# Aligned Scheduling: Cache-efficient Instruction Scheduling for VLIW Processors *

Vasileios Porpodas † and Marcelo Cintra †*

School of Informatics, University of Edinburgh†
Intel Labs Braunschweig*
{v.porpodas@, mc@staffmail.}ed.ac.uk

**Abstract.** The performance of statically scheduled VLIW processors is highly sensitive to the instruction scheduling performed by the compiler. In this work we identify a major deficiency in existing instruction scheduling for VLIW processors. Unlike most dynamically scheduled processors, a VLIW processor with no load-use hardware interlocks will completely stall upon a cache-miss of any of the operations that are scheduled to run in parallel. Other operations in the same or subsequent instruction words must stall. However, if coupled with non-blocking caches, the VLIW processor is capable of simultaneously resolving multiple loads from the same word. Existing instruction scheduling algorithms do not optimize for this VLIW-specific problem.
We propose Aligned Scheduling, a novel instruction scheduling algorithm that improves performance of VLIW processors with non-blocking caches by enabling them to better cope with unpredictable cache-memory latencies. Aligned Scheduling exploits the VLIW-specific cache-miss semantics to efficiently align cache misses on the same scheduling cycle, increasing the probability that they get serviced simultaneously. Our evaluation shows that Aligned Scheduling improves the performance of VLIW processors across a range of benchmarks from the Mediabench II and SPEC CINT2000 benchmark suites up to 20%.

## 1 Introduction

Very Long Instruction Word (VLIW) processors are wide-issue statically scheduled processors. They are used in a wide range of domains: in GPUs (AMD's VLIW-5 architecture on Radeon GPUs and in APUs [4]), in embedded systems as DSPs (Texas Instrument's VelociTI, HP/ST's Lx [8], Analog's TigerSHARC [11], BOPS' ManArray [23]) and as targets of dynamic binary translation (e.g. Transmeta's Crusoe [5, 14]). A VLIW-like architecture (with many unique dynamic hardware additions for run-time optimizations) is also used in servers (Intel's Itanium/Itanium2 EPIC architecture [20, 27]).

Compared to dynamically scheduled processors, VLIW designs operate at an attractive power/performance point. This is because they are by design both simple (no dynamic scheduling hardware [10]) and wide-issue. They rely on the compiler's instruction scheduling pass to optimally schedule instructions. Instruction scheduling algorithms re-arrange the instructions of the input program to hide pipeline latencies. Schedulers for VLIW processors in particular, explicitly express instruction level parallelism (ILP) in long instruction words.

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

The simplicity of the VLIW hardware design, however, comes at a cost: VLIW processors are more sensitive to dynamic latencies triggered by micro-architectural events, such as cache misses, than their dynamically scheduled counterparts. This is because a traditional VLIW processor comes to a complete halt upon a cache miss caused by any instruction in the long instruction word, due to the absence of load-use hardware interlocks. Therefore even if there exist instructions that could execute while the miss is being serviced, they do not because the VLIW hardware does not allow it. We refer to these VLIW cache-miss semantics as *Stall-On-Miss* (SOM) (Fig.1c).

Performance can be improved once we deviate from the VLIW design philosophy and introduce data hazard detection in hardware. This limits the processor stalls to the cases when a VLIW instruction tries to use data that is not available (brought in by the Load-miss). We refer to this model as *Stall-On-Use* (SOU) (Fig.1d). In this model, the long instruction words remain intact and the dependencies are tracked at the VLIW word level.

If we apply a full-blown register scoreboarding in hardware, we can break down the instruction words into individual instructions and we can allow each instruction to issue and stall independently of the others (Fig.1e). This allows for optimal pipeline throughput as the execution only stalls when dictated by the data dependencies. This approach, however, requires hardware components that are normally found in dynamically scheduled superscalar processors, thus deviating from the VLIW design concept of keeping the hardware simple. This is the reason why most VLIW processors are designed to be either SOM or SOU. In this work we only consider the SOM and SOU models.

A SOU architecture requires Non-Blocking caches [15] to function. These caches are equipped with a simple hardware mechanism that allows them to resolve multiple misses simultaneously. Their impact on performance on dynamically scheduled processors is significant since they decrease the pipeline stalls. The performance improvement however, on a VLIW processor with SOM semantics is not as impressive under existing instruction schedulers.

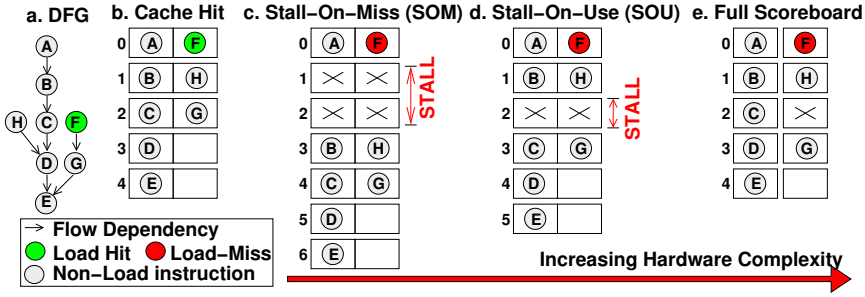



**Fig. 1.** Dynamic Schedules of DFG (a) as we increase hardware complexity.

Most schedulers can effectively deal with regular long-latency instructions, such as integer division. They try to hide long latencies by executing other low-latency instructions in parallel. Existing instruction schedulers consider Load instructions as regular instructions of some latency: either low-latency (cache-hit), high-latency (cache miss) or something in between. This effectively changes how the scheduler treats the loads: as hits, misses or in between. This approach works fine for dynamically-scheduled processors. The Stall-On-Miss semantics of

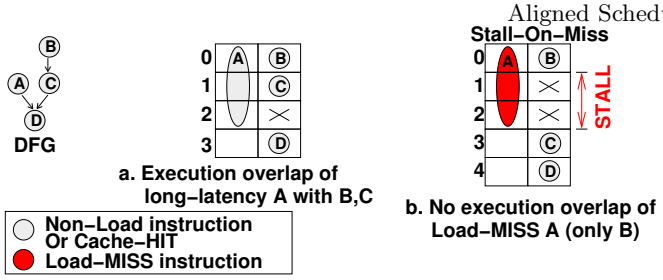



**Fig. 2.** VLIW semantics of a regular long-latency instruction (a) VS a cache-miss (b).

a VLIW processor however, require special treatment by the instruction scheduler. Fig.2 shows that trying to hide load miss latency by scheduling other instructions in parallel is not suitable for VLIWs. This is because on a VLIW with no load-use interlocks, the semantics of a regular long-latency instruction (Non-Load instruction Fig.2a) are different from a cache-miss of equal latency (Load instruction Fig.2b). On one hand the high-latency regular instruction A in Fig.2a can overlap its execution with B and C. On the other hand, cache-miss A in Fig.2b cannot overlap with instructions C or D due to Stall-On-Miss semantics. Therefore such VLIW architectures require a radically different scheduling approach for hiding cache-miss latencies.

This paper proposes Aligned Scheduling, a novel instruction scheduling algorithm for statically scheduled VLIW processors with non-blocking caches that treats Load instructions differently than existing schemes. It improves the tolerance of VLIW processors to cache-miss latencies by exploiting four concepts:

**1.** The VLIW-specific Stall-On-Miss or Stall-On-Use cache-miss semantics.

**2.** Non-blocking caches ([15, 28]), that can service multiple cache misses simultaneously.

**3.** The statically provable Memory-Level Parallelism (MLP), that allows multiple memory Load operations to execute on the same VLIW cycle.

**4.** The explicit instruction parallelism of VLIW instruction words.

These concepts allow the instruction scheduler to hide cache-miss latencies by aligning memory Load instructions together on the same cycle, in a smart way. In this way, during execution, the probability that multiple Load instructions miss simultaneously increases. We refer to this effect of multiple aligned Load instructions missing simultaneously as *miss overlapping* (Fig.3). Aligned Scheduling proves particularly effective for VLIWs with no load-use hardware interlocks (SOM), but as shown in the Section 5, it could potentially benefit SOU under high miss latency conditions.

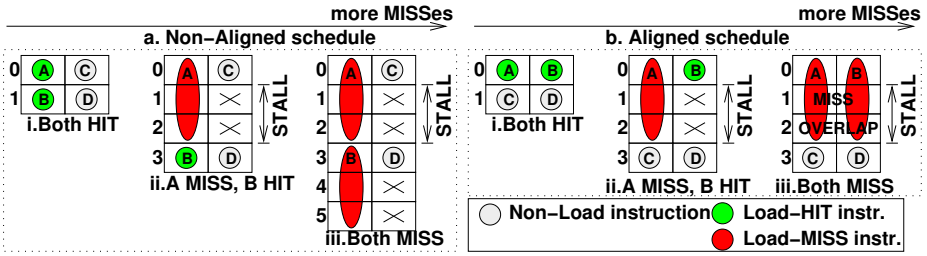



**Fig. 3.** Two different schedules a and b under increasing miss conditions. Schedule b (Aligned) exhibits miss-overlapping under heavy miss conditions (b.iii).

## 2 Motivation

The main concept that Aligned Scheduling is based on is the idea of *miss overlapping* (Fig.3). If the architecture supports non-blocking caches, then more than a single outstanding cache miss can be serviced simultaneously. Instruction schedulers currently do not exploit this feature of the architecture and tend to generate schedules as in Fig.3a, which perform well when there are no or few cache misses (Fig.3a i,ii) but are suboptimal when there are bursts of cache misses (Fig.3a iii). An optimized scheduler for VLIW should exploit the non-blocking caches to schedule loads in parallel, whenever this is profitable. Aligned Scheduling does so selectively and generates a schedule which still performs well under low cache miss conditions (Fig.3b i,ii) but manages to outperform the existing approaches under bursts of cache misses (Fig.3b iii).

The motivating examples (Fig.4(a) and Fig.4(b)) describe two different but complementary heuristics that are used in Aligned Scheduling. Each example is based on its own Data Flow Graph (DFG), Fig.4(a)a and Fig.4(b)a respectively. Both DFGs contain Load instructions (green) and non-Load instructions (light gray). The examples compare the schedules generated by two schedulers: i) The baseline scheduler (top sub-figures b,d,f), a state-of-the-art list-scheduler (like the scheduler in GCC [1]) and ii) Aligned Scheduler (bottom sub-figures c,e,g). The colors on the DFG and schedules are consistent. Red represents a Load that misses in the cache. The leftmost column of each figure (sub-figures b,c) shows the static schedule produced by the scheduler. These schedules also happen to match the dynamic (run-time) schedule when all Load instructions are hits. This is why in both sub-figures b and c the loads are green, suggesting a cache-hit. The other two columns show the case when all Loads miss: The center column (sub-figures d,e) corresponds to a Stall-On-Miss (SOM) architecture and the rightmost column (sub-figures f,g) corresponds to Stall-On-Use (SOU).

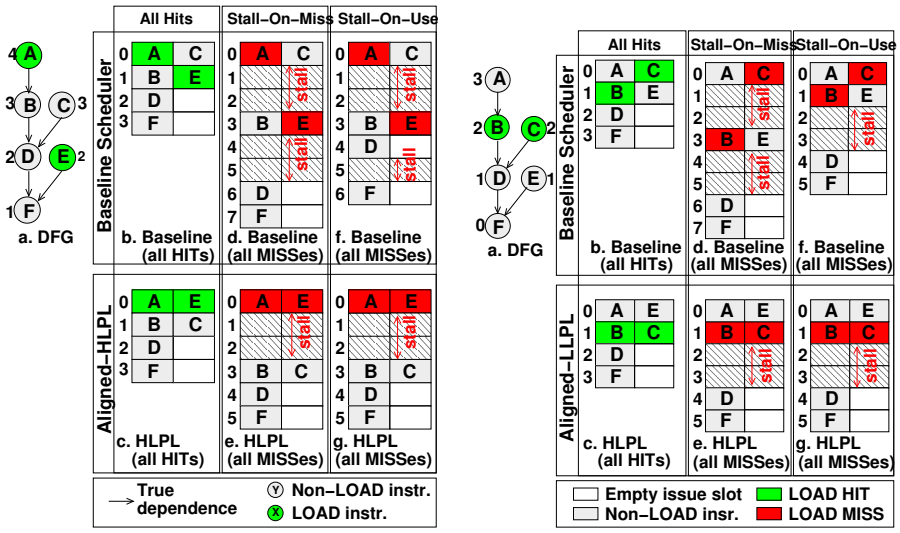
The baseline is a list scheduler, like the default scheduler in most industrial-strength compilers (e.g. GCC). It prioritizes the ready instructions based on a priority function (in this case the height of each node in the graph), and emits the highest priority ready instruction into the schedule. Aligned Scheduling is also a list-scheduler based algorithm, but differs from the baseline in the instruction selection process (see Fig.5 "Aligned-select"). The performance of a scheduler is inversely proportional to the dynamic schedule length. In this example we are interested in comparing the two schedulers in cache-hit (sub-figures b,c) and cache-miss (sub-fiures d,e and f,g) scenarios.

Both examples (Fig.4(a) and Fig.4(b)) motivate the main concept of Aligned Scheduling, which is that the VLIW stall semantics require that a good schedule, resilient to misses, should have Load instructions scheduled in parallel on the same cycle, so that the cache misses can overlap in time.

### 2.1 Hoisting of Low-Priority Loads (HLPL)

The first example (Fig.4(a)) shows that a scheduler that hoists low-priority Loads by giving preference to them instead of other higher priority instructions, can improve performance under a burst of Load misses.

The highest priority instruction of the DFG of Fig.4(a)a is Load A. At cycle 0, the scheduler's ready list contains A, C and E. Since A is the instruction with




(a) HLPL: (Hoist of Low-Priority Load). (b) LLPL: (Lower of Low-Priority Load).

**Fig. 4.** Aligned Scheduling heuristics. The numbers on the DFG are the instr. priorities.

the highest priority (4), it gets issued at cycle 0. Next, an unmodified priority-based list scheduler (Fig.4(a) b,d,f) would select C with priority 3. The HLPL heuristic of Aligned Scheduling, though, will select E with priority 2, since this will allow for both Loads (A and E) to execute on the same cycle (Fig.4(a)c,e,g).

If at run-time none of the Loads miss, the dynamic schedule will look exactly like the static one (Fig.4(a)b). If, however, at run-time both Load instructions (A and E) miss, then the execution will look as in Fig.4(a)d or Fig.4(a)f, depending on the stall semantics. In this case, the run-time performance of the Baseline scheduler is worse than the Aligned one for both Stall-On-Miss and Stall-On-Use semantics.

The Aligned-HLPL heuristic makes sure that the low-priority Load instructions (like Load E), get hoisted and scheduled on the same cycle as high-priority Load instructions, like Load A on cycle 0 (Fig.4(a)c). This suggests that unlike the baseline scheduler, in Aligned-HLPL instruction priority does not always drive the scheduling algorithm. Instead low-priority Load instructions may take precedence over high-priority non-Load instructions. For example the high-priority non-Load instruction C gets deferred to a later cycle than the lower-priority E (Fig.4(a)c). This leads to better performance under bursts of misses, and still a good schedule under the "all HITs" case (Fig.4(a) c,e,g).

### 2.2 Lowering of Low-Priority Loads (LLPL)

The previously described HLPL heuristic can only work if a high-priority load is scheduled first on the current scheduling cycle. The LLPL heuristic complements HLPL, by taking action when a high-priority non-Load instruction is scheduled first on the current scheduling cycle.

The LLPL heuristic (Fig.4(b)) avoids scheduling low priority Load instructions if the highest priority instruction on the current scheduling cycle is not a Load. Even if there are no instructions left to schedule but Loads, LLPL will de-

fer them to some later cycle. This is beneficial for two reasons: **1.** It guarantees that the current cycle remains stall-free, since there are no Load instructions to miss. **2.** It increases the chances that more Load instructions get grouped together and aligned on a future cycle.

LLPL can be better explained through the example of Fig.4(b). As in Section2.1, the Baseline scheduler is driven purely by instruction priorities and issue slot availability. Therefore Load C gets scheduled on a different cycle than Load B, as shown in Fig.4(b)b.

Aligned-LLPL however is not guided solely by the instruction priorities. Instead it focuses on deferring low-priority Load instructions of the ready list (e.g., C at cycle 0 which is not the highest priority instruction) to a later cycle as long as the high priority instruction is not a Load (A at cycle 0). The end result is that instruction C gets scheduled later (at cycle 1) along with Load B.

When all instructions are hits ("all HITs" scenario) both the Baseline and Aligned Scheduling-LLPL perform equally well (Fig.4(b)b and Fig.4(b)c). When both Loads miss however, Aligned-LLPL is faster (Fig.4(b)d,f vs Fig.4(b)e,g). The speedup, is once again due to the overlapping of miss-latencies.

## 3 Aligned Scheduling

### 3.1 Overview

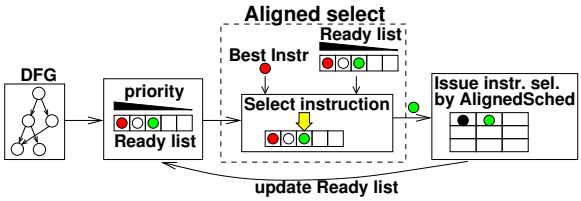



**Fig. 5.** Aligned Scheduling structure.

Aligned Scheduling is based on the commonly used list-scheduling algorithm. An overview of how the common (baseline) list scheduling algorithm works is shown in Fig.5, without the "Aligned select" component.

The input to list scheduling is a Data Flow Graph (DFG) with its nodes tagged with priorities. The priority can be calculated based on various heuristics, a common one being the height from the bottom of the DFG. With the term "ready instructions" we mean the instructions that have all their inputs calculated and available to them. The ready instructions of the DFG are placed into a ready list and are sorted based on their priority. The highest-priority instruction is selected and scheduled. Scheduling an instruction causes its DFG successors to become ready and to be added to the ready list. The scheduler steps to the next cycle under two conditions: 1) The ready list is empty, meaning that there are no available instructions to schedule 2) The current cycle is full, so no more instructions can be scheduled in it. This process repeats until all instructions in the DFG are scheduled.

Aligned Scheduling (Fig.5) adds the "Aligned select" phase to the common list scheduling algorithm. This process is placed in between sorting the ready list

and scheduling an instruction. It uses the ready instruction list and the highest priority instruction of the current cycle to make an informed decision on selecting the instruction that should be scheduled at the current scheduling cycle. This is where HLPL and LLPL are used. The instruction that "Aligned select" returns, gets scheduled at the current cycle.

Although Aligned Scheduling is built on top of GCC's EBB-region-based scheduler, in principle the "Aligned select" step can be plugged in to other schedulers as well (e.g., the Selective Scheduler ([21]), Modulo Scheduling [16, 24, 6, 18] etc.) without major modifications to these algorithms.

The Aligned Scheduling algorithm can be logically split in two parts: **1.** The **main driver** function (Alg.1), which performs the high-level actions of a list-scheduler. **2.** The Aligned Scheduling **selection** function (Alg.2) which is used for the selection of the instruction that gets scheduled by the main driver function.

**Algorithm 1.** Aligned Scheduling algorithm.

```
1  aligned ()
2  {
3   /* While there are unscheduled isntr. */
4   while (instructions left to schedule)
5     update READY_LIST with ready +
          ↪deferred instr.
6     sort READY_LIST based on priorities
7     BEST_INSTR = READY_LIST [0]
8       while (READY_LIST not empty)
9         INSN=aligned_select(BEST_INSTR,
              ↪READY_LIST)
10        if (no INSN selected)
11          break
12        if (INSN can be sched. at CYCLE)
13          schedule INSN
14          remove INSN from READY_LIST
15        /* If failed, defer to cycle+1 */
16        if (INSN unscheduled)
17          remove INSN from READY_LIST and
                ↪re-insert it at CYCLE + 1
18     /* READY_LIST is empty */
19     CYCLE ++
20 }
```

**Algorithm 2.** Aligned Scheduling instruction selection

```
1  /*In1: Highest prio. instr. of curr cycle
2    In2: List of ready instr. of curr cycle
3    Out: Instruction to schedule on cycle*/
4  aligned_select (BEST_INSTR, READY_LIST)
5  {
6   if (BEST_INSTR is LOAD)
7     if (HLPL)
8       for INSTR in sorted READY_LIST
9         if (INSTR is LOAD)
10          return INSTR
11    return READY_LIST [0]
12  else if (BEST_INSTR is not a LOAD)
13    if (LLPL)
14      for INSTR in sorted READY_LIST
15        if (INSTR is not LOAD)
16          return INSTR
17    else
18      return READY_LIST [0]
19  else
20    return READY_LIST [0]
21 }
```

### 3.2 Aligned Scheduling driver

The **main driver** function (Alg.1) performs the main actions of a list-scheduling algorithm adjusted to work with the Aligned Scheduling heuristics. While there are instructions left to schedule (line 4) it keeps iterating. First, it fills in the ready list with any ready instruction (line 5), then it sorts the ready list (line 6) based on the instruction priorities (which is usually the height of the instruction in the DFG). Next it finds the highest priority instruction for this cycle and stores it into BEST_INSTR (line 7).

The algorithm then schedules the ready instructions one by one (lines 8 - 17). This part of the algorithm keeps iterating until: 1) the ready list is empty (line 8), or 2) no instruction is selected by the Align-selection function. The ready list empties in two ways: 1. Scheduled instructions are removed from the ready list 2. When no more instructions fit in the current cycle (due to insufficient execution slots) then the ready instructions still get popped out of the ready list without being scheduled and get deferred to the next cycle (line 17).

Instructions get selected from the ready list by the "aligned_select()" function (line 9). The implementation of this function is shown in Alg.2. If no instruction is selected by "aligned_select" (i.e. there are no instructions left to schedule in this cycle), then the algorithm breaks out of the innermost while loop (lines 10-11) to abandon scheduling on the current cycle and to step to the next cycle. This enables LLPL to leave a cycle partially scheduled even if there are ready instructions left to schedule. Else, if an instruction has been selected, then it gets scheduled and removed from the ready list (lines 12-14). If, due to resource constraints (e.g., no more issue slots) the instruction cannot be scheduled on the current scheduling cycle, then it is removed from the ready list (lines 16, 17). Finally, if there are no instructions left in the ready list, it is time to move to the next scheduling cycle (lines 18, 19) and restart with a fresh ready list at the top of the outer loop (line 4).

### 3.3 Aligned Scheduling selection

At the core of the Aligned Scheduling algorithm lies the **aligned_select ()** function (Alg.2). This function decides which instruction, among the ready ones, will be executed on the current scheduling cycle. This function makes use of the HLPL and LLPL heuristics to decide on the instruction selected.

This function exploits the statically (at compile time) analyzable MLP to improve the schedule's performance of VLIW processors with non-blocking caches under high cache-miss rate conditions. The end result of the instruction selection (with the help of the driver function of Alg.1) is a hoisting and lowering of Load instructions aiming at **grouping loads** together as much as possible.

Internally, the selection algorithm is composed of two different but complementary heuristics: The "Hoist of Low-Priority Load" (HLPL) heuristic as demonstrated in the motivation Section 2.1 and the "Lower of Low-Priority Load" (LLPL) heuristic as discussed in Section 2.2. If both are active, either HLPL or LLPL executes **depending on the type of the highest priority instruction** (BEST_INSTR) of the current scheduling cycle (Alg.2, lines 6,12). If it is a Load then HLPL performs hoisting of other Loads. Else if it is not a Load, then LLPL forms a Load-free cycle by lowering loads to later cycles. The insight behind it is that the critical path should be honored. Therefore the highest priority instruction (BEST_INSTR) of the cycle should guide the type of instructions that are aligned with it. We can enable each or both of these heuristics by controlling the HLPL and LLPL flags (Alg.2 line 7 and line 13, respectively).

The instruction hoisting/lowering of Aligned Scheduling is done in a **balanced** way: **1.** The Load hoisting and lowering is **mild enough** such that the re-arranged instructions do not replace other highly-critical instructions. This guarantees acceptable performance on a low cache-miss rate conditions. **2.** The Load hoisting and lowering is **aggressive enough** that the Load instructions get grouped together so that we get high miss overlapping and performance improvements on high cache-miss scenarios.

The first point is achieved by honoring the critical path and always scheduling the highest priority instruction of the ready list (BEST_INSTR) without any delays (Alg.2 lines 9,15 guarantee this). Also the most critical instruction guides

the kind of hoisting/lowering that takes place (Alg.2 lines 6,12). The second point is achieved by selectively hoisting/lowering all lower priority instructions.

**HLPL:** If BEST_INSTR is a Load (Alg.2, line6), then the HLPL heuristic can be applied (line 7). It iterates over the list of sorted ready instructions (line 8) and selects the first load instruction encountered (lines 9, 10). If there are no ready load instructions to choose from, HLPL will select a non-Load instruction (line 11) as this can only be beneficial. This is because scheduling non-Load instructions, after all Load instructions have been scheduled on the cycle, cannot cause any further stalls or delays for this cycle, so it can cause no harm. Instead, deferring the execution of non-Load instructions to later cycles can only degrade performance. HLPL will usually not harm performance under low miss-rate conditions.

**LLPL:** In the opposite case, if BEST_INSTR, the highest priority instruction of the current cycle, is not a Load (line 12), the LLPL heuristic can be applied. In short, LLPL creates a Load-free cycle. It does so by deferring the execution of any Load instruction to future cycles. This is done by iterating across the ready list (line 14) and selecting only non-Load instructions to schedule (lines 15, 16). Unlike HLPL, when LLPL is "on" then even if there are no other non-Load instructions left in the ready list, the algorithm will **not** select a Load, therefore the current scheduling cycle will be partially empty. This is good for two reasons: **1.** It guarantees that the current cycle does not stall (since it contains no Loads) **2.** It enables future co-execution of Load instructions in later cycles. However, LLPL could potentially harm performance as it deliberately leaves resources under-utilized. LLPL proves to be an aggressive heuristic for high miss-rate conditions, but can cause slowdowns on low miss-rate conditions.

Enabling both heuristics is usually the best practice, since the resulting performance is usually better than either them in isolation (see Section 5).

## 4 Experimental Setup

The target **architecture** is a statically scheduled Stall-On-Miss/Stall-On-Use VLIW, that uses the IA64 [27] instruction set due to widespread availability of tools for this ISA. The architecture has a configurable issue width. It is worth noting that the real Itanium processor used in servers is based on the EPIC architecture, which although looking similar to a VLIW one, has many hardware features not found in common VLIW architectures. One of these hardware features is a hardware register scoreboard. Our target is a common VLIW without the full-blown register scoreboard of the Itanium.

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

We simulated the architecture on a modified version of SKI [2], IA64 cycle accurate simulator that supports a configurable non-blocking cache hierarchy and both SOM or SOU semantics. The issue width is configurable, ranging from 2 to 4 wide and each issue slot can execute an instruction of any type. The L1 cache is 16K-1way, with a block-size of 64 Bytes and a 1 cycle latency. The L2 cache is 256K-4way with a block size of 128 Bytes and an 8 cycle latency. Both caches are non-blocking. The access to the main memory takes 150 cycles.

We evaluated Aligned Scheduling on 6 of the Mediabench II video [12] and 6 of the SPEC2000 CINT [3] **benchmarks**. All benchmarks were compiled with

several optimizations enabled (-O2) and both schedulers running. We ran all benchmarks to completion.

## 5 Results and Analysis

We first present a detailed case study of Aligned Scheduling on the cjpeg benchmark of the Mediabench II benchmark suite (Section 5.1). We then present summarized results for the rest of the benchmarks (Section 5.2).

### 5.1 Case study: cjpeg

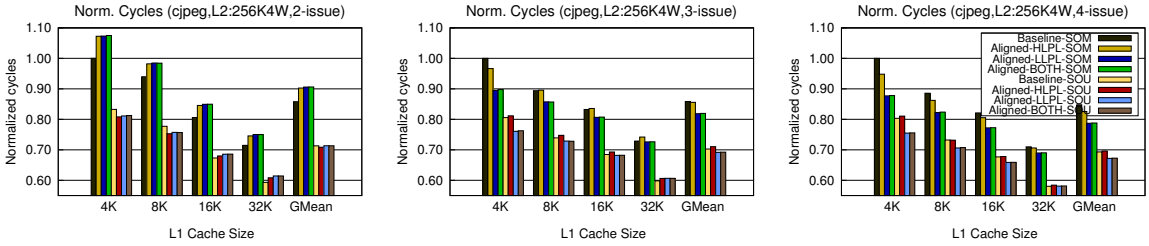
The cjpeg benchmark of the Mediabench II ([12]) video suite is a representative example for evaluating Aligned Scheduling. This benchmark has a working set of 16KB which is small enough that we can test Aligned Scheduling across a broad range of cache-miss scenarios (ranging from high miss-rates to low miss-rates) by simply changing the L1 size.

Fig.6(a) compares the cycle counts of the Aligned Scheduling-{HLPL, LLPL and BOTH} heuristics against the Baseline scheduling. The comparison is done over various L1 cache sizes, ranging from 4KB to 32KB 1-way, and on three different issue widths of the VLIW processor (issue 2-4). The L2 cache is a 256KB 4-way with 8 cycles latency. Fig.6(b),6(c) complements Fig.6(a) by providing the L1 and L2 miss rates respectively for each case. Finally, Fig.6(d) shows the amount of overlapping of cache misses and Fig.6(e) shows the average load latency. These figures provide some important insights on the strengths and weaknesses of Aligned Scheduling:

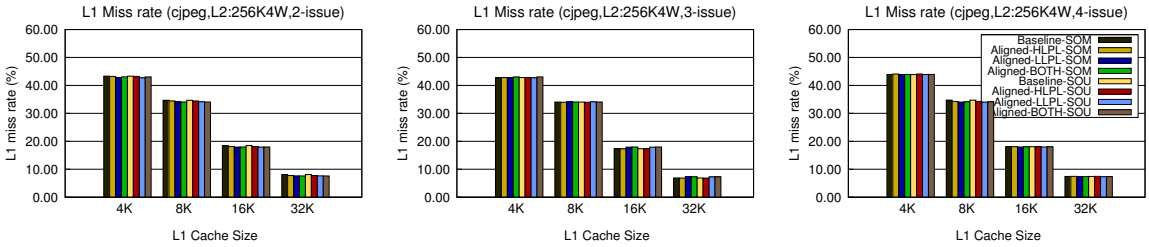
**a.** The first thing to notice in Fig.6(a) is that for the Stall-On-Miss semantics and small L1 sizes, Aligned Scheduling **outperforms** the baseline by a considerable margin, in fact it performs equally well or better than the baseline with twice as much L1 memory (e.g., Fig.6(a) 3/4-issue 4K,8K SOM), improving performance by about 20%. Therefore, for small cache sizes, Aligned Scheduling bridges half the performance gap between a SOM and a SOU architecture, with **no additional hardware**. Aligned Scheduling performance improvements, however, decrease as the cache size increases. This is because cache misses become less frequent (Fig.6(b)), therefore the probability of them happening simultaneously (something that Aligned Scheduling could exploit) decreases. The point of diminishing returns for cjpeg is the point when the working set size equals the cache size (16KB). For sizes greater than 32KB, the L1 miss rate drops below 8% and Aligned Scheduling cannot improve performance, but it does not hurt it either.

**b.** The two Aligned Scheduling heuristics (HLPL and LLPL) work orthogonally and when both enabled they **act cooperatively**. Enabling both (Aligned-BOTH Fig.6(a)) outperforms each individual heuristic Aligned-HLPL or Aligned-LLPL, by a significant margin. This is true for both SOM and SOU semantics.

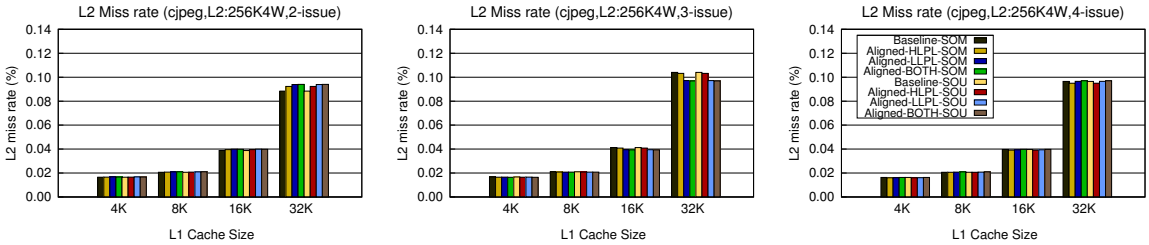
**c.** Aligned Scheduling performs better as the **issue width** increases. In fact, for cjpeg, and for the degenerate VLIW case of 2-issue and for SOM semantics, Aligned scheduling causes a slowdown. This is an example where the alignment cost outweighs the benefit: Since the issue width is too narrow, the cache misses cannot be effectively overlapped, therefore the scheduling penalty of issuing instructions ignoring their priorities outweighs the benefit of doing so. For any



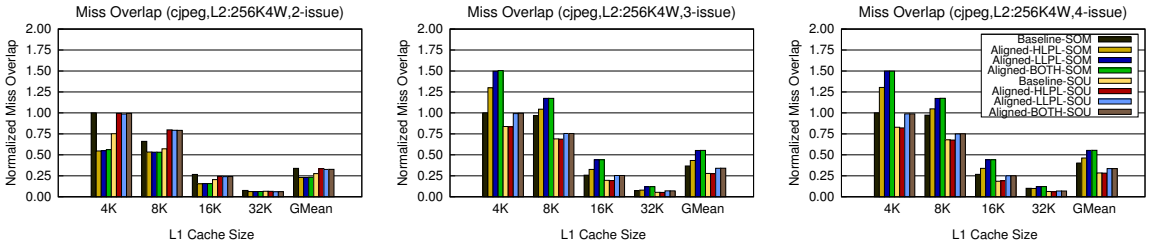

(a) Normalized Cycle counts (cjpeg).



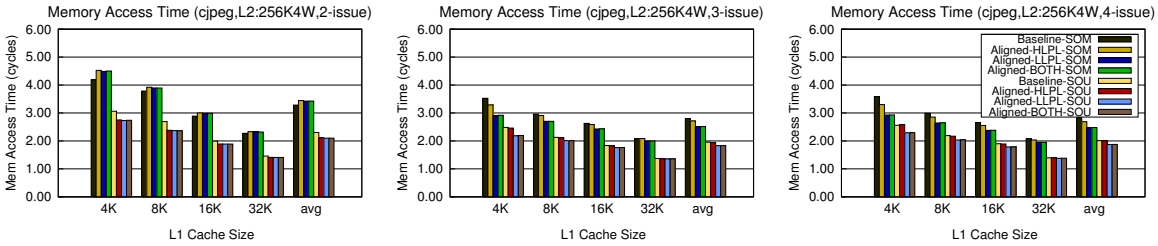

(b) L1 cache miss rate (cjpeg).




(c) L2 cache miss rate (cjpeg).




(d) Normalized cache-miss overlapping (cjpeg).




(e) Memory Access Time (i.e. the average Load latency) (cjpeg).

**Fig. 6.** Performance of cjpeg for issue widths (2-4 issue) and L1 cache sizes (4K-32K)

issue width higher than 2, Aligned Scheduling improves performance considerably. This is intuitive as the more the issue slots, the more loads can get serviced in parallel, which is exactly what Aligned Scheduling is meant to exploit.

**d.** An architecture with Stall-On-Use semantics can still benefit from Aligned scheduling, though the performance improvement is less impressive. For small cache sizes, the performance improvement is about 5%, but as we get close to the working set size, there is little or no improvement. The reason (explained in Section 2) is that with SOU semantics there are fewer opportunities to increase the miss overlap, beyond what the hardware provides.

**e.** A **Miss-Overlap** is the event of multiple cache misses being serviced in parallel. The count of overlapping misses is a measure of the effectiveness of Aligned Scheduling. Fig.6(d) shows that the performance improvements of Fig.6(a) are indeed caused by the increase in cache overlaps and not some other scheduling side-effect.

**f.** According to Fig.6(e), the effective average latency of a Load (**Memory Access Time**) decreases with Aligned scheduling on wide-issue VLIW processors. This proves once more that the performance improvements are due to overcoming the cache bottle-neck.

**g.** Finally, the L1 and L2 **miss-rate** (Fig.6(b)) seems to be largely unaffected by the application of Aligned Scheduling. This is because: i) a miss is still counted as a single miss even if it overlaps with another miss and ii) Aligned scheduling, does not cause large-scale memory access reordering that could affect the cache behavior. Therefore Aligned Scheduling speedups are not due of fewer misses but rather due to decreasing the total amount of time that the VLIW processor has to wait for the misses to be serviced.

### 5.2 All benchmarks

We now consider all benchmarks (Fig.7). We measured the cycle count, the miss rate on both L1 and L2 caches, the overlapping of cache misses, and the average memory access time. We ran the benchmarks on a 4-issue VLIW processor with 16KB-1way L1 and 256KB-4way L2 cache (see Section 4). We focus on the performance of Aligned Scheduling compared to the Baseline Scheduler, all on SOM. We compare them against the Baseline on SOU, which is hardware supported and is therefore an estimate of the best we could expect from Aligned Scheduling, a software-only approach. Aligned-SOM in Fig.7, is equivalent to to Aligned-BOTH-SOM (both HLPL and LLPL enabled).

The results in Fig.7 show that Aligned Scheduling works for a variety of benchmarks and achieves significant speedups on this architecture configuration. In memory-bound benchmarks (e.g. 181.mcf) it even manages to reach the performance levels of the hardware-based SOU. Aligned Scheduling is successful at increasing the count of misses that overlap, as shown in the Miss-overlap graph of Fig.7. In some cases (e.g. h263enc), the performance improvement can also be attributed to a lower miss-rate, a side-effect of the instruction re-ordering. Only few benchmarks (197.parser and 300.twolf) have fewer miss overlaps compared to the baseline, but even in these cases the performance achieved is either close to the baseline or better, due to overlapping fewer misses but ones of greater latency, leading to better average memory access time.

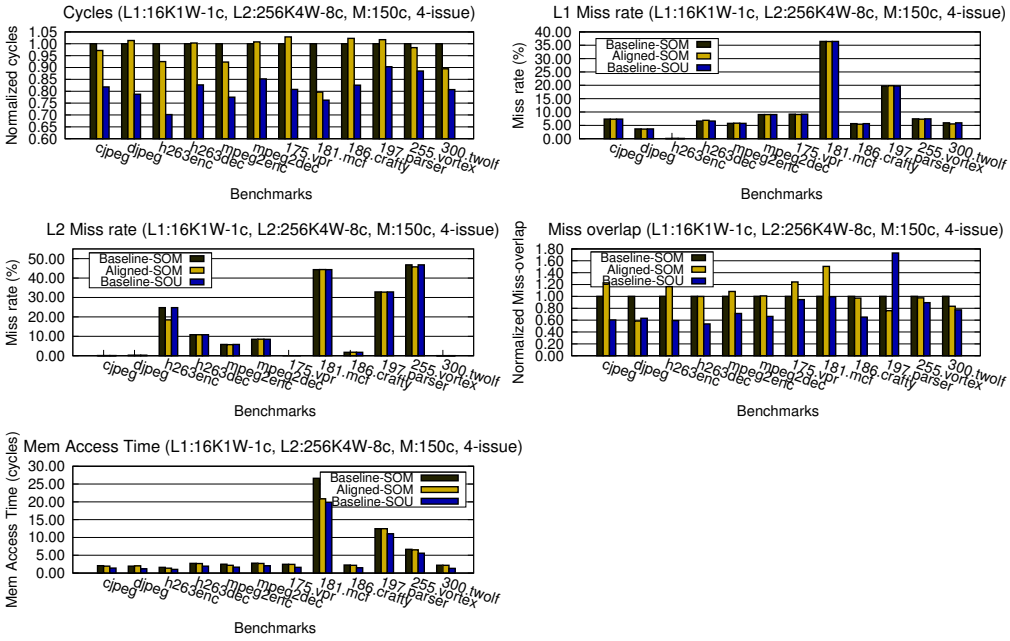



**Fig. 7.** Cycle count, Miss Rates, Miss overlaps and average Memory Access Time for 6 of the Mediabench II and the SPEC CINT2000 benchmarks.

Some of the benchmarks however are marginally worse than the baseline with 175.vpr on this processor setup, reaching a slowdown of 2.5%. These slowdowns can be attributed to one of the following: **1.** The small working set of most benchmarks(e.g. it is 16KB for the majority of the MediabenchII [12]). Therefore with the current cache setup Aligned Scheduling has a small headroom to improve the cache behavior. As shown in Section5.1 in Fig.6(a), Aligned Scheduling can indeed improve performance on such benchmarks as long as the cache sizes are smaller than their working sets. **2.** High sensitivity to the priority of the critical path instructions. In such cases any instruction re-ordering done by Aligned Scheduling can lead to a slowdown (this is true for 186.crafty, 255.vortex and h263dec). In 175.vpr this effect is so strong, that even with substantially increased miss-overlap (more than 20%), it takes a performance hit. **3.** Inability of Aligned Scheduling to form more effective groups of Load instructions than those formed by the baseline. This happens rarely (see "Miss overlap"in Fig.7 djpeg,197.parser).

Benchmarks with high miss rates (L1 or L2) usually perform well under Aligned Scheduling. As long as a benchmark has adequate amounts of statically analyzable MLP, and is not very sensitive on its critical path instructions then a high miss rate should provide opportunities for Aligned Scheduling to improve the execution cycles. This is evident in 181.mcf and h263enc. In particular h263enc, has a low L1 miss rate but a high L2 miss rate and gets a performance improvement of about 7%. This suggests that Aligned Scheduling effectively overlaps some of the performance-critical high latency L2 misses, leading to significant performance improvements.

## 6 Related Work

**Non-blocking** (also known as lockup-free caches) were introduced by [15] and have been studied in detail since (e.g. [28], [26]). Non-blocking caches are a cost-effective optimization and are common in all processors, including VLIW ones. Aligned Scheduling exploits the non-blocking feature to improve performance on VLIW processors.

**Instruction scheduling optimized for cache memories** has been studied in the past. The majority of the work [7, 13, 19, 17] focuses on improving instruction scheduling for processors with non-blocking caches and stall-on-use execution semantics. Balanced Scheduling [13] proposes a scheduling algorithm for pipelined architectures that makes sure that the processors stalls less upon a cache miss. The main goal of the instruction scheduler is to schedule the right number of instructions after a load, such that, in case of a miss, there are enough independent instructions to execute until the loaded value (that missed) is used by an instruction. [19] improves Balanced Scheduling by applying ILP enhancing optimizations. An extension to Balanced Scheduling is introduced in [17], which proposes using profiling information to drive instruction scheduling so that it makes more informed decisions. [7] proposes a static cache-reuse model that helps the instruction scheduler make informed decisions on the latency of a memory instruction. The paper shows that this produces better schedules than considering all memory instructions as either all-hits or all-misses.

Aligned Scheduling is very different from these approaches. It mainly targets VLIW processors that have Stall-On-Miss execution semantics, enabling them to improve their performance close to that of Stall-On-Use. Therefore the optimization that Aligned Scheduling introduces exploits a completely different architectural feature. There is no indication that any of the schemes that target stall-on-use semantics will consistently outperform our baseline on a stall-on-miss VLIW target, which is why we do not compare against them.

The only work we are aware of that focuses on VLIW processors is Cache Sensitive Modulo Scheduling [25]. It proposes a software-pipeline cyclic scheduling algorithm that improves performance in one of two ways: it either schedules memory instructions early or issues pre-fetch instructions. Both ways lead to fewer cache misses, with the former one proving to be the most effective one. This work is orthogonal to Aligned Scheduling as it focuses on the pre-fetching problem rather than on grouping loads together.

**Code optimizations** that exploit the non-blocking caches have been proposed in the past. [22] proposes an analysis and transformation framework for optimizations that cluster misses together, leading to significant performance improvements. The scheme involves high-level transformations, usually at loop level. Aligned Scheduling on the other hand, is a scheduling algorithm, performing fine-grain optimization in the compiler back-end.

## 7 Conclusion

This work proposes Aligned Scheduling, a new scheduling algorithm for VLIW processors that generates schedules that are more resilient to cache misses than the existing schemes. It does so by incorporating the micro-architectural knowledge of non-blocking caches and the absence of load-use interlocks into the

scheduling algorithm. Aligned Scheduling exploits the statically known MLP to group together Load instructions on the same cycle. This increases the probability that cache misses overlap and get serviced simultaneously by the non-blocking cache, therefore decreasing the amount of time the processor spends on cache stalls. Our simulation results show that significant speed-ups can be achieved across a wide range of benchmarks and VLIW architecture configurations.

## References


1. Gcc: Gnu compiler collection. *http://gcc.gnu.org*.
2. ski IA64 simulator. *http://ski.sourceforge.net*.
3. SPEC benchmark. *http://www.spec.org*.
4. A. Branover et al. AMD Fusion APU: Llano. In *IEEE Micro*, 2012.
5. J. Dehnert et al. The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges. In *CGO*, 2003.
6. J. Dehnert et al. Compiling for the Cydra. *The Journal of Supercomputing*, 1993.
7. C. Ding et al. Modulo scheduling with cache reuse information. *Euro-Par'97*.
8. P. Faraboschi et al. Lx: a technology platform for customizable vliw embedded processing. In *ISCA*, 2000.
9. J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 1981.
10. J. A. Fisher, P. Faraboschi, and C. Young. Vliw processors. *Encyclopedia of Parallel Computing*, pages 2135–2142, 2011.
11. J. Fridman and Z. Greenfield. The tigersharc dsp architecture. *IEEE Micro*, 2000.
12. J. Fritts et al. Mediabench II video: expediting the next generation of video systems research. In *SPIE*, 2005.
13. D. Kerns and S. Eggers. Balanced scheduling: Instruction scheduling when memory latency is uncertain. In *PLDI*, 1993.
14. A. Klaiber et al. The technology behind Crusoe processors. *Transmeta Corporation White Paper*, 2000.
15. D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *ISCA 1981*.
16. M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *PLDI*, 1988.
17. G. Lindenmaier, K. McKinley, and O. Temam. Load scheduling with profile information. In *Euro-Par*, 2000.
18. J. Llosa. Swing modulo scheduling: A lifetime-sensitive approach. In *PACT*, 1996.
19. J. Lo et al. Improving balanced scheduling with compiler optimizations that increase instruction-level parallelism. In *PLDI*, 1995.
20. C. McNairy et al. Itanium 2 processor microarchitecture. *IEEE Micro*, 2003.
21. S. Moon et al. An efficient resource-constrained global scheduling technique for superscalar and vliw processors. In *MICRO*, 1992.
22. V. Pai et al. Code transformations to improve memory parallelism. In *MICRO*, 1999.
23. G. Pechanek and S. Vassiliadis. The ManArrayTM embedded processor architecture. In *Euromicro*, 2000.
24. B. Rau and C. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Workshop on Microprogramming*, 1981.
25. F. Sánchez and A. González. Cache sensitive modulo scheduling. In *MICRO*, 1997.
26. C. Scheurich et al. Lockup-free caches in high-performance multiprocessors. *Journal of Parallel and Distributed Computing*, 1991.
27. H. Sharangpani et al. Itanium processor microarchitecture. *IEEE Micro*, 2000.
28. G. Sohi and M. Franklin. High-bandwidth data memory systems for superscalar processors. In *ASPLOS*, 1991.