A Scheduling Algorithm for Optimization and Early Planning in High-Level Synthesis

SEDA OGRENCI MEMIK
Northwestern University
RYAN KASTNER
University of California, Santa Barbara
ELAHEH BOZORGZADEH
University of California, Irvine
and
MAJID SARRAFZADEH
University of California, Los Angeles

Complexities of applications implemented on embedded and programmable systems grow with the advances in capacities and capabilities of these systems. Mapping applications onto them manually is becoming a very tedious task. This draws attention to using high-level synthesis within design flows. Meanwhile, it is essential to provide a flexible formulation of optimization objectives as well as to perform efficient planning for various design objectives early on in the design flow. In this work, we address these issues in the context of data flow graph (DFG) scheduling, which is an essential element within the high-level synthesis flow. We present an algorithm that schedules a chain of operations with data dependencies among consecutive operations at a single step. This local problem is repeated to generate the schedule for the whole DFG. The local problem is formulated as a maximum weight noncrossing bipartite matching. We use a technique from the computational geometry domain to solve the matching problem. This technique provides a theoretical guarantee on the solution quality for scheduling a single chain of operations. Although still being local, this provides a relatively wider perspective on the global scheduling objectives. In our experiments we compared the latencies obtained using our algorithm with the optimal latencies given by the exact solution to the integer linear programming (ILP) formulation of the problem. In 9 out of 14 DFGs tested, our

At the time the research for this article was carried Out, S. O. Memik was affiliated with the University of California, Los Angeles, Los Angeles, CA.
Authors' addresses: S. O. Memik, Department of Electrical and Computer Engineering, Northwestern University, 2145 Sheridan Road, Evanston, IL 60208; email: seda@ece.northwestern.edu; R. Kastner, Department of Electrical and Computer Engineering, University of California, Santa Barbara, Engineering I, Room 4123, Santa Barbara, CA 93106; email: kastner@ece.ucsb.edu; E. Bozorgzadeh, Computer Sciences Department, University of California, Irvine, 408E Computer Science Building, Irvine, CA 92697-3425; email: eli@ics.uci.edu; M. Sarrafzadeh, Computer Science Department, University of Califonia, Los Angeles, 3532C Boelter Hall, Los Angeles, CA 90095-1596; email: majid@cs.ucla.edu.

algorithm found the optimal solution, while generating latencies comparable to the optimal solution in the remaining five benchmarks. The formulation of the objective function in our algorithm provides flexibility to incorporate different optimization goals. We present examples of how to exploit the versatility of our algorithm with specific examples of objective functions and experimental results on the ability of our algorithm to capture these objectives efficiently in the final schedules.

Categories and Subject Descriptors: B.5.2 [**Register-Transfer-Level Implementation**]: Design AIDS—*Automatic synthesis*; J.6 [**Computer-Aided Engineering**]: *Computer-aided design*

General Terms: Design, Algorithms

Additional Key Words and Phrases: Scheduling, high-level synthesis, data flow graph, bipartite matching

1. INTRODUCTION

Traditionally, translation of application descriptions into a synthesizable hardware description language (HDL) was a manual process. Then, designs specified with an HDL were mapped onto embedded or programmable systems by logic and physical synthesis tools. Due to the increasing complexity of the applications, mapping applications manually is becoming harder. Therefore, increasing the level of abstraction for designers and automating the mapping process is becoming more attractive. This alternative paradigm involves automatic compilation from high-level descriptions of applications, such as from a high-level programming language (e.g., C, C++). This approach offers designers a very convenient and familiar computational model. There are several proposed compilation flows from a high-level of abstraction to hardware [Wazlowski et al. 1993; Hammes et al. 1999; Gokhale et al. 2000; So et al. 2002; Haldar et al. 2001; Schreiber et al. 2002].

Figure 1 depicts an example flow for automatic mapping of applications onto various hardware platforms. The application described in a high-level programming language is processed by the compiler stage. The compiler generates an *intermediate representation* (IR) and performs several optimizations such as constant propagation, loop unrolling, and function inlining on this IR. While internal representations in different compilers take different forms and names, essentially they capture two basic pieces of information about an application: control flow and data dependency. A high-level synthesis stage follows the compiler stage and takes the optimized IR as input and generates the *register transfer level* (RTL) description of the design. Back-end tools perform logic synthesis and physical synthesis on this RTL description and create the bit-stream data to program the target system. In the most general form of this flow, feedback paths between major steps may exist to incorporate physical level information into high-level synthesis or hardware/synthesis-related information into compiler stage (denoted by dotted arcs in Figure 1). Feedback is employed in such flows to improve the interaction between different design phases and refine the quality of the solution generated at each stage. However, feedback can also cause problems in convergence and design closure. In order to create a more coherent design flow, planning early on can be a better alternative. This can be achieved by planning of design objectives or through use of correct by construction methodology.

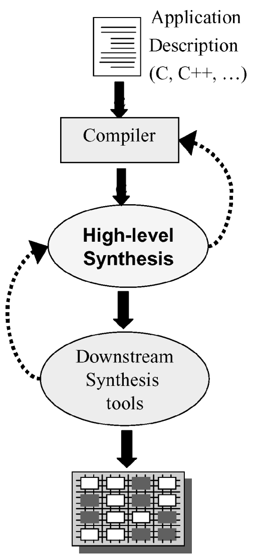


Fig. 1. A representative flow for automatic mapping of applications from high-level descriptions to a programmable system.

In this work, we present a scheduling algorithm for data flow graphs (DFGs), which is an integral element within the high-level synthesis stage of this automated flow. Our method "simultaneously" assigns a set of operations within the input DFG to control steps. (Note that scheduling several operations simultaneously does not refer to assigning them to the same control step. By scheduling them simultaneously we mean to generate a scheduling decision for a collection of nodes at once.) Each set of nodes selected to be considered together constitutes an ordered set. There exists a direct data dependency between every pair of consecutive nodes in each ordered collection of operations. We refer to these sets of operations as *paths*, since they constitute a topological path in the DFG. In the proposed algorithm, the scheduling problem for each individual set of nodes is formulated as maximum-weight noncrossing bipartite matching. This local problem in turn is solved optimally by converting it to the *max-weighted k-chain problem* [Atallah and Kosaraju 1989]. The bipartite matching basically provides the assignment of operations to control steps. DFGs impose a data dependency constraint on the scheduling problem. This is reflected in our algorithm by the noncrossing property of the matching solution. Furthermore, the matching is weighted and the objective is to produce a matching with maximum edge weight total. The particular objective function of the actual scheduling problem is embodied within the maximum-weight objective of the matching. We will elaborate on the specifics as we discuss our algorithm in detail. At this point, however, it is appropriate to comment on the impact of these properties on the global behavior of our scheduling algorithm.

First, our algorithm assigns several nodes along a path to control steps at once. This local assignment is realized by solving the matching between operations and control steps optimally. Each operation-control step matching within this solution is associated with a corresponding weight/gain. The local solution is optimal for the given set of operations in the sense that the summation of the weights of the matching generated for those operations is maximum. This distinguishes our algorithm from other heuristics such as list scheduling [Parker et al. 1986; McFarland et al. 1988; Pangrle and Gajski 1987], force-directed scheduling [Paulin and Knight 1987; Cloutier and Thomas 1990], etc., which generally make a scheduling decision about a single operation at a time. In our algorithm we generate a solution for a collection of operations while maximizing the scheduling objective for this set of operations. While the quality guarantee remains local, this helps to provide a good solution at each step for the operations along each path at hand. Furthermore, by manipulating the weight function associated with the matching between operations and control steps, a wide variety of objectives can be combined within a solution. Hence, our algorithm provides a flexible way of defining and changing our scheduling objective function. In this article, we discuss two specific cases in more detail: efficient utilization of optimized cores embedded in the target architecture and early planning and distribution of time slack within a DFG during scheduling.

The rest of the article is organized as follows. Section 2 states the scheduling problem and defines the objective and the constraints. We give a brief overview of existing scheduling heuristics in Section 2.2. Our algorithm is described in Section 2.3. In Section 3 we present how our algorithm can be applied to target two particular objective functions for schedules. First, we discuss using our scheduler for hybrid target architectures, aiming to optimize the utilization of embedded cores within the target architecture. Next, we present how our flexible scheduling cost function can be used to target efficient distribution and management of time slack within a schedule. We present experimental results for these two problem instances. We discuss our conclusions and future work in Section 4.

2. SCHEDULING OF DATA FLOW GRAPHS

In this section we formulate our problem and state the constraints on the problem. We also define our objective function. Next, we present our scheduling algorithm.

2.1 Problem Formulation

Given a data flow graph (DFG),[1] the scheduling problem is to assign each operation in the DFG to a control step under certain constraints. Any assignment that is feasible under these constraints is a valid schedule. Out of possible valid schedules the goal is to find one that optimizes a given objective function. An immediate objective function for any scheduling algorithm is the length of the schedule or the latency. In addition, depending on the specific context in which the scheduler is used, other components are incorporated into the objective function. Objectives such as power [Musoll and Cortadella 1995; Monteiro

[1]A data flow graph is basically a directed acyclic graph (DAG).

et al. 1996; Shiue and Chakrabarti 2000] and register usage [Wong et al. 2002] have been incorporated into scheduling algorithms in the past. In this work we will introduce two other objective functions and show how the cost function of our algorithm can be easily adjusted to include either of those. We will discuss these objective functions in Section 3. The objective function is maximized under a set of constraints. For our scheduling problem the following constraints are given:

—For each operation a start time must be defined.
—Data dependencies imposed by the DFG must be obeyed. Let, op_i and op_j be two operations in the DFG. In addition, let op_i be the parent of op_j. This means that op_j has a data dependency on op_i. Then, the control step in which op_j starts must be later than the finish time of op_i.
—At any control step, the number of active operations of any type must be less than or equal to the number of available resources of that type.

2.2 Scheduling Algorithms Overview

Most of the practical formulations of the scheduling problem are NP-complete. These instances contain combinations of dependency, timing, and resource constraints. Several efficient heuristics have been proposed in the literature for these problem instances. Two of the most widely used approaches are *list scheduling* and *force-directed scheduling*. List scheduling maintains an ordered list of operations that can potentially be scheduled at a control step with no violation of data dependency. Considering one control step at a time, operations are selected from this ordered list one by one according to some priority function and scheduled at the control step under consideration. There exist a variety of realizations of this approach [McFarland et al. 1988; Thomas et al. 1990; McFarland 1986; Parker et al. 1986; Pangrle and Gajski 1987; Kramer and Rosenstiel 1990]. In force-directed scheduling [Paulin and Knight 1989] the goal is to create a balanced distribution of operations among control steps. Using the mobility of each operation to define possible intervals of execution, the potential *demand* for each control step is determined. The operation-to-control step assignment, which will contribute toward the most homogeneous distribution is accepted at each step. This approach has been incorporated into high-level synthesis systems as well [Paulin and Knight 1987; Cloutier and Thomas 1990]. Another type of scheduling method is referred to as *path-based scheduling* in the literature [Camposano 1991]. This method has been proposed for scheduling control flow graphs. Paths of execution within the control/data flow are handled individually. Each such possible path is scheduled independently in an optimal fashion. Then the final schedule is constructed by imposing the resource constraints and overlapping the path's schedules accordingly. Other popular techniques for scheduling with control flow are trace scheduling from microcode compaction [Fisher 1981] and percolation scheduling [Potasman et al. 1990]. While most of the above-mentioned scheduling heuristics produce a scheduling decision for one operation at a time, our algorithm generates an assignment between multiple operations and control steps at once. The particular set of operations to be scheduled at each step constitute a chain of data dependency. This may be also called a *path spanning* through the DFG. In path-based scheduling algorithms mentioned earlier such as [Camposano 1991], any sequence of operations that can possibly be executed conforming to the control flow is a path. Those operations do not need to create a chain with direct data dependencies. Therefore the path-based scheduling algorithms consider an exponential number of possible execution paths and combine their schedules eventually. In our scheme the definition of a path is restricted to chains of operations with data dependencies among them. The number of such paths extracted from an input DFG is kept small, just enough to have considered all operations within the DFG. In fact, we can find such a set of paths for a DFG in polynomial time.

Our technique to perform the actual assignment of operations to control steps is based on weighted noncrossing bipartite matching. This approach is fundamentally different from list scheduling and force-directed scheduling, and path-based scheduling in nature. Moreover, at the local level, we can provide theoretical guarantees on the quality of the operation-control step assignment for individual operation chains. An algorithm proposed by Timmer and Jess [1995] uses bipartite graph matching for scheduling DFGs. In their work, a bipartite matching was performed between operations in the DFG and control steps without considering dependencies and the result was pruned with a heuristic in order to comply with precedence constraints. In our approach we show how to *optimally* solve the matching problem between a set of operations along a path and control steps *while satisfying* precedence constraints within the path. Hence, we have integrated the two problems handled separately by Timmer and Jess [1995]. Also, the method proposed by Timmer and Jess does not incorporate any objective function into the bipartite matching stage. In our method, we are additionally maximizing an objective function by generating a *maximum weight* matching. We will present a detailed analysis of our algorithm in the next section.

2.3 Our Scheduling Algorithm

Our scheduling algorithm consists of two main tasks. First, selecting a chain of operations to schedule together. Next, scheduling this set of operations by generating a noncrossing maximum-weight bipartite matching. The input to our algorithm is a data flow graph, and resource constraints for each resource type. In the following we will present details of the two main tasks within our algorithm.

2.3.1 *Selection of Operation Chains.* At each step we extract a chain of operations from the input DFG. We start with extracting the longest path from the DFG. After one set of operations are scheduled those operations are removed from the DFG and the next longest path within the remaining graph is found. In this manner paths are selected in decreasing order of path delays. While doing that we retain information regarding the latencies of the scheduled operations. In other words, for nodes already scheduled in the DFGs their delay is added on top of the expected delays of remaining paths to preserve the ordering of criticality among different paths. Finding the longest path in a directed acyclic

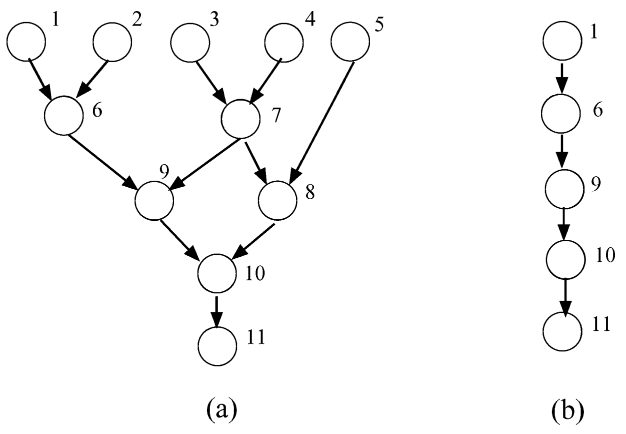


Fig. 2. (a) Example data flow graph. (b) A sample path extracted from the data flow graph.

graph (DAG)—as a graph type, a DFG is basically a DAG—can be done in polynomial time, in $O(V + E)$ time (V is number of nodes and E is number of edges in the DAG). The first chain of operations we process are those along the critical path of the DFG.

As the algorithm progresses the schedule of each new set of operations is constrained by the existing partial schedule of the DFG so far. The task of scheduling a given chain of operations at each scheduling step is tackled with a geometric approach.

2.3.2 *Scheduling Using Maximum-Weight Noncrossing Bipartite Matching.* The input to the local problem, that is, scheduling of a path, is a chain of operations. Consider the example DFG shown in Figure 2(a). Figure 2(b) illustrates a chain extracted from this DFG.

We visualize the problem of scheduling the operations along this chain as a bipartite matching between operations and control steps. A bipartite graph depicting this formulation for the chain in our example is shown in Figure 3(a). This bipartite graph consists of nodes corresponding to operations and control steps. An edge between an *operation-node* and a *control step-node* represents the possibility of assigning that operation to that control step. Constraints of the scheduling problem decide whether it is feasible to have an edge between a certain operation and a control step. For instance, at some point during the execution of our algorithm, at a certain control step the number of operations scheduled might have reached the number of available resources of some type. As we build the bipartite graph for the next set of operations to match to control steps, there cannot be an edge between any operation demanding that resource type and this control step. For any two consecutive operations along the path, the earliest possible matching between the child any control step must be later than the earliest possible matching between the parent and any control step. The latencies of individual operations may be single cycle or multiple cycles. When deciding the latest cycle at which a predecessor can finish we take this into account for multicycle operations. For instance, consider the operations

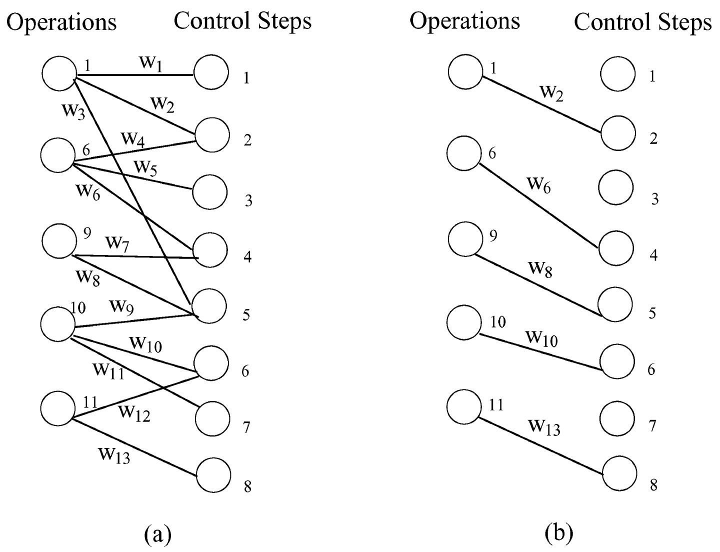


Fig. 3. (a) Bipartite graph representing the matching between operations in a path and control steps. (b) A noncrossing bipartite matching between operations and control steps.

Op_6 and Op_9. The earliest possible matching for the parent operation is to control step 2. Therefore, there cannot be any edge between the child operation, Op_9, and any control step earlier or equal to control step 2. Also, as operations from extracted paths are scheduled, their start and finish times restrict the intervals of control steps within which their predecessors and successors need to be scheduled. Our matching is also weighted, that is, each edge in the bipartite formulation is assigned a weight. In fact, for each path under consideration our goal is to find a matching such that the sum of the edge weights in the matching is maximum. The edge weights are used to formulate the objective function of the schedule; hence maximizing the sum of the weights in turn maximizes our objective function.

A mathematical function F needs to be defined to compute the weights. F is constructed according to what we want to achieve in our schedule. In different cases and applications different objectives can be more relevant or important. A traditional objective for scheduling is minimizing latency. However, a versatile and flexible scheduling algorithm should be able to consider other objectives depending on the particular context. Our approach in constructing F is to combine different optimization objectives in a weighted sum. User-defined coefficients for each term of F maintains the balance between different objectives. In the next section we will provide specific examples of objectives that can be incorporated into our cost functions.

A valid schedule for a path is a noncrossing matching between operations and control steps. This is a different problem than the general bipartite matching. Techniques, such as max-flow [Cormen et al. 1990], to find a bipartite matching do not guarantee yielding a noncrossing matching. A noncrossing bipartite matching means that in the matching solution no edges are allowed to cross.

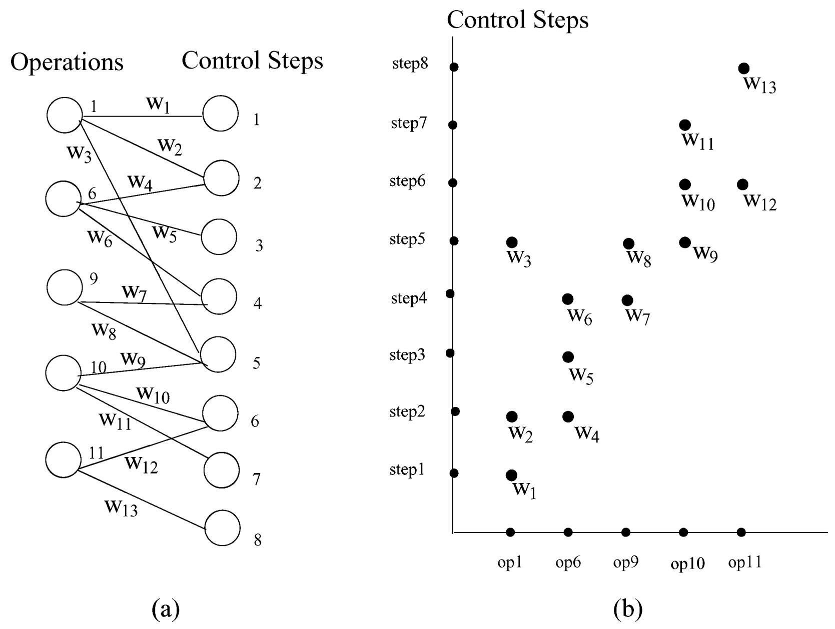


Fig. 4. (a) Weighted bipartite formulation for scheduling a path. (b) The geometric formulation of scheduling.

This actually corresponds to the data dependency constraint. Since along each path consecutive operations have data dependency, their matching to control steps must be noncrossing. A possible noncrossing matching for the example path from Figure 2(b) is shown in Figure 3(b).

Now the question is how to find a noncrossing matching with maximum weight total out of all possible noncrossing matchings. The bipartite matching formulation is a conceptual aide to visualize our problem. To find the actual solution, we will transform our problem into geometric domain and use the dominance concept in computational geometry [Lee 1996; Atallah and Kosaraju 1989] to solve our scheduling problem. To do this we first create a point in the x-y plane for each possible matching between operations and control steps. Figure 4 shows the original bipartite graph and the corresponding set of points in the x-y plane for our example path. Operations are placed along the x-axis and the control steps are placed along the y-axis. A possible matching between an operation Op_i and control step c is represented by a point in the plane with coordinates (x, y). To create the point set for a path of length k (k is equal to 5 for our example, since there are 5 operations along the path), we enumerate the operations with indices starting from 1 to k, following the topological order of the operations along the path. Then, the coordinates of each point representing the edge between Op_i and control step c are defined as follows:

$$x(Op_i) = index(Op_i) \quad \text{and}$$
$$y(Op_i) = c.$$

In our example index(Op_1) = 1, index(Op_6) = 2, and so on. The weights associated with edges in the bipartite graph are attached to the points on the plane as weights.

The following definitions will provide the background and the necessary terminology for understanding our method to perform the scheduling.

Definition 1. On the two-dimensional plane, point P *dominates* point Q iff $x(P) > x(Q) AND y(P) > y(Q)$.

Definition 2. A set of k points $(P_1, P_2, \ldots, P_k)$ in the x-y plane, where P_i dominates P_{i-1} is called a *k-chain*.

Definition 3. Given a set of points in the x-y plane, among all chains of length k that exist within this set, the k-chain with the maximum total of weights is called the *maximum weighted k-chain*. The weights are those attached to each point in our formulation.

Having created a set of points in the plane as explained above, the schedule for the path is generated by finding the *maximum weighted k-chain* within this point set. k is equal to the number of operations on the path that is being scheduled. By finding a chain of length k, that is, selecting k points from the plane, we will have found a matching between each operation and a control step. The property described in Definition 1 ensures that the matching found complies with dependencies among operations. Also we will ensure that our method finds a chain of length k and does not return any chain of shorter length, which would leave some operations unscheduled. Once we guarantee finding a chain of length k, combined with the dominance property from Definition 1, each point in the resulting k-chain must correspond to a matching for a distinct operation. Therefore, this k-chain corresponds to the schedule of the path. Assuming that we find a chain of length k, no two points can have the same x-coordinate, since the k-chain would not possess the dominance property from Definition 1 in that case. Hence we guarantee that each operation is included in the solution with a valid matching. We will explain how we guarantee finding a chain of length k every time after we introduce the method of finding the *maximum-weighted chain*. For illustrative purposes, a possible k-chain from the point set of our example and the corresponding partial schedule is shown in Figure 5.

Atallah and Kosaraju [1989] proposed an optimal $O(n \log n)$ (n is the number of points in the plane) algorithm for finding the maximum-weight k-chain. We refer the readers to Atallah and Kosaraju [1989] for the proof of optimality. If for each point P in the plane the weight $w(P) = 1$, then this algorithm returns the longest possible chain, which naturally corresponds to the maximum sum of weights. However, when arbitrary weights are assigned to the points in the plane, this algorithm yields the maximum weighted chain, but not necessarily of length k. As explained earlier, we need to guarantee a chain of length k and depending on our weight function F the weights can take arbitrary values. We propose an adjustment to the weights, such that the algorithm proposed by Atallah and Kosaraju [1989] can be adapted to our problem.

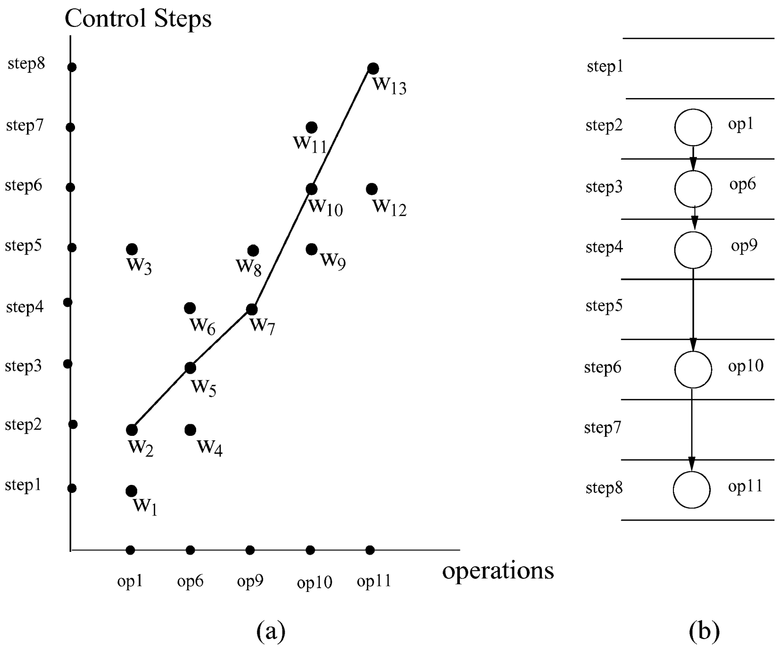


Fig. 5. (a) A hypothetical matching found using a maximum-weighted k-chain. (b) Corresponding partial schedule.

Theorem 2.1. *Let the weight $w(P_i)$ of each point P_i in the plane have arbitrary values. If each weight $w(P_i)$ is replaced with*

$$w(P_i)' = 1 + \frac{w(P_i)}{\left(\sum_{i=1}^{n} w(P_i)\right) + 1}$$

then the maximum weighted chain will be of length k, if any chain of length k exists in the point set.

Proof. Given a set of n points at k different x-coordinates (because we have k operations along the path to be scheduled), assume there exists a maximum weighted chain of length $k-1$. Then, the sum of weights on this chain would be

$$(k-1)\cdot 1 + \sum_{i=1}^{k-1} \frac{w(P_i)}{\left(\sum_{i=1}^{n} w(P_i)\right) + 1}.$$

The second term in the above sum is the sum of weights for $k-1$ points divided by the total sum of weights plus 1. This term is always less than 1. Therefore,

$$k - 1 + \sum_{i=1}^{k-1} \frac{w(P_i)}{\left(\sum_{i=1}^{n} w(P_i)\right) + 1} < k.$$

On the other hand, if we take any k points (strictly one from each x-coordinate obeying dominance condition; assuming at least one chain of length k exists, we should be able to do that) the sum of the weights of these k, points will be

$$k \cdot 1 + \sum_{i=1}^{k} \frac{w(P_i)}{\left(\sum_{i=1}^{n} w(P_i)\right) + 1},$$

which is definitely larger than the sum of $k-1$ points. This contradicts the initial assumption of having a $(k-1)$-chain with a maximum-weight sum. Hence, the weight sum of any chain of length k will be larger than any chain of shorter length with this weight assignment. (The case for chains shorter than $k-1$ follows same arguments.) □

COROLLARY 2.1. *For any two weights w_1 and w_2, if $w_1 > w_2$, then $w_{1'} > w_{2'}$. This property ensures that if a k-chain found with adjusted weights is a maximum-weighted chain, then by converting back to the original weights we can show that the original weights would yield the same maximum-weight k-chain.*

So far we have explained how to guarantee finding a maximum-weighted k-chain if at least one chain of length k exists in the point set. To guarantee the latter, that is, that there is at least one k-chain in the point set, for each operation along the current path there needs to be at least one feasible matching control step. In other words, there must be at least one point with each x-coordinate from 1 to k. At each scheduling step, we check the point set generated for each path of operations to verify that this is true. If not, we perform a correction pass. This actually corresponds to situations when the schedules of different paths need to be combined. Note that in our algorithm once a path is selected all operations along the path are scheduled regardless of whether any of their predecessors are scheduled. Sometimes, the parent of an already scheduled operation is included in a path that is extracted from the DFG later in the execution of the algorithm. It can be the case that at some point due to resource and/or dependency constraints we cannot find any possible feasible matching between control steps and the parent operation that appeared later in the scheduling process. In these cases we restore the feasibility of the matching, that is, we ensure finding at least one chain of length k in the point set by inserting extra control steps into the schedule. We do this according to the following rules. Assume at some point we are about to schedule a path that contains an operation which has some already scheduled successor(s). If the earliest possible start for the predecessor operation is later than the earliest scheduled successor, then we insert extra control steps right before the earliest scheduled successor and push all operations starting at or later than this step by the delay of the predecessor operation. If the earliest possible start of the predecessor operation is earlier or at the same time as the earliest scheduled successor operations (probably the predecessor operation has large delay, such that its result is not ready for the earliest scheduled successor although it can start before the earliest scheduled successor), then we need to determine the number of extra control steps to be inserted as follows. If the earliest possible start time of the predecessor operation is equal to the start of successor, then we insert as many control steps as the delay of the predecessor operation. If the

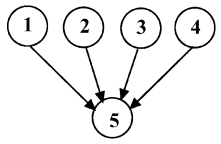


Fig. 6. A sample DFG where $O(V)$ correction passes will be required.

predecessor operation can start earlier than the earliest scheduled successor but cannot finish on time, then we push the schedule by the overlapping amount of control steps by inserting extra control steps.

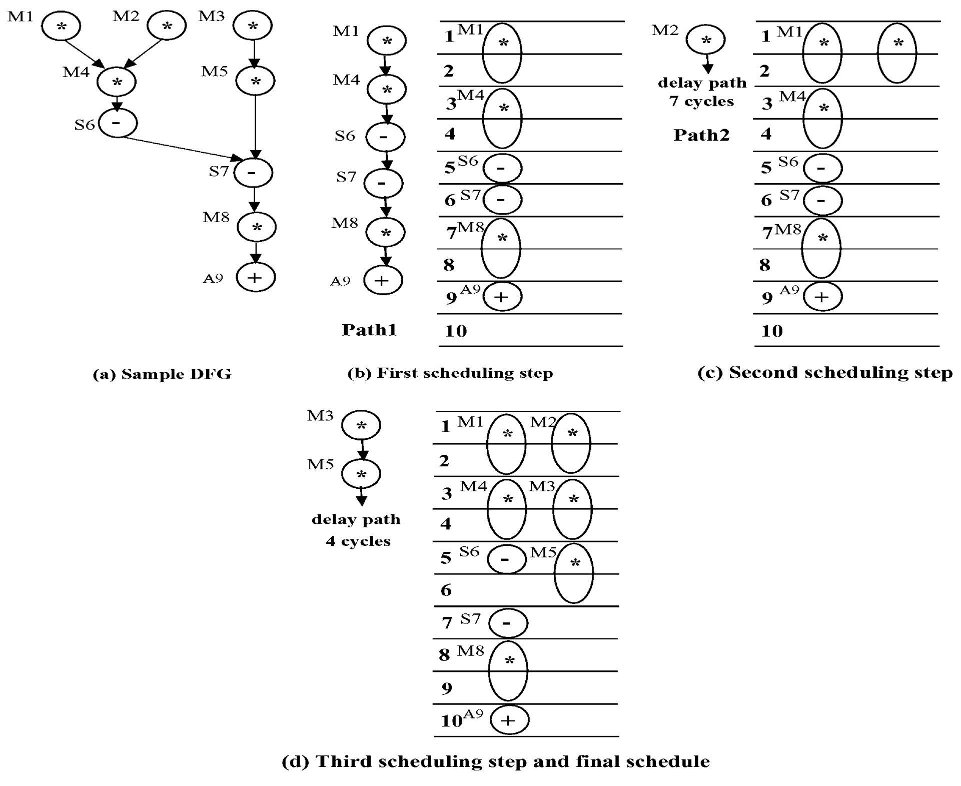
In the following we will make a quick analysis of the worst-case behavior expected from our algorithm in terms of number of correction passes that have to be performed. Consider a DFG as shown in Figure 6. Assume that all operations are of same type and also assume that there is only one resource available. Finally, assume that all operations take a single control step. Given these conditions, our algorithm would first schedule a path such as (Op_1–Op_5) in the first and second control steps. Then, the algorithm encounters Op_2 (plus the delay of Op_5) as the second path. At that point we will need the correction pass and push operation Op_5 by one control step to accomodate time for Op_2. Similarly, in the consequent steps of the algorithm, a correction pass will be required before scheduling Op_3 and Op_4. This shows that we might need corrections proportional to V, which is the number of operations in the DFG. Note that, in each step, the scheduling decision of at least one node (the node with no parent) will be finalized. Therefore, we will not need more than V correction passes. Consequently, in the worst case our algorithm will require $O(V)$ correction steps passes.

The need for correction passes is correlated with three factors. First, it is related to the DFG topology. For DFGs with low connectivity and with more independent paths spanning throughout the DFG, possibly there will be fewer conflicts. Second, the resource constraint will impact the frequency of occurence of conflicts. Consider the earlier example of Figure 6. If there were four resources available, there would be no need for correction passes. If there were three resources, there would be a single correction pass. For more stringent resource constraints, the conflicts during scheduling will increase. Finally, the selection of paths is important. We try to minimize possible conflicts by scheduling the longest paths first. By doing this, operations on long paths would be more likely to be scheduled much later than their yet unscheduled predecessors in shorter paths.

Although our algorithm is locally optimal, it might not yield a globally optimal schedule. This lack of optimality is related to the correction passes. Although we schedule a given path optimally under the given conditions, it does not mean that the final global schedule will require this path to be scheduled within the shortest time. In fact, correction passes try to merge optimally scheduled paths into nonoptimal paths that will be required by the optimal global schedule. However, due to the selection order of the paths and our local decision-making mechanism, we might not be able to merge into the globally optimal schedule.

```
geom_Scheduling(DFG(V,E))
1  while there are unscheduled operations in the DFG
2      Select the most critical (partial) path
3      Generate the point set in the X-Y plane
4      Check for feasibility of point set (insert extra control steps if needed)
5      Assign weight_i = F(p_i) to each point p_i
6      Apply max_weighted_k_chain()
7      Update the DFG
8  end while
```

Fig. 7. The overall scheduling algorithm.



M1
M2
M3
M4
M5
S6
S7
M8
A9
(a) Sample DFG
Path1
(b) First scheduling step
delay path 7 cycles
Path2
(c) Second scheduling step
delay path 4 cycles
(d) Third scheduling step and final schedule

Fig. 8. Illustration of the algorithm with a sample DFG.

Combining all the steps explained above, the overall scheduling algorithm is summarized in Figure 7.

In Figure 8 we illustrate the execution of the algorithm on a sample DFG. We assume that one ALU to execute additions and subtrations and two multipliers is available. In addition, the latency of the ALU is a single cycle and the latencies of the multipliers are two cycles. In this case the scheduling algorithm executes in three steps. In the first step, the chain consisting of M1, M4, S6, S7, M8, A9 is scheduled. For this example we assume that weights of matchings are adjusted such that the earliest schedule step is preferred for each operation. In the second step, the operation M2 is scheduled. M2 is chosen next considering that M2 combined with the path following M2, which consists of already scheduled operations, has the next longest delay. In the third and last step, the chain consisting of operations M3, M5 is scheduled. However, at this step the initial bipartite graph created for this chain is infeasible. This is due to the fact that the earliest possible matching between operation M5 and a clock step with a free multiplier is later than the step at which the successor of M5, which is S7, is scheduled. Therefore, at this point a correction step is applied. S7 and all transitive successors of S7 are shifted by a cycle. After this correction, step M3 and M5 can be successfully scheduled. The final schedule is shown in Figure 8(d).

3. TUNING THE MAXIMUM WEIGHTED MATCHING FOR SCHEDULING OBJECTIVES

We can use the maximum-weighted matching procedure to optimize different objectives within the schedule. Any possible matching between an operation and a control step would contribute to the quality of the certain feature(s) that we aim to embody in the final schedule. Different assignments of operations to control steps can result in different amounts of resource requirements, interconnect structure, switching activity, operation slack, etc., in the final synthesized design. In this article we present how we can utilize the flexibility of the weight assignment to pursue two specific objectives: utilization of embedded fixed cores within an embedded system or within the programmable fabric of a reconfigurable device, and early planning and distribution of slack within a schedule.

An immediate objective for our scheduling algorithm is to minimize latency. In order to optimize for this objective only, we need a function that assigns monotonically decreasing values to weights. We need such a distribution among the weights assigned to all feasible matchings between an operation and all candidate control steps. In other words, for each operation the weight of matching it with a certain feasible control step c must be larger than matching the same operation with any *later* feasible control step. A possible function to create these weights could be as follows:

$$F = \kappa - c.$$

Here, κ is some constant and c is the index of the control step. As we go further in time axis, the value of F will be monotonically decreasing.

3.1 Embedded Core Utilization Objective

A possible objective in high-level synthesis can arise due to the specific architectural features of the target system. For instance, looking back at the evolution of programmable systems, configurability was first available in standalone chips. These devices possess 100 % programmability. Currently, reconfigurable fabric is not only considered to be confined to standalone chips, but also is part of hybrid systems such as system-on-chip (SoC) solutions. While one trend is toward embedding reconfigurable cores into SoCs with processors, DSPs, etc. [Hauck et al. 1997; Actel ; Altera Corp. ; Chameleon Systems], another direction of new architectures considers integration of optimized cores and hardwired blocks with reconfigurable fabric. The main goal here is to utilize the optimized blocks to improve the system performance. Such programmable devices are targeted for a class of applications, such as DSP [Xilinx, Inc.] networking, or data communications [Lucent Technologies]. Embedded fixed blocks are tailored for the critical operations common to the application class. In essence, the flexible programmable logic is supported with the high-density, high-performance cores. This can be applied at various levels, such as the functional block level [Lucent Technologies] or the level of basic arithmetic operations, for example, multipliers [Xilinx, Inc.].

In the context of reconfigurable systems, there have been efforts to create compilation frameworks where templates for frequently occurring operations or operation clusters are extracted and mapped onto specialized cores. Such a framework is described by Kastner et al. [2001]. In a synthesis environment, where such templates are recognized for a given application during compilation, it is crucial to utilize these optimized modules during the actual synthesis of the datapath at the high-level synthesis stage.

For mapping designs on such special architectures, there is a need for synthesis tools that are aware of the features of the underlying hardware resources. The customized cores certainly improve the application's running time since they are superior in delay to their counterparts implemented with reconfigurable logic. A similar argument is valid for the power consumption. Finally, those blocks will lessen the reconfiguration overhead for the overall design. The power of the context-based reconfigurable architectures lies in the efficient utilization of the fixed cores within the system. By customizing our weight function F accordingly, we can make our scheduler aware of the customization of the target architecture. As a result, as we schedule DFGs we can exploit the optimized embedded cores without causing their limited availability become a bottleneck. Functional units for any desired operation type can be instantiated using reconfigurable logic. For operations that cannot be performed by the embedded cores, this is a necessity. For other operations, this can be done in order to exploit parallelism in the schedule. However, as mentioned earlier, the available blocks are highly preferred for those operations. It is the task of the scheduler to do the tradeoff in such situations.

When considering two different feasible control steps for a matching with an operation, the control step with a free customized block would be preferred over the other control step at which no customized embedded block is free. There still can be a feasible matching between the operation and the later step, assuming a reconfigurable module is available or can be instantiated. Nevertheless, the weight assigned to the first feasible matching should be made larger in order to make the algorithm aware of the resource preferences. Also, the value of the weight for the matching of the latter step can reflect the willingness of the synthesis to instantiate new reconfigurable modules. If this matching is associated with a very small value, the tendency would be to avoid instantiation of further reconfigurable modules, possibly in order to control the reconfiguration

Table I. Origins of DFGs from MediaBench

Benchmark	C File	Description
adpcm	adpcm.c	ADPCM to/from 16-bit PCM
epic	convolve.c	2D image convolution
rasta	fft.c	Fast Fourier Transform
mpeg2	getblk.c	DCT Block Decoding
jpeg	jdmerge.c	Color Conversion

overhead. We can introduce these objectives as additional terms within the weight function F. In this case F can take the following form:

$$F = \kappa - \alpha \times c + \beta \times \mathit{customized\_block\_preference} - \gamma \times \mathit{reconfiguration\_overhead\_preference}.$$

We can formulate architecture-related preferences as Boolean variables taking two values, 0 or 1. α, β, and γ are user-defined constants. We have adjusted their values for our algorithm experimentally.

3.1.1 *Results for Embedded Core Utilization Objective.* We have used DFGs extracted from representative functions of C programs within MediaBench multimedia benchmark suite [Lee et al. 1997] and also some additional representative DSP functions such as EWF, FIR, and ARF. Table I shows the files from MediaBench suite, from which input DFGs were generated. The corresponding applications containing these files are given as well. Two or more DFGs were extracted from different procedures within these C Files. Those are indicated as DFG0, DFG1, and DFG2 within each application. These DFGs were mainly selected due to the fact that they were representative of the corresponding applications in terms of size, topology, and operation variety. We tried to select the largest possible DFGs out of those extracted from the Mediabench applications. Each of these DFGs correspond to a basic block[2] in the application codes. Many of those basic blocks tend to be small in size. We tried to avoid such small basic blocks. In order to increase the input DFG sizes and the parallelism, entities containing multiple basic blocks (e.g., hyperblocks [Mahlke et al. 1992], superblocks [Hwu et al. 1993], and traces [Fisher 1981]) can be equivalently given to our scheduler as input. We have not attempted to create such formations at this point, since it is beyond the scope of this work.

Individual DFGs are scheduled with two different methods, as shown in Table III. For each application, a set of hardware resources are specified as given in Table II. Within the available set of resources, there can be multiple components with same functionality, but different delays. This represents the existence of optimized cores for certain operations. In Table III scheduling results for a number of selected DFGs are given. We compare the results of our algorithm against the results obtained from the linear programming solver, CPLEX. The scheduling problem has been described as a linear integer program for our problem instance. The objective function of the integer linear model tries

[2]A basic block is the same entity as in the compiler terminology, which refers to a straight line code segment with a single entry and a single exit point.

Table II. Resource Sets of Benchmark DFGs

Benchmark	Resource set
adpcm	DFG20:(ior, add, comp, mult-fast, mult) DFG36:(comp-fast, comp, add, mult-fast, mult)
convolve	DFG0: (comp, add/sub, mult-fast, mult, div-fast, div) DFG19:(add, mult-fast, mult) DFG98:(comp, add/sub, mult-fast, mult, div-fast, div)

fft	DFG18:(add, mult-fast, mult)
	DFG27:(add, mult-fast, mult, div-fast, div)

getblk	DFG42:(add, comp, AshiftR, mult-fast, mult)
	DFG91:(add, comp, AshiftR,
	LshiftL, mult-fast, mult)
	DFG165:(add, comp, AshiftR, mult-fast, mult)

jdmerge	DFG2: (add, AshiftR, mult-fast, mult) DFG21:(add, AshiftR, mult-fast, mult)
	DFG165:(add, comp, AshiftR, mult-fast, mult)
ewf	(add, mult-fast, mult)
arf	(add, add, mult-fast, mult, mult)
fir	(add, mult-fast, mult)

Table III. Scheduling Results in Terms of DFG Latencies, in Cycles

Benchmark (DFG)	ASAP	CPLEX	Our algorithm
adpcm			
DFG1	6	7	7
DFG2	6	6	6
convolve			
DFG1	12	16	20
DFG2	6	12	12
DFG3	17	18	19
fft			
DFG1	12	26	27
DFG2	19	22	24
getblk			
DFG1	14	14	14
DFG2	18	18	18
jdmerge			
DFG1	7	28	31
DFG2	8	28	28
ewf	17	28	28
arf	8	12	12
fir	7	12	12

to minimize the latency and the number of *slow blocks* used. A higher priority is given to latency minimization. Similarly in our algorithm, we try to minimize the latency, while trying to utilize the optimized blocks as well as possible. We actually perform binding of operation to resources simultaneously with scheduling. This enables us to handle resources of same type but with different delay characteristics. Through simultaneous binding we obtain operation delay

Table IV. Number of Operations Performed on the Optimized Block Versus Total Number of Operations of Matching Type

Benchmark (DFG)	CPLEX	Our algorithm
adpcm		
DFG1	1/1	1/1
DFG2	1/1	1/1
convolve		
DFG1	4/6	3/6
DFG2	2/3	2/3
DFG3	5/6	5/6
fft		
DFG1	6/9	6/9
DFG2	5/6	**6/6**
getblk		
DFG1	3/4	3/4
DFG2	4/4	3/4
jdmerge		
DFG1	5/8	5/8
DFG3	4/5	3/5
ewf	3/8	3/8
arf	10/16	8/16
fir	7/11	7/11

information based on the particular resource executing the operation. CPLEX provides us an optimal solution for the given objective function. Therefore, we are comparing our results to those generated by the exact solver. For comparison, the ASAP scheduling latencies are also provided. In Table IV the utilization of high-performance components are presented. For each DFG the total number of operations assigned to optimized blocks is given versus the total number of operations of suitable type that can be performed by any available optimized block.

As depicted in Table III, our algorithm was able to produce latencies compatible with CPLEX results for most cases. In 9 out of 14 cases our algorithm was able to produce the optimal latency. Out of the remaining five cases, four were within 12% of the optimal value. In two of the suboptimal cases, convolve-DFG3 and fft-DFG1, our algorithm was able to utilize the optimized blocks as good as CPLEX results. In the case of fft-DFG2, the increase in latency resulted from a tradeoff aiming to increase usage of optimized blocks. Table IV shows that, for this particular DFG, our algorithm was able to outperform the optimized block utilization obtained from the CPLEX solution.

For 9 out of the 13 remaining cases our algorithm reached the same core utilization as the CPLEX solution. For six of those cases the latency produced by our scheduler was optimal. For the remaining three cases our scheduling solution was within 10% of the optimal value. In four cases our algorithm yielded lower core utilization. We observe that in three out of those four cases the latency produced by our scheduling algorithm was equal to the optimal latency. Hence, the core utilization, although lower than the solution produced by CPLEX, seems to have been sufficient to reach an optimal schedule. We report

Table V. Number of Nodes and Edges in Each Benchmark DFG and the Runtimes of Two Scheduling Methods

DFG	Size (nodes/edges)	CPLEX	Our algorithm
adpcm			
DFG1	17/19	26 s	4 ms
DFG2	21/18	20 s	9 ms
convolve			
DFG0	49/41	121 s	14 ms
DFG1	25/19	30 s	9 ms
DFG2	18/10	24 s	4 ms
fft			
DFG1	17/12	30 s	3 ms
DFG2	11/9	4.8 s	2 ms
getblk			
DFG1	33/29	34 s	12 ms
DFG2	40/30	42 s	13 ms
jdmerge			
DFG1	79/66	985 s	23 ms
DFG2	54/46	436 s	14 ms
ewf	34/47	13.67 s	25 ms
arf	28/30	700 s	14 ms
fir	21/20	8.8 s	4 ms

the sizes of the benchmark DFGs in terms of number of nodes and edges, and the compare the runtimes of the two schedulers in Table V.

3.2 Early Planning and Distribution of Slack

Another possible use of our flexible objective function is distribution of operation slack within a schedule. We define slack as the amount of extra delay an operation can tolerate without violating any dependency constraints. There can be various uses of this extra amount of allowed time per operation.

Depending on the available slack for an operation, resource selection, IP utilization, power management, clock tree construction, etc., can be different. To give a more specific example: from a single operation's point of view, slack on this operation can be exploited for slowing the module executing this operation or performing power shutdown for this module. Hence, early planning for this objective can have various uses and an impact on the later optimization stages.

We can incorporate this new objective into our weight function F in the following manner. Given a latency constraint λ, we evaluate the weight of a feasible matching for each operation considering the amount of slack that the operation can attain after this matching. Specifically, we aimed to distribute slack along paths homogenously at each step. For this purpose, we first determine the average slack each operation can have along a path. This is found by dividing the total slack along a path to the number of operations on the path. Then, while computing the weight for a certain assignment of an operation to a clock step, we determine the difference between the average slack the operation could attain and the amount it can attain if the current assignment were to take place. The latency constraint is necessary in this case, since the flexibility in finish times of operations needs to be bounded.

Table VI. Summary of DFG Properties, Latency, and Runtime Results

DFG	Optimal latency	Our latency	Runtime
ewf	28	28	10 ms
arf	18	20	7 ms
fir	16	16	6 ms

Table VII. Incorporating Operation Slack into Scheduling Objective

	ewf		arf		fir	
	W/O slack objective	With slack objective	W/O slack objective	With slack objective	W/O slack objective	With slack objective
Num. of ALU operations with nonzero slack	4	5	4	2	3	4
Total slack on ALU operations	14	16	9	8	7	9
Num. of MUL operations with nonzero slack	2	2	7	9	4	5
Total slack on MUL operations	3	3	23	26	6	8

3.2.1 *Results on Planning for Slack.* First, we have used a weight function that only considers to minimize latency. This corresponds to the Without (W/O) Slack Objective case. Alternatively, we have added the slack component to our weight function and used the latencies obtained in the first case as our λ. By doing this, we are able to compare two schedules fairly. Table VI presents the DFGs, the optimal latencies, the latencies obtained with our algorithm, and the runtimes of our algorithm. Table VII shows our results. For this experiment we present results for a subset of our original DFG collection. The particular DFGs selected were those with suitable topologies to reflect improvement in slack. Availability of slack in a schedule depends on the topology of the input DFG as much as on the scheduling method. Therefore, in some DFGs it was not possible to see any effect of slack planning due to their structure. For the selected DFGs, we have used two ALU resources and two multipliers. Each ALU has a delay of one clock cycle, and each multiplier has a delay of two clock cycles. The slack of operations is calculated as the difference between the control step when the result of an operation is ready and the control step at which the earliest scheduled successor of this operation demands the result. We report the number of operations that have nonzero slack values, that is, the rest of the operations in the scheduled DFG had slack values equal to zero or they were I/O operations for which we do not report slack. We only report the slack values of arithmetic operations (MUL and ADD). We also give the total sum of the slack values on each operation type. We present a breakdown of these two measurements for the two types of arithmetic operations in the DFG, that is, the ALU operations and multiplications (abbreviated as MUL in Table VII).

The results in TableVII show that with proper planning to distribute slack on arithmetic operations our algorithm could indeed transform available flexibility in a schedule into additional slack for a spefically targeted set of operations, arithmetic operations in this case. For the arf benchmark we observe a degradation in slack value for ALU operations. The reason is due to the assignment of priorities for the two operations types ALU and MUL. The sensitivity toward increase the slack for MUL operations was set higher for these experiments. Therefore, the gain of increasing the slack for MUL operations is evaluated to be larger than for ALU operations, leading to a greater tendency to allocate slack for MUL operations than for ALU operations. In the case of ewf, due to the DFG topology and the order in which operations were scheduled, MUL operations could not attain any slack whereas ALU operations could gain larger slack-using the slack objective. For the fir benchmark, slack objective affected the slack distribution for both operation types positively.

By allowing a larger number of operations to possess nonzero slack, further optimizations can be possible. For instance, by ensuring slack on an increased number of operations, we would have increased the tolerance of the schedule toward future uncertainties in operation delays. Those uncertainties can arise due to a mismatch between high-level delay estimations and actual operation delays after synthesis of functional modules and interconnect in the datapath.

Additional optimization steps can be incorporated into the synthesis flow following the scheduling in order to exploit the available slack. For instance, a slack-oriented binding methodology can take advantage of this planning by assigning operations with nonzero slack to the same resource. If operations that possess various amounts of slack were assigned to the same resource, then the minimum out of those slack values would determine the extra amount of time by which that resource could be made slower. Using this information, some resources can be replaced by their slower and more power- and/or area-efficient versions. We have investigated possible benefits of exploiting slack distribution in a schedule during binding where we used our scheduling algorithm integrated with a slack-driven binding technique. After distributing time slack to arithmetic operations, as described earlier, we performed binding while trying to group operations with large slack values to the same resource. By doing this, the delay constraint on that resource could be relaxed by the minimum amount of extra slack available on the operations assigned to the resource. Then this information was passed onto the logic synthesis tool as a delay constraint relaxation of the corresponding module. We have observed significant benefits through this relaxation in terms of final design quality. Results to this end are reported in Srivastava et al. [2003] in detail.

Similarly, even if some operations assigned to one resource posses nonzero slack while others have no extra slack, some power optimizations are still possible. In such cases, techniques such as dynamic voltage scaling and dynamic power shutdown can be used if the schedule has been planned for slack distribution early on.

Optimization for slack as described in this section can also be helpful for tasks preceding scheduling. By creating a feedback loop between high-level synthesis and the compilation stage, different compiler optimizations can be leveraged. One such optimization is template extraction and template matching during compilation. At early stages it can be beneficial to extract frequently occuring subgraphs as templates from the DFGs and replace them with optimized cores. Usage of such cores was discussed in Section 3.1. Especially for reconfigurable platforms, usage of preoptimized precharacterized modules can speed up the synthesis process and improve performance. However, information regarding scheduling and binding is not available at the compiler stage. Therefore, the decisions of the compiler to lump subgraphs into larger nodes corresponding to optimized cores might not be accurate enough. This is due to the fact that scheduling of nodes internal to such subgraphs can affect the latency of the overall DFG schedule. Therefore, scheduling and also distribution of time slack among nodes needs to be aware of any clustering of nodes. In turn, after scheduling, a more accurate assessment of the benefits of clustering can be done. Especially, it would not be desirable to leave time slack within a group of nodes which are going to be assigned as a subgraph onto a specialized core. Based on the slack distribution and the schedule information, the compiler can be informed about the quality of clustering decisions made earlier during compilation. If a certain template is not yieling any gain or if it is hurting the performance of the overall DFG schedule, the compiler can be advised to undo that template clustering or redo it in a different way based on the current schedule information.

4. CONCLUSIONS

In this article we presented an algorithm for scheduling data flow graphs. Our algorithm uses a geometric representation of the problem. We applied the maximum weight k-chain technique to generate schedules for paths extracted from the input DFG. This technique is essentially a realization of maximum-weight noncrossing bipartite matching in the geomteric domain. The maximum-weight k-chain method enables us to provide a theoretical guarantee on the quality of each local matching problem. By exploiting the weighted matching feature inherent in our technique, we are able to provide a flexible objective function for the scheduling problem. We manipulate the function that generates the weights for the matching in order to create suitable objective functions for different scheduling problems. Our experiments indicate good results in terms of the combined effort of minimizing latency and utilizing optimized cores available in a given system. We also demonstrated the use of the flexible objective function in managing operation slack within a schedule.

REFERENCES

ACTEL. Visit the Web site http://varicore.actel.com.

ALTERA CORP. Visit the Web site http://www.altera.com/products/prd-index.html.

ATALLAH, M. J. AND KOSARAJU, S. R. 1989. An efficient algorithm for maxdominance with applications. *Algorithmica 4*, 2, 221–236.

CAMPOSANO, R. 1991. Path-based scheduling for synthesis. *IEEE Trans. Comput.-Aided Des. 10*, 85–93.

CHAMELEON SYSTEMS. Visit the Web site www.chameleonsystems.com.

CLOUTIER, R. AND THOMAS, D. 1990. The combination of scheduling, allocation and mapping in a single algorithm. *Proceedings of the Design Automation Conference*.

CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. 1990. *Introduction to Algorithms*. MIT Press, Cambridge, MA.

Fisher, J. A. 1981. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Comput. 30*, 7, 478–490.

Gokhale, M. B., Stone, J. M., Arnold, J., and Kalinowski, M. 2000. Stream-oriented FPGA computing in the streams-c high level language. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*.

Haldar, M., Nayak, A., Choudhary, A., and Banerjee, P. 2001. A system for synthesizing optimized FPGA hardware from matlab. In *Proceedings of the International Conference on Computer Aided Design*.

Hammes, J., Rinker, R., Bohm, W., Najjar, W., Draper, B., and Beveridge, R. 1999. Cameron: High-level language compilation for reconfigurable systems. In *Proceedings of the Conference on Parallel Architectures and Compilation Techniques*.

Hauck, S., Fry, T. W., Hosler, M. M., and Kao, J. P. 1997. The Chimaera reconfigurable functional unit. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*.

Hwu, W. W., Haab, G. E., Holm, J. G., Lavery, D. M., Mahlke, S. A., Chen, W. Y., Chang, P. P., Warter, N. J., Bringmann, R. A., Ouellette, R. G., Hank, R. E., and Kiyohara, T. 1993. The superblock: An effective structure for vliw and superscalar compilation. *J. Supercomput. 7*, 1–2, 229–248.

Kastner, R., Memik, S. O., Bozorgzadeh, E., and Sarrafzadeh, M. 2001. Instruction generation for hybrid reconfigurable systems. In *Proceedings of the International Conference on Computer-Aided Design*.

Kramer, H. and Rosenstiel, W. 1990. System synthesis using behavioral descriptions. In *Proceedings of the European Design Automation Conference*.

Lee, D. T. 1996. “Computational geometry”. In *The Computer Science and Engineering Handbook*, Chap. 6, Allen B. Tucker, Ed. CRC Press, Boca Raton, FL.

Lee, C., Potkonjak, M., and Mangione-Smith, W. H. 1997. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the International Symposium on Microarchitecture*.

Lucent Technologies. Lucent technologies announces high-speed communications cores for customizing ORCA FPGAs.

Mahlke, S. A., Lin, D. C., Chen, W. Y., Hank, R. E., and Bringmann, R. A. 1992. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the International Symposium on Microarchitecture*.

McFarland, M. C. 1986. Using bottom-up design techniques in the synthesis of digital hardware from abstract behavioral descriptions. In *Proceedings of the Design Automation Conference*.

McFarland, M. C., Parker, A. C., and Camposano, R. 1988. Tutorial on high-level synthesis. In *Proceedings of the Design Automation Conference*.

Monteiro, J., Devadas, S., Ashar, P., and Mauskar, A. 1996. Scheduling techniques to enable power management. In *Proceedings of the Design Automation Conference*.

Musoll, E. and Cortadella, J. 1995. Scheduling and resource binding for low power. In *Proceedings of the International Symposium on System Synthesis*.

Pangrle, B. M. and Gajski, D. 1987. Design tools for intelligent silicon compilation. *IEEE Trans. Comput.-Aided Des. 6*, 6, 1098–1112.

Parker, A. C., Pizarro, J., and Mlinar, M. 1986. Maha: A program for datapath synthesis. In *Proceedings of the Design Automation Conference*.

Paulin, P. G. and Knight, J. P. 1987. Force directed scheduling in automatic data path synthesis. In *Proceedings of the International Conference on Computer Design*.

Paulin, P. J. and Knight, J. P. 1989. Force directed scheduling for behavioral synthesis of asics. *IEEE Transactions on Comput.-Aided Des. 8*, 6, 661–679.

Potasman, J., Lis, J., Nicolau, A., and Gajski, D. 1990. Percolation based synthesis. In *Proceedings of the Design Automation Conference*.

Schreiber, R., Aditya, S. G., Rau, B. R., Mahlke, S., Kathail, V., Cronquist, D., and Sivaraman, M. 2002. Pico-npa high-level syntesis of nonprogrammable hardware accelerators. *J. VLSI Signal Process. 31*, 2, 127–142.

Shiue, W. and Chakrabarti, C. 2000. Low-power scheduling with resources operating at multiple voltages. *IEEE Trans. Circ. Syst. II: Analog Digital Sign. Process. 47*, 6, 536–543.

So, B., Hall, M., and Diniz, P. 2002. A compiler approach to fast design space exploration in fpga-based systems. In *Proceedings of the Conference on Programming Language Design and Implementation.*

Srivastava, A., Memik, S. O., Choi, B. K., and Sarrafzadeh, M. 2003. Achieving design closure through delay relaxation parameter. In *Proceedings of the International Symposium on Computer Aided Design.*

Thomas, D. E., Lagnese, E. D., Walker, R. A., Nestor, J. A., Rajan, J. V., and Blackburn, R. L. 1990. *Algorithmic and Register Transfer Level Synthesis: The System Architect's Workbench*. Kluwer Academic Publishers, Norwell, MA.

Timmer, A. H. and Jess, J. A. G. 1995. Exact scheduling strategies based on bipartite graph matching. In *Proceedings of the European Design and Test Conference.*

Wazlowski, M., Agarwal, L., Lee, T., Smith, A., Lam, E., Athanas, P., Silverman, H., and Gosh, S. 1993. Prism-ii compiler and architecture. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines.*

Wong, J. L., Megerian, S., and Potkonjak, M. 2002. Forward-looking objective functions: Concepts and applications in high level synthesis. In *Proceedings of the Design Automation Conference.*

Xilinx, Inc. Visit the Web site www.xilinx.com/apps/appsweb.htm.

Received January 2003; revised August 2003, October 2003; accepted November 2003