A Constraint Programming Approach for Instruction Assignment

Mirza Beg
David R. Cheriton School of Computer Science
University of Waterloo, Canada
mbeg@cs.uwaterloo.ca

Peter van Beek
David R. Cheriton School of Computer Science
University of Waterloo, Canada
vanbeek@cs.uwaterloo.ca

Abstract

A fundamental problem in compiler optimization, which has increased in importance due to the spread of multi-core architectures, is to find parallelism in sequential programs. Current processors can only be fully taken advantage of if workload is distributed over the available processors. In this paper we look at distributing instructions in a block of code over multi-cluster processors, the *instruction assignment problem*. The optimal assignment of instructions in blocks of code on multiple processors is known to be NP-complete. In this paper we present a constraint programming approach for scheduling instructions on multi-cluster systems that feature fast inter-processor communication. We employ a problem decomposition technique to solve the problem in a hierarchical manner where an instance of the master problem solves multiple sub-problems to derive a solution. We found that our approach was able to achieve an improvement of 6%-20%, on average, over the state-of-the-art techniques on superblocks from SPEC 2000 benchmarks.

1. Introduction

Modern architectures feature multiple processors and can only be fully taken advantage of if the compiler can effectively utilize the available hardware parallelism. Traditionally, instruction scheduling has been employed by compilers to exploit instruction level parallelism in straight-line code in the form of basic blocks [6, 8, 15, 16, 21, 25, 32, 33] and superblocks [7, 13, 17, 19, 26, 31]. A *basic block* is a sequence of instructions 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. The idea behind instruction scheduling is to minimize the runtime of a program by reordering instructions in a block given the dependencies between them and the resource constraints of the architecture for which the program is being compiled. This stage of compilation is all the more important for in-order processors as they strictly follow the schedule suggested by the compiler. Almost all current architectures are based on the chip multi-processor model, a trend which necessitates changes in compiler design to make better use of the available resources.

Current processors communicate with each other using shared memory where the communication latency can be as low as 14 cycles. Luo et al. [24] describe an approach to lower the cost of communication between processor cores by communicating through the memory hierarchy. Recent trends in architecture research places an extra emphasis on designing and developing on-chip networks for inter-processor communication. Bjerregaard and Mahadevan [5] conducted a survey of different designs and implementations of on-chip networks. Owens et al. [27] delve into the design of infrastructure support for on-chip networks and the research challenges for implementing them on multi-cores. It is expected that these networks will provide fast communication between cores which will facilitate parallelization of programs.

The primary goal of parallelization is to identify parts of the program which can be executed concurrently on different processors. Previous work has proposed heuristic approaches to partition straight-line regions of code for multi-cluster (see [1] and the references therein) architectures (for some recent work, also see [9, 14, 22, 23, 28]). Ellis [14] gives a greedy approach to assign instructions on processors in a clustered VLIW architecture known as BUG. This approach proceeded by assigning instructions on the critical path to the same processor. Lee et al. [23] present convergent scheduling as a combined method for assigning and scheduling instructions on clustered VLIW architecture. Chu et al. [9] describe a hierarchical approach to find balanced partitions of a given dependence graph for a block. This technique is known as RHOP. RHOP is the state-of-the-art in terms of parallelizing local blocks of code for multiple processors.

In this paper, we present a constraint programming approach to instruction assignment on clustered processors that is robust and optimal. 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, which reduces the effort needed in the 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. It is worth noting that on the PowerPC 15% of the instructions executed by the processor are serializing instructions.

In our experiments we evaluated our approach on the SPEC 2000 integer and floating point benchmarks, using different architectural models. We compared our results against RHOP using the same communication cost as in [9]. We found that in our experiments we were able to improve on RHOP up to 79% depending on the architectural model. Also in our experiments we were able to

INTERACT-15, February 2011.

solve a large percentage of blocks optimally with a ten minute time limit for each block. This represents a significant improvement over existing solutions.

The rest of the paper is organized as follows. Section 2 gives an overview of the related work. An overview of the background material is given in Section 3. Section 4 gives details of our approach and improvements to the model. Section 5 describes the experimental setup and results. Section 6 discusses and analyzes the approach given in this paper. Finally, the paper concludes with Section 7.

2. Related Work

In this section we review the different approaches towards solving the instruction assignment problem.

Lee et al. [23] 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 processors 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 processor for a specific time slot, according to a specific constraint. The main constraints are pre-placement, 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 processors. The framework has been implemented on two different spatial architectures, Raw and clustered VLIW. 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 [22]. This technique iteratively clustered together instructions with little or no parallelism and then assigned these clusters to available processors. A similar approach was used to schedule instructions on a decoupled access/execute architectures [28].

The most well known solutions to the assignment problem are greedy and hierarchical partitioning algorithms which assign the instructions before the scheduling phase in the compiler. The bottom-up greedy, or BUG algorithm [14] proceeds by recursing depth first along the data dependence graph, assigning the critical paths first. It assigns each instruction to a processor 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 processors.

Chu et al. [9] describe a region-based hierarchical operation partitioning algorithm (RHOP), which is a pre-scheduling method to partition operations on multiple processors. 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-processor communication in case the two vertices connected by an edge are assigned to different processors. In the partitioning phase, vertices are grouped together by two processes called *coarsening* and *refinement* [18, 20]. 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 processors 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-processor communications. The algorithm has been implemented in the Trimaran framework. Subsequent work using RHOP partitions data over multi-core architectures with a more complex memory hierarchy [10, 11].

3. Background

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

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

- Clusters are homogeneous. This means that all processors have the same number of identical functional units, with the same issue-width.
- The processor model is realistic in the sense that the instruction set may contain non-pipelined instructions as well as serializing instructions. These are instructions which may disrupt the instruction pipeline of superscalar architectures.
- Clusters can communicate with each other with a constant cost of c cycles. After the result of an instruction is available, it would take c cycles to transfer the resultant value on a different cluster where it is needed. We use $c = 1$ as in RHOP for our experiments.

The assumptions given above are similar to those used to test RHOP [9] with the difference being that RHOP does not assume homogeneous processors and does not consider non-pipelined or serializing instructions which are a common feature in realistic instruction set architectures.

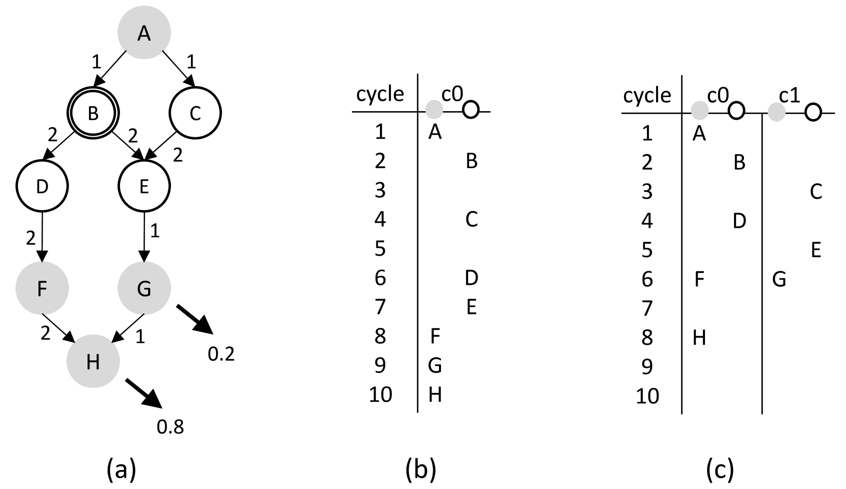
We use the standard directed graph (DAG) representation for the basic blocks and superblocks. A *basic block* is a sequence of instructions with a single entry point and a single exit. A *superblock* is a sequence of instructions with a single entry point and multiple possible exits. Each vertex in the 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. *Exit vertices* are special nodes in a DAG representing branch instructions. 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 1(a) for a DAG representing a superblock.

With the given architectural model and the dependency DAG for a basic block or a superblock, the assignment 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.

Definition 3.1 (Schedule)**.** The schedule S for a block is a mapping of each instruction in a DAG to a time cycle.

Definition 3.2 (Schedule Length)**.** The length of a schedule for a basic block is the cycle in which the last instruction in the block is issued.

Definition 3.3 (Weighted Completion Time)**.** The weighted completion time for a superblock schedule is $\sum_{i=1}^{n} w(i)S(i)$, where n



cycle	c0	
1	A	
2		B
3		
4		C
5		
6		D
7		E
8	F	
9	G	
10	H	

cycle	c0		c1	
1	A			
2		B		
3				C
4		D		
5				E
6	F		G	
7				
8	H			
9				
10				

Figure 1. (a) Graph representation of a superblock: G and H are branch instructions with exit probabilities of 0.2 and 0.8 respectively. B is a serializing instruction and C is a non-piplined 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 9×0.2 + 10×0.8 = 9.8 cycles. (c) A possible schedule for the same superblock for a dual-cluster where the processors can communicate with unit cost and each processor is the same as the single-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.

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 schedule length and weighted completion time, which applies to basic blocks and superblocks respectively, the assignment problem can be stated as follows.

Definition 3.4 (Assignment)**.** The assignment A for a block is a mapping of each instruction in a DAG to a cluster.

Definition 3.5 (Instruction Assignment Problem)**.** Given the dependence graph $G = (V, E)$ for a basic block or a superblock and the number of available clusters k in a given architectural model, the *instruction assignment problem* is to find an assignment A and a schedule S such that $A(i) \in \{1, \ldots, k\}$ for each instruction i in the block and start time $S(i) \in \{1, \ldots, \infty\}$ for all the instructions $i \in V$ that minimizes the schedule length of the basic block or the weighted completion time in case of superblocks.

Instruction scheduling on realistic multiple issue processor is known to be a hard problem and compilers use heuristic approaches to schedule instructions. Instruction assignment can be simply stated as scheduling instructions on multiple processors. The idea would be to partition the DAG and schedule each partition on a processor.

Definition 3.6 (Graph Partitioning)**.** The graph partitioning problem consists of dividing a graph G into k disjoint parts while satisfying some balancing criterion.

When $k = 2$, the problem is also reffered to as the graph bisection problem. Balanced graph partitioning problem is known to be NP-hard for $k \geq 2$ [2]. The assignment 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 2 to k). We use constraint programming to model and solve the instruction assignment problem.

Constraint programming is a methodology for solving hard combinatorial problems, where a problem is modeled in terms of variables values and constraints (see [30]). Once the problem has been modeled in which the variables along with their domains have been identified and the constraints specified, backtracking over the variables is employed to search for an optimal solution.

4. Constraint Programming Approach

In this section we present a constraint model for the instruction assignment problem.

Each instruction is represented by a node in the basic block dependency graph. Each node i in the graph is represented by two variables in the model, x_i and y_i. The variable $x_i \in \{1, \ldots, \infty\}$ is the cycle in which the instruction is to be issued. The upper-bound to these variables can be calculated using a heuristic scheduling method. The variable $y_i \in \{1, \ldots, k\}$ identifies the cluster to which instruction i is assigned. The key is to scale up to large problem sizes. In developing an optimal solution to the assignment problem we have applied and adapted several techniques from the literature including symmetry breaking, branch and bound and structure based decomposition techniques.

The main idea is to solve the two stage problem of assignment and scheduling. The idea was inspired by integer programming, Benders [4] and Dantzig-Wolfe [12] decomposition techniques, to decompose the problem into master-slave. We model the assignment problem as master which models and solves multiple slave problems to schedule instructions for a given assignment at each stage.

4.1 Symmetry Breaking

Symmetry can be exploited to reduce the amount of search needed to solve the problem. If the search algorithm is visiting the equivalent states over and over again then excluding these states such that equivalent states are not visited multiple times as the result in all cases is the same.

Using this technique we aim to remove provably symmetric assignments to instructions. An example would be the first instruction being assigned to the first cluster and thus discarding all the solutions where the first instruction is on any other cluster, while preserving 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 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} assigned a value $=$ means that variables y_i and y_j will take the same values and vice versa for $\neq$. Once the variables $z_{ij} \in \{=, \neq\}$ are set, an assignment to all instructions can be determined, i.e. values can be uniquely assigned to all variables y_i for $i \in \{1, .., n\}$. Once the assignment to all instructions is available the existing optimal instruction scheduler [26] can be used to compute the schedule length or the weighted completion time for the block for the given 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.

4.2 Branch and Bound

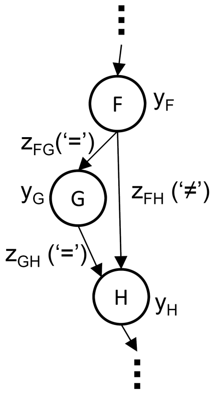


Figure 2. An example of inconsistent assignment to z variables for which valid values cannot be assigned to the y variables.

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. The first case: if 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 $\{1, ..., k\}$ without violating the constraints given by the z variables. An example of such a violation is given in Figure 2. The second case: if the partial assignment to the y variables can be proven to result in a partial schedule with a cost greater than the established upper bound. The search space can be reduced by eliminating all such assignments containing this sub-assignment. 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.

4.3 Connected Sub-structures

The amount of search done by the algorithm can be reduced if it can be pre-determined that certain instructions are strongly connected and would be assigned to the same cluster in at-least one optimal solution to the assignment problem. To this end we define the connected sub-structure as follows. Some examples of connected sub-structures are given in Figure 3. For the purpose of the experiments we only consider chains as connected sub-structures.

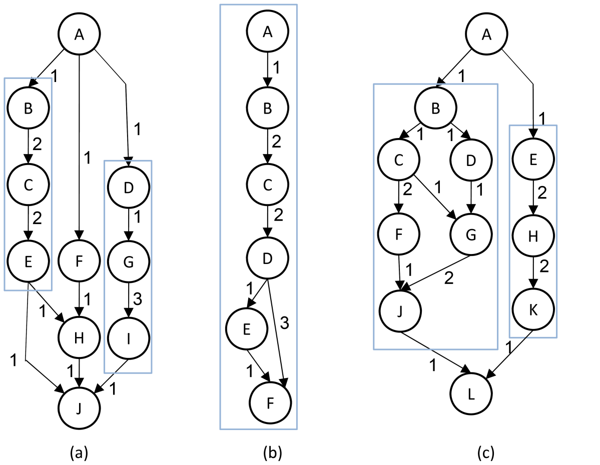


Figure 3. Examples of connected sub-structures in blocks. Each of them is marked by bounding box. Chains like the ones given in (a) and (b) form connected sub-structures in all architectures where as complex connected sub-structures may also exist like in (c) where the connectedness is conditional upon the types of instructions and/or the architecture of the processor the code is being compiled for.

Definition 4.1 (Connected Sub-structure)**.** A connected sub-structure in the DAG is a set of instructions with a single entry instruction into the structure and a single exit from the structure and can be executed on a single processor in the early start time of the last instruction with respect to that structure.

Theorem 4.2. *A chain of size at least three is a connected sub-structure.*

Proof: On the contrary, assume that two or more instructions in the chain are assigned to different clusters. Every edge that is cut will incur an additional penalty of the communication cost. Hence the length of path from the source to the sink of the connected sub-structure increases by that amount. □

Theorem 4.3. *All instructions in a connected sub-structure can be assigned to the same processor without eliminating at-least one optimal assignment of the instructions to clusters.*

Proof: A connected sub-structure cannot be scheduled more efficiently on multiple processors as compared to a single one. It is possible however to distribute the instructions of a connected sub-structure over multiple processors without increasing the schedule length of the sub-structure. Thus the assignment of all instructions in the sub-structure to the same processor would result in a schedule with the same cost as any other optimal schedule where the instructions of the sub-structure are not on the same processor. □

Lemma 4.4. *All instructions in a chain of size at least three can be assigned to same cluster without eliminating all optimal solutions to the assignment problem.*

Proof: Follows from the theorems above. □

4.4 Solving an Instance

Solving an instance of the instruction assignment problem proceeds with the following steps. 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.

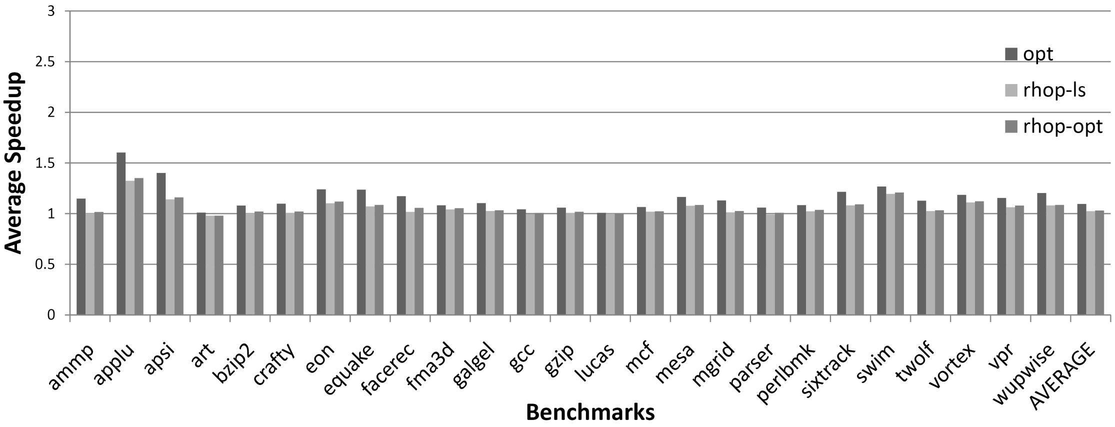


Figure 4. Average speedups of superblocks in SPEC 2000 for a 4-cluster 2-issue architecture.

The lower bound is computed using an extension of the optimal scheduler [26] for multi-cluster architecture. To compute the lower-bound we assume a communication cost of zero and no serializing instructions. The upper-bound is initially established using an extension to the list-scheduling algorithm [3]. These 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. 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. If the cost is equal to the lower-bound then a solution is 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 and bounds checks are intended to prune the search tree and save search time. An outline of our solution to the instruction assignment problem is given as Algorithm 1.

Algorithm 1 Instruction Assignment ($DAG\ \mathcal{G}$)

```
$ub \leftarrow$ Establish upper bound using list-scheduler extension
$lb \leftarrow$ Establish lower bound using ext. optimal scheduler
$\mathcal{E} \leftarrow$ set of edges in $\mathcal{G}$ with domain$\{=, \neq\}$
find connected sub-structures set edges to $\{=\}$
backtrack on edges $e \in \mathcal{E}$
for each node $n$ of the search tree do
    if $n$ is an internal node of search tree then
        consistency check ($n$)
        bounds check ($n$)
        if any of the checks fail, discard subtree rooted at $n$
    end if
    if $n$ is a leaf node of search tree then
        $A \leftarrow$ generate assignment for $n$
        $S \leftarrow$ determine schedule for assignment $A$
        $ub \leftarrow$ update $ub$ using $S$
    end if
    if $ub = lb$ then
        return solution
    end if
end for
return $ub$ as solution
```

5. Evaluation

We evaluated our solution to the instruction assignment problem on the SPEC 2000 integer and floating point benchmarks, as all of the superblocks from these benchmarks were readily available to us. 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 paper are for superblocks. The benchmarks were compiled using the IBM Tobey compiler targeted towards the IBM PowerPC processor, and the superblocks were captured as they were passed to Tobey's instruction scheduler. 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. 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.

We conducted our evaluation using four realistic architectural models given in Table 1. In these architectures, the functional units are not fully pipelined, the issue width of the processor is not equal to the number of functional units, and there are serializing instructions. We assume homogeneous clusters; i.e., all processors are considered to be the same. Additionally we also assume that clusters can communicate with each other with a latency of one cycle.

architecture (issue width)	int. func. units	mem. units	branch units	floating pt. units
1-issue (1)	1			
2-issue (2)	1	1	1	1
4-issue (4)	2	1	1	1
6-issue (6)	2	2	3	2

Table 1. Architectural models.

We evaluated our instruction assignment algorithm with respect to how much it improves on previous approaches. The state-of-the-art instruction assignment algorithm for instruction assignment is a hierarchical graph partitioning technique known as RHOP [9]

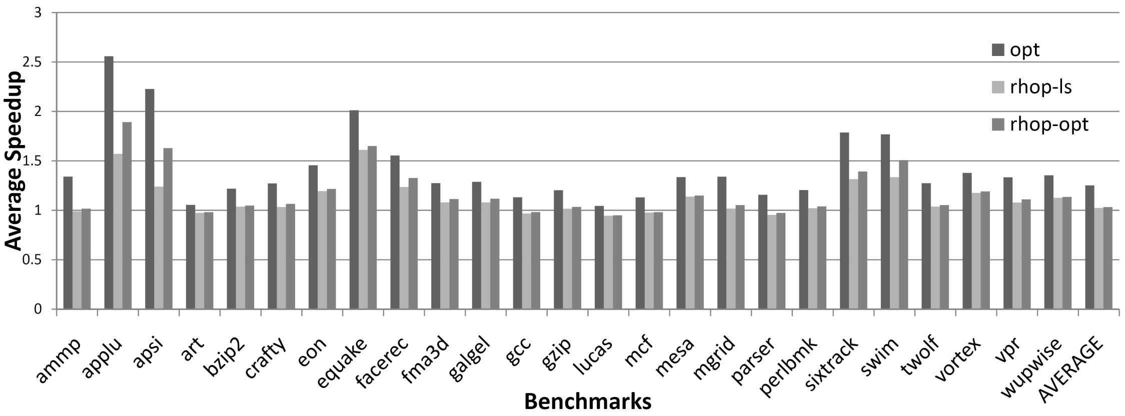


Figure 5. Average speedups of superblocks in SPEC 2000 for a 8-cluster 1-issue architecture.

	4-cluster-2-issue		8-cluster-1-issue	
benchmark	rhop-ls	rhop-opt	rhop-ls	rhop-opt
ammp	14.6%	12.9%	35.6%	31.9%
applu	21.1%	18.7%	62.8%	35.2%
apsi	22.8%	20.7%	79.9%	36.7%
art	3.0%	3.0%	8.1%	7.5%
bzip2	7.6%	5.7%	17.6%	16.4%
crafty	9.0%	7.6%	23.0%	19.4%
eon	12.5%	10.7%	22.0%	19.7%
equake	15.4%	13.8%	24.9%	21.9%
facerec	15.2%	10.9%	25.8%	17.2%
fma3d	3.9%	2.7%	18.0%	14.4%
galgel	7.6%	6.9%	19.2%	15.2%
gcc	4.8%	3.6%	16.9%	15.2%
gzip	5.2%	4.0%	18.4%	16.2%
lucas	0.8%	0.8%	10.5%	9.8%
mcf	4.6%	4.2%	15.7%	15.4%
mesa	8.1%	7.4%	17.3%	16.3%
mgrid	11.6%	10.2%	31.8%	27.3%
parser	7.0%	5.1%	21.4%	18.8%
perlbmk	6.1%	4.6%	18.0%	15.9%
sixtrack	12.4%	11.3%	35.8%	28.3%
swim	6.0%	4.9%	32.5%	17.3%
twolf	10.0%	9.1%	22.8%	21.0%
vortex	6.7%	5.6%	17.3%	15.7%
vpr	8.7%	7.0%	23.7%	20.1%
wupwise	11.4%	10.8%	20.2%	19.3%
AVERAGE	7.0%	6.3%	22.3%	21.3%

Table 2. The percentage improvement of our technique over rhop-ls and rhop-opt for SPEC 2000 benchmarks on 4-cluster-2-issue and 8-cluster-1-issue architectures.

which uses coarsening and refinement to partition the dependency graph for multiple processors. Once the partitions are formed our simulation uses a list scheduler (using dependence height and speculative yield (DHASY) heuristic, which is also the default heuristic in the Trimaran compiler) to determine the schedule for the superblocks. To determine the effectiveness of our partitions the results are also reported for RHOP-OPT where the schedule for the given partitions is determined by an optimal scheduler [26]. The performance improvements are measured over the list scheduling algorithm for a single cluster architecture. For running our algorithm we used a timeout of ten minutes for each superblock. A ten minute timeout on each superblock allows an entire benchmark to be compiled in a reasonable amount of time.

		1-cluster	2-clusters	4-clusters	8-clusters
apsi	1-issue	96%	79%	71%	29%
	2-issue	97%	80%	69%	38%
	4-issue	95%	80%	70%	40%
	6-issue	96%	83%	79%	67%
mgrid	1-issue	94%	92%	86%	61%
	2-issue	96%	92%	86%	64%
	4-issue	96%	92%	86%	64%
	6-issue	100%	100%	98%	89%
lucas	1-issue	100%	100%	98%	30%
	2-issue	100%	100%	100%	46%
	4-issue	100%	100%	100%	51%
	6-issue	100%	100%	100%	94%

Table 3. The percentage of superblocks solved optimally for each benchmark for different architectural models.

Figures 4 and 5 compare the performance of a 4-cluster dual issue architecture using our algorithm(opt), RHOP (rhop-ls) and RHOP-OPT (rhop-opt) with the performance of 8-cluster single issue architecture. For the 4-cluster architecture our algorithm achieves improvements from 0.8% (lucas) upto 22.8% (apsi) over rhop-ls and from 0.8% upto 20.7% over rhop-opt. On average we improved over the previous techniques by 6%-7% for the 4-cluster configuration. In the case of the 8-cluster configuration our algorithm improves over the rhop-ls by 22% and over rhop-opt by 21% on average. These results are given in Table 2. These results show that our algorithm scales better with the number of processors which is because RHOP sometimes partitions the blocks more aggressively than necessary which results in slowdowns instead of speedups. Also note that the results for RHOP reported here are giving higher speedups than those reported by Chu et al. [9]. This is because we were unable to incorporate the effect of inter-block dependencies which would also incur a penalty if they are across partitions assigned to different clusters.

We also evaluated our optimal assignment algorithm to see how it scales with varying issue width for different number of processors. Figures 6, 7 and 8 give the performance of our algorithm compared to rhop-ls and rhop-opt for various configurations. The improvements for lucas benchmark are moderate but for mgrid our algorithm gives good improvements. The results show that speedups from our algorithm increase with the number of processors and decrease with increasing issue-width. It is worth noting that our algo-

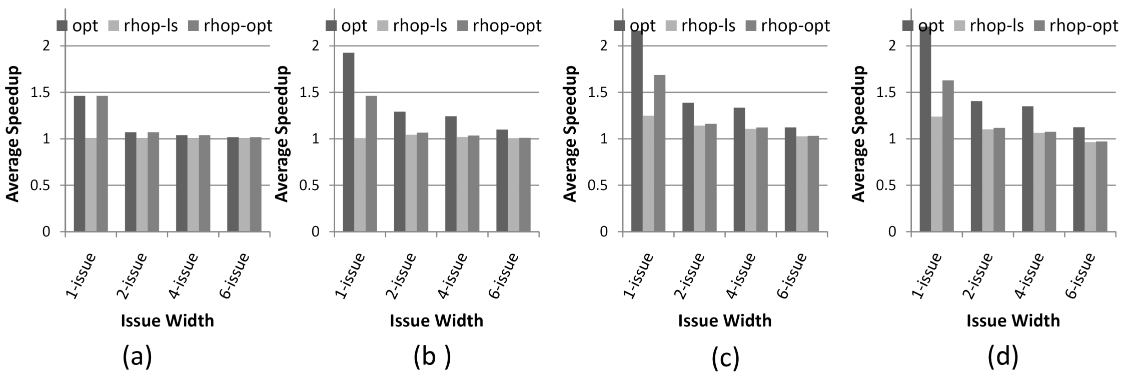


Figure 6. Average speedups of superblocks for the apsi SPEC benchmark for different architectures.

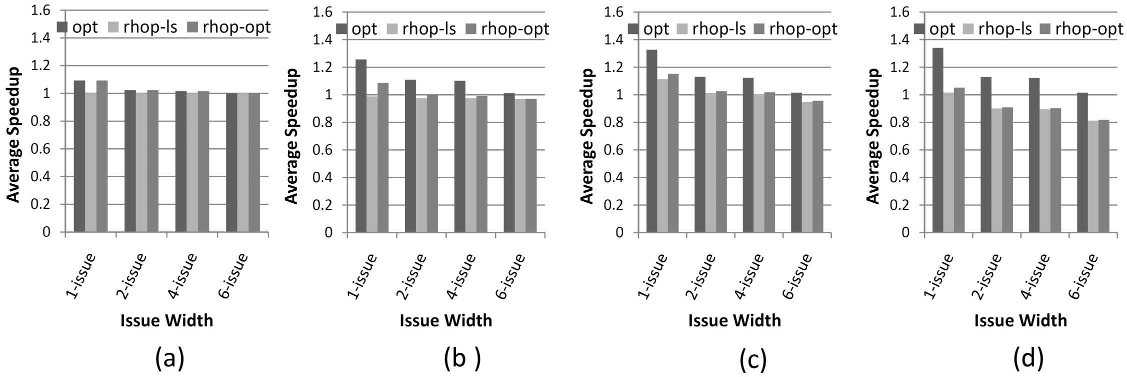


Figure 7. Average speedups of superblocks for the mgrid SPEC benchmark for different architectures.

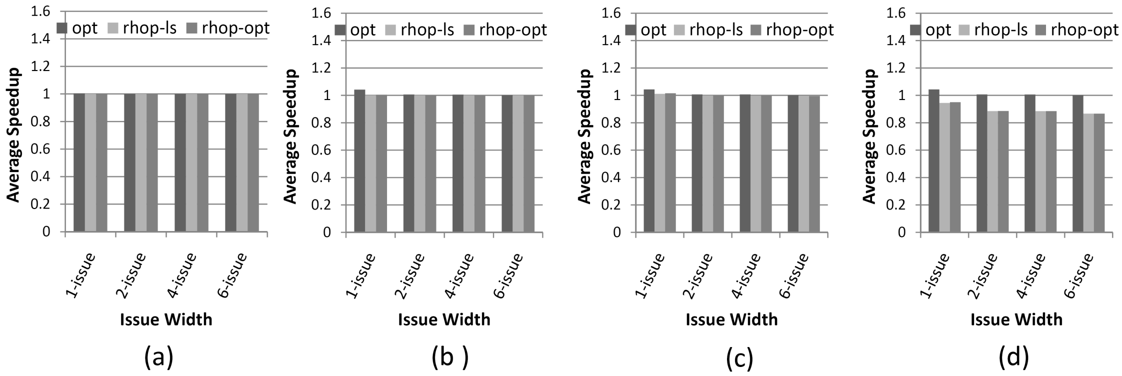


Figure 8. Average speedups of superblocks for the lucas SPEC benchmark for different architectures.

rithm does not give slowdowns in any case where as RHOP regularly dips below the 1.0 line as the number of processors increase.

We also evaluated our algorithm to see what percentage of superblocks it can solve optimally. Table 3 summarizes the percentage of superblocks solved optimally within the set timeout for three benchmarks; apsi, mgrid and lucas. The results show that our algorithm finds it harder to solve for large number of processors with small issue-width and hence times out more often.

6. Discussion

The application of constraint programming to the instruction assignment problem has enabled us to solve the problem optimally for a significant number of code blocks. Solving the assignment problem optimally has an added value over heuristic approaches in instances where longer compilation time is tolerable or the codebase is not very large. This approach can be used in practice for software libraries, digital signal processing or embedded applications. The optimal assignment can also be used to evaluate the performance of heuristic approaches.

The optimal solution also gives the added performance benefits by distributing the workload over processors and the ability to utilize resources that might otherwise remain idle.

7. Conclusions

This paper 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 found that our approach was able to achieve an improvement of 6%-20%, on average, over the state-of-the-art techniques on superblocks from SPEC 2000 benchmarks.

References

[1] A. Aleta, J. M. Codina, J. Sanchez, A. Gonzalez and D. Kaeli. AGAMOS: A graph-based approach to modulo scheduling for clustered microarchitectures. IEEE Transactions on computers, vol. 58, no. 6, June 2009.

[2] K. Andreev and H. Räcke. Balanced graph partitioning. In Proceedings of the Sixteenth annual ACM symposium on Parallelism in Algorithms and Architectures, pages 120-124, 2004.

[3] M. Beg. Instruction scheduling for multi-cores. Student Research Competition at the Conference on Programming Language Design and Implementation, 2010.

[4] J. F. Benders. Partitioning procedures for solving mixed-variables programming problems. Numerische Mathematik 4, 238-252, 1962.

[5] T. Bjerregaard and S. Mahadevan. A survey of research and practices of network-on-chip. ACM Computing Surveys, 38(1), 1-51, 2006.

[6] C. M. Chang, C. M. Chen and C. T. King. Using integer linear programming for instruction scheduling and register allocation in multiple-issue processors. Computers and Mathematics with Applications 34(9):1-14. 1997.

[7] C. Chekuri, R. Johnson, R. Motwani, B. Natarajan, B. Rau and M. Schlansker. Profile-driven instruction level parallel scheduling with application to superblocks. In Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture, pages 58-67, 1996.

[8] H. C. Chou and C. P. Chung. An optimal instruction scheduler for superscalar processors. IEEE Transactions on Parallel and Distributed Systems, 6(3):303-313, 1995.

[9] M. Chu, K. Fan and S. Mahlke. Region-based hierarchical operation partitioning for multicluster processors. PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, pages 300-311, 2003.

[10] M. Chu and S. Mahlke. Compiler-directed data partitioning for multicluster processors. In CGO '06: Proceedings of the International Symposium on Code Generation and Optimization. pp 208–220, 2006.

[11] M. Chu, R. Ravindran and S. Mahlke. Data access partitioning for fine-grain parallelism on multicore architectures. In MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture. pp 369–380, 2007.

[12] G. B. Dantzig and P. Wolfe. Decomposition principle for linear programs. Operations Research 8, pages 101-111, 1960.

[13] A. E. Eichenberger and W. M. Meleis. Balance scheduling:weighting branch tradeoffs in superblocks. In Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture, 1999.

[14] J. R. Ellis. Bulldog: a compiler for VLSI architectures. MIT Press. Cambridge, MA, USA, 1986.

[15] S. Haga and R. Barua. EPIC instruction scheduling based on optimal approaches. In Workshop on Explictly Parallel Instruction Computing Architectures and Compiler Technologies, 2001.

[16] M. Heffernan and K. Wilken. Data-dependency graph transformations for instruction scheduling. Journal of Scheduling, 8:427-451, 2005.

[17] M. Heffernan, K. Wilken and G. Shobaki. Data-dependency graph transformations for superblock scheduling. In Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, pages 77-88, 2006.

[18] B. Hendrickson and R. Leland. A Multilevel Algorithm for Partitioning Graphs. In Supercomputing '95: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing. pp 28, 1995.

[19] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm and D. M. Lavery. The superblock: An effective technique for VLIW and superscalar compilation. The Journal of Supercomputing, 7(1), 229-248, 1993.

[20] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM Journal on Scientific Computing. 201:359–392, 1998.

[21] D. Kastner and S. Winkel. ILP-based instruction scheduling for IA-64. In LCTES '01: Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems pp. 145-154, 2001.

[22] W. Lee, R. Barua, M. Frank, D.Srikrishna, J. Babb, V. Sarkar and S. Amarasinghe. Space-time scheduling of instruction-level parallelism on a RAW machine. In ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems. pp 46–57, 1998.

[23] W. Lee, D. Puppin, S. Swenson and S. Amarasinghe. Convergent scheduling. In MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture. pp 111–122, 2002.

[24] C. Luo, Y. Bai, C. Xu and L. Zhang. FCCM: A novel inter-core communication mechanism in multi-core platform. In Proceedings of International Conference on Science and Engineering, 215-218, 2009.

[25] A. M. Malik, J MacInnes and P. van Beek. Optimal basic block instruction scheduling for multiple-issue processors using constraint programming. International Journal on Artificial Intelligence Tools, 17(1):37-54.

[26] A. M. Malik, M. Chase, T. Russell and P. van Beek. An application of constraint programming to superblock instruction scheduling. Proceedings of the Fourteenth International Conference on Principles and Practice of Constraint Programming. 97–111, 2008.

[27] J. D. Owens, W. J. Dally, R. Ho, D. N. Jayasimha, S. W. Keckler and L. Peh. Research challenges for on-chip interconnection networks. IEEE Micro, 27(5), pages 96-108, 2007.

[28] K. Rich and M. Farrens. Code partitioning in decoupled compilers. In Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing. pp 1008–1017, 2000.

[29] T. Russell, A. Malik, M. Chase, and P. van Beek. Learning heuristics for the superblock instruction scheduling problem. IEEE Transactions on Knowledge and Data Engineering, 21(10):1489-1502, 2009.

[30] F. Rossi, P. van Beek and T. Walsh (Ed). Handbook of Constraint Programming. Elsevier 2006.

[31] G. Shobaki and K. Wilken. Optimal superblock scheduling using enumeration. In Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture, pages 283-293, 2004.

[32] P. van Beek and K. Wilken. Fast optimal instruction scheduling for single-issue processors with arbitrary latencies. Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming, pages 625-639, 2001.

[33] K. Wilken, J. Liu and M. Heffernan. Optimal instruction scheduling using integer programming. In Proceedings of the Conference on Programming Language Design and Implementation, pages 121-133, 2000.