CAeSaR: unified Cluster-Assignment Scheduling and communication Reuse for clustered VLIW processors *

Vasileios Porpodas and Marcelo Cintra †

School of Informatics, University of Edinburgh

{v.porpodas@, mc@staffmail.}ed.ac.uk

Abstract

Clustered architectures have been proposed as a solution to the scalability problem of wide ILP processors. VLIW architectures, being wide-issue by design, benefit significantly from clustering. Such architectures, being both statically scheduled and clustered, require specialized code generation techniques, as they require explicit Inter-Cluster Copy instructions (ICCs) be scheduled in the code stream. In this work we propose CAeSaR, a novel instruction scheduling algorithm that improves code generation for such architectures. It combines cluster assignment, instruction scheduling and inter-cluster communication reuse all in one single unified algorithm. The proposed algorithm improves performance by any phase-ordering issues among these three code generation and optimization steps. We evaluate CAeSaR on the MediabenchII and SPEC CINT2000 benchmarks and compare it against the state-of-the-art instruction scheduling algorithm. Our results show an improvement in execution time of up to 20.3%, and 13.8% on average, over the current state-of-the-art across the benchmarks.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors; C.1.1 [*Processor Architectures*]: Single Data Stream Architectures

General Terms Algorithms, Experimentation, Performance

Keywords Cluster Assignment, Instruction Scheduling, Clustered VLIW

1. Introduction

Very Long Instruction Word (VLIW) processors are by definition wide-issue high-performance processors that execute instructions in a parallel fashion, as dictated by the compiler through the long instruction words. They are statically scheduled processors, where Instruction Level Parallelism (ILP) is determined by the instruction scheduler in the compiler. They have been used in a wide range of domains: in servers (Intel's Itanium/Itanium2 [23, 33]), in embedded systems as DSPs (Texas Instruments's VelociTI, HP/ST's Lx [10], Analog's TigerSHARC [12], BOPS' ManArray [30]), in GPUs (AMD's VLIW-5 architecture on Radeon GPUs and in APUs [3]) and as general purpose processors (e.g., Transmeta's Crusoe [7, 19]). The simple hardware design and the good power/performance ratio makes VLIW processors an attractive alternative to dynamically-scheduled processors. Compared to the latter, VLIW designs are simpler since they do without the dynamic scheduling hardware. Instead they rely on the compiler and more specifically on an aggressive instruction scheduler to extract ILP.

Clustering is a way of improving the scalability of processor designs by keeping the shared resources as small and local as possible. Architects have applied clustering to both statically (e.g., [10, 32, 34]) and dynamically scheduled (e.g., [18, 29]) processors to either achieve higher performance (by increasing the clock speeds) and/or improve the power characteristics of the design. More recently, clustering has been the major design decision in experimental, highly scalable ILP architectures such as the EDGE (TRIPS) architecture [4, 14] and WiDGET [36].

VLIW processors are good candidates for clustering, as they are wide-issue by design. Clustered VLIW processors (as in Figure 1.a) can reach higher issue-widths, higher clock speeds and lower energy consumption than their non-clustered counterparts [35]. In a clustered design, the global resources are partitioned into multiple private ones (Figure 1.a). The Register File (RF) is partitioned and the Functional Units (FUs) are grouped, such that each cluster contains a few FUs and a local slice of the RF. The communication between clusters takes place through scalable point-to-point communication buses. Clustered architectures can scale up (meaning that we can replicate the clusters) to large numbers, without affecting the clock frequency. The clustered VLIW processor, is statically scheduled, so it is up to the compiler's code generator to optimize the code for it.

1.1 Clustered VLIW Architectures

Clustered VLIWs rely on the compiler to orchestrate the communication between clusters, using explicit Inter-Cluster Copy instructions (ICCs). In such machines, it is up to the code generator to optimize the schedule and the communication. Examples of such architectures are the RAW processor [34] (with explicit send/receive instructions instead of our bi-directional ICCs) and the HP/ST architecture [10].

Internally these machines are designed in such a way that the instructions of each cluster can only access the local register file. Whenever some data is needed from a distant register file, an ICC instruction has to be issued to bring the data in. This is a good design decision for two reasons:

i) Converting an architecture into a clustered one requires only a small ISA change for adding the ICC instruction.

* This work was supported in part by the EC under grant ERA 249059 (FP7).

† Marcelo Cintra is currently on sabbatical leave at Intel Labs.

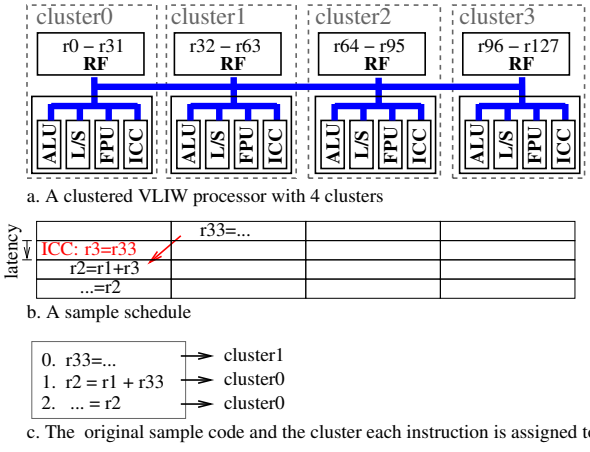


Figure 1. A 4-cluster 4-issue clustered VLIW architecture (a). The instruction schedule in (b) corresponds to the code in (c).

ii) Scaling up the clustered design requires no major re-design of the ISA, apart from the ICCs that need to access a larger register space.

A design with no ICCs would require that instructions have access to all remote registers. This would have the drawback that converting a non-clustered processor to a clustered one would require a significant ISA change affecting all instructions with a register addressing mode, since more register address bits would be required per instruction for accessing the remote register files. Moreover, scaling up to more clusters would require a modification of similar magnitude. Therefore the clustered design with ICCs is preferred.

An example of a clustered machine is shown in Figure 1. It is composed of 4 clusters, each with a register file of 32 GP registers (the Floating Point (FP) and Predicate (PR) register files are not shown) and one issue slot capable of executing Arithmetic (ALU), Load/Store (L/S), Floating Point (FPU) and Inter-Cluster Copy (ICC) instructions.

The code sequence of Figure 1.c will run on the clustered machine as in Figure 1.b. The ADD instruction *r2=r1+r33* is assigned to cluster0, and therefore it can not access register r33 that belongs to cluster1. It therefore has to be modified to *r2=r1+r3*, where r3 is local to cluster0. The data is transferred from cluster1 to cluster0 by the ICC *r3=r33*, which is the only instruction capable of accessing registers belonging to different clusters. Any inter-cluster communication is associated with an inter-cluster latency, that of the latency of the ICC instruction.

It might seem that clustered architectures have an additional overhead compared to their non-clustered counterparts: that of the inter-cluster delay. In reality there is an advantage. The clustered design, with the explicit inter-cluster delays, lets the clustered architecture operate at higher frequencies within a cluster compared to monolithic non-clustered designs [35].

1.2 Code Generation

Code generation for clustered architectures differs from the traditional one for non-clustered machines. It requires an additional cluster assignment pass that decides on the cluster that each instruction will be executed at. The difference is shown in Figure 2.a and 2.b. Cluster assignment tags each instruction with a cluster number tag. The cluster assignment algorithm decides on the cluster with the inter-cluster communication latencies and the per-cluster hardware resources in mind. After each cluster is tagged with a cluster number, the instruction sequence gets scheduled by the instruction scheduler just as in traditional code-generation.

ICC instructions are required for correct execution. They are inserted by the instruction scheduling pass and are placed before each instruction that belongs to one cluster but reads a register from a different cluster. The ICC transfers the remote register value to a local register and then the instruction using the value is modified to use the local register instead of the remote one.

The challenge for the code generator is to optimally balance communication and computation since ICC instructions compete with other regular instructions for the same resources (issue slots). It is this harder resource allocation problem, not present in the non-clustered VLIWs, that existing code-generation schemes are not designed to handle effectively.

Optimized code generation for clustered architectures requires that ICC instructions be optimized away. This can be done by re-using the data brought in by past ICCs instead of bringing them again multiple times. We refer to this optimization as Communication Reuse or **ICC-reuse**. This is a critical optimization for a clustered architecture where the inter-cluster communication is a critical resource. None of the existing approaches reuse ICCs.

1.3 Contributions

In our work we show that clustered architectures require an improved instruction scheduling algorithm that unifies all clustering, scheduling and ICC-reuse. The reason why all these phases should be unified is that otherwise a phase-ordering problem exists that leads to sub-optimal solutions:

- If clustering is done separately from instruction scheduling then many ICCs may be generated at scheduling time that will harm performance. This has been shown in [28]. Therefore, a unified clustering + scheduling pass is required (Figure 2.c).
- If the unified scheduling+clustering is done separately from the communication reuse (ICC-reuse) (Figure 2.d) then the clustering+scheduling decisions will be based on the assumption that all ICCs exist in the schedule, which will not hold after the ICCs get reused. This can lead to bad clustering/scheduling decisions. A unified clustering + scheduling + ICC-reuse algorithm, however, will provide the best solution (Figure 2.e).

In this paper we introduce CAeSaR, a unified instruction scheduling algorithm, that improves code-generation for clustered architectures where inter-cluster communication is a critical resource. In more detail our contributions are:

- Identification and quantification of ICC overhead.
- Introduction of the first unified instruction scheduler for clustered VLIW processors that performs all clustering, scheduling and communication minimization in a single algorithm.
- A detailed comparison against the state-of-the art across a wide range of benchmarks and showing that the approach performs better.

In the following sections, we first motivate our work through a motivating example (Section 2), we then describe in detail the methodology and the CAeSaR algorithm (Section 3). Then we briefly describe the experimental setup (Section 4), we present the experimental results (Section 5), and we discuss the related work on the topic (Section 6). We finally conclude in Section 7.

2. Motivation

Existing code generation schemes do not optimize the inter-cluster communication. In the motivating example that follows, we show the shortcomings of the existing state-of-the-art and how we improve it with the CAeSaR algorithm.

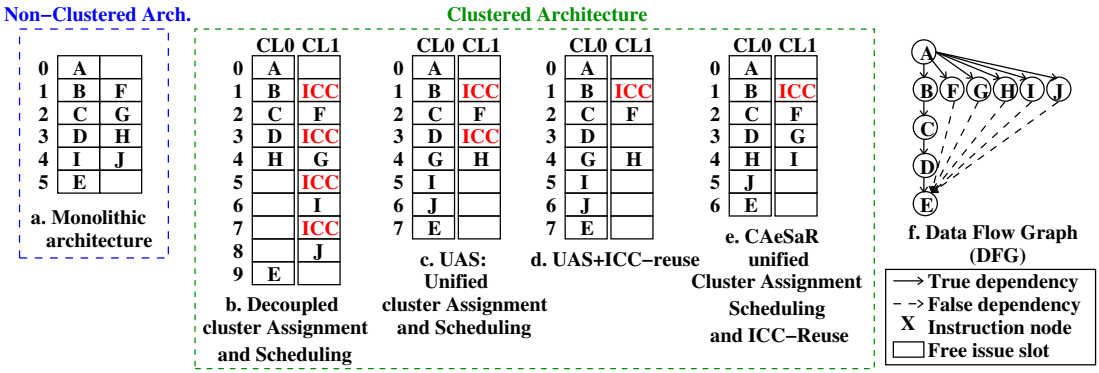


Figure 3. Instruction schedules for the Data Flow Graph (DFG) in (f), based on various scheduling algorithms. The first one (a) is on a monolithic non-clustered VLIW architecture. The rest are on a clustered architecture: (b) Decoupled Cluster Assignment and Scheduling, (c) Unified Assignment and Scheduling (UAS), (d) UAS + ICC-reuse optimization, (e) CAeSaR (proposed).

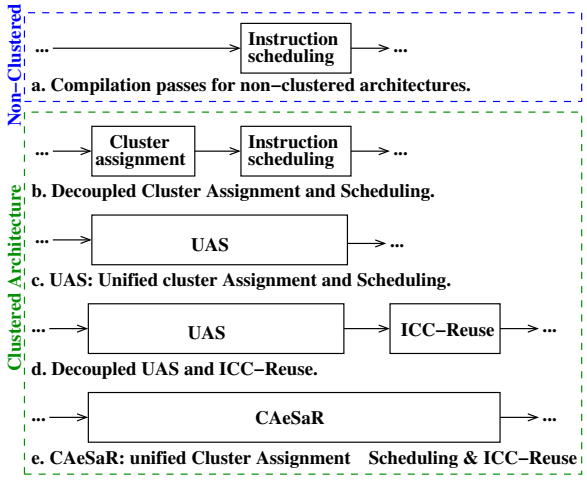


Figure 2. The compilation pipeline for schedules of Figure 3.

The example is based on the Data Flow Graph (DFG) of Figure 3.f mapped on both a monolithic non-clustered (Figure 3.a) and clustered (Figures 3.b-e) architecture. The example shows the schedules for both architectures. The non-clustered one is shown as a reference. The example focuses mainly on the schedules for the clustered architecture generated by i) a naive decoupled scheduler (Figure 3.b), ii) the state-of-the-art (Figure 3.c) UAS [28] and iii) the proposed CAeSaR scheduler (Figure 3.e). Figure 3.d is an intermediate step between the state-of-the-art and CAeSaR which helps us get more insights on the workings of CAeSaR. The architecture of the example is a dual-issue dual-clustered (that is single-issue per cluster) architecture with a single cycle inter-cluster delay, meaning that the earliest a dependent instruction can execute on the remote cluster is $current\_cycle + 2$. The Data Flow Graph of Figure 3.f contains both True and False dependencies. The False ones do not imply any data communication to their successors, they just denote an ordering. To help visualize the compilation process for each of these schedules, Figure 2 shows the compilation passes involved in each case.

In what follows we introduce each optimization individually and we discuss its impact on the instruction schedule (Figure 3).

i. The first schedule in Figure 3.a is the schedule obtained on a non-clustered VLIW architecture by applying instruction scheduling (Figure 2.a). This schedule is not split in clusters nor does it contain any inter-cluster communication instructions (ICCs). Cycle-wise it is the shortest (fastest) since there are no inter-cluster overheads involved.

ii. From this point on we are concerned only with scheduling for the clustered architectures. The same compilation technique as in (i.), if applied on a clustered architecture leads to the schedule of Figure 3.b. We refer to this as the naive "Decoupled" scheme. The instructions are placed on the cluster that the clustering algorithm dictates. That is: A, B, C, D, H, E on CL0 and F, G, I, J on CL1. The scheduling pass inserts the ICCs, which occupy many issue slots. Since the scheduler cannot change the clustering decision, the final schedule is full of unused slots. The need to insert ICC instructions during scheduling creates a phase-ordering issue between cluster assignment and instruction scheduling.

iii. Unifying the cluster assignment and the instruction scheduling (UAS [28], Figures 2.c and 3.c) solves this phase-ordering problem. The clustering decision is now made while the code and the ICC instructions get scheduled. UAS decides on the cluster that an instruction will be scheduled at by taking into account the issue slot occupancy of the ICCs in each case. The decision that UAS makes is a much more informed one than of the previous decoupled approach. The resulting schedule is shown in Figure 3.c. This is the current state-of-the-art.

iv. What is still missing from UAS is the reuse of data already communicated to a cluster. This is possible in clustered VLIW architectures because each cluster contains a local Register File. Figure 3.c shows that two ICC instructions are in place, even though both instructions F and H read the same value from A. This is where the ICC-reuse pass takes action (Figures 2.d and 3.d). It removes the redundant ICCs while making sure that H gets its data from the already transmitted value. The resulting schedule has fewer ICCs, but its size is still the same as that of UAS (iii.). This step is an intermediate one.

v. CAeSaR (Figures 3.e and 2.e) integrates the ICC-reuse optimization into a unified clustering, scheduling and communication reuse algorithm. The unified approach makes more informed decisions on clustering and scheduling as it is aware that not only ICC instructions are required but also that some can be optimized away. This removes the phase ordering issue between UAS (that is unified clustering + scheduling with ICC-insertion) and the ICC-reuse pass. CAeSaR is therefore free of any phase ordering issues in all clustering, scheduling with ICC-insertion and ICC-reuse. As shown in Figure 3.e, CAeSaR makes an ICC-reuse-aware decision for instruction G, which gets scheduled on CL1 instead of CL0. This leads to more compact schedules than UAS, or UAS+ICC-reuse.

3. CAeSaR

3.1 High Level Overview

The CAeSaR scheduling algorithm unifies cluster assignment, instruction scheduling and communication reuse in a single unified instruction scheduling pass. The algorithm's structure is based on the commonly used list scheduler. In short the algorithm schedules all instructions in a single traversal of the DFG. It fills in the scheduling slots cycle-by-cycle. Once a cycle is scheduled it is never revisited. The code of the algorithm comprises two levels of nested loops. The outer one iterates until all instructions in the DFG are scheduled. The inner one iterates until the current scheduling cycle is either full or no other instructions are ready to be scheduled on it. The integration of cluster assignment and communication reuse is done within the innermost loop. CAeSaR can work with various clustering heuristics, but the implementation shown makes use of the Start-Cycle heuristic [9, 17] which, according to [31], is the best for clustered architectures with low inter-cluster communication delays (like the 1-cycle delay we consider). Other heuristics such as the Completion-Cycle [9] or the Critical-Successor [37] could also be used instead.

3.2 CAeSaR Main Body

Algorithm 1. CAeSaR scheduling algorithm.

```
1  /* In1: Data Flow Graph (DFG)
2     Out: Scheduled Code. */
3  caesar ()
4  {
5  /*While there are instructions unscheduled*/
6  while (instructions left to schedule)
7    update READY_LIST with ready+deferred instr
8    sort READY_LIST based on priorities
9    while (READY_LIST not empty)
10     INSN = the highest priority of READY_LIST
11     LIST_OF_CLUSTERS[] = possible clusters for INSN
           ↪on CYCLE
12     Sort LIST_OF_CLUSTERS[] using start_cycle()
13     while (unvisited clusters in LIST_OF_CLUSTERS[])
14       BEST_CLUSTER = first not visited
             ↪LIST_OF_CLUSTERS[]
15       /*Try scheduling INSN on best cluster*/
16       if (INSN can be scheduled on BEST_CLUSTER at
             ↪CYCLE)
17         ICC_LIST = compute_ICCs(INSN, BEST_CLUSTER)
18         if (ICC_LIST != NULL)
19           Try scheduling ICCs of ICC_LIST before
                 ↪CYCLE
20           if (failed)
21             Tag BEST_CLUSTER as visited
22             continue /* next cluster */
23           Schedule ICCs in ICC_LIST
24           Tag INSN to be renamed with ICC
                 ↪destination reg
25         if (INSN requires reg renaming)
26             INSN = register renamed INSN
27         Schedule INSN
28         Remove INSN from READY_LIST
29     /*If scheduling failed defer to CYCLE+1*/
30     if (INSN unscheduled)
31         remove INSN from READY_LIST and re-insert it
               ↪ at CYCLE+1
32   /*READY_LIST is empty*/
33   CYCLE ++
34 }
```

The main body of the CAeSaR algorithm is listed in Algorithm 1. CAeSaR is based on list-scheduling, and therefore it is composed of two nested loop levels: the outermost one that starts on Algorithm 1 line 6 and the innermost on line 9. CAeSaR has a third innermost nested loop (line 13) which iterates over all possible clusters to select the best one to schedule an instruction.

The outer loop (first) updates the ready list (line 7) with any new ready instructions from the DFG or any deferred instructions from a previous scheduling step. The ready list is then sorted based on priority (line 8), which is usually the longest latency-weighted path of the instruction node to the roots of the DFG.

The inner loop (second) (line 9) tries to fill up the current scheduling cycles with as many instructions as possible. It first gets the highest priority instruction from the sorted ready list (line 10), then it forms a prioritized list of all clusters that INSN (INStructioN) could be scheduled at (lines 11 and 12). The sorting of the list is done with the help of the clustering heuristic (start_cycle Algorithm 3, see Section 3.4).

After the list of clusters is sorted, we step into the innermost (third) loop (line 13). This loops over all clusters in the list and on each iteration selects the first unvisited cluster. This is the cluster with the highest priority according to the clustering heuristic (BEST_CLUSTER in line 14) among the clusters that are not tried out.

Once the BEST_CLUSTER is determined, the algorithm will try to schedule INSN on that cluster. However, since ICCs may be required (line 17, Section 3.3), scheduling on the BEST_CLUSTER may fail, so the innermost loop (line 13) keeps checking all cluster candidates until INSN gets scheduled (lines 19 to 22). If an ICC is emitted or if an ICC is reused, then INSN has to be register renamed to use the register written by the ICC. In either case, INSN gets tagged with the appropriate register number (Algorithm 1 line 24, Algorithm 2 lines 13 and 18 respectively). Renaming takes place right before INSN gets scheduled (lines 25 and 26).

If INSN cannot be scheduled on any cluster, then INSN is removed from the ready list and deferred until the next cycle (lines 29 to 31). The algorithm proceeds to the next cycle when all instructions of the ready list have either been scheduled, or have been deferred to a later cycle (lines 32 and 33).

3.3 Compute ICCs

The function that determines the list of ICCs required by the scheduled instruction (line 17 in Algorithm 1) is listed in Algorithm 2. If we ignore reusing the ICCs, then this is done in the following steps:

1. Check all flow predecessors of INSN (lines 6 and 7) and for each one of them determine the register REG_W used to pass the value from the predecessor to INSN.
2. If INSN is tried on a cluster different than the predecessor's cluster, then an ICC is required to transfer the data to the consumer's cluster (line 9).
3. Create a new ICC instruction to copy the data across register files: REG_NEW = REG_W (where REG_NEW is a register mapped to INSN's cluster) that transfers the value from one cluster to the other (lines 15 and 16).
4. Append the newly created ICC instruction to the list of ICCs required by INSN (line 17). This is the list that is returned by this function.
5. Tag INSN to be renamed with REG_NEW when renaming is done later on (line 18). This is required so that INSN will read the value from new register, the target of the ICC.
6. Return the list of ICCs (line 21).

This approach, however, introduces many redundant ICCs. Reusing the ICCs is described in Section 3.5.

3.4 Clustering Heuristic

Although CAeSaR can sort its LIST_OF_CLUSTERS (Algorithm 1 line 12) using any clustering heuristic (as it is decoupled from the actual heuristic used), in this implementation we use the Start-Cycle heuristic [9]. This is because this heuristic works best for clustered VLIW architectures with inter-cluster latency of 1 cycle [31]. The actual heuristic is orthogonal to our approach, since ICC

Algorithm 2. Get list of ICCs required for INSN scheduled on CLUSTER.

```
1  /*In1: Instruction INSN
2    Out: List of ICCs required, NULL if empty*/
3  compute_ICCs (INSN, CLUSTER)
4  {
5   ICC_LIST = NULL
6   for all DEP flow backward dependencies of INSN
7     PRO = producer of DEP
8     REG_W = register written by PRO and read by INSN
9     if (cluster of PRO != CLUSTER)
10      /* Read ICC Reuse Data Structure */
11      if (REG_W already present in CLUSTER)
12        REG_OLD = register that holds the value of
              ↪REG_W on CLUSTER
13        Tag INSN to be renamed with REG_OLD
14        continue
15      REG_NEW = new free register
16      ICC = ''REG_NEW = REG_W''
17      append ICC to ICC_LIST
18      Tag INSN to be renamed with REG_NEW
19      /* Update ICC Reuse Data Structure */
20      Record that REG_W exists in CLUSTER as REG_NEW
21   return ICC_LIST
22 }
```

Algorithm 3. Return the earliest cycle that an instruction can be scheduled at on CLUSTER.

```
1  /*In1: Instruction INSN
2    In2: The cluster CL that INSN is tried on
3    Out: Earliest cycle INSN can be scheduled on CL*/
4  start_cycle (INSN, CL)
5  {
6  start_c = 0
7  for all DEP backward dependencies of INSN
8    PRO = producer of DEP
9    PRO_CYCLE = cycle that PRO is scheduled at
10   PRO_CL = cluster of PRO
11   if (CL != PRO_CL)
12     if (DEP is flow dependence)
13       pro_start_c = PRO_CYCLE + communication delay
               ↪from PRO_CL to CL
14       start_c = MAX (start_c, pro_start_c)
15     else
16       start_c = MAX (start_c, PRO_CYCLE + 1)
17 return start_c
18 }
```

reuse is supported by our framework, no matter the decision of the clustering heuristic. Therefore, we can plug in any other clustering heuristic, such as the Completion-Cycle (CC) [9] or the Critical Successor (CS) [37].

The algorithm for the Start-Cycle heuristic is listed in Algorithm 3. It can be easily calculated by looping over all INSN's backward dependencies (Algorithm 3 line 7) and determining the earliest cycle that INSN can get its data inputs from its predecessors (lines 8 to 16).

3.5 ICC Reuse

Re-using the ICCs means that if an ICC instruction has transmitted a *valueA* to *clusterX* some time in the past, then any future use of *valueA* in *clusterX* should not require an additional ICC instruction. Instead the instruction that uses *valueA* is modified to reuse the value transmitted by the earlier ICC. This is a feature unique to CAeSaR that was neglected by previous scheduling algorithms because they targeted architectures where the ICC instructions did not compete with actual program instructions for issue slots.

Reusing the ICCs impacts performance in two distinct ways:

1. It reduces the count of the instructions that get scheduled (code size reduction).

2. It creates new opportunities for more ILP.

Both of these mechanisms contribute to the performance improvements. An example of this is shown in Figure 3. Saving up a single ICC instruction (that of cycle 3 in Figure 3.d), not only decreases the code size (1 less ICC) , but it also creates new opportunities for greater ILP: the empty slot created by re-using the ICC later gets occupied by instruction G. As will be shown later in Section 5.3, due to these phenomena, and particularly due to the second, a small decrease in the ICC count can have a much larger impact on performance.

Support for ICC-reuse requires some changes in the scheduling algorithm:

1. Keeping track of the ICCs that bring in data to each cluster. Map both registers of a new ICC (the source and the destination) to enable easy future reuse of the ICCs (Algorithm 2 line 20). This data gets stored in a dictionary structure which uses the source register as the key and the destination register as the content. We refer to it as *"ICC Reuse Data Structure"*. This is visualized for simplicity as a table of two columns (one for the source register and one for the destination) (Figure 4). For example if the ICC "Rx = Ry" is emitted, then the entry Rx→Ry is inserted into the Data Structure (see Figure 4).

2. Disabling the action of emitting a new ICC if data can be reused (Algorithm 1 line 18). This is done by querying the ICC Reuse Data Structure (Algorithm 2 line 11). If an entry exists for the register read by the instruction to be scheduled, then no ICC should be emitted.

3. Register renaming. Once an ICC is to be reused, then INSN has to be register renamed so that it reads the appropriate register. The register is determined in Algorithm 2 lines 12 and INSN is tagged with it in line 13. It later gets renamed as normally in Algorithm 1 line 26.

3.6 Register File Coherence

Keeping the distributed register files of a clustered processor coherent is required for correct execution. The problem is similar to the cache coherence problem in shared-memory multiprocessors. The baseline approach (UAS) issues an ICC copy whenever data from a distant cluster is required. This guarantees correctness as the value brought in is always the latest one. Problems can occur when reusing ICCs (like in CAeSaR). Reusing the data brought in by earlier ICCs could lead to using wrong data if the the original cluster has updated the register with a more recent value.

To further explain the problem, we follow the example of Figure 4. In this example a register (Rx) is updated twice in cluster0 (instructions A and D) and used twice in cluster1 (instructions B and C), with the 2nd update on cluster0 (instruction D) being in between the two uses in cluster1 (Figure 4.a instructions B and C). A non-coherent implementation is shown in Figure 4.b. The second use on cluster1 (instruction C) reuses the data brought in to cluster1 by the existing ICC1. This is incorrect, since Rx is updated before C by instruction D.

In CAeSaR, we solve this coherence problem in a similar way as in the write-invalidate cache coherence protocols, but at compile time. Once a register R is updated on a cluster, the entry for R on the ICC Reuse Data Structures of all other clusters are invalidated. This is shown in the example of Figure 4.c. Upon the second register update (instruction D: Rx=...) of cluster0, the ICC Reuse Data Structure of cluster1 invalidates the entry "Rx→Ry". As the algorithm encounters instruction C, it realizes that it cannot reuse Ry, and therefore it has to issue a new ICC2.

The complexity of this write-invalidate approach is small. Accessing the ICC Reuse Data Structure is done in constant time,

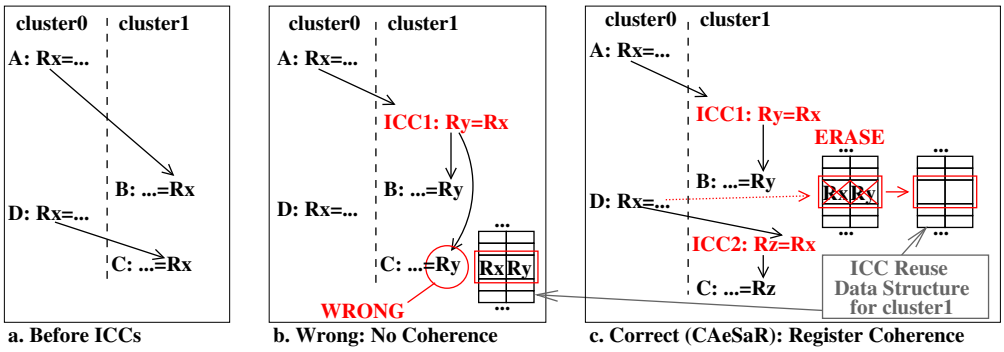


Figure 4. The Register File Coherence.

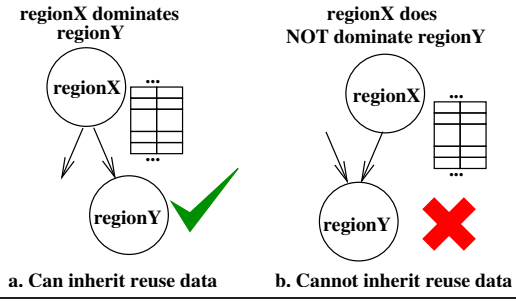


Figure 5. The ICC reuse challenges across scheduling regions.

Complexity		
Algorithm	Worst-Case	Observed
UAS (baseline)	$O(N^3)$	$O(N)$
CAeSaR	$O(N^3)$	$O(N)$

Table 1. Complexity of UAS (baseline) and CAeSaR algorithms.

since it is an indexed access to an array. Therefore, the whole process of invalidating all entries on an N-clustered machine has a complexity of N-1, a small single-digit integer. This process runs on every instruction that updates a register, and therefore the total overhead of the Register File Coherence is linear to the program size.

3.7 ICC Reuse Across Scheduling Regions

CAeSaR performs ICC-reuse at the scheduling-region level (EBBs) [26]. The data brought in to a cluster by an ICC, could be reused outside the region as well. This global-reuse approach has further complications, as shown in Figure 5. If the scheduler moves from regionX to regionY and regionX dominates regionY (Figure 5.a), then it is correct to inherit reuse information from regionX to regionY. Otherwise (Figure 5.b) this is not allowed as it will break the program semantics. CAeSaR currently completely flushes the ICC Reuse Data Structure upon a new scheduling region and therefore does not deal with this extra complexity.

3.8 Complexity Analysis

This section calculates and compares the complexity of CAeSaR and UAS.

To calculate the complexity of the **CAeSaR** algorithm we need to examine its source code (Algorithms 1, 2 and 3). For the computation we consider an input DFG of N nodes. The CAeSaR Scheduling algorithm has 3 levels of nested loops :

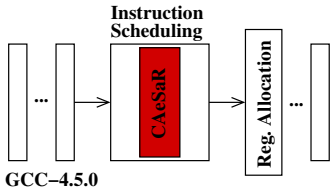


Figure 6. The compilation flow.

1. The outer loop iterates until all instructions in the DFG are scheduled. In each iteration a single cycle gets scheduled. If on average S (with $S \leq issuewidth$) instructions get scheduled, then this loop iterates N/S times. On each iteration of this loop, the ready list is sorted using quicksort. Given an average ready list size of R, this usually costs $R \times logR$ and R^2 in the worst case.

2. The middle loop iterates until all instructions in the ready list are examined for scheduling. Therefore it iterates R times. It sorts the list of clusters based on the Start-Cycle clustering heuristic. The Start-Cycle heuristic iterates over all flow predecessors of the instruction to be scheduled and gets calculated once for each cluster. If P is the number of flow predecessors and C is the number of clusters, then creating and sorting the list of clusters (using quicksort) costs $CP + ClogC$ in the usual case and $CP + C^2$ in the worst case.

3. The innermost loop iterates over all clusters in the order specified by the clustering heuristic. This loop always iterates C times (constant). On each loop iteration, compute_ICCs() is called (Algorithm 2), which iterates over all predecessors of the instruction to be scheduled. Therefore it iterates CP times.

The complexity of CAeSaR scheduling is computed as: $N/S \times R \times (logR + ClogC + 2CP)$ in the usual case and $N/S \times R \times (R + C^2 + 2CP)$ in the worst case. In all practical cases all S, R and P are small constants with typical values: $S \leq 3$, $R \leq 10$, $P \leq 10$. This is an $O(N)$ complexity. The worst-case scenario involves $S = 1$, $R = N$ and $P = N$ which leads to complexity $O(N^3)$.

UAS has a similar 3-nested loop structure and exhibits similar complexity. For all practical cases, the complexity of UAS is $O(N)$ and in the worst-case it is $O(N^3)$. Therefore both schedulers have practically the same complexity. The complexities are summarized in Table 1.

4. Experimental Setup

The target **architecture** is a clustered VLIW architecture based on the IA64 (Itanium)[23] ISA. The target architecture used for the evaluation, even though IA64-based, is a generic one, as it is not constrained by the IA64 bundles [33]. Our target architecture sup-

Target Architecture: IA64 based clustered VLIW	
Issue width:	4
Instr. Types per issue slot	ALU, L/S, FPU, ICCs
Clusters:	Configurable: 2, 4
Instruction Latencies:	Same as Itanium2 [23]
Inter-Cluster Latency:	1 cycle
Register File:	128GP, 64FP, 64PR in total

Table 2. Target Architecture Configuration.

Mediabench II	SPEC CINT2000
cjpeg	164.gzip
djpeg	175.vpr
h263enc	181.mcf
h263dec	186.crafty
mpeg2enc	197.parser
mpeg2dec	255.vortex
jpg2000enc	256.bzip2
jpg2000dec	300.twolf

Table 3. Benchmarks.

ports issuing any type of instruction (ALU/Load-Store/FPU/ICC) at any issue slot. The target configuration used for our measurements is shown in Table 2.

We implemented CAeSaR in the instruction scheduling pass (haifa-sched) of GCC-4.5.0 [1] **compiler** for IA64. CAeSaR runs just before register allocation, as shown in Figure 6. To evaluate CAeSaR's performance, we measure the total size (in cycles) of the schedules generated by the compiler under CAeSaR and compare it against two state-of-the-art clustering algorithms (UAS [28] and CS [37]).

We evaluated CAeSaR on 8 of the Mediabench II video [13] **benchmarks** and 8 of the SPEC CINT2000 [2] as listed in Table 3. Since our compiler is a heavily modified one, we only managed to fully compile the benchmarks shown. All benchmarks were compiled with several optimizations enabled (-O2).

5. Results and Analysis

5.1 Overview

We evaluate CAeSaR by measuring several metrics that give us some vital insights. We measure: i) the ICC instruction count overhead over the original program instructions (Figures 7.a and 8.a), ii) the count of ICC instructions issued by each scheduler (Figures 7.b and 8.b), iii) the total cycle count of all the scheduled regions (Figures 7.c and 8.c), and iv) the number of original (without ICCs) instructions per cluster (Figure 9), for the two machine configurations: (4-cluster, 4-issue) and (2-cluster, 4-issue).

We directly compare CAeSaR against the two state-of-the-art unified cluster assignment and scheduling algorithms: (UAS) [28], and Critical-Successor (CS) [37]. We also measure the intermediate scheme: decoupled UAS + ICC reuse (as shown in Figure 3.e). The measurements for UAS + ICC, though less interesting from the performance perspective, provide some vital insights on the workings of CAeSaR.

5.2 ICC Overhead

One of the most important results is the ICC instructions overhead in the baseline case (Figures 7.a and 8.a). It shows that ICC instructions are indeed a significant portion of the scheduled instructions. On average, ICCs add a 19.4% overhead on the instruction count for the 4-cluster machine and about 8.4% for the 2-cluster machine. This strongly motivates CAeSaR's goal to decrease the number of ICCs emitted during instruction scheduling.

Figures 7.b and 8.b show the normalized number of ICCs for both hardware configurations. Although the intermediate ICC-Reuse step does save about 12% and 10% of the ICCs on average for each configuration respectively, CAeSaR achieves savings of about 33% and 32%.

The number of ICCs that a scheduler emits relates to the performance of the generated code. Ignoring the ICC-reuse optimization, there are two interesting opposing phenomena that affect performance: i) The more the ICCs, the more aggressive the scheduler is and the more likely it is to generate high performance code. ii) The more the ICCs, the more the overhead due to ICCs consuming issue slots. Achieving good performance requires a solution that balances between these two phenomena. In that respect UAS is more conservative as it issues fewer ICCs compared to CS. However, the performance of both schedulers is very close (Section 5.3).

The ICC-reuse optimization allows the schedulers to be more aggressive at scheduling instructions across clusters since there are more ICC slots available for more useful computation. These slots enable either i) more ILP as more useful ICCs can be issued, or ii) more useful computations using the free issue slots for further progressing the program state. Therefore we expect that CAeSaR, which generates fewer ICCs, will generate more compact schedules.

5.3 Performance

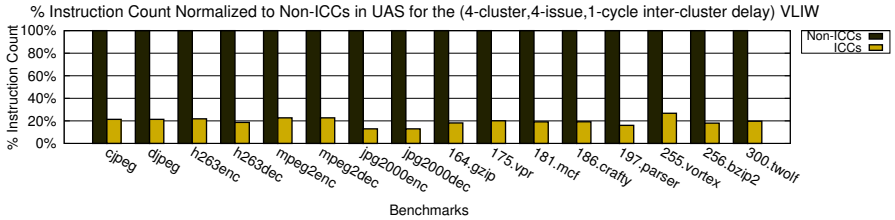
The total schedule length for CAeSaR, UAS and CS is shown in Figures 7.c and 8.c. These results show that CAeSaR generates more compact schedules than the state-of-the-art in all benchmarks. CAeSaR outperforms UAS by up to 20.3% and 13.8% on average for the 4-cluster machine. The results for the 2-cluster machine are equally impressive with an average of 8.4% improvement against UAS. CS performs similarly to UAS, which is expected as i) the heuristic defaults to UAS for several cases and ii) it does not reuse ICCs either.

The two machine configurations (2-cluster and 4-cluster) have the same issue width (4-issue) and the same inter-cluster delay (1-cycle). However, due to the fact that the 2-cluster machine can accommodate 8 execution units (2×ALU, 2×L/S, 2×FP, 2×ICC, twice as many as the 4-cluster machine) in each cluster, most general purpose applications fit nicely in a single cluster and therefore the distant cluster is under-utilized. Therefore the ICCs present in the schedule for that machine are fewer (Figure 8.a vs 7.a) and therefore the performance improvements CAeSaR can accomplish by ICC-reuse are smaller. It is up to the hardware designer to decide on the trade-off between issue per cluster and the operating frequency.

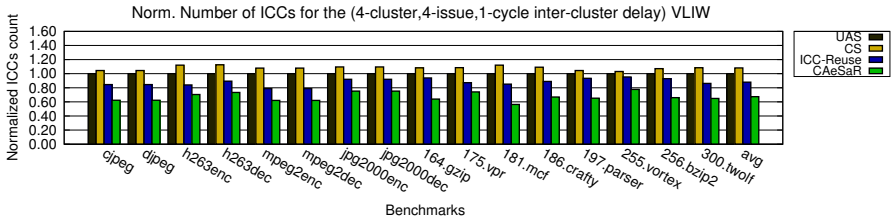
5.4 Phase-ordering

UAS is ICC-aware, meaning that the algorithm considers the communication as scheduled resources. But UAS is not ICC-reuse aware, meaning that it cannot calculate the communication reuse while scheduling. Therefore, when we combine the stages UAS + ICC-reuse, we end up with a sub-optimal solution: UAS will be conservative at distributing instructions across clusters because of the inter-cluster cost associated with each communication even though at the following stage ICC-reuse will remove some of the communication instructions, freeing up some slots. The end result is a sub-optimal schedule containing some empty slots (those that were reused, like in Figure 3.e CL1, cycle3), which could have been used for other useful instructions, or other useful communication.

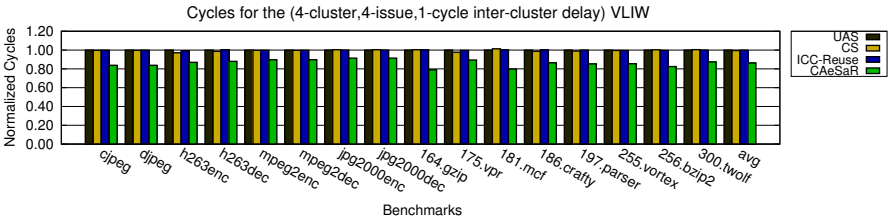
This is exactly the problem that CAeSaR solves by unifying scheduling, clustering and communication minimization into a single algorithm. In contrast to UAS, CAeSaR is more effective at distributing instructions across clusters (Figure 9) as long as this leads to better performance. CAeSaR can calculate the communication cost (including the communication reuse) more accurately than UAS. The ILP richer code, that CAeSaR generates, is faster



.a The ICC overhead. The percentage of ICCs compared to Non-ICC original program instr. for UAS.

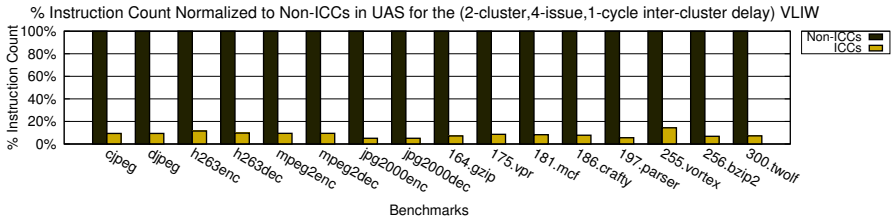


.b The count of ICC instructions per scheduler normalized to the UAS scheduler.

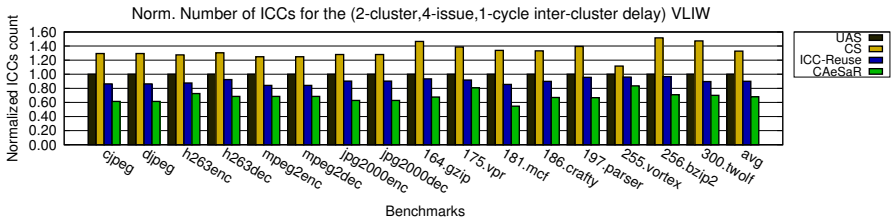


.c Total schedule cycles of each scheduler, normalized to UAS.

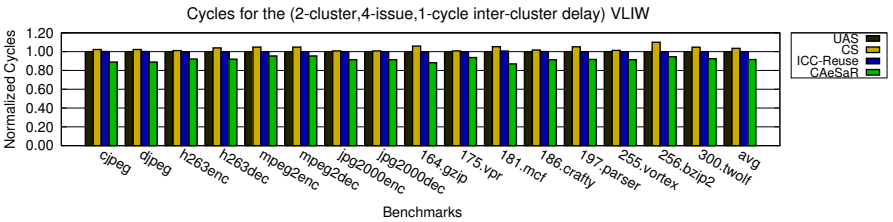
Figure 7. Measurements for the 4-cluster, 4-issue, 1-cycle inter-cluster delay VLIW machine.



.a The ICC overhead. The percentage of ICCs compared to Non-ICC original program instr. for UAS.



.b The count of ICC instructions per scheduler normalized to the UAS scheduler.



.c Total schedule cycles of each scheduler, normalized to UAS.

Figure 8. Measurements for the 2-cluster, 4-issue, 1-cycle inter-cluster delay VLIW machine.

and requires less frequent inter-cluster communication. Figure 9 shows that CAeSaR is consistently more effective at scheduling more instructions in the less used clusters and fewer in the more busy one. It achieves this by making good use of the slots saved by the unified ICC-reuse mechanism.

If we examine Figures 7.b and 8.b, we can observe that CAeSaR consistently reuses more ICCs compared to UAS+ICC-reuse (32.6% vs 12.1% and 32.0% vs 10.0% on average respectively). This result is a strong indication that the phase-ordering problem between clustering, scheduling and ICC-reuse is handled effectively by CAeSaR. Not only do ICCs get reused, but the clustering decision adapts as well so that even fewer ICCs are required.

6. Related Work

A comprehensive taxonomy of inter-cluster communication implementations on VLIW architectures is presented in [35]. The design features (such as operating frequency, performance, energy consumption, etc.) of each implementation are quantified and discussed.

Clustered Architectures: Pioneering work on code generation for clustered architectures appeared in [9], where the Bottom-Up-Greedy (BUG) cluster-assignment algorithm was introduced. This work differs from later cluster assignment algorithms in the order the instructions are considered for clustering, which in this case is a critical-path based ordering. The main heuristic used is the Completion-Cycle, which calculates the completion cycle of an instruction on each of the possible cluster candidates.

Much work on clustered machines has been done in the context of the Multiflow compiler [21]. It reused to a large extent Ellis' work on clustering [9]. The various design points (heuristic tuning, order of visiting the instructions, etc.) of instruction scheduling, including the cluster assignment, are discussed in detail in this work.

The work in [6] uses clustering to optimize the design of the register files. The goal is to partition the register file so as to have more register files with fewer ports each. Cluster assignment and ICC insertion takes place after scheduling the code since the input of this code generator is the output of a compiler that targets an ideal VLIW core. This, however, is sub-optimal since the ICCs should be re-scheduled so that the inter-cluster latencies can be hidden. The clustering heuristic used tries to minimize the inter-cluster communication. This however is a poor clustering heuristic as it is not guided by the schedule length. This work is the first to mention an optimization that minimizes the count of ICCs by reusing the copied data, however, no implementation details are given.

The work in [8] provides an iterative solution to clustering. Each iteration of the algorithm measures the schedule length by performing instruction scheduling and by doing a fast register pressure and ICC count estimation. Unlike in our approach, the ICCs are not optimized away. This being an iterative algorithm, it has a long run-time and its use is not practical in production compilers.

The first work that combines cluster assignment and instruction scheduling was [28]. Unlike BUG [9], this is a list-scheduling based, not critical-path based solution. The Completion Weighted Predecessor (CWP) clustering heuristic behaves very similarly to the Start-Cycle heuristic (first implemented in BUG [9]). The inter-cluster bandwidth is considered as a scheduling resource, but the Inter-Cluster Copy instructions (ICCs) are not optimized away.

CARS [17] is a combined scheduling, clustering, and register allocation code generation framework based on list scheduling. Depth and height heuristics are used to guide the algorithm. Scheduling is based on a variant of percolation scheduling [27] and as such it is capable of performing both acyclic and cyclic scheduling, similar to [24, 25].

	Unified				Performance
Algorithm	Clustering	Scheduling	Reg. Alloc.	ICC-Reuse	
BUG [9]	√	×	×	×	Low
UAS [28]	√	√	×	×	Med
CS [37]	√	√	×	×	Med
CARS[17]	√	√	√	×	Med
CAeSaR	√	√	×	√	High

Table 4. Summarized features of CAeSaR and others.

The RAW clustered architecture ([20, 34]) communicates data across clusters with send/receive instructions which are similar to ICCs. The scheduler visits instructions in a topological order and uses the completion time heuristic to guide the process. The authors, without identifying the challenges associated with ICCs, do mention that a multi-cast inter-cluster communication operation could be used as an optimization, without providing any further details. This, however, is a hardware-based approach, specific to the RAW architecture. CAeSaR provides a generic solution that works on standard clustered architectures.

Recently, a new clustering heuristic was introduced by [37]. This differs from the previously mentioned ones in that, under certain conditions, the clustering decision is based on earliest schedule cycle of the most critical successor of the current instruction. Similarly to the UAS heuristic, it does not try to minimize the ICCs in any way. This heuristic quite often defaults to the UAS, which is why its performance is similar to it. In our evaluation we refer to this heuristic as Critical-Successor (CS).

LUCAS [31] is a unified clustering and scheduling algorithm that aims to achieve good performance across a wide-range of inter-cluster latencies. It is powered by a hybrid Start-Cycle and Completion-Cycle heuristic. Its performance is compared against a variety of schemes, including UAS and CS and it is shown to outperform the best performing of them across a wide-range of ICC latencies. Unlike CAeSaR, ICCs are not re-used.

The main instruction scheduling algorithms proposed in the literature are summarized and compared to CAeSaR in Table 4.

Clustered superscalars, such as [18, 29], use simpler clustering algorithms. A review of the state-of-the-art dynamic heuristics is presented in [5]. Such heuristics make use of the Register Dependence Graph and steer instructions based on the cluster where their operands where steered to. Being dynamic approaches, they also try to balance the run-time load of the clusters.

Instruction Scheduling for VLIWs was pioneered by [11] with the Trace-scheduling algorithm. This algorithm expands the scheduling region beyond basic blocks to larger profile-guided regions called traces. These large regions provide enough instructions for the scheduler to re-order effectively. A less complicated but highly effective alternative to traces are the superblocks [16]. These regions simplify the scheduler's work by only allowing for outgoing control edges from within a region. VLIW architectures with support for predicated execution can benefit from hyperblock scheduling [22].

Several instruction schedulers form regions not based on profiling information. This is useful in two cases: i) when applications have unpredictable control flow and ii) when profiling is impractical. Such schemes [24, 25, 27] perform global code hoisting across multiple control paths concurrently and perform scheduling on the resulting code blocks. Extended Basic Blocks (EBBs) [26] form tree-like regions which are then scheduled by a normal list scheduler. Treegions [15] are also tree-shaped, and are similar to EBBs. CAeSaR is implemented on top of GCC's [1] Haifa Scheduler which operates on EBBs.

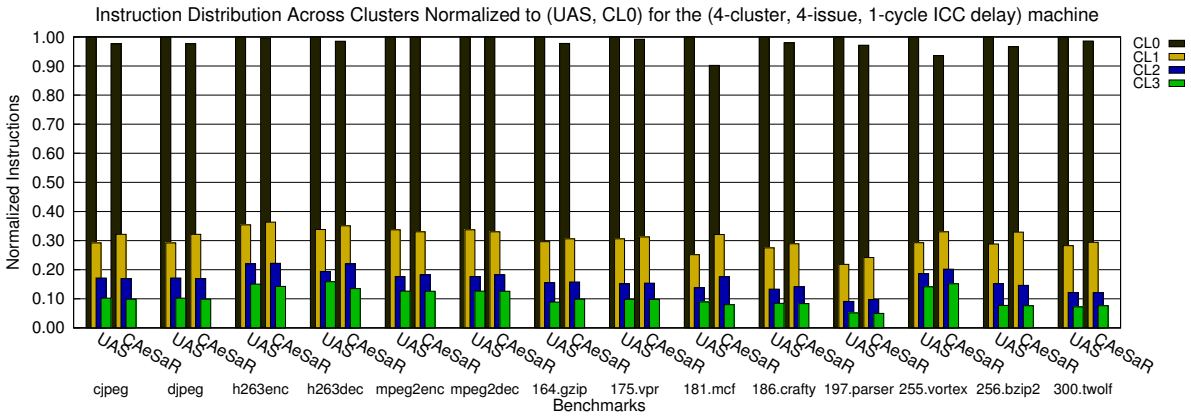


Figure 9. Distribution of original instructions across clusters.

7. Conclusion

This paper proposes CAeSaR, a new high-performance instruction scheduling algorithm for clustered VLIW architectures. The proposed algorithm is the first to solve all three problems: i) cluster assignment, ii) instruction scheduling and iii) inter-cluster communication reuse, within a single unified algorithm. CAeSaR not only minimizes the count of the Inter-Cluster Copy instructions, but also generates more compact code. Our evaluation shows that CAeSaR generates shorter schedules than the state-of-the-art across a range of benchmarks and machine configurations.

References

[1] Gcc: Gnu compiler collection. *http://gcc.gnu.org*.

[2] SPEC benchmark. *http://www.spec.org*.

[3] A. Branover et al. Amd fusion APU: Llano. *IEEE Micro*, 2012.

[4] D. Burger et al. Scaling to the End of Silicon with EDGE Architectures. *IEEE Computer*, 2004.

[5] R. Canal et al. Dynamic cluster assignment mechanisms. *HPCA*, 2000.

[6] A. Capitanio et al. Partitioned register files for vliws: A preliminary analysis of tradeoffs. *MICRO*, 1992.

[7] J. Dehnert, B. Grant, et al. The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges. *CGO*, 2003.

[8] G. Desoli. Instruction assignment for clustered vliw dsp compilers: A new approach. *Technical Report, HP Labs*, 1998.

[9] J. Ellis. Bulldog: A compiler for VLIW architectures. MIT Press, Cambridge, MA, USA, 1986.

[10] P. Faraboschi, G. Brown, et al. Lx: a technology platform for customizable vliw embedded processing. *ISCA*, 2000.

[11] J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 1981.

[12] J. Fridman and Z. Greenfield. The Tigersharc DSP architecture. *IEEE Micro*, 2000.

[13] J. Fritts, F. Steiling, and J. Tucek. Mediabench II video: expediting the next generation of video systems research. *SPIE*, 2005.

[14] M. Gebhart, B. Maher, et al. An evaluation of the TRIPS computer system. *ASPLOS*, 2009.

[15] W. Havanki, S. Banerjia, and T. Conte. Treegion scheduling for wide issue processors. *HPCA*, 1998.

[16] W.-M. W. Hwu, S. A. Mahlke, et al. The superblock: An effective technique for vliw and superscalar compilation. *The Journal of Supercomputing*, 1993.

[17] K. Kailas, K. Ebcioglu, and A. Agrawala. Cars: a new code generation framework for clustered ilp processors. *HPCA*, 2001.

[18] R. Kessler. The alpha 21264 microprocessor. *IEEE Micro*, 1999.

[19] A. Klaiber et al. The technology behind Crusoe processors. *Transmeta Corporation White Paper*, 2000.

[20] W. Lee, R. Barua et al. Space-time scheduling of instruction-level parallelism on a raw machine. *ASPLOS*, 1998.

[21] P. G. Lowney, S. M. Freudenberger et al. The Multiflow trace scheduling compiler. *The Journal of Supercomputing*, 1993.

[22] S. A. Mahlke, D. C. Lin et al. Effective compiler support for predicated execution using the hyperblock. *MICRO*, 1992.

[23] C. McNairy and D. Soltis. Itanium 2 processor microarchitecture. *IEEE Micro*, 2003.

[24] S. Moon and K. Ebcioğlu. An efficient resource-constrained global scheduling technique for superscalar and vliw processors. *MICRO*, 1992.

[25] S. Moon and K. Ebcioglu. Parallelizing nonnumerical code with selective scheduling and software pipelining. *TOPLAS*, 1997.

[26] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann, 1997.

[27] A. Nicolau. Percolation scheduling: A parallel compilation technique. *Technical Report, Cornell University*, 1985.

[28] E. Ozer, S. Banerjia, and T. Conte. Unified assign and schedule: a new approach to scheduling for clustered register file microarchitectures. *MICRO*, 1998.

[29] S. Palacharla, N. Jouppi, and J. Smith. Complexity-effective superscalar processors. *ISCA*, 1997.

[30] G. Pechanek and S. Vassiliadis. The ManArrayTM embedded processor architecture. *Euromicro*, 2000.

[31] V. Porpodas and M. Cintra. LUCAS: latency-adaptive unified cluster assignment and instruction scheduling. *LCTES*, 2013.

[32] K. Sankaralingam, R. Nagarajan, et al. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. *ISCA*, 2003.

[33] H. Sharangpani and H. Arora. Itanium processor microarchitecture. *IEEE Micro*, 2000.

[34] M. Taylor, J. Kim, et al. The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 2002.

[35] A. Terechko and H. Corporaal. Inter-cluster communication in vliw architectures. *TACO*, 2007.

[36] Y. Watanabe, J. Davis, and D. Wood. WiDGET: Wisconsin decoupled grid execution tiles. *ISCA*, 2010.

[37] X. Zhang, H. Wu, and J. Xue. An efficient heuristic for instruction scheduling on clustered vliw processors. *CASES*, 2011.