UCIFF: Unified Cluster assignment Instruction scheduling and Fast Frequency selection for heterogeneous clustered VLIW cores *

Vasileios Porpodas and Marcelo Cintra**

School of Informatics, University of Edinburgh
{v.porpodas@, mc@staffmail.}ed.ac.uk

Abstract. Clustered VLIW processors are scalable wide-issue statically scheduled processors. Their design is based on physically partitioning the otherwise shared hardware resources, a design which leads to both high performance and low energy consumption. In traditional clustered VLIW processors, all clusters operate at the same frequency. Heterogeneous clustered VLIW processors however, support dynamic voltage and frequency scaling (DVFS) independently per cluster. Effectively controlling DVFS, to selectively decrease the frequency of clusters with a lot of slack in their schedule, can lead to significant energy savings.
In this paper we propose UCIFF, a new scheduling algorithm for heterogeneous clustered VLIW processors with software DVFS control, that performs cluster assignment, instruction scheduling and fast frequency selection simultaneously, all in a single compiler pass. The proposed algorithm solves the phase ordering problem between frequency selection and scheduling, present in existing algorithms. We compared the quality of the generated code, using both performance and energy-related metrics, against that of the current state-of-the-art and an optimal scheduler. The results show that UCIFF produces better code than the state-of-the-art, very close to the optimal across the mediabench2 benchmarks, while keeping the algorithmic complexity low.

Keywords: clustered VLIW, heterogeneous, DVFS, scheduling, phase-ordering

1 Introduction

Energy consumption has become an important design constraint for microprocessors. Clustered VLIW processors were introduced with performance and energy scalability in mind: **i.** They are statically scheduled, which removes the instruction scheduling burden from the micro-architecture. **ii.** The clustered design improves energy efficiency, operating frequency and reduces design complexity[17]. Clustered VLIW processors operate at an attractive power/performance ratio point. Examples are the Texas Instrument's VelociTI, HP/ST's Lx [6], Analog's TigerSHARC [7], and BOPS' ManArray [15].

A clustered processor has its shared non-scalable resources (such as the register file which is shared among functional units) partitioned into smaller parts. Each part of the partitioned resource, along with some of the resources that

* This work was supported in part by the EC under grant ERA 249059 (FP7).
** Marcelo Cintra is currently on sabbatical leave at Intel Labs.

communicate with it are grouped together into a cluster. For example a cluster often contains a slice of the register file along with several functional units. Within a cluster signals travel fast, faster than in the shared resource case, and energy consumption remains low, due to the improved locality. Between clusters, communication is subject to an inter-cluster delay and there is additional energy consumption on the inter-cluster interconnect.

Traditionally, all clusters of a clustered VLIW processor operate at the same frequency and voltage. Considerable energy savings can be achieved by freeing each cluster to operate at its own frequency and voltage level. The reason for this is that the cluster utilization usually varies; some clusters are fully loaded while others have a fraction of the load. It is therefore sensible to lower the frequency of the under-utilized clusters to save energy.

In this paper we raise an important issue of the existing compilation techniques for heterogeneous clustered VLIW processors. Compiling for these architectures comprises of solving two distinct but highly dependent sub-problems:

1. Selecting the frequency that each cluster should operate at.
2. Performing cluster assignment and instruction scheduling for the selected frequencies (we refer to both as "scheduling" for simplicity).

There is a **phase-ordering** issue between these two sub-problems: **A.** One cannot properly select the frequencies per cluster without scheduling and evaluating the schedule. **B.** One cannot perform scheduling without having decided on the frequencies.

State-of-the-art work in this field ([2]) treats these two sub-problems independently and solves the first (1.) before the second (2.). At first a good set of frequencies is found by estimating the scheduling outcome for each configuration (without actually scheduling). Then scheduling is performed for this set of frequencies. We will refer to this approach as the "Decoupled" one.

The problem is that the frequency decision (1.) has a great impact on the quality of scheduling (2.). We observed that the estimation of the scheduling outcome without performing the actual scheduling, as done in [2], can be inaccurate. Nevertheless, it is a critical compilation decision since selecting a non-optimal frequency set can lead to a schedule with poor performance, energy consumption or both.

In this work we provide a more concrete solution to the problem by solving both sub-problems (frequency selection and scheduling) in a single algorithm thus alleviating the phase-ordering issue altogether. The proposed scheduling algorithm for heterogeneous clustered VLIW processors performs cluster assignment, instruction scheduling and fast frequency selection, all in a unified algorithm, as a unified scheduling pass.

The algorithm can be configured to generate optimized code for any of the commonly used metrics (Energy, $Energy \times Delay$ Product (EDP) and $Energy \times Delay^2$ (ED2), Delay). The output of the algorithm is twofold: **i.** The operating frequency of each cluster such that the scheduling metric is optimized. **ii.** Fully clustered and scheduled code for the frequencies selected by (i).

In the text that follows we use the terms "frequencies per cluster", "set of frequencies" and "frequency configuration" interchangeably.

2 Motivation

2.1 Homogeneous vs Heterogeneous

This section motivates the heterogeneous clustered VLIW design by demonstrating how energy can be saved without sacrificing performance in the example of Fig.1.

Fig.1a is the Data Flow Graph (DFG) to be scheduled. Fig.1b,c show the instruction schedules that correspond to this DFG on a two-clustered machine (single-issue per-cluster). Fig.1b is the homogeneous design with both clusters operating at the same frequency (f), while Fig.1c is the heterogeneous one with ClusterB operating at half the frequency of ClusterA ($f/2$). Nevertheless both configurations have the **same performance** as the schedule length is 4 cycles for both. The heterogeneous can perform as well as the homogeneous because ClusterB was initially under-utilized (there was slack in part of the schedule).

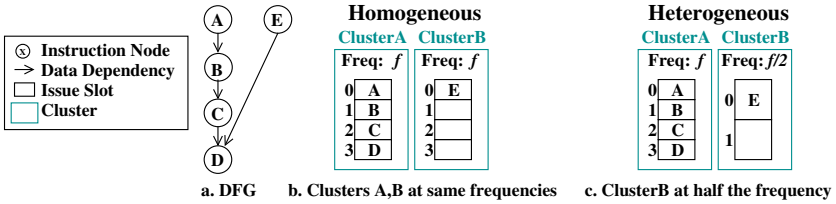

Fig. 1. Under-utilized ClusterB can have half the frequency with no performance loss.

Since the target architecture is a statically scheduled clustered VLIW one, it is the job of the scheduler to find the best frequency for each cluster so that the desired metric (Energy, EDP, ED2, Delay) is optimized.

2.2 Phase Ordering

As already discussed in Section 1, there is a phase ordering issue between frequency selection and instruction scheduling. Fig.2 shows a high-level view of the scheduling algorithms for a 2-cluster processor with 3 possible frequencies per cluster (f_0, f_1, f_2).

The Decoupled algorithm (existing state-of-the-art based on [2]) is in Fig.2a. As already mentioned, there are two distinct steps:

1. The first step selects one of the many frequency configurations as the one that should be the best for the given metric (e.g. EDP). This is based on a simple estimation (before scheduling) of the schedule time ($cycles \times T$) and energy consumption that the code will have after scheduling. The exact calculations are described in detail in Section 5:*Decoupled*.
2. The second step performs scheduling on the architecture configuration selected by step 1. This includes both cluster assignment and instruction scheduling, which in an unmodified [2] are in two separate steps.

It is obvious that if step (1) makes a wrong decision (which is very likely since the decision is based on a simple estimate), then the processor will operate at a point far from the optimal one. Therefore step (2) will schedule the code for a non-optimal frequency configuration which will lead to a non-optimal result.

This phase-ordering issue is dealt with by UCIFF, the proposed unified frequency selection and scheduling algorithm (Fig.2b). The proposed algorithm solves the two sub-problems simultaneously and outputs a combined solution which is both the frequency configuration (that is the frequency for each cluster) and the scheduled code for this specific configuration.

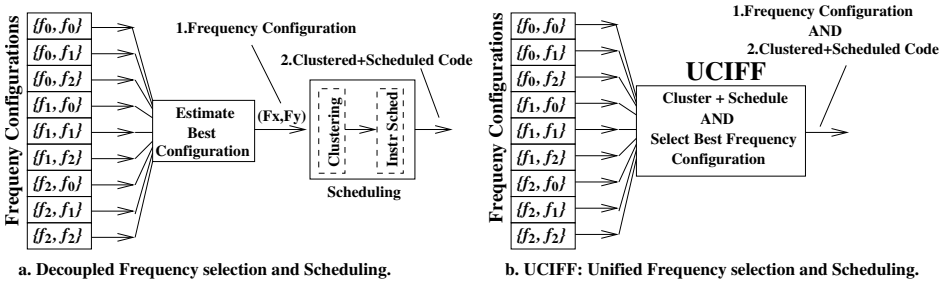

Fig. 2. The two-phase scheduling of the current state-of-the-art (a). The proposed unified approach (b) is free of this phase-ordering problem.

3 UCIFF

The proposed Unified algorithm for Cluster assignment, Instruction scheduling and Fast Frequency selection (UCIFF) can be more easily explained if two of its main ingredients are explained separately. That is: **i.** scheduling for a fixed heterogeneous processor and **ii.** unifying scheduling and frequency selection.

3.1 Scheduling for fixed heterogeneous processors

An out-of-the-box scheduler for a clustered architecture can only handle the homogeneous case, where all clusters operate at the same frequency. A heterogeneous architecture on the other hand, has different frequencies across clusters. This is because schedulers work in a cycle-by-cycle manner. They schedule ready instructions on Free cluster resources and move to the next cycle. This cycle-by-cycle operation is inapplicable when clusters operate at different frequencies. The problem gets worse if cluster frequencies are not integer multiples of one another (e.g. cluster 0 operating at frequency f and cluster 1 at $1.5f$).

UCIFF introduces a scheduling methodology for heterogeneous clustered architectures with arbitrary frequencies per cluster which can be applied to existing scheduling algorithms. The idea is that the scheduler operates at a higher base frequency (f_{sched}) such that the clock period of any cluster is an integer multiple of the clock period of the scheduler (T_{sched}). It works in two steps:

i. The scheduler's base frequency f_{sched} is calculated as the lowest integer common multiple of all possible frequencies of all clusters. The scheduler internally works at a cycle $T_{sched} = 1/f_{sched}$, which is always an integer multiple of the cycle that each cluster operates at. For example in Fig.3b the scheduler's base cycle is T_{sched} while the cycles of cluster0 and cluster1 are $3 \times T_{sched}$ and $2 \times T_{sched}$ respectively.

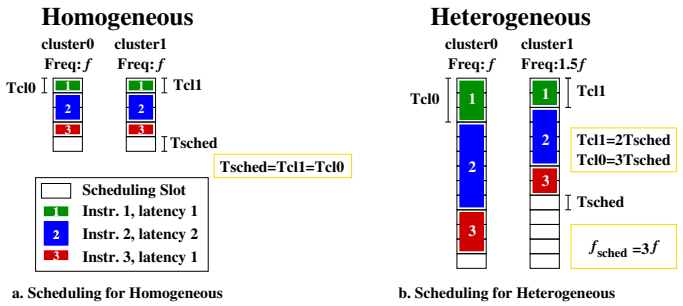

Fig. 3. The scheduler's internal clock period T_{sched} compared to the periods of the two clusters T_{cl0} and T_{cl1}, for a homogeneous (a) and a heterogeneous (b) architecture.

ii. The instruction latencies for each cluster are increased and set to be a multiple of the original one, equal to $(T_{cluster}/T_{sched}) \times OrigLatency$. In the example of Fig.3, the instruction latencies for cluster0 are multiplied by 3 while the ones for cluster1 are multiplied by 2.

In this way the problem of scheduling for different frequencies per cluster is transformed to the problem of scheduling instructions of various latencies, which is a solved problem and is indeed supported by any decent scheduler.

3.2 Scheduling for non-fixed heterogeneous processors (UCIFF)

In contrast to the existing state-of-the-art, UCIFF solves the phase-ordering problem between frequency selection and scheduling. It does so by combining them into a single unified algorithm. In addition, the scheduling algorithm performs cluster assignment and instruction scheduling together thus removing any phase ordering issues between all clustering, scheduling and frequency selection.

The UCIFF algorithm is composed of three nested layers: The driver function (Alg.1) at the outermost layer, the clustering and scheduling function (Alg.2) at the second layer and the metric calculation function (Alg.3) at the innermost.

1. The driver: The highest level of the UCIFF algorithm (Alg.1) performs the frequency selection. It decides on a single frequency configuration for the whole scheduling region. Instead of solving the global optimization problem, of determining the optimal frequency, with a full-search over all configurations, UCIFF uses a fast hill climbing approach.

Hill climbing, in general, searches for a globally good solution by evaluating, at each point, its neighbors and by "moving" towards the best among them. Due to the nature of the problem, trying out a large number of neighbors is computationally expensive. This is because we cannot evaluate a configuration at cycle c unless we schedule all instructions up to c. This makes probabilistic algorithms (such as simulated annealing) very expensive since trying out random configurations will lead to almost the whole configuration space being scheduled to a very large extent, thus leading to a time complexity comparable to that of the full-search.

Formally, a **frequency configuration** is an ordered multiset of each cluster's frequency: $\{f_a, f_b, f_c, ...\}$. Each of $f_a, f_b, f_c, ...$ is one of the l valid frequency levels in the set $\{f_0, f_1, ..., f_{l-1}\}$. For example a valid configuration for a 2-cluster

machine with 3 possible frequency levels (f_0, f_1, f_2) is $\{f_2, f_0\}$ (where clusters 0,1 operate at f_2, f_0 frequencies respectively).

The **neighbors** of a configuration c are the configurations which are close frequency-wise to c. More precisely, the configuration $\{f_{na}, f_{nb}, f_{nc}, ...\}$ is a UCIFF neighbor of $\{f_a, f_b, f_c, ...\}$ if $nx = x$ for all x except one (say y) such that $|ny - y| < NDistance$. For example, the neighbors of $\{f_1, f_1\}$ for $NDistance = 1$ are $\{f_0, f_1\}$, $\{f_2, f_1\}$, $\{f_1, f_0\}$ and $\{f_1, f_2\}$.

In UCIFF the hill climbing search is done gradually, in steps of cycles, while the code gets scheduled for the duration of the step. After each step there is an evaluation. We refer to this step-evaluation-step approach as "gradual hill climbing" and to the act of scheduling within a step as "partial scheduling". This makes UCIFF fast and accurate. The hill climbing search stops when all instructions of the best neighbors have been scheduled. All of the above will be further explained through the following example.

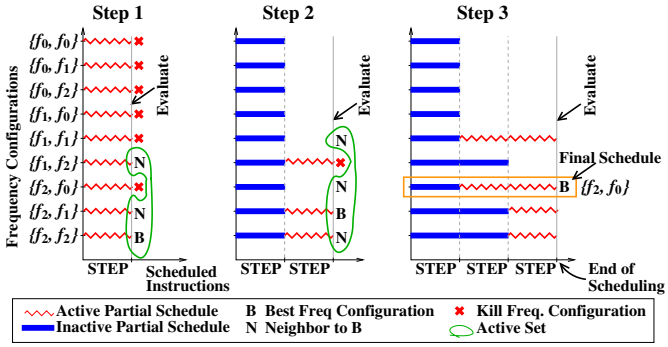

Fig. 4. Overview of the UCIFF gradual hill climbing algorithm for a schedule that consists of three steps.

A high level example of the UCIFF algorithm for the 2-cluster machine of Fig.2 is illustrated in Fig.4. On the vertical axis there are all 9 possible frequency configurations. The horizontal axis represents the scheduler's cycles (of T_{sched} duration). The partial schedule of each configuration is a horizontal line that starts from the vertical axis at the configuration point and grows to the right. The evaluation (every STEP instructions) is represented by the vertical gray line.

At first (**Step 1**) all configurations are partially scheduled for "STEP" instructions. Once partially scheduled, they are evaluated and the best configuration is found and marked as "B". At this point the neighbors of "B" are found, according to the definition given earlier. The neighbors are marked as "N". The neighbors ("N") along with the best ("B") form the active set. The configurations not in the active set are marked with a red "X".

In **Step2** the configurations in the active set get partially scheduled for another "STEP" instructions (curly red lines). They get evaluated and the best one ("B") and its neighbors ("N") are found.

In **Step3** the active set of Step2 gets partially scheduled for another "STEP" instructions. At this point it is interesting to note that $\{f_2, f_0\}$ and $\{f_1, f_1\}$ have to be scheduled for both the 2nd and 3rd "STEP". Now there are no instructions

left to schedule for the active configurations, therefore the algorithm terminates. After the final evaluation, the best configuration of the active set is found ("B", $\{f_2, f_0\}$). The **full schedule** for this configuration is **returned** (gold rectangle).

Note that the bar lengths are not proportional to any metric value. They just show the progress of the algorithm while instructions get scheduled.

The detailed algorithm is listed in Alg.1. The algorithm initially performs partial scheduling of all frequency configurations for "STEP" instructions (Alg.1 lines 11, 14-20). This determines the best configuration and stores it into "BFC". For the rest of the algorithm, each frequency configuration in the neighboring set of "BFC" (lines 13,14) gets partially scheduled for "STEP" instructions and evaluated (lines 15,16). The best performing of the neighbors gets stored into "BFC" (line 19). The algorithm repeats until no instructions in the neighboring set of "BFC" (a.k.a. active set) are left unscheduled (line 20). Each iteration of the algorithm decreases "STEP" by "STEPVAR" (line 18) so that re-evaluation of the schedules keeps getting more frequent. This makes the algorithm track the best configuration faster.

This gradual hill-climbing process accurately selects a good configuration among many without resorting to a full-search across all frequency configurations. The end result is a fully scheduled code for the selected configuration.

It is interesting to note that partial scheduling of all neighbors could be done in parallel. This could speed up the UCIFF scheduler, to reach speeds close to those of the Oracle.

Algorithm 1. UCIFF

```
1  /* Unified Cluster assignment Instr. Scheduling and Fast Frequency selection.
2     In1: METRIC_TYPE that the scheduler should optimize for.
3     In2: Schedule STEP instructions before evaluating and getting the best.
4     In3: STEPVAR: Decrement STEP by STEPVAR upon each evaluation.
5     In4: NEIGHBORS: The number of neighbors per cluster.
6     Out: Scheduled Code and Best Frequency Configuration. */
7  uciff (METRIC_TYPE, STEP, STEPVAR, NEIGHBORS)
8  {
9      do
10         if (BFC not set)   /* If first run */
11            NEIGHBORS_SET = all frequency configurations
12         else
13            NEIGHBORS_SET = neighbors of BFC /*up to NEIGHBORS per cluster*/
14         for FCONF in NEIGHBORS_SET
15            /* Partially schedule the ready instructions of FCONF frequency
                  ↪configuration for STEP instructions, optimizing METRIC_TYPE
                  ↪*/
16            SCORE = cluster_and_schedule (METRIC_TYPE, STEP, FCONF)
17            Store the scheduler's calculated SCORE into  SCORECARD [FCONF]
18         Decrement STEP by STEPVAR until 1. /* Variable steps (optional) */
19         BFC = Best Freq Configuration of SCORECARD, clear SCORECARD
20     while there are unscheduled instructions in active set
21     return BFC and scheduled code of BFC
22 }
```

2. The Core: At one level lower lies the core of the scheduling algorithm (Alg.2). It is a unified **cluster assignment and scheduling algorithm** which shares some similarities with UAS [14] but has several unique attributes: **i.** It operates on a heterogeneous architecture where clusters operate at different frequencies (as described in Section 3.1). **ii.** It only issues an instruction to the cluster chosen by the heuristic. It does not try to issue on any other cluster if it cannot currently issue on the chosen cluster. **iii.** It is capable of performing partial scheduling for "STEP" number of instructions. **iv.** It can optimize for various metrics (not just Delay). This includes energy related ones: Energy, EDP, ED2.

v. The start_cycle calculation is extended to work for heterogeneous clusters, which is done by adding to it the latency of the instruction on that cluster (see Alg.3 line 10).

In more detail, the algorithm is a list-scheduling based one, that operates on a ready list. The scheduler performs partial scheduling on each active frequency configuration for a small window of "STEP" instructions. Once a (configuration, cycle) pair is scheduled it is never revisited. Switching among configurations requires that the scheduler maintains a private instance of its data structures (ready list, reservation table, current cycle) for each configuration. To that end, it saves and restores the snapshot of its structures upon entry and exit (Alg.2 lines 7-11, 31). The ready list gets filled in with ready and deferred instructions (line 13). Then it gets sorted based on priority (calculated on the Data Dependence Graph) (line 14) and the highest priority one is selected for scheduling (line 16). A list of candidate clusters is created (line 17) and the best cluster is found based on the values of the metric used for scheduling (line 18). The instruction is then tried on the best cluster at the current cycle (lines 19,20). If successful, then its presence in the schedule is marked on the reservation table for as many cycles as its latency as specified by LATENCY [ ] array (line 21), the IPCL (Instructions Per CLuster) counts the issued instruction (line 22), and INSN gets removed from the ready list (line 23). If unsuccessful, INSN's execution is deferred to next cycle (lines 24-26). We move to the next cycle only if the current ready list is empty (lines 27-28).

Algorithm 2. Clustering and Scheduling for various metrics.

```
1  /* In1: METRIC_TYPE: The metric type that the scheduler will optimize for.
2     In2: STEP: Num of instrs to schedule before switching to next freq. conf.
3     In3: FCONF: The architecture's current frequency configuration.
4     Out: Scheduled Code and metric value. */
5  cluster_and_schedule (METRIC_TYPE, STEP, FCONF)
6  {
7     /* Restore ready list for this frequency configuration */
8     READY_LIST = READY_LIST_ARRAY [FCONF]
9     /* Restore current cycle. CYCLE is the scheduler's internal cycle. */
10    CYCLE = LAST_CYCLE [FCONF]
11    Restore the Reservation Table state that corresponds to FCONF
12    while (instructions left to schedule && STEP > 0)
13       update READY_LIST with ready to issue at CYCLE, include deferred
14       sort READY_LIST based on list-scheduling priorities
15       while (READY_LIST not empty)
16          select INSN, the highest priority instruction from the READY_LIST
17          create LIST_OF_CLUSTERS[] that INSN can be scheduled at on CYCLE
18          BEST_CLUSTER=best of LIST_OF_CLUSTERS[] by comparing for each cluster
                ↪ calculate_heuristic(METRIC_TYPE,CLUSTER,FCONF,INSN,IPCL[])
19          /* Try scheduling INSN on the best cluster */
20          if (INSN can be scheduled on BEST_CLUSTER at CYCLE)
21              schedule INSN, occupy LATENCY[FCONF][BEST_CLUSTER][INSN] slots
22              IPCL [CLUSTER] ++  /* count number of instructions per cluster */
23              remove INSN from READY_LIST
24          /* If failed to schedule INSN on best cluster, defer to next cycle */
25          if (INSN unscheduled)
26              remove INSN from READY_LIST and re-insert it at CYCLE + 1
27       /* No instructions left in ready list for CYCLE, then CYCLE ++ */
28       CYCLE ++
29       /* If we have scheduled STEP instructions, finalize and exit */
30       if (instr. scheduled > STEP instructions)
31           Update READY_LIST_ARRAY[], LAST_CYCLE[] and Reservation Table
32           return metric value of current schedule
33 }
```

3. The metrics: The combined clustering and scheduling algorithm used in UCIFF is a modular one. It can optimize the code not only for cycle count, but

also for several other metrics that are useful in the context of a heterogeneous clustered VLIW. It supports energy-related metrics (Energy, EDP, ED2) and also execution Delay (Alg.3). The metric type controls the clustering heuristic which decides on the BEST_CLUSTER in Alg.2 line 18.

The energy-related metrics require that the scheduler have an **energy model** of the resources. The energy model is a small module in the scheduling algorithm and it is largely decoupled from the structure of the algorithm. The energy is calculated as the sum of the static and dynamic energy consumed by the clusters and the inter-cluster communication network. Static energy consumption is relative to the time period that the system is "on". Each instruction that executes on a cluster consumes dynamic energy relative to its latency. Each inter-cluster communication consumes dynamic energy as much as an instruction of the fastest cluster. The exact formulas for these calculations are in Table 1.

$E = \sum_{clusters}[E_{st}(cl) + E_{dyn}(cl)]$	
$E_{st}(cl) = P_{st} \times cycles_{cl} \times T_{cl}$	$E_{dyn}(cl) = E_{dyn,ins}(cl) + E_{dyn,icc}$
$P_{st}(cl) = C_{st} \times V_{cl}$	$E_{dyn,ins}(cl) = \sum_{ins}[P_{ins}(cl) \times Latency(ins, cl)]$
	$P_{ins}(cl) = C_{dyn} \times f_{cl} \times V_{cl}^2$
	$E_{dyn,icc} = P_{icc} \times NumICCs$
	$P_{icc} = C_{dyn} \times f_{fastest} \times V_{fastest}^2$

Table 1. Formulas for energy calculation.

Algorithm 3. Heuristic calculation.

```
1  /* In1: METRIC_TYPE: The metric type that the scheduler will optimize for.
2     In2: CLUSTER: The cluster that INSN will be tested on.
3     In3: FCONF: The architecture's current frequency configuration.
4     In4: INSN: The instruction currently under consideration.
5     In5: IPCL: The Instruction count Per CLuster (for dyn energy).
6     Out: metric value of METRIC_TYPE if INSN scheduled on CLUSTER under FCONF*/
7  calculate_heuristic (METRIC_TYPE, CLUSTER, FCONF, INSN, IPCL[])
8  {
9      START_CYCLE = earliest cycle INSN can be scheduled at on CLUSTER
10     UCIFF_SC = START_CYCLE + LATENCY[FCONF][CLUSTER][INSN]
11     switch (METRIC_TYPE)
12         case ENERGY: return energy (CLUSTER, FCONF, UCIFF_SC, IPCL[])
13         case EDP: return edp (CLUSTER, FCONF, UCIFF_SC, IPCL[])
14         case ED2: return ed2 (CLUSTER, FCONF, UCIFF_SC, IPCL[])
15         case DELAY: return UCIFF_SC
16 }
```

3.3 DVFS region

UCIFF determines the best frequency configuration at a per-scheduling-region basis. This is the natural granularity for a scheduling algorithm. This however is not the right granularity for Dynamic Voltage and Frequency Scaling (DVFS), which usually takes longer time. Therefore UCIFF's decisions on the frequency and voltage levels occur more frequently than what a real DVFS system could follow. As a result, UCIFF's per-region decisions have to be coarsened by some mapping from multiple UCIFF decisions to a single DVFS decision.

There are both hardware and software solutions to this. A possible micro-architectural solution involves pushing UCIFF's decision into a FIFO queue. Once the queue is full, a DVFS decision is made based on the average of the items in the queue, and the queue gets flushed.

A software solution is to perform sampling on the UCIFF configurations at a rate at most as high as the one supported by the system. Another way is to

come up with a single DVFS point for the whole program by calculating the weighted average of the region points generated by UCIFF. A more accurate solution could be based on the control-edge probabilities. This knowledge can be acquired by profiling and can be used to form super-regions which operate at a single DVFS point.

The mapping decision for the DVFS points is completely decoupled from the UCIFF algorithm. A thorough evaluation of the possible solutions is not in the scope of this paper.

4 Experimental Setup

The target **architecture** is an IA64 (Itanium) [16] based statically scheduled clustered VLIW architecture. The architecture has 4 clusters and an issue width of 4 in total (that is 1 per cluster), similar to [2]. Each cluster's cycle time is 4, 5, 6 or 7 times a reference base cycle. Therefore the ratio of the fastest frequency to the slowest one is 7:4.

We have implemented UCIFF in the scheduling pass (haifa-sched) of GCC-4.5.0 [1] cross **compiler** for IA64.

Our experimental setup has some of its aspects deliberately idealized so that the generated code quality is isolated from external noise. **i.** Each cluster has all possible types of resource units available for all its issue slots. This alleviates any instruction bundling issues (which exist in the IA64 instruction set). **ii.** No noise from register allocation / register spills. Although the scheduler runs twice (before and after register allocation) our measurements are taken before register allocation. At this stage the compiler still considers an infinite register file. This is not far from reality though, as clustered machines have abundant register resources (each cluster has a whole register file for its own use).

We evaluated UCIFF on 6 of Mediabench II video [8] **benchmarks**. All benchmarks were compiled with optimizations enabled (-O flag).

5 Results

We evaluate UCIFF by comparing it against the Decoupled, the Oracle and the Full-Search algorithms.

The **Decoupled** scheduler is the state-of-the-art acyclic scheduler for heterogeneous clustered VLIW processors (based on the cyclic scheduler of [2]). It decouples frequency selection from instruction scheduling. The frequency selection step is done via a simple estimation of the energy consumption and the execution (schedule) time. The estimation was done as in [2]:

The schedule time is equal to the cycle count of a profiled homogeneous architecture ($cycles_{hom}$) multiplied by the arithmetic mean of the clock periods of the heterogeneous clusters: $Time = cycles_{hom} \times (\sum_{cl} T_{cl})/NumOfClusters$. The cycle count of each cluster is easily calculated as: $cycles_{cl} = Time/T_{cl}$.

The energy calculation is similar to that of UCIFF (Table 1) with two main differences:

1. The dynamic energy of a cluster is equal to a fraction of that of a homogeneous cluster, proportional to the ratio of f_{cl} to the average frequency: $E_{dyn,ins}(cl) = E_{dyn,ins\_hom}(cl) \times f_{cl}/[\sum_{cl}(f_{cl})/NumOfClusters]$

2. The energy of the interconnect is equal to that of the homogeneous:
   $E_{dyn,icc} = P_{icc} \times NumICCs_{homogeneous}$

The **Oracle** scheduler is a decoupled scheduler with a perfect frequency selection phase. The frequency configuration selected will always produce the best schedule with 100% accuracy. This scheduler is the upper bound (optimal) in code quality (Fig.6) and the lowest bound (optimal) in the scheduler run-time (Fig.7). It is non-implementable as it requires future knowledge.

A **Full-Search** UCIFF-based scheduler does not perform any kind of pruning on the frequency space. It is structured as UCIFF, but instead of a hill climbing search, it does a full search over the frequency configurations. This makes it the slowest (Fig.7), but in the meantime it always achieves the optimal code quality, same as that of the Oracle (Fig.6).

Although in the vanilla Decoupled ([2]) clustering and scheduling are in separate steps, in all implementations of the above algorithms, scheduling includes both cluster assignment and instruction scheduling in a unified pass as discussed in Section 3.2.Core and Alg.2. This lets us focus only on the phase ordering problem we are interested in: the one between frequency selection and scheduling.

The high-level features of these algorithms are summarized in Table 2.

	Decoupled-based		*UCIFF-based*	
	Decoupled	Oracle	Full-Search	UCIFF
Phase-ordering problem	Yes	No	No	No
Code quality	Low	High	High	High
Algorithmic complexity	Low	Low	High	Medium
Realistic (implementable)	Yes	No	Yes	Yes

Table 2. Some features of the algorithms under comparison.

Since UCIFF unifies two otherwise distinct phases (frequency selection and scheduling), we show some results (Section 5.1) that quantify the first phase separately. This provides vital insights as to why the unified solution performs better.

5.1 Accuracy of Frequency Selection

The outcome of the Decoupled algorithm relies heavily on the accuracy of the frequency selection phase. The stand-alone frequency selection step makes its decision based on estimations of the energy consumption and the scheduled code's schedule length as in [2]. The estimations are based on the energy and cycle numbers of a homogeneous architecture and on the ratio of the clock cycle of each cluster of the heterogeneous against that of the homogeneous.

On the other hand UCIFF is not based on estimation, but rather on real partial scheduling results. Its frequency decision is therefore much more informed.

UCIFF's frequency selection superiority over the Decoupled algorithm is shown in Fig.5. The horizontal axis shows the error margins in the scheduling outcome when compared to that of the Oracle. For example, a 5% error margin includes the frequency selections that generate results at most 5% worse than that of the Oracle. The vertical axis shows the percentage of frequency selections that have the error margin shown in the horizontal axis. The Decoupled accuracy fluctuates significantly for various metrics; In ED2 it is about 5 times less accurate than in EDP. UCIFF, on the other hand, is constantly very accurate with the fluctuations being less than 10% over all error margins.

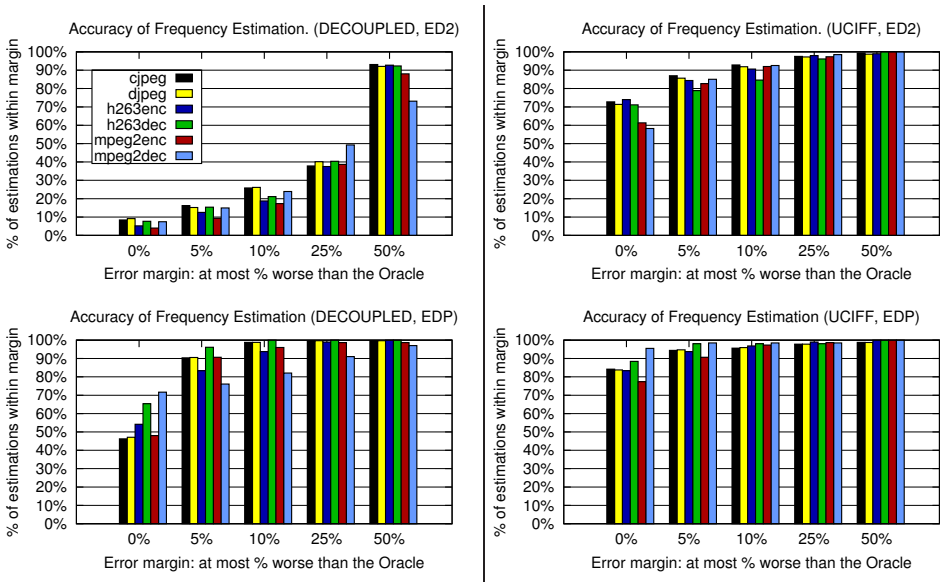

Fig. 5. The Accuracy of the Frequency Selection (Y axis) within the range from the oracle (X axis) for Decoupled (Left) and UCIFF (Right). UCIFF is tuned with STEP=8, STEPVAR=2 and NEIGHBORS=4

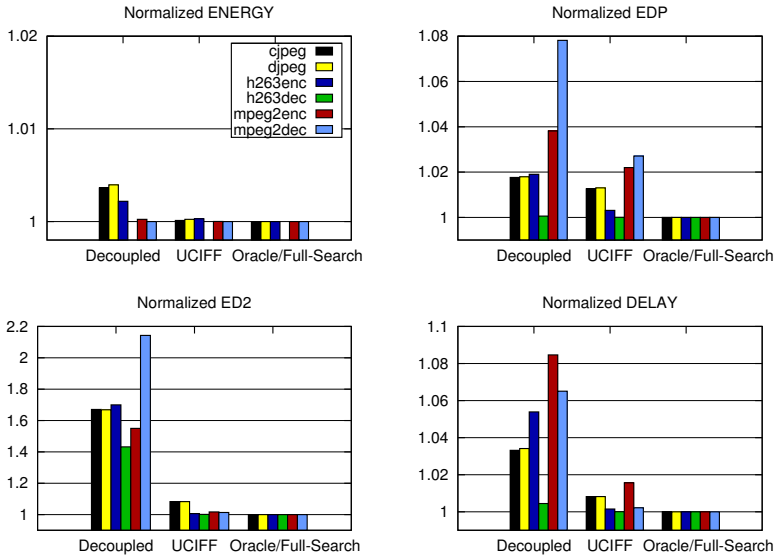

Fig. 6. Code quality (Energy, EDP, ED2, Delay) for Decoupled, UCIFF and Oracle/Full-Search(FS), over the Mediabench2 benchmarks[2] UCIFF is tuned with STEP=8, STEPVAR=2 and NEIGHBORS=4

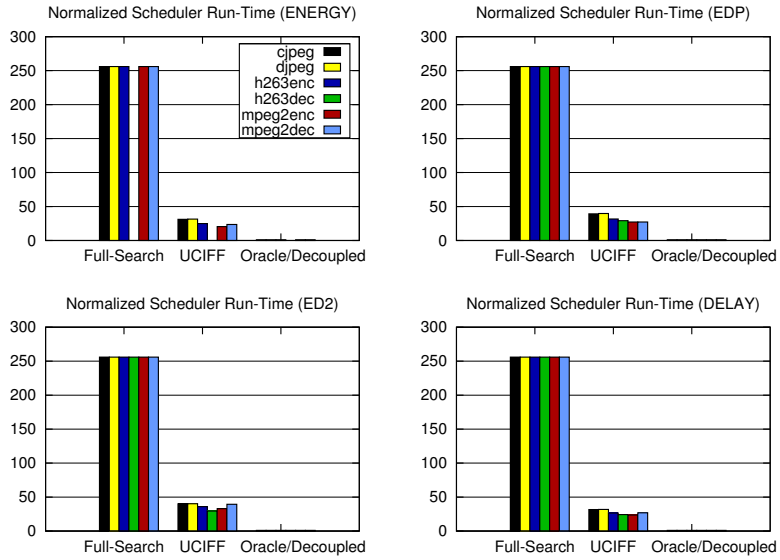

Fig. 7. The scheduler's Run-Time in terms of scheduling actions for Energy, EDP, ED2 and Delay, over the Mediabench2 benchmarks[2] normalized to the Oracle/Decoupled. UCIFF is tuned with STEP=8, STEPVAR=2 and NEIGHBORS=4

5.2 UCIFF code quality vs algorithmic complexity

The quality of the code generated by each scheduling algorithm when optimizing for various metrics (Energy, EDP, ED2 and Delay) is shown in Fig.6.

We provide an estimate of the algorithmic complexity by measuring the execution time of each algorithm based on the count of "scheduling actions" each algorithm performs. By "scheduling action" we refer to the action of scheduling an instruction at a specific point. This is an accurate estimate of the time complexity since all algorithms share the same scheduling core. The results are shown in Fig.7.

UCIFF achieves a code quality close to that of the Oracle and the Full-Search, but with a much lower run-time than the Full-Search (Fig.7). This is because UCIFF performs a smart pruning of the frequency configuration space.

The ED2 metric is the hardest to predict at the frequency selection step. This is obvious from the code-quality results of Fig.6. It is there that the estimation of the Decoupled algorithm proves not accurate enough, being 2.15× worse than the Oracle in the worst case. UCIFF, on the other hand, is constantly more accurate than the Decoupled and very close to the Oracle.

UCIFF can be **tuned** to operate at various points in the trade-off space of code quality versus scheduling time complexity. It can get closer or even match Oracle's performance by searching more frequency configurations. There are three knobs that we can configure. In decreasing order of importance they are: NEIGHBORS, STEP and STEPVAR (see Alg.1). The NEIGHBORS variable controls the number of neighboring configurations in the neighboring set. A NEIGHBORS value of 4 means that at most 4 neighbors per cluster are in the

[2] h263dec energy results are missing due to failure in compilation

neighboring set (that is equivalent to $NDistance = 2$ of Section 3.2). The higher its value, the more accurate the result but the longer it takes for the scheduler to run. The STEP controls the cycle distance before evaluating and re-selecting the neighbors. For very small regions STEP should be as high as the size of the region, to allow for a full-search over it. A high value of STEP however makes the algorithm less adaptive to changes. This is the job of STEPVAR. It decreases STEP by STEPVAR until STEP reaches 1. The results shown were taken with NEIGHBORS=4, STEP=8, STEPVAR=2. A full investigation of optimally selecting these variables is beyond the scope of this paper.

6 Related Work

The vast majority of code generation related literature on clustered VLIW processors is on **homogeneous designs**.

Pioneering work on code generation for clustered architectures appeared in [5], where the Bottom-Up-Greedy (BUG) cluster-assignment algorithm was introduced. The main heuristic used is the completion-cycle, which calculates the completion cycle of an instruction on each of the possible cluster candidates.

Significant contributions to compilation for clustered VLIW machines were made in the context of the Multiflow compiler [12]. Clustering is based on Ellis' work ([5]). 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.

[4] provides an iterative solution to cluster assignment. Each iteration of the algorithm measures the schedule length by performing instruction scheduling and by doing a fast register pressure and communication estimation. This being an iterative algorithm, it has a long run-time and its use is not practical in compilers.

The first work that combines cluster assignment and instruction scheduling was UAS [14]. Unlike BUG ([5]), this is list-scheduling based, not critical-path based solution. Several clustering heuristics are evaluated with the start-cycle heuristic (that is the first half of BUG's completion-cycle heuristic ([5])) shown to be the best one on an architecture with a 1-cycle inter-cluster delay. This work considers the inter-cluster bandwidth as a scheduling resource. UCIFF's scheduling core extends UAS, as discussed in detail in Section 3.2.

CARS ([9,10]) 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. UCIFF could be adapted to work in such a framework for architectures with small register files, where register pressure becomes a bottleneck.

The RAW clustered architecture ([11]) communicates data across clusters with send/receive instructions. The scheduler visits instructions in a topological order and uses the completion time heuristic to guide the process.

A dynamically-scheduled **heterogeneous clustered** processor was proposed in [3]. The dual-cluster design has one high-performance and one low-performance cluster. It does not support DVFS. A DVFS-capable heterogeneous clustered processor was introduced by [13]. The proposed design though is a dynamically scheduled one, and as such no contributions are made on the compiler side.

The most closely related work to UCIFF is [2]. It proposes **code generation** techniques for a heterogeneous clustered VLIW processor, very similar to ours.

It proposes a loop scheduling algorithm based on modulo scheduling. This approach however, as we have discussed extensively in Section 2.2, suffers from the phase ordering issue of frequency selection and scheduling which are completely decoupled from one another. The frequency selection is done by estimating the energy and the execution time of each frequency configuration based on profiling data from a homogeneous run.

7 Conclusion

Energy efficiency is becoming a predominant design factor in high performance microprocessors. Heterogeneous clustered VLIW architectures are a viable choice under these design goals. This paper proposes a code generation algorithm for such architectures that performs cluster assignment, instruction scheduling and per-cluster fast frequency selection in a unified manner. Our evaluation shows that the proposed algorithm produces code of superior quality than the existing state-of-the-art and reaches the quality of a scheduler with an oracle frequency selector. This is achieved with a modest increase in algorithmic complexity.

References

1. Gcc: Gnu compiler collection. *http://gcc.gnu.org*.
2. A. Aleta, J. Codina, A. González, and D. Kaeli. Heterogeneous clustered vliw microarchitectures. In *CGO*, pages 354–366, 2007.
3. A. Baniasadi and A. Moshovos. Asymmetric-frequency clustering: a power-aware back-end for high-performance processors. In *ISLPED*, pages 255–258, 2002.
4. G. Desoli. Instruction assignment for clustered vliw dsp compilers: A new approach. *HP laboratories technical report HPL*, 1998.
5. J. Ellis. Bulldog: A compiler for vliw architectures. Technical report, Yale Univ., New Haven, CT (USA), 1985.
6. P. Faraboschi, G. Brown, et al. Lx: a technology platform for customizable vliw embedded processing. In *ISCA*, pages 203–213, 2000.
7. J. Fridman and Z. Greenfield. The tigersharc dsp architecture. *Micro, IEEE*, 20(1):66–76, 2000.
8. J. Fritts, F. Steiling, et al. Mediabench ii video: expediting the next generation of video systems research. In *Proceedings of SPIE*, volume 5683, page 79, 2005.
9. K. Kailas, K. Ebcioglu, and A. Agrawala. Cars: a new code generation framework for clustered ilp processors. *Technical Report UMIACS-TR-2000-55*, 2000.
10. K. Kailas, K. Ebcioglu, and A. Agrawala. Cars: a new code generation framework for clustered ilp processors. In *HPCA*, pages 133 –143, 2001.
11. W. Lee, R. Barua, et al. Space-time scheduling of instruction-level parallelism on a raw machine. In *ASPLOS*, 1998.
12. P. G. Lowney, S. M. Freudenberger, et al. The multiflow trace scheduling compiler. *Journal of Supercomputing*, 7:51–142, 1993.
13. N. Muralimanohar et al. Power efficient resource scaling in partitioned architectures through dynamic heterogeneity. In *ISPASS*, pages 100–111, 2006.
14. E. Ozer et al. Unified assign and schedule: a new approach to scheduling for clustered register file microarchitectures. pages 308 –315, 1998.
15. G. Pechanek and S. Vassiliadis. The ManArray embedded processor architecture. In *Euromicro*, volume 1, pages 348–355, 2000.
16. H. Sharangpani and H. Arora. Itanium processor microarchitecture. *Micro, IEEE*, 20(5):24–43, 2000.
17. A. Terechko and H. Corporaal. Inter-cluster communication in vliw architectures. *ACM Transactions on Architecture and Code Optimization (TACO)*, 4(2):11, 2007.