Scheduling Algorithms for Automated Synthesis of Pipelined Designs on FPGAs for Applications described in MATLAB[*]

Malay Haldar, Anshuman Nayak, Alok Choudhary and Prith Banerjee
MACH Design Systems, Inc.
Schaumberg, IL, USA.

ABSTRACT

We present a high-level synthesis framework to synthesize optimized hardware on FPGAs from algorithms described in MATLAB. We focus on a framework to pipeline loops present in the input application. We present a range of scheduling algorithms to obtain the pipeline schedule and discuss their comparative strengths. The synthesized hardwares have been mapped to a Xilinx XC4028 FPGA with external memory and corresponding experimental results are included.

1. INTRODUCTION

Field Programmable Gate Arrays (FPGAs) pose a unique solution to the complex needs of signal/image processing and communications architectures because of their high performance and flexibility. Recent advancements in FPGA technology and commercial availability has made reconfigurable computing an attractive option for many applications. FPGAs in the million gates range are already available today. Current trends indicate that FPGAs have a faster growth of transistor density than even general processors. Thus, more and more complex designs will be put on FPGAs to utilize the increased transistor budget. This motivates the need for system level design tools. Current hardware designers use hardware description languages like VHDL/Verilog and low level CAD tools to implement their designs. These tools are inadequate for designing mutli-million gate designs as they involve understanding the cycle-by-cycle behavior of gates. System level design tools will enable these complex designs within acceptable time-to-market.

As a result, there has been tremendous interest in using general purpose languages for the purpose of hardware synthesis. Most of the work has been on synthesizing hardware from algorithms described in C/C++ [14, 18, 13, 15, 17, 12]. Other languages such as Java have also received some consideration [16]. We have chosen MATLAB as the target language due to the following considerations :

- MATLAB is very widely used within the signal/image processing and networking communications communities. Providing a synthesis path directly from MATLAB would shorten the development time of next generation signal/image processing applications dramatically.
- MATLAB has a well defined set of libraries for common functions in signal and image processing. This makes MATLAB very conducive to design reuse which is an important strategy to tackle mutli-million gate designs.
- Absence of pointers and complex data structures and presence of regular loops make MATLAB very amenable to extract parallelism which is the key to get performance out of mapping applications to hardware.

Most of the recent work depends on an optimization compiler framework of the target high level language. The SUIF compiler framework has been a popular choice for synthesis tools targeting C/C++. Closely related works to this paper include the Modula Pipeline compiler[8] and an earlier version of the MATCH compiler[9]. The Modula compiler is based on the SUIF framework. It explores various loop optimizations including pipelining. The principal difference between the Modula compiler and the MATCH compiler is that MATCH attempts to pipeline the VHDL generated from the MATLAB loops, whereas the Modula compiler pipelines the C loops first and then generate hardware for it. Thus, the optimizations of the MATCH compiler are closer to the hardware achieving higher performance, but also expose the complexities of the hardware to the compiler. The earlier version of the MATCH compiler relied on a modified ASAP algorithm to produce the pipeline schedules[9]. While the schedules generated were optimal in terms of states of the schedule, little attention was paid to the resource usage of the pipeline. In this paper, we improve upon the previous framework by taking into account resource constraints while producing the pipeline schedule. We also present a comparison of performance and resource usage among the different scheduling strategies.

The contributions of this paper can be summarized as follows :

[*]This research was supported by DARPA under contract F30602-98-2-0144.

- We present an improvement over [9] concerned with the production of pipelined hardware corresponding to MATLAB loops. We include resource constraints and resource optimizations into the scheduling framework.
- We present an experimental evaluation of the different scheduling strategies and demonstrate the resource performance trade-off paradigm in the context of pipelining hardware on FPGAs.

2. OVERVIEW OF MATCH

The work presented in this paper is part of a MATLAB compiler for heterogeneous systems consisting of general purpose processors, embedded processors and FPGAs[11]. Our compiler takes the description of a system in MATLAB and partitions it into software to be executed on general purpose and embedded processors and hardware to be mapped to FPGAs. In this paper we address the issues involved in generating an efficient hardware once the front-end of the compiler has partitioned the system into hardware and software. In particular we focus on the pipelining framework that does dependency analysis on the input algorithm and produces a pipelined design if possible. Figure 1 shows an overview of the framework. First the input MATLAB code is parsed to construct an Abstract Syntax Tree (AST). Since MATLAB variables do not have any notion of type-shape, a compiler phase infers the types of the variables and the dimensions of the matrices. The AST is scalarized, where the operations on matrices are expanded out into loops. Next the AST is levelized, where complex expressions are broken down into simpler expressions containing at most three operands. The basic techniques for obtaining a hardware description from a MATLAB AST is discussed in [10]. A dependency analysis phase infers the control and data dependency present in the AST. Finally a pipelining phase analyzes the loops to check if pipelining is possible, determines the initiation rate and produces a corresponding pipelined hardware description in VHDL. The output register transfer level (RTL) VHDL code is then passed through commercial synthesis and physical design tools to generate a netlist and bit-stream to configure the FPGA.

2.1 Overview of the $WildChild^{TM}$ Architecture

Our compiler is designed to produce code for most current FPGA architectures. The compiler reads in a description of the FPGA board architecture and mechanisms to access the external memory which is used to produce the RTL VHDL for the particular FPGA board. Our current experimental results are on the $WildChild^{TM}$ FPGA board from Annapolis Micro Systems. It is a VME compatible board with eight *Xilinx* 4010 FPGAs and one *Xilinx* 4028 FPGA. In this paper we focus on our work regarding the generation of hardware for a single FPGA with an external memory. For that purpose we have used the *Xilinx* 4028 FPGA having 1024 CLBs. The memory connected to the FPGA is 32-bit wide and contains 2^{18} addressable locations.

3. OVERVIEW OF THE PIPELINING FRAMEWORK

Figure 2 shows an overview of the pipelining framework. Given a series of nested loops, the framework attempts to pipeline the innermost loop. If the loop body contains conditional statements, they are converted to predicated statements [9]. A dataflow graph is constructed from the loop body and a scheduling algorithm is applied. The schedule for the loop body is used to produce a pipeline schedule. The modulo variable expansion technique is applied to scalars that have live overlapping ranges [6]. Finally loop conditionals are added around the kernel of the pipeline schedule and VHDL code is generated. Details of the pipelining framework can be found in [9].

```
Input  :  Dependence graph of loop body G(V,E).
Output  :  A pipelined hardware, if possible.
Algorithm  :
M : Number of memory references in the loop body.
I : Initiation Rate.
MM : Next modulo memory number to be assigned.
CS : Current state number to be assigned.
IF reverse dependence edge then RETURN.
$M \leftarrow$ Number of memory references.
$I \leftarrow M$.
$MM \leftarrow 0, CS \leftarrow 0$
FOR all vertices $v \in V, state[v] \leftarrow \infty$
UNTIL $\exists$ vertex $v \in V$, s.t $state[v] = \infty$
   $state\_advanced \leftarrow false$
   FOR all vertices $v \in V$ not memory references
      IF( predecessor ready AND $state[v] = \infty$)
         $state[v] \leftarrow CS$
      ENDIF
   ENDFOR
   FOR all vertices $v \in V$ memory reference
      IF( predecessor ready AND $state[v] = \infty$)
         $state\_advanced \leftarrow true$
         IF( $CS\%M \geq MM$)
            $state[v] \leftarrow CS + M - (CS\%M - MM)$ ;
         ELSE
            $state[v] \leftarrow CS + MM - CS\%M$
         ENDIF;
         $MM \leftarrow MM + 1$
         $CS \leftarrow state[v] + 1$
      ENDIF
   ENDFOR
   IF( $state\_advanced = false$) $CS \leftarrow CS + 1$
ENDUNTIL
Produce code for prologue,epilogue and steady state
Calculate loop bounds
Produce VHDL code
```

Figure 3: Modified ASAP algorithm to find pipeline schedule of a loop body

In this paper we discuss various scheduling algorithms that are applied to the dataflow graph. These algorithms have different objectives producing various trade-offs between resources and performance.

4. ASAP SCHEDULING

The simplest of the algorithms is a modification of the ASAP (As Soon As Possible) algorithm [3]. The key assumption behind the algorithm is that the arrays present in the application are stored in an external memory, and there is a single memory port through which the external memory can be accessed. For most signal/image processing applications, the arrays involved are simply too large to fit on an on-chip memory module in the FPGA. Moreover, most current FPGA board architectures provide a single-port external memory. Thus, the assumption holds for most cases.

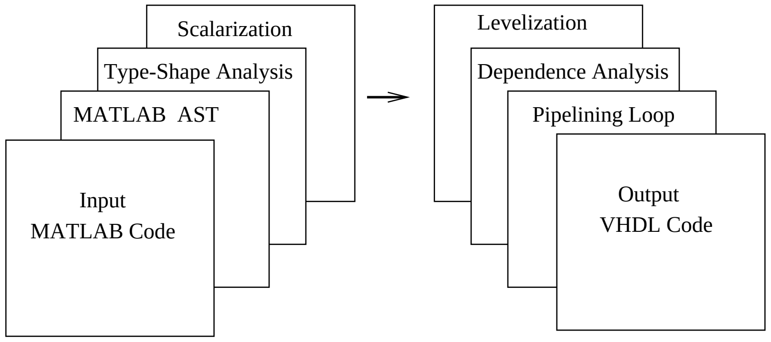


Figure 1: Overview of the synthesis framework.

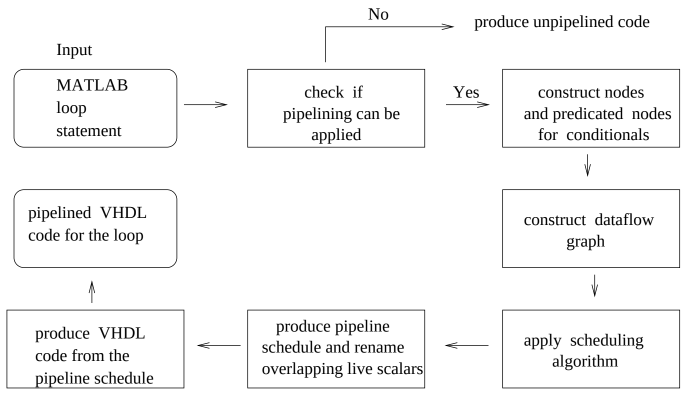

No
produce unpipelined code
Input
MATLAB loop statement
check if pipelining can be applied
Yes
construct nodes and predicated nodes for conditionals
construct dataflow graph
apply scheduling algorithm
produce pipeline schedule and rename overlapping live scalars
produce VHDL code from the pipeline schedule
pipelined VHDL code for the loop

Figure 2: Overview of the pipelining framework.

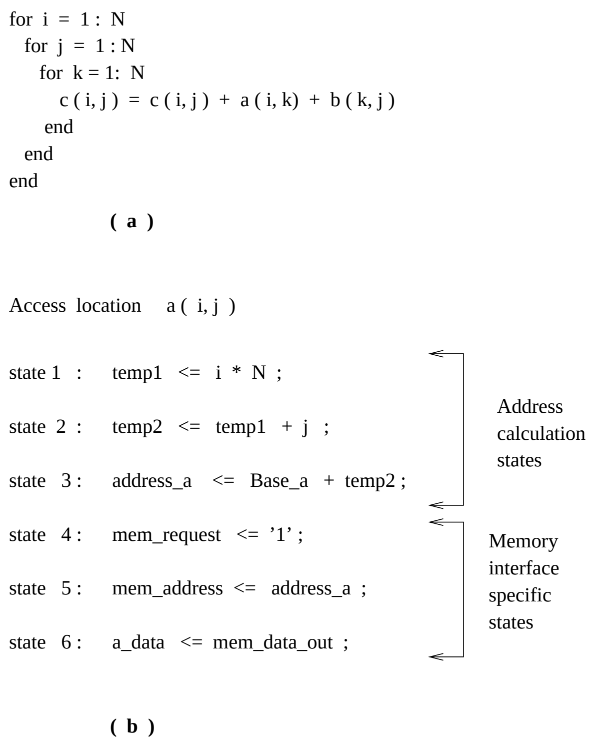
For the case where a single memory can be accessed via multiple ports, the problem can be handled by having multiple instances of the memory resource and applying a resource constrained scheduling as discussed in Section 6. In the case where multiple memories can be accessed, the problem can be dealt as a parallelization problem with distributed memory. For present considerations, as there is only one memory port, there can be only one memory access at any stage. Thus the schedule for the loop body is developed such that each memory access is placed in a state so that the state number modulo the number of memory accesses in the loop body is unique for each memory access. This ensures that two memory access never occur concurrently.

Since each vertex is visited once to assign the state and each edge is considered once for updating the dependencies, the complexity of the algorithm is $O(E+V)$. E is the number of edges in the dependency graph $G(V,E)$ and V is the number of vertices in the dependency graph. For the other algorithms presented in this paper we express the computational complexity in the same notation.

5. ALAP SCHEDULING

A shortcoming of the modified ASAP algorithm is that the resource usage towards the beginning of the schedule is considerably higher than the rest of the states. In other words, resource usage throughout the schedule is highly non-uniform. To understand the phenomenon consider a typical signal processing loop body, the matrix multiplication. The loop body has a single statement shown in Figure 4(a).

The basic data types in MATLAB are multi-dimensional



```
for i = 1 : N
  for j = 1 : N
    for k = 1: N
       c ( i, j ) = c ( i, j ) + a ( i, k) + b ( k, j )
     end
  end
end
```

(a)

```
Access location    a ( i, j )

state 1  :   temp1  <=  i * N ;

state 2  :   temp2  <=  temp1  + j  ;

state 3  :   address_a   <=  Base_a  + temp2 ;

state 4  :   mem_request  <=  '1' ;

state 5  :   mem_address  <=  address_a ;

state 6  :   a_data  <=  mem_data_out  ;
```

(b)

Figure 4: (a)Matrix multiplication code (b) Example of states generated for an array access a(i,j).

arrays (or matrices). Each of the array accesses corresponds to a memory access. For each memory access, first the address is computed and then appropriate signals applied to the memory interface (an abridged version is shown in Figure 4(b)). The bulk of the computation is not in the matrix multiplication, but in the computation of the addresses corresponding to the memory accesses. Each address computation requires two additions and one multiplication (without subexpression elimination optimization). As there are 4 memory accesses, address computation involves 8 additions and 4 multiplications. Since the address computations are not dependent on any data within the loop body, all of them get scheduled towards the beginning of the schedule creating an imbalance of resource utilization (Figure 5(a)).

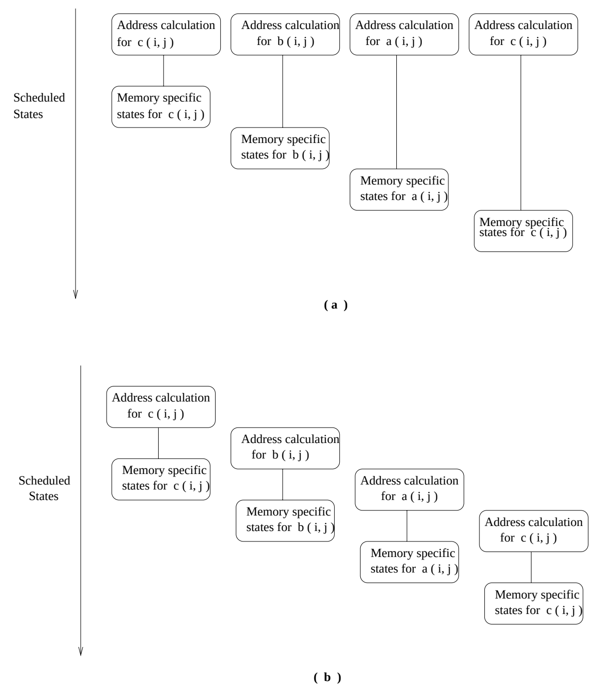

Scheduled States
Address calculation for c (i, j)
Address calculation for b (i, j)
Address calculation for a (i, j)
Address calculation for c (i, j)
Memory specific states for c (i, j)
Memory specific states for b (i, j)
Memory specific states for a (i, j)
Memory specific states for c (i, j)
(a)
Scheduled States
Address calculation for c (i, j)
Memory specific states for c (i, j)
Address calculation for b (i, j)
Memory specific states for b (i, j)
Address calculation for a (i, j)
Memory specific states for a (i, j)
Address calculation for c (i, j)
Memory specific states for c (i, j)
(b)

Figure 5: (a)Accumulation of resources in the beginning due to ASAP scheduling (b) ALAP schedule produces more uniform resource requirements.

The modified ALAP (As Late As Possible)[3] scheduling is motivated by the observation that the memory accesses are serialized. Hence, if the address computations are pushed as close to the memory accesses as possible, then the resource imbalance across the schedule will be alleviated. The modification of the ALAP algorithm is similar to the algorithm shown in Figure 3, with a few modifications made to schedule nodes in the reverse order. Initially, all nodes that do not have any successors are considered ready. A node becomes ready when all its successors are assigned some state. The initial state is assigned an arbitrarily large state number, and the states are assigned in a decreasing order. The key idea once again is to create a schedule such that memory accesses do not occur concurrently in the stable state of the pipeline. Figure 5(b) shows how resource requirements become uniform due to modified ALAP scheduling.

As in the case of modified ASAP scheduling, the complexity of the algorithm is $O(E+V)$.

```
Input  :  Dependence graph of loop body $G(V,E)$,
          Resource Vector $R[n_{res}]$.
Output  :  A Pipeline Schedule $\{S(V), II\}$.
Algorithm  :
$II \leftarrow 1$
WHILE( $true$ )
   CREATE Resource-Table RT of size $II \times n_{res}$
   FOR all vertices $v \in V, state[v] \leftarrow \infty$
   $S \leftarrow 0$
   UNTIL $\exists$ vertex $v \in V$, s.t $state[v] = \infty$
      $\acute{V} \leftarrow Ready\_Vertices\_ofG(V,E)$
      $\acute{R}[\acute{n}_{res}] \leftarrow Available\_Resources\_ofR[n_{res}]$
      FOR all resources $r \in \acute{R}$
         $v_{best} \leftarrow$ Best Vertex for resource $r$.
         $state[v] \leftarrow S$
         Update Ready Vertices
         Update Resource-Table $RT$
      ENDFOR
   IF Resource Used Up
      BREAK
   $S \leftarrow S+1$
   ENDUNTIL
IF $\exists$ vertex $v \in V$, s.t $state[v] = \infty$
   $II \leftarrow II+1$
ELSE
   RETURN $Schedulestates[V], II$
ENDIF
ENDWHILE
```

Figure 6: List scheduling algorithm to find pipeline schedule of a loop body

6. LIST SCHEDULING WITH RESOURCE CONSTRAINTS

All the earlier algorithms discussed produced optimal schedules with different amounts of resource usage. By optimal we mean a schedule with the lowest possible initiation rate. Such an approach is suitable when the applications are small and the design fits on the FPGA. However, for two key reasons we may want to devise an algorithm that produces a schedule given a predetermined number of resources, so that various resource versus performance options can be explored. The reasons are :

- The loop body under consideration may be large and complex, and hence the resources required by an optimal schedule are not available. In this case, we want to start with the resources required by the optimal schedule and decrease them incrementally to determine the point where the resource requirements are met without sacrificing too much performance.
- There are multiple loops and the combined resources required to produce an optimal schedule for all the loops is not available. In that case, we must be able to deallocate resources from non-critical loops and allocate them to critical loops and achieve a balance where performance is maximized.

Next, we present a list scheduling algorithm to produce a pipeline schedule under resource constraints. The list scheduling framework is well studied[3]. The critical step is to design the best match function, or to compute the priority of each node. The selection of the node by the best match function should reflect its criticality in the schedule.

Next, we discuss a couple of strategies we explored regarding the selection of the best node for a resource. Note that list scheduling for pipelines differ from conventional list scheduling as all the iterations of the loops are scheduled together in the pipelined version as opposed to scheduling the loop body for a particular iteration only in conventional list scheduling.

6.1 Maximum Successors Heuristic

A node can get scheduled only when all of its predecessors in the dependency graph have been scheduled. Thus, scheduling a node that is predecessor to a lot of nodes will in turn enable the scheduling of all its successors. This motivates our first selection heuristic, wherein given a set of nodes to be assigned to an available resource, we select the node with the maximum number of successors. The idea is to have as many nodes as possible in the ready state, so that each resource is utilized as much as possible. If we are able to schedule each resource at each step of the schedule to some node, then we can achieve an optimal schedule.

The complexity of the basic list scheduling algorithm is $O(E+V)$, as each vertex and edge is considered once. Computing the priority of each node also takes $O(E+V)$ as each vertex is visited and an edge is considered twice from the vertices it connects. Hence, total complexity is $O(E+V)$.

6.2 Longest Path to Sink Heuristic

Although the maximum successors heuristic gives an estimate of the criticality of a node in the schedule, it is very myopic in the sense that it takes into account immediate successors only. In our next heuristic we attempt to take into account successors beyond the immediate ones. The heuristic is motivated by the fact that the schedule without resource constraints is dominated by the longest path from the source to the sink of the dependency graph. For example consider Figure 7. While choosing between node A and B to schedule, if node A is chosen on the basis of larger number of successors, an inferior schedule will be produced as the schedule length is dominated by the path starting at node B. Thus, the length of the path from a node to the sink node gives an indication of its criticality. This is the basis of Hu's scheduling algorithm [7]and is at the core of many popular heuristics.

For computing the longest path to the sink from each node we can apply ALAP scheduling to the dependency graph. The difference between the state assigned to a node and the state assigned to the sink node will give the longest path to the sink. As the complexity of ALAP scheduling is $O(E+V)$, the total complexity of the algorithm is also $O(E+V)$.

6.3 Aggregated Resource Heuristic

The longest path in the dependency graph is the longest path in the schedule too only when there are no resource constraints. Consider Figure 8. Clearly, the longest path in the dependency graph is not the longest path in the schedule due to resource constraints, and hence a heuristic based on the longest path heuristic produces sub-optimal schedule. This motivates our next heuristic. Our attempt is to incorporate the fact that resource constraints may produce a dominant path in the schedule that is different from the longest path in the dependency graph. In our current heuristic, for each node we consider the subgraph of the dependency graph for which the node is the root (for example, see Figure 9).

We compute the total resource requirements of the sub-

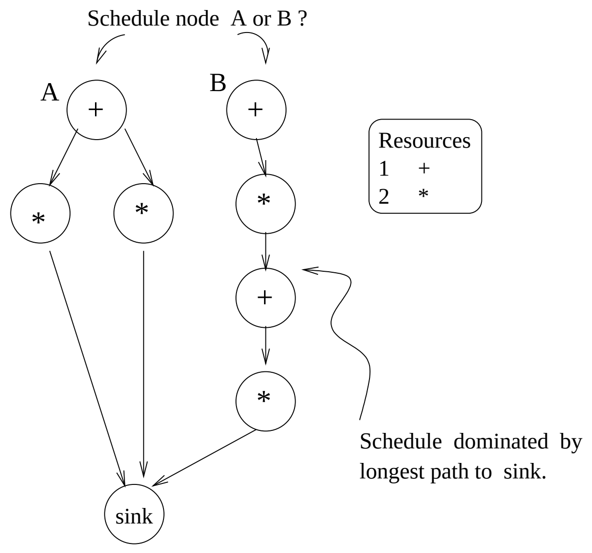


Figure 7: Maximum Successor vs Longest Path to Sink Heuristic.

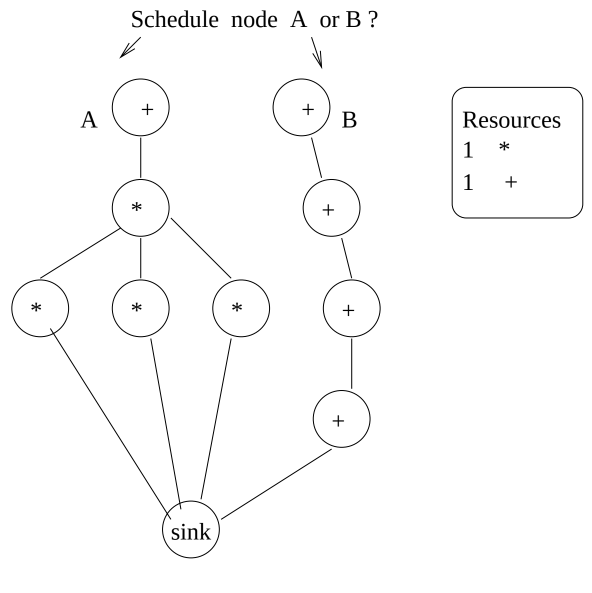


Figure 8: Longest Path to Sink vs Aggregated Resource Heuristic.

graphs. Then for each resource type required by the sub-graph, we divide it by the instances of each resource type available in the resource constraint. This gives a lower bound on the schedule length dictated by each resource type. We take the maximum of these lower bounds to indicate the criticality of the node. In Figure 9, the resource requirements to schedule the sub-graph with node A as root is four multipliers, whereas for the sub-graph with node B as root, we need three adders. Given a multiplier and an adder as the resource constraint, the estimate for number of states required to schedule the subgraph rooted at A is higher ($4/1 = 4$) than the subgraph rooted at node B ($3/1 = 3$). Hence, although the length of the path from node B is longer, node A gets scheduled first.

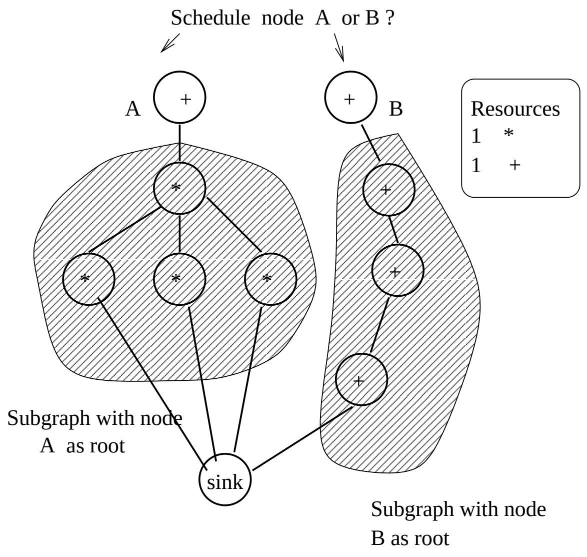


Figure 9: Computing the aggregated resource requirements for the sub-graph with the particular node as root.

Of course, we have neglected the fact that the sub-graph corresponding to two nodes may overlap and hence their schedules may interact. The resource estimation of the sub-graph is used just as a heuristic and not as a rigid allocation requirement, hence some inaccuracy is tolerable.

For computing the resource requirements of the sub-graph induced by each node, the dependency graph is first topologically sorted. The complexity of this operation is $O(E+V)$. The nodes are then considered in reverse order. For each node, if it has only one successor in the dependency graph, then the resource requirement of the sub-graph of the successor and resource requirement of the node are added together to obtain the resource requirement of the sub-graph for the node. If the node has multiple successors, then the set union of the resource requirements of each successor is computed and assigned as the resource requirement of the sub-graph for the node. For taking the union, each resource must have two pieces of associated information : (1) the node in the dependency graph corresponding to the resource and (2) the conditions that must be true for the resource to be used. Resources corresponding to the same node are counted only once. Resources with mutually exclusive conditions are also considered once. To accomplish this efficiently, the resources are maintained in sorted order corresponding to the topological sort. In that case, the worst case for each union is $O(V)$, as each node can get visited once while merging to get the union. Thus, the complexity of computing the resource requirements of the sub-graph for each node is $O(V^2)$. Since the list scheduling algorithm itself takes $O(E+V)$, the total complexity of the algorithm is $O(V^2)$. Note that control dependencies due to $if - then - else$ are already transformed to predicated statements before the application of this algorithm as described in [9].

7. EXPERIMENTAL RESULTS

In this section we present some experimental results. The benchmarks presented include matrix multiplication, FIR filter, sobel edge detection algorithm, average filter and a motion estimation algorithm. The matrix multiplication algorithm multiplies two input matrices. Matrix multiplication is at the core of many signal and image processing algorithms. FIR (Finite Impulse Response) filter is very important in the signal processing domain as it suggests a system that passes certain frequency components and rejects all other frequencies. Sobel edge detection algorithm takes an input image and performs a two dimensional linear convolution with 3×3 kernels. The average filter takes an input image and for each pixel of the image computes the average values of the pixels in the neighborhood. A comparison is then made between the pixel and the average value. The motion estimation algorithm is used to lower the bandwidth requirement in video transmission by comparing blocks of the current frame with blocks of the previous frame and selecting a best match. We have chosen these applications as benchmarks as they are representative of the applications that are most suitable for implementation in hardware.

For the benchmarks described above, we present two different sets of experimental results. First we present a comparison of the resource unconstrained algorithms (modified ASAP and ALAP). Since both algorithms produce schedules with optimal initiation rate, the execution times of the designs produced are similar. For comparison of the two algorithms, we present the resource utilization on the FPGAs by the designs produced for the benchmarks. Secondly, we compare different heuristics used in the list scheduling algorithm. For a given resource constraint, we compare the execution times of the designs produced corresponding to the different heuristics. We also present the resource utilization for designs produced. All the designs were mapped to a $Xilinx$ 4028 FPGA with an external memory as described in Section 2.1.

7.1 ASAP vs ALAP

Figure 10 shows the resource utilization for the benchmarks using the two scheduling algorithms to produce the pipeline schedule - modified ASAP and modified ALAP. The ratio of the CLB (configurable logic blocks) usage on the FPGA is shown. As can be seen, ALAP scheduling produces much better designs due to less resource utilization. In particular, the resource utilization for the sobel benchmark is six times less for the ALAP scheduling as compared to ASAP scheduling. This can be attributed to the fact that the sobel loop has twelve array accesses and the phenomenon discussed in Section 5 totally dominates. On the other hand for the average filter benchmark the difference between the ASAP and the ALAP algorithms is not significant as the inner loop in the benchmark is simple with only one array access. The performance of the designs shown in terms of execution times were identical. In general, ALAP scheduling limits unnecessary concurrency of operations resulting in improved resource requirements.

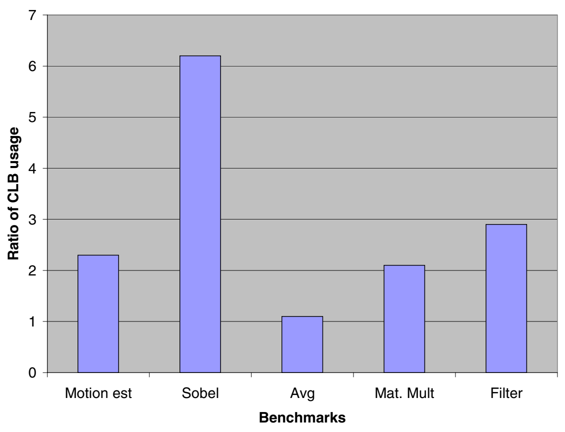


Figure 10: CLB usage of ASAP/ CLB usage of ALAP.

7.2 List Scheduling Heuristics

We compare the three heuristics : (1) the maximum successor (2) the longest path to sink and (3) the aggregated resource heuristic within the list scheduling framework for producing the pipeline schedule given some resource constraints. Figure 11 shows the execution time for the three heuristics corresponding to different resource constraints for the sobel filter benchmark. Figure 12 shows the same data for the motion estimation benchmark. By execution time we refer to the computation time taken by the design produced on the FPGA (excluding configuration and communication time with the board). The execution times are shown for different resource constraints such that sub-optimal schedules are produced. The resource constraints increase from left to right and execution time for the optimal design without any resource constraints is shown on the far right. As can be seen, under very tight resource constraints, the aggregated resource heuristic performs better than the other two heuristics. Although the performance of the heuristics are close to each other, maximum successor heuristics performs slightly worse than the other two heuristics. The reason behind the close performance of the heuristics may be attributed to the fact that the concurrency between the operations within the loop body are limited and the heuristics do not have much scope to differ. In particular, the parallelism for the applications is across the loop iterations and not within the loop body.

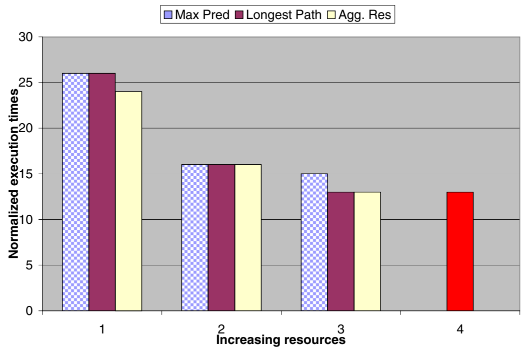


Figure 11: Normalized execution times (normalized to 5ms) for the sobel benchmark for the different list scheduling heuristics. The optimal execution time obtained by the ALAP scheduling is shown on the far right.

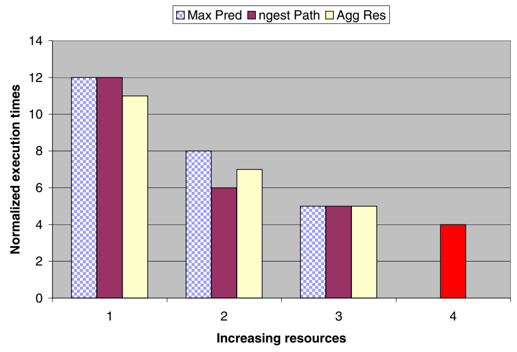


Figure 12: Normalized execution times (normalized to 5ms) for the motion estimation benchmark for the different list scheduling heuristics. The optimal execution time obtained by the ALAP scheduling is shown on the far right.

8. CONCLUSIONS AND FUTURE WORK

We presented a compiler to synthesize optimized hardware on FPGAs from applications described in MATLAB. We discussed a range of scheduling techniques to obtain pipelined hardware from loops present in the input application. Our future work leverages this framework to perform design space exploration. In particular we are focusing on three primary issues

- Integration of complex IP cores into the designs and modification to the scheduling techniques to accommodate them.
- Accurate predication techniques to predict the area/delay of designs given the intermediate structure synthesized by the compiler.
- An integrated global scheduling framework capable of optimizing the design across loops and function bodies to produce high performance design.

We hope these enhancements will enable us to achieve our goal of producing automated designs that are within a factor of two of manually optimized hardwares, cutting down the design time from weeks to minutes.

9. REFERENCES

[1] Peter J. Ashenden *The Designer's Guide To VHDL*, Morgan Kaufmann Publishers, Inc.

[2] Steven S. Muchnick *Compiler Design Implementation*, Morgan Kaufmann Publishers, Inc.

[3] Giovanni De Micheli *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, Inc.

[4] Villasenor, J., Mangione-Smith, W. H. *Configurable Computing*, Scientific American, June 1997, pp. 66-71

[5] V. H. Allan, R. B. Jones, R. M. Lee and S. J. Allan,*Software Pipelining*, ACM Computing Surveys, Vol.27,No.3, September 1995.

[6] M. Lam,*Software Pipelining: An Effective Scheduling Technique for VLIW Machines* , Proc. Programming Language Design and Implementation, June 1988.

[7] T. C. Hu,*Parallel Sequencing and Assembly Line Problems*,Operation Research No.9,1961.

[8] M. Weinhardt and W. Luk, *Pipeline Vectorization for Reconfigurable Systems* , Proc. Field-Programmable Custom Computing Machines, April 1999.

[9] M. Haldar, A. Nayak, A. Choudhary and P. Banerjee, *Automated Synthesis of Pipelined Designs on FPGAs for Signal and Image Processing Applications Described in MATLAB*, $submittedtoASP-DAC'$2001.

[10] M. Haldar, A. Nayak, A. Choudhary and P. Banerjee, *FPGA Hardware Synthesis from MATLAB*, 14th Intl. Conf. VLSI Design, Jan 2001.

[11] Prith Banerjee, Nagraj Shenoy, Alok Choudhary, Scott Hauck, Chris Bachmann, Malay Haldar, Pramod Joisha, Alex Jones, Abhay Kanhare, Anshuman Nayak, Suresh Periyacheri, Mike Walkden and David Zaretsky, *A MATLAB Compiler for Distributed, Heterogeneous, Reconfigurable Computing Systems* , Proc. IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM'00, April 2000.

[12] Ian Page *Constructing hardware-software systems from a single description*, Journal of VLSI Signal Processing, pp. 87-107, 1996

[13] J. Hammes, B. Rinker, W. Bohm and W. Najjar, *Cameron: High Level Language Compilation for Reconfigurable Systems*, Proc. Parallel Architectures and Compilation Techniques (PACT'99), October 1999.

[14] J. Babb, M. Rinard, C.A. Moritz, W. Lee, M. Frank, R. Barua, S. Amarasinghe *Parallelizing Applications into Silicon*, FCCM 1999

[15] Y. Li, T. Callahan, E. Darnel, R. Harr, U. Kurkure and J. Stockwood, *Hardware-Software Co-Design of Embedded Reconfigurable Arhitectures*, Proc. 37th DAC, June 2000.

[16] B. L. Hutchings and B. E. Nelson,*Using General-Purpose Programming Languages for FPGA Design*, Proc. 37th Design Automation Conference, June 2000.

[17] M. Gokhale, J. Stone, J. Arnold and M. Kalinowski, *Stream-Oriented FPGA Computing in the Streams-C High Level Language*, Proc. Field-Programmable Custom Computing Machines, April 2000.

[18] G. De Micheli, *Hardware Synthesis from C/C++ Models*, Proc. Design, Automation and Test in Europe Conference and Exhibition, March 1999.