW local memory are exclusive read and exclusive write. HW and SW tasks communicate with each other through the shared memory.

A SW (HW) task, when it begins execution, first reads data from the shared memory and then from its local memory (on-chip registers). After computation, the SW (HW) task first writes to the SW local memory (on-chip registers) and then to the shared memory. The SW processor is considered busy from the start of the read operation of the executing SW task till the end of the write operation of the same task. A task during its execution occupies memory required by the data items belonging to both its read set and write set. The communication memory required by the codesign implementation is estimated as the maximum memory occupied during one complete execution of the task graph. The memory calculation includes shared memory, SW local memory and on-chip registers of the HW coprocessor.

The embedded system designer specifies the communication times per data item for interprocessor (SW–HW) communication and intraprocessor (SW–SW, HW–HW) communication. We denote the time to read and write a data item to the shared memory (SW local memory) as $shrd(swrd)$ and $shwr(swwr)$, respectively. Two HW tasks communicate through on-chip registers and there communication time is included in their task execution times. The designer also specifies the communication area overhead, a_{comm} due to the interface logic required by a HW task to communicate with the shared memory. Since the shared memory is exclusive read-exclusive write, the HW tasks can share the communication resource.

C. Motivating Example

Consider the example in Fig. 2, the task graph is shown on the left-hand side along with run times of the tasks and data items transferred across each dependence. There exists an area constraint on the HW coprocessor such that only two (any two) tasks can be implemented in HW. The objective is to partition and schedule the tasks such that throughput is maximized. We assume that the source and sink nodes write to and read from the shared memory, respectively. The time required to read or write one data item from shared memory or SW local memory is assumed to be 1 ns. As mentioned earlier the communication memory required by the codesign implementation is estimated as the maximum memory occupied during one complete execution of the task graph.

Partitioning for sequential implementation is significantly different from partitioning for pipelined implementation. When partitioning for sequential implementation, the objective is to minimize the time for one complete execution of the task graph. In the example, tasks B and C have been mapped to the HW coprocessor. Task A that is assigned to SW initially reads the input data from the shared memory. On completion, task A writes its output to shared memory. Tasks B and C then read their inputs from the shared memory, execute concurrently and write their outputs back to the shared memory. Finally, task D reads its data from the shared memory and produces the output of the task graph. The schedule takes 555 ns for completion. The communication memory required by the schedule is estimated to be 40-data items, as shown by the vertical dashed line. At the time instance indicated by the dashed line, tasks B and C require memory for their read and write data items, respectively.

The pipelined implementation is shown in the lower half of the figure. A pipelined schedule is characterized by its *initiation interval*, (II), which is the time difference between the start of two consecutive iterations of the steady state. For example, the pipelined schedule in Fig. 2 has an $II = 360$ ns. Given a partitioned task graph, there exists a theoretical lower bound on the II of its pipelined schedule called the *minimum initiation interval* (MII).*The objective of the HW–SW partitioner is to map the tasks to HW and SW so that MII is minimized, subject to the area constraints.* We present a HW–SW partitioner that uses a branch and bound approach to obtain optimal HW–SW partitions. We discuss techniques for generation of good initial solution and search-space bounding. As the experimental results will show later, the initial solution is on an average within 6.3% deviation of the optimal. As a result, we are able to effectively limit the search space and generate optimal partitions in a shorter period of time.

We evaluate a given partition by generating its pipelined schedule. We obtain pipelined schedules by retiming and scheduling. Pipelining leads to an increase in the communication memory requirements of the design. This increase is due to the extra memory required by the pipeline buffers that are used to communicate between tasks belonging to different iterations. The total memory requirement of the pipelined schedule is 70 variables. The horizontal dashed lines from tasks A, B, and C indicate the extra memory required by the pipeline buffers. *The objective of the pipelined scheduler is to obtain a schedule with II as close as possible to MII with least number of pipeline stages and least increase in memory requirements.* In this paper, we present a novel retiming heuristic that optimizes the initiation interval, number of pipeline stages, and memory requirements due to pipeline buffers of a pipelined HW–SW implementation.

D. Application Specification

The application is described as a directed acyclic data dependence based task graph (DAG) $G = (V, E, So, Si)$, where V is the set of tasks, the edge set E, represents the data dependence between any two tasks, So and Si are special nodes called the source node and sink node, respectively. We assume that the DAG is executed iteratively over different sets of input data. Associated with each task $v \in V$ are four quantities: v_{sw}, the execution time of the task v on the general purpose SW processor, v_{hw} the execution of the task v in HW, v_{area} the area occupied by the task when implemented on the HW coprocessor and $\lambda(v)$ the iteration index of the task. v_{sw} can be obtained by software profiling. v_{hw} and v_{area} of a task are obtained by using a high-level synthesis tool. Each edge $e \in E$ has two quantities associated with it: e_{data} the number of data items transferred across a dependence and $\delta(e)$, the dependence distance. Iteration index λ and dependence distance δ are used to retime the DAG for pipelined scheduling. The source and sink nodes are used to model the environment and specify the throughput constraint. The throughput constraint is specified as the number of input data sets consumed per second by the task graph from the source node.

The iteration index of a task $\lambda(v)$ implies that at the ith iteration of the steady state of the pipeline, instance of task v belonging to $i+\lambda(v)$ iteration of the original DAG is executed. In the pipelined design shown in Fig. 2, at the zeroth iteration of the steady state, instance of task A belonging to the second iteration of the original DAG is executed. Hence, $\lambda(A) = 2$. Similarly, $\lambda(B) = 1$, $\lambda(C) = 1$, and $\lambda(D) = 0$. The dependence distance $\delta(e)$ of an edge $e = (u, v)$ implies that the data produced by task u in the ith iteration of the steady state is consumed by task v in the $i+\delta(e)$ iteration of the steady state. For example, in the pipelined design shown in Fig. 2, the data produced by task A in iteration 0 of the steady state is consumed by task B in iteration 1. Hence, $\delta(A, B) = 1$. Similarly, $\delta(A, C) = 1$, $\delta(B, D) = 1$, and $\delta(C, D) = 1$.

E. Problem Description

Given an application specified as a task graph, area constraint ($\alpha_{\mathrm{constraint}}$) on the HW coprocessor and/or time constraint ($\tau_{\mathrm{constraint}}$) on the execution of the task graph:

1) partition the task nodes between the HW coprocessor and SW processor;
2) obtain a pipelined schedule for the task execution and task communication;

such that:

1) the performance constraints are satisfied;
2) the number of pipeline stages in the implementation are minimized;
3) the increase in memory due to pipeline buffers is minimized.

Since resource constrained scheduling is a nonpolynomial (NP) complete problem [5], pipelined scheduling is also NP complete [6]. Pipelined schedules are obtained by decomposing the problem into two subproblems: retiming transformation followed by scheduling. Retiming under resource constraints has been shown to be NP complete [7].

The paper is organized as follows: In Section II we discuss previous work, Section III presents the tool, Section IV discusses the experimental results, Section V discusses the possible extensions and limitations of our work and finally, Section VI concludes the paper.

II. Previous Work

A. HW–SW Codesign

In recent years, a number of approaches for HW–SW codesign have been proposed. Most approaches focus on a particular design stage in the codesign process and can be differentiated as such. For example, the Ptolemy [8] system addresses the problem of specification and cosimulation of HW and SW components. Chinook [9], on the other hand focuses on interface synthesis during HW–SW cosynthesis. Our work addresses the problem of automatic HW–SW partitioning and pipelined scheduling of transformative applications. Therefore, the related work discussed in the following paragraphs concentrates on approaches that perform automated HW–SW partitioning and scheduling.

Gupta *et al.* in [10] presented a fine grained HW oriented partitioning algorithm that moved nodes from HW to SW while the timing constraint was satisfied. In [11], Henkel

et al. proposed a SW oriented HW–SW partitioner based on simulated annealing. The partitioner initially mapped all the components to SW and then moved them from SW to HW until the performance constraints were satisfied. Kalavade *et al.* in [12] discussed a modified list scheduling algorithm that mapped the tasks to HW and SW. The algorithm used an adaptive global criticality local phase (GCLP) heuristic that either minimized the execution time of the design or the area. Niemann *et al.* in [13] presented an integer linear programming formulation for HW–SW partitioning. Knudsen *et al.* in [14], presented a dynamic programming based approach for HW–SW partitioning. Eles *et al.* in [15], proposed HW–SW partitioning algorithms based on simulated annealing and tabu search. Our tool differs significantly from these approaches. All of them used a partitioning strategy and an objective function to satisfy the performance constraints for a sequential implementation. As explained earlier (see Fig. 2), partitioning for sequential implementation is significantly different from partitioning for pipelined implementation. In contrast, our partitioner uses a unique objective function aimed at maximizing the throughput of the pipelined implementation. Further most of these approaches [10], [11], [14], and [15] perform HW–SW partitioning and scheduling in isolation, whereas we adapt an integrated iterative approach.

The approach proposed by Jinhwan *et al.* in [16], minimizes the latency of the HW–SW implementation by increasing the parallelism through pipelining the inner loops of the specification. They apply loop pipelining before HW–SW partitioning. In contrast, the objective of our technique is to maximize the throughput of the specification. Loop pipelining before HW–SW partitioning is tedious since the nodes are not mapped and their delays are unknown. We perform pipelined scheduling after HW–SW partitioning.

Our methodology and the techniques mentioned in the previous paragraphs partition the specification on to a fixed heterogeneous architecture template. Prakash *et al.* [17], Dave *et al.* [18], Dick *et al.* [19], and Li *et al.* [20] proposed approaches for synthesizing a heterogeneous architecture that satisfies the timing constraints on the specification. Prakash *et al.* [17] formulated the problem as a mixed-integer linear-programming model and obtained a sequential (nonpipelined) implementation. Dave *et al.* [18] used a heuristic based task clustering, allocation, and scheduling approach for synthesizing hierarchical heterogeneous architectures. Their technique implements a pipelined design when the period constraint on the task graph is smaller than the deadline constraint. The designer specifies the number of pipeline stages and algorithm then performs clustering on the task graph to obtain the desired number of stages. In our approach, the designer is required to specify only throughput and (or) area constraint. Our technique optimizes the number of pipeline stages and memory required for pipelining. However, we partition the specification on to a fixed architecture template. Dick *et al.* [19] proposed a genetic algorithm based architecture synthesis approach that uses a scheduling technique similar to [18]. Li *et al.* [20] uses a heuristic based approach for architecture and memory hierarchy (cache) synthesis. They use a preemptive static scheduling algorithm that hierarchically allocates and schedules tasks on multiprocessors and memory transfers on the bus to meet the real-time constraints. Task preemption is too expensive both in time and space for many high-volume low-cost embedded systems [21]. Hence, we use a nonpreemptive static pipelined scheduling policy.

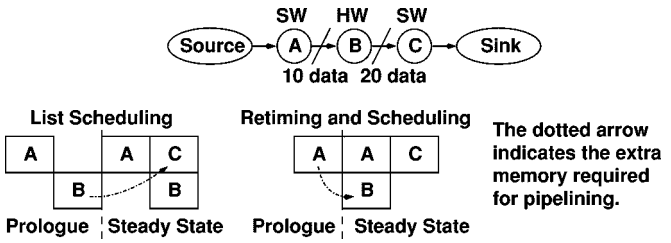

Fig. 3. List scheduling versus iterative retiming and scheduling.

Bakshi *et al.* [22] proposed an architecture synthesis approach for pipelined implementation of HW–SW codesigns. In their approach, they do not consider the communication delays. Their pipelined scheduling approach is based on modulo scheduling that was introduced by Rau *et al.* [23]. Modulo scheduling involves using a list scheduling algorithm with a modulo resource reservation table whose height is equal to pipeline initiation interval. Since the list scheduling algorithm is a greedy algorithm, modulo scheduling based techniques cannot explore the task graph to obtain a pipelined implementation with lower pipelining memory (or buffers). In the example shown in Fig. 3, the pipelining memory (as shown by the dotted arrow) is more for the modulo scheduling approach as opposed to our technique. In contrast to the list scheduling based approach, our technique can select which data dependence should contribute to the pipelining memory. Hence, we are able to optimize pipeline stages and memory requirement of the design. In contrast to [22], we account for communication overheads due to actual communication and shared memory conflicts during scheduling.

B. Retiming Transformation and Pipelined Scheduling

The term “Retiming transformation” was introduced by Leiserson and Saxe [24] when they used it to solve the problem of optimizing the throughput of a synchronous circuit. Leiserson *et al.* presented polynomial time techniques for clock–period minimization and area (register) minimization of a logic circuit. Since then, retiming transformation has been extensively used as an optimization technique during logic synthesis [25]–[29]. Shenoy *et al.* [30] give a good survey of logic circuit retiming techniques and their application to realistic circuits. Retiming transformation has also been applied for optimizing the throughput of DSP circuits specified as dataflow graphs [31], [32]. Retiming at architectural level [33] has been applied to reduce the latency of a constraining path in a design.

Retiming transformation applied to combinational circuit pipelining [24] assumes that there is no resource constraint on the combinational (or computation) blocks. Retiming transformation under resource constraints is NP complete [7]. Hence, we use a heuristic-based retiming-transformation to obtain the pipelined schedules. Our problem formulation is similar to software pipelining [34] in VLIW compiler literature and functional pipelining [35] in high-level synthesis literature.

In the following paragraphs, we discuss existing software and functional pipelining approaches. Although techniques that belong to these domains may be extended to HW–SW codesign, there are certain limitations. Our technique optimizes for system level coarse grained asynchronous pipeline design, whereas software and functional pipelining techniques concentrate on fine grained synchronous pipelined designs. In our application, domain communication times and shared memory access conflicts need to be considered. These overheads are ignored in fine-grained pipelining. HW–SW codesign introduces heterogeneity in terms of processing elements that can execute either multiple (HW) or single (SW) task. This kind of heterogeneity is not encountered in either software pipelining or functional pipelining. To the best of our knowledge, [36] is the only other technique that optimizes both the initiation interval and pipeline registers of the final design. DSP codes like fast Fourier transform (FFT), fifth-order elliptic filter, and fourth-order lattice filter, etc., are implemented as fine-grained pipelines, whereas transformative applications that are much larger are implemented as coarse-grained pipelines. Finally, fine-grained techniques typically concentrate on optimizing dependence loops that have interiteration (or loop carried) dependences. Such dependences are not common in transformative applications. In Section V we discuss extensions to our work for handling dependence loops.

Rau *et al.* [23] proposed the modulo scheduling technique for VLIW software pipelining. Lam *et al.* [34] discussed modifications to modulo scheduling for handling conditional branches inside the loops. Moon *et al.* [37] proposed a technique that repeatedly applied semantic preserving transformations to obtain a pipelined loop. Aiken *et al.* [38] proposed a technique for software pipelining based on unrolling and scheduling the loop. Similar to our technique, Wang *et al.* [39] and Calland *et al.* [40] obtain pipelined software loops by decomposing the problem into two subproblems. In contrast to our approach, these techniques do not optimize the additional registers required for pipelining. To the best of our knowledge, only Govind *et al.* [36] have proposed a technique that minimizes the number of pipeline registers of a software pipeline. They proposed an integer linear programming (ILP) formulation that is limited by large solution times.

Park *et al.* [41] proposed feasible scheduling algorithm for pipeline datapath synthesis from behavioral specifications subject to either resource or time constraints. Paulin *et al.* [42] proposed extensions to their force directed scheduling algorithm for functional pipelining. Lee *et al.* [43] proposed modulo scheduling heuristics for functional pipelining under timing and resource constraints. Cathedral II [44], rotation scheduling [45], and MARS (I and II) synthesis system [46], and [47] generate pipelined datapaths for DSP applications. Cathedral II applies iterative loop folding (similar to retiming) under timing constraints. Rotation scheduling utilizes implicit retiming to obtain pipelined schedules under resource constraints. The MARS (I-II) system accepts time constraints and utilizes heuristics with enhanced modulo scheduling to optimize the interiteration dependences. Sánchez [48] proposed a decomposition based technique for functional pipelining that applies the retiming transformation for optimizing the throughput. In contrast to these approaches, our technique optimizes both the throughput and memory requirements of the pipelined design under resource and/or time constraints.

Our graph representation and problem formulation is similar to the paradigm of synchronous data-flow machines of Lee *et al.* [49]. They assume a multiprocessor environment where each DSP processor can execute a single task at a given-time instance. Although, there might be heterogeneity in terms of DSP processors, the processing elements display homogeneity in terms of task execution. This is not true in our application domain. The HW coprocessor can execute multiple tasks concurrently. Further, the concept of area constraint on the HW coprocessor does not apply to their problem formulation.

III. HW–SW Partitioning and Pipelined Scheduling

In this section, we present our technique for HW–SW partitioning and pipelined scheduling of transformative applications. We give an overview of the technique in Section III-A; Section III-B discusses the HW–SW partitioner; Section III-C discusses the pipelined scheduler, and finally, Section III-D presents our retiming heuristic.

A. Overview

We use iterative HW–SW partitioning and pipelined scheduling to obtain a pipelined implementation that satisfies the performance constraints. The objective of the HW–SW partitioner is to obtain a mapping, such that the MII of the partitioned task graph satisfies the time constraint, and the total area of the tasks mapped to HW satisfies the area constraint. The area of the HW coprocessor is estimated by adding the areas of all tasks mapped to HW. Before we present an overview of the tool we explain the procedure to calculate MII.

1) Minimum Initiation Interval (MII): Given a partitioned task graph, it is possible to establish a lower bound on the initiation interval of the pipelined design. The initiation interval (II) as stated earlier is the time difference between the start of two successive iterations of the steady state. The theoretical lower bound on the II is called the minimum initiation interval (MII). Since we consider a directed acyclic task graph, the MII of the pipelined design is determined by the task execution times and the number of resources (HW or SW) in the implementation architecture.

The execution time for a task v in a task graph with all the tasks mapped to either HW or SW is given by

$$v_{\text{exec}} = \begin{cases} v_{\text{rdtime}} + v_{\text{sw}} + v_{\text{wrtime}}, & \text{if } v_{\text{map}} = \text{SW} \\ v_{\text{rdtime}} + v_{\text{hw}} + v_{\text{wrtime}}, & \text{if } v_{\text{map}} = \text{HW} \end{cases}$$

where v_{map} denotes resource (HW or SW) to which task V has been mapped. v_{rdtime} and v_{wrtime} are the task read and write times, respectively. They are defined as $v_{\text{rdtime}} = \sum_{\forall e=(u,v)\in E} e_{\text{rdtime}}$ and $v_{\text{wrtime}} = \sum_{\forall e=(v,w)\in E} e_{\text{wrtime}}$, where $e_{\text{rdtime}}(e_{\text{wrtime}})$ are the read (write) associated with a data dependence. The read (write) time of task is given by the sum of the read (write) times of all its predecessor (successor)

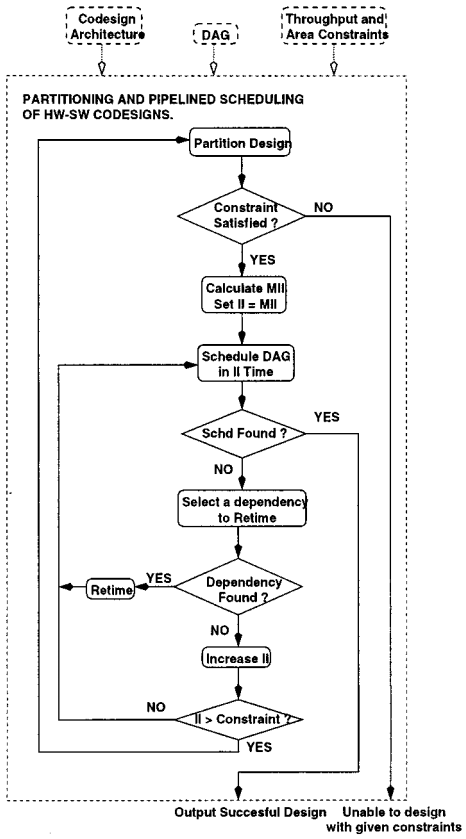

Fig. 4. Pipelined HW–SW implementation.

dependencies. The read and write time of a dependence $e = (u, v)$ is defined as follows:

$$e_{\text{rdtime}} = \begin{cases} e_{\text{data}} \cdot \text{swrd}, & u_{\text{map}} = v_{\text{map}} = \text{SW} \\ 0, & u_{\text{map}} = v_{\text{map}} = \text{HW} \\ e_{\text{data}} \cdot \text{shrd}, & \text{otherwise,} \end{cases}$$

$$e_{\text{wrtime}} = \begin{cases} e_{\text{data}} \cdot \text{swwr}, & u_{\text{map}} = v_{\text{map}} = \text{SW} \\ 0, & u_{\text{map}} = v_{\text{map}} = \text{HW} \\ e_{\text{data}} \cdot \text{shwr}, & \text{otherwise.} \end{cases}$$

We assume that the read time and write time of a dependence are zero when the predecessor and successor tasks have both been mapped to HW. In this case, the communication is assumed to take place through on-chip registers of the HW coprocessor and the associated communication overhead is included in the HW run times of the tasks.

The SW processor is a uniprocessing system. As a result, the minimum time required to execute all the tasks mapped to SW is given by the sum of execution times of SW tasks. This sum denotes the MII due to the SW tasks, MII_{SW}. The HW coprocessor supports concurrent execution of tasks. Hence, the minimum time required to execute the HW tasks is given by the maximum execution time of all tasks mapped to HW. This quantity denotes the MII due to HW tasks, MII_{HW}. The MII for the partitioned task graph is then given by maximum of MII_{SW} and MII_{HW}, that is $MII = \max(MII_{\text{SW}}, MII_{\text{HW}})$.

2) Pipelined HW–SW Implementation: The tool for partitioning and pipelined scheduling of HW–SW systems is shown in Fig. 4. The partitioner tries to obtain a HW–SW mapping whose MII and area are less than the specified constraints. Although, MII takes SW resource conflicts into account, it does not compensate for extra communication delays that might occur due to shared memory access conflicts. Therefore, it is necessary to evaluate the performance of the partition by generating the pipelined schedule.

The tool first tries to find a schedule of the DAG with MII as the time constraint. If it is unsuccessful, it selects a dependence to be retimed. Retiming transformation reduces the number of dependencies that constrain the scheduler and results in an equivalent task graph with tasks belonging to different iterations. The tool then schedules the new task graph to obtain the steady state of the pipelined implementation. The inner loop of scheduling and retiming continues till a successful schedule is obtained or all the dependencies have been retimed. In the latter case, we increase the initiation interval II and try scheduling again. We set the increment factor to the maximum of the following two values: one-time unit or 1% of MII. We exit the outer pipelined scheduling loop when the II becomes greater than the time constraint. The design flow then returns back to the partitioner to generate a new mapping. The outer loop of partitioning and pipelined scheduling continues till a successful schedule is obtained or the partitioner cannot generate a constraint satisfying mapping.

Let the set $S = \{s_1, s_2, s_3, \ldots, s_n\}$ denote the set of all possible mappings of the tasks to HW and SW. For a particular mapping $s_i \in S$, let $II(s_i)$ denote the achieved initiation interval of the corresponding pipelined schedule and let $\alpha(s_i)$ denote the total HW area of the mapping. Let $\tau_{\text{constraint}}$ and $\alpha_{\text{constraint}}$ denote the time and area constraints specified by the user. We have the following four cases:

1) *Both area and time constraint specified*: In this case, the tool searches for a mapping s_{soln} that satisfies the performance constraints, that is: $II(s_{\text{soln}}) \leq \tau_{\text{constraint}}$ and $\alpha(s_{\text{soln}}) \leq \alpha_{\text{constraint}}$. It returns the first solution that satisfies the constraints.
2) *Only area constraint*: In this case the tool searches for an optimal solution whose steady state executes in minimum time subject to the area constraint. Let A denote the set of mappings whose area is less than the area constraint. Then the tool searches for a mapping, $s_{\text{opt}} \in A$ such that: $II(s_{\text{opt}}) \leq II(s_i), \forall s_i \in A$.
3) *Only time constraint*: In this case, the tool searches for a solution that satisfies the time constraint but has the least HW area. Let T denote the set of mappings whose II is less than the time constraint. Then the tool searches for an optimal mapping, $s_{\text{opt}} \in T$ such that: $\alpha(s_{\text{opt}}) \leq \alpha(s_i), \forall s_i \in T$.
4) *No area and time constraint*: In this case, the tool searches the design space for an optimal solution $s_{\text{opt}} \in S$ such that: $II(s_{\text{opt}}) \leq II(s_i), \forall s_i \in S$.

In the worst case, our approach will exhaustively map the tasks to HW and SW, and execute with exponential time. Due to a good initial solution and tight search space bounding, we are able to obtain optimal partitions for graphs having up to 30 nodes in a reasonable amount of time (30 mins). Since the application is modeled at a coarse level of granularity, 30 tasks

are enough to model many applications. We provide a time out option for large graphs. The tool then returns the best solution that it obtains before the time out. As the experimental results will show later, the initial solution is on an average within 6.3% deviation from the optimal. This implies that the partitions obtained by the time out option will also have good performance characteristics.

In the explanation given above (and in rest of the paper), we assume the optimal solution to be the one with respect to the partitioner. It is different from the global optimum that is with respect to both the partitioner and pipelined scheduler. The experimental results will show that although we use a heuristic scheduler the solution obtained by our tool is on an average within 4.2% deviation of the global optimum.

B. HW–SW Partitioner

The codesign partitioner uses a branch and bound approach with backtracking to explore the design space. The algorithm traverses a binary search tree that includes the entire design space. At each level of the search tree, the algorithm selects an unmapped task and decides the mapping of the task. The branch and bound approach is characterized by the strategy used to generate the initial solution, the technique used to select a task and decide the mapping of the task, and the techniques used to limit the search space. In this section, we will discuss all these strategies.

1) Generation of Initial Solution: The HW–SW partitioner tries to minimize the MII of the solution subject to the area constraints. Since $MII_{\rm SW}$ is given by the sum of the SW tasks, it is the dominant quantity in determining the MII. The MII of a partitioned design can be minimized by mapping fewer tasks to SW. However, this would lead to an increase in the HW area and perhaps a violation of the area constraint. The partitioner tries to balance these two conflicting objectives by initially mapping the tasks to HW and SW such that the sum of the execution times of tasks mapped to HW and SW is balanced. This ensures that $MII_{\rm SW}$ is not too large and the area of the coprocessor is also small.

The initial solution maps the tasks based on their individual characteristics and user specified area constraint. The mapping of a task is influenced by the following three properties 1) its run time on the SW processor versus its run time on the HW coprocessor; 2) its area in the HW coprocessor; and 3) its estimated communication time in SW versus its estimated communication time in HW. We capture these properties by calculating the speed up of the task, area factor of the task, suitability of the task, and communication ratio of the task as explained below.

a) Speed up of a task: We Define the *speed ratio* of a task $v_{\rm sr}$ as follows:

$$v_{\rm sr} = \begin{cases} \frac{v_{\rm sw}-v_{\rm hw}}{v_{\rm sw}}, & \text{if } (v_{\rm sw}-v_{\rm hw}) \geq 0 \\ \frac{v_{\rm sw}-v_{\rm hw}}{v_{\rm hw}}, & \text{if } (v_{\rm sw}-v_{\rm hw}) < 0. \end{cases}$$

The speed ratio varies between -1 and 1. It is greater (smaller) than zero for a task whose run time on the SW processor is greater than (less than) its run time on the HW coprocessor. We scale the speed ratio from 0 to 1 and define the *speed up* of a task v as follows: $v_{\rm speedup} = v_{\rm sr} - S_{\rm min}/S_{\rm max} - S_{\rm min}$ where $S_{\rm min}(S_{\rm max})$ is the minimum (maximum) speed ratio over all tasks.

b) Area factor of a task: Let $A_{\rm sum}$ denote the summation of areas of all the tasks $v \in V$. Let $A_{\rm max}(A_{\rm min})$ denote the maximum (minimum) area over all tasks. We define the *area ratio* of a task $v_{\rm ar}$ as follows:

$$v_{\rm ar} = \begin{cases} \frac{A_{\rm max}-v_{\rm area},}{A_{\rm max}-A_{\rm min},} & \text{if } v_{\rm area} \leq A_{\rm constraint} \\ 0, & \text{if } v_{\rm area} > A_{\rm constraint}. \end{cases}$$

The area ratio ranges from 0 to 1. The area ratio of task is closer to 0 (1) if its area is nearer to $A_{\rm max}(A_{\rm min})$. We use the area ratio to define the *area factor* of a task v as follows:

$$v_{\rm area\_factor} = (v_{\rm ar} \times (1 - A'_{\rm constraint})) + A'_{\rm constraint}.$$

Where

$$A'_{\rm constraint} = \min\left(1, \frac{A_{\rm constraint}}{A_{\rm sum}}\right).$$

The area factor scales the area ratio to range from $(A_{\rm constraint}/A_{\rm sum})$ to 1. In case of tight area constraint $(A_{\rm constraint} \ll A_{\rm sum})$ the area factor ranges from a small value to 1. In case of a loose area constraint, $(A_{\rm constraint} \approx A_{\rm sum} - \delta)$ the variation in area factor is small. The objective is to be able to distinguish between tasks in terms of their respective area depending upon the area constraint. When the area constraint is very tight, we see a larger difference in area factors of two tasks as opposed to the case when the area constraint is very loose.

c) Suitability of a task: The suitability of a task $v_{\rm suit}$ to be assigned to HW is given by the product of its speed up and area factor, that is $v_{\rm suit} = v_{\rm speedup} \times v_{\rm area\_factor}$. The influence of the speed up and area factor of the task on its suitability depends on the area constraint. In the case of a tight area constraint, the suitability of a task is influenced by both its area factor and speed up. When the area constraint is not tight, the area factors of the tasks are close to each other. As a result when we distinguish two tasks based on their suitabilities our decision is influenced more by their individual speed up values than their area factors. We scale the suitabilities of tasks from 0 to 1 and define suitability factor $v_{\rm sf}$ of a task as follows: $v_{\rm sf} = (v_{\rm suit} - St_{\rm min}/St_{\rm max} - St_{\rm min})$, where $St_{\rm min}(St_{\rm max})$ is the minimum (maximum) suitability over all tasks $v \in V$. We will use suitability factor as the probability of the task to be assigned to HW.

d) Communication ratio of a task: We estimate the initial communication times of the tasks based on their suitability factor. The initial read time and write time of a task v when it is mapped to SW is calculated as follows:

$$v_{\rm init\_SW\_rdtime} = \sum_{\forall e=(u,v)\in E} e_{\rm data} \cdot((1-u_{\rm sf})\cdot {\rm swrd} + u_{\rm sf}\cdot {\rm shrd})$$

and

$$v_{\rm init\_SW\_wrtime} = \sum_{\forall e=(v,w)\in E} e_{\rm data} \cdot((1-w_{\rm sf})\cdot {\rm swwr} + w_{\rm sf}\cdot {\rm shwr}).$$

Since we assume that the communication time when two HW tasks communicate through the on-chip registers is zero, the ini-

tial read time and write time when task v is mapped to HW are given by

$$v_{\text{init\_HW\_rdtime}} = \sum_{\forall E=(u,v)\in E} e_{\text{data}} \cdot((1-U_{\text{sf}})\cdot \text{shrd}$$

and

$$v_{\text{init\_hw\_wrtime}} = \sum_{\forall E=(v,w)\in E} E_{\text{data}} \cdot((1-W_{\text{sf}})\cdot \text{shwr} \quad .$$

The communication ratio of a task v is then given by

$$v_{\text{comm\_ratio}} = \frac{v_{\text{init\_sw\_rdtime}} + v_{\text{init\_sw\_wrtime}}}{v_{\text{init\_hw\_rdtime}} + v_{\text{init\_hw\_wrtime}}}.$$

The communication ratio of a task is greater (less) than one if the estimated communication time with task in SW is greater (less) than the estimated communication time with task in HW.

e) Final Suitability of a Task: The suitability of a task as calculated above does not reflect the communication overheads. We modify the suitability of the task by multiplying it by the communication ratio to obtain the final suitability as follows: $v_{\text{final\_suit}} = v_{\text{suit}} \times v_{\text{comm\_ratio}}$.

f) Initial Solution: After we have obtained the final suitability of each task we sort the tasks in the descending order of their suitabilities. Then the initial solution of the branch and bound algorithm is obtained by choosing one task alternatively from the front and back of the sorted list and mapping them to HW and SW, respectively. A task near the front (back) of the sorted list has a higher (lower) suitability and it is bound to HW (SW). During the generation of the initial solution we try to balance the following two quantities: $\sigma_{\text{hw}} \approx \sigma_{\text{sw}}$, where σ_{hw} and σ_{sw} are the running sums of estimated execution times of tasks bound to HW and SW resources, respectively. We try to balance to these two quantities subject to the area constraint on the HW coprocessor. We maintain α_{hw} that gives the running sum of the areas of the task mapped To HW. We compare α_{hw} with $\alpha_{\text{constraint}}$ to ensure that the area constraint is satisfied. If the user does not specify an area constraint we assume the value of $\alpha_{\text{constraint}}$ to be a very large number. As explained earlier we try to balance σ_{hw} and σ_{sw} with an aim to achieve the conflicting goals of minimizing MII and the area associated with HW tasks.

During the generation of the initial solution when a task v is mapped to a resource, it is possible that some of the neighboring tasks of v have not yet been mapped. In such a case, we estimate the read and write times of the task as the minimum value possible. Consider a dependence $e=(u,v)$ where u is unmapped and v is the task to be mapped. Then the minimum read time is given by

$$e_{\text{min}_{\text{rdtime}}} = \begin{cases} e_{\text{data}} \cdot \min(\text{swrd}, \text{shrd}) & v_{\text{map}} = \text{SW} \\ 0 & v_{\text{map}} = \text{HW} \end{cases}$$

The minimum write time is similarly defined. We use minimum communication times to estimate the execution time of the task. At a later stage, when the neighboring task has been mapped we update the communication times to their correct values. These changes are also reflected in σ_{hw} and σ_{sw}.

```
Algorithm Initial Solution
Input : DAG(V, E)
Output : level[] and binding[] arrays
begin
   ∀v ∈ V calculate(final_suit(v))
   suit_array[] = tasks sorted in descending order of their final_suit.
   size = |V|, σ_hw = σ_sw = i = k = 0, j = size − 1
   while (k < size)
      while (k < size AND σ_hw ≤ σ_sw AND
             α_hw + suit_array[i]_area ≤ α_constraint)
         task = suit_array[i]
         task_select[k] = task
         init_map[k] = hw
         update_times(σ_hw, σ_sw, task)
         k = k + 1, i = i + 1
      endwhile
      while ((k < size AND σ_sw ≤ σ_hw) OR
             (k < size AND α_hw + suit_array[i]_area > α_constraint)
         task = suit_array[j]
         task_select[k] = task
         init_map[k] = sw
         update_times(σ_hw, σ_sw, task)
         k = k + 1, j = j - 1
      endwhile
   endwhile
end
```

Fig. 5. Algorithm for initial solution.

The algorithm to generate the initial solution is shown in Fig. 5. The outer "while" loop continues until all the tasks have been mapped. The two inner "while" loops try to balance σ_{hw} and σ_{sw} subject to the area constraint. The array $\text{task\_select}[]$ stores the task to be mapped at level *K* and the initial mapping of a task is stored in array $\text{init\_map}[]$.

2) Selecting a Task and Deciding its Mapping: The search procedure for the branch and bound algorithm resembles a binary tree (see Fig. 6). At a level $k(0 \le k \le |V|)$ in the search tree we make a decision about the mapping of a particular task specified by the array $\text{task\_select}[k]$. This array is used during the entire algorithm to select the task to be mapped. The initial mapping of the task at the level k is given by the array $\text{init\_map}[k]$. During the search process we maintain the variables σ_{hw} and σ_{sw}. After we have selected the task from the $\text{task\_select}[]$ array, the mapping of the task is done with an objective of balancing σ_{hw} and σ_{sw}.

Sorting the tasks according to their suitabilities and then mapping them in the above fashion to generate the solution has two important effects. First the initial mapping is a fairly good solution and it helps in limiting the search space of the algorithm. When the branch and bound algorithm begins its search (see Fig. 6), it first exhaustively maps the tasks that are at the higher levels of the search tree. In comparison to tasks at lower levels (tasks t1, t5), the tasks at higher level (task t3) are not inclined toward either HW or SW implementation. Hence, the second important effect is that the algorithm tries to obtain a solution by keeping the mapping of tasks that are strongly inclined toward either HW or SW implementation fixed and changing the mapping of tasks that are not inclined toward either implementa-

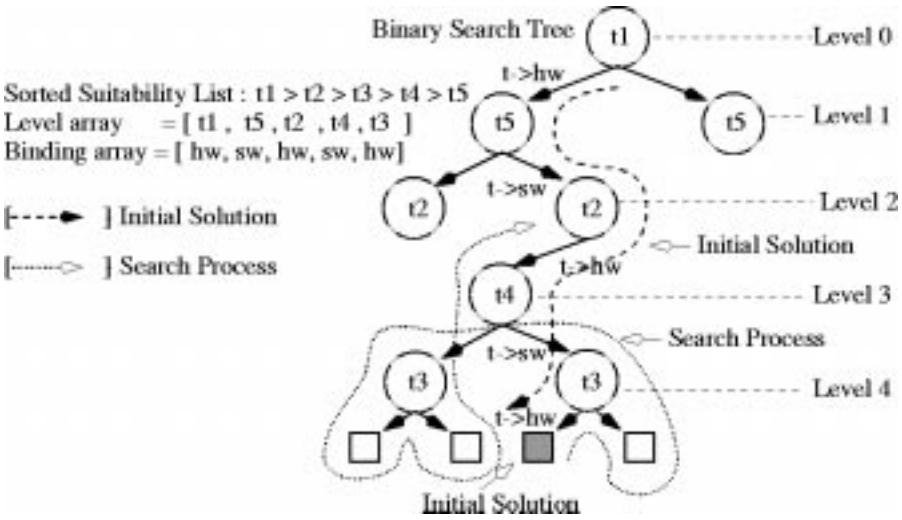

Fig. 6. Initial solution and search process.

tion. Such a search strategy coupled with a good initial solution leads to faster execution times of the algorithm.

3) Techniques for Limiting the Search Space: We first explain the terms that are used in the discussion. At any time during the search process $s_{\min}$ denotes the best solution found so far. Initially $s_{\min} = s_{\text{init}}$. At a particular level *K* in the search tree we map the task specified by the $\text{task\_select}[]$ array. During the search process we maintain four variables α_{hw}, μ_{hw}, σ_{hw} and σ_{sw}. At a particular level *K* in the search tree.

- α_{hw} gives the estimated area of all tasks mapped to HW from levels 0 to $(K-1)$;
- μ_{hw} gives the maximum estimated execution time of all tasks mapped to HW from levels 0 to $(K-1)$;
- σ_{hw} gives the sum of estimated execution times of all tasks mapped to HW from levels 0 to $(K-1)$;
- σ_{sw} gives the sum of estimated execution times of all tasks mapped to SW from levels 0 to $(K-1)$.

Let us assume that we are at level *K*, we have mapped the task at level *K* to both HW and SW and we are about to backtrack. We can then associate two terms $\mu_{\min\_\text{hw}}$ and $\sigma_{\min\_\text{hw}}$ with the task at level *K* defined as follows:

- $\mu_{\min\_\text{hw}}$ gives the maximum of the exact HW execution time of all tasks mapped to HW from levels 0 to $(K-1)$ in the solution $s_{\min}$;
- $\sigma_{\min\_\text{sw}}$ gives the sum of the exact execution times of the tasks that have been mapped to SW from levels 0 to $(K-1)$ in the solution $s_{\min}$.

The exact execution time of a task v, $v_{\text{exact\_exec}}$ differs from v_{exec} that we have defined earlier. $v_{\text{exact\_exec}}$ is found from the pipelined schedule of $s_{\min}$ and it takes the extra communication delays due to shared memory access conflicts into account ($v_{\text{exact\_exec}} \geq v_{\text{exec}}$). The initial values of $\mu_{\min\_\text{hw}}$ and $\sigma_{\min\_\text{sw}}$ are infinity.

Consider the case when no constraints have been specified and we are trying to obtain a solution with minimum II. Let us assume that we are at level *K* and map the task to SW. Before we proceed to the level $(K+1)$ we check if the following two conditions are satisfied:

$$\max(\sigma_{\text{sw}} + v_{\text{exec\_sw}}, \mu_{\text{hw}}) \leq II(s_{\min}) \tag{1}$$

$$(\sigma_{\text{sw}} + v_{\text{exec\_sw}} \leq \sigma_{\min\_\text{sw}}) \text{ OR } (\mu_{\text{hw}} \leq \mu_{\min\_\text{hw}}) \tag{2}$$

where $v_{\text{exec\_sw}}$ is the estimated SW execution time of the task. If the two conditions are not satisfied we backtrack and change the previous decision. We can similarly define two more conditions when the task is mapped to HW

$$\max(\max(\mu_{\text{hw}}, v_{\text{exec\_hw}}), \sigma_{\text{sw}}) \leq II(s_{\min}) \tag{3}$$

$$(\max(\mu_{\text{hw}}, v_{\text{exec\_hw}}) \leq \mu_{\min\_\text{hw}}) \text{ OR} (\sigma_{\text{sw}} \leq \sigma_{\min\_\text{sw}}) \tag{4}$$

where $v_{\text{exec\_hw}}$ is the estimated HW execution time of the task. In the presence of an area constraint we use the following condition along with the (1),(2),(3), and (4) to limit the search:

$$\text{Sum}(\alpha_{\text{hw}}, v_{\text{area}}) \leq \alpha_{\text{constraint}}. \tag{5}$$

In the case that only a time constraint is specified we use the following three conditions to limit the search process:

$$\text{Sum}(\alpha_{\text{hw}}, v_{\text{area}}) \leq \alpha(s_{\min}) \tag{6}$$

$$\max(\sigma_{\text{sw}} + v_{\text{exec\_sw}}, \mu_{\text{hw}}) \leq \tau_{\text{constraint}} \tag{7}$$

$$\max(\max(\mu_{\text{hw}}, v_{\text{exec\_hw}}), \sigma_{\text{sw}}) \leq \tau_{\text{constraint}}. \tag{8}$$

When both area and time constraints are specified we use conditions 5, 7, and 8 to limit the search process.

4) HW–SW Partitioning Algorithm: The partitioning algorithm is shown in Fig. 7. In the algorithm "k" denotes the level of the binary search tree. " *first*" is a boolean variable that is true when we reach a particular level for the first time and false otherwise. $b1$ and $b2$ specify the mapping (HW or SW) of the task. Initially, all the four parameters in the call are zero. When k is equal to $|V|$ all the tasks are mapped to HW or SW and we evaluate the candidate partition. When we traverse the graph for the first time we generate the initial solution according to the $\text{init\_map}[]$ array. Otherwise, we try to balance the two sums, σ_{hw} and σ_{sw}. The function *map()* maps a task to HW or SW and updates $\sigma_{\text{hw}}, \sigma_{\text{sw}}$, and α_{hw}. The boolean function *check()* as the name suggests checks if the conditions are satisfied.

```
Algorithm Partition (k,σ_hw, σ_sw, α_hw)
begin
 if (k = |V|) /* Candidate Partition */
    II = pipeline_schedule()
    if (constraint_satisfied(II, α_hw)) return(1)
     else return(0) endif
 endif
 if (init_map[k] = hw OR (NOT(first) AND σ_hw < σ_sw))
    b1 = hw, b2 = sw /* First map task to HW then SW */
 else
    b1 = sw, b2 = hw /* First map task to SW then HW */
 endif
 task = task_select[k]
 map(task,b1)
 if (check() = TRUE) S = Partition(k+1,σ_hw, σ_sw, α_hw)
 if (S = 1) return(1)
 map(task,b2)
 if (check() = TRUE) S = Partition(k+1,σ_hw, σ_sw, α_hw)
 if (S = 1) return(1)
 return(0)
end
```

Fig. 7. Branch and bound based HW–SW partitioner.

g) Time Complexity: Since each task can be mapped to either HW or SW the time complexity of the partitioner is $O(2^{|V|})$.

C. Scheduler for Pipelined HW–SW Implementation

We evaluate the performance of a particular design alternative by obtaining a pipelined schedule. 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 $0 \leq S(v) \leq II$ [48]. For a dependence $e = (u, v)$, the schedule time of u and v must honor the data dependence, that is

$$S(v) + \delta(e) \times II \geq S(u) + u_{\text{exec}} \Rightarrow S(v) \geq S(u) + u_{\text{exec}} - \delta(e) \times II.$$

The pipeline scheduler takes resource conflicts and communication delays due to SW processor and shared memory into account. We use a list based schedule [50] and retiming transformation in an iterative manner to obtain a pipelined schedule. We calculate the MII and try scheduling the DAG for MII. However, due to schedule constraining dependencies we may not be able to schedule the DAG in MII. If we cannot accomplish this, we retime the DAG and try again. Retiming transformation reduces the number of schedule constraining dependencies. We discuss the retiming transformation in the next section.

1) List Based Scheduler: The list based scheduler maintains a schedule table with four columns for SW local memory, SW processor, shared memory, and HW coprocessor. It also maintains a ready list of tasks that are ready to be scheduled. The list scheduler selects a task from the ready list based on maximum *urgency*. The urgency of a task is given by $v_{\text{urgency}} = v_{\text{exec}} + \max_{\forall(v,w)\in E}(w_{\text{urgency}})$. This is a well known heuristic that has been widely use in literature [35]. The start time of the selected task on the mapped resource is based on the earliest time that satisfies the data dependencies (based on the equation given above) and resource constraints.

a) Time complexity: The list based scheduling algorithm operates in a loop. In each iteration of the loop, the algorithm schedules one task. In order to schedule the task, the algorithm traverses the schedule table in a top down manner. The worst case complexity of the traversal is $O(|V|+|E|)$. Hence, the time complexity of the list based scheduler is $O(|V|(|V|+|E|)$.

D. RECOD: Retiming Heuristic

We apply retiming transformation when we cannot schedule the DAG in the given initiation interval, II. Retiming transformation reduces the number of data dependencies that constrain the scheduler by increasing their dependence distance δ. However to produce an equivalent task graph it is also necessary to increase the iteration indices of the tasks. Two task graphs $DAG = G(V, E)$ and $DDG' = G(V', E')$ (obtained after retiming) are equivalent if, $\forall e = (u, v) \in E, E'$, the following equation holds $\lambda(v) - \lambda(u) + \delta(e) = \lambda'(v) - \lambda'(u) + \delta'(e)$. The retimed task graph consists of tasks belonging to different iterations of the original loop. Hence, retiming transformation results in a pipelined task graph. Our Retiming heuristic is oriented toward HW–SW CODdesigns, therefore we call it RECOD. RECOD optimizes the initiation interval, pipeline buffers, and number of pipeline stages of a pipelined HW–SW codesign. Before we present our retiming heuristic, we discuss the factors that influence the performance of the pipelined design.

1) Schedule Constraining Dependencies: A dependence $e = (u, v)$ with $\delta(e) = 0$ 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. Hence, a dependence with $\delta(e) = 0$ constrains the schedule. Such a dependence is called a *intra-loop dependence (ILD)*. We assume that all the tasks belonging to one iteration of the steady state complete their execution before any task belonging to the next iteration starts its execution. We also assume that the task execution in HW and on the SW processor is sequential (nonpipelined). Then a dependence $e = (u, v)$ with $\delta(e) > 0$ does not constrain the pipelined schedule since for all values of $S(u)$ and $S(v)$ the data dependence is satisfied. Such a dependence is called a *loop-carried dependence (LCD)*. *LCDs* represent data dependence between tasks belonging to different iterations of the steady state. The two assumptions stated above ensure that the*LCDs* do not constrain the scheduler. Therefore, the set of schedule constraining dependencies, E^S is given by $E^S = \{e = (u, v) \in E | \delta(e) = 0\}$.

A path $p = \{e_1, \ldots, e_n\}$ in the DAG is called a constraining path, if $\forall e \in p$, $e \in E^S$. The length of p is given by $\text{Length(p)} = (\text{w}_{\text{exec}} + \sum_{(\text{u,v})\in \text{p}} \text{u}_{\text{exec}})$, where w_{exec} is the execution time of the tail task of p. A critical path *CP* in the DAG is a constraining path p, such that for any other constraining path $p' \subseteq E$, $\text{Length(p)} \geq \text{Length(p}')$. The length of the critical path is called the critical path time, CPT of the DAG. For a feasible pipelined schedule of the DAG with initiation interval II, $CPT \leq II$. Therefore, retiming transformation should try to reduce the number of schedule

constraining dependencies that belong to a longer constraining path.

2) Communication Memory Estimation: 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 that is given by $\text{Mem}_{LCD} = \sum_{e \in LCD} \delta(e) \times e_{\text{data}}$. Mem_{LCD} is the communication memory required by the pipeline buffers. The communication 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 communication memory requirement of a pipelined design, MemReq is then given by $\text{MemReq} = \max(\text{Mem}_{LCD}, \text{Mem}_{\text{exec}})$. As can be seen by the above discussion Mem_{LCD} is a lower bound on the memory requirement of a pipelined schedule. During retiming we convert a schedule constraining dependence (*ILD*) in to a LCD leading to an increase in communication memory requirement. Therefore, during retiming we should try to minimize the memory required for pipeline buffers.

We can minimize the increase in memory requirements due to pipelining by using good heuristics to select the dependence to be retimed. But this is not enough. In order to produce an equivalent DAG other dependencies need to be retimed. The increase in memory requirement due to these dependencies should also be minimized. Hence, RECOD does retiming in two steps. In the first step it heuristically selects a dependence to be retimed. In a DAG there might exist a number of sets of dependencies that could be retimed to obtain an equivalent DAG. In step 2 we select the set of dependencies that on retiming result in the least increase in shared memory requirement.

3) RECOD Step 1: Heuristic to Select a Dependence for Retiming Transformation: The priority of a dependence to be retimed depends on its following four properties in decreasing order:

1) *Dependence is an* ILD: The primary objective of RECOD is to reduce scheduling constraints in the DAG; and give the scheduler greater freedom in scheduling tasks on the resources. Since only *ILDs* constrain the scheduler the dependence to be retimed should be an *ILD*.
2) *Dependence whose two tasks are not mapped to SW processor*: The main objective of the retiming heuristic is to reduce scheduling constraints in the graph. Increasing the distance of a dependence between tasks mapped to the SW resource does not necessarily help the scheduler. Basically the two SW tasks will be scheduled one after the other. On the other hand retiming a dependence between tasks mapped to HW coprocessor definitely gives more freedom to the scheduler. Similarly, retiming a dependence between tasks mapped to heterogeneous processors also gives more freedom to the scheduler.
3) *Dependence whose predecessor task belongs to a longer constraining path*: As discussed in the previous section the constraining paths limit the II of a pipeline schedule. Retiming a dependence whose predecessor task belongs to a longer constraining path helps in obtaining a pipelined schedule with smaller II.
4) *Dependence representing the least number of data items transferred*: A secondary objective of retiming transformation is to minimize the increase in pipeline memory requirement of the DAG. Increasing the distance of a dependence with more data items definitely results in a larger increase in memory requirement. Hence, we select a dependence that represents the transfer fewer data items.

We use property 1 to select dependencies to be retimed and use properties 2 , 3, and 4 (in that order) to break ties.

4) RECOD Step 2: Partitioning to Minimize Increase in MEM_{LCD}: In step 2 we select the set of dependencies that give us the least increase in memory required for pipeline buffers (MEM_{LCD}). Given a dependence $e = (u, v)$ (selected in step 1) to be retimed we define the following three sets with respect to u:

$$\begin{aligned} P =& \{w \in V | \text{there is a path from } w \text{ to } u\} \cup \{u\} \\ S =& \{w \in V | \text{there is a path from } u \text{ to } w\} \\ R =& V - \{P \cup S\}. \end{aligned}$$

Fig. 8 gives an illustration of the three sets. We can retime the dependence $e = (u, v)$ by retiming all dependences belonging to *cutset c1* as follows: $\forall u \in P, \lambda(u) = \lambda(u) + 1$ and $\forall (u, v) \in E, u \in P, v \notin P, \delta(u, v) = \delta(u, v) + 1$. Another way to retime dependence $e = (u, v)$ to retime the dependencies belonging to *cutset c2* as follows: $\forall u \in \{P \cup R\}, \lambda(u) = \lambda(u) + 1$ and $\forall (u, v) \in E, u \notin S, v \in S, \delta(u, v) = \delta(u, v) + 1$. However it is possible that neither cutset*c1* nor *c2* result in a minimum increase in MEM_{LCD}. We could obtain another *cutset c3* (see Fig. 8) by partitioning the set R into P and S, so that the memory increase is minimized. The cost function being minimized is defined as follows. For a cut $c_i = \{e^1, e^2, \ldots, e^n\}$, the cutsize cost is given by : $\text{Cost} = \sum_{j=1}^{n} e^j_{\text{data}}$. where e^j_{data} is the number of data items transferred across the dependence e_j. In the cost function the sum gives us the extra memory required by the *LCDs* after retiming. During partitioning we ensure that if a task u is in partition $P(S)$ then all its predecessors (successors) are also in partition $P(S)$. After partitioning set R in to sets P and S we do retiming using the following two equations:

$$\begin{aligned} \forall u \in P, \lambda(u) =& \lambda(u) + 1 \\ \forall e =& (u, v) \in E, u \in P, v \notin P, \delta(u, v) \\ =& \delta(u, v) + 1. \end{aligned}$$

5) RECOD: Algorithm: The algorithm to do retiming transformation is shown in Fig. 9. The function $\text{heuristic\_select}()$ selects a dependence to be retimed (see RECOD step 1). The function*partition()* as the name suggests partitions R between P and S (see RECOD step 2). The embedded *for-loops* apply the retiming transformation.

a) Time Complexity: The time complexity of the retiming heuristic is determined by the partitioning algorithm used in recod step 2. We use a simulated annealing based algorithm with a start temperature of 100, final temperature of 0.001 and a decrement factor of 0.9. At a particular temperature the algorithm makes atmost $|V|$ moves. During each move the algorithm explores the immediate neighbors of the selected task. Hence, the time complexity of the retiming algorithm is $O(|V| + |E|)$.

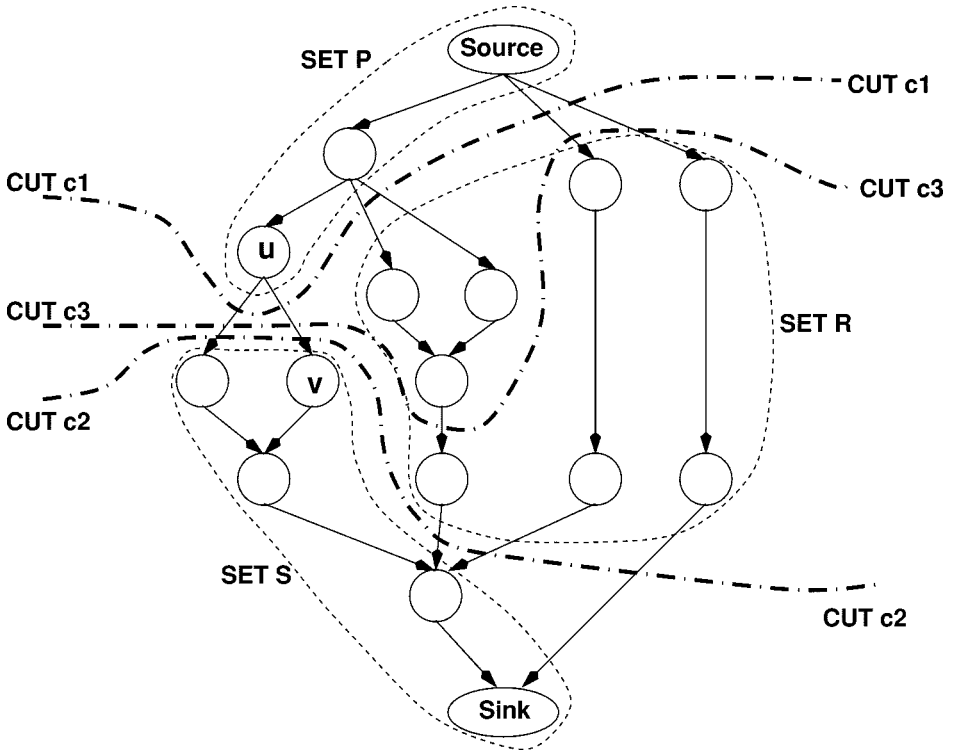

Fig. 8. P, S, and R sets for dependence (u, v).

```
Algorithm RECOD(DAG)
begin
    edge_(u,v) = heuristic_select(DAG)
    if (edge_(u,v) = 0) then return(failure) endif
    S = {v ∈ V | there is a path from u to v}
    P = {v ∈ V | there is a path from v to u} ∪ {u}
    R = V − {S ∪ P}
    partition(R,P,S)
    for each u ∈ P
        λ(u) = λ(u) + 1 endif
        for all (u, w) ∈ E, w ∉ P, δ(u, w) = δ(u, w) + 1 endfor
    endfor
    return(DAG)
end
```

Fig. 9. RECOD: Algorithm.

IV. Results

In this section we evaluate our approach to designing pipelined HW-SW systems. We first evaluate the performance our retiming heuristic by comparing it with another existing heuristic. We then evaluate the the run time of the algorithm, quality of the initial solution and quality of the final solution generated by our design by conducting experiments with synthetically generated task graph. We finally conduct a case study of the JPEG image compression algorithm and establish the effectiveness of our approach with realistic task graphs.

In two of the experimental studies that will be presented in the following subsections we use synthetically generated task graphs. We conduct experiments with random task graphs due to lack of established benchmarks in the codesign area. The task graphs were generated randomly. They differed in the number of tasks, depths, number of dependences, number of data items transferred across each dependence, HW run times, area and SW run times.

A. RECOD Versus UNRET

We compare the performance of RECOD with UNRET [48]. UNRET retimes the head dependence or tail dependence of a maximum positive path (*MPP*). *MPP* is similar to our critical path *CP*. We conducted the experiment with synthetically generated task graphs. We randomly mapped the tasks to HW and SW and compared the pipelined schedules generated by the two heuristics. The results of the study are shown in Table I. Columns two and three indicate the number of tasks and the depth of the task graph. Column four gives the minimum initiation interval of the task graph. Columns five to eight and nine to twelve indicate the achieved initiation interval, number of pipeline stages, amount of memory required for LCD and time required on a SPARC 5 machine by RECOD and UNRET respectively. Columns thirteen to fifteen indicate the percentage reduction in II, pipeline stages, and MEM_{LCD} due to RECOD in comparison with UNRET. Column 16 gives the percentage increase in time required for the RECOD solution over UNRET solution.

A few interesting characteristics of the retiming heuristic can be observed from the results. The number of pipeline stages and MEM_{LCD} for a design generated by RECOD are always less than a design generated by *UNRET*. In particular for task graphs with larger depth (rows 6, 8, 10, 12) RECOD gives far fewer number of pipeline stages then UNRET. Since the task graphs have a larger depth, RECOD step 2 explores a very limited portion of the task graph. Hence, the improvement is predominantly due to RECOD step 1. In step 1, RECOD selects

TABLE I
COMPARISON OF RECOD WITH UNRET

	DAG		MII	RECOD				UNRET				% Decr			% Incr
	$\|V\|$	D	(ms)	II(ms)	S	M_L	T(s)	II(ms)	S	M_L	T(s)	II	S	M_L	T
1	10	6	300	315	4	93	0.4	315	5	105	0.3	0	20	11.4	25
2	10	9	310	328	5	99	0.3	328	7	104	0.3	0	28	4.8	0
3	15	7	770	784	5	144	1	784	6	149	0.8	0	16.7	3.3	20
4	15	12	860	880	6	94	0.8	898	9	124	0.7	2	33.3	24.2	12.5
5	20	7	870	903	5	261	1.9	923	6	375	1.2	2	16.7	30.4	36.8
6	20	14	1010	1038	6	205	1.6	1044	10	225	1.1	1	40	8.8	31.2
7	25	4	2630	2745	4	894	2.7	2850	4	2767	1.5	2.8	0	67.7	44.4
8	25	18	2835	2940	6	783	2.4	3015	12	814	1.3	2.4	50	3.9	45.8
9	30	5	2440	2562	4	741	4.7	2634	5	1732	2.9	2.7	20	57.2	38.3
10	30	20	4230	4620	7	633	4.2	4765	11	684	2.6	3	36	7.4	38.1
11	50	6	5630	5905	5	1984	7.1	5945	6	7094	4.2	0.6	16.7	72	40.8
12	50	24	6410	6450	9	542	5.9	6530	15	565	3.6	1.8	40	4.1	38.98

a dependence that on retiming would give maximum freedom to the scheduler. As a result, we apply retiming transformation a lesser number of times resulting in fewer-pipeline stages. Fewer iterations of the retiming transformation also result in lesser amount of pipeline memory required by *LCDs*. On the other hand, for task graphs with smaller depth (rows 5, 7, 9, 11) RECOD gives fewer pipeline memory than UNRET. This is because the task graphs are very wide and RECOD step 2 can explore a wide-search space and give designs that require fewer pipeline buffers. The initiation interval of the designs generated by RECOD and UNRET are almost similar. Both the heuristics give designs whose $II > MII$ and this is due to communication conflicts on the shared memory. RECOD gives slightly better II than UNRET because it produces a design with fewer pipeline stages and hence less concurrency. As a result, there are fewer conflicts on the shared memory. The drawback of RECOD is the increased time required (over 44% more) for solution generation as compared to UNRET. This overhead is essentially due to RECOD step 2.

B. Evaluation of Overall Approach

In this section, we conduct four experiments aimed at evaluating the run time of the tool, establishing the quality of the initial solution and evaluating the overall approach. Due to a lack of accepted benchmarks, we conduct the experiment with synthetically generated task graphs. We generated random task graphs with eight to thirty nodes in increments of two. At each node size we generated five task graphs by varying the depth, connectivity, number of data items, transferred across the dependence and task run times, respectively. We generated a total of 60 task graphs.

We evaluated the run time of the tool by invoking it for each of the task graphs and obtaining an optimal solution under no constraints (see Fig. 10). In the figure we have plotted the average execution time for each node size. The maximum-run time of the tool was 30 min for a graph with 30 nodes. A low run time was possible because of the good quality of the initial solution and the search space bounding techniques. In the second study, we analyzed the quality of the initial partition generated by the tool. We calculated the percentage deviation in the initiation interval of the initial partition from the final partition for all the

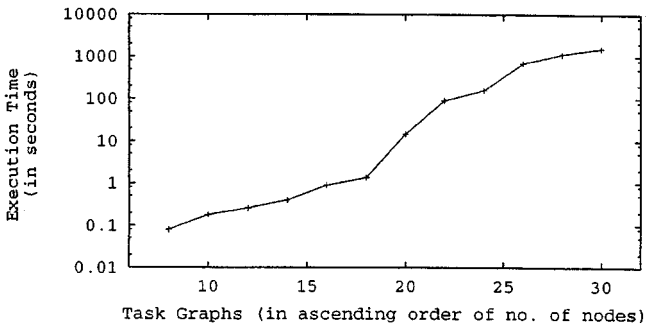

Fig. 10. Execution time of algorithm.

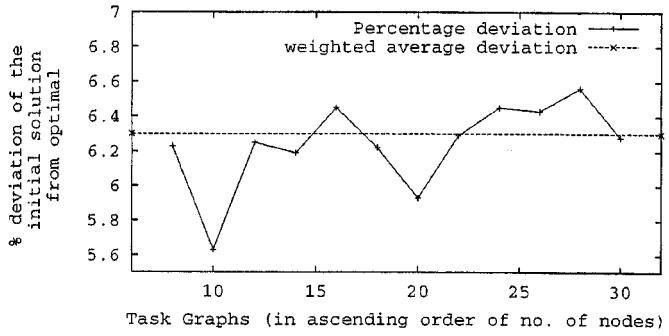

Fig. 11. Percentage deviation of initial solution from final solution.

graphs (see Fig. 11). The weighted average percentage deviation was found to be 6.3%. In the third study we plotted the number of solutions generated by the partitioner before it found the final solution (see Fig. 12). The weighted average number of solutions generated by the partitioner before the final solution was 10.2. Studies two and three demonstrate the superior quality of the initial solution. Therefore, for large graphs we can use a timeout option for the codesign partitioner and still have a high degree of confidence in the quality of the design.

Finally, in the fourth study we compared the solutions obtained by our tool against the minimum MII that was obtained by the partitioner during design space exploration (see Fig. 13). The minimum MII is a lower bound on the global-optimum solution for a particular task graph. The final solution is on an

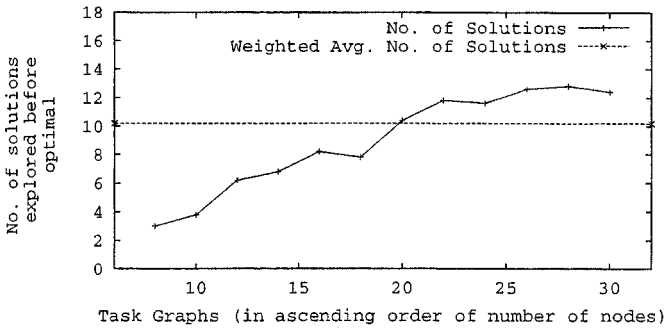

Fig. 12. Number of solutions explored before final solution.

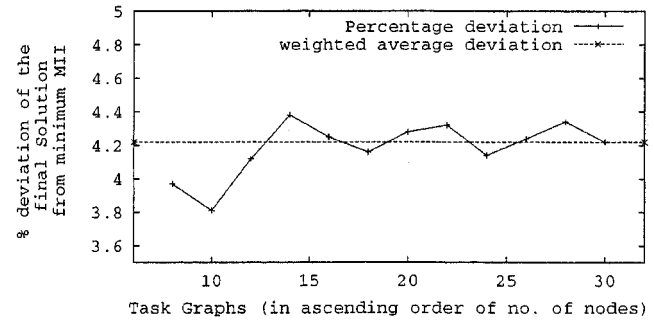

Fig. 13. Percentage deviation of final solution from minimum MII.

average within 4.2% of the global optimum. This result validates our overall approach. We are able to generate high-performance designs because of the superior quality of the retiming heuristics.

C. Case Study: Pipelined Implementation of JPEG Algorithm

We consider the HW–SW codesign of the JPEG image compression algorithm. It is a loop oriented application and therefore ideal for pipelined implementation. We specified the algorithm as a DAG with 12 tasks (see Fig. 14): preprocessing, eight tasks that perform vector-matrix multiplication, zig-zag encoding, quantization, and a task that performs runlength and Huffman encoding. The eight tasks of vector-matrix multiplication together perform the forward discrete cosine transformation (FDCT). The only difference with the JPEG standard was that our specification operates on a $[4 \times 4]$ matrix of pixels instead of a $[8 \times 8]$ matrix. The FDCT task was split to expose the parallelism present in it. The "C" specification of the tasks required about 1900 lines of code. The SW times were obtained by profiling the task graph on a 100 MHz pentium based PC. The HW time and area for each task were obtained for an ASIC implementation by using a high level synthesis tool. The system bus was assumed to be a PCI bus operating at 33 Mhz with two-cycle transfer time. We then obtained pipelined codesign implementations for the algorithm by specifying different constraints on the II and area. The results of the experimentation are in Fig. 15. The results vary from the fastest implementation (in the top left) that occupies the maximum area to the slowest all software sequential implementation (bottom right). The clock frequency of the coprocessor was estimated as the reciprocal of the maximum register to register delay over all tasks mapped to HW. The clock frequency was determined to be about 20 Mhz for all cases.

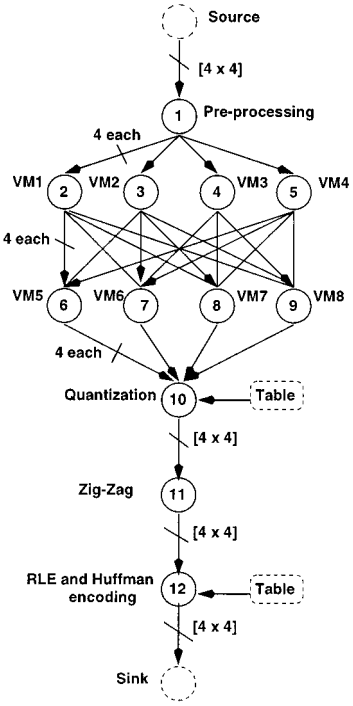

Fig. 14. Task graph for JPEG encoding.

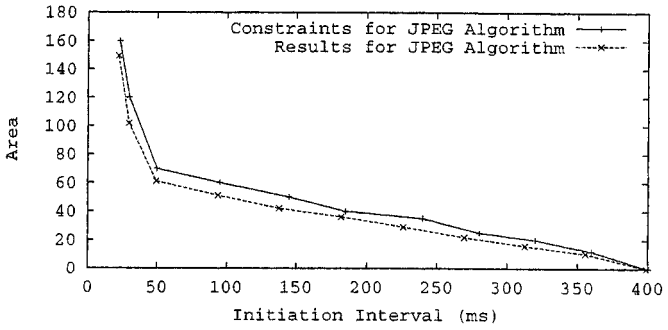

Fig. 15. Design space exploration for the JPEG algorithm.

The tool took less than 2 s to generate a feasible solution for each of the design constraints. Moreover the pipelined schedules for all the partitions executed with MII, that is, all were optimal-pipelined schedules. This observation justifies the objective function of the codesign partitioner. It also indicates that the pipelined scheduler is able to generate high quality schedules.

V. Discussion

In this section, we briefly discuss extensions to our work, its limitations, and pipeline hazards.

A. Extensions

We assume that the application specification does not contain dependence loops. Dependence loops at a coarse level of granularity typically occur as control loops. For example, the MPEG-2 encoding algorithm might contain a control loop to adjust the quantization task to compensate for variable decoding rate. Dependence loops also limit the attainable MII of a task graph. The MII due to a dependence loop l in a data dependence graph DDG is given by $MII^i_{\text{loop}} =$

$(\sum_{u\in l} u_{\text{exec}}/\sum_{(u,v)\in l}\delta(u,v))$ [48]. The MII for the task graph is given by $MII = \max(MII_{\text{loop}}, MII_{\text{sw}}, MII_{\text{hw}})$ where $MII_{\text{loop}} = \max(MII^i_{\text{loop}}), \forall \text{loops}\, l \in DDG$. We can handle such dependence loops by detecting them and collapsing the loops into single tasks during retiming. This technique, however, does not optimize the dependence loop by redistributing the delays.

Our tool considers a single HW design point for every task with one HW area and HW runtime. In reality, for each HW task it is possible to generate multiple design points that have varying area and HW run times. Our tool can be easily modified to handle multiple HW design points. The modifications to be made are as follows:

- We modify the suitability calculation (Section III-BI) by calculating the suitability for each design point of the task. For a HW task the initial solution algorithm selects the design point that has the maximum suitability over all design points of a task.
- During the branch and bound search while mapping a task to HW we first select the design point with maximum suitability.

The conditions for limiting the search space can be applied without modifications.

The tool assumes a fixed codesign architecture consisting of a SW processor and HW coprocessor. Although, this architecture has been widely used in literature[10]–[16] it is limited by the number of processors. A multiprocessor architecture with multiple HW and SW processors would be more versatile. We can easily extend our tool for an architecture that has multiple HW coprocessor and single SW processor. This problem is similar to multiple HW design points problem described above. In this case we would calculate the suitability of the task for different design points on each HW coprocessor. The branch and bound algorithm would then map a HW task to the HW coprocessor that has the maximum suitability design point. We would like to stress that both these modifications would result in an exponential increase in the execution time of the algorithm.

B. Limitations

The tool is limited by its branch and bound-based partitioner. Although, the tool generates solutions for 30 node tasks in reasonable amount of time, it cannot handle larger task graphs. Any modification to the algorithm as described above causes an exponential increase in the execution time of the algorithm. Also, as mentioned earlier our technique cannot optimize dependence loops. Future work will involve alleviating these two limitations.

The task graph format assumed by the tool does not support conditional constructs at top level. An individual task itself may contain control-flow constructs inside of it. Most transformative applications can be described in our task graph format and we can obtain pipelined implementations for them. Since the task graph format does not support control-flow constructs the application domain of our tool is limited to transformative applications.

C. Pipeline Hazards

Pipeline hazards are of three types: read after write, write after write, and write after read [51]. A pipeline hazard occurs when a data dependence is violated. The runtimes of the different tasks may depend on the input-data set. In such a case, if a task finishes execution in a shorter time it may cause a hazard if it executes its write operation. We avoid pipeline hazards by strictly enforcing the schedule during cosynthesis. The schedule honors all data dependencies and establishes an order on task execution and task communication. During cosynthesis, both of these are embedded in the final implementation. Hence, even if the execution time of the task varies it does not cause a pipeline hazard since the communication schedule is not violated.

VI. Conclusion

We presented a tool for HW–SW partitioning and pipelined scheduling of transformative applications. The HW–SW partitioner used a branch and bound approach with a unique objective function that optimized the initiation interval of the final design. We discussed techniques for generating the initial solution, selecting a task to be mapped and limiting the search space of the algorithm. We presented results that established the quality of the initial solution and effectiveness of the search space bounding techniques. We then presented a novel retiming heuristic that optimized the initiation interval, number of pipeline stages, and pipeline buffers of a pipelined implementation. We compared the performance of our retiming heuristic with an existing heuristic. We then discussed a case study for pipelined HW–SW implementation of the JPEG image compression algorithm. The results of the study demonstrated the effectiveness of our approach to a realistic example. We concluded the paper with a discussion on possible extensions to our tool, its limitations, and pipeline hazards.

References

[1] J. T. J. Van Eijndhoven, F. Sijstermans, K. Vissers, E. Pol, M. Tromp, P. Struik, R. Bloks, P. V. D. Wolf, A. Pimentel, and H. Vranken, "Trimedia CPU64 architecture," in *Proc. Int. Conf. Computer Design*, Oct. 1999, pp. 586–592.

[2] H. Takata, T. Watanabe, T. Nakajima, T. Takagaki, H. Sato, A. Mohri, A. Yamada, T. Kanamoto, Y. Matsuda, S. Iwade, and Y. Horiba, "The D30V/MPEG multimedia processor," in *IEEE Micro*. Piscataway, NJ: IEEE Press, 1999.

[3] M. Ikeda, T. Kondo, K. Nitta, K. Suguri, T. Yoshitome, T. Minami, H. Iwasaki, K. Ochiai, J. Naganuma, M. Endo, Y. Tashiro, H. Watanabe, N. Kobayashi, T. Okubo, T. Ogura, and R. Kasai, "SuperEnc: MPEG-2 video encoder chip," in *IEEE Micro*. Piscataway, NJ: IEEE Press, 1999.

[4] H. De Man, I. Bolsens, B. Lin, K. V. Rompaey, S. Vercauteren, and D. Verkest, "Co-design of DSP systems," in *Hardware/Software Codesign (Proc. the NATO Advanced Study Institute on Hardware/Software Codesign)*, G. D. Micheli and M. Sami, Eds. Norwood, MA: Kluwer , 1996.

[5] K. Melhorn, *Graph Algorithms and NP-Completeness*. New York: Springer-Verlag, 1977.

[6] C. V. Ramamoorthy and H. F. Li, "Some problems in parallel and pipeline processing," in *Proc. COMPCON*, Sept. 1975.

[7] B. Fluiter, E. H. L. Aarts, J. H. M. Korst, W. F. J. Verhaegh, and A. V. D. Werf, "The complexity of generalized retiming problem," *IEEE Trans. Computer-Aided Design*, vol. 15, pp. 1340–1353, Nov. 1996.

[8] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," *Int. J. Computer Simulation*, vol. C-36, no. 1, pp. 24–35, Jan. 1987.

[9] P. Chou, R. B. Ortega, and G. Borriello, *The Chinook Hardware/Software Co-Synthesis System*. Piscataway, NJ: IEEE Press, 1995.

[10] R. K. Gupta and G. D. Micheli, "Hardware–software cosynthesis for digital systems," *IEEE Design Test Comput.*, vol. 10, pp. 29–41, 1993.

[11] R. Ernst, J. Henkel, and T. Benner, "Hardware–software cosynthesis for microcontrollers," *IEEE Design Test Comput.*, vol. 10, no. 4, pp. 64–75, 1994.

[12] A. Kalavade and E. A. Lee, "The extended partitioning problem: Hardware/software mapping, scheduling and implementation-bin selection," *J. Design Automat. Embedded Syst.*, vol. 2, no. 2, pp. 125–163, 1997.

[13] R. Niemann and P. Marwedel, "Hardware/software partitioning using integer programming," in *Proc. European Design and Test Conf, ED&TC*, 1996.
[14] J. Madsen, J. Grode, P. V. Knudsen, M. E. Petersen, and A. Haxthausen, "LYCOS: The Lyngby Co-Synthesis System," *J. Design Automat. Embedded Syst.*, vol. 2, no. 2, 1997.
[15] P. Eles, Z. Peng, K. Kuchinski, and A. Doboli, "System level hardware/software partitioning based on simulated annealing and tabu search," *J. Design Automat. Embedded Syst.*, vol. 2, pp. 5–32, 1996.
[16] J. Jeon and K. Choi, "Loop pipelining in hardware–software partitioning," in *Proc. ASPDAC*, 1998.
[17] S. Prakash and A. C. Parker, "SOS: Synthesis of application-specific heterogeneous multiprocessor systems," *J. Parallel Distributed Computing*, vol. 16, pp. 338–351, 1992.
[18] B. P. Dave and N. K. Jha, "COHRA: Hardware–software co-synthesis of hierarchical heterogenous distributed embedded systems," *IEEE Trans. Computer-Aided Design*, vol. 17, Oct. 1998.
[19] R. P. Dick and N. K. Jha, "MOGAC: A multiobjective genetic algorithm for hardware–software cosynthesis of distributed embedded systems," *IEEE Trans. Computer-Aided Design* , vol. 17, Oct. 1998.
[20] Y. Li and W. H. Wolf, "Hardware/software co-synthesis with memory hierarchies," *IEEE Trans. Computer-Aided Design* , vol. 18, Oct. 1999.
[21] F. Balarin, L. Lavagno, P. Murthy, and A. Sangiovanni-Vincentelli, "Scheduling for embedded real-time systems," *IEEE Design Test Comput.*, Jan.–Mar. 1998.
[22] S. Bakshi and D. D. Gajski, "Partitioning and pipelining for performance-constrained hardware/software systems," *IEEE Trans. VLSI Syst.*, vol. 7, Dec. 1999.
[23] B. R. Rau and C. D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing," in *Proc. 14th Annu. Workshop Microprogramming*, Oct. 1981, pp. 183–198.
[24] C. E. Leiserson and J. B. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, no. 1, pp. 5–35, 1991.
[25] S. Malik, K. J. Singh, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Performance optimization of pipelined logic circuits using peripheral retiming and resynthesis," *IEEE Trans. Computer-Aided Design*, vol. 12, May 1993.
[26] N. Shenoy and R. Rudell, "Efficient implementation of retiming," in *Int. Conf. Computer-Aided Design*, 1994.
[27] S. T. Chakradhar, S. Dey, M. Potkonjak, and S. G. Rothweiler, "Sequential circuit delay optimization using global path delays," in *Design Automation Conf., DAC*, 1993.
[28] N. Maheshwari and S. Sapatnekar, "An improved algorithm for minimum-area retiming," in *Design Automation Conf., DAC*, 1997.
[29] V. Sundararajan, S. Sapatnehar, and K. K. Parhi, "MARSH: Min-area retiming with setup and hold constraints," in *Proc. Int. Conf. Computer-Aided Design*, 1999.
[30] N. Shenoy, "Retiming: Theory and practice," *Integration, VLSI J.*, vol. 22, pp. 1–21, 1997.
[31] S. Huang and J. Rabaey, "Maximizing the throughput of high performance DSP applications using behavioral transformations," in *Proc. European Design Automation Conf., EuroDAC*, Mar. 1994, pp. 25–40.
[32] L.-F. Chao and E. H.-M. Sha, "Scheduling data-flow graphs via retiming and unfolding," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, Dec. 1997.
[33] S. Hassoun and C. Ebeling, "Architectural retiming: Pipelining latency-constrained circuit," in *Proc. Design Automation Conf., DAC*, 1996.
[34] M. Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," in *Proc. SIGPLAN Conf. Programming Language Design Implementation*, Atlanta, GA, June 1988, pp. 318–328.
[35] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, 1994.
[36] R. Govindrajan, E. R. Altman, and G. R. Gao, "A framework for resource-constrained rate-optimal software pipelining," *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, Nov. 1996.
[37] S.-M. Moon and K. Ebcioglu, "An efficient resource-constrained global scheduling technique for superscalar and VLIW processors," in *Proc. Int. Symp. Workshop Microarchitecture*, Dec. 1992.
[38] A. Aiken and A. Nicolau, *Perfect Pipelining: A New Loop Parallelization Technique, Tech. Rep., Department of Computer Science*. Ithaca, NY: Cornell Univ. Press, 1987.
[39] J. Wang, C. Eisenbeis, M. Jourdan, and B. Su, "Decomposed software pipelining: A new perspective and a new approach," *Int. J. Parallel Programming*, vol. 22, no. 3, 1994.
[40] P.-Y. Calland, A. Darte, and Y. Robert, "Circuit retiming applied to decomposed software pipelining," *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, Jan. 1998.
[41] N. Park and A. C. Parker, "Sehwa: A software package for synthesis of pipelines from behavioral specifications," *IEEE Trans. Computer-Aided Design*, vol. 7, Mar. 1998.
[42] P. G. Paulin and J. P. Knight, "Force-directed scheduling for behavioral synthesis of ASIC's," *IEEE Trans. Computer-Aided Design* , vol. 8, June 1989.
[43] T.-F. Lee, A. C.-H. Wu, D. D. Gajski, and Y.-L. Lin, "An effective methodology for functional pipelining," in *Int. Conf. Computer-Aided Design*, 1992.
[44] G. Goossens, J. Vandewalle, and H. D. Man, "Loop optimization in register-transfer scheduling of DSP-systems," in *Proc. Design Automation Conf., DAC*, 1989.
[45] L.-F. Chao, A. S. LaPaugh, and E. H.-M. Sha, "Rotation scheduling: A loop pipelining algorithm," *IEEE Trans. Computer-Aided Design*, vol. 16, Mar. 1997.
[46] C.-Y. Wang and K. K. Parhi, "High-level DSP synthesis using concurrent transformations, scheduling and allocation," *IEEE Trans. Computer-Aided Design*, vol. 14, Mar. 1995.
[47] C.-Y. Wang, Y.-N. Chang, and K. K. Parhi, "Heuristic loop-based scheduling and allocation for DSP synthesis with heterogeneous functional units," *J. VLSI Signal Processing*, vol. 19, pp. 243–256, 1998.
[48] F. Sánchez, "Loop Pipelining With Resource And Timing Constraints ," Ph.D., UPC Universitat Politècnica de Catalunya, 1995.
[49] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. C-36, Jan. 1987.
[50] T. C. Hu, "Parallel sequencing and assembly line problems," *Operations Res.*, vol. 9, pp. 841–848, 1961.
[51] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. New York: McGraw-Hill, 1993.

Karam S. Chatha (M'01) received the B.E. (Hons.) degree in computer technology from Bombay University, India, and the M.S. and Ph.D. degrees in computer science and engineering from the University of Cincinnati, OH, in 1993, 1997, and 2001, respectively.

He is currently an Assistant Professor in the Department of Computer Science and Engineering at the Arizona State University, Tempe. His research interests include computer-aided design of VLSI systems, HW–SW codesign, system-level design embedded systems and reconfigurable computers.

Dr. Chatha received the Best Paper Award at the Field Programmable Logic and Applications Workshop (FPL), in 1999.

Ranga Vemuri (S'87–M'88–SM'00) received the M.Tech. degree from the Indian Institute of Technology, Kharagpur, and the Ph.D. degree from Case Western Reserve University, Cleveland, OH, in 1985 and 1988, respectively.

He is currently a Professor of the Department of Electrical and Computer Engineering and Directs the Laboratory for Digital Design Environments with the University of Cincinnati, OH. He is coauthor of about 130 research publications. His research interests include computer-aided design of VLSI systems, reconfigurable computers, mixed signal design automation, formal verification, high-level synthesis, and hardware description languages.

Dr. Vemuri is the recipient of the University of Cincinnati Faculty Achievement Award, the Sigma Xi Outstanding Young Researcher Award, the William Middendorf Distinguished Research Award, the William Restemeyer Distinguished Teaching Award, the Engineering Tribunal Award for Outstanding Teaching, and several best paper awards at various international conferences, including VLSI Design 2000, FPL 1999, and ICCD 1998. He currently serves as an Associate Editor of IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS. He was a Guest Editor of the IEEE Computers Special Issue on Reconfigurable Computers: Technology and Applications, and of the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS Special Issue on Adaptive and Reconfigurable VLSI Systems.