Implementing Software-Hardware Cooperative Memory Disambiguation
Technical Report

Alok Garg, Ruke Huang, and Michael Huang
Department of Electrical & Computer Engineering
University of Rochester
Dec. 2005
{garg,hrk1,michael.huang}@ece.rochester.edu

Abstract

In high-end processors, increasing the number of in-flight instructions can improve performance by overlapping useful processing with long-latency accesses to the main memory. Buffering these instructions requires a tremendous amount of microarchitectural resources. Unfortunately, large structures negatively impact processor clock speed and energy efficiency. Thus, innovations in effective and efficient utilization of these resources are needed. In this paper, we target the load-store queue, a dynamic memory disambiguation logic that is among the least scalable structures in a modern microprocessor. We propose to use software assistance to identify load instructions that are guaranteed not to overlap with earlier pending stores and prevent them from competing for the resources in the load-store queue. We show that the design is practical, requiring off-line analyses and minimum architectural support. It is also very effective, allowing more than 40% of loads to bypass the load-store queue for floating-point applications. This reduces resource pressure and can lead to significant performance improvements.

1 Introduction

To continue exploiting device speed improvement to provide ever higher performance is challenging but imperative. Simply translating device speed improvement to higher clock speed does not guarantee better performance. We need to effectively bridge the speed gap between the processor core and the main memory. For an important type of applications that have never-ending demand for higher performance (mostly numerical codes), an effective and straightforward approach is to increase the number of in-flight instructions to overlap with long latencies. This requires a commensurate increase in the effective capacity of many microarchitectural resources. Naive implementation of larger physical structures is not a viable solution as it not only incurs high energy consumption but also increases access latency which can negate improvement in clock rate. Thus, we need to consider innovative approaches that manages these resources in an efficient and effective manner.

We argue that a software-hardware cooperative approach to resource management is becoming an increasingly attractive alternative. A software component can analyze the static code in a more global fashion and obtain information hardware alone can not obtain efficiently. Furthermore, this analysis done in software does not generate recurring energy overhead. With energy consumption being of paramount importance, this advantage alone may justify the effort needed to overcome certain inconvenience to support a cooperative resource management paradigm.

In this paper, we explore a software-hardware cooperative approach to dynamic memory disambiguation. The conventional hardware-only approach employs the load-store queue (LSQ) to keep track of memory instructions to make sure that the out-of-order execution of these instructions do not violate the program semantics. Without the a priori knowledge of which load instructions can execute out of program order and not violate program semantics, conventional implementations simply buffer all in-flight load and store instructions and perform cross-comparisons during their execution to detect all violations. The hardware uses associative arrays with priority encoding. Such a design makes the LSQ probably the least scalable of all microarchitectural structures in modern out-of-order processors. In reality, we observe that in many applications, especially array-based floating-point applications, a significant portion of memory instructions can be statically determined not to cause any possible violations. Based on these observations, we propose to use software analysis to identify certain memory instructions to bypass hardware memory disambiguation. We show a proof-of-concept design where with simple hardware support, the cooperative mechanism can allow an average of 43% and up to 97% of loads in floating-point applications to bypass the LSQ. The reduction in disambiguation demand results in energy savings and reduced resource pressure which can improve performance.

The rest of the paper is organized as follows: Section 2 provides a high-level overview of our cooperative disambiguation model; Sections 3 and 4 describe the software and hardware support respectively; Section 5 discusses our experimental setup; Section 6 shows our quantitative analyses; Section 7 summarizes some related work; and Section 8 concludes.

2 Resource-Effective Memory Disambiguation

2.1 Resource-Effective Computing

Modern high-end out-of-order cores typically use very aggressive speculations to extract instruction-level parallelism. These speculations require predictors, book-keeping structures, and buffers to track dependences, detect violations, and undo any effect of mis-speculation. High-end processors typically spend far more transistors on orchestrating speculations than on the actual execution of individual instructions. Unfortunately, as the number of in-flight instructions increases, the effective size of these structures has to be scaled up accordingly to prevent frequent pipeline stalls. Increasing the actual size of these resources presents many problems. First and foremost, the energy consumption increases. The increase is especially significant if the structure is accessed in an associative manner such as in the case of the issue queue and the LSQ. At a time when energy consumption is perhaps the most important limiting factor for high-end processors, any change in microarchitecture that results in energy increase will need substantial justifications. Second, larger physical structures take longer to access, which may translate into extra cycles in the pipeline and diminish the return of buffering more instructions. Therefore, we need to innovate in the management of these resources and create *resource-effective* designs. Whether the speculative out-of-order execution model can continue to exploit technology improvements to provide higher single-thread performance is to a large extent determined by whether we can effectively utilize these resources.

Much research has been done in microarchitectural resource management such as providing two-level implementations of register files, issue queues, and the LSQ [2–4, 12, 16, 23, 28]. This prior research focuses on hardware-only approach. A primary benefit of hardware-only approaches is that they can be readily deployed into existing architectures and maintain binary compatibility. However, the introduction of software to gather information has many advantages over a hardware-only approach. First, a software component can analyze the static code in a more global fashion and obtain information hardware alone can not (efficiently) obtain. For instance, a compiler can easily determine that a register is dead on all possible subsequent paths, whereas in hardware, the same information would be highly inefficient to obtain. Thus, a hardware-software *cooperative* approach can achieve better optimization with lower overall system complexity. Second, even if certain information is practical to obtain via hardware look-ahead, there is a recurring energy overhead associated with it, possibly for every dynamic instance of some event. With the increasing importance of energy efficiency, we argue that a cooperative approach to resource management (or optimization in general) is a promising area that deserves more attention.

A cooperative approach does raise several new issues. One important issue is the support for a general-purpose interface to communicate information between the software and hardware components *without creating compatibility obligations*. Although this is a different topic altogether and an in-depth study is beyond the scope of this paper, we note that this could be achieved through decoupling the architected ISA (instruction set architecture) and the physical ISA and rely on binary translation between the two. Such virtualization of ISA is feasible, well understood, and tested in real-world products [15]. In Figure 1, we illustrate one example system where the hardware can directly execute un-translated “external” binaries as well as translated internal ones. In such a system, different implementations are compatible at the architected ISA level but *do not* maintain compatibility at the physical ISA level. Thus, necessary physical ISA changes to support certain optimization can be easily removed when the optimization is no longer appropriate such as when superseded by a better approach or when it prevents/complicates a more important new optimization. In our study, we assume such support to extend the physical ISA is available.

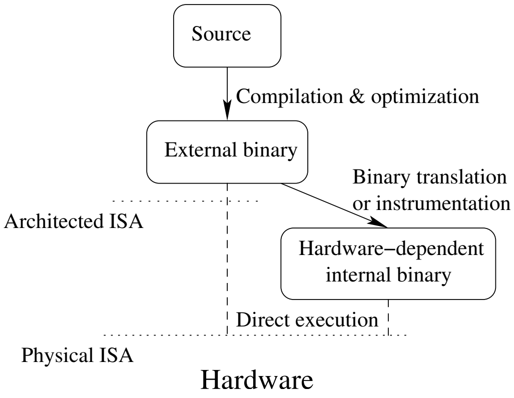

Source
Compilation & optimization
External binary
Binary translation or instrumentation
Architected ISA
Hardware–dependent internal binary
Direct execution
Physical ISA
Hardware

Figure 1. Instruction set architecture support for low-level software-hardware cooperative optimization.

2.2 Cooperative Memory Disambiguation

In this paper, we look at a particular microarchitectural resource, the LSQ used in dynamic memory disambiguation. For space constraint, we do not detail the general operation of the LSQ [10, 27]. Because of the frequent associative searching with wide operands (memory addresses) and the complex priority encoding logic, the LSQ is probably the least scalable structure in an out-of-order core. Yet, *all* resources need to scale up in order to buffer more in-flight instructions. In Figure 2, we show the average performance improvements of increasing the load queue (LQ) size from 48 entries in the otherwise scaled-up baseline configuration (see Section 5). In contrast, we also show the improvement from doubling the number of functional units and issue-width (16-issue) and from doubling the width throughout the pipeline (decode/rename/issue/commit). Predictably, simply increasing issue width or even the entire pipeline's width has a small impact. In contrast, increasing LQ size has a larger impact than doubling the width of the entire processor, which is far more costly. In floating-point applications, this difference is significant. Ironically, these applications tend to have a more regular memory access pattern and in fact do not actually have a high demand for dynamic memory disambiguation.

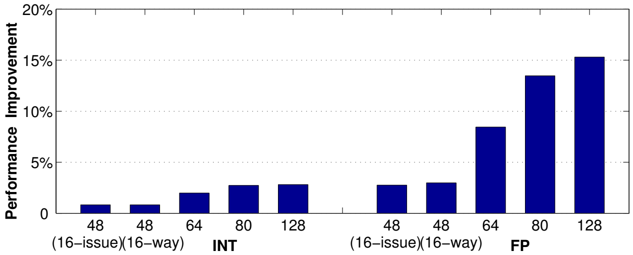


Figure 2. Average performance improvement for SPEC Int and SPEC FP applications as a result of increasing issue width, entire processor pipeline width, or the LQ size.

We envision a *cooperative memory disambiguation* mechanism which uses software to analyze the program binary and, given implementation details, annotate the binary to indicate to hardware what set of memory operations need dynamic memory disambiguation. The hardware can then spend resources only on those operations. In this paper, we focus on load instructions and identify what we call *safe loads*. These instructions are guaranteed (sometimes conditionally) not to overlap with older in-flight stores and hence do not need to check the store queue (SQ) when they execute and do not need an LQ entry. This saves energy needed to search the SQ associatively and reduces the pressure on the LQ.

Using a binary parser, we identify two types of safe loads. First, read-only loads are safe by definition. We use the parser to perform extended constant propagation in order to identify addresses pointing to read-only data segments. Second, in the steady state of loops, any pending stores come from the loop body. In those loops with regular array-based accesses, we can relatively easily determine the relationship between the address of a load and those of all older pending stores. We can thus identify loads that can not possibly overlap with any older pending stores, given architecture details, which determine the number of in-flight instructions. Load identified as safe will be encoded differently by the binary parser and handled accordingly by the hardware.

In addition to identify safe loads statically, we also use software and hardware to cooperate in identifying safe loads dynamically. We use the same binary parser to identify safe stores that are guaranteed not to overlap with future loads (within a certain scope). Safe stores thus identified can indirectly lead to the discovery of more safe loads at runtime: at the dispatch time of a regular (unsafe) load, if all in-flight stores are safe stores, the load can be treated as a safe load. In the following, we will discuss the algorithms we use in the parser and the hardware support needed.

3 Static Analysis with Binary Parsing

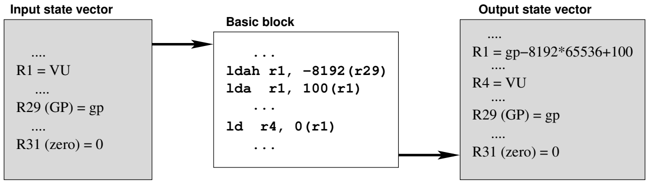
We use a parser based on `alto` [20] and work on the program's binary. If the source code or an information-rich intermediate representation (*e.g.*, [1]) is available, more information can be extracted to identify safe loads more effectively. Without a sophisticated compiler infrastructure, our analysis presented in this work is much less powerful than the state-of-the-art compiler-based dependence analysis or alias analysis. However, this lack of strength does not prevent our proof-of-concept effort to show the benefit of a cooperative approach to memory disambiguation: a more advanced analysis can only improve the effectiveness of this approach.

Our parser targets two types of memory accesses: load from read-only data segments and regular array-based accesses. We emphasize that the goal of using static memory disambiguation is to reduce the unnecessary waste of LSQ resources: to remove those easily analyzable accesses from competing for the resource with those that truly require dynamic disambiguation. Therefore, we do not expect to reduce LSQ pressure for *all* applications. In fact, it is conceivable that for many applications, the parser may not be able to analyze a majority of the read accesses.

3.1 Identifying Read-Only Data Accesses

By definition, read-only data will not be written by stores and therefore, a load of read-only data (referred to as a read-only load hereafter) does not need to be disambiguated from pending stores. To study the potential of identifying read-only loads, we experiment with statically linked Alpha COFF binary format. In this format, there are a few read-only sections storing literals, constants, and other read-only data such as addresses. These sections include `.rconst`, `.rdata`, `.lit4`, `.lit8`, `.lita`, `.pdata`, and `.xdata`. The global pointer (GP), which points to the starting point of the global data section in memory, is a constant in these binaries. The address ranges of read-only sections and the initial value of GP are all encoded in the binary and are thus known to the parser. Since our goal is to explore the potential of cooperative resource management, our effort is not about addressing all possible implementation issues given different binary conventions, or non-conforming binaries. Indeed, when cooperative models are shown to be promising and subsequently adopted in future products, new conventions may be created to maximize their effectiveness.

Knowing the locations of the read-only sections, we can identify static load instructions whose runtime effective address is guaranteed to fall into one of the read-only sections. If a load uses GP as the base address register, it is straightforward to determine whether it is a read-only load. However, to determine if a load using another register as the base is read-only or not, we need to perform data-flow analysis. Our analysis is very similar to constant propagation. The difference is that a register may have different incoming constant values but all point to the read-only sections. In normal constant propagation, the register is usually considered unknown, whereas for our purpose, we know that if a load instruction uses this register as the base *with a zero offset* it is a safe load.



```
    ...
ldah r1, -8192(r29)
lda  r1, 100(r1)
    ...
ld  r4, 0(r1)
    ...
```

Figure 3. An example of register state propagation via symbolic execution. `lda` and `ldah` are address manipulation instructions equivalent to add with a literal.

In our algorithm, a register can be in four different states: no information (NI), value known (VK), value is an address in read-only sections (RO), and value unknown (VU). Except for the GP and ZERO registers, whose value we know at all time, all other registers are initialized to NI. After initialization, we symbolically execute basic blocks on the work list, which is set to contain all the basic blocks at the beginning. During the symbolic execution, only when an instruction is in the form of adding a literal (*i.e.*, $R_i = R_j + literal$) and the source register's state is VK do we set the state of destination register to VK and compute the actual value. In all other cases, the destination register's state is assigned VU (see example in Figure 3).

When joining all predecessors' output vectors to form a basic block's input state vector, a register is VK only if its state in all incoming vectors is VK and the value is the same (normal constant propagation rule). Additionally, a register can be in state RO if in all predecessor blocks it either has a state of RO or has a state of VK and the value points to a read-only section. Otherwise, the register's state is set to VU. Any change in a basic block's input state puts it in the work list for another round of symbolic execution. Essentially, our algorithm is a special constant propagation algorithm with a slightly different lattice as shown in Figure 4. Thus, termination can be similarly proved. Once the data-flow process converges, we perform another pass of symbolic execution for each basic block to determine which load instruction is a read-only load.

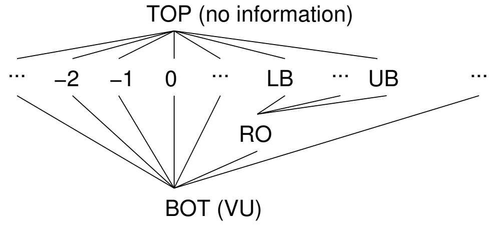


Figure 4. Lattice used in the special constant propagation algorithm. LB and UB indicate the lower and upper address bound of a read-only section. Only one address pair is shown.

In this analysis, we assume the availability of a complete control flow graph with help from the relocation table in the binary [20]. When the table is not embedded in the binary, we can adopt a number of different approaches with different tradeoff between implementation complexity and coverage of read-only loads. On the conservative side, we can do address propagation only within basic blocks or none at all (*i.e.*, identifying only read-only loads with GP as the base register). In a more aggressive implementation, we can profile the application to find out destinations of indirect jumps. We can use the information to augment the control flow. In such a profile-based implementation, as a runtime safety net, a wrapper for all the indirect jumps is employed to detect jumps to destinations not seen before [8]. When such a jump is detected, the runtime system can disable the optimization for the current execution and record the new destination so that the parser can fix the binary for future runs.

3.2 Identifying Other Safe Loads

During an out-of-order program execution, loads are executed eagerly and may access memory before an older store to the same location has been committed, thereby loading stale data from memory. In theory, any load could load stale data and thus the LSQ disambiguates all memory instructions indiscriminately [10, 27]. In practice, however, out-of-order execution is only performed in a limited scope. If the load instruction is sufficiently "far away" from the producer stores, in a normal implementation, we can *guarantee* the relative order. For example, if there are more dynamic store instructions between the producer store and a consumer load than the size of the SQ, then by the time the load is executed, we can guarantee that the producer store has been committed. Notice that the software component in the cooperative optimization model is part of the implementation and therefore can use implementation-specific parameters such as the size of the re-order buffer (ROB) and the SQ. With this knowledge of the processor, we can deduct which stores can still be in-flight when a load executes. We can then analyze the relationship between a load and only those stores. When a load does not overlap with these stores, it is a safe load. To make the job of analyzing all possible prior pending stores tractable, we target loops.

Scope of analysis We only consider loops that do not have other loops nested inside or any function calls/indirect jumps. Additionally if a loop overlaps with a previously analyzed loop, we also ignore it. When a loop has internal control flows, the number of possible execution paths grows exponentially and the analysis becomes intractable. To avoid this problem, we can form traces [14] within the loop body and treat any diversion from the trace as side exits of the loop (which we did in an earlier implementation). This, however, does not significantly increase the coverage of loads in the applications we studied. For simplicity of discussion, we stick with the more limited scope: inner loops without any internal control flows. Note that the loop can still have branches inside, only that these branches have to be side exits. In our study, this scope still covers a significant fraction of dynamic loads (63% for floating-point applications).

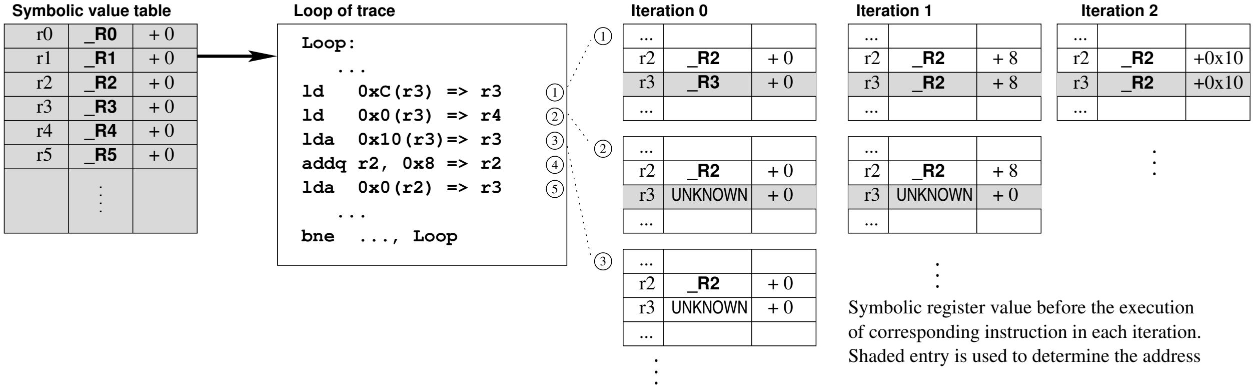
In the *steady state* of these loops, only different iterations of the loop will be in-flight. For every load, the maximum number of older in-flight instructions is finite due to various resource constraints and can be determined as $min(C(S_{ROB}), C(S_{SQ}), ..)$, where $C(S_r)$ is the maximum capacity of in-flight instructions when resource r's size is S_r. The set of store instances a load needs to disambiguate against can be precisely determined given the loop body. For convenience, we refer to this set as the disambiguation store set (DSS) hereafter. For example, if the ROB has n entries, the DSS of a load is at most all the stores in the preceding n instructions from the load. If the parser can statically determine that the load does not conflict with any store instance in the DSS then the load is safe in the steady state. Before reaching this steady state, however, a load can be in-flight together with stores from code sections prior to the loop, outside the scope of the analysis. For this initial transient state, we revert to hardware disambiguation to guarantee memory-based dependences. We place a marker instruction (`mark_sq` in the example shown later in Figure 7) before the loop and any identified safe load will be treated by the hardware as a normal load until all stores prior to the marker drain out of the processor. The design of the hardware support is discussed in Section 4.

Symbolic execution Intuitively, strided array access is a frequent pattern in many loops. With strided accesses, the address at any particular iteration i can be calculated before entering the loop and therefore whether a load overlaps with the stores from the DSS can also be known before entering the loop. Thus, we can generate condition testing code to put in the prologue of the loop. This prologue computes conditions under which a load does not overlap with any stores in its DSS for *any* iteration i. We can then allow the load to become a conditional safe load based on the generated condition. Conditional safe load can be implemented via condition registers reminiscent of predicate registers (Section 4).

To identify these strided accesses and derive the expressions of the address, we use an ad hoc analysis that symbolically executes the loop and tracks the register content. When an address register's state converges to a strided pattern, we can derive its value expression, and hence the steady-state address expression.

Each entry of the symbol table contains a $Base$ and an $Offset$ component ($r_i = Base + Offset$). We use symbols $\_R0$, $\_R1$, ..., and $\_R30$ to represent the loop inputs: the initial values of registers r0 through r30 upon entering the loop (r31 is the hard-wired zero register in our environment). Thus the table starts with ($r_i = \_R_i + 0$) as shown in Figure 5. The symbolic execution then propagates these values through address manipulation instructions. To keep the analysis simple and because we are interested in strided access only, we only support one form of address manipulation instructions: add-constant instructions (or ACI for short). This type includes instructions that perform addition/subtraction of a register and a literal (*e.g.*, in Alpha instruction set: `lda`, `ldah`, some variations of `add`/`sub` with a literal operand, etc.) and addition/subtraction of two registers but one is loop-invariant. When such an instruction is encountered, the source register's $Base$ and $Offset$ component is propagated to the destination register with the adjustment of the constant (literal or the content of a loop-invariant register) to $Offset$. Any other instructions (*e.g.*, load) would cause the $Base$ of destination register to be set to UNKNOWN. Therefore, at any moment, a register can be either UNKNOWN or of the form ($\_R_i + const$).

To further clarify the operations, we walk through an example shown in Figure 5. The figure shows some snapshots of the register symbolic value table *before* executing instructions ①, ②, and ③. In iteration 0, $r3$'s value is initial value $\_R3 + 0$. After instruction ①, which loads into $r3$, its symbolic value becomes UNKNOWN. However, after instruction ⑤, the value becomes known again, in the form of $\_R2 + 8$. To detect strided accesses and compute stride, the symbolic value of the address register (shaded entries in Figure 5) in one iteration is recorded to compare to that of the next iteration. In iteration 0 and 1, $r3$'s values at instruction ① do not "converge" because of the two different reaching definitions. However, in iteration 1 and 2, the values converge to $\_R2 + const$ (with different constants). Since every register used in the loop can have up to two reaching definitions (one from within the loop which is essentially straight-line code and another from before the loop), it may take several iterations for a register to converge. In certain cases, where there is a chain of cyclic assignments, there may not be a convergence. Therefore, our algorithm iterates until the $Base$ component of all registers converge or until we reach a certain


Symbolic value table
Loop of trace
Iteration 0
Iteration 1
Iteration 2
①
②
③
Symbolic register value before the execution
of corresponding instruction in each iteration.
Shaded entry is used to determine the address

Symbolic value table

r0	_R0	+ 0
r1	_R1	+ 0
r2	_R2	+ 0
r3	_R3	+ 0
r4	_R4	+ 0
r5	_R5	+ 0
	⋮	

Loop of trace

```
Loop:
    ...
ld   0xC(r3) => r3     ①
ld   0x0(r3) => r4     ②
lda  0x10(r3)=> r3     ③
addq r2, 0x8 => r2     ④
lda  0x0(r2) => r3     ⑤
    ...
bne  ..., Loop
```

Iteration 0 ①

...		
r2	_R2	+ 0
r3	_R3	+ 0
...		

Iteration 0 ②

...		
r2	_R2	+ 0
r3	UNKNOWN	+ 0
...		

Iteration 0 ③

...		
r2	_R2	+ 0
r3	UNKNOWN	+ 0
...		

Iteration 1 ①

...		
r2	_R2	+ 8
r3	_R2	+ 8
...		

Iteration 1 ②

...		
r2	_R2	+ 8
r3	UNKNOWN	+ 0
...		

Iteration 2 ①

...		
r2	_R2	+0x10
r3	_R2	+0x10
...		

Figure 5. Example of symbolic execution. The register symbolic value is expressed as the sum of an initial register value (*e.g.*, $\_R1$) and a constant (*e.g.*, +0x10). In this example, we show that the first load renders register $r3$ to become UNKNOWN. This makes the second load un-analyzable. However, register $r3$ is always known before the execution of the first load, which makes it analyzable. `ld` is a load, `addq` is a 64-bit add, `lda` is an address calculation equivalent to adding constant, and `bne` is a conditional branch.

limit of iterations (100 in this paper).

Once the $Base$ component converges at *each point* of the loop (*i.e.*, after symbolic execution of every instruction, the destination register's $Base$ is the same as in the prior iteration at the same program point), no "new" propagation of $Base$ is done and therefore the $Base$ component of all registers stays the same in subsequent iterations. After convergence, the only change to the symbolic table is that of the $Offset$, and only an ACI (whose source register's $Base$ is not UNKNOWN) changes that. The set of ACIs in the entire loop are fixed and always add the same constants, and therefore the change to the $Offset$ in the symbol table will be constant for each entry.

Before the base address register of a load converges, the address can have transient-state expressions. In the example shown in Figure 5, the first load's effective address ($r3$+0xC) can be $\_R3$+0xC or $\_R2$+0xC+$8 * i$ ($i = 1, 2, ...$). When generating conditions, we make sure all possibilities are considered. We also note that for *any* load to be safe, *all* stores in the loop have to be analyzable.

Condition generation After the address expressions are computed, we analyze those of the loads against those of the stores and determine under what conditions a load never accesses the same location as any store in the DSS. Since each static store may have multiple instances in a load's DSS, we summarize all locations accessed by these store instances as an address range. Given a strided load, we find out the condition that the load's address falls outside all address ranges for all static stores. Such a range test is a sufficient but not necessary condition to guarantee the safety of the load. The pseudo code of this algorithm is shown in Figure 6. We use i to indicate any iteration. The conditions generated have to be loop-invariant (*i.e.*, independent of i) since they will be tested in the prologue of the loop once for the entire loop. Therefore, when the loads and stores have different strides, our algorithm would not compare them. To remove this limitation, one could apply other tests such as the GCD test or the Omega test [22].

```
foreach $l$ in {all static loads with stride}
    $cs[l] = \{\ \}$          // initial condition set empty
    foreach $s$ = {all static stores}
        // find out range of static instances of $s$ in $l[i]$'s DSS
        $j$ = min $n$, $s[n] \in$ DSS($l[i]$)
        $k$ = max $n$, $s[n] \in$ DSS($l[i]$)
        $[r_{lb}, r_{ub}]$ = Address range of $\{s[j] \ldots s[k]\}$

        // Load $l$'s current iteration address or its transient-
        // state addresses can not overlap with the address
        // range of outstanding instances of store $s$ or its
        // transient-state addresses
        $cs[l] = cs[l] \cup (Addr(l[i]) > r_{ub} || Addr(l[i]) < r_{lb})$
        $cs[l] = cs[l] \cup (TrAddr(l) > r_{ub} || TrAddr(l) < r_{lb})$
        $cs[l] = cs[l] \cup (Addr(l[i]) \neq TrAddr(s))$
        $cs[l] = cs[l] \cup (TrAddr(l) \neq TrAddr(s))$
    end
    Simplify conditions in $cs[l]$
end
```

Figure 6. Pseudo code of the algorithm that determines the condition for a strided load to be safe. $l[i]$ ($s[j]$) indicates the dynamic instance of l (s) in iteration i (j). DSS($l[i]$) is $l[i]$'s disambiguation store set.

We now show a typical code example based on a real application in Figure 7-(a). In this loop, there are 17 instructions, two of them stores. In our baseline configuration, DSS membership is limited by the 32-entry SQ. Then, in the steady state, there can be at most 16 outstanding iterations. In this particular example, every load has the same set of 32 dynamic store instances in its DSS. Also, none of the loads or stores has transient-state address. In iteration i, the (quad-word aligned) address range of these store instances is $[\_R11 + (i - 16) * 16, \_R11 + (i - 1) * 16 + 8]$ and the address of $Ld1$ is $\_R3 + 16 * i$. ($\_R11$ and $\_R3$ are the initial values at loop entrance of register $r11$ and $r3$ respectively.) If the address of $Ld1$ falls outside the range, $Ld1$ becomes a safe load. The condition for that is $(\_R3 + 16 * i < \_R11 + (i - 16) * 16)$ **OR** $(\_R3 + 16 * i > \_R11 + (i - 1) * 16 + 8)$. After solving the inequalities, we get $(\_R3 - \_R11 + 8 > 0)$ **OR** $(\_R3 - \_R11 + 256 < 0)$. Likewise, we can compute the condition for $Ld2$ to be safe: $(\_R3 - \_R11 + 16 > 0)$ **OR** $(\_R3 - \_R11 + 264 < 0)$. The two conditions can be combined into one: $(\_R3 - \_R11 + 8 > 0)$ **OR** $(\_R3 - \_R11 + 264 < 0)$. The addresses of $Ld3$ and $Ld4$ are $\_R11 + 16 * i$ and $\_R11 + 8 + 16 * i$. They can be statically determined to be safe, without the need for runtime condition testing. So they are assigned a special condition register CR_TRUE (Section 4). Figure 7-(b) shows the resulted code after binary parser's analysis and transformation. To be concise, we only show pseudo code of condition evaluation.

```
0x120033140: ldl r31, 256(r3)     ;   prefetch
0x120033144: ldt f21, 0(r3)       ; Ld1
0x120033148: lda r27, -2(r27)     ;   r27 <- r27-2
0x12003314c: lda r3, 16(r3)       ;   r3 <- r3+16
0x120033150: ldt f22, -8(r3)      ; Ld2
0x120033154: ldt f23, 0(r11)      ; Ld3
0x120033158: cmple r27, 0x1, r1   ;   compare
0x12003315c: lda r11, 16(r11)     ;   r11 <- r11+16
0x120033160: ldt f24, -8(r11)     ; Ld4
0x120033164: lds f31, 240(r11)    ;   prefetch
0x120033168: mult f20, f21, f21   ;
0x12003316c: mult f20, f22, f22   ;
0x120033170: addt f23, f21, f21   ;
0x120033174: addt f24, f22, f22   ;
0x120033178: stt f21, -16(r11)    ; St1
0x12003317c: stt f22, -8(r11)     ; St2
0x120033180: beq r1, 0x120033140  ;
```

(a) Original code

```
New_loop_entry: mark_sq
                if(r3-r11+8>0) or (r3-r11+264<0) then
                    cset CR0, 1

   0x120033140: ldl r31, 256(r3)
   0x120033144: sldt f21, 0(r3), [CR0]; safe load with
                 :                    ; cond. reg. CR0
   0x120033150: sldt f22,-8(r3), [CR0]
   0x120033154: sldt f23, 0(r11), [CR_TRUE]
                 :
   0x120033160: sldt f24, -8(r11), [CR_TRUE]
                 :
                 :
   0x120033178: stt f21, -16(r11)
   0x12003317c: stt f22, -8(r11)
   0x120033180: beq r1, 0x120033140
```

(b) Transformed code

Figure 7. Code example from application *galgel*.

Pruning and condition consolidation In the most straightforward implementation, every analyzable load has its own set of conditions and allocates a condition register. Optionally, we can perform profile-driven pruning. Using a training input, we can identify conditions that are likely to be true and those that are not. This allows us to transform unlikely safe loads back to normal loads and thus eliminate unnecessary condition calculation. Perhaps a more important implication of profiling is condition consolidation. Since the remaining safe loads' conditions tend to be true, we can "AND" them together to use fewer condition registers. In the extreme, we can use only one condition register and thus make it the implied condition (even for the unconditional safe loads). Furthermore, we can limit the types of safe load to a few common cases. These measures together will reduce the (physical) instruction code space needed to support our cooperative memory disambiguation model. The tradeoff is that fewer loads will be treated as safe at runtime. We study this tradeoff in Section 6.

Finally, we note that the address used in the parser is virtual address and if a program deliberately maps different virtual pages to the same physical page, the parser can inaccurately identify loads as safe. In general, such address "pinning" is very uncommon: none of the applications we studied does this. In practice, the parser can search in the binary for the related system calls to pin virtual pages and insert code to disable the entire mechanism should those calls be invoked at runtime.

Bypassing load through identifying safe stores Like loads, stores can also be "safe" if it is guaranteed not to overlap with any *future* in-flight loads. In this paper, we identify safe stores in order to indirectly discovery more safe loads. If there is an unanalyzable store in a loop, usually none of the loads may be safe because the DSS of any load is very likely to contain at least one instance of the unanalyzable store. However, the DSS is defined very conservatively and in practice, when a load is brought into the pipeline, usually only a subset of these store instances in the DSS are still in-flight. If this subset does not contain any instance of unanalyzable stores, then the load may still be safe. If we can identify and mark safe stores that do not overlap with future in-flight loads, then at runtime when a normal load is dispatched while there are only safe stores in-flight, we can guarantee that the load will not overlap with any single store. Consequently, we do not need any further dynamic disambiguation and therefore can re-encode the load as a safe load.

The algorithm to identify safe stores mirrors the abovementioned algorithm to identify safe loads: (1) Instead of finding a load's DSS, we find a store's DLS (disambiguation load set), which contains loads *later* than the store; (2) For a store to be safe, all *loads* in the loop have to be analyzable; (3) Since a safe store is only "safe" with respect to loads within the loop, we place a marker (`mark_sq`) upon the *exit* of the loop. As before, an in-flight marker indicates transient state, during which period all loads are handled as normal loads.

4 Architectural Support

Encoding safe loads For those safe loads identified by software, we need a mechanism to encode the information and communicate it to the hardware. There are a number of options. One possibility is to generate a mask for the text section. One or more bits are associated with each instruction differentiating safe loads from other loads. The mask can be stored in the program binary separate from the text. During an instruction cache fill, special predecoding logic can fetch the instructions and the corresponding masks and store the internal, predecoded instruction format in the I-cache. A more straightforward approach is to extend the *physical* ISA to represent safe loads and modify load instructions *in situ*, in the text section. Since we use a binary parser, this extension of the physical ISA does not affect the architected ISA (Section 2). Our study assumes this latter approach.

Conditional safe loads When the parser transforms a normal load into a safe load, there is a condition register associated with it. Only when the condition register is true will the safe load instruction be treated as safe. The architectural support needed includes (a) a few single-bit condition registers, similar to predicate registers, (b) a special instruction (`cset`) that sets a condition register, and (c) a safe load instruction (`sld`) that encodes the condition register used. At the dispatch time of an `sld` instruction, if the value of the specified condition register is false, the safe load will be treated just like a normal load and placed into the LQ. Since the `sld` instructions after a `cset` instruction (in program order) can be dispatched before the `cset` has set the condition (at the execution stage), the condition register is conservatively reset (set to false) when the `cset` instruction is dispatched. Alternatively, we can flash-reset all condition registers when dispatching the marker (`mark_sq`) instruction. A special condition register CR_TRUE is dedicated for unconditional safe loads. It can be set to true either explicitly by a `cset` or implicitly when a `mark_sq` is dispatched.

SQ marker The analyzer places a `mark_sq` instruction to indicate the scope of the analysis: all the dynamic stores older than the marker are outside the scope of the analysis and can overlap with subsequent loads. Therefore, even though the condition register's value may be true, conditional safe loads still need to be treated as normal loads until the stores older than the marker drain out of the SQ. By that time, future safe loads can be dispatched as safe loads (if the condition is satisfied).

While conceptually a marker can be a special occupant of an SQ entry, in a real implementation, we use an extra (marker) bit associated with each entry to represent a scope marker: when a `mark_sq` instruction is dispatched, the marker bit of the youngest valid entry in the SQ (if any) is set. This bit is cleared when that entry is recycled. This design allows two practical advantages. First, we do not waste an SQ entry just to store a marker. Second, and more importantly, the special processing of multiple markers in the SQ is simpler. It is possible that more than one marker appears in the SQ, and only when all markers drain out of the SQ can we let conditional safe loads to bypass the LQ. With the marker bits, it is easy to detect if all markers are drained: any bit that is set pulls down a global signal line. A high voltage in the line indicates the lack of in-flight marker.

Indirect Jumps Though exceedingly unlikely, it is possible that the control flow transfers into a loop through an indirect jump without going through the prologue where the analyzer places the SQ marker and condition testing instructions. To ensure that we do not incorrectly use an uninitialized condition register, we flash-clear all condition registers (including CR_TRUE) when an indirect jump instruction is dispatched.

Safe stores In terms of instruction encoding and the use of condition registers, safe stores are no different from safe loads. However, the handling of safe stores is quite different: because our purpose of identifying them is to further increase the number of safe loads, we are only interested in when the SQ contains just safe stores. The hardware implementation is simple: any entry with a valid, normal (unsafe) store can pull down a global signal line. When this signal is high, we can dynamically dispatch a regular load as a safe one. Of course, software-identified safe stores are safe only within the scope of the analysis (loop). When a loop terminates, the hardware needs to be notified. This is handled by the same SQ marker mechanism described above: when a marker is in-flight, the hardware treats all loads as normal loads. We note that a degenerate form of this mechanism is to dispatch a load as a safe load when there is no in-flight stores at all. This mechanism can be implemented purely in hardware without any software support.

In contrast to the simple support needed in our design, safe stores could be exploited to reduce the pressure of SQ but would require more extensive hardware support. Very likely, we need to split the functionalities of SQ and implement a FIFO queue for buffering and in-order committing of stores and an associative queue for disambiguation and forwarding. Perhaps the more challenging aspect of the design is that we need to ensure that when the scope of analysis (in our case loops) is exited, the identified safe stores from the loop have to participate in the disambiguation/forwarding process with loads from after the exit.

Support for coherent I/O Moving a load out of the LQ prevents the normal monitoring by the coherence and consistency maintenance mechanism. Therefore, the design requires additional support to function in a multiprocessor environment. We note that in a uni-processor environment, if the system provides coherent I/O, there is also the need to monitor load ordering to enforce write serialization, an implicit requirement of coherence. Maintaining write serialization is often done by monitoring the execution of load instructions to detect violations: two loads to the same location executed out of program order and separated by an invalidation to the same location (caused by a DMA transfer). However, invalidations due to DMA transfers are exceedingly infrequent compared to stores issued by the processor. Consequently, we use a separate, light-weight mechanism such as hash tables to keep track of load ordering involving safe loads, thereby avoiding undue increase of LQ pressure. We discuss one such implementation here.

The key observation is that in uniprocessor environment, a write serialization violation is exceedingly rare, primarily because writes from I/O are far less frequent compared to memory accesses from the processor. Thus, we can afford to simplify the hardware and conservatively handle them. In other word, we simplify the tracking of *every* load instruction, at the expense of having false write serialization detection. Recall that a write-serialization violation happens when two loads to the same location execute out of program order *and* there is an intervening external write in between. Conventional LQ tracks this by doing two associative searches: when an invalidation (indicating an external write) occurs, a search using the address of the invalidation message to mark an invalidation bit for all LQ entries to the same location (cache line). During the execution of a load, a second search on the LQ entries corresponding to the younger loads is conducted. If there is a match and the invalidation bit is set, then a younger load fetched the older value, and this current load will get the updated value, violating write serialization. The younger load (and typically all subsequent instructions) will be replayed [10].

This tracking process can be relaxed in several ways. First, exact addresses can replaced by hashing: two addresses hashing to the same entry can be treated as the same address. Second, we can relax age tracking: if two loads to the same location are separated by an invalidation (to that location), we can conservatively replay. When we replay, we only need to replay from the load younger in program order, but we can conservatively replay from the older load. In order words, we do not need to track age for replay purpose either. When we detect a potential violation, we can simply replay from the instruction following the load that triggered the replay. This will guarantee that if the triggering load is indeed earlier in program order, the younger load will be replayed.

With these relaxations in mind, let us start from a simple hash table and progressively describe the entire mechanism. Each entry has two bits, a load (L) bit and an invalidation bit (Inv). Every load, upon execution, uses the address to index the table and set the L bit. Every external invalidation sets the Inv bit if the L bit of the same entry is set. If a load hashes into an entry that has the Inv bit set, there is potentially a violation, we replay from the next instruction following this load.

Clearly, if we only set bits in this table and do not clear any bits, the table will eventually become "clogged" and will result in continuous replays. To properly clean up the table, we can use a set of tables in rotation. We start from the simple (but impractical) example of assigning one table for every load in program order (T_0 .. T_{n-1} for a maximum of n in-flight loads). For convenience, we name the in-flight loads l_0 to l_{n-1} following program order (from oldest to youngest). In this case, a load l_i sets the L bit in its hash entry of table T_i at execution time and the table can be cleaned and recycled when this load retires.

Setting of the Inv bit is as follows. Suppose the address of invalidation hashes into row r. If no table has an L bit set in the row r, the invalidation is ignored. Otherwise, let j be the largest number such that the L bit of row r of table T_j is set. Then, we set the Inv bit of row r for all tables T_0 to T_j. The idea is that we know the youngest load that has already accessed the memory location being invalidated is l_j, then later, if any load older than l_j accesses the same location, write serialization is (potentially) violated and a replay is needed. This is detected when an older load (say, l_k, $k < j$) executes: the Inv bit of row r of table T_j is set. Such a design is essentially the same as the conventional LQ, only that the load address and the invalidation bit are stored in a decoded format.

In a more practical design, multiple consecutive loads share a single table. The total number of tables therefore is smaller. In the extreme case, only two tables are needed. With only two tables, the logic of searching the youngest entry becomes much simplified. In addition, we propose one optimization: instead of clearing a table when all the loads represented by the table retire, we do so when they are all *issued*. When the group of the oldest loads are issued, the table representing these loads is no longer needed and can be recycled. When the issue queue is a compacting queue [11], it is straightforward to perform this table rotation, especially when rotating between two tables: We augment each issue queue entry (only needed for the issue queue containing load instructions) with 1 bit to indicate which table tracks the load. This bit is assigned by the dispatch logic. When all loads in the issue queue have the same bit, say 0, the dispatch will then start to assign the opposite bit (*i.e.*, 1) to future loads. At this point, the table T_0 will be cleared and become the logically "younger" table. With this design, the two tables rotate efficiently and as a result, the bits set in each table remains sparse. Finally, to reduce hash table conflict, we can either use a large physical table or apply the skew principle [26] and use two smaller tables. In this paper, we choose to use skewed tables with the skew functions proposed in [26].

5 Experimental Setup

To evaluate our proposal, we perform a set of experiments using the SimpleScalar [6] 3.0b tool set with the Wattch extension [5] and simulate 1 billion instructions from each of the 26 SPEC CPU2000 benchmarks. We use Alpha binaries.

We made a few simple but important modifications to the simulator. First, we do not allocate an entry in the LQ for loads to the zero register (R31). These essentially prefetch instructions are safe loads that do not need to participate in the dynamic disambiguation process as they do not change program semantics. We note that in our baseline architecture, the LQ only performs disambiguation functions. Buffering information related to outstanding misses is done by the MSHRs (miss status holding registers). If we allocate LQ entries for prefetches, we would exaggerate the result by increasing the pressure on the LQ unnecessarily and quite significantly, since the heavily optimized binaries (compiled using -O4 or -O5) include many prefetches, around 20% of all loads. Second, to model high-performance processors more closely, we simulate speculative load issue (not blocked by prior unresolved stores) and store-load replay. The simulated baseline configuration is listed in Table 1.

Processor core	
Issue/Decode/Commit width	8 / 8 / 8
Issue queue size	64 INT, 64 FP
Functional units	INT 8+2 mul/div, FP 8+2 mul/div
Branch predictor	Bimodal and Gshare combined
- Gshare	8192 entries, 13 bit history
- Bimodal/Meta table/BTB entries	4096/8192/4096 (4 way)
Branch misprediction latency	10+ cycles
ROB/LSQ(LQ,SQ)/Register(INT,FP)	320/96(48,48)/(256,256)
Memory hierarchy	
L1 instruction cache	32KB, 64B line, 2-way, 2 cycle
L1 data cache	32KB, 64B line, 2-way, 2 cycle 2 (read/write) ports
L2 unified cache	1MB, 64B line, 4-way, 15 cycles
Memory access latency	250 cycles
Hash tables	
Table size	32B ($128entry \times 2bit$)
Number of tables	2×2
Mapping function for table 0	$A(10:4) \oplus (A(11:17)$ & 0x55)
Mapping function for table 1	$A(10:4) \oplus (A(11:17)$ & 0xAA)

Table 1. Baseline system configuration.

6 Evaluation

Percentage of safe loads identified The most important metric measuring the effectiveness of our design is the percentage of instructions that bypass the LQ. In Figure 8, we present a breakdown of these safe loads based on their category: (a) read-only loads (ROL), (b) statically safe loads (SSL): loads (other than read-only load) that are encoded as safe loads by the parser and dispatched as safe loads, (c) dynamically safe loads (DSL): normal loads dispatched as safe because all pending stores in the SQ are safe, and (d) degenerate dynamically safe loads (DDSL): normal loads dispatched as safe because the SQ is empty at that time. In Figure 9 we show the number of safe stores identified.

As we can see from Figure 8, in floating-point applications, a significant portion of the loads are safe, suggesting the effectiveness of the cooperative approach. As can be expected, the parser identifies a larger portion of safe loads in floating-point applications than in integer applications. In three applications, about 80% or more loads are dispatched as safe. Even targeting just read-only loads, we can still mark up to 20% of loads as safe.

We can also see that there is only a small portion of dynamically safe loads although Figure 9 shows an average of 30% and up to 98% of stores in floating-point applications are safe. Apparently, we need a very significant number of safe stores to get a sufficient amount of DSL. In applications *applu* and *mgrid*, we do observe a notable fraction of DSL correlated with the high percentage of safe stores. However, in *galgel* and *swim*, the memory access pattern is very regular. So much so, that more than 90% of loads are statically safe loads, subsuming most would-be dynamically safe loads.

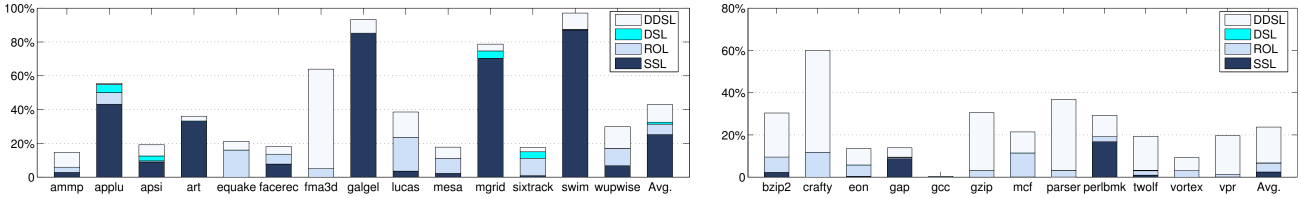
In addition, we see that the percentage of degenerate dynamically safe loads is quite small in floating-point applications, suggesting that only targeting these loads is unlikely to be very effective.

Overall, these results show the effectiveness of cross-layer optimizations, where information useful for optimization in one layer can be hard to obtain in that layer (*e.g.*, hardware), but is easy to obtain in another layer (*e.g.*, compiler, programming language). With simple hardware support, our cooperative disambiguation scheme filters out an average of 43% and up to 97% of loads from doing the unnecessary dynamic disambiguation or competing for related resources.

	Not Safe		Safe		
	A	B	C	D	E
INT	9.2%	10.2%	12.9%	4.0%	40.0%
FP	7.7%	6.6%	13.5%	3.7%	25.6%

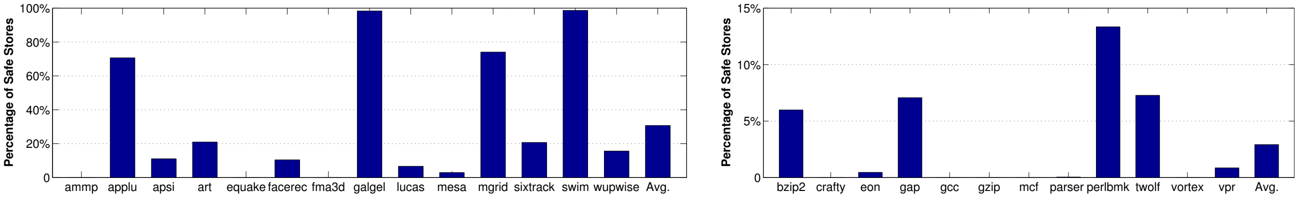
Table 2. Breakdown of loads not dispatched as safe.

Finally, in Table 2, we show the breakdown of the dynamic load instructions not identified as safe, including: (A) those that actually read from an in-flight store; (B) those that read from a committed store that is in the load's disambiguation store set (this category excludes those loads dynamically identified as safe – DSL or DDSL); (C) those that are analyzed by the parser but not marked as a safe load; (D) those that are dispatched in the transient state when a marker is still in-flight; and (E) those that are outside the scope of analysis. Loads in categories C, D, and E do not read from any stores in their DSS. In categories A and B, the parser correctly keeps the load instructions regular, whereas in categories C, D, and E, a more powerful parser may be able to prove some of them safe. We see that to further enhance the effectiveness, we can target category E by broadening the scope of analysis. For example, with the capability to perform inter-procedural analysis, we can handle loops with function calls inside.



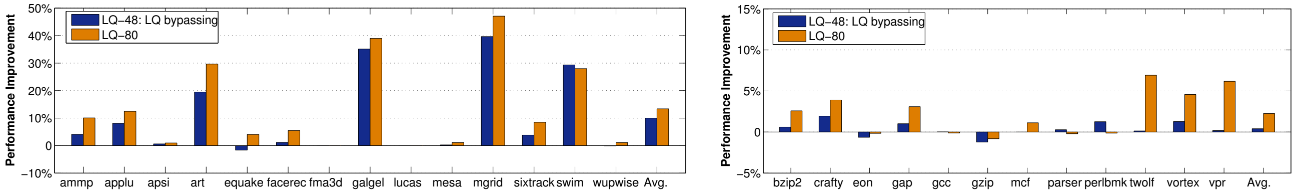
(a) Floating-point applications (b) Integer applications

Figure 8. The breakdown of dynamic load instructions dispatched as safe.



(a) Floating-point applications (b) Integer applications

Figure 9. The percentage of store instructions that are safe.



(a) Floating-point applications (b) Integer applications

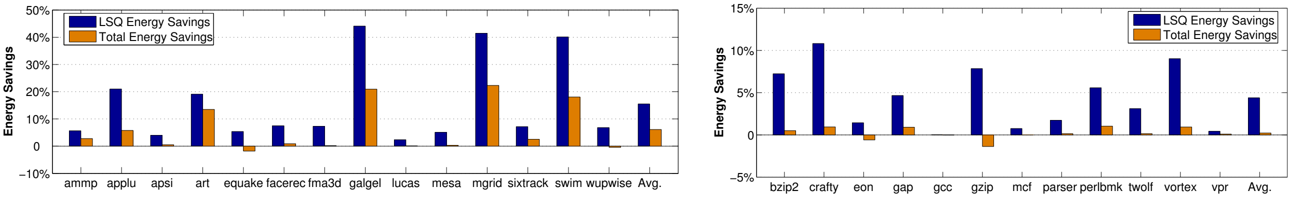
Figure 10. The performance improvement of cooperative memory disambiguation.

Performance impact Reducing resource pressure ameliorates bottleneck and allows a given architecture to exceed its original buffering capability, which in turn increases exploitable ILP. However, quantifying such performance benefit is not entirely straightforward: reducing the pressure on one microarchitectural resource may shift the bottleneck to another, especially if the system is well balanced to start with. Thus, to get an understanding of how effective cooperative disambiguation can be, we experiment with a baseline configuration where other resources are provisioned more generously than the LQ. In Figure 10, we show the performance improvement obtained through LQ bypassing in this baseline configuration. For comparison, we also show the improvement obtained when the LQ size is significantly increased to 80 entries.

For some applications, we can clearly observe the correlation between the percentage of loads bypassing the LQ and the performance improvement. For example, the three floating-point applications that have about 80% or more loads bypassing the LQ (*galgel*, *mgrid*, and *swim*) obtain a significant performance improvement of 29-40%. In general, the effect of identifying safe loads to bypass LQ brings the performance potential of a much larger LQ without the circuit and logic design challenges of building a large LQ.

Clearly, increasing the LQ size only increases the *potential* of performance improvement. Indeed, integer applications in general do not show significant improvement when the LQ size is increased. For a few applications, performance actually degrades. This is possible because, for example, the processor may forge ahead deeper on the wrong path and creates more pollution in the cache. We can also see this degradation in the configuration with an 80-entry LQ. Through instrumentation, however, we can identify loops whose overall performance was negatively affected after transforming regular loads to safe loads. We verified that changing these safe loads back to regular ones eliminates all the performance degradation. Predictably, such a feedback-based pruning has an insignificant impact on other applications.

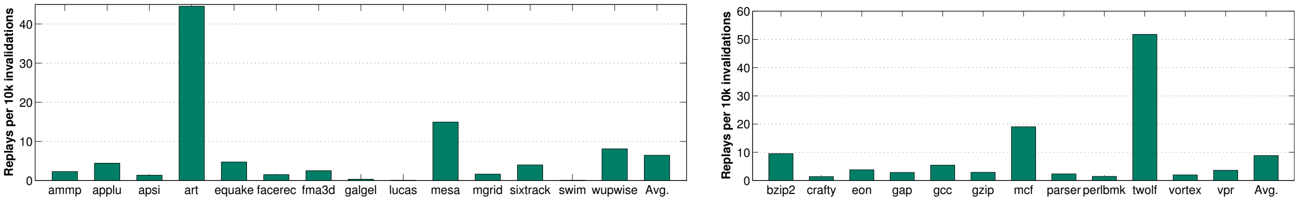
Energy impact In Figure 11, we show the energy impact of our optimization. Specifically, we compute the energy savings in the LSQ and throughout the processor. Energy savings in the LSQ mainly come from the fact that safe loads do not search the SQ. Note that our cooperative memory disambiguation does not reduce energy spent by store instructions accessing the LSQ or the clock power in the LSQ. Thus even with close to 100% loads bypassing the LQ in some applications, the energy savings in the LSQ is less than half. The processor-wide energy savings are mainly the byproduct of expedited execution as according to our Wattch-based simulator, the energy consumption of the LQ and SQ com-



(a) Floating-point applications

(b) Integer applications

Figure 11. The energy savings of cooperative memory disambiguation.



(a) Floating-point applications

(b) Integer applications

Figure 12. Load-load replays triggered per 10k invalidations generated.

bined is only about 3%. This is also reflected in the results of some applications. For example, in *equake*, *eon* and *gzip*, the total energy savings are negated because of the slowdown. Again, after we apply the feedback-guided pruning mentioned above, the slowdown is eliminated, the performance and energy consumption stay almost unchanged as only a small number of loads still bypass the LQ.

Consolidation of condition registers In the above analysis, we assume we have a sufficient number of condition registers, therefore each conditional load instruction uses its own condition register. In our application suite, at most 14 such registers are needed. As explained before, for implementation simplicity, we may choose to use fewer or even just one (implied) condition register. When we limit the number of condition registers to two, we observe no noticeable performance impact for any application we studied. With only one condition register, a naive approach is to set it to the "AND" of all conditions. This creates some "pollution" as one unsatisfied condition prevents all loads in the same loop from becoming safe loads. However, we found that even when we use the naive approach to share the sole condition register, only 3 applications show performance degradation compared to using unlimited number of condition registers: *ammp* (-2.5%), *applu* (-5.9%), and *art* (-15.3%). The rest of the applications show no observable impact. Intuitively, a feedback-based approach can help reduce the impact of condition register deficiency. We found that even simple pruning can be very effective: by filtering out the loads whose condition is never satisfied in a training run, we eliminated the performance degradation of *ammp* and *applu*. However, with such a small set of applications to study, we can not draw many general conclusions.

Overhead of condition testing code We also collect statistics on the actual performance overhead incurred because of executing condition-testing instructions for safe loads. The overhead turns out to be very small. On average, it is about 0.2% of the total dynamic instructions. The maximum overhead is only 1.6%. This overhead can be further reduced by applying profile-based pruning. It is worth mentioning that the offline analysis incurs very little overhead too. On a mid-range PC, our parser takes between 1 and 16 seconds analyzing the suite of applications used. The average run time is 3 seconds.

Impact of alternative support for coherent I/O Finally, we evaluate the support for coherent I/O. Recall that our focus in this paper is still the uni-processor environment. Though the design described in Section 4 handles any coherence activity, and thus would allow correct execution of parallel programs on a shared-memory multiprocessor, though we believe extra optimizations are need to improve the efficiency. At the time of this writing, our simulation infrastructure can not evaluate the design's efficiency in this environment. This is our future work.

We analyzed a uni-processor environment with DMA support and our data suggest invalidations generated by DMA are unlikely to create any noticeable amount of replays. We thus performed a set of experiments that "stress-test" the system by introducing another processor (the "aggressor") to generate memory accesses and hence invalidation messages at higher rates and observe the amount of spurious replays the "victim" processor suffers. These replays will not occur in a conventional LQ-based design as the two processors access disjoint physical memory spaces. However, with the hash table-based design described in Section 4, entry conflict will trigger replays.

In Figure 12, we show the number of spurious replays observed in these experiments. The numbers are reported as per 10,000 *invalidations*, which reflect how well the hash tables are filtering out "environmental noise". We see that on average, 8.9 and 6.5 replays are triggered per 10,000 invalidations for integer and floating-point applications respectively. These frequencies are exceedingly low to cause any noticeable slowdown. The number of invalidations observed ranges from 0.37 to 161.2 per 10,000 instructions with an average of 47. This result suggests that even if the system is used in a dual-core processor, the extra replays caused by the unrelated activities of another processor is quite negligible.

We also did a worst-case scenario experiment where the aggressor runs the same application with the same input in lock-step with the victim processor. The intention is to generate artificially high overlap between the addresses of the invalidation messages and those of the loads in the victim processor. When we use the virtual address in the experiments, we indeed see the replays increase significantly (Table 3. The average number of replays per 10,000 invalidations becomes 247 and 127 for integer and floating-point applications respectively, a 20- to 30-fold increase. Even these many replays are unlikely to cause significant slowdown. We note that this is just an artificial experiment. In reality, even if two processors manage to run two applications in lock step, these two application instances will get different physical addresses for the same virtual address and map to different hash table entries: Our index functions [26] use several bits from the page number portion of the address. In fact, when only a few bits of the addresses are different, they are much more likely to map to different entries than two unrelated addresses. We did an experiment where we mimic the TLB function by changing the few page number bits that are used in the hash table indexing functions. Assuming a 4KB page size, four bits from the page number are used in these functions. We reversed these four bits for the aggressor so sometimes the virtual and the physical versions of these bits are the same but more often they are different. With this change, the average number of replay reduces drastically and becomes negligible. Table 3 shows the detailed statistics of these experiments.

	A	B	C	D	E
bzip2	395691	425	211120	14367	88
crafty	1239434	251	427925	8369	3
eon	625446	314	160556	1534	54
gap	659509	199	306651	3716	84
gcc	672752	367	7017806	11948	0
gzip	715013	328	934156	1198	253
mcf	370780	777	77717	1	1
parser	1328789	508	565677	5696	73
perlbmk	1688068	294	861125	20841	54
twolf	2374032	12920	2374032	12920	2589
vortex	1359904	277	524777	1643	112
vpr	1059054	471	583181	8726	125
ammp	3681944	874	1680886	9	2
applu	3700638	1641	3067759	3726	803
apsi	3280337	450	1991126	102	87
art	8055962	36096	8055962	36096	2802
equake	1100934	576	18529	92	0
facerec	1860321	284	2095629	296	5
fma3d	1340238	351	16726	0	0
galgel	7521284	253	241463	173	0
lucas	2800487	20	1045858	597	0
mesa	1520263	2288	523727	21924	105
mgrid	4380754	726	1142578	12630	12
swim	5000857	37	3104139	11	53
wupwise	2686654	2279	910787	261252	0

Table 3. Total number of invalidations and load-load replays triggered using the hash tables for SPEC applications (each simulated for 0.5 billion instructions). A - Total number of invalidations generated from the aggressor running a different application, B - Total number of replays, C - Total number of invalidations generated from the aggressor running the same application in lock step with the victim, D - Total number of replays if virtual address is used, E - Total number of replays if the 4 bits used in the hash table indexing functions are reversed to mimic virtual to physical address translation.

7 Related Work

To increase the number of in-flight instructions, the effective capacity of various microarchitectural resources need to be scaled accordingly. The challenge is to do so without significantly increasing access latency, energy consumption, and design complexity. There are several techniques that address the issue by reducing the frequency of accessing large structures or the performance impact of doing so. Sethumadhavan et al. propose to use bloom filters to reduce the access frequency of the LSQ [25]. When the address misses in the bloom filter, it is guaranteed that the LQ (SQ) does not contain the address, and therefore the checking can be skipped.

A large body of work adopts a two-level approach to disambiguation and forwarding. The guiding principle is largely the same. That is to make the first-level (L1) structure small (thus fast and energy efficient) and still able to perform a large majority of the work. This L1 structure is backed up by a much larger second-level (L2) structure to correct/complement the work of the L1 structure. The L1 structure can be allocated according to program order or execution order (within a bank, if banked) for every store [2, 12, 28] or only allocated to those stores predicted to be involved in forwarding [4,23]. The L2 structure is also used in varying ways due to different focuses. It can be banked to save energy per access [4, 23]; it can be filtered to reduce access frequency (and thus energy) [2, 25]; or it can be simplified in functionality such as removing the forwarding capability [28].

Most of these approaches are hardware-only techniques and focus on the *provisioning* side of the issue by reducing the negative impact of using a large load queue. Every load still "rightfully" occupies some resource in these designs. Our approach, on the other hand, addresses the *consumption* side of the issue: loads that can be statically disambiguated do not need redundant dynamic disambiguation and therefore are barred from competing for the precious resources. We have shown that in some applications, a significant percentage of loads are positively identified as safe. With increased sophistication in the analysis methods, we expect an even larger portion to be proven safe. When only provisioning-side optimizations are applied, these loads will still consume resources. Additionally, our design is a very cost-effective alternative. It incurs minimal architectural complexity and does not rely on prediction to carry out the optimization, thereby avoids any recurring energy cost for training or table maintenance. Finally, because we are addressing a different part of the problem, our approach can be used in conjunction with some of these hardware-only approaches.

Memory dependence prediction is a well-studied alternative to address-based mechanisms to allow aggressive speculation and yet avoid penalties associated with squashing [9, 17–19]. A key insight of prior studies is that memory-based dependences can be predicted without depending on actual address of each instance of memory instructions and this prediction allows for stream-lined communication between likely dependent pairs. Detailed studies between schemes using dependence speculation and address-based memory schedulers are presented in [19]. A predictor to predict communicating store-load pairs is used by Park et al. to filter out loads that do not belong to any pair so that they do not access the store queue [21]. To ensure correctness, stores check the LQ at commit stage to ensure incorrectly speculated loads are replayed. They also use a smaller buffer to keep out-of-order loads (with respect to other loads) to reduce the impact of LQ checking for load-load order violations.

Value-based re-execution presents a new paradigm for memory disambiguation. In [7], the LQ is eliminated altogether and loads re-execute to validate the prior execution. Notice that the SQ and associated disambiguation/forwarding logic still remain. Filters are developed to reduce the re-execution frequency [7, 24]. Otherwise, the performance impact due to increased memory pressure can be significant [24].

Finally, a software-hardware cooperative strategy has been applied in other optimizations [13, 29]. In [13], a compile-time and run-time cooperative strategy is used for memory disambiguation. If instruction scheduling results in reordering of memory accesses not proven safe by the static disambiguation, it is done speculatively through a form of predicated execution. Code to perform runtime alias check is inserted to generate the predicates. In [29], compiler analysis helps significantly reduce cache tag accesses.

8 Conclusions

In this paper, we have proposed a software-hardware cooperative optimization strategy to reduce resource waste of the LSQ. Specifically, a software-based parser analyzes the program binary to identify loads that can safely bypass the dynamic memory disambiguation process. The hardware, on the other hand, only provides support for the software to specify the necessity of disambiguation. Collectively, the mechanism is inexpensive since the complexity is shifted to software and it is effective: on average, 43% of loads bypass the LQ in floating-point applications, and this translates into a 10% performance gain in our baseline architecture.

Our technique demonstrates the potential of a vertically integrated optimization approach, where different system layers communicate with each other beyond standard functional interfaces, so that the layer most efficient in handling an optimization can be used and pass information on to other layers. We believe such a *cooperative* approach will be increasingly resorted to as a way to manage system complexity while continue to deliver system improvements.

Acknowledgments

This work is supported in part by the National Science Foundation through grant CNS-0509270. We wish to thank the anonymous reviewers for their valuable comments and Jose Renau for his help in cross-validating some statistics.

References

[1] V. Adve, C. Lattner, M. Brukman, A. Shukla, and B. Gaeke. LLVA: A Low-level Virtual Instruction Set Architecture. In *International Symposium on Microarchitecture*, pages 205–216, San Diego, California, December 2003.

[2] H. Akkary, R. Rajwar, and S. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *International Symposium on Microarchitecture*, pages 423–434, San Diego, California, December 2003.

[3] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures. In *International Symposium on Microarchitecture*, pages 245–257, Monterey, California, December 2000.

[4] L. Baugh and C. Zilles. Decomposing the Load-Store Queue by Function for Power Reduction and Scalability. In *Watson Conference on Interaction between Architecture, Circuits, and Compilers*, Yorktown Heights, New York, October 2004.

[5] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *International Symposium on Computer Architecture*, pages 83–94, Vancouver, Canada, June 2000.

[6] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 2.0. Technical report 1342, Computer Sciences Department, University of Wisconsin-Madison, June 1997.

[7] H. Cain and M. Lipasti. Memory Ordering: A Value-based Approach. In *International Symposium on Computer Architecture*, pages 90–101, Munich, Germany, June 2004.

[8] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. Yadavalli, and J. Yates. FX!32: A Profile-Directed Binary Translator. *IEEE Micro*, 18(2):56–64, March/April 1998.

[9] G. Chrysos and J. Emer. Memory Dependence Prediction Using Store Sets. In *International Symposium on Computer Architecture*, pages 142 –153, Barcelona, Spain, June–July 1998.

[10] Compaq Computer Corporation. *Alpha 21264/EV6 Microprocessor Hardware Reference Manual*, September 2000. Order number: DS-0027B-TE.

[11] J. Farrell and T. Fischer. Issue Logic for a 600-Mhz Out-of-Order Execution Microprocessor. *IEEE Journal of Solid-State Circuits*, 33(5):707–712, May 1998.

[12] A. Gandhi, H. Akkary, R. Rajwar, S. Srinivasan, and K. Lai. Scalable Load and Store Processing in Latency Tolerant Processors. In *International Symposium on Computer Architecture*, Madison, Wisconsin, June 2005.

[13] A. Huang, G. Slavenburg, and J. Shen. Speculative Disambiguation: A Compilation Technique for Dynamic Memory Disambiguation. In *International Symposium on Computer Architecture*, pages 200–210, Chicago, Illinois, April 1994.

[14] 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*, pages 229–248, 1993.

[15] A. Klaiber. The Technology Behind Crusoe^{TM}Processors. Technical Report, Transmeta Corporation, January 2000.

[16] A. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A Large, Fast Instruction Window for Tolerating Cache Misses. In *International Symposium on Computer Architecture*, pages 59–70, Anchorage, Alaska, May 2002.

[17] A. Moshovos, S. Breach, T. Vijaykumar, and G. Sohi. Dynamic Speculation and Synchronization of Data Dependences. In *International Symposium on Computer Architecture*, pages 181–193, Denver Colorado, June 1997.

[18] A. Moshovos and G. Sohi. Streamlining Inter-operation Memory Communication via Data Dependence Prediction. In *International Symposium on Microarchitecture*, pages 235–245, Research Triangle Park, North Carolina, December 1997.

[19] A. Moshovos and G. Sohi. Memory Dependence Speculation Tradeoffs in Centralized, Continuous-Window Superscalar Processors. In *International Symposium on High-Performance Computer Architecture*, pages 301–312, Toulouse, France, January 2000.

[20] R. Muth, S. Debray, S. Watterson, and K. De Bosschere. `alto`: A Link-Time Optimizer for the Compaq Alpha. *Software: Practices and Experience*, 31(1):67–101, January 2001.

[21] I. Park, C. Ooi, and T. Vijaykumar. Reducing Design Complexity of the Load/Store Queue. In *International Symposium on Microarchitecture*, pages 411–422, San Diego, California, December 2003.

[22] W. Pugh. The Omega Test: a Fast and Practical Integer Programming Algorithm for Dependence Analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[23] A. Roth. A High-Bandwidth Load-Store Unit for Single- and Multi- Threaded Processors. Technical Report (CIS), Development of Computer and Information Science, University of Pennsylvania, September 2004.

[24] A. Roth. Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization. In *International Symposium on Computer Architecture*, Madison, Wisconsin, June 2005.

[25] S. Sethumadhavan, R. Desikan, D. Burger, C. Moore, and S. Keckler. Scalable Hardware Memory Disambiguation for High ILP Processors. In *International Symposium on Microarchitecture*, pages 399–410, San Diego, California, December 2003.

[26] Andre Seznec. A Case for Two-Way Skewed-Associative Caches. In *International Symposium on Computer Architecture*, pages 169–178, San Diego, California, May 1993.

[27] J. Tendler, J. Dodson, J. Fields, H. Le, and B. Sinharoy. POWER4 System Microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25, January 2002.

[28] E. Torres, P. Ibanez, V. Vinals, and J. Llaberia. Store Buffer Design in First-Level Multibanked Data Caches. In *International Symposium on Computer Architecture*, Madison, Wisconsin, June 2005.

[29] E. Witchel, S. Larsen, C. Ananian, and K. Asanovic. Direct Addressed Caches for Reduced Power Consumption. In *International Symposium on Microarchitecture*, pages 124–133, Austin, Texas, December 2001.