LUCAS: Latency-adaptive Unified Cluster Assignment and instruction Scheduling *

Vasileios Porpodas, and Marcelo Cintra †
School of Informatics, University of Edinburgh
{v.porpodas@, mc@staffmail.}ed.ac.uk

ABSTRACT

Clustered VLIW architectures are statically scheduled wide-issue architectures that combine the advantages of wide-issue processors along with the power and frequency scalability of clustered designs. Being statically scheduled, they require that the decision of mapping instructions to clusters be done by the compiler. State-of-the-art code generation for such architectures combines cluster-assignment and instruction scheduling in a single unified pass. The performance of the generated code, however, is very susceptible to the inter-cluster communication latency. This is due to the nature of the two clustering heuristics used. One is aggressive and works well for low inter-cluster latencies, while the other is more conservative and works well only for high latencies.

In this paper we propose LUCAS, a novel unified cluster-assignment and instruction-scheduling algorithm that adapts to the inter-cluster latency better than the existing state-of-the-art schemes. LUCAS is a hybrid scheme that performs fine-grain switching between the two state-of-the-art clustering heuristics, leading to better scheduling than either of them. It generates better performing code for a wide range of inter-cluster latency values.

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

Clustered designs were introduced as a solution to the poor scalability of wide-issue processors. This is done by partitioning the design into smaller sections called clusters. Within the cluster, data transfers are fast and energy efficient, while across clusters there is a performance and energy penalty. On the contrary, monolithic (non-clustered) architectures have some bulky resources (such as the register file) that are shared across many functional units and therefore they do not exploit the opportunity to improve performance or to save energy whenever global communication is not required. A clustered design, on the other hand, does exactly that as its resources are partitioned into smaller, locally accessible sections. Each cluster usually contains a portion of the register file tightly connected to a small number of other resources (e.g. functional units). In this way any local communication within the cluster is fast and efficient while any inter-cluster communication comes at extra cost, often higher than that of a monolithic design. It is this partitioning of the global resources and their localization within a cluster that gives the clustered design an advantage in both energy and frequency scaling [26].

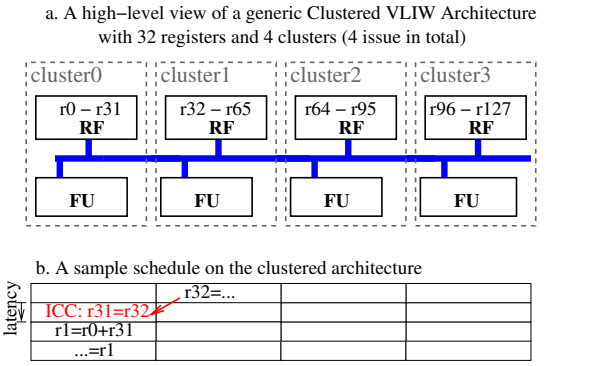

Figure 1. A 4-cluster, 4-issue architecture with 32 registers per cluster (a). An example instruction schedule on the architecture (b).

The clustering approach has been applied to both statically (e.g., [9, 23, 25]) and dynamically (e.g., [15, 22]) scheduled processors. Statically scheduled processors are based on simpler, smaller and more efficient hardware designs than their dynamically scheduled counterparts. VLIW processors, which are both statically scheduled and wide-issue ILP processors, combine the hardware simplicity and energy advantage of statically scheduled processors with the performance of wide-ILP processors, thus operating at a good energy-performance point. Since they are statically scheduled, VLIWs rely on the compiler to generate high performance code. Compared to dynamically scheduled processors, VLIW processors require that instruction scheduling be done in the compiler.

A clustered VLIW processor (as in Figure 1a) has an additional performance and energy advantage compared to its non-clustered counterpart due to the scalability of the design. In this case though, the compiler has to perform yet another task, that of cluster assignment, deciding the cluster where each instruction should be executed at (as shown in the code example of Figure 1b).

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

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

LCTES'13, June 20–21, 2013, Seattle, Washington, USA.

1.1 Code Generation

Originally, cluster assignment was done just before instruction scheduling, in a separate pass [8]. The clustering algorithm traverses the data-flow graph and assigns the instructions to clusters in a greedy manner. The cluster selected is the one suggested by the clustering heuristic. The two state-of-the-art clustering heuristics (Start-Cycle and Completion-Cycle) differ in their aggressiveness. The first one will eagerly spread instructions across clusters as long as the one-way latency cost is covered, hoping for good performance, whereas the latter will only do so if the round-trip cost is covered. The clustering scheme has a major impact on performance and is strongly affected by the inter-cluster latency.

More recent work has combined the instruction scheduling pass with the cluster assignment pass in an attempt to remove some phase-ordering issues between the two [14, 21]. This is done by modifying the instruction scheduler so that upon scheduling an instruction, it also decides on the cluster where it should be assigned to based on the value of the clustering heuristic.

1.2 Contributions

In this paper we identify a fundamental weakness of the existing state-of-the-art heuristics for combined instruction scheduling and cluster assignment. The code generated by these algorithms performs well under very limited conditions. Depending on the heuristic used, they work well under either: i) low inter-cluster latencies, or ii) high instruction latencies. To make matters worse, the intersection point, where one heuristic overtakes the other, varies significantly and is benchmark specific.

In short in this paper:

1. We present a detailed comparison of the best state-of-the-art clustering heuristics (built inside an instruction scheduler) on a range of inter-cluster delays.
2. We propose a novel clustering heuristic that **i)** adapts to the inter-cluster latency and performs best across a wide range of inter-cluster latencies and **ii)** often outperforms both existing heuristics

In the rest of the paper we start by describing some of the fundamental concepts involved (Section 2). Afterwards we motivate the proposed work by identifying the weaknesses of the state-of-the art (Section 3), then we discuss the proposed work in full detail (Section 4) and the experimental setup (Section 5) used to get the results shown in Section 6. Finally we present an overview of the related work (Section 7) and we conclude in Section 8.

2. BACKGROUND

Our work is based on two fundamental concepts. The first one is the clustering heuristics and the second one is instruction scheduling.

2.1 Clustering heuristics

In this section we present the state-of-the-art clustering heuristics which are implemented in several algorithms.

Start-Cycle (SC): Existing combined cluster-assignment and instruction scheduling schemes [14, 21] make use of the same clustering heuristic, which according to [21] proves to be the highest performing one when compared to other greedy heuristics. It is the resource-constrained earliest schedule cycle heuristic, also known as the Start-Cycle. In more detail, it returns the earliest cycle that an instruction can be scheduled at on any given cluster, taking into account not only the dependence constraints of the predecessors but also the inter-cluster latency and the issue-slot occupancy (resource) constraint (Algorithm 1). An example that visualizes how the heuristic works is in Figure 2b,c in red color. The two Start-Cycle values (for (B,CL0) and (B,CL1)) show the value returned by the heuristic for cluster 0 and 1 respectively.

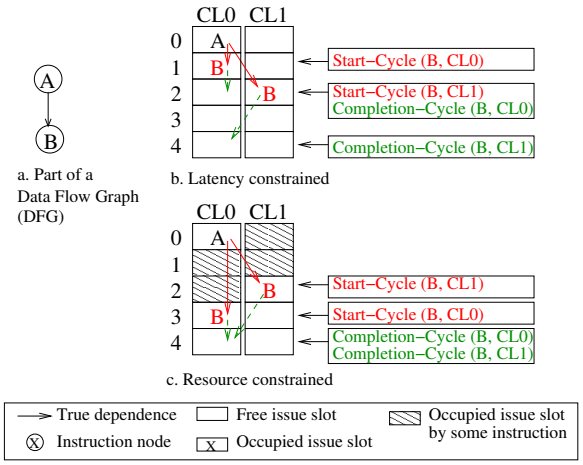

Figure 2. The internal workings of Start-Cycle and Completion-Cycle clustering heuristics. Heuristic(N, CLx) signifies the value of the Heuristic when node N is placed on cluster X.

The Start-Cycle heuristic spreads the instructions across the clusters in an aggressive and greedy manner. Each and every instruction gets scheduled on the cluster where it will execute the earliest. As shown in Section 3, this strategy proves to work best on low inter-cluster communication latencies, but the performance degradation on high latencies is unbounded.

Algorithm 1. Start-Cycle heuristic.

```
1  /* Start-Cycle Heuristic */
2  start_cycle (insn, cluster)
3  {
4   i = 0
5   for pred in insn's predecessors:
6      dst = Distance (pred.cluster, cluster)
7      latency_aware_sc = pred.cycle +pred.latency +dst
8      /* Increase cycle until we get a free resource*/
9      cycle = latency_aware_sc
10     while reservation_table_not_free(cluster,cycle):
11        cycle ++
12     resource_and_latencyaware_sc = cycle
13     sc [i++] = resource_and_latency_aware_sc
14   return MAX of all sc[ ]
15 }
```

Completion-Cycle (CC): The problem of Start-Cycle's unbounded performance degradation is addressed in other clustering works (e.g., [8, 17]), which are based on the Completion-Cycle heuristic. This is a conservative clustering heuristic that distributes the instructions only if it is guaranteed that they will not cause a slow-down at that scheduling point.

It works by calculating the Start-Cycle and adds to it the latency of the instruction and the latency until this instruction's data is sent over to its earliest successor (Algorithm 2). Since the cluster number of the successors is only known for the instructions just before the end of a region, the cluster number of the successors is zero for the majority of cases. This case is shown in Figure 2b,c in green color.

Critical-Successor (CS): A more recently introduced clustering algorithm was presented in [28]. The clustering heuristic introduced by it is based on the observation that when a sibling instruction node has been already assigned to a cluster, then it is highly probable that there exists an immediate successor of it that is also a highly critical immediate successor of the current instruction node. In this case the clustering heuristic should select the cluster that achieves the best start-cycle, not of the current instruction but of the critical-successor node instead. To be more precise, the critical-

successor start-cycle is selected only if it can pinpoint a single clear winner out of all clusters. The heuristic defaults to standard start-cycle if the code does not meet any of these constraints. The CS heuristic exhibits similar behavior to SC with respect to the increasing inter-cluster delay, mostly due to the fact that it is built upon the start-cycle heuristic. We will therefore focus on the other two heuristics for the following sections.

All heuristics are greedy and are calculated once on a single top-down walk of the DFG with no backtracking. Thus they cannot guarantee a globally optimal solution.

Algorithm 2. Completion-Cycle heuristic.

```
1  /* Completion-Cycle Heuristic */}
2  completion_cycle (insn, cluster)
3   {
4     i = 0
5     start_c = start_cycle (insn, cluster)
6     for succ in insn's successors:
7        dist = distance (succ.cluster, cluster)
8        cc [i++] = start_c + dist
9     return MIN of all cc[ ]
10  }
11 }
```

2.2 Instruction Scheduling

Instruction scheduling is traditionally done by a list scheduler. The list scheduling algorithm works as shown in Algorithm 3. Its input is a Data Dependence Graph (DDG) and its output is the instruction schedule. In short it follows the following steps:

1. Walk the dependence graph and prioritize the nodes (usually based on their height from the bottom of the DDG) (Algorithm 3 line 4).

2. While there are unscheduled nodes, form a list of ready instructions (instructions with scheduled predecessors or with no predecessors at all) (Algorithm 3 lines 6 - 7).

3. Sort the ready list based on the priority of each instruction (Algorithm 3 line 8).

4. Start from the instruction with the highest priority and try to issue it on the current cycle (Algorithm 3 lines 9-11). If this is not possible then skip it (line 13) and try the next instruction in the ready list. In any case remove the current instruction from the ready list (line 14).

5. Once tried all the instructions in the ready list have been considered then increase the current cycle by 1 (Algorithm 3 line 15).

Algorithm 3. Simplified List Scheduling.

```
1  /* List Scheduling: Input DDG, Output: Schedule */
2  list_schedule (ddg)
3  {
4   walk down the ddg and prioritize the nodes
5   cycle = 0
6   while (exist unscheduled nodes):
7      ready_list = list of ready nodes
8      sort ready_list based on priority
9      for node in prioritized ready_list:
10        if (can issue node.instruction on cycle):
11           Issue (node.instr, cycle)
12        else
13           Skip node
14        Remove node from ready_list
15     cycle ++
16 }
```

3. MOTIVATION

The major weakness of the state-of-the-art cluster-assignment and instruction-scheduling algorithms is that their clustering heuristics perform well on a limited range of inter-cluster communication latencies. Figure 3 points out this fact. The Start-Cycle (SC) (and Critical-Successor (CS)) heuristics perform well only on low-latency configurations. The Completion-Cycle (CC), on the other hand performs well only on high-latency configurations. Moreover the intersection point is highly specific to the benchmark and varies unpredictably.

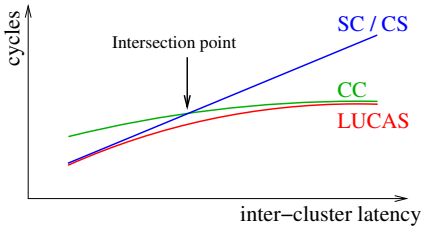

Figure 3. Qualitative performance comparison of clustering heuristics under increasing inter-cluster latency: Start-Cycle (SC), Critical-Successor (CS), Completion-Cycle (CC) and the proposed heuristic used in LUCAS.

The proposed scheme (LUCAS) addresses the shortcomings of both heuristics by adapting to the inter-cluster latency. LUCAS switches between the aggressive (SC) and conservative (CC) heuristic on a per-instruction basis. As shown in Figure 3 the goal of the proposed approach is to provide the best performance across the whole range of inter-cluster latencies.

3.1 Clustering Heuristics

The reason why the state-of-the-art heuristics perform in general as in Figure 3 and why our heuristic performs the way it does, can be explained by the motivating examples of Figures 4 and 5. The LUCAS heuristic uses two sub-heuristics: i) the cycle-congestion (Figure 4) and ii) the instruction mobility (Figure 5), to guide the decision on when to use the start-cycle or the completion-cycle heuristic. This will be explained in more detail later on. The examples of Figures 4 and 5 show the schedules acquired after scheduling the nodes of the Data Flow Graph (DFG) (Figure 4a and 5a) using the clustering heuristics (vertical axis) for inter-cluster latencies of 1 to 3 cycles (horizontal axis).

The Start-Cycle heuristic (Figures 4 and 5 b-d) performs well on low latencies but the schedule length increases almost linearly to the inter-cluster latency. This is because the heuristic is very aggressive at dispersing the instructions across distant clusters.

On the contrary, the Completion-Cycle heuristic (in both Figures 4 and 5 e-g) performs best under high inter-cluster communication latencies. The schedule length remains unchanged no matter the inter-cluster latency. The reason for this is that an instruction will only be scheduled on a distant cluster if its descendants are not slowed down. This conservative policy bounds the schedule length for high latencies but proves not as effective for low latencies.

LUCAS adjusts better to the inter-cluster latency. We show how it does so by demonstrating how each of the sub-heuristics works in each example (Figures 4 and 5). The Cycle-Congestion sub-heuristic (Figure 4) measures the congestion on each scheduling cycle. If there are too many ready instructions to fit in a single cluster, then it chooses to follow the aggressive Start-Cycle heuristic. This happens in cycles 0 and 1 in Figure 4.h and i (latency 1 and 2). On later cycles however, there is no congestion and therefore instruction 'E' is scheduled based on the conservative Completion-Cycle heuristic.

The instruction Mobility sub-heuristic is shown in Figure 5. The concept is that if an instruction has a high enough mobility, then its slack is high and thus there is little chance that it can degrade the schedule if assigned to a distant cluster (the mobility is calculated as ALAP-ASAP as in [16]). Therefore high-mobility instructions are scheduled with the Start-Cycle heuristic. The mobility numbers

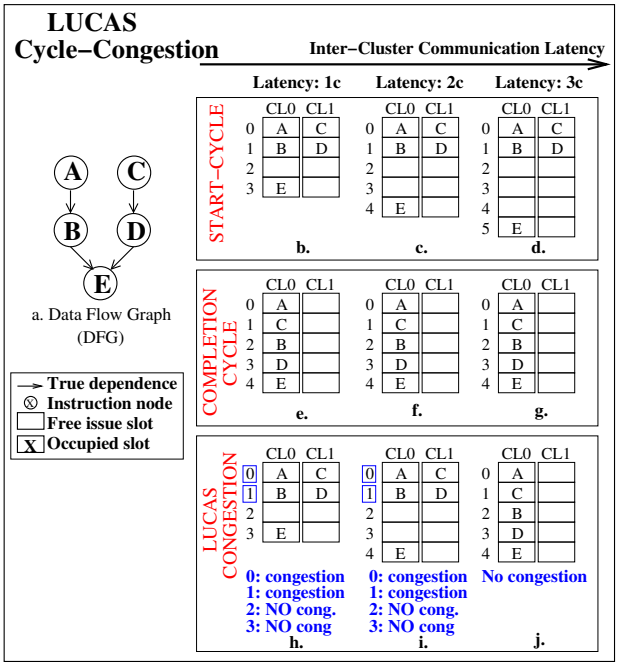

Figure 4. Motivating example 1. Schedules for the instructions in the Data Flow Graph (DFG) (a) on a 2-cluster 2-issue clustered architecture, for the Start-Cycle, Completion-Cycle and LUCAS-Cycle-Congestion clustering heuristics. The inter-cluster delay ranges from 1 to 3 cycles.

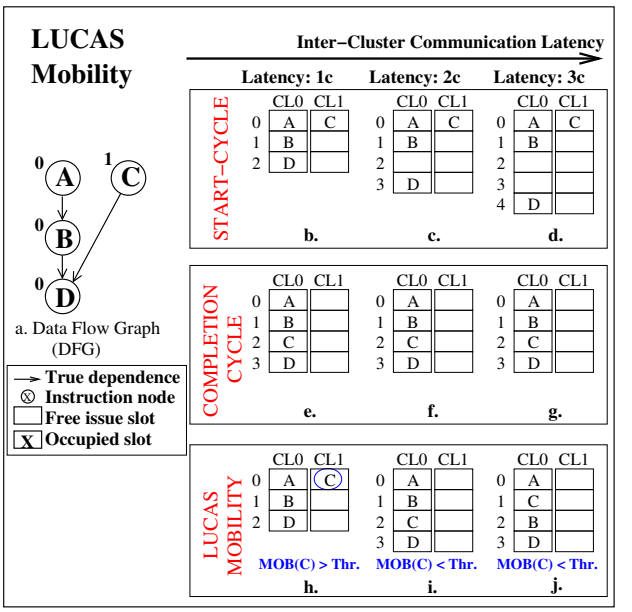

Figure 5. Motivating example 2. Schedules for the instructions in the Data Flow Graph (DFG) (a) on a 2-cluster 2-issue clustered architecture, for the Start-Cycle, Completion-Cycle and LUCAS-Mobility clustering heuristics. The inter-cluster delay ranges from 1 to 3 cycles. Each node in the DFG is tagged with its mobility number.

are shown in the DFG of Figure 5 on the left side of each instruction. Instruction 'C' has mobility 1 which is higher than the threshold for Latency 1. Therefore in that case 'C' is scheduled in Cluster 1, as dictated by the Start-Cycle heuristic.

As shown in the motivating examples, LUCAS is capable of adapting to the best clustering heuristic, for the whole range of inter-cluster communication latencies. The detailed description of the LUCAS algorithm and the sub-heuristics used is presented later in Section 4.

3.2 Scheduling

While both UAS and CARS [14, 21] make use of a list scheduler, they have embedded the clustering decision inside the instruction scheduler in a different way.

CARS [1] always honors the clustering decision and schedules only on the cluster chosen by it (see Figure 6a). The clustering heuristic tags each cluster with a score and next the cluster with the best score wins (Figure 6.a.2 BEST CLUSTER).

On the contrary UAS [21] is more aggressive. It tries to honor the clustering decision only at the first attempt, but if it fails to issue the instruction on the specified cluster, it will try other clusters as well (Figure 6.b). Therefore the cluster with the best score does not always win (Figure 6.b.3). This is an aggressive technique that might work on low inter-cluster latencies but it performs poorly on higher latencies. As shown in Section 6, this method has no major impact on performance even for low inter-cluster latencies when combined with the Start-Cycle heuristic of Algorithm 1 as its aggressiveness is overshadowed by that of the Start-Cycle heuristic.

LUCAS aims at performing best on the whole range of inter-cluster latencies. Therefore it honors the clustering decision made by the heuristic (similarly to CARS) as in Figure 6a.

[1] CARS also performs register allocation, which is not shown.

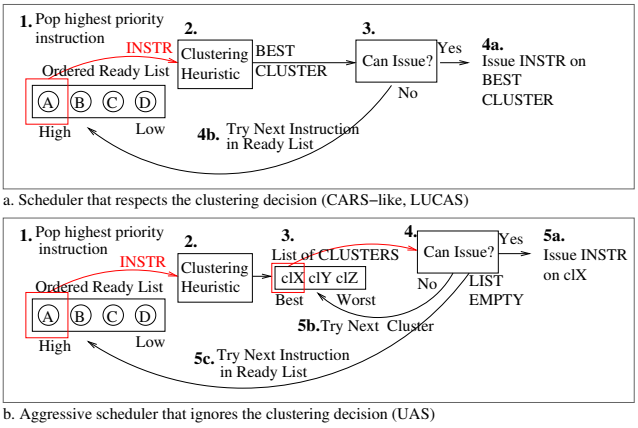

Figure 6. The two variants of embedding the clustering heuristic into the instruction scheduler. The numbers denote the order of execution of each step.

4. LUCAS

The proposed Latency-aware Unified Cluster-Assignment and instruction Scheduling algorithm addresses the shortcomings of the existing algorithms (discussed in Section 3).

LUCAS is a list-scheduling-based algorithm that performs cluster assignment and instruction scheduling simultaneously. The novelty lies in the clustering heuristic. The algorithm is listed in Algorithm 4. A high-level view of the structure of the algorithm is shown in Figure 6a.

In detail, LUCAS performs the following actions:

1. It assigns a priority number to all instruction nodes of the DFG (Algorithm 4 line 6) using a priority function (for example the instruction height in the DFG).

2. It updates the ready list with instructions ready to be issued on the current cycle (line 9).
3. It sorts the ready list based on the node priorities of step 1 (line 10).
4. Before scheduling the instruction under consideration, the algorithm determines the *best_cl* (best cluster) by evaluating the heuristic for each candidate cluster and choosing the best among those (Algorithm 4 line 23). The "get_best_cluster()" function incorporates the adaptive heuristic.
5. Then the algorithm tries to schedule the instruction only if it meets the Start-Cycle constraint (which includes both dependence and clustering-related structural constraints) (Algorithm 4 line 13).
6. If all processor structural constraints allow scheduling the instruction at the current cycle on *best_cl* (Algorithm 4 line 14), then we can proceed.
7. If the required Inter-Cluster Copies (ICCs) can be emitted on the inter-cluster network (that is if the network is not fully occupied) (line 15), then it emits the ICCs and register renames the instructions that use the register brought in by the ICCs (line 16) and it finally cluster-assigns and issues the instruction on *best_cl* (lines 17,18).
8. If the instruction has been placed on a distant cluster, then update its mobility metric (decrement it by the inter-cluster delay (ICD)) to reflect this change (line 19). The intuition behind this is that the ICD consumes some of the ability of the instruction to move freely.
9. Repeat steps 5-9, by selecting the highest priority node until the ready list is empty (line 11).
10. Finally repeat steps 2-10 until all instructions are scheduled (Algorithm 4 line 8).

The LUCAS heuristic is a hybrid Start-Cycle / Completion-Cycle heuristic. It decides per instruction which of the two to use based on two metrics:

1. The **cycle congestion** (Algorithm 4 line 38). This is a binary metric. It returns true if there are too many instructions to schedule on the current cycle. That is if the number of instructions that are ready on the current cycle are greater than the congestion threshold. The threshold reflects both the issue resources of a cluster and the inter-cluster penalty. It is computed as the product: Issue-Width Per Cluster (IWPC) times the Inter-Cluster Delay (ICD).
2. The **mobility** of the instruction (Algorithm 4 line 39). The mobility is calculated as ALAP-ASAP values in the Data-Flow-Graph [16]. A high mobility value suggests that there is enough slack in the schedule for the instruction to be executed later with no guaranteed side-effects in the schedule. The mobility threshold corresponds to the inter-cluster round-trip time.

The actual algorithm for the lucas heuristic is listed in Algorithm 4 in *get_best_cluster()* function. It works as follows:

- At first each candidate cluster is tagged with the heuristic value (Algorithm 4 line 25). This uses the *lucas()* function (Algorithm 4 line 36).
- The LUCAS heuristic checks the two metrics (cycle congestion and instruction mobility sub-heuristics) (lines 38-39) for the instruction to be scheduled and decides on the heuristic to be used for the clustering decision (line 40). This is the core of the LUCAS heuristic. The metrics decide whether the aggressive Start-Cycle heuristic (line 41) or the more conservative Completion-Cycle heuristic is used (line 43).

Algorithm 4. LUCAS: Latency-adaptive Clustering and Scheduling.

```
1  /* LUCAS Scheduling and clustering.
2     Input:  DFG
3     Output: Clustered Schedule  */
4  lucas_schedule_and_cluster (DFG)
5  {
6   walk down the DFG and prioritize the nodes
7   cycle = 0
8   while (exist unscheduled nodes)
9     Fill in ready_list
10    sort ready_list based on priority
11    for node in prioritized ready_list
12      best_cl = get_best_cluster(node.instr, cycle)
13      if (start_cycle (node.instr, cluster)<=cycle)
14        if (can issue node.instr on cycle)
15          if (can schedule Inter-Cluster Copies)
16            Emit ICCs and reg. rename node.instr
17            node.cluster = best_cl
18            Issue (node.instr, cycle, best_cl)
19            Update MOBILITY(node.instr) if it gets
                  ↪data from a distant cluster
20    cycle ++
21 }
22
23 get_best_cluster (insn, cycle)
24 {
25  for cluster in all clusters
26    heuristic[cluster] = lucas(insn,cluster)
27  /* Find best cluster: MIN_CL */
28  min_cl = 0
29  for cli in clusters
30    if (heuristic[cli] < heuristic[min_cl])
31      min_cl = cli
32  return min_cl
33 }
34
35 /* Return the score of CLUSTER */
36 lucas (insn, cluster)
37 {
38  high_congestion = (#Ready-instr. > IWPC × ICD)
39  high_mobility = (MOBILITY(insn) > IWPC×2×(ICD-1))
40  if (high_congestion OR high_mobility)
41    return start_cycle (insn, cluster)
42  else
43    return completion_cycle (insn, cluster)
44 }
```

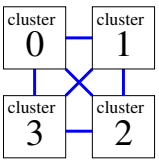

Figure 7. The fully-connected point-to-point interconnect.

- Finally, the algorithm does a linear search over all clusters to find the cluster with the minimum heuristic value (line 29) (as shown in Figure 6.a.2). Once found, the cluster that corresponds to the minimum value of the heuristic is returned as the best cluster (line 32).

5. EXPERIMENTAL SETUP

5.1 Architecture

The target architecture is an IA64 (Itanium2) ISA based statically scheduled clustered VLIW architecture. The architecture is configured to have 4 clusters with an issue-width of 4 or 8 (1 or 2 issue per cluster).

Processor: IA64 based clustered VLIW				
Issue Width:	4 or 8			
Clusters:	4			
Instruction Latencies:	Same as Itanium2 [19]			
Register File:	(32GP, 32FL, 16PR) per cluster			
Inter-Cluster Delay:	1 - 4 cycles			
Inter-Cluster Bus Bandwidth:	∞			
Branch Prediction:	Perfect			
Cache: Levels 3 (same as Itanium2 [19])				
Levels :	L1	L2	L3	Main Mem.
Size (Bytes):	16K	256K	3M	∞
Block size (Bytes):	64	128	128	-
Associativity:	4-Way	8-way	12-way	-
Latency (cycles):	1	5	12	150

Table 1. Processor configuration.

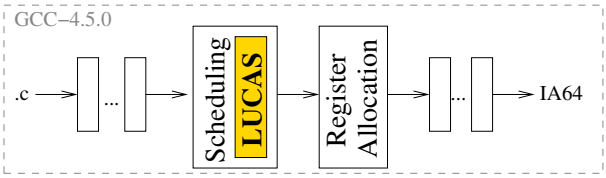

Figure 8. Overview of the GCC compilation pipeline.

The inter-cluster communication bandwidth is infinite [2] , meaning that there is no limit in the count of the simultaneous inter-cluster communications. Thus our results have no noise from any inter-cluster bandwidth effects.

The clusters communicate through a fully-connected point-to-point interconnect as shown in Figure 7. All clusters communicate with each other with equal latencies. The latency is adjustable and in our experiments it ranges from 1 to 4 cycles.

The architecture configuration is summarized in Table 1.

5.2 Compiler

We implemented both UAS [21] and the proposed (LUCAS) unified clustering and scheduling algorithms along with all clustering heuristics (see below) in the instruction scheduling pass of GCC-4.5.0 [1] cross compiler with Itanium ([24]) as the target ISA (IA64). As shown in Figure 8 the instruction scheduler (with the clustering built-in) runs before register allocation.

The implementation of the scheduler enables us to easily swap the clustering heuristics while the rest of the instruction scheduling pass remains unchanged. The heuristic is one of the following: i) Start-Cycle ([21]), ii) Completion-Cycle ([8]), iii) Critical-Successor ([28]) or iv) LUCAS (the proposed one).

5.3 Evaluation

We evaluated LUCAS on the 4-cluster architecture described in Section 5.1 configured as a 4-issue and an 8-issue machine. We compare the LUCAS heuristic against the state-of-the-art Start-Cycle (SC) and Completion-Cycle (CC) as well as the recently proposed Critical-Successor (CS) clustering heuristic. In addition we compared all these against an accurate implementation of the UAS algorithm. The algorithms and heuristics compared are summarized in Table.2.

We evaluated LUCAS against the existing state-of-the-art heuristics on 6 of the Mediabench II video [11] benchmarks. All benchmarks were compiled with -O2 optimizations enabled. Each benchmark is compiled several times, once with each clustering heuristic enabled, and each binary is then executed on our modified ski simulator [2], configured as discussed in Section 5.1.

[2] This means that the condition in Algorithm 4 line 15 is always true.

	Algorithm	Heuristic		
	Obeys Heuristic	Start Cycle	Completion Cycle	Critical Successor
UAS	×	√	×	×
SC	√	√	×	×
CC	√	×	√	×
CS	√	√	×	√
LUCAS	√	√(Hybrid)	√(Hybrid)	×

Table 2. Evaluated schemes.

6. RESULTS AND ANALYSIS

We have two kind of results: i) the performance results (normalized to the Start-Cycle for delay 1), shown in Figures 9 and 10, which show that LUCAS meets its performance goals and ii) the instruction distribution measurements (Figures 11 and 12) that provide important insights into the workings of the heuristics. Both of these results show two LUCAS heuristics: LUCAS-C which is based only on the Congestion sub-heuristic and LUCAS-C-M which is the full version with both Congestion and Mobility enabled. This is a useful breakdown that lets us better understand the effects of each part individually.

6.1 Performance

The first thing that stands out is the non-scalability of the UAS[21], the Start-Cycle (SC) ([8], Algorithm 1) and the Critical-Successor (CS) ([28]) heuristics. The performance degradation increases almost linearly with the delay as seen in Figure 9. This is caused by the aggressiveness of the Start-Cycle heuristic, which spreads instructions on distant clusters, disregarding the cost of communicating the results back after they have been computed. The Critical-Successor heuristic is partly based on the Start-Cycle, which contributes to its non-scalability.

The performance of **UAS** is very close to that of the Start-Cycle heuristic. As already explained in Section 3.2, the UAS scheduler uses a variation of the Start-Cycle clustering heuristic (which in [21] it is referred to as CWP), but the scheduling algorithm follows a different approach in selecting a cluster. UAS may ignore the decision of the clustering heuristic if it cannot schedule on the chosen cluster due to resource constraints (see Figure 6a). This is a greedy gamble as the scheduler tries to assign an instruction to any cluster possible, even if this means ignoring the primary decision of the clustering heuristic. This does not happen in the unified clustering and scheduling algorithm that we propose (Figure 6.b). In our approach, the primary decision of the clustering heuristic is honored by the scheduler. The CC, SC, CS and LUCAS heuristics follow this second approach. As shown in the results, UAS performs on average very similarly to the Start-Cycle heuristic of our algorithm. The reason is that the Start-Cycle heuristic is aggressive enough and usually overshadows the aggressiveness of the UAS algorithm.

The **Completion-Cycle** heuristic (Algorithm 2) keeps performance at a reasonable level. The reason is that the heuristic is conservative. It only issues an instruction on a distant cluster if it can prove that it is beneficial even in case it needs to send the data back. Therefore if the inter-cluster latency is high, usually the round-trip latency is too expensive and the Completion-Cycle heuristic will keep the instructions on the same cluster. This however proves to be inadequate for low inter-cluster latencies (e.g. Figure 9 mpeg2dec). In the worst case the Start-Cycle heuristic outperforms the Completion-Cycle by over 40% (Figure 9 mpeg2dec).

The measurements of Figure 9 show that while the Completion-Cycle heuristic is better at high inter-cluster delays (e.g. Figure 9 djpeg latency 2 or more), the Start-Cycle heuristic usually works best at low inter-cluster delays. That is when being aggressive at spreading the instructions across clusters as much as possible proves a better choice than being conservative. This is the main motivation behind LUCAS. If both of these approaches are com-

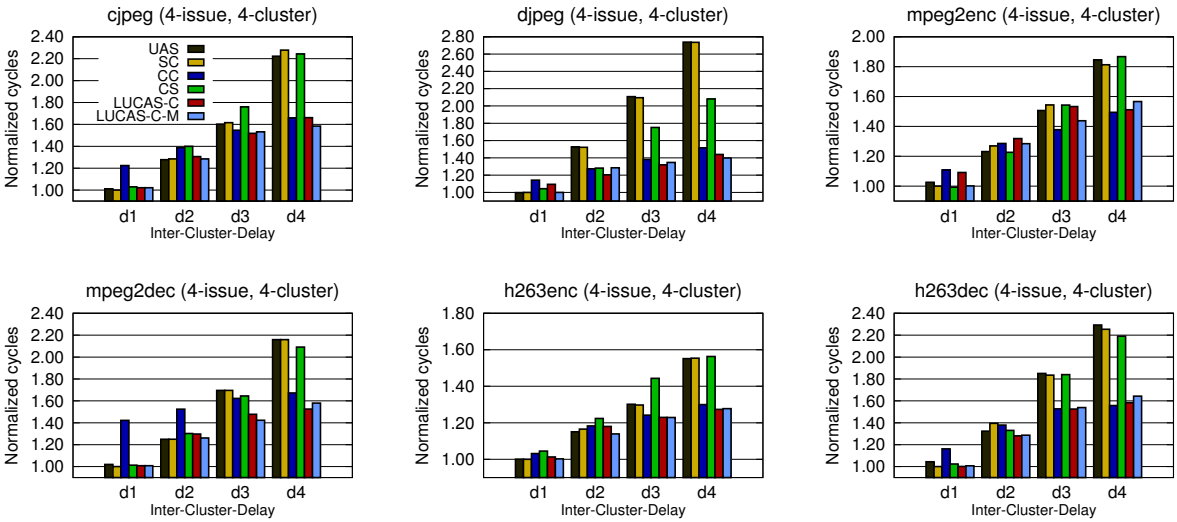

Figure 9. Cycles of the 4-issue,4-cluster configuration for inter-cluster delay 1 to 4, normalized to Start-Cycle (SC), delay 1.

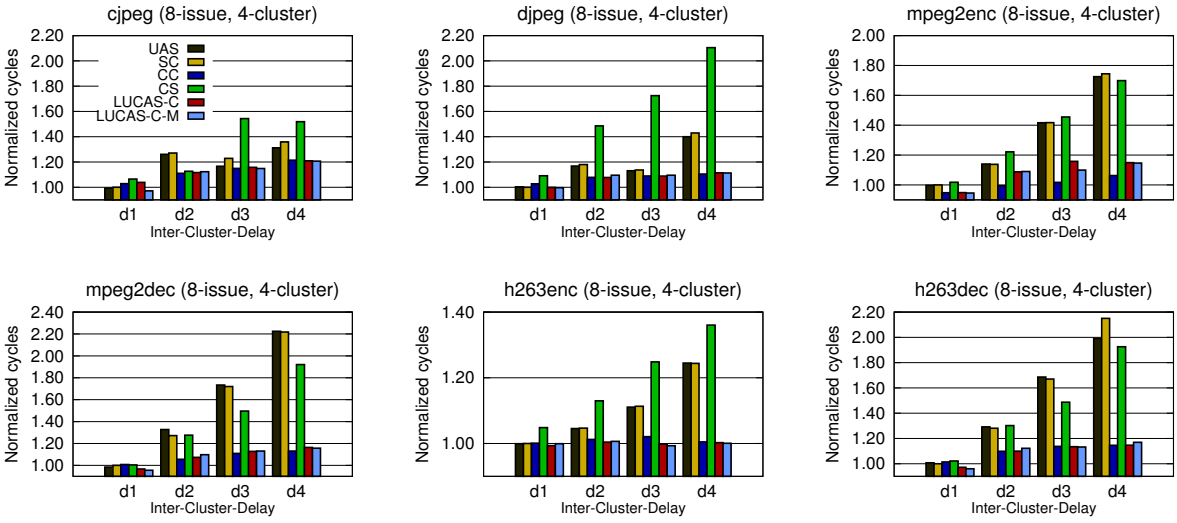

Figure 10. Cycles of the 8-issue,4-cluster configuration for inter-cluster delay 1 to 4, normalized to Start-Cycle (SC), delay 1.

bined together, then we can get a clustering heuristic that performs well across all inter-cluster delays. This assumption is confirmed by the LUCAS results of Figure 9.

The **intersection point** where the Start-Cycle heuristic overtakes the Completion-Cycle heuristic is not fixed. It is can be between delay 1 and 2 (Figure 9 djpeg) or between delay 2 and 3 (e.g. Figure 9 cjpeg). Therefore selecting the right heuristic cannot be based on some fixed magic number. LUCAS performs an effective switching between Start-Cycle and Completion-Cycle with the help of two metrics: the cycle congestion and the instruction mobility.

LUCAS does not only adapt to the best heuristic, but it quite often outperforms both heuristics (e.g. Figure 9 mpeg2dec d3,d4, h263enc d2,d3,d4 and Figure 10 mpeg2dec d1, h263dec d1). This is intuitive because LUCAS performs a **fine-grain switching** between the Start-Cycle and Completion-Cycle heuristic at the instruction level. This can select the best heuristic at a fine granularity, when it is needed, which is better in the long run than selecting one of the two for the duration of the whole program.

The two **sub-heuristics** that form LUCAS, Congestion (C) and Mobility (M), do work together and when combined (logical OR) usually lead to better overall performance. The gains from applying the Mobility heuristic on top of the Congestion one are up to 9% (Figure 9 mpeg2enc d3). In a few cases however, performance decreases (3.5% in the worst case). The reason behind the behavior is that under high inter-cluster delays, any further aggressiveness (introduced by the logical OR-ing of the heuristics), is usually for the worse.

Overall, in most cases LUCAS performs very closely to the best heuristic or better than it (e.g. cjpeg). There are some outliers though. The mpeg2enc stands out from the rest, as for both the 4-issue and 8-issue setups LUCAS cannot keep up with the best for high inter-cluster delays, although it is still much better than UAS, SC and CS. In case of the 4-issue machine, the differences are great,

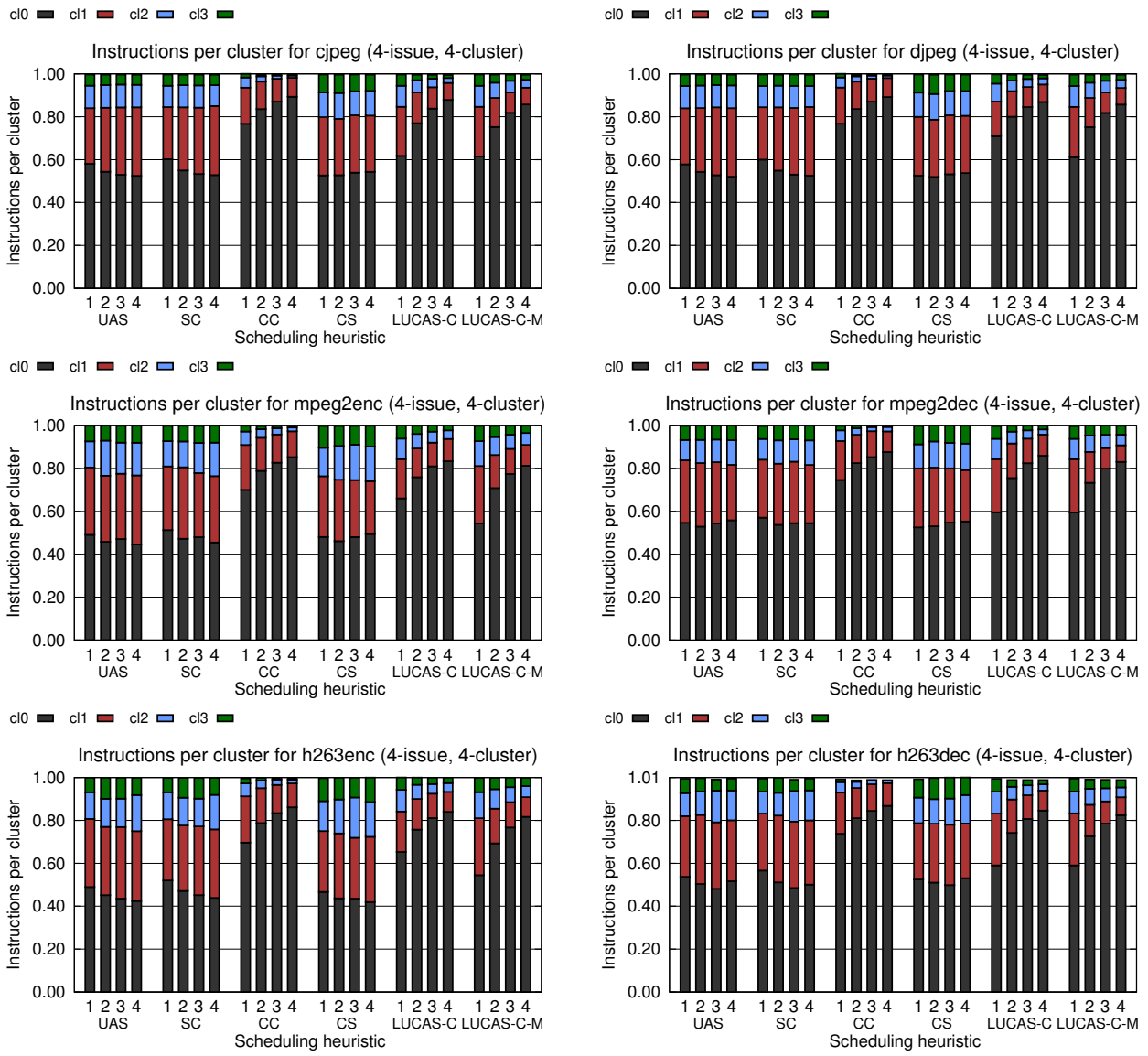

Figure 11. Distribution of instructions on each cluster, for all clustering heuristics and for delays ranging from 1 to 4. This is for the 4-issue 4-cluster machine.

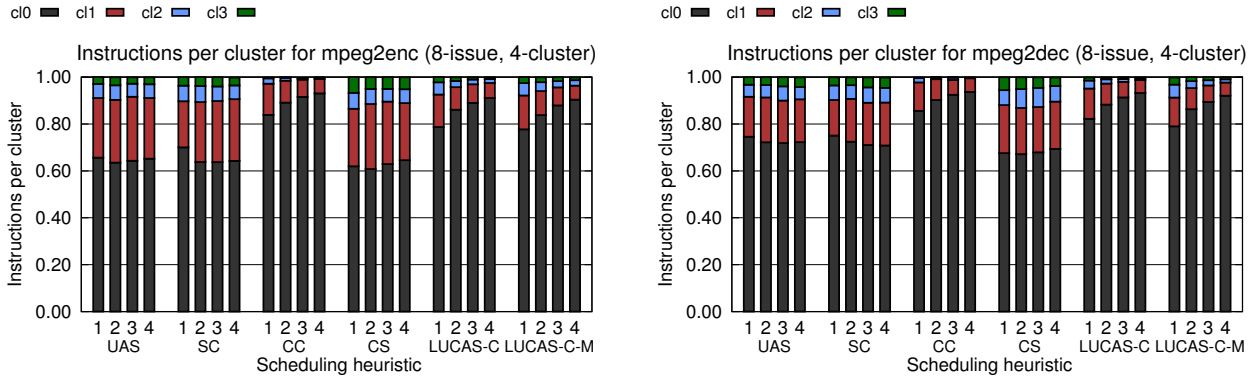

Figure 12. Distribution of instructions on each cluster, for all clustering heuristics and for delay ranging from 1 to 4. This is for the 8-issue 4-cluster machine and just for the mpeg2 benchmarks.

however on the 8-issue machine, where the performance penalties get amplified, this effect is more evident. The mpeg2enc, 8-issue case is a special case as it is the only one that is strongly biased against the Start-Cycle heuristic even for delay 1. Therefore any attempt to spread the instructions to distant clusters will lead to a slowdown. In most other cases if LUCAS performs worse than the best performing heuristic it performs marginally worse (e.g. Figure 10 djpeg d2,d3).

6.2 Instruction Distribution

To provide more insights into the internals all clustering heuristics, including LUCAS, we show the distribution of the program instructions across clusters for all heuristics and for both machine types (Figures 11 and 12). Each of the stacked bar shows the breakdown of the instructions on each cluster (each cluster is represented by a color). Each heuristic corresponds to 4 stacked bars, one for each inter-cluster delay (ranging from 1 to 4). We observe that:

1. First of all, some of the graphs look strikingly similar. For example the breakdowns for Figure 11 cjpeg and djpeg look very similar. This is due to the fact that these benchmarks share a lot of common source files. Since the instruction counts are statically computed, the differences between the benchmarks are minimized.
2. On the 4-issue machine (Figure 11), about 60% of the code is executed on the first cluster, and the rest of it is spread across the rest for inter-cluster delay of 1. The further away from cluster 0, the fewer the instructions. The second cluster (cl1) usually contains about 25% of the instructions, the third cluster (cl2) about 10% and the last one contains about 5%. This behavior is intuitive as any inter-cluster communication has an extra overhead, forcing the scheduler to be reluctant on spreading the instructions across clusters, doing so only when absolutely necessary. This effect gets amplified on the 8-issue machine (Figure 12), where there is usually little need for extra issue slots on other clusters. This is why, on this configuration there are even more instructions ($>$ 80% in some cases) in cluster 0 and fewer in the rest. It is worth noting that the first cluster (cl0) is of no particular significance as the architecture is a symmetric one, as shown in Figure 7.
3. The fundamental difference of the heuristics can be observed as we increase the inter-cluster delay. The aggressive heuristics (UAS, SC and CS) do not seem to adjust to the increase in the inter-cluster delay. Instead of being more conservative in scheduling across clusters, they seem to become even more aggressive (the instructions on cl0 decrease as the delay increases). On the other hand the conservative CC heuristic behaves in the opposite way. As the inter-cluster delay increases, it tries to keep more instructions within cl0. The LUCAS heuristic (LUCAS-C-M in particular), bridges the gap between these two opposite strategies. For small inter-cluster delays it behaves almost like the aggressive heuristics, but as the inter-cluster delay increases, it behaves as the conservative one.

6.3 Algorithmic Complexity

This section calculates the algorithmic complexity of LUCAS. We do that by examining the algorithm (Algorithms 4, 1 and 2). Let's consider an input DFG of N nodes. The LUCAS Scheduling algorithm has 2 visible levels of nested loops (the 3rd is in the Start-Cycle calculation):

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 quick sort. 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. The best cluster is found by get_best_cluster(). This iterates once over all clusters and sets the Start-Cycles. 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 this costs RCP.

The complexity of LUCAS Scheduling is computed as:

- $N/S \times R \times (logR + CP)$ in the usual case
- $N/S \times R \times (R + CP)$ in the worst case

In all practical cases all S, R, 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 $S = 1$ and $R = N$, $P = N$ which leads to complexity $O(N^3)$.

UAS has a similar 3-nested loop structure and exhibits similar complexity. For all practical cases, the UAS is $O(N)$ and in the worst-case it is $O(N^3)$. Therefore both schedulers have similar complexity.

7. RELATED WORK

This section discusses the previous work on cluster assignment and instruction scheduling for clustered VLIW architectures, that is closely related to our work.

7.1 Combined Cluster Assignment & Instruction Scheduling

The first work that proposes a combined instruction scheduling and clustering pass is Unified Assignment and Scheduling (UAS) [21]. The scheduling algorithm is a modified list scheduler. In this work cluster assignment is aggressive in two ways:

i) This work uses the aggressive Start-Cycle (SC) heuristic for the clustering part (in the terminology of [8]) (or CWP in the terminology of [21]) which is shown to be the best performing one over several others on the architecture that was evaluated. The inter-cluster delay is fixed to 1 cycle, which explains why the Start-Cycle heuristic was found to be the best performing of the heuristics tried out. In our work we show that the Start-Cycle causes an unbounded performance degradation as the inter-cluster latency is increased.

ii) The scheduling algorithm is such that will try to schedule an instruction on the current cycle even if this cluster is not the first choice of the clustering heuristic.

Compared to UAS, LUCAS will always obey the decision of the clustering heuristic. In LUCAS, the heuristic is a hybrid one that switches between the aggressive Start-Cycle and the more conservative Completion-Cycle (CC).

CARS ([14]) is a combined scheduling, clustering, and register allocation code generation framework based on list-scheduling. Similarly to UAS, the Start-Cycle is the heuristic that steers the clustering decisions.

Recently, a new clustering heuristic was introduced by [28]. 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 Start-Cycle (SC) and Completion-Cycle (CC) heuristics, it is not meant to operate across a wide range of inter-cluster delays. This heuristic quite often defaults to the Start-Cycle, which is why its performance is also unbounded as the inter-cluster delay increases. In our evaluation we name this heuristic as Critical-Successor (CS).

Finally there are several combined loop-scheduling and clustering algorithms [3, 6, 27]. These are based on the software-pipeline scheduling technique of modulo-scheduling. These techniques are

only applicable on innermost loops under very specific and strict conditions.

7.2 Clustering on a separate pass

Pioneering work on code generation for clustered architectures was introduced in [8], with the Bottom-Up-Greedy (BUG) cluster-assignment algorithm. 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 is more conservative than the Start-Cycle, since it will select a distant cluster only if the instruction's consumers can still get their input data in time.

[5] partitions the register file so as to have more register files with fewer ports each. Cluster assignment 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 inter-cluster latencies can not be hidden effectively. 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.

[7] is one of the first iterative solutions to clustering. Each iteration of the algorithm measures the schedule length by performing instruction scheduling and doing a fast register pressure and ICC count estimation. This being an iterative algorithm, it has a long run-time and its use is not practical in compilers.

7.3 Clustered Architectures

A comprehensive taxonomy of inter-cluster communication implementations on VLIW architectures is presented in [26]. The design features (such as operating frequency, performance, energy consumption, etc.) of each implementation are quantified and discussed.

Clustered super-scalars, such as [22],[15], use simpler clustering algorithms. A review of the state-of-the-art heuristics are presented in [4]. 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.

7.4 Instruction Scheduling for VLIW processors

Instruction Scheduling for VLIWs was pioneered by [10] with the Trace-scheduling algorithm. This algorithm expands the scheduling region beyond basic blocks to larger profiling-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 [13]. 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 [18]. Extended Basic Blocks (EBB) [20] form tree-like regions which are then scheduled by a normal list scheduler. Treegions [12] are also tree-shaped, and are similar to EBBs. They are shown to outperform superblock scheduling. LUCAS is implemented on top of GCC's [1] Haifa Scheduler which operates on EBBs.

8. CONCLUSION

This paper proposes LUCAS, a new unified cluster assignment and instruction scheduling algorithm for clustered VLIW processors, that is powered by a novel hybrid clustering heuristic. LUCAS outperforms the state-of-the-art as it is capable of switching between two heuristics at a very fine granularity. The switching is controlled by two metrics, the cycle congestion and the instruction mobility. The end result is a scheduler that generates code that performs best across a wide range of inter-cluster latencies.

References

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

[2] ski ia64 simulator. *http://ski.sourceforge.net*.

[3] A. Aletà, J. Codina, J. Sánchez, A. González, and D. Kaeli. Agamos: A graph-based approach to modulo scheduling for clustered microarchitectures. *IEEE Transactions on Computers*, 2009.

[4] R. Canal, J. M. Parcerisa, A. Gonzlez, D. D. D. Computadors, and J. Girona. Dynamic cluster assignment mechanisms. In *HPCA*, 2000.

[5] A. Capitanio, N. Dutt, and A. Nicolau. Partitioned register files for vliws: A preliminary analysis of tradeoffs. In *MICRO*, 1992.

[6] J. Codina, J. Sanchez, and A. Gonzalez. A unified modulo scheduling and register allocation technique for clustered processors. In *PACT* 2001.

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

[8] J. Ellis. Bulldog: A compiler for vliw architectures. Technical report, Yale Univ., 1985.

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

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

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

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

[13] W.-M. 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*, 1993.

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

[15] R. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 1999.

[16] V. Lapinskii, M. Jacome, and G. De Veciana. Cluster assignment for high-performance embedded vliw processors. *ACM TODAES*, 2002.

[17] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. Odonnell, and J. C. Ruttenberg. The multiflow trace scheduling compiler. *Journal of Supercomputing*, 1993.

[18] 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 *MICRO*, 1992.

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

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

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

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

[23] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. Keckler, and C. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *ISCA*, 2003.

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

[25] M. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J. Lee, W. Lee, et al. The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. In *IEEE Micro*, 2002.

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

[27] J. Zalamea, J. Llosa, E. Ayguade, and M. Valero. Modulo scheduling with integrated register spilling for clustered vliw architectures. In *MICRO*, 2001.

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