CARS: A New Code Generation Framework for Clustered ILP Processors

Krishnan Kailas[†] Kemal Ebcioğlu[‡] Ashok Agrawala[*]

[†] Department of Electrical & Computer Engineering
[*] Institute for Advanced Computer Studies and
Department of Computer Science
University of Maryland, College Park, MD
{krish,agrawala}@cs.umd.edu

[‡] T. J. Watson Research Center
IBM Corporation
Yorktown Heights, NY
kemal@watson.ibm.com

TECHNICAL REPORT
UMIACS-TR-2000-55 / CS-TR-4165

July 2000

Abstract

Clustered ILP processors are characterized by a large number of non-centralized on-chip resources grouped into clusters. Traditional code generation schemes for these processors consist of multiple phases for cluster assignment, register allocation and instruction scheduling. Most of these approaches need additional re-scheduling phases because they often do not impose finite resource constraints in all phases of code generation. These phase-ordered solutions have several drawbacks, resulting in the generation of poor performance code. Moreover, the iterative/back-tracking algorithms used in some of these schemes have large running times. In this report we present CARS, a code generation framework for Clustered ILP processors, which combines the cluster assignment, register allocation, and instruction scheduling phases into a single code generation phase, thereby eliminating the problems associated with phase-ordered solutions. The CARS algorithm explicitly takes into account all the resource constraints at each cluster scheduling step to reduce spilling and to avoid iterative re-scheduling steps. We also present a new on-the-fly register allocation scheme developed for CARS. We describe an implementation of the proposed code generation framework and the results of a performance evaluation study using the SPEC95/2000 and MediaBench benchmarks.

Keywords: Code generation, Clustered ILP processors, Cluster scheduling, Register allocation, VLIW, Instruction-level parallelism.

1 Introduction

There is a recent interest in statically [1, 2, 3, 4, 5] and dynamically [6, 7, 8, 9] scheduled Clustered ILP processor microarchitectures as a complexity-effective alternative to wide issue monolithic microprocessors for effectively utilizing a large number of on-chip resources with minimal impact on the cycle time. The function units are partitioned and resources such as register file and cache are either partitioned or replicated and then grouped together into on-chip clusters in these processors. All the local register files share the same name space in the replicated register file scheme, whereas in the partitioned register file scheme each one of the local register files has a unique name space. The clusters are usually connected via a set of inter-cluster communication buses or a point-to-point network [10].

Several resources are required to execute an operation (OP) in a clustered processor. As in a single-cluster processor, the OPs need local resources in the cluster such as function units for execution and registers/memory to save the results. In addition to this, the OPs often need shared resources such as the inter-cluster communication mechanism to access those operands that reside in remote clusters. Some form of copy operation (using either hardware techniques [8] or inter-cluster copy OPs) needs to be explicitly scheduled to access a remote register file in the case of a partitioned register file scheme, or to maintain coherency in the case of a replicated register file scheme. Clearly, a good code generation scheme is very crucial to the performance of these processors in general and especially for statically scheduled clustered ILP processors in which each one of these local and shared resources has to be explicitly reserved by the code generator on a cycle-by-cycle basis.

The basic functions[1] that must be carried out the by the code generator for a clustered ILP processor are: 1) cluster assignment, 2) instruction scheduling, and 3) register allocation. All of the three functions are closely inter-related to each other. If these functions are performed one after another, often their ordering can have a significant impact on the performance of the generated code. For example, register allocation can affect cluster assignment and vice versa. This is because the register access delays (due to inter-cluster copy OPs) of an OP are dependent on the proximity of the cluster in which the operand register is defined to the cluster that tries to access it. The ordering of cluster assignment and instruction scheduling steps can also affect the performance of the compiled code. If scheduling is done before cluster assignment, it may not be possible to incorporate inter-cluster copy OPs in the schedule made by the earlier scheduling step, often necessitating a re-scheduling step after the cluster assignment. Cluster assignment of an OP depends on the ready times of its operands and the availability of resources, which in turn depend on the cycle in which the OPs that define the operands are scheduled. Therefore, cluster assignment, if carried out before the scheduling step can often result in poor resource utilization and longer schedule lengths. There are several problems with the approaches using separate phases for register allocation and instruction scheduling [11, 12]. Global register assignment, if carried out first, can create unnecessary dependences due to re-definition of registers, thereby restricting

[1]We assume that operation selection has been already made.

the opportunities for extracting ILP by instruction scheduler. Instruction scheduling if performed before register assignment can result in inefficient use of registers, thereby increasing the register pressure, possibly causing unnecessary spills.

In general, phase-ordered solutions at each phase make a "best effort" attempt to get a feasible cluster schedule, often making unrealistic assumptions such as infinite resources (registers, function units, etc) or zero time copy operations resulting in poor performance code. Clearly, this *phase-ordering problem* can affect the performance of the code generated for clustered processors. An alternative approach is to iterate the cluster scheduling process until a feasible schedule meeting some performance criteria is reached. The main drawback of these approaches is their large running time.

In this report, we introduce a new code generation framework for clustered ILP processors called CARS (*Combined cluster Assignment, Register allocation and instruction Scheduling*). In CARS, as the name suggests, the cluster assignment, register allocation and instruction scheduling phases of traditional code generation schemes are performed concurrently in a single phase, thereby avoiding the drawbacks of the phase-ordered solutions mentioned above. In order to maximize the extraction of instruction-level parallelism (ILP), independent OPs (that do not cause exceptions) are often cluster scheduled out-of-order in CARS. To facilitate this as well as to combine register allocation with cluster scheduling, we developed a new on-the-fly register allocation scheme. The scheme does not rely on any information that depends on predetermined relative ordering of OPs such as the live ranges and interference graphs used by traditional register allocators. Our preliminary experimental results indicate that CARS generates efficient code for a variety of benchmark programs across a spectrum of eight different clustered ILP processor configurations.

Roadmap: In section 2 we describe the CARS framework and algorithms. The details of an implementation of CARS are given in section 3. We discuss the related work in section 4. Preliminary results of an experimental evaluation of CARS are given in section 5, followed by some comments and conclusions in section 6.

2 Combined Cluster Assignment, Register Allocation and Instruction Scheduling

2.1 Overview

A generic clustered ILP processor model is shown in figure-1. In this report we assume a partitioned register file architecture with local register files containing registers with unique/private name space. However, our scheme can be easily adapted for replicated register file architectures as well [13]. We assume that an OP can only write to its local register file and an explicit inter-cluster copy OP is needed to access a register from a remote cluster. These copy OPs use communication function units (a local cluster resource) and inter-cluster communication network (a shared global resource). Either single/multiple shared buses or point-to-point network may be used for inter-cluster communication.

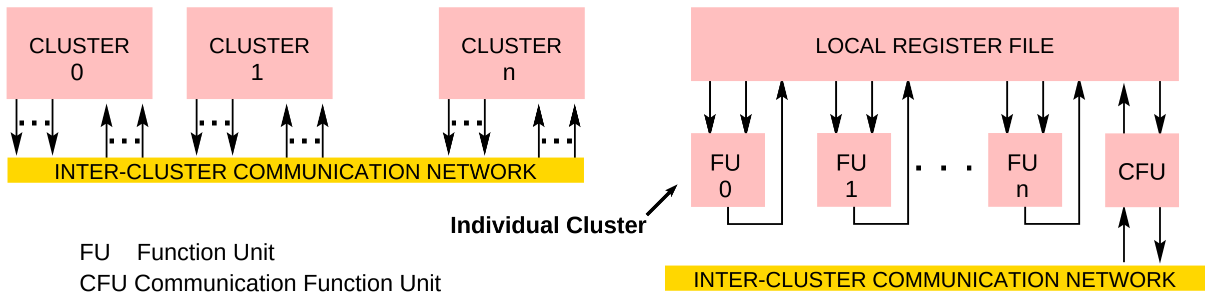
CLUSTER 0
CLUSTER 1
CLUSTER n
INTER-CLUSTER COMMUNICATION NETWORK
FU Function Unit
CFU Communication Function Unit
Individual Cluster
LOCAL REGISTER FILE
FU 0
FU 1
FU n
CFU
INTER-CLUSTER COMMUNICATION NETWORK

Figure 1: Generic Clustered ILP Processor Model.

Our code generation framework consists of 3 stages as shown in figure-8(b). In the first stage some of the data structures required for the CARS algorithm are set up and initialized. The real work is done in the second stage in which the combined cluster assignment, register allocation and instruction scheduling (henceforth referred to as CARS*cheduling*) is carried out, followed by the final code printing stage with peephole optimizations.

The input to the code generator is a dependence flow graph (DFG) [14] with nodes representing OPs and directed edges representing data/control flow. The DEF-USE relationship between nodes is represented using the static single assignment (SSA) form [15] with a join and fork node at the beginning and end of basic blocks respectively. Join node DEFs represent *phi*s (ϕs) of SSA and fork node DEFs represent *anti-phi*s (ϕ^{-1}s). Join nodes are also scheduled by CARS, even though they do not consume any resources or issue slots, for implementing on-the-fly register allocation (explained in section 2.3).

The basic scheduling unit for CARS can be a set of OPs created using any of the "region" formation or basic block grouping heuristics of the various global scheduling schemes such as superblock scheduling [16], hyperblock scheduling [17], treegion scheduling [18] or execution-based scheduling [19]. These scheduling units, which we refer to as *regions*, are selected for scheduling strictly in topological order. OPs within a region are scheduled in top down fashion. The output of the code generator is a directed graph of tree-VLIWs [20].

During code generation CARS dynamically partitions the OPs in the scheduling region into a set of mutually exclusive aggregates (groups) — `unscheduled`, `ready list`, and `vliws` [2]. Initially all the nodes in the DFG belong to the `unscheduled` aggregate. The *data ready*[3] nodes in the `unscheduled` aggregate are identified and moved into the `ready list`. The nodes in the `ready list` are selected based on a heuristic for CARScheduling. After CARScheduling, the nodes are moved to the appropriate `vliws`. This process is repeated until all the nodes of the DFG are scheduled.

2.2 Pre-CARS initializations

The pre-CARS initialization stage is used for preprocessing information needed by the CARS algorithm. For each node we compute its height and depth in the DFG, based on the height and depth of its dependent successor/predecessor nodes and its latency. Depth is the earliest execution cycle

[2]`vliws` is a tree of aggregates.

[3]A node becomes *data ready* when all of its dependent predecessor nodes are scheduled.

of the OP, counting from the beginning of the DFG; height is the latest execution cycle of the OP, counting from the end of the DFG, in an infinite resource machine. Associated with each SSA DEF that needs to be register allocated, we maintain a `RegMap` structure. Inside the `RegMap` of a DEF, we keep the number of uses (`use_count`) and the ID of the preferred register (`prfrd_reg_map`) to be assigned to the DEF. DEFs and USEs of certain nodes such as the ones at the entry and exit of the procedure and call OPs must be assigned to specific registers as per the calling convention. We mark such DEFs and USEs with a flag and initialize their `prfrd_reg_map` to the ID of the corresponding mapped register. The register allocator of CARS uses another field of `RegMap`, the *register mask bitvector* (`regmask_bv`) to prevent a set of registers from being allocated to certain DEFs. For example, we initialize `regmask_bv` of those DEFs that are live across a call OP so that caller-save registers will not be allocated to them. We also identify and flag loop-invariant DEFs, back-edge DEFs and loop-join-DEFs of nodes in loops. This information will be used by CARS, for example to eliminate copy OPs along the back-edges of loops [13]. Physical registers are treated as a resource in CARS. Based on the input parametric description of the machine model, we initialize the register resource structures and their bit-vector representations, local resource counters for function units and global resource counters for shared resources such as inter-cluster buses.

2.3 On-the-fly register allocation in CARS

Registers are allocated on-the-fly in CARS without using live range information [21] or explicit interference graphs [22]. In order to do this, it maintains and dynamically updates 1) the remaining number of uses (`ruse_count`) of each physical register, 2) the availability of registers (`lcl_reg_bv` and `glbl_reg_bv`), and 3) preferred register mapping (`prfrd_reg_map`) of DEFs, as explained below.

We use `ruse_count` to identify and mark when a live register becomes dead. The `ruse_count` of a register is decremented whenever an OP that uses it is scheduled; when `ruse_count` becomes zero we mark the register as dead. Since the only information we have at any time during scheduling is the pre-computed `use_count` of SSA DEFs in the current scheduling region, we initialize and dynamically update the `ruse_count` as follows. As we start scheduling a new region, the DEFs (ϕs) of the join node are allocated the same register its scheduled predecessor forks' DEFs (ϕ^{-1}s) are allocated. The pre-computed `use_count` of join DEF is then added to its allocated register's `ruse_count`. Similarly, prior to scheduling fork nodes at the exits of a region we update the `ruse_count` of registers used by ϕ^{-1}s of fork node. The number of unvisited join ϕs connected to the fork's ϕ^{-1} is added to the `ruse_count` of the ϕ^{-1}'s mapped register. This prevents marking the registers allocated to DEFs that are live beyond the current scheduling region as dead (see the example in figure-2).

In addition to `ruse_count`, the availability (live or dead status) of registers are also maintained in two bit-vectors – one representing the global status (`glbl_reg_bv`) and the other representing the local (i.e., within the scheduling region) status (`lcl_reg_bv`). All registers that are not used in the current scheduling region are marked as dead in `lcl_reg_bv`, whereas the status of registers in `glbl_reg_bv` does not depend on whether they are used in the current scheduling region or not. We use the information in these two bit-vectors to identify non-interfering lifes in the current

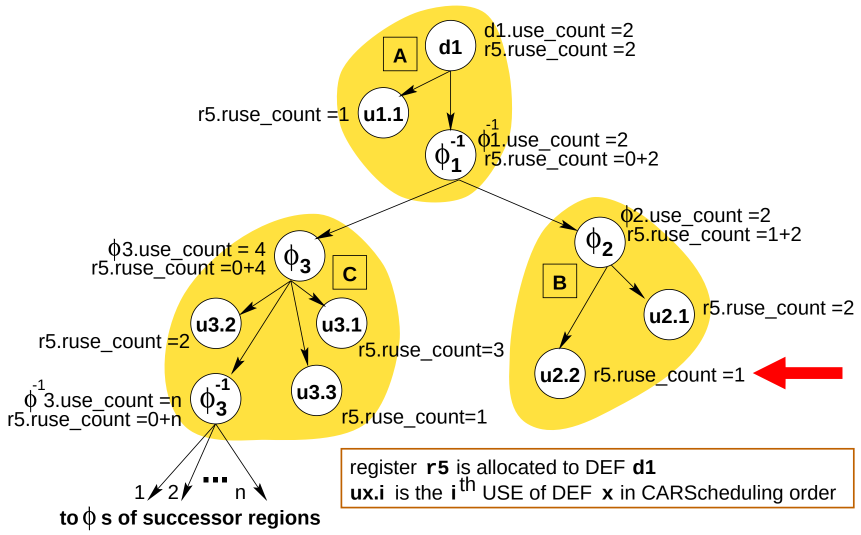

Figure 2: An example illustrating `ruse_count` update process in CARS. Regions are CARScheduled in the order: A $\rightarrow$ B $\rightarrow$ C. The register `r5` is not marked dead even after scheduling its last use u2.2 in B.

scheduling region for efficient use of registers as in a graph coloring based allocator [22]. Registers that are marked dead in `glbl_reg_bv` may be allocated to any DEF if they are not masked by DEF's `regmask_bv`. Also, a register from the set of registers that are marked as dead in `lcl_reg_bv` but live in `glbl_reg_bv` may be allocated to a DEF, if the DEF is not live beyond the current region and the register does not belong to the set of preferred registers of all ϕ^{-1}s at the exits of the region. All of the above logic can be implemented by a set of logical operations on bit-vectors [13].

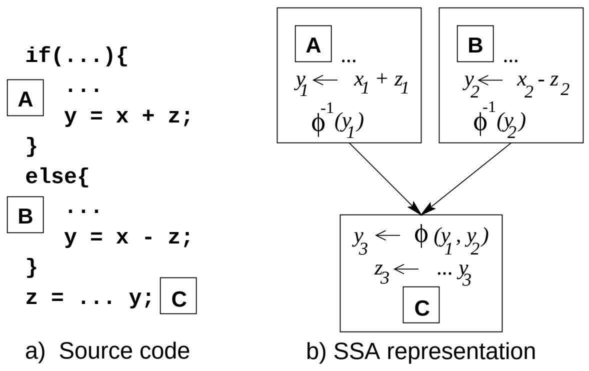

Figure 3: An example illustrating the problem due to different register mappings: If the SSA DEFs y_1 and y_2 are mapped to `r10` and `r11` respectively, then $y_3 \leftarrow \phi(y_1, y_2)$ becomes a non-identity assignment and hence cannot be ignored. To fix this problem, we have to insert a copy OP `r11` $\leftarrow$ `r10` in block $\boxed{\text{A}}$ or `r10` $\leftarrow$ `r11` in block $\boxed{\text{B}}$.

The ϕs of a join node require copy OPs along its incoming edges if all of its predecessor ϕ^{-1}s were not allocated to the same register (see the example in figure 3). To avoid these copy OPs we use the dynamically generated `prfrd_reg_map` information in the `RegMap` of DEFs as follows. The `prfrd_reg_map` of each DEF which is live beyond the current region is initialized when the DEF is register allocated. This `prfrd_reg_map` information is propagated to all the *directly* and *indirectly connected* ϕs of join nodes as shown in the figure 4. (The set of directly and indirectly connected join DEFs may be thought of as a "web" of intersecting DEF-USE chains [23] – an

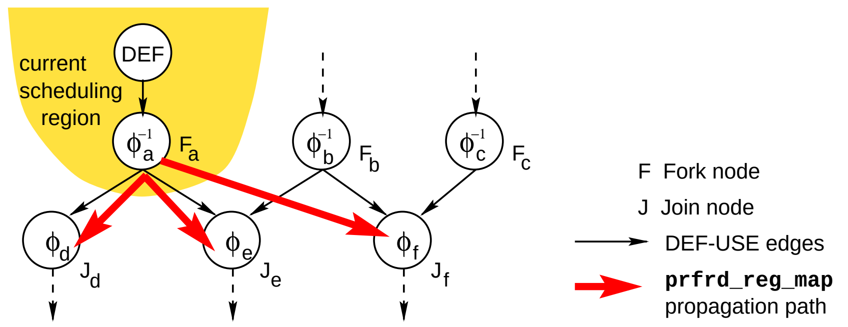

Figure 4: `prfrd_reg_map` propagation: ϕ_d and ϕ_e are the directly connected and ϕ_f is the indirectly connected ϕs of ϕ_a^{-1}.

object for register allocation in graph coloring based allocators). In general, if a DEF has an ϕ^{-1} use and there exist a valid `prfrd_reg_map` for any of the ϕs connected to this ϕ^{-1}, then we allocate the register as specified by that `prfrd_reg_map` to the DEF. Otherwise, we allocate a register that is dead in `glbl_reg_bv` and propagate the `prfrd_reg_map` to all connected ϕs as explained above. Because the regions are scheduled strictly in topological order and registers that are live beyond the current region are never marked dead, this scheme almost completely eliminates unnecessary copy OPs that might otherwise be needed to handle mismatches in the register mapping of incoming edges of ϕs.

To those DEFs that have a fixed register mapping due to our calling convention, we allocate registers as per their `prfrd_reg_map` initialized in the pre-CARS stage. Copy OPs are inserted on-the-fly during CARScheduling for those DEFs that are live across call OPs, if necessary, by preempting the scheduling of call OP.

To avoid pseudo name dependencies we select registers for assignment in a round-robin fashion using an efficient data structure [13]. This data structure also allows us to quickly search for dead registers in a cluster. Inside each physical register's resource structure, we also maintain a list of SSA DEFs (belonging to the same live range) that are currently mapped to the register. This information is used for spilling a live range.

Spilling: If there are no dead registers to assign to the DEF(s) of any of the OPs in the `ready list`, then a live register is selected for spilling, based on a set of heuristics [13]. In order to spill the live range mapped to a register, spill store OPs are inserted in the DFG for the scheduled DEFs (if any) mapped to the register. These spill store OPs will be merged to one of the `vliws` aggregates by the peephole compaction routine after CARScheduling. The spilled register (now residing on stack) is renamed to a unique ID outside the name space of all local register files. This facilitates easy identification of the DEFs mapped to spilled registers and their uses in the un-scheduled regions, so that spill load/store OPs can be inserted in the DFG and CARScheduled on-the-fly.

2.4 The CARS Algorithm

The CARS algorithm is given in figure-5, which is a modified version of the list-scheduling algorithm [24]. In order to find the best cluster to schedule an OP, we first compute the resource-constrained *schedule_cycle* (lines 3-5 of figure-5) in which the OP can be scheduled in each cluster

Algorithm 1 CARS: list-scheduler version

```
1: while number of OPs in unscheduled $\neq 0$ do
2:    select an $Op$ from ready list
3:    for $i = 0$ to $MAX\_CLUSTERS$ do
4:       compute resource-constrained $schedule\_cycle[i]$ for $Op$
5:    end for
6:    $earliest\_cycle = \min\{schedule\_cycle[i]\}$
7:    if $earliest\_cycle \leq current\_vliw\_cycle$ then
8:       update depth of OPs and processor resource counters
9:       allocate register(s) to DEF(s) of $Op$
10:      assign cluster and schedule $Op$ in the current vliws aggregate
11:      insert and cluster schedule copy OP(s) (if required)
12:      update ready list
13:   else
14:      increment $scheduling\_attempts$ of $Op$
15:      move $Op$ back to ready list
16:   end if
17:   if $current\_vliw\_cycle < \min\{$ depth of OPs in vliws$\}$ then
18:      open a new vliws aggregate and increment $current\_vliw\_cycle$
19:   end if
20: end while
```

Figure 5: CARS algorithm.

based on the following factors: 1) the cycle in which its operands are defined, 2) the cluster in which its operands are located, 3) the availability of function unit in the current cycle, 4) the availability of destination register, and 5) whether inter-cluster copy OP(s) can be scheduled or not on the source node's cluster in a cycle earlier than the current cycle. Based on the `earliest_cycle` computed (line 6), the OP will be either scheduled in the current cycle on one of the clusters[4] corresponding to `earliest_cycle` (lines 7-10) or pushed back into the `ready list` after incrementing its *scheduling_attempts* (lines 13-16). A new `vliws` aggregate will be created if none of the OPs in the `ready list` can be scheduled in the `current_vliw_cycle` (lines 17-19). This process is repeated until all OPs in the `unscheduled` aggregate are cluster-scheduled.

We use one of the commonly used heuristics – schedule those data ready OPs that are on the critical path first – for selecting an OP from the `ready list` (line 2 of figure-5). The sum of the OP's height and depth is used to identify OPs that are likely to be in the critical path(s) and to assign priority to the data ready OPs. The *scheduling_attempts* variable associated with each OP (updated in line 14) is used to change the OP selection heuristics so that no OP in the `ready list` will be repeatedly considered in succession for CARSscheduling. This also ensures termination of the algorithm. The depth of a scheduled OP is often increased after scheduling due to finite resources available in each cycle, causing the set of OPs in the critical path(s) to change dynamically during the cluster-scheduling process. Therefore, in order to greedily select OPs in the critical paths first, after scheduling each OP, we update the depth of the OP and the depth of all of its dependent nodes that became data ready as a result of scheduling the OP (lines 8 and 12 of figure-5). Coupled with the prioritized selection of nodes in the critical path, this fully resource-aware cluster scheduling

[4]If the OP can be scheduled on more than one cluster in the `earliest_cycle`, then a cluster is selected for assignment based on a set of heurisitcs [13].

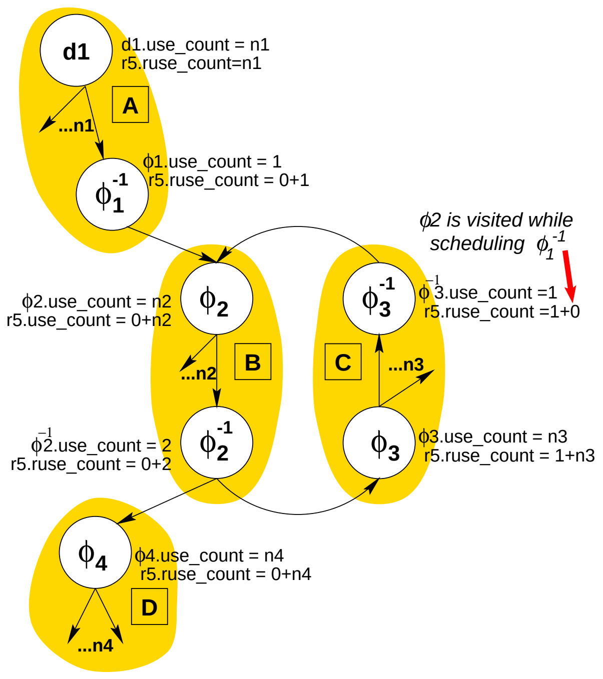

Figure 6: An example illustrating how `ruse_count` is updated during on-the-fly register allocation of a live range which is live at the exit of the loop. Register `r5` is allocated to DEF d1. Regions are CARSscheduled in the order: A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D.

approach lets CARS assign and schedule OPs in the critical paths in appropriate clusters such a way that the stretching of critical paths is minimal and subject to the finite resource constraints of target machine.

Often inter-cluster copy OPs have to be inserted in the DFG and retroactively scheduled in order to access operands residing in remote clusters (line 11 of figure-5). We use an operation-driven version of the CARS algorithm for this purpose. In order to find the best VLIW to schedule the copy OP, the algorithm searches all the `vliws` aggregates starting from the DEF cycle of the operand (or from the cycle in which the join node of the current region is scheduled, if operand is not defined in the current region) to the current cycle. We use the *tree VLIW instruction* [20] representation so that independent OPs from multiple regions can be scheduled in the same `vliws` aggregate. Due to lack of space, we will not further describe the details of operation-driven CARS algorithm and tree-VLIW scheduling, which may be found elsewhere [13].

2.5 CARSscheduling cyclic regions

Loops are CARScheduled similar to acyclic DFG. A loop-head join node becomes data ready when all of its predecessor fork nodes, except the back-edge fork node(s), are scheduled. The `ruse_count` update process is exactly similar to CARScheduling acyclic regions as illustrated in figure 6. However, some additional techniques are required to avoid unecessary insertion of copy OPs along the back-edges of loops due to mismatches in register mappings. In the following, we first describe the problem using an example and then present our solution which uses the information obtained from pre-CARS initialization pass.

```
for(i = a; i < b; i++){
    ...
}
... = i ...;
```

a) Source code of loop.

A: $i_1 \leftarrow a$
   goto B

B: $i_2 \leftarrow \phi_2(i_1, i_4)$
   if $(i_2 \geq b)$ goto D
   ...
   $i_3 \leftarrow i_2 + 1$
   goto C

C: $i_4 \leftarrow \phi_3(i_3)$
   goto B

D: $i_5 \leftarrow \phi_4(i_2)$
   ...

b) Loop in SSA form.

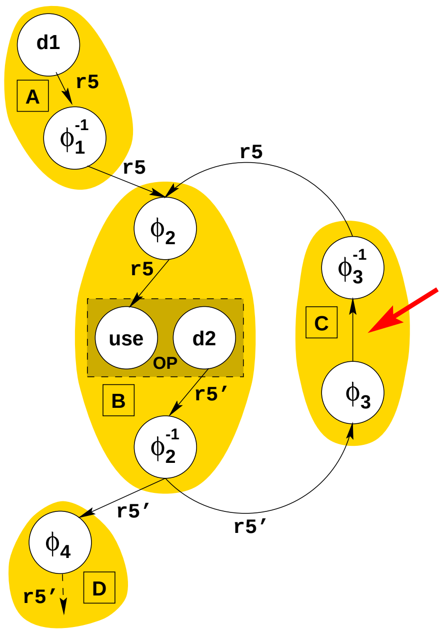

Figure 7: An example illustrating `prfrd_reg_map` propagation during on-the-fly register allocation of a loop. Regions are CARScheduled in the order: A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D. Register `r5` is allocated to DEF `d1`. If a different register `r5`$'$ is allocated to DEF `d2`, then a copy OP `r5`$'\rightarrow$`r5` must be inserted along the back-edge $\overline{\phi_3\,\phi_3^{-1}}$.

Let us consider the live ranges associated with the loop index computation shown in figure 7. We call the DEF which is connected to the loop-head join via back-edge of the loop, such as the DEF `d2` of the OP which updates the loop index, as a *back-edge DEF*. The back-edge DEF `d2` does not have any preferred register assigned at the time of CARScheduling its OP, because the successor join DEFs (ϕ_3 and ϕ_4) of `d2`'s fork use ϕ_2^{-1} do not have their `prfrd_reg_map`s initialized yet. Consequently, the register allocator may choose a register `r5`$'$, which is different from the one (`r5`) mapped to ϕ_2, for allocation to `d2`. This necessitates a copy OP such as `r5`$'\rightarrow$`r5` to be inserted along the back-edge of the loop to take care of the mismatch between register maps of ϕ_3 and ϕ_2. CARS eliminates such copy OPs as follows. During the pre-CARS pass, for all back-edge DEFs that are not loop-invarient (such as `d2`), we identify the loop-head join DEF that can be reached via the back-edge of the loop (ϕ_2 in our example), and save this information in the back-edge DEF's `RegMap` structure. Whenever such a back-edge DEF is encountered during CARScheduling, we try to allocate it the register assigned to its associated loop-head join DEF. In the above example, hence we allocate `r5` to `d2`, since the join DEF associated with `d2` is ϕ_2, thereby eliminating the copy OP `r5`$'\rightarrow$`r5`. If there are no OPs in the back-edge block after scheduling the loop body, the scheme also help eliminate an extra branch by deleting the empty back-edge block.

3 Implementation

We have implemented a code generator based on CARS on top of CHAMELEON [25, 26] VLIW research testbed. The input to CHAMELEON is object code (.o files) produced by a modified version of **gcc** compiler. An *object-code translator* processes these .o files to generate an assembly-like sequential representation. The modified version of the VLIW compiler with the CARS-based backend takes this sequential code as input and outputs VLIW code (tree-instructions) as shown in Figure-8(a).

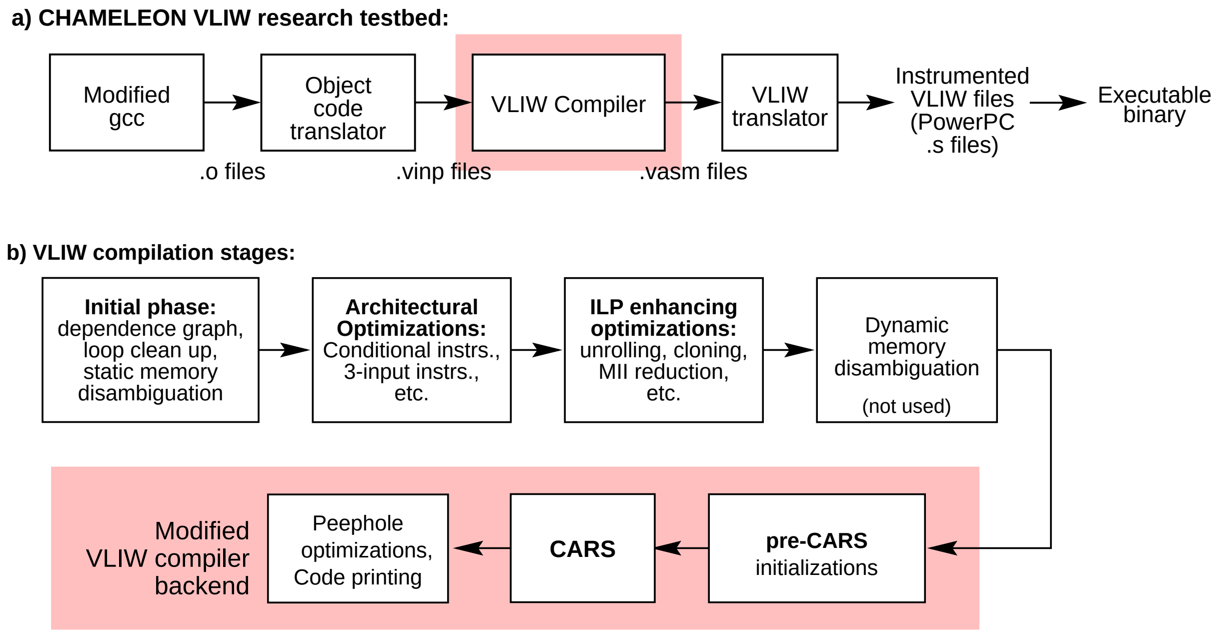
a) CHAMELEON VLIW research testbed:
Modified gcc
.o files
Object code translator
.vinp files
VLIW Compiler
.vasm files
VLIW translator
Instrumented VLIW files (PowerPC .s files)
Executable binary
b) VLIW compilation stages:
Initial phase: dependence graph, loop clean up, static memory disambiguation
Architectural Optimizations: Conditional instrs., 3-input instrs., etc.
ILP enhancing optimizations: unrolling, cloning, MII reduction, etc.
Dynamic memory disambiguation (not used)
pre-CARS initializations
CARS
Peephole optimizations, Code printing
Modified VLIW compiler backend

Figure 8: Implementation of CARS on the CHAMELEON VLIW research testbed.

The VLIW compiler first builds the DFG and then performs a series of optimizations as shown in figure-8(b). Compilation is performed at procedure level and without function in-lining. The prologue and epilogue code are added to the DFG after CARSscheduling the procedure. These pro/epilogue OPs are then cluster scheduled using CARS. The **`vliws`** aggregates are then passed through a peephole optimizer and to the final code printing stage. The output of CARS, the tree-VLIWs, are then instrumented and translated into PowerPC assembly code that emulate the target Clustered ILP processor.

A parametric description of the target clustered machine model can be specified to the code generator. The number, type and latency of function units in each cluster, any kind of arbitrary mapping of register name space to local register files, number, type and latency of global interconnect are some of the configurable parameters that are currently supported by our code generator. A subset of the machine resource information is maintained on a per-VLIW basis during code generation.

As of writing this report spilling is not fully implemented. Currently, akin to the Bulldog compiler [27], our compiler exits if an OP cannot be scheduled even after making a constant number of repeated scheduling attempts. However, by controlling the extent of loop-unrolling, we are able to compile 95-100% of all functions in the benchmark programs that we have used in the experimental evaluation of CARS (section 5). Moreover, in single cluster configuration, CARS could compile more number of functions without spilling compared to the unmodified version of CHAMELEON compiler (which performs register allocation after scheduling for single cluster machines). This clearly shows the ability of CARS to pick different OPs from the `readylist` until a register is available to schedule an OP. While compiling for clustered machines, the CARS algorithm automatically migrates computation to a cluster with lower register pressure.

4 Related work

To the best of our knowledge CARS is the first code generation scheme that combines the cluster assignment, instruction scheduling and register allocation phases for a partitioned register file clustered ILP processor.

Solutions for phase-ordering problem: Several schemes have been proposed to combine different phases of code generation for clustered as well as non-clustered processors. The most recent one, the UAS algorithm [28, 29] for clustered VLIW processors by Ozer, Banerjia and Conte performs cluster assignment and scheduling of instructions in a single step, using a variation of list scheduling. The UAS algorithm, however, does not consider registers as a resource during cluster scheduling. In contrast, the CARS algorithm treats registers as one of the resources and performs on-the-fly register allocation along with cluster scheduling in a resource-constrained manner.

A variety of techniques have been proposed for combining register allocation and code scheduling of single cluster processors. Goodman and Hsu proposed an integrated scheduling technique in which register pressure is monitored to switch between two scheduling schemes [30]. Operation-driven version of the CARS algorithm is motivated by this work. However, unlike their scheme, we switch to operation-driven scheduling only for scheduling inter-cluster copy OPs. Bradlee et al [31] proposed a variation of the Goodman-Hsu scheme and another technique. Pinter [32] proposed a technique that incorporates scheduling constraints into the interference graph used by graph coloring based allocators. Berson et al [33] proposed a technique based on measuring the resource requirement first and then using that information for integrating register allocation in local as well as global schedulers. Brasier et al proposed a scheme called CRAIG [34] that makes uses of information obtained from a pre-pass scheduler for combining scheduling and register allocation phases. The compiler for TriMedia processor uses a technique much like the scheme in [33] to combine register allocation and scheduling [35]. Hanono and Devadas [36], and Novak et al [37] proposed code generation schemes for embedded processors, which combine the code selection, register allocation, and instruction scheduling phases. Early examples of techniques that did scheduling and register allocation concurrently for single cluster VLIW machines include the resource-constrained scheduling scheme by Moon and Ebcioğlu [38]. The fundamental difference between our scheme and all the above are: 1) the use of register mapping information and separate local and global register status during CARSscheduling, and 2) combining all the three phases involved instead of two. The vLaTTe compiler handles fast scheduling and register allocation together in the context of a JAVA JIT compiler for a single cluster VLIW machine [39].

Register allocation: Local register allocation via usage counts [40] is a well known technique. More recently, a number of fast global register allocation schemes have been proposed. For example, the Linear Scan register allocation scheme [21] by Poletto and Sarkar use *live interval* information to allocate registers in a single pass. All these schemes and the graph coloring based allocators [22] need the information about the precise ordering of OPs for computing interfering live ranges which is only available after scheduling. The on-the-fly register allocation scheme used in CARS is not based on any such information that is available only after scheduling the entire DFG.

The preferred register map approach in CARS is similar to the scheme proposed by Yang, Moon et al [41] for the LaTTe just-in-time compiler. However, LaTTe makes two passes (a "backward sweep" collects information on preferred registers for OPs and a "forward sweep" performs register assignment) for local register allocation of *tree regions* and it needs copy OPs due the mapping mismatches (as illustrated in Figure 3). In contrast, CARS performs global register allocation in a single CARScheduling pass using pre-computed `use_count` of DEFs. Moreover, CARS by design tries to prevent the mapping mismatches.

Cluster Scheduling: Pioneering work in code generation for clustered VLIW processors is done by Ellis [27]. The Multiflow compiler [42] performs cluster assignment using a modified version of the Bottom-Up Greedy (BUG) algorithm proposed by Ellis in a number of steps and then performs register allocation and instruction scheduling in a combined manner. Desoli's Partial Component Clustering (PCC) algorithm [43] for clustered VLIW DSP processors is an iterative algorithm that treats the clustering problem as a combinatorial optimization problem. In the initial phases of the PCC algorithm, "partial components" of DAG are grown and then these partial components are assigned to clusters much like the cluster scheduling scheme using components, equivalent classes and virtual clusters [11] of the Multiflow compiler. In the subsequent phases, the initial cluster assignments are further improved iteratively. In contrast, the cluster assignment approach of our scheme is fundamentally different from the recursive propagation of preferred list of functional units and cluster assignment as in the BUG algorithm. Cluster assignment, register allocation, and instruction scheduling are performed concurrently in a resource-aware manner in CARS.

Another work is the cluster scheduling for the Limited Connectivity VLIW (LC-VLIW) architecture [44]. Cluster scheduling for the LC-VLIW architecture is performed in three phases. In the first phase, the DAG is built from the compiled VLIW code for an ideal single cluster VLIW processor. The second phase uses a min-cut graph partitioning algorithm for code partitioning. In the third phase, the partitioned code is recompacted after inserting copy operations.

The dynamic binary translation scheme used in DAISY performs "Alpha [6] style" cluster scheduling (without using inter-cluster copy OPs) along with register allocation for a duplicated register file architecture [19, 3].

Multiprocessor Scheduling: A large number of DAG clustering algorithms have been proposed for multiprocessor task scheduling in the past [45]. Sarkar's partitioning algorithm [46] and the more recent ones such as Dominant Sequence Clustering (DSC) [47] and CASS-II [48] are examples of such algorithms. The input to these algorithms is a DAG of tasks with *known* edge weights corresponding to the inter-node communication delays. Clustering is carried out in multiple steps. In the first step, these algorithms assume an infinite resource machine and each node is assumed to be in a different cluster. A sequence of refinement steps are performed in the second step, in which two clusters are merged by "zeroing" the edge weight (communication delay between nodes) based on different heuristics. The clusters produced in the previous steps are merged together in the third step so that the resulting number of clusters does not exceed the number of processors in the multiprocessor. The compiler for the MIT RAW project, RAWCC [49], employs a greedy technique based on the DSC algorithm for performing cluster scheduling in multiple phases. In contrast, the CARS algorithm performs cluster scheduling with register allocation in a single pass, always assuming a finite resource machine. We use the same heuristics used by DSC, CASS-II and BUG algorithms to select data ready nodes in the critical path for cluster scheduling.

Eichenberger and Nystrom proposed an iterative modulo scheduling scheme for clustered processors [10]. Fernandes et al proposed a distributed modulo scheduling scheme for clustered VLIW machines that uses register queues for inter-cluster communication [50].

5 Experimental Results

SPEC CINT95	MediaBench	SPEC CINT2000
099.go	rawcaudio	181.mcf
124.m88ksim	rawdaudio	197.parser
126.gcc	g721.encode	
129.compress	g721.decode	
130.li	epic	
132.ijpeg	unepic	
134.perl		
147.vortex		

Table 1: Benchmark programs

Function unit	Latency
Fix	1
FP	2
FP divide	9
FP sqrt	9
Load	1
Branch	1
Communication	1

Table 2: Latencies of function units.

We used a set of programs[5] listed in Table-1 from the SPEC95 [51], MediaBench [52] and SPEC2000 [53] benchmark suites for performance evaluation. The compiled simulation binaries are run to completion with the input data sets of corresponding benchmark programs. These instrumented binaries upon execution provide the number of times each tree VLIW instruction and each path in it are executed. For comparing the code generated for different clustered machine configurations we used the total number of VLIWs executed as a metric, which corresponds to the infinite cache execution time in cycles.

We used two base configurations – both are single cluster machines with 8 and 16 function units with latencies as listed in Table-2. Small changes in function unit latencies do not have much effect on the relative performance of code generated for single cluster vs. multi-cluster machines using CARS. The register file size of both base configurations are identical: 64 INT, 64 FP and 16 condition bit registers. The 8 and 16 ALU machines can issue 8 and 16 OPs per cycle respectively, of which at most 3 can be branch OPs on both configurations. The clustered machines are configured such that the issue width and resources of the base machine are evenly divided and assigned to each cluster. We compared the number of cycles taken to execute the code generated

[5]The functions that cannot be compiled due to high register pressure in these programs are compiled separately and treated as system calls for the compiled binary simulation. The same set of functions are excluded for all the machine configurations studied even though a subset of them could be compiled without spilling due to the variations in register pressure of different machine configurations.

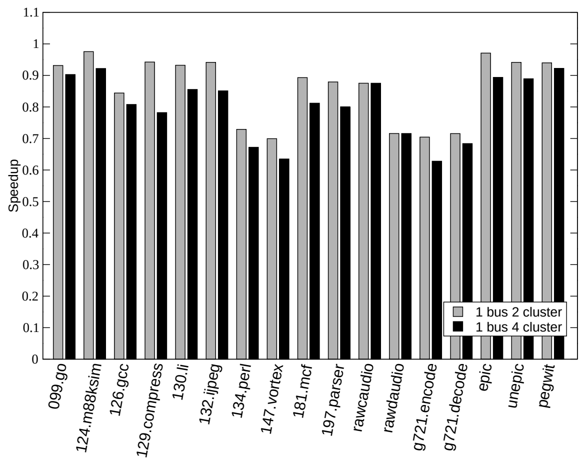

Figure 9: Speedup for 8-ALU 1-bus configurations over single cluster.

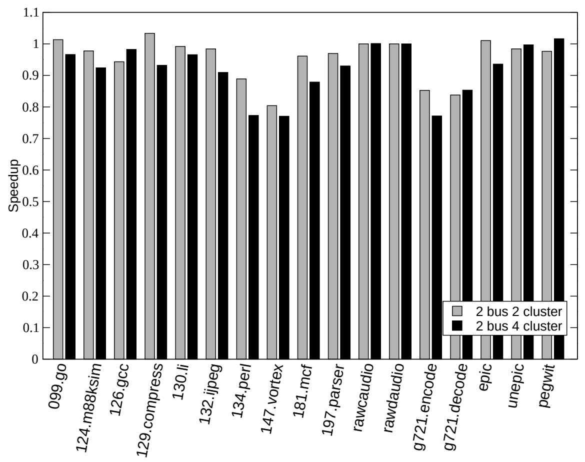

Figure 10: Speedup for 8-ALU 2-bus configurations over single cluster.

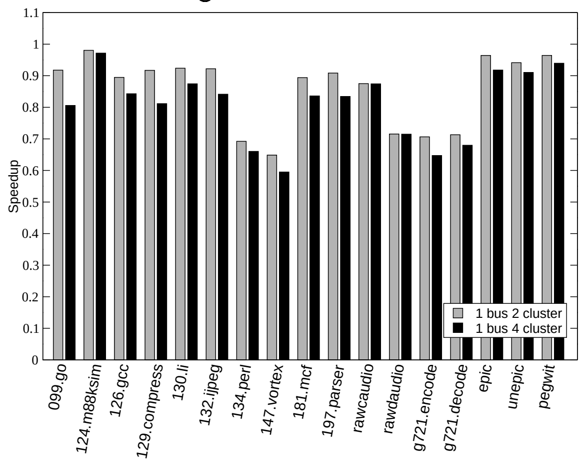

Figure 11: Speedup for 16-ALU 1-bus configurations over single cluster.

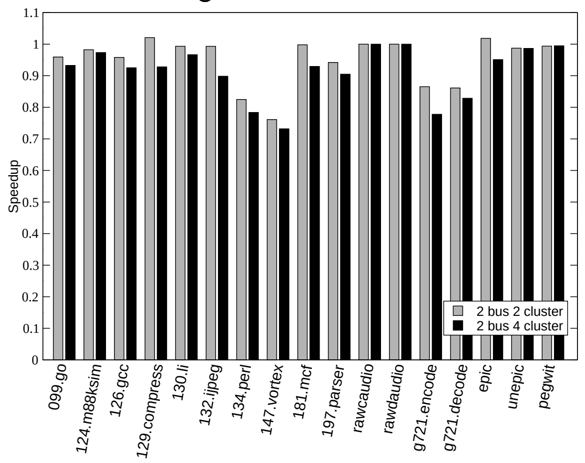

Figure 12: Speedup for 16-ALU 2-bus configurations over single cluster.

for 2-cluster and 4-cluster machines. Figures 9, 11, 10 and 12 show the speedup (ratio of number of VLIWs executed) with respect to the corresponding base configurations of clustered machines with single and two inter-cluster communication buses. Speedup higher than one was observed on clustered machines for some benchmarks. This is primarily due to the aggressive peephole optimizations done after CARScheduling and also due to the non-linearity of the cluster-scheduling process. Similar observations were reported for the code generated using PCC algorithm [43]. On average the additional number of cycles due to clustering compared to single cluster machines is less than 9.6% and 9.2% for 2-cluster machines with 16 and 8 ALUs, whereas the corresponding figures for 4-cluster machines are 13.9% and 13.9%. This clearly shows the efficiency and scalability of the CARS-based code generation scheme. The CARS algorithm tries to distribute computation across clusters, fully utilizing the available inter-cluster communication bandwidth. This is evident from the observed 10.1% increase in average performance while going from single bus to 2 bus configurations.

Figure 13 shows the distribution of the number of inter-cluster copy OPs and other executed

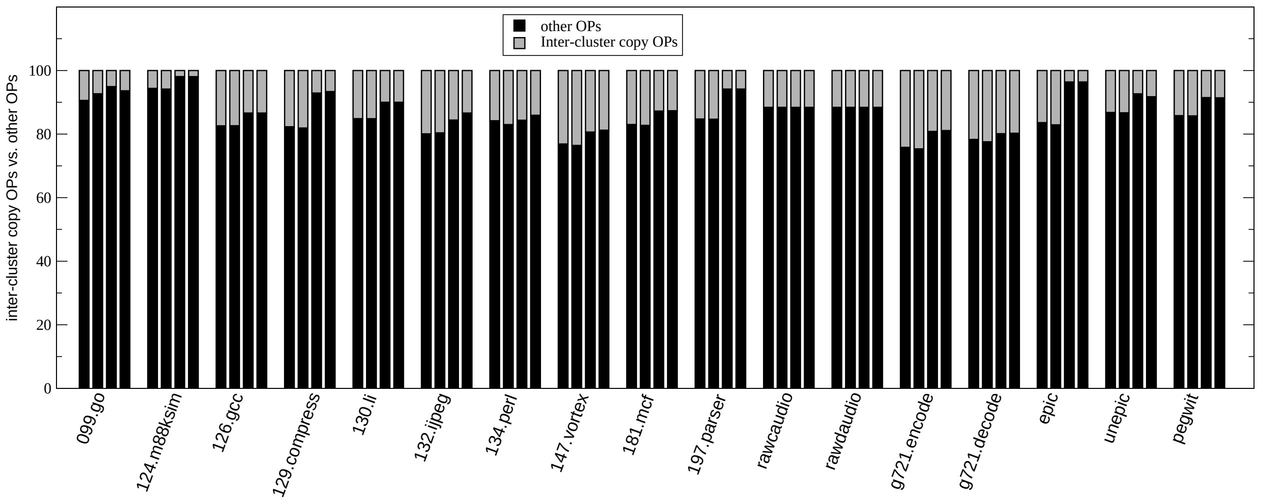

Figure 13: Percentage distribution of executed inter-cluster copy OPs vs. other OPs for clustered 8-ALU machines. The set of four bars (left to right) for each benchmark program correspond to the machine configurations 1 bus 4 clusters, 2 buses 4 clusters, 2 buses 2 clusters and 1 bus 2 clusters respectively.

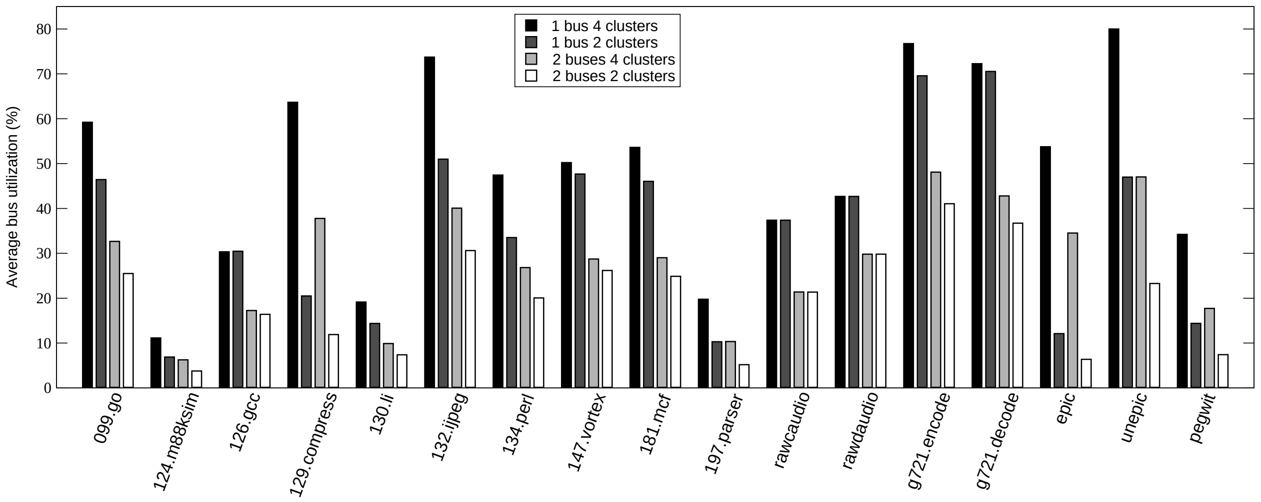

Figure 14: Average bus utilization for clustered 8-ALU machines.

OPs for clustered 8-ALU machine configurations. We can see that the average number of executed inter-cluster copy OPs varies from 11% to 16% of the total number of executed OPs. Figure 14 shows the average bus utilization for different configurations of clustered 8-ALU machines studied. The observed high utilization of inter-cluster communication bus for the configurations that have either less number of buses or more number of clusters is a clear indication of the adaptive nature of CARS-based code generation framework and its ability to distribute OPs across clusters in a resource-constrained manner.

We believe that CARS can generate more efficient code than other schemes because of its capability to use heuristics based on local register pressure feedback for better cluster assignment and scheduling, as well as its ability to reduce spilling caused by phase-ordering problem.

6 Conclusion

We have presented CARS, a new code generation framework for clustered ILP processors. Our work is motivated by the phase-ordering problems of code generators for clustered ILP processors. Our scheme completely avoids the phase-ordering problem by integrating the cluster assignment, instruction scheduling and register allocation into a single phase, which in turn helps eliminate unnecessary spills and other inefficiencies due to multiple phases in code generation. The fully resource-aware cluster scheduling scheme of CARS not only helps avoid unnecessary stretching of critical paths in the code but also distribute computation evenly across the clusters whenever possible. We also described an efficient on-the-fly register allocation technique developed for CARS. Even though the register allocation scheme is described in the context of code generation for clustered ILP processors, the technique is well suited for other applications such as "just-in-time compilation" and dynamic binaray translation for efficiently generating high performance code. Our experimental results show that CARS-based code generation scheme is scalable across a wide range of clustered ILP processor configurations and generates efficient code for a variety of benchmark programs.

Incorporating software pipelining into CARS and making an iterative version of CARS for generating highly optimized code for small DSP kernels are some of the directions we plan to explore in the future.

Acknowledgments

We would like to thank Mayan Moudgill and Erik Altman for their helpful suggestions on the implementation of CARS and modification of the CHAMELEON's VLIW translator for clustered machine model, and Vance Waddle for providing us the NARC graph visualization software. We are grateful to Manoj Franklin for helpful discussions and for his comments. The first author would like to thank IBM Research for an year and a half long technical co-op position at IBM Thomas J. Watson Research Center during the initial phases of the work presented in this report.

References

[1] P. Faraboschi, G. Desoli, and J. Fisher, "Clustered instruction-level parallel processors," Technical Report HPL-98-204, HP Labs, Cambridge, Dec. 1998.

[2] P. Faraboschi, J. Fisher, G. Brown, G. Desoli, and F. Homewood, "Lx: A Technology Platform for Customizable VLIW Embedded Processing," in *Proceedings of the 27th International Symposium on Computer Architecture*, June 2000.

[3] K. Ebcioğlu, J. Fritts, S. Kosonocky, M. Gschwind, E. Altman, K. Kailas, and T. Bright, "An Eight-Issue Tree-VLIW Processor for Dynamic Binary Translation," in *Proceedings of International Conference on Computer Design (ICCD'98)*, IEEE Press, Oct. 1998.

[4] Texas Instruments, Inc., *TMS320C62x/C67x Technical Brief*, Apr. 1998. Digital Signal Processing Solutions, Literature No. SPRU197B.

[5] J. Fridman and Z. Greenfield, "The TigerSHARC DSP architecture," *IEEE Micro*, vol. 20, pp. 66–76, Jan./Feb. 2000.

[6] R. E. Kessler, "The Alpha 21264 microprocessor," *IEEE Micro*, vol. 19, pp. 24–36, Mar./Apr. 1999.

[7] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-effective superscalar processors," in *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, vol. 25,2 of *Computer Architecture News*, (New York), pp. 206–218, ACM Press, June2–4 1997.

[8] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic, "The multicluster architecture: Reducing cycle time through partitioning," in *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-97)*, (Los Alamitos), pp. 149–159, IEEE Computer Society, Dec.1–3 1997.

[9] R. Canal, J. M. Parcerisa, and A. Gonzalez, "Dynamic cluster assignment mechanisms," in *Proceedings of the 6th International Conference on High-Performance Computer Architecture (HPCA-6)*, Jan. 2000.

[10] E. Nystrom and A. E. Eichenberger, "Effective cluster assignment for modulo scheduling," in *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-98)*, (Los Alamitos), pp. 103–114, IEEE Computer Society, Nov. 30–Dec. 2 1998.

[11] S. M. Freudenberger and J. C. Ruttenberg, "Phase ordering of register allocation and instruction scheduling," in *Code Generation – Concepts, Tools, Techniques. Proceedings of the International Workshop on Code Generation, May 1991* (R. Giegerich and S. L. Graham, eds.), (London), pp. 146–172, Springer-Verlag, 1992.

[12] R. Motwani, K. V. Palem, V. Sarkar, and S. Reyen, "Combining register allocation and instruction scheduling," Technical Note CS-TN-95-22, Stanford University, Dept of Computer Science, Aug. 1995.

[13] K. K. Kailas, *Microarchitecture and Compilation Support for Clustered ILP Processors*. PhD thesis, Dept. of Electrical & Computer Engineering, University of Maryland, College Park, 2000. in preperation.

[14] K. Pingali, M. Beck, R. Johnson, M. Moudgill, and P. Stodghill, "Dependence flow graphs: an algebraic approach to program dependencies," in *POPL '91. Proceedings of the eighteenth annual ACM symposium on Principles of programming languages*, pp. 67–78, 1991.

[15] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, pp. 451–490, Oct. 1991.

[16] W. mei 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*, vol. 7, pp. 229–248, May 1993.

[17] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pp. 45–54, Dec. 1992.

[18] W. A. Havanki, S. Banerjia, and T. M. Conte, "Treegion scheduling for wide-issue processors," in *Proceedings of the 4th International Symposium on High-Performance Computer Architecture (HPCA-4) (Las Vegas)*, Feb. 1998.

[19] K. Ebcioğlu, E. R. Altman, S. Sathaye, and M. Gishwind, *Execution-based scheduling for VLIW Architectures*, pp. 1269–1280. Lecture Notes in Computer Science, Springer-Verlag, 1999. Proc. Europar '99.

[20] K. Ebcioğlu, "Some design ideas for a VLIW architecture for sequential-natured software," in *Parallel Processing (Proc. IFIP WG 10.3 Working Conference on Parallel Processing, Pisa, Italy)* (M. Cosnard and et al., eds.), pp. 3–21, North Holland, 1988.

[21] M. Poletto and V. Sarkar, "Linear scan register allocation," *ACM Transactions on Programming Languages and Systems*, vol. 21, pp. 895–913, Sept. 1999.

[22] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register allocation via coloring," *Computer Languages*, vol. 6, no. 1, pp. 47–57, 1981.

[23] S. S. Muchnick, *Advanced compiler design and implementation*, ch. 16. Morgan Kaufmann Publishers, San Francisco, CA., 1997.

[24] E. G. Coffman, ed., *Computer and job-shop scheduling theory.* Wiley, New York., 1976.

[25] J. H. Moreno, M. Moudgill, K. Ebcioğlu, E. Altman, C. B. Hall, R. Miranda, S.-K. Chen, and A. Polyak, "Simulation/evaluation environment for a VLIW processor architecture," *IBM Journal of Research & Development: Performance analysis and its impact on design (PAID)*, vol. 41, no. 3, pp. 287–302, 1997.

[26] M. Moudgill, "Implementing an Experimental VLIW Compiler," *IEEE Technical Committee on Computer Architecture Newsletter*, pp. 39–40, June 1997. Also see the web-page: http://www.research.ibm.com/vliw/compiler.html.

[27] J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures.* ACM Doctoral Dissertation Award; 1985, MIT Press, Cambridge, Mass., 1986.

[28] E. Ozer, S. Banerjia, and T. Conte, "Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures," in *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-31)*, 1998. A detailed version is available as an ECE Department Technical Report NC 27695-7911, North Carolina State University, March 2000.

[29] S. Banerjia, *Instruction Scheduling and Fetch Mechanisms for Clustered VLIW Processors.* PhD thesis, North Carolina State University, Raleigh, NC, 1998.

[30] J. R. Goodman and W.-C. Hsu, "Code scheduling and register allocation in large basic blocks," in *Conference Proceedings 1988 International Conference on Supercomputing*, pp. 442–452, July 1988.

[31] D. G. Bradlee, S. J. Eggers, and R. R. Henry, "Integrating register allocation and instruction scheduling for RISCs," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pp. 122–131, 1991.

[32] S. S. Pinter, "Register allocation with instruction scheduling: A new approach," *SIGPLAN Notices*, vol. 28, pp. 248–257, June 1993. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation.*

[33] D. A. Berson, R. Gupta, and M. L. Soffa, "Resource spackling: A framework for integrating register allocation in local and global schedulers," in *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '94*, pp. 135–145, Aug. 24–26, 1994.

[34] T. S. Brasier, P. H. Sweany, S. J. Beaty, and S. Carr, "CRAIG: A practical framework for combining instruction scheduling and register assignment," in *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '95*, pp. 11–18, June 1995.

[35] J. Hoogerbrugge and L. Augusteijn, "Instruction scheduling for TriMedia," *The Journal of Instruction-Level Parallelism*, vol. 1, Feb. 1999. (http://www.jilp.org/vol1).

[36] S. Hanono and S. Devadas, "Instruction selection, resource allocation, and scheduling in the Aviv retargetable code generator," in *Proceedings of 35th Design Automation Conference*, pp. 510–515, 1998.

[37] S. Novack, A. Nicolau, and N. Dutt, *A Unified Code Generation Approach Using Mutation Scheduling*, ch. 12. Kluwer Academic Publishers, 1995.

[38] S.-M. Moon and K. Ebcioğlu, "An efficient resource-constrained global scheduling technique for superscalar and VLIW processors," in *25th Annual International Symposium on Microarchitecture (MICRO-25)*, pp. 55–71, 1992.

[39] S. Kim, S.-M. Moon, and K. Ebcioğlu, "vLaTTe: A Java Just-in-Time Compiler for VLIW with Fast Scheduling and Register Allocation," July 2000. submitted for publication.

[40] R. A. Freiburghouse, "Register allocation via usage counts," *Communications of the ACM*, vol. 17, pp. 638–642, Nov. 1974.

[41] B.-S. Yang, S.-M. Moon, S. Park, J. Lee, S. Lee, J. Park, Y. C. Chung, S. Kim, K. Ebcioğlu, and E. Altman, "LaTTe: A Java VM just-in-time compiler with fast and efficient register allocation," in *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT '99)*, pp. 128–138, Oct. 1999.

[42] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. C. Ruttenberg, "The Multiflow Trace Scheduling compiler," *The Journal of Supercomputing*, vol. 7, pp. 51–142, May 1993.

[43] G. Desoli, "Instruction assignment for clustered VLIW DSP compilers: A new approach," HP Labs Technical Report HPL-98-13, HP Labs, Jan. 1998.

[44] A. Capitanio, N. Dutt, and A. Nicolau, "Partitioned register files for VLIWs: A preliminary analysis of tradeoffs," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, (Portland, Oregon), pp. 292–300, IEEE Computer Society TC-MICRO and ACM SIGMICRO, Dec. 1–4, 1992.

[45] I. Ahmad, K. Yu-Kwong, and W. Min-You, "Analysis, evaluation, and comparison of algorithms for scheduling task graphs on parallel processors," in *Proceedings of the Second International Symposium on Parallel Architectures, Algorithms, and Networks*, pp. 207–213, June 1996.

[46] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*, ch. 4. Research monographs in parallel and distributed processing, MIT Press, Cambridge, MA., 1989.

[47] T. Yang and A. Gerasoulis, "DSC: Scheduling parallel tasks on an unbounded number of processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, pp. 951–967, Sept. 1994.

[48] J. Liou and M. Palis, "A new heuristic for scheduling parallel programs on multiprocessor," in *Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, Oct.13–17 1998.

[49] 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 *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, (San Jose, CA), Oct. 4–7, 1998.

[50] M. M. Fernandes, J. Llosa, and N. Topham, "Distributed modulo scheduling," in *Proceedings of the 5th International Conference on High-Performance Computer Architecture (HPCA-5 '99)*, Jan. 1999.

[51] "SPEC CPU95 Benchmarks." on web http://www.spec.org/osg/cpu95/, June 2000. Standard Performance Evaluation Corporation, USA.

[52] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communications systems," in *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pp. 330–335, Dec. 1–3, 1997.

[53] J. L. Henning, "SPEC CPU2000: Measuring CPU Performance in the New Millennium," *Computer*, vol. 33, pp. 28–35, July 2000.