2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture

Decoupling Loads for Nano-Instruction Set Computers

Ziqiang Huang, Andrew D. Hilton, Benjamin C. Lee
Electrical and Computer Engineering
Duke University
{*ziqiang.huang, andrew.hilton, benjamin.c.lee*}*@duke.edu*

***Abstract*—We propose an ISA extension that decouples the data access and register write operations in a load instruction. We describe system and hardware support for decoupled loads. Furthermore, we show how compilers can generate better static instruction schedules by hoisting a decoupled load's data access above may-alias stores and branches. We find that decoupled loads improve performance with geometric mean speedups of 8.4%.**

I. Introduction

Architects who design out-of-order (OoO) processors accept higher costs in return for performance. By aggressively scheduling instructions out of program order during execution, OoO processors exploit instruction level parallelism to a greater degree than in-order (IO) ones. However, dynamic scheduling requires sophisticated control and numerous bookkeeping structures—*e.g.*, reorder buffer, load-store queue, register alias table—that increase complexity, area, and power. Faced with these costs, designers may ask whether alternative (micro)architectures could deliver a significant portion of OoO's performance without its hardware overheads.

OoO derives most of its performance from a better instruction schedule, not the ability to react to dynamic events. One insightful study finds that better schedules account for 88% of OoO's advantage over IO [1]. Large windows permit aggressive re-ordering and register renaming ensures that schedules are constrained only by true dependencies. In contrast, compilers are constrained by memory aliasing and anti-/output dependencies when scheduling statically. Static and dynamic scheduling have been studied extensively [2], [3]. However, the discovery that dynamic schedules are the key to OoO performance [1], not reactive mechanisms that mitigate variable-latency operations and wrong-path execution, motivates new architectures that permit better static schedules and produce OoO-competitive performance on IO hardware.

A compiler produces better static schedules when the instruction set defines simple operations. As argued in the case for RISC [4], simplicity benefits code generation since the compiler need not find the rare opportunities to use complex instructions that bundle multiple operations. Moreover, simplicity benefits code motion since the compiler has more flexibility when re-ordering finer-grained computation and pursuing good static schedules. Thus, RISC gives the compiler fewer challenges and more opportunities.

In this paper, we note that even RISC bundles multiple basic operations into an instruction and breaking these bundles could further improve static scheduling – a strategy we call Nano Instruction Set Computing (NISC). The most promising candidates for NISC include branches and loads, instructions that disproportionately impact performance and are amenable to separation into nano-instructions. Indeed, prior work decouples branches into prediction and resolution to improve schedules [5]. In contrast, we decouple functionality in loads and perform code motion to hide their long latencies.

Scheduling decoupled loads is complicated by stores and branches. Loads cannot be hoisted before a may-alias store without jeopardizing program correctness. Loads cannot be moved across basic block boundaries without ensuring exception safety, which is frequently quite difficult. Existing techniques such as data speculation [6], [7], [8] and superblock formation [9], [10] mitigate constraints imposed by stores and branches. However, such speculative techniques incur overheads when recovering from misspeculation.

We study non-speculative approaches to circumvent the constraints posed by stores and branches, and show how decoupled loads produce better schedules. Specifically, we make the following contributions:

- We propose decoupled loads, which separate a load instruction into data access and register writeback. We describe instruction semantics and support required from the operating system and microarchitecture. (§II)
- We modify a compiler to exploit decoupled loads and produce high-quality, static instruction schedules. The compiler hoists loads' data accesses and schedules independent instructions to hide latency. (§III)
- We evaluate performance gains using cycle-level simulations for varied IO processor configurations and SPEC2006 benchmarks. Decoupled loads offer a geometric mean speedup of 8.4% (§IV)

Collectively, our results show the potential of NISC architectures. Decoupling an instruction's constituent operations produces better static schedules and brings us a step closer to OoO performance on IO design.

1063-6897/16 $31.00 © 2016 IEEE
DOI 10.1109/ISCA.2016.43

II. Decoupled Loads

Architects have long known that load instructions present performance challenges. In this paper, we seek performance by extending a RISC instruction set with decoupled loads that separate data accesses and register writebacks. To support the instruction extension, the microarchitecture must hold new state – data supplied by the cache hierarchy but not yet written to registers. Moreover, the system must accommodate new load semantics for consistency/coherence, exception handling, and context switches. With such support, decoupled loads meet key design objectives such as hiding load-to-use latency without expensive software speculation or hardware overhead.

A. Design Objectives

Hiding Latency. Decoupled loads help a compiler hide load-to-use latency. Loads, with their long and variable latencies, often occupy the critical path. However, if a load is scheduled well before the first instruction that uses its data, computation for intervening instructions hides the load latency. By separating data access and register write, a decoupled load increases scheduling flexibility. Because the register is written separately, the compiler can hoist a load's data access higher and more often than it could have hoisted a conventional load.

Although superficially similar, decoupling and hoisting a load's data access is orthogonal to data prefetching. Prefetchers bring potentially useful data up the memory hierarchy and into the data cache before a load is executed. Thus, they reduce the probability of cache misses and average load latency. Once data resides in the L1 data cache, however, decoupled loads are needed to hide load-to-use latency.

Avoiding Speculation. We decouple load functionality such that the compiler can hoist a load's data access safely and non-speculatively. When hoisting above an aliasing store, the microarchitecture supplies the correct value to the load. When hoisting across basic block boundaries, no exception is generated if the branch resolves such that the original load should not be executed.

Speculation permits aggressive code motion but requires fix-up code to correct the effects of misspeculation. For example, the Itanium employs two types of loads, speculative and advanced, that allow the compiler to schedule loads earlier [11]. Speculative loads can be moved before one or more branches. Advanced loads can be moved before stores even when the alias analysis is inconclusive. When either of these loads are used and hoisted, the compiler inserts check and branch instructions at the load's original location to detect misspeculation, and it inserts fix-up code that recovers from misspeculation when necessary.

Correction and recovery code for misspeculation introduces overheads. When the compiler misspeculates about the code path or memory aliasing, the fix-up code re-executes the computation correctly. Even when misspeculation is rare, fix-up code increases register pressure as the compiler allocates architected registers for the additional instructions. The Itanium's 128 registers might accommodate this pressure. However, a RISC architect designing an IO core has a more difficult choice – use 32 registers and spill to memory, or use 128+ registers and incur hardware costs like those for OoO physical register files.

Minimizing Hardware Overhead. We decouple loads to improve the performance of static schedules on inexpensive IO cores. Decoupled loads require modest microarchitectural support – a small structure to hold values after data access and before register write. Foreshadowing our experimental findings, a small table is sufficient to realize the performance potential for decoupled loads.

In contrast, other approaches to mitigate load latency require far more hardware. OoO's dynamic schedules perform well but require a physical register file, reorder buffer, and load/store queue. Itanium's speculative and advanced loads require many architected registers to support re-execution and fix-up code [11]. Sophisticated microarchitectures produce a load value earlier in the pipeline by enhancing the front-end with instruction pre-decode, base register caching, and fast address calculation [12]. Each of these approaches require bookkeeping, control, and recovery mechanisms with larger cost-benefit ratios than those for decoupled loads.

B. Instruction Semantics

A conventional load instruction – `load rD, I(rA)` – has three significant pieces:

- **Memory Hierarchy Access:** The first piece of a load instruction is the computation of its effective address (`I + rA`), and the memory hierarchy access. Access includes translation from virtual to physical address, the L1 data cache read, etc.
- **Ordering:** The second piece of a load instruction is its ordering relative to other memory operations. Ordering specifies whether the load comes logically before or after a store from the same or another thread.
- **Register Write:** The third piece of a load instruction is writing the value to destination register `rD`.

Conventional loads constrain scheduling as all three pieces are bundled together in a single instruction. In contrast, the separation of these pieces into multiple instructions gives greater flexibility. We focus on separating memory hierarchy access from ordering. Doing so allows a load to perform parts of its work with longer latencies (*i.e.*, load-to-use latency of a data cache hit or the even longer latencies of a miss) prior to the point at which it must be ordered relative to other instructions. These separated instructions are linked by a new architectural identifier that we call the *load tag*.

We consider splitting the load into two instructions. Note that a load could be divided into three instructions, but the additional split benefits performance only under heavy register pressure:

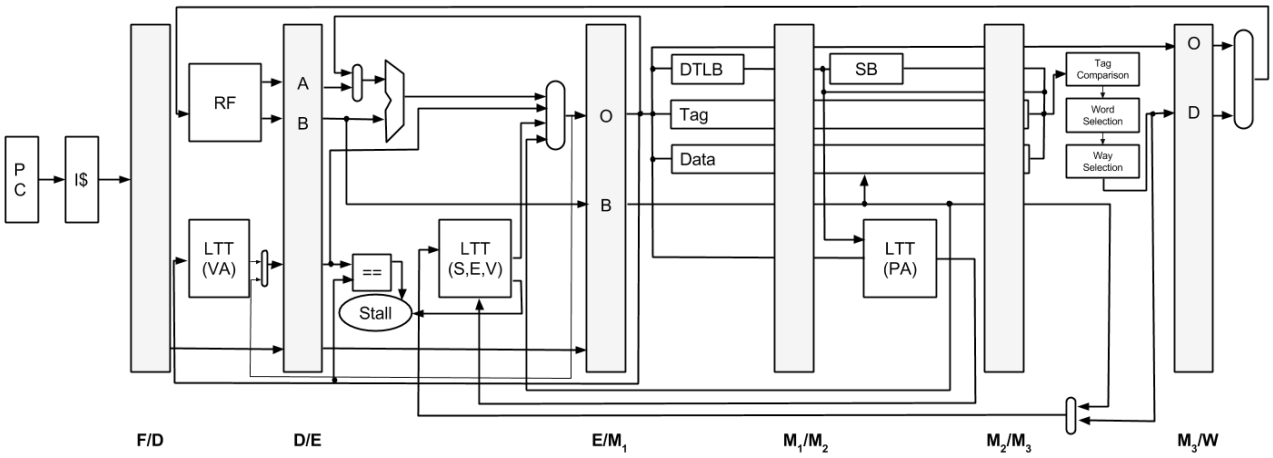

Figure 1. In-order pipeline extended with load tag table (LTT).

- **Data Access [`load.D$ ltD, I(rA)`]:** Data access behaves much like a conventional load except that it places the contents at memory address `I(rA)` into the destination load tag `ltD`. If the instruction faults, the exception is raised only when the load is ordered by a subsequent instruction that uses the same load tag.
- **Order and Write [`load.wb rD, ltA)`]:** This instruction *orders* the load and writes the result back to the register file. Writing the result copies the value from the load tag `ltA` into the destination register `rD`. If this value is not yet available, the instruction will stall. Order and write preserves load semantics, causing the decoupled load to behave as if the conventional load were written at this point in program order.

Translating a conventional load into a decoupled one is trivial.[1] The compiler can freely hoist the data access instruction above stores and branches to hide load-to-use latency. However, the compiler must ensure that the ordering instruction for the load remains at the conventional load's original position or at some place the compiler could have safely moved the conventional load within the program.

```
load r2, 0(r1)  →  load.D$ lt0, 0(r1)
                   load.wb r2, lt0
```

C. Microarchitectural Support

We present an overview of a microarchitecture that supports decoupled loads. The primary microarchitectural addition is a table to maintain the state of decoupled loads between its `load.D$` and the `load.wb` instructions. We call this structure the Load Tag Table (LTT) as it has one entry per load tag. Each entry includes status bits, exception information, addresses, and values.

[1]Loads from location declared `volatile` cannot be split.

Load Tag Table. Two status bits encode the state of an entry. 00 indicates that the entry is invalid, which means that no `load.D$` has been executed since the `load.wb` completed for the previous load that used the entry. 01 indicates that the memory hierarchy access is in progress. 10 indicates that the memory hierarchy access has been completed, but subsequently invalidated by coherence. Finally, 11 indicates that the entry holds valid data.

Each entry holds exception information that tracks any exception detected for a load that has not yet been ordered. For example, `load.D$` may have encountered an invalid virtual address. Each entry tracks the virtual and physical addresses specified and read by `load.D$`, respectively. Finally, the entry holds the value retrieved by `load.D$`.

Figure 1 shows a 7-stage, in-order pipeline extended with an LTT. The LTT is physically divided into three parts: the virtual address (VA) in the decode stage; the status, exception and value (S,E,V) in the execute stage; and the physical address (PA) in the second memory stage. Decoupled load instructions, `load.D$` and `load.wb`, index into the table with their load tag. The structure that holds physical addresses is a content-addressable memory (CAM), which is searched by store instructions and coherence invalidations.

Operation and Example. We describe LTT operation with a simple example. The compiler schedules a store between a load's data access and register writeback, highlighting the load's interaction with a may-alias store.

```
load.D$ lt0, 4(r2)
st 4(r4), r3
load.wb r3, lt0
```

When `load.D$` enters the execute stage (E), it sets the corresponding LTT entry's status to 01. Then, `load.D$` updates the LTT entry with its virtual address in the first memory stage (M_1) and with its physical address in the

second memory stage (M_2). After retrieving its data from the cache hierarchy, `load.D$` updates the LTT entry with its value in the third and last memory stage (M_3). Finally, `load.D$` sets status to 11 during normal execution and to 10 if an exception arises.

When `store` enters the second memory stage (M_2), it searches the LTT's CAM. If it finds an LTT entry with the same address, `store` updates the entry's value in the third memory stage (M_3). Finally, when `load.wb` enters the execute stage (E), it retrieves the entry's value and sets status to 00. Thus, the LTT ensures correctness in the presence of aliasing stores while potentially shrinking load-to-use latency from 4 cycles to 1.

Suppose `load.wb` executes and reads an entry with status other than 11. If status is 00, the entry is invalid and an invalid-instruction exception is raised. If status is 01, the data access is in progress and `load.wb` stalls until the value returns from the memory hierarchy. If status is 10, coherence invalidation requires `load.wb` to re-access the memory hierarchy, as if it were a conventional load, using the virtual address stored in the LTT.

Note that `load.wb` reads the LTT value two stages earlier than `store` searches the LTT for possible aliases. Even with bypassing, a `load.wb` that immediately follows a `store` risks reading an outdated value from the LTT in the execute stage (E). A conservative microarchitecture could stall if any store is in flight.

An efficient alternative compares page offsets in virtual addresses. The `store`'s offset is available after address generation and the `load.wb`'s is read from the LTT during decode. If offsets differ, `load.wb` and `store` may proceed in parallel. An aggressive alternative compares virtual addresses early to bypass from `store` to `load.wb`. In the event of synonyms (*i.e.,* same physical but different virtual addresses), the `load.wb` is squashed and re-executed.

Discussion. The LTT has some similarity to the load queue in an OoO processor. Both structures hold physical addresses in a CAM. Stores and invalidations must search both. However, LTTs differ in several significant ways.

First, in the LTT, a CAM match either updates the entry's value to reflect the value stored or changes its status to reflect an invalidation. In a load queue, a CAM match flushes instructions because a load executed at the wrong time. Second, the LTT holds only entries for decoupled loads and is much smaller than a load queue. Third, the compiler assigns LTT entries during register allocation whereas the load queue is a FIFO structure.

Although one might view the LTT as additional "registers" to mitigate register pressure, the performance benefits of decoupled loads cannot be achieved by simply increasing the register file size by the number of LTT entries. From the compiler's perspective, more registers reduces spilling and mitigates anti/output dependencies. But they cannot help loads circumvent constraints from stores and branches.

D. System Support

Decoupled load semantics are defined naturally by its two instructions for data access (`load.D$`) and register writeback (`load.wb`). Together, these instructions provide the system enough information to order loads, handle exceptions, and switch contexts.

Load Ordering. The system must order decoupled loads relative to loads and stores from other cores. The `load.wb` is the ordering point of the decoupled load. Because the compiler ensures that `load.wb` occupies the original load's location in the program, the decoupled load completes at the same point in time as a conventional load would have. Thus, `load.D$` is not visible to other cores until `load.wb` commits, guaranteeing the decoupled load's correctness under any consistency model.

However, the core responds to invalidation differently to accommodate interaction between decoupled loads and stores from other cores. When a load is decoupled, another core's store could be ordered after its data access but before its register write. Thus, invalidation needs to associatively search the LTT just as invalidation is required to search the load queue in OoO cores. If a matching address is found, invalidation updates the LTT entry's status, forcing the subsequent `load.wb` to re-access the memory hierarchy.

Exception Handling. When a decoupled load's `load.D$` produces an exception, the system defers handling until its ordering point at `load.wb`. With deferred exception handling, decoupled loads provide the same semantics as conventional loads. Exceptions are precise with respect to the register write. Moreover, the compiler can hoist `load.D$` above branches without risk of handling exceptions from unexecuted code paths unnecessarily.

Context Switching. Load tags, like registers, are part of a process or thread's context. Recall that the LTT contains valid bits, addresses, and values. When a thread is context switched out, the operating system (OS) saves each valid LTT entry's virtual address, but not value, to memory.

When the thread is context switched in, the OS restores each valid LTT entry by re-loading the value from its saved address. To accomplish this reload, the OS re-executes `load.D$` for each decoupled load in flight. Re-execution is necessary after a context switch because other threads may have modified values residing at LTT-held addresses. Re-execution may encounter page faults if the context switch paged out LTT-held addresses. Page faults, like exceptions, are deferred to the decoupled load's ordering point.

III. Compiler Support

Compiler support is essential when using decoupled loads to produce high-performance schedules. First, the compiler must determine which loads to decouple into data access and register write. Naïvely decoupling every load would increase pressure on the load tag table, which is small to

```
void saxpy (float *dest, float *x,
            float *y, float a, int n) {
  for (int i=0; i < n; i+=2) {
    dest[i] = a*x[i]+y[i];
    dest[i+1] = a*x[i+1]+y[i+1];
  }
}
```

Figure 2. SAXPY code example

ensure single-cycle access, and constrain performance. Furthermore, naïve approaches would increase instruction cache pressure and cause the program to execute more dynamic instructions, harming performance and power efficiency.

Second, the compiler must hoist data access instructions, often above may-alias stores and branches, to hide load-to-use latency. We first describe scenarios in which scheduling benefits from decoupled loads. Then, we propose two compiler scheduling policies that exploit decoupled loads to improve performance.

A. Hoisting Over (May-)Aliasing Stores

Alias analysis attempts to determine whether two pointers to memory refer to the same address. Given a query with two pointers, the analysis responds with one of three possible answers—must, may, or no alias. The compiler uses alias analysis conservatively and does not schedule conventional loads above may-alias stores. With decoupled loads, the compiler circumvents the constraints posed by memory aliasing. We show how data access in decoupled loads can be hoisted above may-alias stores for representative functions.

Scheduling Example. The compiler makes assumptions about machine parameters during instruction scheduling. In our example, the compiler considers a two-wide, in-order machine with two ALUs and one load/store unit. The load-to-use latency is four cycles, which optimistically assumes loads hit in the L1 cache and provides a challenging scenario for decoupled loads. Multiplication requires four cycles and all other instructions require one cycle. Each functional unit is fully pipelined.

We consider single-precision, scalar multiplication and vector addition (SAXPY). Figures 2–3 present the source code and resulting static schedule. Instructions within the same loop iteration are serialized by true data dependencies such that these instructions cannot be reordered or scheduled in the same cycle. In contrast, instructions across loop iterations are not constrained by true dependencies.

Modern compilers extract instruction-level parallelism across iterations by unrolling the loop. However, loop unrolling by a factor of two fails to improve SAXPY performance. The compiler fails to find parallelism in the unrolled loop because it cannot determine whether the store instruction at line 5 aliases the loads in lines 6-7. As a result, the compiler conservatively specifies a may-alias dependence

```
Loop:
1  ld   r9,0(r4)
2  ld   r10,0(r5)
3  mul  r9,r9,r6
4  add  r9,r9,r10
5  st   0(r3),r9
6  ld   r9,4(r4)
7  ld   r10,4(r5)
8  mul  r9,r9,r6
9  add  r9,r9,r10
10 st   4(r3),r9
11 addi r3,r3,8
12 addi r4,r4,8
13 addi r5,r5,8
14 addi r8,r8,2
15 bne  r8,r7,Loop
```

Cycle	Issue-1	Issue-2
1	1	14
2	2	
3-4		
5	3	
6-8		
9	4	
10	5	
11	6	12
12	7	13
13-14		
15	8	
16-18		
19	9	
20	10	11
21	15	

Figure 3. SAXPY schedule with conventional loads

```
Loop:
1  ld   r9,0(r4)
2  ld   r10,0(r5)
3  mul  r9,r9,r6
4  add  r9,r9,r10
5  st   0(r3),r9
6  ld.D$ lt0,4(r4)
7  ld.D$ lt1,4(r5)
8  ld.wb r9,lt0
9  ld.wb r10,lt1
10 mul  r9,r9,r6
11 add  r9,r9,r10
12 st   4(r3),r9
13 addi r3,r3,8
14 addi r4,r4,8
15 addi r5,r5,8
16 addi r8,r8,2
17 bne  r8,r7,Loop
```

Cycle	Issue-1	Issue-2
1	1	16
2	2	
3	6	
4	7	
5	3	
6-8		
9	4	
10	5	
11	8	9
12	10	14
16	11	15
17	12	13
18	17	

Figure 4. SAXPY schedule with decoupled loads

between the store and its following loads, producing a static schedule that spans 21 cycles.

Decoupled loads separate data access from register write, allowing the compiler to hoist the data access. When splitting a conventional load instruction, which it assumes requires four cycles, the compiler generates a three-cycle `load.D$` and a one-cycle `load.wb`; most of the load's latency is attributed to data access. The compiler uses idle issue slots to schedule data accesses in lines 6-7 earlier, which reduces load-to-use latency from four cycles to one. Because stores check and update the load tag table, the schedule ensures program correctness regardless of aliasing. The improved new static schedule spans only 18 cycles, as shown in Figure 4.

SAXPY represents a broader class of codes for which alias analysis must be conservative and thus restricts the compiler's ability to schedule code efficiently. We take SAXPY as an example because it is simple and easy to understand. More generally, SAXPY illustrates code in which read-modify-write instructions are performed on data

```
void P7EmitterPosterior(
  int L, struct plan7_s *hmm,
  struct dpmatrix_s * forward,
  struct dpmatrix+s * backward,
  struct dpmatrix_s *mx) {
  ...
  for (i = L; i>=1; i++) {
    mx->xmx[i][XMC]=
    forward->xmx[i-1][XMC]
    + hmm->xsc[XTC][LOOP]
    + backward->xmx[i][XMC]
    -sc;
  ...
  }
}
```

Figure 5. SPEC Hmmer Code Example.

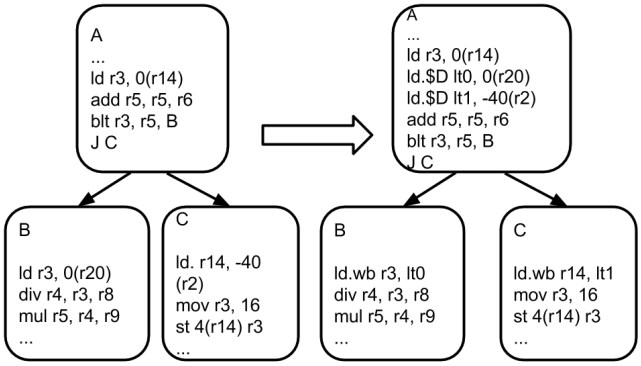

Figure 6. Hoisting decoupled loads over branch (shared predecessor).

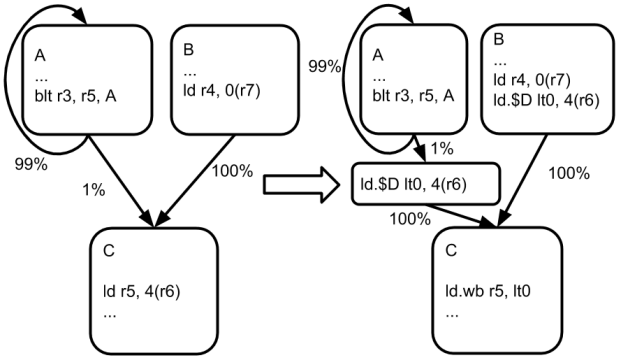

Figure 7. Hoisting decoupled load over branch (multiple predecessors).

structures indexed with variables. These codes are prevalent and Figure 5 shows a similar loop pattern from hmmer in the SPEC2006 benchmark suite.

B. Hoisting Over Branches

Compilers have difficulty hoisting a conventional load instruction over a branch, primarily because the branch could resolve in another direction. If a load is hoisted from a basic block in an unexecuted code path, the load's destination register would be written with the wrong value. Moreover, the load may cause an exception and trigger a handler unnecessarily. For these reasons, Itanium's advanced loads require fix-up code and propagate a token that tracks deferred exceptions to each instruction that depends on the advanced load's value [11]. We show how decoupled loads can be hoisted above branches non-speculatively and without fix-up code.

Scheduling and Examples. Figure 6 presents a simplified control flow graph from the spec_random_load function in bzip2, a benchmark from the SPEC2006 suite. Basic block A has two successors, B and C, both of which start with a load. The compiler cannot hoist these conventional loads above A's branch because it cannot determine the branch direction.

Decoupled loads enable a new schedule. The compiler decouples each load into `load.D$` and `load.wb` instructions. It hoists both `load.D$` instructions over A's branch while ensuring that both `load.wb` instructions remain in their respective positions. Because `load.D$` places data in load tags and does not fault until ordered by a `load.wb`, the transformation ensures program correctness regardless of branch direction. By hoisting `load.D$` above branches, the compiler overlaps latency of B and C's loads with the latency of A's load.

Naïvely decoupling loads and hoisting data accesses above branches may harm performance. Figure 7 presents a scenario in which basic block C has two predecessors, A and B. The compiler may wish to decouple C's load and hoist its data access into B to overlap the latency with the load in B. However, it must hoist data access into both predecessors to ensure a valid value for the writeback regardless of control flow. Unfortunately, A is a loop body and its branch into C has a very low bias. Hoisting C's data access into A may cause A to execute more slowly due to more dynamic instructions. Since A is executed multiple times, overall performance may suffer.

The compiler can implement one of two solutions to this scenario. First, the compiler can be conservative and refrain from decoupling C's load instruction. Alternatively, the compiler can create a new basic block between A and C that contains only the `load.D$` instruction, which has several advantages. When control flows from A to C, computation is correct and `load.D$` does not affect A's performance. When control flows from B to C, the hoisted `load.D$` improves performance.

In our implementation, the compiler determines its handling of decoupled loads and branches based on static analysis. Estimating bias is easier in some cases (*e.g.* loops)

and more difficult in others. In future, decoupled loads could be generated and scheduled with profiles and dynamic binary translation. Dynamic frameworks might enable a panoply of more precise optimizations.

C. Scheduling Policy

Instruction scheduling is a critical stage in the compiler's code generation pipeline. The compiler tries to reorder instructions to increase instruction-level parallelism and reduce structural hazards. Even when the compiler has a holistic view of the program, it typically reorders instructions within a basic block and not across them. Scheduling instructions across basic blocks is difficult because it risks executing instructions from the wrong code path.

Scheduling for Conventional Loads. List scheduling is a common algorithm used by most modern compilers for scheduling instructions within a basic block. The scheduler constructs a data dependency graph (DDG) in which nodes represent instructions, edges represent instruction dependencies, and edge weights represent instruction latencies.

For a given basic block, Algorithm 1 shows a simplified procedure for list scheduling. The algorithm first builds DDG and then uses the graph to identify instructions that are ready to execute. Whether an instruction is ready at a given cycle depends on when its predecessors were scheduled and their latencies. If no instruction is ready, the algorithm simply increments the cycle count and checks again. Thus, the algorithm repeatedly selects ready instructions, inserts them into the schedule, and updates the graph. The process continues until all instructions are scheduled and the graph is empty.

Algorithm 1 Scheduling with Conventional Loads

```
1: procedure SCHEDULE(BASICBLOCK *B)
2:   build DDG
3:   while B has unscheduled instructions do
4:     if no instruction ready at this cycle then
5:       goto next
6:     pick instructions
7:     release successor instructions
8: next:
9:     cycle++
```

Extensions for Decoupled Loads. We extend list scheduling to use decoupled loads in Algorithm 2. The new scheduling algorithm discovers new opportunities to exploit instruction- and memory-level parallelism. When the algorithm cannot find a ready instruction in a given cycle, it will expand its search by decoupling a load and determining whether its data access instruction can be scheduled.

The scheduler has several strategies for decoupling loads and hoisting data accesses. For any given cycle, the scheduler considers decoupled loads only when no other instruction is ready for scheduling and the load/store unit is idle.

Algorithm 2 Scheduling with Decoupled Loads – Bubble

```
1: procedure SCHEDULE(BASICBLOCK *B)
2:   build DDG
3:   while B has unscheduled instructions do
4:     if no instruction ready at this cycle then
5:       if available Ld/St Unit then
6:         if exist load only constrained by may-alias then
7:           goto decouple
8:         if exist independent load in successor blocks then
9:           goto decouple
10:        goto next
11: decouple:
12:        decouple load into load.D$ and load.wb
13:        mark load.D$ as ready
14:        continue
15:    pick instructions
16:    release successor instructions
17: next:
18:    cycle++
```

To find an instruction for this "bubble" cycle, the scheduler seeks to decouple loads from the current basic block and, if that fails, to decouple loads from the block's successors.

First, within the current block, the scheduler searches for loads that are constrained only by a may-alias dependence. The scheduler can decouple the load, hoist its data access instruction above the may-alias store, and rely on the load tag table to supply the value for register write. Second, the scheduler searches for independent loads in the current block's successors. The scheduler can decouple the load and hoist its data access instruction above the branch.

When a candidate load is found, the compiler decouples the conventional load into `load.D$` and `load.wb` instructions. Then, the compiler updates the graph with dependencies and latencies for the new instructions. To ensure that the ordering point remains in the original load's position, `load.wb` inherits all dependency information from the original load plus an additional predecessor – `load.D$`. If the scheduler assumes an n-cycle latency for the original load, it assumes $n-1$-cycle latency for `load.D$` and one-cycle latency for `load.wb`. With the new graph, the scheduler attempts to fill bubbles in the schedule.

Algorithm 3 offers a more aggressive approach to scheduling decoupled loads. Instead of decoupling loads only in the presence of bubble cycles during scheduling, the algorithm seeks candidate loads for decoupling immediately after building the data dependence graph. Doing so adds flexibility because the compiler may schedule the entire basic block differently and, for example, prioritize the `load.D$` over other low-latency instructions. However, such an aggressive policy requires new, comprehensive compiler heuristics to determine when decoupling loads is worthwhile.

Algorithm 3 Scheduling with Decoupled Loads – Early

```
1:  procedure SCHEDULE(BASICBLOCK *B)
2:    build DDG
3:    for each instruction do
4:      if is load instruction then
5:        if no predecessor instruction then
6:          decouple into load.D$ and load.wb
7:          for each of B's predecessor block P do
8:            hoist load.D$ to the end of P
9:            mark P to be rescheduled
10:       if constrained by may-alias then
11:         decouple into load.D$ and load.wb
12:         add load.wb's data dependency edges to load.D$
13:   (rest same as Algorithm 1)
```

D. Load Tag Allocation

The compiler allocates and frees load tags, which are required when decoupling loads, in a proccess that closely resembles register allocation. When the compiler decides to decouple a load in the scheduling stage, it assigns a virtual load tag to the pair of `load.D$` and `load.wb` instructions, much like a virtual register number for traditional register allocation. The `load.D$` opens the liveness of the corresponding load tag whereas the `load.wb` kills it.

When the compiler enters the register allocation stage, it allocates load tags just as it allocates registers. Because load tags look very much like registers, we can take advantage of the liveness analysis and register allocation frameworks that already exist in the compiler to support decoupled loads.

The allocator handles load tag pressure differently than register pressure. When the allocator encounters register pressure, it spills register contents to memory. In contrast, when the allocator encounters load tag pressure, it reverses the decision to decouple the load and couples the `load.D$` and `load.wb` to produce a conventional load.

Reverting to a conventional load is preferable to spilling, which generates a new pair of store/load instructions and defeats the purpose of decoupling loads (*i.e.*, hiding load latency). Register allocation follows scheduling in a conventional compiler pipeline, but the compiler must re-schedule a basic block if allocation reverts a decoupled load into a conventional one. Rescheduling incurs no additional overhead since the compiler requires a post-register-allocation scheduling pass to accommodate normal register spills.

IV. Experimental Evaluation

We extend the OpenRISC instruction set with decoupled loads. To generate code with decoupled loads, we extend the Machine Instruction Scheduler in LLVM 3.5 [13] with Algorithms 2-3. The scheduler searches for two types of conventional loads that could be decoupled to improve performance. First, it seeks loads in the same basic block that have been constrained by may-aliasing stores. Second, it seeks independent loads in successor blocks that could be hoisted above branches depending on the static analysis of branch biases.

Structure	Configuration
Branch Predictor	GShare, 8KB table, 13 history bits, 4K-entry BTB, 64-entry RAS
Machine Width	Varied – 2/4
Functional Units	LD/ST varied – 1/2, INT ALU 2×, FP-ALU 1×
Load Tag Table	Varied – 8/16/32 entries
L1 Caches	8-way 32KB L1-D$, 4-way 32KB L1-I$, 64B lines, 4-cycle latency
L2 Cache	16-way 256KB, 12-cycle latency
L3 Cache	32-way 4MB, 25-cycle latency
Miss Handling	8-entry MSHR
DRAM	140-cycle latency

Table I
MACHINE MODEL.

Machine Model. To understand the performance of decoupled loads, we perform cycle-level simulation by extending the OpenRISC architectural simulator Or1ksim with an in-order timing model that includes dependency checking, superscalar support, a three-level cache hierarchy and a branch predictor.

Table I summarizes machine parameters that are used in the simulations. LLVM also uses some of these parameters – for example, the number of functional units, instruction latency – to guide static scheduling. Load latency varies dramatically across applications and even across specific load instructions within the same application. Yet the scheduler requires a single static value that estimates load latency. Our compiler schedules loads using L1 hit latency.

Benchmarks. We evaluate eight integer benchmarks in SPEC2006. We also experiment decoupled loads with floating-point benchmarks in SPEC2006, but are unable to produce meaningful results because OpenRISC does not support double-precision arithmetic in its 32-bit architecture. Double-precision arithmetic is implemented in software, which generates many library calls that restrict code motion for decoupled loads; the compiler cannot hoist loads above a function call.

The standard approaches in performance measurement sample workloads and run a limited number of representative instructions. Yet sampling is difficult when comparing different ISAs. Simulating X instructions after fast-forwarding Y instructions could measure performance in very different parts of the workload as code generation and scheduling

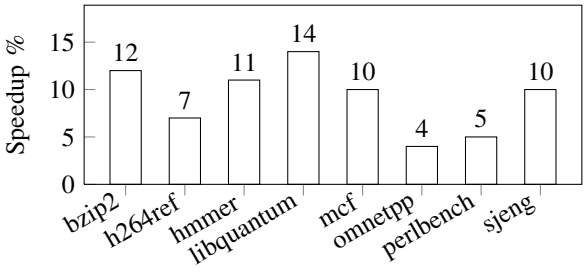

Figure 8. Performance speedup from decoupled loads over baseline with conventional loads.

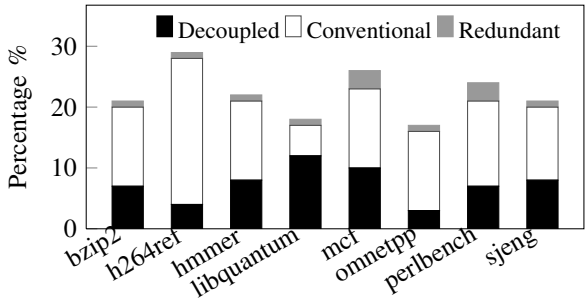

Figure 9. Load breakdown, relative to number of dynamic instructions.

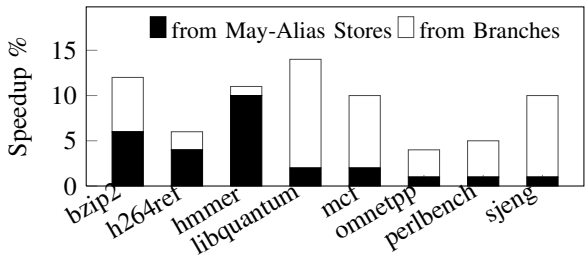

Figure 10. Contributors to decoupled load performance.

could produce very different dynamic instruction counts. For this reason, we run each benchmark to completion with its TRAIN input set and measure end-to-end performance speedup.

Evaluation Strategy. Graphs show results from a 4-wide, 2-LD/ST, 32-entry load tag table configuration unless otherwise specified. We present the geometric mean of speedups obtained over a baseline in-order machine. We first supply insight into the sources of performance by showing how often loads are decoupled and hoisted. We further differentiate decoupled loads hoisted above may-aliasing stores versus those hoisted above branches. We assess performance sensitivity to machine width, prefetching, scheduling policy and microarchitectural resources. Finally, we discuss side effects of decoupled loads.

A. Performance Analysis

Figure 8 presents performance gains when generating code with decoupled loads instead of conventional loads. Overall, decoupled loads improve performance with geometric mean speedup of 8.4% and a max speedup of 14%. Performance gains vary from benchmarks as six out of the eight benchmarks report substantial speedups, ranging from 7% - 14%, while the other two benefit significantly less.

Instruction Mix. Figure 9 presents the number of loads relative to number of dynamic instructions. The black bar denotes the number of useful decoupled loads, measured by the number of completed `load.wb` instructions. The white bar denotes the number of conventional loads that are not decoupled. The gray bar denotes the number of redundant loads, measured by the number of `load.D$` instructions from unexecuted code paths.

Load instructions often comprise about 20% of the total. The compiler can decouple a large percentage of these loads. For example, loads comprise 17% of libquantum's instruction mix and 70% of these loads are decoupled. Similarly, loads comprise 23% of mcf's instruction mix and 43% of these are decoupled, which corresponds to a large number of loads in absolute terms. The percentage of decoupled loads indicates how often the compiler uses the new instructions and correlates with performance gains.

Although the compiler decouples loads and hoists data access instructions aggressively, it rarely creates extra work for the application. Unnecessary and redundant data access instructions comprise only 1.4%, on average, of the total. Redundant data access instructions are those that are hoisted from basic blocks in unexecuted code paths. In these scenarios, the application executes a `load.D$` instruction but fails to execute the corresponding `load.wb` instruction. Thus, redundant loads are a measure of wasted work. Figure 9 indicates that redundant loads are rare and the compiler generates efficient code.

Stores and Branches. We hoist decoupled loads above may-aliasing stores and branches with two different mechanisms. Hoisting above stores requires a load-tag table that forwards updated values to decoupled loads. Hoisting above branches requires bias analysis and new basic blocks. We find that applications are diverse and require support for both types of instruction re-ordering.

Figure 10 illustrates contributions to performance when hoisting a load's data access above stores and branches. Although all benchmarks benefit from both types of code motion permitted when decoupling loads, the extent of these benefits vary. For example, 90% of hmmer's performance gain comes from hoisting loads over may-aliasing stores whereas benchmarks like libquantum and mcf benefit mostly hoisting loads over branches. In contrast, bzip2 benefits equally from both. Thus, both scheduling techniques are required to realize the full potential of decoupled loads.

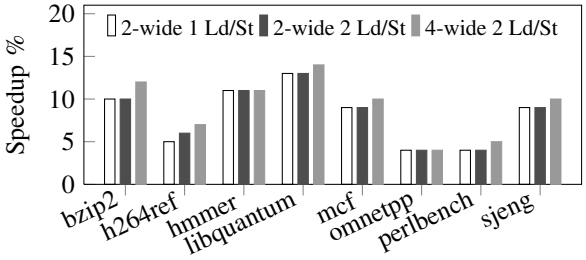

Figure 11. Performance sensitivity to machine width.

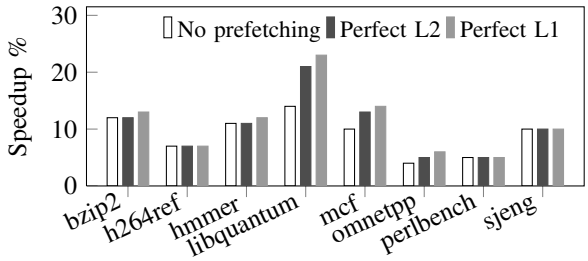

Figure 12. Performance sensitivity to prefetching.

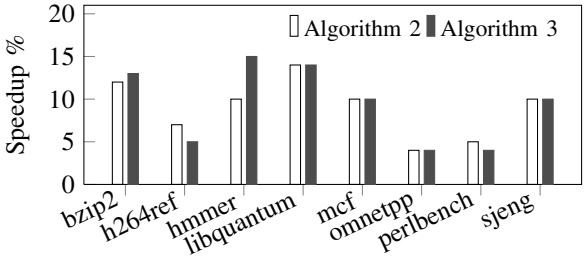

Figure 13. Performance sensitivity to scheduling policy.

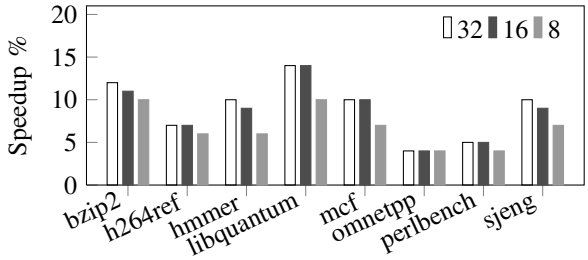

Figure 14. Performance sensitivity to load tag table (LTT) size.

B. Sensitivity Analysis

Machine Width. Figure 11 illustrates performance sensitivity to an in-order machine's issue width and number of load/store units. Decoupled loads perform better in wider machines. Given a wider machine, the compiler is more likely to find idle slots and "bubbles" in the static schedule, which trigger a search for loads to decouple and data access instructions to hoist. Thus, the compiler can decouple loads more aggressively in a wider machine. However, performance differences are modest as the compiler is limited by the number of candidate loads in the application.

Prefetching. Figure 12 shows the speedup obtained by decoupled loads over a baseline with no prefetching, with a perfect L2 (every L1 miss served by L2), and with a perfect L1 (every access hits L1). The perfect caches model the effect of ideal prefetchers on decoupled loads.

We find that prefetchers do not degrade benefits from decoupled loads. On the contrary, for benchmarks with relatively bad data cache behavior, such as mcf and libquantum, bundling decoupled loads with prefetching increases speedups. While decoupled loads can potentially overlap cache miss latency, consecutive cache misses are rare because most benchmarks are characterized by regular memory access patterns and good data locality. The majority of decoupled loads' benefits come from hiding the load-to-use latency of a cache hit.

Scheduling Policy. Figure 13 compares the intuitive policy in Algorithm 2, which hides load latency in bubble cycles, with the more aggressive policy in Algorithm 3. For benchmarks with long load latencies and prevalent bubbles, such as mcf, decoupling loads in the presence of bubbles or immediately after building the dependence graph makes little difference. A few benchmarks perform better with Algorithm 3. These benchmarks tend to have more instruction-level parallelism, providing the compiler more opportunities to prioritize decoupled loads over low-latency instructions. However, several benchmarks perform worse as the compiler generates more redundant load.D$ instructions when branches are unbiased.

Load-Tag Table Size. A decoupled load that is in flight requires a load tag, which links the data access instruction to its corresponding register write instruction. Our analysis thus far assumes 32 entries in the load tag table, a conservative configuration that evaluates compiler effectiveness when resources are abundant. In practice, however, we balance performance gains against load-tag table size.

Figure 14 evaluates performance for a range of LTT sizes and shows that only a few entries are sufficient. Load tag pressure is usually much smaller than conventional register pressure for two reasons. First, load tags are used only for decoupled loads. Second, load tags have a very small liveness range. As we vary LTT size from 32 to 8 entries, we find that 16 load tags are sufficient to capture 95% of the performance gains from decoupled loads. When we reduce the number of tags from 16 to 8, hmmer, libquantum and mcf performance suffers. This sensitivity analysis satisfies a key design objective, minimal microarchitectural support, and motivates future work in adapting the processor datapath for decoupled loads.

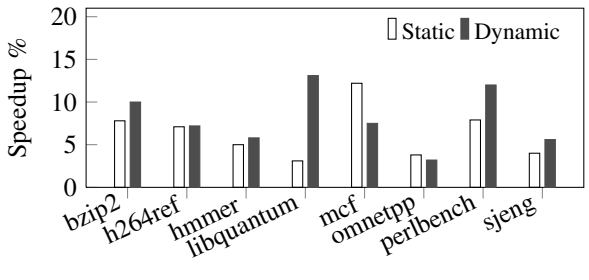

Figure 15. Code size and overhead.

C. Side Effects

Figure 15 shows instruction overheads from decoupled loads. Decoupled loads separate a load instruction into two parts – data access and register write – and may increase code size. For our benchmarks, decoupled loads increase static and dynamic code size by an average of 5.8% and 7.4%, respectively.

Increases in static code size could reduce the instruction cache's hit rates, harming performance. However, we observe negligible performance degradation. SPEC2006 benchmarks behave well from the instruction cache's perspective [14]. Moreover, performance penalties from instruction cache misses are small for in-order cores, which suffer from head-of-line blocking frequently. Misses do not harm performance when instructions are off the critical path or are not immediately ready for execution upon entering the window [5].

Increases in dynamic code size could increase contention for issue slots. In practice, however, the compiler decouples loads to fill slots that would otherwise stall the pipeline. Only the data access part of redundant loads contend with useful instructions for issue slots. But we find that the number of redundant loads is very small.

V. Related Work

Branches perform multiple operations in a single instruction. By decomposing a branch into prediction and resolution instructions, control flow transfers can be hoisted above branch resolution [5]. Breaking both branches and loads into their constituent parts will produce even better schedules.

Load Latency. Microarchitectural mechanisms have been proposed to tolerate cache miss latencies [15], [16], [17], [18], [19]. CFP uses a slice buffer to drain the missed load and its dependent instructions, freeing issue queue and register file resources [17]. iCFP adapts CFP for in-order pipelines, unblocking latches and allowing independent instructions to execute [18]. SLTP and Multipass Pipelining present other implementations of non-blocking schemes [20], [21]. State-of-the-art iCFP achieves reach 57% of OoO performance with IO design.

Prefetching reduces cache miss latency. Software prefetching inserts explicit prefetch instructions for memory references that are likely to miss in cache [22]. Compilers can detect memory access patterns and tune for varied latencies [23]. Prefetching is especially effective given regular memory access patterns [24], [25], but has also been applied to pointer-based data structures [26], [27]. Prefetching does not hide load-to-use latency once data is in the L1 cache. Indeed, decoupled loads could improve performance even if the system were to use an ideal prefetcher.

The Decoupled Access/Execute Architecture (DAE) [28] decouples operand access and execution with two instruction streams that comunicate via queues. DAE is orthogonal to decoupled loads since the former is a microarchitectural implementation and the latter is an architectural extension. However, in a direct comparison with decoupled loads, DAE requires much more complex hardware and its loads remain serialized by stores and branches.

Instruction Scheduling. Dynamic optimization reschedules code to reflect runtime behavior, adding new code to the application address space [29], [30] or into a hardware cache[31], [32]. rePLay and Region Slip support dynamic optimization and allow region schedules to overlap [33], [34]. Each of these techniques require extensive hardware support (*e.g.*, frame constructor, optimization engine and scheduler, frame cache, recovery mechanism).

Schedulers could perform better when given broader scope. Trace scheduling is an early solution to the global microcode optimization problem [35]. Similar approaches include Superblock and Hyperblock formation[9], [36]. These profile-driven techniques are data dependent. They also require mechanisms to support prediction and recovery after misspeculation, either with fix-up code or checkpoint recovery.

IA-64 provides "advanced loads" and "speculative loads" which allow the compiler to schedule a load before one or more prior stores and branches [11]. These instructions place a check instruction at the original load's location to detect misspeculation and branch to fix-up codes. Our scheme differs because the compiler transforms code to execute the data access portion of the load non-speculatively.

VI. Conclusion

Current instruction set architectures bundle multiple operations into one instruction, which prevents compilers from aggressively re-ordering instructions. We propose decoupled loads to separate data accesses and register writes. Decoupled loads enable better static schedules by allowing compilers to hoist data access above may-alias stores and branches, two major barriers for code motion on conventional loads. Decoupled loads require modest system and microarchitectural support and improve performance by enabling better static schedules.

ACKNOWLEDGEMENTS

This work is supported by the National Science Foundation under grants CCF-1149252 (CAREER), CCF-1337215 (XPS-CLCCA), SHF-1527610, and AF-1408784. This work is also supported by STARnet, a Semiconductor Research Corporation program, sponsored by MARCO and DARPA. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of these sponsors.

REFERENCES

[1] D. McFarlin, C. Tucker, and C. Zilles, "Discerning the dominant out-of-order peformance advantage: Is it speculation or dynamism?" in *ASPLOS*, 2013.

[2] P. Chang, W. Chen, S. Mahlke, and W. Hwu, "Comparing static and dynamic code scheduling for multiple-instruction-issue processors," in *MICRO*, 1991.

[3] C. Love and H. Jordan, "An investigation of static versus dynamic scheduling," in *ISCA*, 1990.

[4] D. Patterson and D. Ditzel, "The case for the Reduced Instruction Set Computer," *SIGARCH Computer Architecture News*, 1980.

[5] D. McFarlin and C. Zilles, "Branch Vanguard: Decomposing branch functionality into prediction and resolution instructions," in *ISCA*, 2015.

[6] X. Dai, A. Zhai, W. Hsu, and P. Yew, "A general compiler framework for speculative optimizations using data speculative code motion," in *CGO*, 2005.

[7] J. Lin, T. Chen, W. Hsu, P. Yew, R. Ju, T. Ngai, and S. Chan, "A compiler framework for speculative analysis and optimizations," in *PLDI*, 2003.

[8] J. Dehnert, B. Grant, J. Banning, R. Johnson, and T. Kistler, "Using speculation, recovery, and adaptive retranslation to address real-life challenges," in *CGO*, 2003.

[9] W. Hwu, S. Mahlke, W. Chen, P. Chang, N. Warter, R. Bringmann, R. Ouellette, R. Hank, T. Kiyohara, G. Haab, J. Holm, and D. Lavery, "The Superblock: An effective technique for VLIW and superscalar compilation," *Journal of Supercomputing*, 1993.

[10] S. Mahlke, D. Lin, W. Chen, R. Hank, and R. Bringmann, "Effective compiler support for predicted execution using the hyperblock," in *MICRO*, 1992.

[11] H. Sharangpani and K. Arora, "Itanium processor microarchitecture," *IEEE Micro*, 2000.

[12] T. Austin and G. Sohi, "Zero-cycle loads: Microarchitecture support for reducing load latency," in *MICRO*, 1995.

[13] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *CGO*, 2004.

[14] A. Jaleel, "Memory chracterization of workloads using instrumentation-driven simulation: A pin-based memory characterization of the spec cpu2000 and spec cpu2006 benchmark suites," = http://www.jaleels.org/ajaleel/publications/SPECanalysis.pdf.

[15] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg, "A large, fast instruction window for tolerating cache misses," in *ISCA*, 2002.

[16] A. Cristal, O. J. Santana, M. Valero, and J. F. Martínez, "Toward kilo-instruction processors," *ACM Trans. Archit. Code Optim.*, vol. 1, no. 4, pp. 389–417, Dec. 2004. [Online]. Available: http://doi.acm.org/10.1145/1044823.1044825

[17] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton, "Continual flow pipelines," in *ASPLOS*, 2004.

[18] A. Hilton, S. Nagarakatte, and A. Roth, "icfp: Tolerating all-level cache misses in in-order processors," in *HPCA*, 2009.

[19] A. Hilton and A. Roth, "Bolt: Energy-efficient out-of-order latency-tolerant execution," in *HPCA*, Jan 2010, pp. 1–12.

[20] S. Nekkalapu, H. Akkary, K. Jothi, R. Retnamma, and X. Song, "A simple latency tolerant processor," in *ICCD*, Oct 2008, pp. 384–389.

[21] R. Barnes, S. Ryoo, and W.-M. Hwu, ""flea-flicker" multipass pipelining: an alternative to the high-power out-of-order offense," in *MICRO*, 2005.

[22] U. Ramachandran, G. Shah, A. Sivasubramaniam, A. Singla, and I. Yanasak, "Architectural mechanisms for explicit communication in shared memory multiprocessors," in *Supercomputing, 1995. Proceedings of the IEEE/ACM SC95 Conference*, 1995, pp. 62–62.

[23] T. Mowry and A. Gupta, "Tolerating latency through software-controlled prefetching in shared-memory multiprocessors," *J. Parallel Distrib. Comput.*, vol. 12, no. 2, pp. 87–106, Jun. 1991.

[24] A. Klaiber and H. Levy, "An architecture for software-controlled data prefetching," in *Computer Architecture, 1991. The 18th Annual International Symposium on*, 1991, pp. 43–53.

[25] T. C. Mowry, "Tolerating latency in multiprocessors through compiler-inserted prefetching," *ACM Trans. Comput. Syst.*, vol. 16, no. 1, Feb. 1998.

[26] M. Karlsson, F. Dahlgren, and P. Stenstrom, "A prefetching technique for irregular accesses to linked data structures," in *High-Performance Computer Architecture, 2000. HPCA-6. Proceedings. Sixth International Symposium on*, 2000, pp. 206–217.

[27] A. Roth, A. Moshovos, and G. S. Sohi, "Dependence based prefetching for linked data structures," *SIGOPS Oper. Syst. Rev.*, vol. 32, no. 5, Oct. 1998.

[28] J. Smith, "Decoupled access/execute architectures," in *ACM Transactions on Computer Systems*, 1984.

[29] K. Ebcioglu and E. R. Altman, "Daisy: Dynamic compilation for 100% architectural compatibility," in *ISCA*, 1997.

[30] M. Merten, A. Trick, C. George, J. Gyllenhaal, and W.-M. Hwu, "A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization," in *ISCA*, 1999, pp. 136–148.

[31] S. Jee and K. Palaniappan, "Dynamically scheduling vliw instructions with dependency information," in *Interaction between Compilers and Computer Architectures, 2002. Proceedings. Sixth Annual Workshop on*, 2002, pp. 15–23.

[32] R. Nair and M. Hopkins, "Exploiting instruction level parallelism in processors by caching scheduled groups," in *ISCA*, June 1997, pp. 13–25.

[33] S. Patel and S. S. Lumetta, "replay: A hardware framework for dynamic optimization," *Computers, IEEE Transactions on*, vol. 50, no. 6, pp. 590–608, Jun 2001.

[34] F. Spadini, B. Fahs, S. Patel, and S. S. Lumetta, "Improving quasi-dynamic schedules through region slip," in *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, March 2003, pp. 149–158.

[35] J. Fisher, "Trace scheduling: A technique for global microcode compaction," *Computers, IEEE Transactions on*, vol. C-30, no. 7, pp. 478–490, July 1981.

[36] S. Mahlke, D. Lin, W. Chen, R. Hank, and R. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *MICRO*, Dec 1992, pp. 45–54.