A

A Constraint Programming Approach for Integrated Spatial and Temporal Scheduling for Clustered Architectures

MIRZA BEG, University of Waterloo
PETER VAN BEEK, University of Waterloo

Many embedded processors use clustering to scale up instruction level parallelism in a cost effective manner. In a clustered architecture, the registers and functional units are partitioned into smaller units and clusters communicate through register-to-register copy operations. Texas Instruments, for example, has a series of architectures for embedded processors which are clustered. Such an architecture places a heavier burden on the compiler, which must now assign instructions to clusters (spatial scheduling), assign instructions to cycles (temporal scheduling), and schedule copy operations to move data between clusters. We consider instruction scheduling of local blocks of code on clustered architectures to improve performance. Scheduling for space and time is known to be a hard problem. Previous work has proposed greedy approaches based on list scheduling to simultaneously perform spatial and temporal scheduling, and phased approaches based on first partitioning a block of code to do spatial assignment and then performing temporal scheduling. Greedy approaches risk making mistakes that are then costly to recover from and partitioning approaches suffer from the well-known phase ordering problem. In this paper, we present a constraint programming approach for scheduling instructions on clustered architectures. We employ a problem decomposition technique that solves spatial and temporal scheduling in an integrated manner. We analyze the effect of different hardware parameters—such as the number of clusters, issue-width and inter-cluster communication cost—on application performance. We found that our approach was able to achieve an improvement of up to 26%, on average, over a state-of-the-art technique on superblocks from SPEC 2000 benchmarks.

Categories and Subject Descriptors: C.2.2 [**Compiler Optimization**]: Parallelization

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Automatic parallelization, constraint programming

ACM Reference Format:
Beg, M., van Beek, P. 2011. A constraint programming approach for integrated spatial and temporal scheduling for clustered architectures. ACM Trans. Embedd. Comput. Syst. V, N, Article A (January YYYY), 23 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Optimizing code for embedded processors is becoming increasingly important because of their pervasive use in consumer electronics. For example, millions of cellular phones are powered by a member of the ARM11 processor family. Similar processors are widely used in consumer, home and embedded applications. Their low power and speed optimized designs (350MHz-1GHz) makes them feasible for mobile devices, media processing and real-time applications. Billions of the ARM processors are shipped each year by manufacturers [ARM 2011].

This work is supported by a grant from the Natural Sciences and Engineering Research Council of Canada.
Author's addresses: M. Beg (and) P. van Beek, Cheriton School of Computer Science, University of Waterloo.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

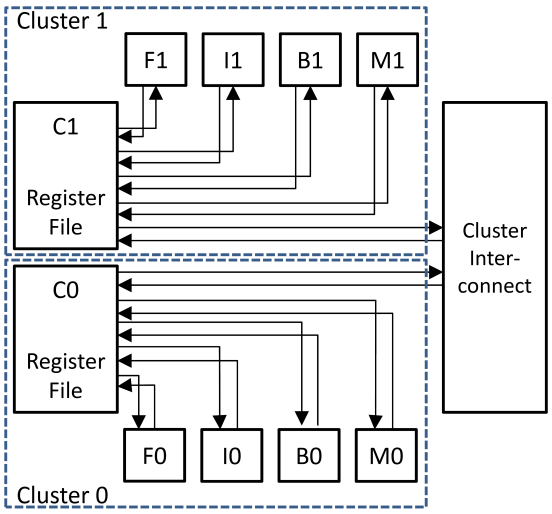

Fig. 1. Datapath model of a dual-cluster processor. The functional units are clustered into two identical sets each having a separate set of registers. In the given model communication between clusters is through an interconnect. (Adapted from [Fisher et al. 2005].)

With the increasing complexity of embedded processor designs, clustering has been proposed to organize the functional units on the processor (see Figure 1). A clustered architecture has more than one register file with a number of functional units associated with each register file called a cluster. Clusters are connected to each other with some interconnection topology [Fisher et al. 2005]. Among recent examples of clustered architectures are the Texas Instruments TMS320C6x family of DSPs [Texas Instruments 2011]. In particular, the TMS320C64x features two clusters with four functional units each and a 32×32 register file (32 registers, each of 32 bits). Data can be moved between two clusters through inter-cluster interconnect using an explicit copy operation.

A compiler for a clustered architecture is responsible for scheduling instructions to both time cycles (temporal scheduling) and clusters (spatial scheduling). The primary goal of scheduling on a clustered architecture is to identify parts of the program which can be executed concurrently on different clusters in the processor and exploit instruction level parallelism. Previous work has proposed heuristic approaches to partition straight-line regions of code for clustered architectures (see [Aleta et al. 2009] and the references therein; for some recent work, also see [Ellis 1986; Rich and Farrens 2000; Lee et al. 1998; 2002]). Chu et al. [2003] describe a hierarchical approach to find balanced partitions of a given dependence graph for a block which is a state-of-the art technique we compare against (see the Related Work section for details).

In this article, we present a constraint programming approach for spatial scheduling for clustered processors where clusters can communicate with each other using the cluster inter-connect with some non-zero cost. Our approach is robust and searches for an optimal solution. In a constraint programming approach, a problem is modeled by stating constraints on acceptable solutions, where a constraint defines a relation among variables, each taking a value in a given domain. The constraint model is usually solved using backtracking search. The novelty of our approach lies in the decomposition of the problem and our improvements to the constraint model in order to reduce the effort required to search for the optimal solution. Our approach is applicable when

larger compile times are acceptable. In contrast to previous work we assume a more realistic instruction set architecture containing non-fully pipelined and serializing instructions.

In our experiments we evaluated our approach on superblocks from the SPEC 2000 integer and floating-point benchmarks, using different clustered architectural configurations. We compared our results against the hierarchical partitioning scheme for spatial and temporal scheduling, RHOP [Chu et al. 2003]. We experimented with various inter-cluster communication costs from one to eight cycles to analyze the effects of inter-cluster communication on program performance. We found that in our experiments we were able to improve schedule costs of superblocks in the SPEC2000 benchmarks up to 26% on average over RHOP, depending on the architectural model. Also in our experiments we were able to solve a large percentage of blocks optimally with a ten minute time limit for each block. This represents a significant improvement over existing solutions. Furthermore, there is no current work that systematically evaluates the impact of communication cost on the amount of extractable parallelism.

The rest of this article is organized as follows. An overview of the background material is given in Section 2. Section 3 gives details of our approach and improvements to a basic constraint model. Section 4 describes the experimental setup, the results, and an analysis of the results. Section 5 gives an overview of related work. Finally, the article concludes with Section 6.

2. BACKGROUND

This section provides the necessary background required to understand the approach described in the rest of the article. It also gives a statement of the problem that this article solves along with the assumptions and the architectural model.

For the purposes of this article the following architectural model is assumed. We consider a clustered architecture, which has a small number of clusters and register values can be transferred between clusters over a fast interconnect using explicit move operations. In general, the following holds for our architecture model.

— Clusters are homogeneous. This means that all clusters have the same number of identical functional units and the same issue-width.
— The instruction set architecture is realistic in the sense that in addition to pipelined instructions, the instruction set contains non-pipelined instructions as well as serializing instructions. A serializing instruction needs the entire cluster in which it is issued in the cycle it is issued. Thus, both of these are instructions which may disrupt the instruction pipeline.
— Clusters can communicate with each other with a constant non-zero cost of c cycles. After the result of an instruction is available, it would take c cycles to transfer the resultant value to a different cluster where it is needed. We assume no limit on the inter-cluster communication bandwidth; i.e., the number of inter-cluster moves that can occur in a given cycle. Our interest is in smaller clustered architectures and as a result we assume that clusters are fully connected.

The assumptions given above are similar to those used to test RHOP [Chu et al. 2003] with the difference being that RHOP does not assume homogeneous clusters and does not consider non-pipelined or serializing instructions which are common features of realistic instruction set architectures. In addition RHOP has so far only been evaluated with an inter-cluster communication cost of one.

Communication between clusters is a well studied problem. Terechko and Corporaal [2007] present a comparative evaluation of five different techniques for inter-cluster communication including dedicated issue slots, extended operands, and multicast has been presented. Parcerisa et al. [2002] discuss an evaluation of various

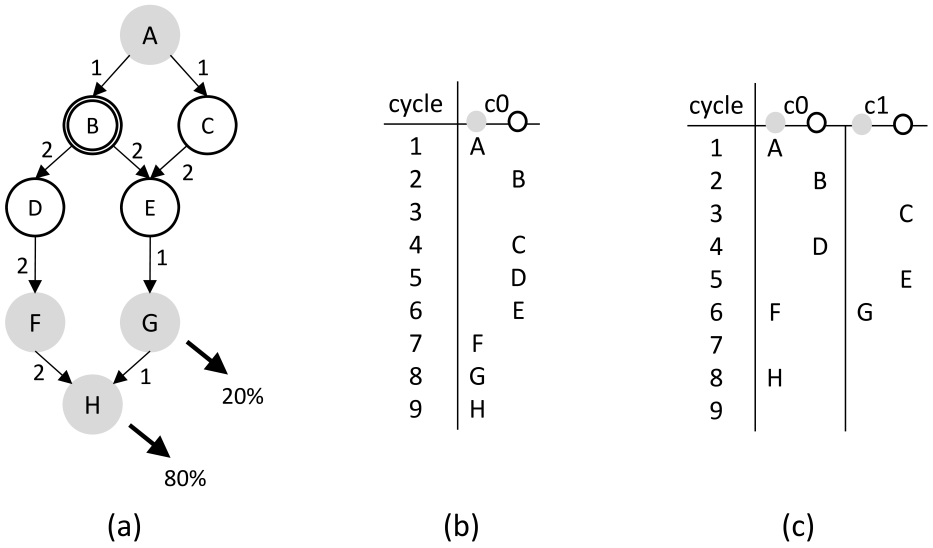

Fig. 2. (a) DAG representation of a superblock, where G and H are branch instructions with exit probabilities of 20% and 80% respectively. B is a serializing instruction and C is a non-pipelined instruction. (b) A possible schedule for the superblock given in (a) for a single-cluster which is dual-issue and has two functional units. One functional unit can execute clear instructions and the other can execute shaded instructions. The weighted completion time for the schedule is 8×0.2 + 9×0.8 = 8.8 cycles. (c) A possible schedule for the same superblock for a dual-cluster processor where the clusters can communicate with unit cost and each cluster is the same as the cluster in (b) The assignment of C, E and G to cluster $c1$ and the rest of the instructions to $c0$ results in a schedule with weighted cost of 6×0.2 + 8×0.8 = 7.6 cycles.

cluster-interconnect topologies including mesh, ring and bus interconnects and their variants. Aggarwal and Franklin [2005] examine hierarchical interconnects. The important item to note here is that, while inter-cluster communication is small on some popular architectures, it is not always negligible in practical clustered architectures.

Instruction scheduling is done on certain regions of a program. A *basic block* is a region of straight-line code with a single entry point and a single exit. A *superblock* is a sequence of instructions with a single entry point and multiple possible exits. We use the standard directed acyclic graph (DAG) representation for basic blocks and superblocks. Each vertex in the DAG corresponds to an instruction and there is an edge from vertex i to vertex j labeled with a non-negative integer $l(i, j)$ which represents the delay or $latency$ between when the instruction is issued and when the result is available for the other instructions on the same cluster. The *critical path distance* from a vertex i to vertex j in a DAG is the maximum sum of the latencies along any path from i to j. The *earliest start time* of a vertex i is a lower bound on the earliest cycle in which the instruction i can be scheduled. *Exit vertices* are special nodes in a DAG representing branch instructions in superblocks. Each exit vertex i is associated with a weight $w(i)$ representing the probability that the flow of control will leave the block through this exit point. These have been calculated through profiling. See Figure 2(a) for a DAG representing a superblock.

With the given architectural model and the dependency DAG for a basic block or a superblock, the spatial scheduling problem can be described as an optimization problem where each instruction has to be assigned to a clock cycle and also assigned to a cluster such that the latency and resource constraints are satisfied.

Table I. Notation.

k	number of clusters
c	cost of an inter-cluster move operation
$l(i,j)$	latency between instructions i and j
$cp(i,j)$	critical path distance from i to j
$w(i)$	exit probability of a node i in the superblock
$S(i)$	clock cycle in which i is issued
$A(i)$	cluster assignment for instruction i
x_i, y_i, z_{ij}	variables for defining the constraint model
$dom(v)$	domain of variable v

Definition 2.1 (*Temporal Schedule*). The temporal schedule S for a block is a mapping of each instruction in a DAG to a time cycle.

Definition 2.2 (*Weighted Completion Time*). The weighted completion time for a superblock schedule is given by the summation $\sum_{i=1}^{n} w(i)S(i)$, where n is the number of exit nodes, $w(i)$ is the weight of exit i and $S(i)$ is the clock cycle in which i is issued in a schedule.

Given the definition of weighted completion time, which applies to both basic blocks and superblocks, the spatial scheduling problem can be stated as follows. Here, it should be noted that basic blocks are special superblocks with a single exit, with the flow of control guaranteed to leave the block from the same instruction.

Definition 2.3 (*Spatial Schedule*). The spatial schedule for a superblock is an assignment A giving a mapping of each instruction in a DAG to a cluster.

Thus the purpose of spatial scheduling is to find a cluster assignment for each instruction in the block while minimizing the weighted completion time of the block.

Definition 2.4 (*Spatial and Temporal Scheduling*). Given the dependence graph $G = (V, E)$ for a superblock and the number of available clusters k in a given architectural model, the *spatial and temporal scheduling problem* is to find an assignment A and a schedule S for all vertices in the graph G such that $A(i) \in \{0, \ldots, k-1\}$ for each instruction i in the block and start time $S(i) \in \{1, \ldots, \infty\}$ that minimizes the weighted completion of the code block. The assignment and schedule must satisfy resource and communication constraints of the given architectural model.

Temporal scheduling on realistic multiple issue processors is known to be a hard problem and compilers use heuristic approaches to schedule instructions. On clustered architectures the compiler has an additional task of spatial scheduling, partitioning instructions across the available computing resources. The compiler has to carefully consider the tradeoffs between parallelism and locality because a small spatial mistake is more costly than a small temporal mistake. For example, if a critical instruction is scheduled one cycle late then only a single cycle is lost. But if the same is scheduled on a different cluster then multiple cycles may be lost from unnecessary communication delays and resource contention. The combination of spatial and temporal scheduling is a much harder problem than simple temporal scheduling alone. The main idea behind our approach is to partition the DAG and schedule each partition on a cluster. To overcome the well-known phase ordering problem, we backtrack over the possible partitions, searching for a partition that leads to an optimal schedule. The key to our approach is a set of techniques for speeding up the search sufficiently to make our approach useful in practice.

Definition 2.5 (*Balanced Graph Partitioning*). The balanced graph partitioning problem consists of splitting a graph G into k disjoint components of roughly equal size such that the number of edges between different components is minimized.

When $k = 2$, the problem is also referred to as the graph bisection problem. The balanced graph partitioning problem is known to be NP-hard for $k \geq 2$ [Andreev and Räcke 2004]. The spatial scheduling problem described above can be harder than balanced graph partitioning because the feasible partitions of the DAG can also be fewer than k (so it would need to consider solutions with number of partitions from 1 to k).

We use constraint programming to model and solve the integrated spatial and temporal scheduling problem. Constraint programming is a methodology for solving hard combinatorial problems, where a problem is modeled in terms of variables, values and constraints (see [Rossi et al. 2006]).

Definition 2.6 (*Constraint Model*). A constraint model consists of a finite set of variables $X = \{x_1, \cdots, x_n\}$, a finite domain of values $dom(x_i)$ that each variable $x_i \in X$ can take and a set of constraints $C = \{C_1, \cdots, C_m\}$ where each constraint is defined over a subset of variables in X. A solution to the constraint model is an assignment of a value to each variable in X such that all of the constraints in C are satisfied.

Once the problem has been modeled such that the variables along with their domains have been identified and the constraints specified, backtracking over the variables is employed to search for a solution. At every stage of the backtracking search, there is some current partial solution that the algorithm attempts to extend to a full solution by assigning a value to an uninstantiated variable. One of the keys behind the success of constraint programming is the idea of constraint propagation. During the backtracking search when a variable is assigned a value, the constraints are used to reduce the domains of the uninstantiated variables by ensuring that the values in their domains are consistent with the constraints.

3. CONSTRAINT PROGRAMMING APPROACH

In this section we present a constraint model for the spatial scheduling problem. Each instruction is represented by a node in the superblock DAG. Each node i in the graph is represented by two variables in the model, x_i and y_i. The variable $x_i \in \{1, \ldots, m\}$ is the temporal variable representing the cycle in which the instruction is to be issued. The upper-bound m to these variables can be calculated using a heuristic scheduling method for a single cluster. The variable $y_i \in \{0, \ldots, k-1\}$ is the spatial variable that identifies the cluster to which instruction i is to be assigned. The key is to scale up to large problem sizes. In developing an optimal solution to the spatial scheduling problem we have applied and adapted several techniques from the literature including symmetry breaking, branch and bound and structure based decomposition techniques. It should be noted here that spatial scheduling cannot be feasibly and reliably solved independently as it heavily relies on temporal scheduling to determine the cost of a given cluster assignment. This leads us to an integrated solution design.

The main technique is to solve the problem using a master-slave decomposition which preserves optimality. We model the spatial scheduling as master which solves multiple slave problems to schedule instructions for a given cluster assignment. The master problem determines the assignment to the y variables (i.e. the cluster assignment to each instruction) and the slave problem schedules each instruction to a time cycle.

Example 3.1 (*Example Basic Block*). Figure 3(a) shows a simple dependency DAG for a basic block. The search tree for a simple constraint model for a 4-cluster architecture is shown in Figure 3(b) where the assignment of each instruction to a cluster

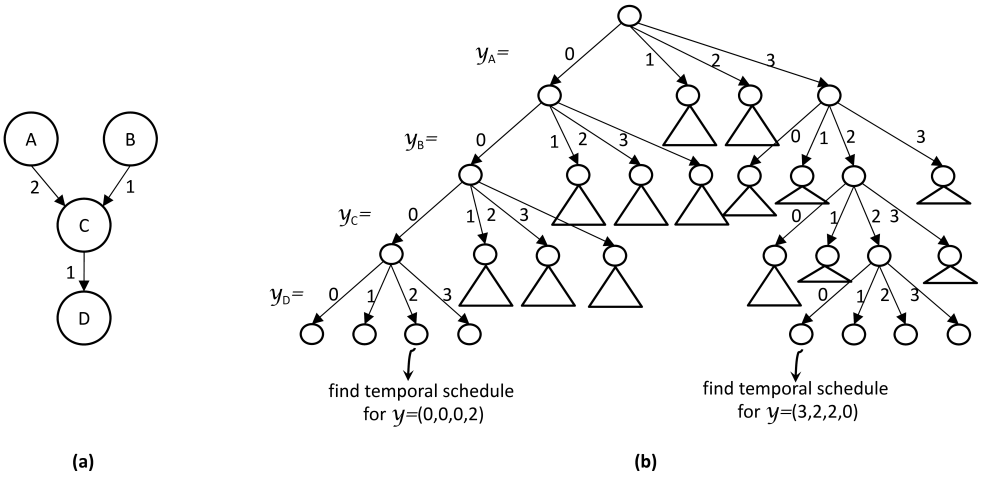

Fig. 3. (a) Example basic block. (b) Search tree for the simple constraint model associated with the basic block.

is determined at the leaf nodes and the optimal scheduler is used to calculate the temporal schedule for the given assignment. We use this as our running example.

The design was inspired by Benders [1962] and Dantzig-Wolfe [1960] decomposition techniques in integer programming, where an integer program is decomposed into a master-slave problem and the master problem generates many subproblems (or slave problems) which are solved hierarchically.

3.1. Symmetry Breaking

Symmetry can be exploited to reduce the amount of search needed to solve the problem. Backtracking over symmetric states does not improve the solution and consumes valuable computation time. If the search algorithm is repeatedly visiting similar states then recognizing and excluding equivalent states can significantly reduce the size of the search space. Using the technique of symmetry breaking, we aim to remove provably symmetric assignments to instructions. An example of symmetry breaking would be assigning the first instruction to the first cluster and thus discarding all the solutions where the first instruction is on any other cluster. This guarantees the preservation of at least one optimal assignment.

Our approach to symmetry breaking is to reformulate the problem such that it has a reduced amount of symmetry. We model the problem such that each edge (v_i, v_j) in the DAG is represented by a variable $z_{ij} \in \{=, \neq\}$. Our model inherently breaks symmetry by using backtracking search to assign values to the z variables, which represent the edges in the blocks. For a variable z_{ij}, assigning a value of $=$ means that variables y_i and y_j must take the same value and assigning a value of $\neq$ means that y_i and y_j must take different values.

Example 3.2 (*Improved Model for Running Example*). Consider the basic block of our running example given in Figure 3. The search tree for the improved model for the example is shown in Figure 4.

The improved model reduces the size of the search tree significantly. It should be noted here that the improved model holds for an architecture where the inter-cluster communication cost is the same for each pair of clusters. This results in the equiva-

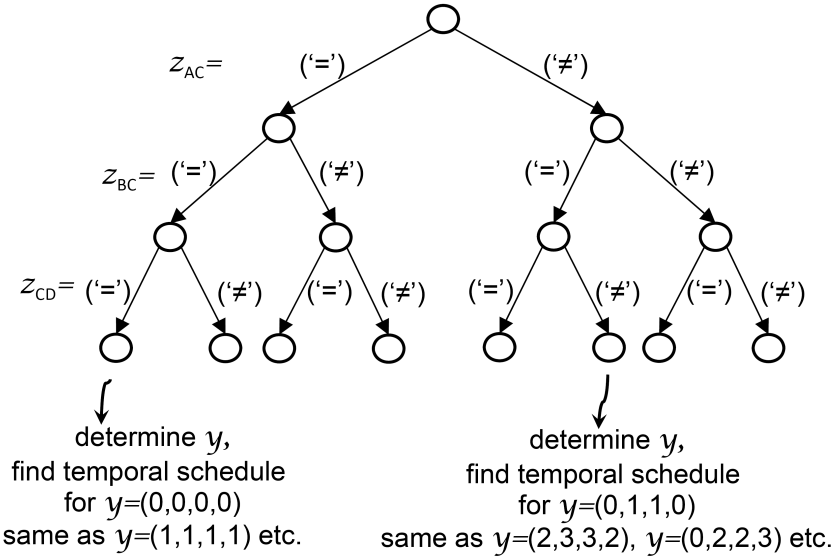

Fig. 4. Search tree of the improved constraint model.

lence of spatial schedules such as $y = (2, 3, 3, 2)$ and $y = (0, 2, 2, 3)$ where it does not matter if instructions A and D are assigned to the same cluster or not because there is no direct dependency between A and D and hence no constraint between variables y_A and y_D.

Once the variables $z_{ij} \in \{=, \neq\}$ are set, an assignment to all instructions can be determined, i.e. values can be assigned to all variables y_i for $i \in \{1, .., n\}$. Once an assignment to all instructions is available, an existing optimal temporal scheduler [Malik et al. 2008] is used to compute the best weighted completion time for the block for the given cluster assignment. The backtracking algorithm continues exhaustively, updating the minimum cost as it searches the solution space. In the case where an assignment is not possible for the given values of z variables, a conflict is detected (see Figure 5).

3.2. Branch and Bound

During the search for a solution, the backtracking algorithm can determine a complete assignment at the leaf nodes of the search tree. But certain branches of the search tree can be pruned if it can be guaranteed that all of the leaf nodes in that branch can be safely eliminated without eliminating at least one optimal solution. There are two cases in which an internal node of the search tree can be labeled as such.

1. The first case is where an assignment to the y variables is not possible for the partial assignment to the z variables. This can be detected if even one of the y variables cannot be assigned a value in $\{0, ..., k-1\}$ without violating the constraints given by the z variables. An example of such a violation is given in Figure 5. To discover such violations early in the search, the z variables are assigned in a fixed order that corresponds to a breadth-first traversal of the DAG.
2. The second case is where the partial assignment to the y variables can be proven to result in a temporal schedule with a cost greater than the established upper bound and any assignment that contains the given subset of cluster assignment cannot result in a better schedule. The search space can be reduced by eliminating all

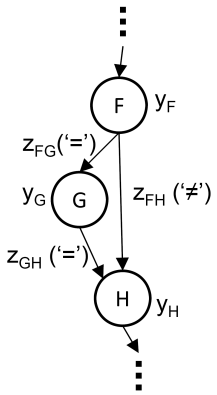

Fig. 5. An example of inconsistent assignment to z variables for which valid values cannot be assigned to the y variables.

such assignments containing this sub-assignment. Also note that the upper bound is gradually improved upon as better solutions are found.

In both the above mentioned cases the backtracking algorithm does not descend further in the search tree. This is done continuously during the algorithm as upper-bounds are improved upon.

3.3. Connected Sub-structures

The amount of search done by the algorithm can be reduced if it can be pre-determined that certain instructions are connected and would be assigned to the same cluster in at least one optimal solution to the assignment problem. To this end we define a connected sub-structure as follows.

Definition 3.3 (*Connected Sub-structure*). A connected sub-structure of a dependency DAG is a set of instructions with the properties: (i) there is at most a single instruction in the set with external incoming dependency edges; (ii) there is at most a single instruction in the set with external outgoing dependency edges; and (iii) the set of instructions can be scheduled on a single cluster such that the latency and resource constraints are satisfied and each instruction can be scheduled at its earliest start time.

The definition implies that the given set of instructions, if considered separately, cannot have a better schedule even if there are more functional units in the cluster or if there are more clusters. Some examples of connected sub-structures are given in Figure 6. For example, in Figure 6(a) the two connected sub-structures in the block are identified with boxes. A *chain* is a totally ordered set of three or more instructions in the dependency DAG. It should be noted that to preserve optimality we consider the pre-assignment chain optimization only if the number of concurrent chains is less than the number of available clusters.

LEMMA 3.4. *A chain is a connected sub-structure in our restricted architectural models.*

Proof: As a chain consists of a set of totally ordered instructions, the second instruction in the chain cannot be executed until the result of the first is available. Similarly, the third instruction cannot begin execution until the second instruction has completed

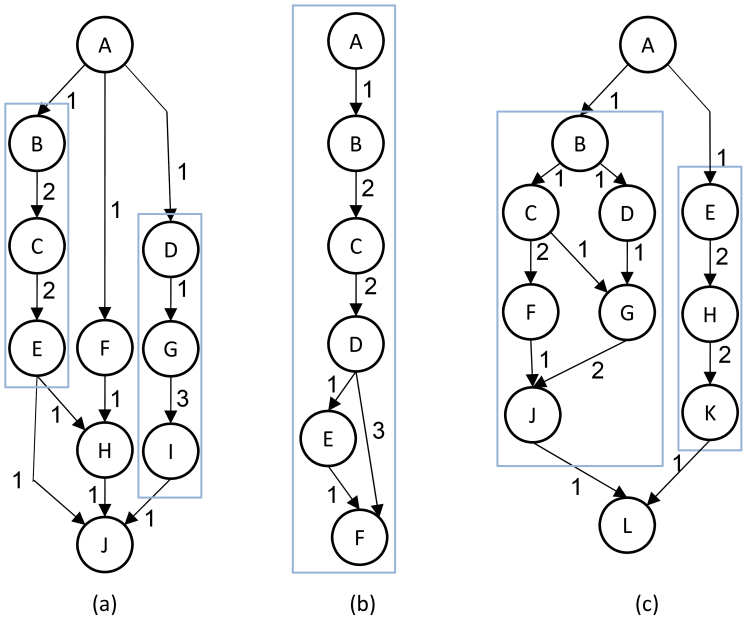

Fig. 6. Examples of connected sub-structures in blocks. Each of the connected sub-structures is marked by bounding box. Chains like the ones given in (a) and (b) form connected sub-structures in all architectures where as complex connected sub-structures may also exist like in (c) where the connectedness is conditional upon the types of instructions and the architectural configuration for which the code is being compiled.

execution and so forth. The simplest architectural model is a single issue clustered architecture having a single functional unit. The instructions in the chain can be executed on this functional unit one by one. Now consider that the number of functional units on this cluster are increased along with the issue-width. There is no better schedule for the chain compared to the previous architecture since there is no instruction level parallelism (ILP) that can be exploited by extending the architecture. Similarly, if we increase the number of clusters, there is no more ILP that can be exploited by the additional clusters. Hence the number of cycles required to execute the chain remain the same regardless of the architectural model. □

THEOREM 3.5. *Given a superblock with one or more chains, if the number of chains is less than or equal to the number of available clusters, the instructions within each chain can be assigned to the same cluster without eliminating at least one optimal assignment of the instructions to clusters.*

Proof: There are two scenarios to consider. The first scenario is where a chain contains one or more exits. An assignment of any two instructions in this chain to different clusters could increase the weighted completion time associated with the exits in the chain. (To see this, consider the block in Figure 6b. Scheduling the instructions B and C on different clusters would add the communication latency to the critical path, hence delaying the final exit and increasing the weighted completion time.) On the other hand a distributed assignment would not improve the weighted completion time and therefore could eliminate an optimal assignment. Hence, any optimal solution will assign the instructions in the connected sub-structure to the same cluster. The second scenario is where the sub-structure does not contain any exit instruction. There will exist a path from the last instruction to an exit in the superblock. From the proof in

ALGORITHM 1: Spatial Scheduling

Input: DAG G, an architectural model.
Output: The spatial and temporal schedule of G.
1 Construct constraint model for edge assignment;
2 $U \leftarrow$ Establish upper bound using list-scheduler extension;
3 $L \leftarrow$ Establish lower bound using optimal scheduler;
4 $E \leftarrow$ Edges in G with domain$\{=, \neq\}$;
5 Identify connected sub-structures and set edges to $\{=\}$;
6 // start backtracking on the first edge;
7 backtrack($E[0], U, L$);
8 **return** schedule and assignment given by U;

the previous scenario any distributed assignment of the instructions in the connected sub-structure will possibly delay the last instruction and hence increase the weighted cost of the subsequent exit.

□

For the purpose of the experiments reported in this article we only consider chains as connected sub-structures.

3.4. Solving an Instance

Given an architectural model which consists of the number of clusters k, the communication cost c, the issue width, and the number and type of the functional units, solving an instance of the spatial scheduling problem proceeds with the following steps (see Algorithm 1). First, a constraint model for edge assignment is constructed. The lower-bound and the upper-bound on the cost of the schedule on the given number of clusters is established. The lower bound is computed using the optimal temporal scheduler [Malik et al. 2008]. To compute the lower-bound for the given clustered architectural model, we schedule for a simpler architecture that has no serializing instructions and a single cluster. The single cluster has the same total number and types of functional units as all of the clusters in the given architectural model combined. Effectively this simulates a communication cost of zero between clusters and gives us a lower bound on the true cost of the schedule. The upper-bound is initially established using an extension to the list-scheduling algorithm. The extension to the list scheduler consists of a fast greedy heuristic to assign superblock instructions to clusters. The algorithm greedily assigns instructions to clusters as soon as the dependency, resource and communication constraints are satisfied. The lower and upper bounds are passed on to the backtracking algorithm along with the constraint model.

The backtracking search interleaves propagation of branch and bound checks with branching on the edge variables (see Algorithm 2). During constraint propagation the validity check of an assignment at each search node is enforced. Once a complete assignment can be computed, it is passed on to the optimal instruction scheduler to determine the cost of the block (line 7). The optimal scheduler computes the cost of the schedule using an extended constraint model of the problem considering the cost of inter-cluster communication. If the schedule cost is equal to the lower-bound then an optimal solution has been found. On the other hand if the cost is better than the existing upper-bound, the upper-bound as well as the upper-bound assignment is updated. This is repeated, until the search completes. The returned solution is the final upper-bound assignment. If the algorithm terminates, a provably optimal solution has been found. If, instead, the time limit is exceeded, the existing upper-bound solution is returned as the best result. Consistency check (line 3), which examines the search node for the first case in sub-section 3.2 and bounds check (line 12) are intended to prune the

ALGORITHM 2: Backtrack

Input: $E[i]$ the current edge, architectural model, an upper bound on the schedule cost U, and a lower bound on the schedule cost L.
Output: Spatial and temporal schedule associated with U.

```
1  for all values that can be assigned to the current edge do
2      n ← search node corresponding to the current assignment of variables;
3      consistency_check( n );
4      if n is a leaf node of search tree then
5          if n is consistent then
6              A ← generate assignment for n;
7              S ← determine schedule for assignment A;
8              U ← update ( U ) using S;
9          end
10     end
11     if n is an internal node of search tree then
12         bounds_check( n ) ;
13         if n is consistent and within bounds then
14             // continue onto the next edge;
15             backtrack( E[i + 1], U, L );
16         end
17     end
18     if U = L then
19         return A, S for U as solution;
20     end
21 end
22 return A, S for U as solution;
```

search tree and save search time. The following step-by-step execution on the running example provides a better description of the algorithms.

Example 3.6 (*Solving the Running Example*). Consider the basic block DAG from our running example given in Figure 3(a) on a 4-cluster 1-issue architecture with inter-cluster communication cost of 1. Spatial scheduling (Algorithm 1) proceeds by creating a constraint model. Determining the upper bound U on the schedule length yields 4 (i.e. $U \leftarrow 4$) and lower bound (L) is determined to be 3 ($L \leftarrow 3$). Since there are no connected sub-structures in the DAG, the algorithm proceeds by backtracking on the edges (AC, BC and CD). Algorithm 2 iteratively assigns $\{=, \neq\}$ to the z variables. For example, initially it z_{AC} is assigned $=$. Then a consistency check is run to make sure that it is possible to assign the y variables valid values if $z_{AC} \leftarrow =$ constraint is added to the model. This corresponds to the first case in subsection 3.2. Since the current search node (n) is an internal node of the search tree (corresponding to the left child of the root in the search tree shown in Figure 4) the second condition starting at line 11 is executed. It runs a bounds check on n which computes a lower bound for a partial assignment (which is 2) where $y_A = y_C$ making sure that it does not exceed U. Backtracking continues recursively on the edges. Consider the search node where $\{z_{AC} \leftarrow =, z_{BC} \leftarrow \neq, z_{CD} \leftarrow =\}$. The algorithm finds it to be consistent, generates an assignment $(0, 1, 0, 0)$, and determines the optimal schedule for the given spatial assignment (lines 6, 7). The optimal scheduler uses an extended model with the inter-cluster communication constraints for instructions which are scheduled on different clusters. The condition on line 18 determines that since $U = L$ the assignment is an optimal solution and hence returns it without searching the entire tree.

4. PERFORMANCE EVALUATION

In this section, we present an empirical evaluation of our scheduler for clustered architectures.

4.1. Experimental Setup

We evaluated our integrated solution to the spatial and temporal scheduling problem on superblocks from the SPEC 2000 integer and floating point benchmarks. Our approach works just as well for the basic blocks, but we present only the results for superblocks as these consistently show better improvements than the basic blocks. The benchmark suite consists of source code for software packages chosen to represent a variety of programming languages and types of applications. The results given in this article are for superblocks. The benchmarks were compiled using the IBM Tobey compiler [Blainey 1994] targeted towards the PowerPC processor [Hoxey et al. 1996], and the superblocks were captured as they were passed to Tobeys instruction scheduler. The compiler also marks serializing instructions and non-pipelined instructions. Here, it is worth noting that on the PowerPC, for example, 15% of the instructions in the superblocks are serializing instructions.

Table II. Architectural models and their composition in terms of the number and types of functional units.

issue width	integer units	memory units	branch units	floating point units
1-issue	1			
2-issue	1	1	1	1
4-issue	2	1	1	1

The compilations were done using Tobey's highest level of optimization, which includes aggressive optimization techniques such as software pipelining and loop unrolling. The Tobey compiler performs instruction scheduling once before global register allocation and once again afterward. Spatial scheduling is performed on the superblocks after register allocation. The results given are for the most frequently executed superblocks in the benchmarks but previous experiments have shown that the overall result of experiments remain the same in general. In the experiments we present our results relative to a baseline configuration which is an architecture with a single cluster having the same number of functional units and same issue width as a single cluster in the multi-cluster configuration being experimented with, as in [Faraboschi et al. 1998]. We use this baseline in order to remain consistent with the presentation of data from the experiments.

We compare against two versions of the RHOP implementation; the first using the regular list scheduler for scheduling (rhop-ls), as in Trimaran, and the second using the optimal scheduler (rhop-opt) also being used by our algorithm.

We conducted our evaluation using the three architectural models for each cluster shown in Table II. We experimented with 2-8 fully connected homogeneous clusters [Terechko 2007] with issue widths ranging from 1 to 4 on each cluster. In these architectures, the functional units are not fully pipelined, the issue width of the cluster is not always equal to the number of functional units, and there are serializing instructions. We assume homogeneous clusters; i.e., all clusters have exactly the same number and type of functional units. Additionally we also assume that clusters can communicate with each other with a non-zero latency. In our model, communication between clusters happens via an inter-cluster interconnect which is an explicit copy operation. A realistic bus model has a 4-cycle latency on a four cluster and 6-cycles on

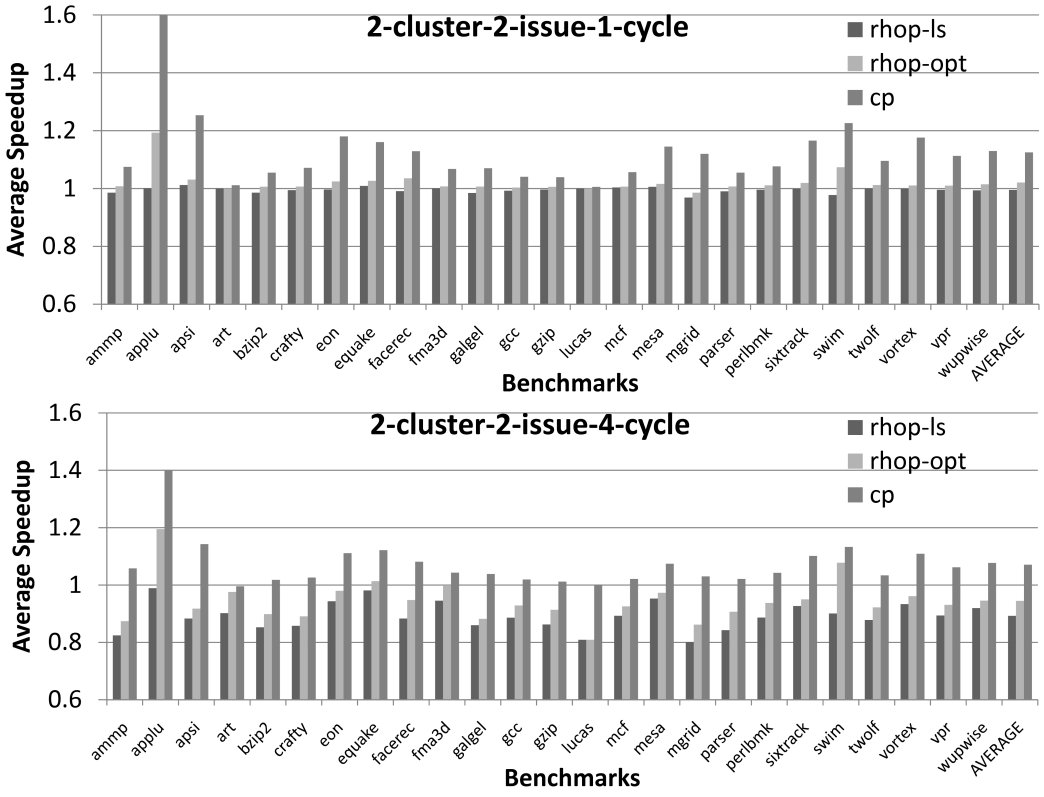

Fig. 7. Average speedups of superblocks in SPEC 2000 for a 2-cluster 2-issue architecture with inter-cluster communication cost of one and four cycles respectively. Note the non-zero origin.

an eight cluster processor [Parcerisa et al. 2002]. We also study the impact of various communication latencies on performance.

4.2. Performance Results & Analysis

In this section we present the results of our experiments. We structure the presentation of the results and our analysis as follows. First, we perform a general comparison of our constraint-programming-based integrated spatial and temporal scheduler, referred to as cp, with two versions of RHOP: RHOP using the regular list scheduler for scheduling (rhop-ls) and RHOP using the optimal instruction scheduler (rhop-opt), which is also being used by our algorithm. Second, we perform a detailed comparison that examines the impact of the number of clusters on the performance of the algorithms. Third, we perform a detailed comparison that examines the impact of the communication cost due to the different cluster-interconnect topologies on the performance of the algorithms. Finally, as our integrated scheduler is more costly in terms of scheduling time, we examine the time taken to schedule the superblocks in the various benchmarks. The speedups given are reductions in cycle count improvements over the baseline.

A general comparison of our integrated spatial and temporal scheduler with RHOP. In Figures 7 and 8 we present detailed performance results for for the constraint programming algorithm cp as compared to the two flavors of the RHOP algorithm, rhop-ls and rhop-opt. Note that rhop-ls is the original approach presented in [Chu et al.

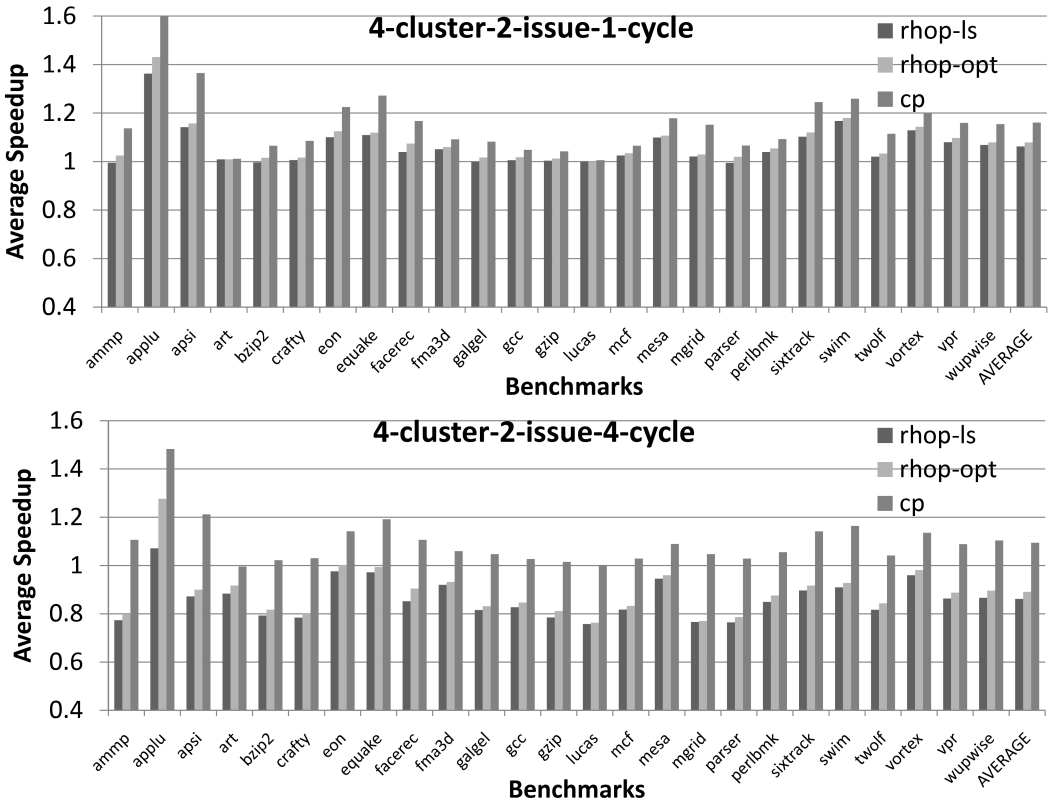

Fig. 8. Average speedups of superblocks in SPEC 2000 for a 4-cluster 2-issue architecture with inter-cluster communication cost of one and four cycles respectively. Note the non-zero origin.

2003] and that our results for rhop-ls closely match the experimental results presented therein. We include rhop-opt to factor out the contribution of the optimal instruction scheduler and examine the contribution of our partitioning scheme in improving performance. We compare the algorithms on a 2-cluster-2-issue architecture and a 4-cluster-2-issue. In our experiments cp always performs better than rhop-opt which in turn always performs better than rhop-ls. It can also be noted that the speedups from cp never fall below 1.0—i.e., cp never results in a slowdown over the baseline—whereas RHOP often results in slowdowns.

Consider the 2-cluster configurations (Figure 7). On the benchmark applu, our cp approach attains a speedup of 60% compared to 20% for rhop-opt when the inter-cluster communication cost is one cycle, a performance gap of 40%, and our cp approach attains a speedup of 40% compared to 20% for rhop-opt when the inter-cluster communication cost is four cycles, a performance gap of 20%. On average across all 26 benchmarks the performance gap between cp and rhop-opt is close to 15% when the communication cost is one cycle and approximately 10% when the communication cost is four cycles.

Consider next the 4-cluster configurations (Figure 8). On the benchmark ammp, our cp approach attains a speedup of 15% compared to 2% for rhop-opt when the inter-cluster communication cost is one cycle, a performance gap of 13%, and our cp approach attains a speedup of 10% compared to -20% for rhop-opt when the inter-cluster communication cost is four cycles, a performance gap of 30%. On average across all 26 benchmarks the performance gap between cp and rhop-opt is close to 7% when the

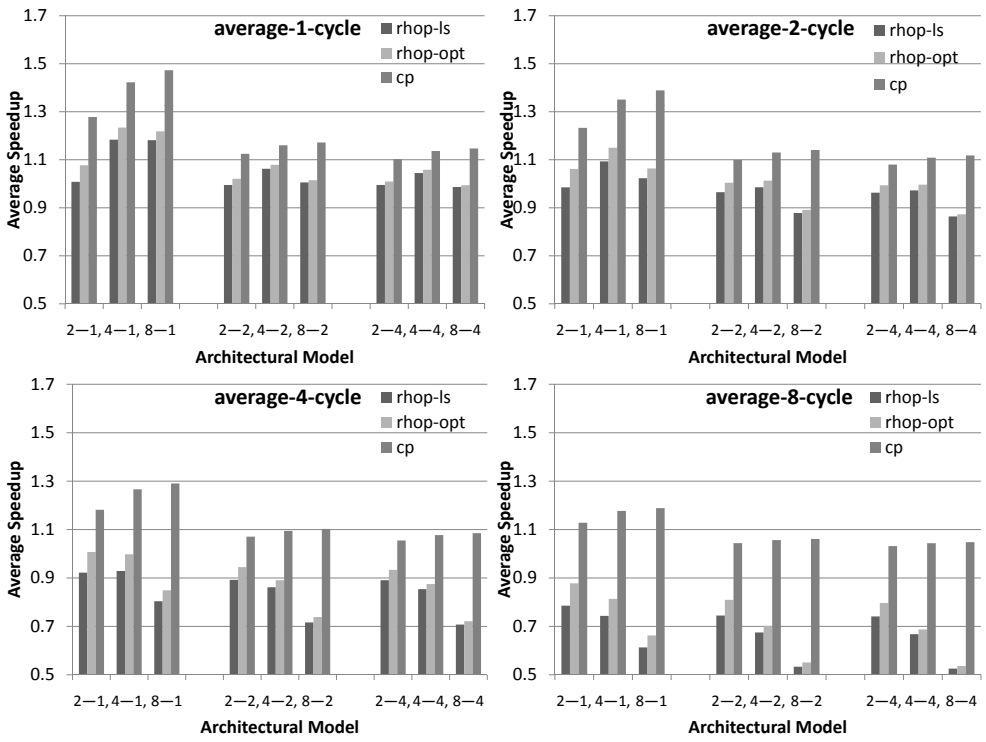

Fig. 9. Average speedups of superblocks in SPEC 2000 for a different architectures with inter-cluster communication cost of one, two, four and eight cycles respectively. Note the non-zero origin. On the x axis $\alpha - \beta$ means α clusters, $\alpha = 2, 4, 8$, and issue width of β, $\beta = 1, 2, 4$.

communication cost is one cycle and approximately 12% when the communication cost is four cycles.

The impact of the number of clusters on the performance of the algorithms. We examine the scalability of the algorithms as the number of clusters increases. Figure 9 presents the average improvements over all the benchmarks for various architectural configurations with inter-cluster communication latency varying from one to eight cycles. In general, in our experiments as the number of clusters increases the performance gap between cp and rhop-ls and rhop-opt increases. As well, the speedups for cp increases with the number of clusters whereas the speedups of rhop-ls and rhop-opt decreases as the number of clusters increase.

Consider the configurations where the communication cost is four cycles (see Figure 9, bottom left). On the architectures with an issue width of one, as the number of clusters α = 2, 4, 8 increases—i.e., architectural models 2–1, 4–1, and 8–1—the performance gap of our cp approach over rhop-opt increases from approximately 10% to more than 40%. As well, as the number of clusters increases, cp achieves increasing speedups over the baseline, whereas both rhop-opt and rhop-ls decrease in performance. In general, similar observations can be made for the architectures with larger issue widths and for the architectures with different communication costs.

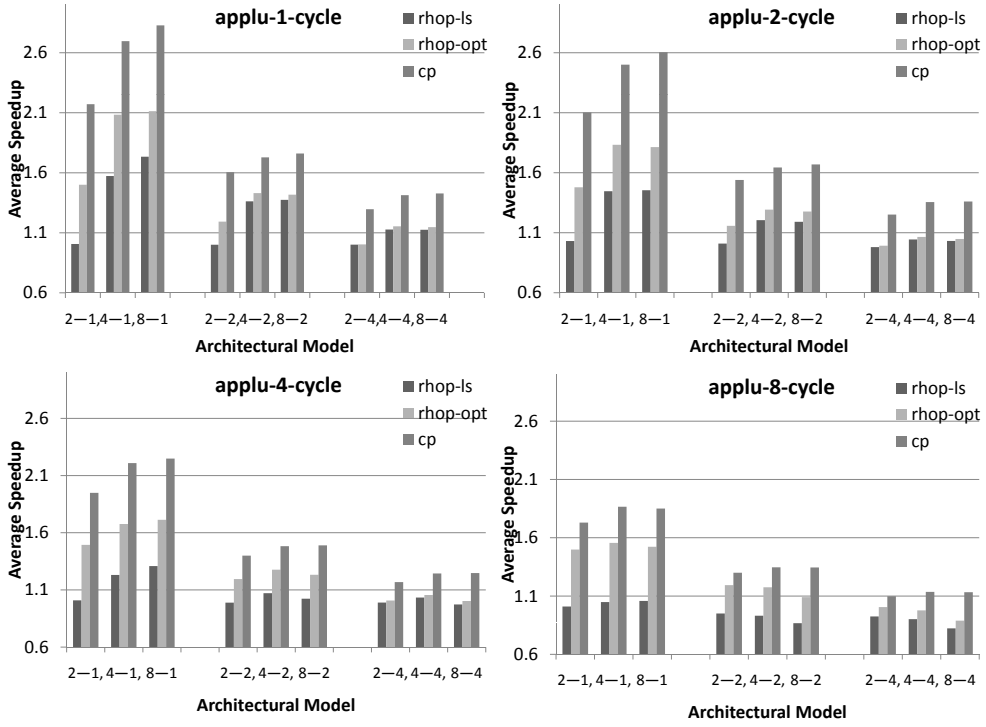

Fig. 10. Average speedups of superblocks for the applu benchmark for different architectures with inter-cluster communication cost of one, two, four and eight respectively. On the x axis $\alpha - \beta$ means α clusters, $\alpha = 2, 4, 8$, and issue width of β, $\beta = 1, 2, 4$.

The impact of the inter-cluster communication cost on the performance of the algorithms. Figure 9 also presents some results on what performance improvements we can obtain with various inter-cluster topologies which have different communication latencies. In our experiments, as inter-cluster communication cost increases speedups for all algorithms decrease, but the gap in performance of cp over rhop-ls and rhop-opt increases. This is because once RHOP makes poor decisions it is expensive to recover—a well-known drawback of a phased approach. It is worth noting here that even with a high communication cost, speedups increase with the number of clusters. However, as expected, topologies with faster inter-cluster communication always yield higher performance.

Consider the configurations with four clusters and an issue width of one (see Figure 9). As expected, as the communication costs increases the performance of all of the schedulers cp, rhop-opt, and rhop-ls decreases. More surprisingly, as the communication cost c increases, the gap between the performance of cp and rhop-opt increases from 20% when $c = 1$ to more than 35% when $c = 8$ (see architectural model 4–1 in Figure 9, for $c = 1$ top left, $c = 2$ top right, $c = 4$ bottom left, and $c = 8$ bottom right). In general, similar observations can be made for the architectures with larger issue widths and for the architectures with different numbers of clusters.

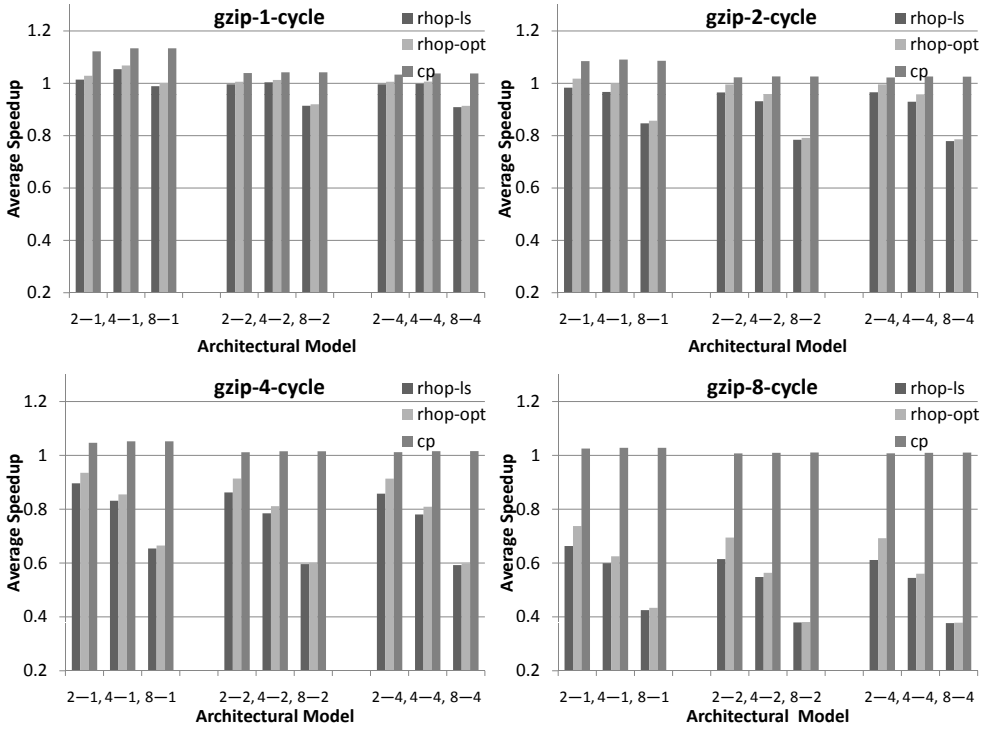

Fig. 11. Average speedups of superblocks for the gzip benchmark for different architectures with inter-cluster communication cost of one, two, four and eight respectively. On the x axis $\alpha - \beta$ means α clusters, $\alpha = 2, 4, 8$, and issue width of β, $\beta = 1, 2, 4$.

Figures 10 and 11 present the breakdown of performance improvements for two specific benchmarks—applu and gzip, respectively—for various architectural configurations. The applu benchmark (a floating point benchmark) is an example for which cp gets the best speedups that approach a factor of 2.8 on an eight cluster architecture. Conversely, the gzip benchmark (an integer benchmark) is an example where the speedups are more modest and approach 15% on an eight cluster architecture, which is due to the lack of instruction-level parallelism (ILP) in most SPEC integer benchmarks.

The scheduling time and percentage of provably optimal schedules. Table III lists the time it takes for the benchmarks to compile on two architectural configurations along with the percentage of superblocks on which our algorithm proved optimality within the ten minute timeout. The good news is that for almost all benchmarks cp can solve a majority of the superblocks in the SPEC benchmark to optimality. However this comes at a cost of increased compilation time with some benchmarks requiring more than a day to schedule all the superblocks in the benchmark. Also note that even in the case where most of the schedules are not provably optimal, we still get speedups. For example, for the benchmark eon (see Table III), only 37% of the superblocks are solved optimally yet cp yields a speedup of 15% on eon (see Figure 7). The scheduling times for RHOP alone are not given as they are negligible and the scheduling times for RHOP-

Table III. For each SPEC 2000 benchmark, total number of superblocks (num.), average size of superblocks (ave.), maximum size of superblocks (max.), total scheduling time for our scheduler, percentage of superblocks for which a provably optimal schedule was found, for various architectural models and communication costs $c = 1, 2$.

	superblocks			2-cluster-2-issue ($c = 1$)		4-cluster-2-issue ($c = 2$)	
benchmark	num.	ave.	max.	comp. time	% solved	comp. time	% solved
ammp	94	35	332	4 h: 51 m	70%	4 h: 52 m	70%
applu	21	58	200	0 h: 31 m	86%	0 h: 31 m	86%
apsi	156	28	95	11 h: 3 m	58%	11 h: 41 m	57%
art	29	16	40	0 h: 33 m	90%	0 h: 33 m	90%
bzip2	113	21	157	7 h: 43 m	62%	9 h: 4 m	56%
crafty	508	25	160	26 h: 28 m	68%	28 h: 32 m	67%
eon	132	39	225	14 h: 9 m	37%	14 h: 31 m	35%
equake	26	40	213	0 h: 33 m	89%	0 h: 33 m	89%
facerec	57	29	159	2 h: 11 m	78%	2 h: 12 m	78%
fma3d	389	26	586	11 h: 20 m	85%	11 h: 28 m	84%
galgel	71	23	75	3 h: 32 m	71%	3 h: 33 m	71%
gcc	2383	23	219	27 h: 1 m	94%	28 h: 26 m	93%
gzip	136	19	221	4 h: 55 m	79%	4 h: 58 m	79%
lucas	43	20	31	2 h: 40 m	63%	2 h: 41 m	63%
mcf	64	21	94	1 h: 57 m	80%	2 h: 12 m	80%
mesa	74	37	226	5 h: 2 m	63%	5 h: 13 m	59%
mgrid	28	17	69	1 h: 22 m	72%	1 h: 23 m	72%
parser	628	19	681	20 h: 59 m	82%	22 h: 31 m	80%
perlbmk	878	26	278	28 h: 49 m	81%	29 h: 60 m	80%
sixtrack	95	34	108	4 h: 1 m	75%	3 h: 51 m	76%
swim	6	31	77	0 h: 1 m	100%	0 h: 1 m	100%
twolf	186	25	380	11 h: 1 m	64%	11 h: 36 m	65%
vortex	476	41	303	14 h: 41 m	82%	14 h: 25 m	82%
vpr	229	26	173	8 h: 5 m	80%	8 h: 19 m	80%
wupwise	47	31	157	4 h: 36 m	43%	4 h: 60 m	45%

opt are similar to the scheduling times for the optimal temporal scheduler alone (see [Malik et al. 2008]).

Overall, our experimental results show that our constraint programming approach scales better than RHOP, both in terms of the number of clusters and the inter-cluster latency. RHOP sometimes partitions the superblocks more aggressively than necessary which results in slowdowns instead of speedups, whereas our approach always results in speedups. The application of constraint programming to the spatial scheduling problem has enabled us to solve the problem to near optimality for a significant number of code blocks. Solving the spatial scheduling problem with constraint programming has an added value over heuristic approaches in instances where longer compilation time is tolerable or the code-base is not very large. This approach can be successfully used in practice for software libraries, digital signal processing in addition to embedded applications. Our approach can also be used to evaluate the performance of heuristic techniques. Our solution also gives an added performance benefit by distributing the workload over clusters and the ability to utilize resources that might otherwise remain idle.

5. RELATED WORK

Traditionally, instruction scheduling has been employed by compilers to exploit instruction level parallelism in straight-line code in the form of basic blocks [Heffernan and Wilken 2005; Malik et al. 2008] and superblocks [Heffernan et al. 2006; Shobaki and Wilken 2004; Malik et al. 2008]. In this section we review the different approaches towards solving the spatial scheduling problem.

The most well known solutions for spatial scheduling are greedy and hierarchical partitioning algorithms which assign the instructions before the scheduling phase in the compiler. The bottom-up greedy, or BUG algorithm [Ellis 1986], which is the earliest among spatial scheduling algorithms, proceeds by recursing depth first along the data dependence graph, assigning the critical paths first. It assigns each instruction to a cluster based on estimates of when the instruction and its predecessors can complete execution at the earliest. These values are computed using the resource requirement information for each instruction. The algorithm queries this information before and after the assignment to effectively assign instructions to the available clusters. This technique works well for simple graphs, but as the graphs become more complex the greedy nature of the algorithm directs it to make decisions that negatively affect future decisions. Chung and Ranka [1995] also gave an early solution to spatial scheduling for distributed memory multiprocessors based on heuristics for list scheduling algorithms. Leupers [2000] present a combined partitioning and scheduling technique using simulated annealing. Lapinskii et al. [2002] propose a binding algorithm for instructions which relies on list scheduling to carry out temporal scheduling.

Lee et al. [2002] present a multi-heuristic framework for scheduling basic blocks, superblocks and traces. The technique is called *convergent scheduling*. The scheduler maintains a three dimensional weight matrix $W_{i,c,t}$, where the ith dimension represents the instructions, c spans over the number of clusters and t spans over possible time slots. The scheduler iteratively executes multiple scheduling phases, each one of which heuristically modifies the matrix to schedule each instruction on a cluster for a specific time slot, according to a specific constraint. The main constraints are preplacement, communication minimization and load balancing. After several passes the weights are expected to converge. The resultant matrix is used by a traditional scheduler to assign instructions to clusters. The framework has been implemented on two different spatial architectures, RAW and Chorus clustered VLIW infrastructure. The framework was evaluated on standard benchmarks, mostly the ones with dense matrix code. An earlier attempt was made by the same group for scheduling basic blocks in the Raw compiler [Lee et al. 1998]. Inter-cluster moves on RAW take 3 or more cycles and the Chorus infrastructure assumes single cycle moves in its simulation. This technique iteratively clustered together instructions with little or no parallelism and then assigned these clusters to available clusters. A similar approach was used to schedule instructions on a decoupled access/execute architectures [Rich and Farrens 2000]. These techniques seem to work well on selective benchmark suits with fine tuned system parameters which are configured using trial and error. It is difficult to evaluate the actual effectiveness of these technique mainly because it attempts to solve the temporal and spatial scheduling intermittently. In contrast our approach attempts to solve spatial scheduling first. In an earlier attempt on spatial scheduling [Amarasinghe et al. 2002] presented integer linear formulations of the problem as well as an 8-approximation algorithm for it. The evaluation in the unpublished report only included results from heuristic algorithms and were from a simulation over a select group of benchmarks.

Chu et al. [2003] describe a region-based hierarchical operation partitioning algorithm (RHOP), which is a pre-scheduling method to partition operations on multiple clusters. In order to produce a partition that can result in an efficient schedule, RHOP uses schedule estimates and a multilevel graph partitioner to generate cluster assignments. This approach partitions a data dependence graph based on weighted vertices and edges. The algorithm uses a heuristic to assign weights to the vertices to reflect their resource usage and to the edges to reflect the cost of inter-cluster communication in case the two vertices connected by an edge are assigned to different clusters. In the partitioning phase, vertices are grouped together by two processes called *coarsening*

and *refinement* [Hendrickson and Leland 1995; Karypis and Kumar 1998]. Coarsening uses edge weights to group together operations by iteratively pairing them into larger groups while targeting heavy edges first. The coarsening phase ends when the number of groups is equal to the number of desired clusters for the machine. The refinement phase improves the partition produced by the coarsening phase by moving vertices from one partition to another. The goal of this phase is to improve the balance between partitions while minimizing the overall communication cost. The moves are considered feasible if there is an improvement in the gain from added parallelism minus the cost of additional inter-cluster communications. The algorithm has been implemented in the Trimaran compiler and simulation framework. The framework has the capability to model homogeneous as well as heterogeneous architectures and assumes a single cycle cost for inter-cluster moves. Their technique was evaluated on the SPEC benchmark and compared against BUG, which RHOP always outperforms. Subsequent work using RHOP partitions data over multi-core architectures with a more complex memory hierarchy [Chu et al. 2007; Chu and Mahlke 2006]. Unlike other approaches which are mostly evaluated on basic blocks, RHOP has also been evaluated over hyperblocks.

Nagpal and Srikant [2004; 2008] give an integrated approach to spatial and temporal scheduling by binding the instructions to functional units in clusters. The approach extends the list scheduling algorithm to incorporate a resource need vector for effective functional unit binding. Their scheme utilizes the exact information about the available communication requirements, functional units and the load on different clusters in addition to the constraints imposed by the architecture to prioritize instructions that are ready to be scheduled. The algorithm and its variations have been implemented for Texas Instruments VelociTI architecture using SUIF compiler framework. They evaluated their technique using the TI simulator for TMS320C6X on the most frequently executed benchmark kernels from MediaBench and report speedups of up to 19%.

In contrast to our work, which presents an optimal integrated approach for spatial and temporal scheduling, Kessler, Bednarski, and Eriksson [2006; 2006; 2009] pursue a much more ambitious agenda of integrating spatial and temporal scheduling with instruction selection, register allocation, and software pipelining. Although successful on smaller basic blocks, their fully integrated approaches, which use dynamic programming and integer linear programming, do not scale beyond blocks of size 20–40 instructions using a timeout of one hour (our constraint programming technique scales consistently to blocks with up to 100 instructions using a timeout of 10 minutes).

Other related works have also dealt with software pipelining techniques for clustered VLIW architectures [Nystrom and Eichenberger 1998; Sánchez and González 2000; Codina et al. 2001]. Most of these techniques extend the greedy scheduling algorithms and apply them after unrolling frequently executed loops.

6. CONCLUSIONS

This article presents a constraint programming approach to the instruction assignment problem for taking advantage of the parallelism contained in local blocks of code for multi-cluster architectures. We also study the effect of different hardware parameters including issue-width and cost of inter-cluster communication performance.

Our approach takes advantage of the problem decomposition technique to solve spatial scheduling in two stages, yet it is integrated with temporal scheduling. We also employ various constraint programming techniques including symmetry breaking and branch-and-bound to reduce the time in searching for a solution. Reformulation of the problem model in terms of the edges of the DAG instead of the vertices breaks the symmetry nicely to reduce the search space. In addition we also use techniques from

graph theory to predetermine instructions which can be grouped together before the search algorithm starts.

We compared out implementation against RHOP on various architectural configurations. We found that our approach was able to achieve an improvement of up to 26%, on average, over the state-of-the-art techniques on superblocks from SPEC 2000 benchmarks. Clustered architectures are becoming increasingly important because they are a natural way to extend the embedded processors without significant increase in power utilization, which is vital for these architectures. Also many of the applications which run on embedded devices are compiled once and usually run throughout the lifetime of the device without recompilation and hence reasonably long compile times are also acceptable. Our approach provides good speedups with a lower-bound on the speedup that can be obtained.

REFERENCES

AGGARWAL, A., AND FRANKLIN, M. 2005. Scalablility aspects of instruction distribution algorithms for clustered processors. *IEEE Transactions on Parallel and Distributed Systems, 16,* 10, 944–955.

ALETA, A., CODINA, J. M., SANCHEZ, J., GONZÁLEZ, A., AND KAELI, D. 2009. AGAMOS: A graph-based approach to modulo scheduling for clustered microarchitectures. *IEEE Transactions on Computers, 58*, 6, 770–783.

AMARASINGHE, S., KARGER, D. R., LEE, W., AND MIRROKNI, V. S. 2002. A theoretical and practical approach to instruction scheduling on spatial architectures. MIT, LCS Tech. Report.

ANDREEV, K., AND RÄCKE, H. 2004. Balanced graph partitioning. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures,* 120–124.

ARM, The architecture for the digital world. http://www.arm.com. Retrieved June, 2011.

BEDNARSKI, A. AND KESSLER, C. W. 2006. Optimal integrated VLIW code generation with integer linear programming. In *Proceedings of Euro-Par 2006*, 461–472.

BEG, M., AND VAN BEEK, P. 2011. A constraint programming approach to instruction assignment. *The 15th Annual Workshop on the Interaction between Compilers and Computer Architecture (INTERACT'15).*

BENDERS, J. F. 1962. Partitioning procedures for solving mixed-variables programming problems. *Numerische Mathematik 4,* 238–252.

BJERREGAARD, T., AND MAHADEVAN, S. 2006. A survey of research and practices of network-on-chip. *ACM Computing Surveys, 38,* 1, 1–51.

BLAINEY, R. J. 1994. Instruction scheduling in the TOBEY compiler. *IBM J. Res. Develop., 38*, 5, 577–593.

CHU, M., FAN, K., AND MAHLKE, S. 2003. Region-based hierarchical operation partitioning for multicluster processors. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03),* 300–311.

CHU, M., AND MAHLKE, S. 2006. Compiler-directed data partitioning for multicluster processors. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'06),* 208–220.

CHU, M., RAVINDRAN, R., AND MAHLKE, S. 2007. Data access partitioning for fine-grain parallelism on multicore architectures. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (Micro'07),* 369–380.

CHUNG, Y. C., LIU, C. C., AND LIU, J. S. 1995. Applications and performance analysis of an optimization approach for list scheduling algorithms on distributed memory multiprocessors. *Journal of Information Science and Engineering, 11,* 2, 155–181.

CODINA, J. M., SÁNCHEZ, J. F, AND GONZÁLEZ, A. 2001. A unified modulo scheduling and register allocation technique for clustered processors. In *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques (PACT'01)*, 175–184.

DANTZIG, G. B., AND WOLFE, P. 1960. Decomposition principle for linear programs. *Operations Research, 8,* 101–111.

ELLIS, J. R. 1986. *Bulldog: A compiler for VLSI architectures.* MIT Press.

ERIKSSON, M. V. AND KESSLER, C. W. 2009. Integrated modulo scheduling for clustered VLIW architectures. In *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers (HiPEAC'09)*, 65–79.

FARABOSCHI, P., DESOLI, G., AND FISHER, J. A. 1998. Clustered instruction-level parallel processors. *HP Labs Technical Report HPL-98-204*, 1–29.

FISHER, J. A., FARABOSCHI, P. AND YOUNG, C. 2005. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*, Elsevier.

HEFFERNAN, M., AND WILKEN, K. 2005. Data-dependency graph transformations for instruction scheduling. *Journal of Scheduling, 8,* 427–451.

HEFFERNAN, M., WILKEN, K., AND SHOBAKI, G. 2006. Data-dependency graph transformations for superblock scheduling. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (Micro'06),* 77–88.

HENDRICKSON, B., AND LELAND, R. 1995. A multilevel algorithm for partitioning graphs. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing (Supercomputing'95),* 28.

HOXEY, S., KARIM, F., HAY, B., AND WARREN, H. 1996. *The PowerPC Compiler Writers Guide*, Warthman Associates.

KARYPIS, G. AND KUMAR, V. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing, 20,* 1, 359–392.

KESSLER, C. W. AND BEDNARSKI, A. 2006. Optimal integrated code generation for VLIW architectures. *Concurrency and Computation: Practice and Experience, 18,* 11, 1353–1390.

LAPINSKII, V. S., JACOME, M. F., AND DE VECIANA, G. A. 2002. Cluster assignment for high-performance embedded VLIW processors. *ACM Transactions on Design Automation of Electronic Systems, 7*, 430–454.

LEE, W., BARUA, R., FRANK, M., SRIKRISHNA, D., BABB, J., SARKAR, V., AND AMARASINGHE, S. 1998. Space-time scheduling of instruction-level parallelism on a RAW machine. In *Proceedings of the 8th Int'l Conf. on Architectural Support for Prog. Lang. and Operating Systems (ASPLOS-VIII),* 46–57.

LEE, W., PUPPIN, D., SWENSON, S., AND AMARASINGHE, S. 2002. Convergent scheduling. In *Proceedings of the 35th annual ACM/IEEE International Symposium on Microarchitecture (Micro'35),* 111–122.

LEUPERS, R. 2000. Instruction Scheduling for Clustered VLIW DSPs. In *Proceedings of IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT'00)*, 291–300.

LUO, C., BAI, Y., XU, C., AND ZHANG, L. 2009. FCCM: A novel inter-core communication mechanism in multi-core platform. In *Proceedings of International Conference on Science and Engineering,* 215–218.

MALIK, A. M., MCINNES, J., AND VAN BEEK, P. 2008. Optimal basic block instruction scheduling for multiple-issue processors using constraint programming. *International Journal on Artificial Intelligence Tools, 17,* 1, 37–54.

MALIK, A. M., CHASE, M., RUSSELL, T., AND VAN BEEK, P. 2008. An application of constraint programming to superblock instruction scheduling. In *Proceedings of the Fourteenth International Conference on Principles and Practice of Constraint Programming (CP'08)*, 97–111.

NAGPAL, R., AND SRIKANT, Y. N. 2004. Integrated temporal and spatial scheduling for extended operand clustered VLIW processors. *Conference on Computing Frontiers,* 457–470.

NAGPAL, R., AND SRIKANT, Y. N. 2008. Pragmatic integrated scheduling for clustered VLIW architectures. *Software Practice and Experience, 38,* 227–257.

NYSTROM, E., AND EICHENBERGER, A. E. 1998. Effective cluster assignment for modulo scheduling. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (Micro'98)*.

OWENS, J. D., DALLY, W. J., HO, R., JAYASIMHA, D. N., KECKLER, S. W. AND PEH, L. 2007. Research challenges for on-chip interconnection networks. *IEEE Micro, 27,* 5, 96–108.

PARCERISA, J-M., SAHUQILLO, J., GONZÁLEZ, A., AND DUATO, J. 2002. Efficient interconnects for clustered microarchitectures. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques (PACT'02),* 291–300.

RICH, K. ANDFARRENS, M. 2000. Code partitioning in decoupled compilers. In *Proceedings from the 6th International Euro-Par Conference on Parallel Processing (Euro-Par'00)*, 1008–1017.

RUSSELL, T., MALIK, A., CHASE, M., AND VAN BEEK, P. 2009. Learning heuristics for the superblock instruction scheduling problem. *IEEE Trans. on Knowledge and Data Engineering, 21,* 10, 1489–1502.

ROSSI, F., VAN BEEK, P., AND WALSH, T. (ED). 2006. *Handbook of Constraint Programming*. Elsevier.

SÁNCHEZ, J., AND GONZÁLEZ, A. 2000. Instruction scheduling for clustered VLIW architectures. In *Proceedings of the 13th International Symposium on System Synthesis (ISSS'00)*, 41–46.

SHOBAKI, G. AND WILKEN, K. 2004. Optimal superblock scheduling using enumeration. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (Micro'04)*, 283–293.

TERECHKO, A. S., AND CORPORAAL, H. 2007. Inter-cluster communication in VLIW architectures. *Transactions on Architecture and Code Optimization (TACO), 4,* 2, 1–38.

TERECHKO, A. S. 2007. Clustered VLIW architectures: a quantitative approach. *Doctoral Thesis, Technischie Universiteit Eindhoven*.

TEXAS INSTRUMENTS. http://www.ti.com. Retrieved June, 2011.