Optimizing Memory Accesses Using Advanced Compile-Time Memory Disambiguation Techniques

Abstract

Load latency observed by dependent instructions can significantly delay the initiation of subsequent computation. Therefore various load optimization techniques that expedite the arrival of loaded data can effectively reduce a program's execution time. Memory to register promotion performed by the compiler is the most effective optimization since it completely bypasses the memory system. An alternative compile-time approach is to schedule a load earlier to partially or completely hide its latency. However, their applicability is usually limited by aliases, function boundaries, and the number of registers.

In this paper, we report the performance improvement enabled by advanced compile-time memory disambiguation. We have designed and implemented a new interprocedural pointer analysis framework in the IMPACT compiler. With the improved quality in compile-time memory disambiguation, register promotion and instruction scheduling can be done in a more aggressive manner. The effectiveness of the new compilation strategy has been comprehensively evaluated using the complete SPECint92 and SPECint95 benchmark suites, which have long been conservatively optimized due to the lack of static memory disambiguation technology able to accommodate these programs' size and intensive usage of pointers. The experimental results show that advanced memory disambiguation translates to peak speedup of 1.58 over commonly practiced intraprocedural memory disambiguation, with an average speedup of 1.19.

1 Introduction

Significant performance improvement of a program's execution can be enabled by reducing the number of load instructions executed and load latency observed by dependent instructions. The earlier the data arrives at the processor, the earlier the dependent computation stream can be started. As the performance gap between the processor and memory widens, the potential benefits derived from memory access optimizations scale accordingly.

In a processor's pipeline, the execution of a load instruction involves the address calculation stage and memory access stage. To completely eliminate a load at compile-time, the compiler needs to determine the equivalence of the load's address with a prior memory instruction whose accessed value is still held by a register and up-to-date with the memory. Such compile-time load-reuse optimizations can reduce the traffic to the memory system and allow data to be retrieved at the processor's speed; but the ability of the compiler to perform these optimizations is limited by architecture factors like the number of registers, and by language factors like the ubiquitous usage of pointers and the unknown side-effects of function calls. Even though many interprocedural pointer analysis algorithms have been proposed to solve the aliasing problem, most of them either lack the scalability to handle large programs or the ability to handle the complete *C* language constructs [1, 2, 3, 4, 5]. As an alternative to load-reuse optimizations, code scheduling can hide the load latency that would have be observed by dependent instructions. Without extra software and hardware supports [6], the compiler needs the definite knowledge of independence between a load and an earlier store. This information is still not available if the underlying memory disambiguation is not comprehensive. Therefore the compiler can only optimize load/store instructions in a conservative manner.

To attack the aliasing and function boundary problems encountered by the compiler, several hardware mechanisms have been designed to speculatively detect the dependence between a load and a prior memory instruction. The general idea is to add extra hardware circuitry whose capacity is larger than the register file to forward recently accessed data faster than the first level data cache in the existing memory hierarchy [7, 8]. Such an approach is speculative since the dependence between two memory instructions is determined by their PCs instead of the actual addresses. Verification is done by comparing the loaded value with the forwarded value, meaning that the amount of memory traffic is not reduced. When mis-speculation is detected, dependent instructions speculatively issued on the wrong data will be re-executed, and the amount of memory traffic may be increased. One advantage of the dynamic load-reuse optimization over the static load-reuse optimization is that the dynamic approach potentially offers larger reuse scope: the sourcing memory instruction may reduce the latency of the dependent load instruction even though they are in different functions. The compiler cannot optimize such redundant loads away in general because the memory is often the only communication channel of interprocedural operations except for a small number of parameter values passed through registers. Address prediction [9, 10, 11, 12] and value prediction [13] are other run-time approaches that have been shown to effectively reduce the latency of load instructions.

In this paper, we present an aggressive yet practical compile-time memory access optimization framework implemented in the IMPACT compiler. It is aggressive since it utilizes the memory disambiguation information generated by accurate interprocedural pointer analysis. It is practical since it has been performed and evaluated on the complete SPECint92 and SPECint95 benchmark suites, which have long been conservatively optimized due to the lack of static memory disambiguation technology able to accommodate these programs' size and intensive usage of pointers. To our best knowledge, no other work has conducted comparable performance and functionality studies on static memory disambiguation using these significant benchmarks. In our experiments, we evaluated the performance improvements enabled by static memory access optimizations: register promotion and load/store scheduling. Both of them can be performed much more aggressively with advanced memory disambiguation. In our experiments, we also evaluated the memory renaming technique proposed by Tyson and Austin [8] to understand the impact of aggressively optimized code to the effectiveness of run-time optimization techniques, and to understand the completeness of our compile-time optimizations. Finally, we measured the cumulative performance improvements enabled by combining static and dynamic memory access optimizations, which include the early load-address generation scheme proposed in [12].

The rest of this paper is organized as follows. Section 2 presents the background information and revisits related work. Section 3 presents the compile-time memory disambiguation framework and memory access optimizations performed in the IMPACT compiler. Section 4 provides a detailed quantitative performance analysis in various aspects. Finally, conclusions are presented in Section 5.

2 Background

2.1 Problem description

Table 1 lists three effective compile-time memory access optimizations: *redundant load/store elimination*, *loop-invariant load/store migration*, and *load/store scheduling*, with their conceptual transformations [14]. All three optimizations can expedite the issue of instructions that use the loaded value, while the first two techniques, collectively called as *register promotion* techniques in this paper, can reduce the amount of memory traffic. To enable these

Optimization	Original stream	Transformed stream
Redundant load/store elimination	LD1-USE1-X-LD2-USE2	LD1-USE1-X-USE2
	DEF-ST1-X-LD2-USE	DEF-ST1-X-USE
	DEF1-ST1-X-DEF2-ST2	DEF1-X-DEF2-ST2
Loop-invariant ld/st migration	LOOP_IN-LDs-USE-X-DEF-STs-LOOP_OUT	LD-LOOP_IN-USE-Y-DEF-LOOP_OUT-ST
Load/store scheduling	DEF-ST1-X-LD2-USE	LD1-DEF-ST2-X-USE

Table 1: Compile-time memory access optimizations.

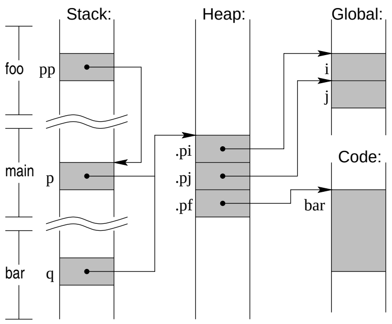
optimizations, the compiler needs to consider the equivalence relation of memory instructions' effective addresses as *dependent*, *independent*, or *ambiguous*, where the ambiguous relation is the most pessimistic and imprecise classification. Assume `X` and `Y` in Table 1 represent arbitrary instructions which can be loads, stores, jsrs, or arithmetic instructions. To enable the compiler to perform redundant load elimination, the prior memory instruction, `LD1` or `ST1`, and the later load instruction, `LD2`, must be dependent. And instructions represented by `X` should not be either dependent or ambiguous with `LD1`, `ST1`, and `LD2`. In addition, `X` should not modify the register that holds the promoted value. To enable the compiler to perform redundant store elimination, besides the dependent relation between `ST1` and `ST2`, `X` should not contain any loads or jsrs that potentially reference the same memory location of the eliminated store.

Loop-invariant load/store migration, which collectively optimizes a group of dependent memory instructions in the loop body, is conceptually a cyclic version of redundant load/store elimination. The conditions enabling this optimization are that all loads/stores represented by `X` are independent with the location of the to-be-promoted group of memory instructions, and all jsrs contained in `X` never modify the promoted location. After eliminating this group of memory instructions, the compiler inserts a promoting load in the loop header and an optional demoting store in each of the loop-exit blocks if the eliminated group contains store instructions. If `X` contains jsrs that may reference the promoted location, demoting stores are inserted before these jsrs to keep the memory up-to-date with the register file, as denoted by `Y`.

To schedule a load earlier to hide its latency, the compiler has to insure that `LD2` and `ST1` are independent, and `X` does not contain either stores or jsrs that potentially modify the location of `LD2`. For all the optimizations discussed above, the overall transformation is beneficial if it does not increase the register pressure.

```
typedef struct s {              bar(s *q)
    int *pi;                    {
    int *pj;                       return *q->pi;
    int (*pf)();                }
} s;
                                main()
int i, j;                       {
                                    s *p;
foo(s **pp)
{                                   foo(&p);
S1: *pp = malloc(sizeof(s));    S5: while (i < 10) {
S2: (*pp)->pi = &i;             S6:     i = (*p->pf)(p) + 1;
S3: (*pp)->pj = &j;             S7:     *p->pj = *p->pj + 2;
S4: (*pp)->pf = bar;                }
}                               }
```

(a)



(b)

```
cb3:
op20: ld_i  r10, SP(-4)    ;p
op21: mov   P0, r10
op22: ld_i  r11, SP(-4)    ;p
op23: ld_i  r12, r11(8)    ;p->pf
op24: jsr   r12
op25: mov   r13, P15
op26: mov   r1, r13
op27: add   r14, r1, 1
op28: st_i  _i(0), r14     ;i
op29: ld_i  r15, SP(-4)    ;p
op30: ld_i  r16, r15(4)    ;p->pj
op31: ld_i  r17, r16(0)    ;*p->pj
op32: add   r18, r17, 2
op33: ld_i  r19, SP(-4)    ;p
op34: ld_i  r20, r19(4)    ;p->pj
op35: st_i  r20(0), r18    *p->pj
op36: ld_i  r21, _i(0)     ;i
op37: blt   r21, 10, cb3
```

SP(-4) is constant
r15(4) = r19(4)
ambiguous anything else

```
cb3:
op20: ld_i  P0, SP(-4)     ;p
op23: ld_i  r12, P0(8)     ;p->pf
op24: jsr   r12
op27: add   r14, P15, 1
op28: st_i  _i(0), r14     ;i
op29: ld_i  r15, SP(-4)    ;p
op30: ld_i  r16, r15(4)    ;p->pj
op31: ld_i  r17, r16(0)    ;*p->pj
op32: add   r18, r17, 2
op35: st_i  r16(0), r18    ;*p->pj
op36: ld_i  r21, _i(0)     ;i
op37: blt   r21, 10, cb3
```

(c)

Figure 1: Code example: (a) source program, (b) points-to relations for pointers contained in the program, (c) conservatively optimized code.

The difficulty behind the above optimizations is not in the plain code transformation, but in the determination of the dependent/independent relations. Languages like *C* provide unrestricted usage of pointers including pointer arithmetics, multi-level pointers, function pointers, dynamically allocated objects, and type casting. Consequently, without complete memory disambiguation, the compiler has to conservatively treat many memory instructions and jsrs as ambiguous, effectively preventing many non-speculative memory optimizations.

We use the example in Figure 1 to illustrate the focus of this paper. In the code segment shown in Figure 1a, there are two global integer variables `i` and `j`, and one dynamically allocated instance of structure s. This instance of s contains three pointers: two integer pointers pi and pj, and one function pointer pf. Statements S1 through S4 in function foo initialize pi to point to i, pj to point to j, and pf to point to function bar. The corresponding points-to relations [3] for pointers contained in this example are shown in Figure 1b. Consider the while loop contained in function main, where the straightforward generated code is shown on the left of Figure 1c. With limited memory disambiguation information acquired intraprocedurally, the compiler can only eliminate some redundant accesses to local variable p. Since the address of p is passed as a parameter to foo, any function call in the loop should be considered as containing potential killing stores to p. Therefore op29 cannot be eliminated even though it is dependent with op20, leaving a lot of redundant memory accesses in the loop as shown on the right of Figure 1c. In Section 3 we will demonstrate how this example can be aggressively optimized in the IMPACT compiler.

2.2 Related work

Here we review related work in the following three domains: *pointer analysis*, *compile-time memory optimizations* and *run-time memory optimizations*, since they are the key components of the study in this paper.

Pointer analysis Pointer analysis plays the key role in enabling aggressive memory access optimizations. Landi and Ryder [1], Choi et al. [2], Emami et al. [3], and Wilson and Lam [15] proposed flow- and context-sensitive interprocedural pointer analysis algorithms to recover pointers' targets at each execution point of the program. Andersen [16], Steengaard [17], and Shapiro and Horwitz [4], took another angle to tackle the intra- and interprocedural pointer analysis problem by using flow-insensitive approaches. Yong et al. [18] studied the situations caused by structures and type casting. The benchmarks studied by these papers are much smaller than SPEC, and how to use the analysis results to guide optimizations was not presented.

The pointer analysis algorithm used in this paper is balanced between all these angles: it employs a flow-insensitive but context-sensitive interprocedural pointer analysis algorithm which can handle all *C* features and SPEC-class benchmarks. Instead of generating dependent relations directly, our analysis greatly improves the classification of independent relations first, then we use copy propagation to derive more dependent relations.

Static optimizations Cooper and Lu [19] examined the register promotion problem in *C* programs. Their technique can handle the promotion of scalar variables, which is less sensitive to the availability of memory disambiguation information. Their results showed the reduction in memory traffic and instruction count, but the performance improvement was not shown. Bodik et al. [20] proposed a comprehensive redundant load elimination algorithm using partial redundancy elimination(PRE). But without memory disambiguation information, pessimistic assumptions about indirect stores and function calls had to be adopted.

We implemented register promotion routines to fully utilize the memory disambiguation information, therefore not only scalar variables but also indirectly accessed values can be promoted to registers. Our optimization is not PRE-complete, since we only remove redundant loads and stores in straight-line blocks and loops. We rely on tail duplication performed by superblock formation to improve the completeness for frequently executed paths.

Dynamic optimizations Golden and Mudge [9], Austin and Sohi [10], Eickemeyer and Vassiliadis [11], and Cheng et al. [12] proposed various techniques to generate the address of a load earlier in the pipeline to reduce the load latency. Lipasti and Shen [13] propose the idea of value prediction to break the limit of data-flow dependence, while Tyson and Austin's technique, *memory renaming* [8], and Moshovos and Sohi's technique, *memory cloaking* [7], extend the source of speculative values by using prior dependent memory instructions.

3 Memory Access Optimizations in IMPACT

3.1 Interprocedural Pointer Analysis

The key component in advanced compile-time memory disambiguation is an efficient and effective pointer analysis algorithm. In the IMPACT compiler, we have designed and implemented a new interprocedural pointer analysis framework. Due to space constraints, the details of the algorithm are omitted in this paper but can be found in [21]. Here we will briefly describe the features that distinguish our analysis from other work:

Generality: Effects caused by features in *C* like type casting, heap-allocated objects, and function pointers are hard to model, therefore except for the work proposed by Wilson

Benchmark	Characteristics		Function-level analysis		Interprocedural analysis		Total time
	Lines	Functions	Time (Sec)	Memory (MB)	Time (Sec)	Memory (MB)	(Sec)
008.espresso	14838	361	9.30	2.12	9.19	9.49	18.49
022.li	7741	357	3.61	1.91	32.81	10.83	36.42
023.eqntott	12053	62	0.87	1.85	0.35	2.23	1.22
026.compress	1503	16	0.24	1.88	0.05	1.74	0.29
072.sc	8639	179	2.91	1.97	1.36	3.57	4.27
085.cc1	90857	1452	43.20	2.79	116.23	33.16	159.43
099.go	29246	372	3.36	2.17	0.37	3.99	3.73
124.m88ksim	19092	252	3.62	2.36	1.19	4.10	4.81
126.gcc	205583	2019	100.90	3.41	241.37	73.86	342.27
129.compress	1934	24	0.13	1.77	0.03	1.65	0.16
130.li	7597	357	3.56	2.49	30.30	10.82	33.86
132.ijpeg	29290	477	11.11	2.37	126.02	22.05	137.13
134.perl	26874	276	23.65	2.31	362.47	42.10	386.12
147.vortex	67205	923	21.20	3.14	30.13	18.02	51.33

Table 2: Benchmark characteristics and resource requirements.

and Lam [15], all pointer analysis algorithms explicitly ignored some of these features by assuming a simplified programming language. For example, type casting enables assembly language-like pointer arithmetics which complicate the alias detection, heap-allocated objects are anonymous and thus are difficult to represent in the analysis, and functions pointers delay the construction of a complete call graph which is vital to any interprocedural analysis. In fact, many SPEC benchmarks contain function pointers that reside as fields in dynamically allocated structures which are manipulated through type casting. As shown in [21], our algorithm is powerful enough to completely accommodate them.

Scalability: Many recently proposed interprocedural pointer analysis algorithms focus on improving the accuracy of analysis results by conducting flow- and context-sensitive analysis [1, 2, 3, 15]. Such algorithms analyze a program statement-by-statement and trace non-recursive function calls along the call graph until leaf functions are reached, causing commonly called functions to be analyzed multiple times. When significant benchmarks like SPEC with dense call graphs are analyzed, the complexity eventually defeats the scalability, therefore none of these algorithms ever demonstrated the success on the complete set of SPEC benchmarks. Our analysis improves the overhead by decomposing the whole analysis into the intraprocedural phase and interprocedural phase. In the intraprocedural phase, statements in each function are analyzed in a flow-insensitive manner to resolve pointers modified by local statements. In the interprocedural phase, only those pointers that can

Benchmark	Targets per pointer					Pointers per target				
	1	2	3	≥ 4	Avg.	1	2	3	≥ 4	Avg.
008.espresso	2378	61	5	10	1.07	1881	173	42	55	1.22
022.li	613	20	8	0	1.06	443	71	20	6	1.25
023.eqntott	229	7	15	0	1.15	145	27	16	7	1.48
026.compress	42	2	0	0	1.05	5	4	2	5	2.88
072.sc	448	6	0	0	1.01	332	40	6	6	1.20
085.cc1	5217	339	92	106	1.26	4292	561	189	219	1.38
099.go	22	0	0	0	1.00	22	0	0	0	1.00
124.m88ksim	470	39	1	3	1.11	471	27	8	4	1.11
126.gcc	9960	946	305	148	1.34	7585	1136	536	599	1.54
129.compress	27	0	0	0	1.00	15	1	2	1	1.42
130.li	613	20	8	0	1.06	443	71	20	6	1.25
132.ijpeg	4317	0	0	0	1.00	3911	47	42	28	1.07
134.perl	2593	62	77	128	1.51	1917	232	70	87	1.87
147.vortex	4545	231	21	3	1.06	4733	93	17	20	1.05

Table 3: Accuracy of analysis results.

be accessed interprocedurally are analyzed in a context-sensitive manner. This analysis model processes less data in the interprocedural phase, and side-effects of each function only need to be collected once in the intraprocedural phase for all calling contexts. Table 2 shows the benchmark characteristics and resource requirements acquired on an HP/9000/780 workstation running at 180 MHz with 256 MB of physical memory. The IMPACT modules were compiled by the HP cc compiler with the -O option. As the table shows, the resource requirements are reasonable.

Accuracy: The accuracy of an interprocedural pointer analysis algorithm can be judged over the temporal and spatial domain. When the temporal domain is concerned, it addresses the ability to determine a pointer's targets at any given point of a program's execution; when the spatial domain is concerned, it addresses whether two isolated pointers in the memory are treated separately in the analysis. Our analysis is temporarily conservative but spatially accurate. Since our analysis is flow insensitive, it provides all possible targets of a pointer over the whole course of a program. When pointers are no longer simple variables, the ability to differentiate every single pointer field in a structure/union and every intermediate pointer of a multi-level pointer, as preserved in our analysis, is more important. However, the current stage of our analysis does not disambiguate each individual element in a linked list or array. Table 3 shows the static breakdown of targets per pointer and pointers per targets obtained by our analysis. If pointers are redefined often or different pointers are collapsed together, the first metric will be much higher than 1; and if the granularity of targets are too coarse in the pointer analysis, the second metrics will be much higher than 1. As the table shows, the averages of both metrics are very close to 1.

3.2 Compilation process

The goal of performing interprocedural pointer analysis is to discover all run-time points-to relations at compile-time. In the IMPACT compiler, interprocedural pointer analysis is performed twice in the front-end analysis phase. The goal of the first invocation is to resolve function pointers. As presented in [22], resolving function pointers in C programs requires the full strength of interprocedural pointer analysis. Completely resolved indirect call sites not only improve the measurement of jsr side-effects, but also reduce the overheads of indirect calls, since some of them can be converted into direct ones thus enabling inlining.

After function inlining, interprocedural pointer analysis is performed for the second time to finalize the disambiguation of memory and jsr instructions [1]. Due to the flow-insensitive nature, the direct results of our analysis can only classify memory and jsr instructions as independent or ambiguous. The notation of *sync arcs* is added between each pair of ambiguous memory and jsr instructions to enforce the semantics order of their execution [23]. For each memory instruction, its potential destination locations can be determined by using its corresponding high-level C expression to trace the points-to relations discovered by the pointer analysis. If two memory instructions' destination sets are not disjoint, they are considered as ambiguous. Each jsr is considered as a macro memory instruction containing some loads and stores. These loads and stores are represented in terms of expressions stemmed from parameters and global variables. Two destination sets, REF and MOD, are constructed for each jsr using these C expressions thereafter. If either set intersects with the destination set of a normal memory instruction, a sync arc is added between the memory and jsr instructions.

When register promotion and load/store scheduling are performed, the absence of sync arcs is strong enough to fulfill the check for independent relations, thus completely enabling

[1] Statistics shown in Table 2 and 3 are gathered by the first invocation of interprocedural pointer analysis, and we see moderate changes in resource requirements and results for the second phase.

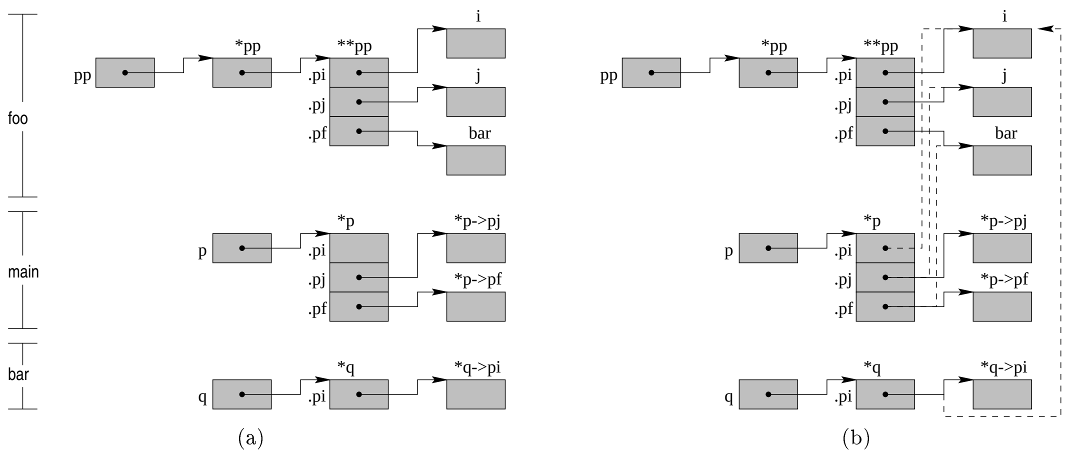


Figure 2: Pointer analysis: (a) intraprocedural phase, (b) interprocedural phase.

scheduling, and partially meeting the requirements for register promotion. To resurrect the dependent relations, we rely on intraprocedural copy propagation: if two memory instructions have the same base registers and offsets, and if the base registers are not redefined in the path, their associated ambiguous relation can be upgraded to dependent relation. This is conceptually similar to the *value name graph* approach proposed by Bodik et al. in [20] except that we use explicit copy propagation and we do not eliminate indirect loads/stores. However, as will be shown by the example below, once redundant indirect memory access are transformed to direct memory accesses, they can be eliminated as well. And an advantage in our approach is that the performed memory disambiguation greatly improves the classification of independent relations, enabling much more optimization opportunities.

3.3 A working example

Here we revisit the example in Figure 1 as a working example to demonstrate how the overall analysis and optimizations proceed. In the intraprocedural pointer analysis phase, the bindings between `pp`, `p`, and `q` are not established yet. So the points-to relations constructed in this phase are based upon the assumption that parameters and global variables are initially undefined. For anonymous objects and indirectly accessed objects through undefined pointers, they are simply named by their corresponding *C* expressions. Figure 2a lists the points-to relations found for `main`, `bar` and `foo`, respectively.

In the interprocedural phase, the bindings between formal and actual parameters are

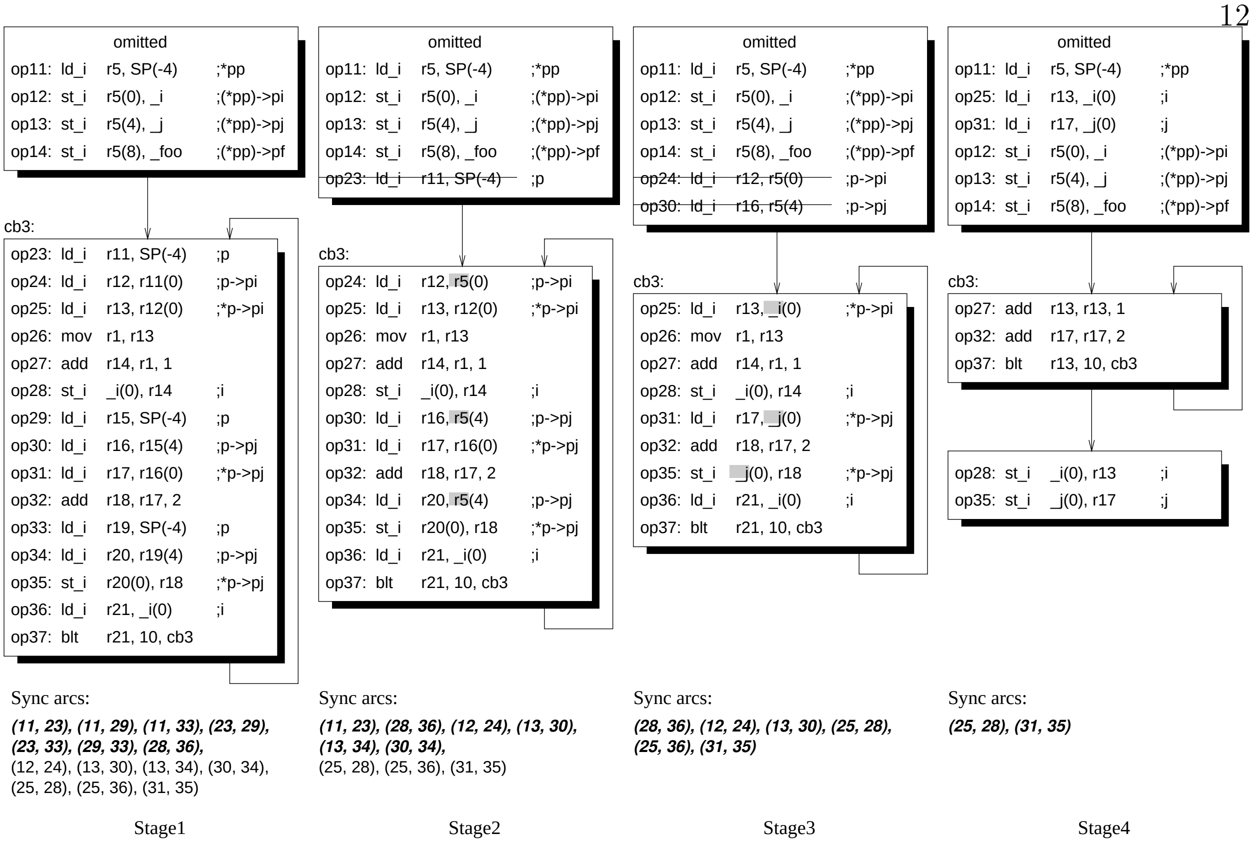


Figure 3: Aggressive code optimization with memory disambiguation information.

taken into consideration: a pointer passed as a formal parameter points to wherever the corresponding actual parameter points to, and the aliases caused by such bindings trigger the propagation of points-to relations. For example, passing the address of `p` to `pp` means that `XpY` and `X*ppY` are aliases, where `X` and `Y` are common sub-expressions. So the points-to relations (`p->pi`, `i`), (`p->pj`, `j`), and (`p->pf`, `bar`) are added as shown in Figure 2b. Similarly, passing `p` to `q` means that `XpY` and `XqY` are aliases as well, so the points-to relation (`p->pi`, `i`) is added, too.

With the completely resolved points-to relations shown in Figure 2b, sync arcs can be added between ambiguous memory instructions and jsrs. Figure 3 shows the step-by-step optimizations on function `main` using memory disambiguation information. Stage 1 shows the non-optimized code after functions `foo` and `bar` have been inlined, where the pairs of operations that have a sync arc between them are listed below the assembly code, and italicized sync arcs in bold mean that they can be convert into dependent ones with copy propagation. Function `bar` can be inlined to `main` since `bar` is the only function name in the destination set of the indirect call-site, `*p->pf`. The points-to relations of the second invocation of interprocedural pointer analysis are not shown since they are very similar to Figure 2b. The sync arc between `op25` and `op28`, for example, is added because their destination sets both have `i`. The sequence of optimizations is explained as follows:

Stage 1 → **Stage 2**: The compiler can perform loop-invariant load/store migration for `op23`, `op29`, and `op33`, since their relations are dependent, and there are no ambiguous instructions conflicting with them. Since all three operations are loads, no stores are inserted in the loop-exit block. The load migrated to the loop header, `op23`, is eliminated since it is dependent with `op11`. So all instructions that access `p` now use `r5` directly, which enables the relations between (`op12`, `op24`), (`op13`, `op30`), (`op13`, `op34`), and (`op30`, `op34`) to be upgraded as dependent.

Stage 2 → **Stage 3**: With the promotion of `p` into `r5`, the memory accesses to `p->pi` and `p->pj` are considered as direct memory accesses in the loop body, since they are not data-dependent on other loads in the loop. Together with the resurrected dependent relations, `op24`, `op30` and `op34` can be migrated out of the loop. `Op24` and `op30` are eliminated in the loop header since they are dependent with `op12` and `op13`, respectively. Copy propagation of the addresses of `i` and `j` enables the relations between (`op25`, `op28`), (`op25`, `op35`), and (`op31`, `op35`) to be upgraded as dependent.

Stage 3 → **Stage 4**: Enabled by similar reasons, the remaining memory instructions in stage 3 can be migrated out of the loop. The difference is that there are two stores, for `i` and `j`, inserted in the loop-exit block. Since there are no sync arcs between `op25`/`op31` and other instructions in the loop header, these migrated loads can be scheduled early.

Significant differences in the final optimized code can be found between Figure 1c and Figure 3. If interprocedural pointer analysis is performed but functions `foo` and `bar` are not inlined to `main`, the optimization strategy discussed above still can greatly improve the quality of optimized code. For example, in the conservatively optimized code shown in Figure 1c, `op24` can be converted into a direct function call to `bar`, and all original loads and stores in `cb3` can be migrated out of the loop since `bar` does not modify the contents of any memory locations. A demoting store for `op35` is inserted before `op27`, since `bar` references the promoted location. The major difference will appear in the ability to eliminate indirect loads in the loop header block. Without inlining `foo`, using copy propagation will not be able to determine the dependent relations between (`*p->pi, i`) and (`*p->pj, j`), causing the need of using three loads to reach the final destination locations instead of one as `op25` and `op31` achieve in stage 4 of Figure 3. Due to space constraints, we will omit the inlining-less optimization example. But as will be shown in Section 4, significant performance improvements can be obtained using the proposed analysis and optimizations.

4 Experimental Results

In order to understand the potential benefits of memory access optimizations for future processors, we have conducted some preliminary experiments. The base architecture is set up to be aggressive in memory access capabilities to provide a conservative estimate of the benefits. The simulated processor can fetch, decode, and issue up to 8 instructions per cycle in order. The processor has eight integer ALU's, four memory ports, two floating point ALU's, and one branch unit, 64 integer registers and 64 floating point registers. The memory system consists of a 64K direct-mapped instruction cache and a 64K direct-mapped, non-blocking data cache, both with 64 byte block size. The data cache is write-through with no write allocate and has a hit latency of 2 cycles and a miss penalty of 4 cycles if hit in the second-level cache. The second-level cache is a unified 1024K, 4-way associative cache with a miss penalty of 32 cycles. The branch prediction scheme is a 1K-entry BTB with 2 bit counters. The instruction set architecture and instruction latencies used match those of the HP PA-7100 microprocessor (besides loads, integer operations have 1-cycle latency) [24]. The benefits of proposed memory optimizations should be greater with the coming generation of machines with 2 load ports and 3-4 cycles of load latency for first-level cache hits.

4.1 Static memory access optimizations

We configured three sets of parameters in the IMPACT compiler to generate three versions of programs with different levels of sophistication in memory disambiguation. The *base* version assumes all memory instructions are ambiguous and all function calls have pessimistic side-effects. Such pessimistic assumptions completely prevent register promotion

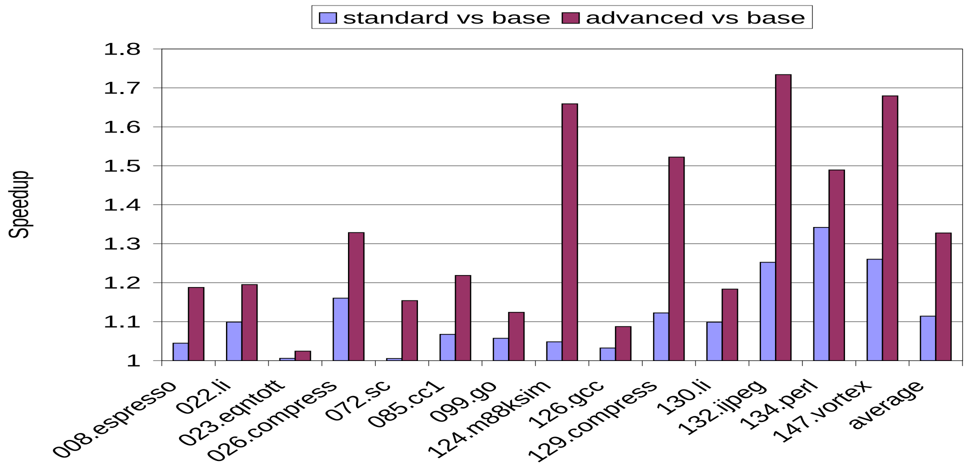
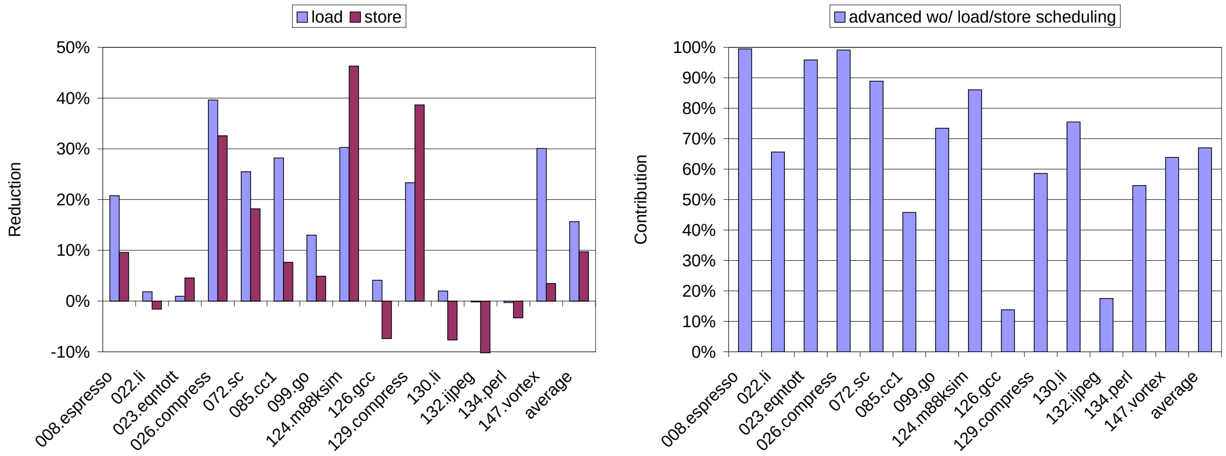


Figure 4: Performance improvements enabled by static analysis and optimizations.

and scheduling [2]. The *standard* version employs memory disambiguation information derived intraprocedurally. Memory disambiguation of this level, which is commonly used by compilers without interprocedural pointer analysis, can effectively disambiguate direct accesses to spill locations/local variables/global variables, and indirect accesses with equivalent base registers and offsets, but revert to ambiguous relations for other combinations. The actual side-effects of library function calls are modeled precisely but pessimistic side-effects are still assumed for user functions. The *advanced* version utilizes the complete memory disambiguation information generated by interprocedural pointer analysis and uses the memory disambiguation information to guide memory access optimizations as described in Section 3. Side-effects of all functions can be accurately modeled in this version. Except for the differences mentioned above, all three versions are compiled through inlining up to 60% of code increase [25], classical optimizations [14], and superblock optimizations [26]. We used the training input sets for all levels of experiments in this paper.

Figure 4 plots the speedups in execution time of the standard and advanced versions over the base version. This figure indicates that memory optimizations with memory disambiguation performed intraprocedurally can provide an average speedup of 1.11. With the best optimizations enabled by advanced memory disambiguation techniques discussed in this paper, it can provide an average speedup of 1.33.

[2]For temporary variables inserted by the compiler and local variables whose addresses are never taken, they are always promoted to registers nevertheless.



(a) (b)

Figure 5: Contribution of register promotion: (a) reduction of dynamic loads and stores, (b) percentage of partial performance improvement enabled without scheduling.

The performance improvement from the standard version to the advanced version reflects the advancement in the compile-time memory disambiguation technology. To better understand its merits, we further studied the run-time behavior of these two versions of code. As discussed in Section 3, promoting values from memory to register is the most effective technique for the compiler to optimize memory accesses, and load/store scheduling can be used to effectively hide the latency of remaining loads. However, it is hard to derive their exact contributions in the final performance improvement since their effects may overlap in some cases. For example, the latency of a redundant but un-eliminated load may be fully hidden by the scheduler. Therefore, we evaluated each optimization's contribution by: (i) showing the amount of reduction of dynamic loads and stores, and (ii) showing the performance improvement achieved without aggressive load/store scheduling. The left bar of each benchmark in Figure 5a shows the reduction of dynamic loads between the standard and advanced versions. As these numbers show, using simple copy propagation to convert ambiguous relations to dependent relations can eliminate as much as 40% of dynamic loads for *026.compress*, with an average reduction rate of 16%. Attributing minor variances ($\pm$ 3%) as the normal behavior of detailed simulation, loads in benchmarks *022.li*, *023.eqntott*, *126.gcc*, *130.li*, *132.ijpeg*, and *134.perl* are not eliminated in a significant manner. Later we will further investigate the possible reasons when we study dynamic optimization techniques.

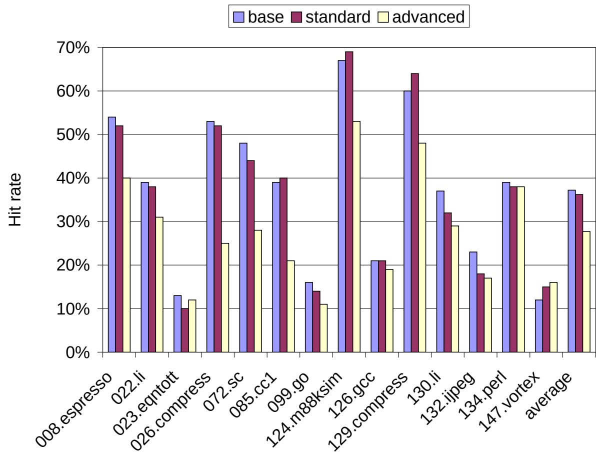
The right bar of each benchmark in Figure 5a stands for the percentage of stores that are eliminated. For some benchmarks, the number of stores is actually increased. The reason is due to the demoting stores inserted by loop-invariant load/store migration before jsrs which use the promoted values [3]. Since stores are usually off the critical path of execution, currently we do not see any noticeable decrease in performance nor increase in resource contentions due to these extra stores, but we are working on better heuristics to prevent such cases.

Figure 5b shows the contribution of register promotion to the overall performance improvement with memory disambiguation. The full-scale speedup is derived from the performance difference between the standard and advanced versions, where the enabled performance improvement shown by each bar is obtained by limiting the use of sync arcs in determining the independence of memory instructions during scheduling. Comparing Figure 5a and 5b, it shows that register promotion and scheduling both can provide significant performance improvements. For benchmarks like *026.compress* and *124.m88ksim* with high load-reduction rates, disabling aggressive load/store scheduling causes little performance degradation. For benchmark *132.ijpeg* with a negligible load-reduction rate, scheduling provides the majority of speedup, which is as significant as 1.38 over the standard version.

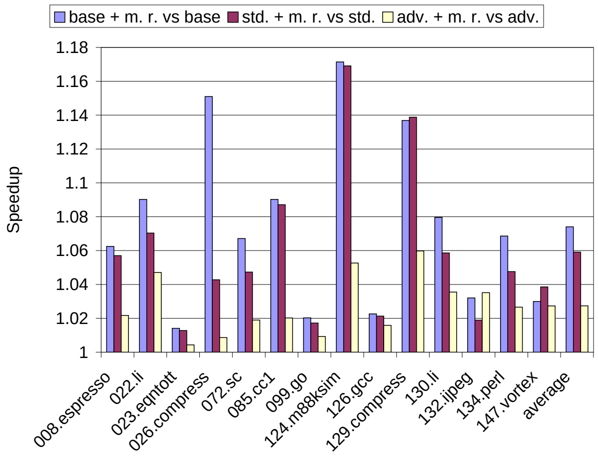
4.2 Dynamic memory access optimizations

We implemented Tyson and Austin's *memory renaming* mechanism in our simulator. The purposes of this experiment are two folds: first, we want to understand the impact of aggressively optimized code to the effectiveness of run-time optimization techniques, and second, we want to understand the completeness of our compile-time optimizations. In our simulation, we made some changes from the original specification to allow a single simulation run to meet both purposes. The first one is that we assumed a perfect dependence predictor which always perfectly knows if a load is redundant at the fetch stage. Although it gives an advantage to the dynamic scheme since there is no penalty for mis-speculation, it captures all redundant loads in the program. For the same purpose, we made the second change

[3]Our heuristics will not promote memory accesses whose contents may be modified by operations in other functions, meaning that no loads are inserted after jsrs.



(a)



(b)

Figure 6: Performance improvements enabled by memory renaming: (a) dependence prediction hit rate, (b) speedup.

to allow the initial binding between sourcing and dependent load instructions propagated through the main memory. The last change we made is that we assumed the dependence is determined by the effective address of the load, not the loaded value as used in the original mechanism. This again truly judges the completeness of static optimizations, and it can offset some advantages given to the dynamic approach since the success rate will be decreased. The PC-indexed load/store cache has 1024 entries with 2-way set associativity, and the value file has 512 entries with LRU replacement, which are the same as the original configuration.

Figure 6a lists the hit rates of dependence predictor when the memory renaming approach is applied to the *base*, *standard*, and *advanced* versions of the programs. The hit rates drop from 0.37, to 0.36, to 0.28, because the compiler removes more redundant loads due to the improvement in memory disambiguation. Figure 6b lists the speedups caused by the memory renaming approach on the three versions of programs. As fewer redundant loads remain in the program and more loads have been scheduled early, the resultant speedups drop from 1.074, to 1.059, to 1.027, respectively. The hit rate of dependence prediction in the memory renaming mechanism also signifies the amount of redundant loads that the compiler fails to eliminate. Therefore we studied the breakdown of remaining loads in the advanced version as shown in Figure 7. We classified the dependences into nine categories as whether a hit is intra- or inter-procedural, and whether the accessed location belongs to the

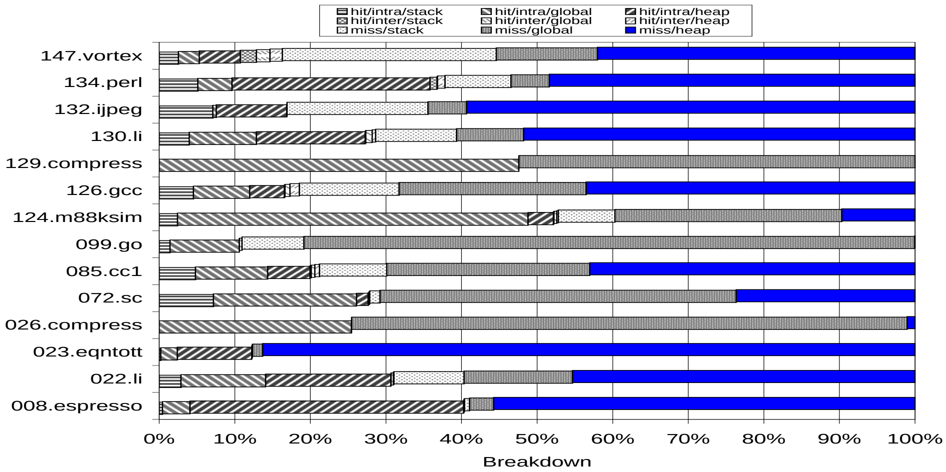


Figure 7: Breakdown of dependence predictions on aggressively optimized code.

stack, global, or heap section. The cumulative portions of the first six categories correspond to the advanced bar reported in Figure 6a. As Figure 7 shows, almost all intraprocedural stack-based redundancy has been eliminated by the compiler, and aggressive inlining of 60% has eliminated much of the interprocedural redundancy. The majority of remaining intraprocedural stack-based redundancy is attributed to spill code generated by the register allocator. However, some benchmarks still contain a significant number of intraprocedural heap- and global-based redundancy. One reason is that the scale of copy propagation is not significant enough to convert ambiguous relations to dependent relations, as the cases for *022.li* and *134.perl*. Although it can be improved by performing flow- and context-sensitive pointer analysis, the exponential complexity may prohibit its practical use. Another reason is that the sourcing memory instruction is not included in the superblock, which can be improved by applying partial redundancy elimination [20].

4.3 Overall performance improvements

Figure 8 shows the continuous performance improvements from code optimized with pessimistic memory disambiguation to accurate memory disambiguation, with both static and dynamic optimization techniques, where all speedup numbers are normalized to the base version. As the trend shows, memory renaming applied to the base version (`base+m.r.`) sometimes outperforms the version with intraprocedural memory disambiguation (`std.`), but memory renaming applied to the standard version (`std.+m.r.`) is consistently outper-

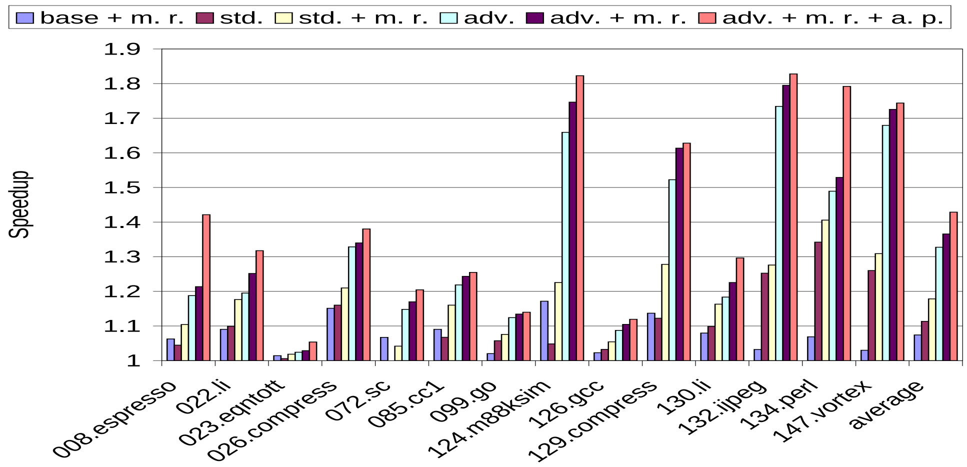


Figure 8: Speedups enabled by various memory access optimization techniques.

formed by advanced memory disambiguation (`adv.`), which enables the sharpest performance improvement. The trend also shows that matured compile-time memory optimization techniques can inject more benefits than run-time load-reuse techniques. Furthermore, run-time optimization techniques can complement compile-time optimization techniques (`adv.+m.r.`).

The right-most bar (`adv.+m.r.+a.p.`) of each benchmark in this figure is generated by further enhancing the pipeline design using the compiler-directed early address generation technique proposed in [12]. In this configuration, we used the compiler to analyze all load instructions in the program. If a load's effective address is found to have non-zero strides carried by loops, the memory renaming mechanism will bypass the load. Instead, its predicted effective address is used to speculatively access the memory. For benchmarks which have loads with loop-carried dependences, the address prediction mechanism can work constructively with memory renaming since memory renaming is not designed to handle loop-carried dependences, which are included in the three miss-segments in Figure 7. As shown in Figure 8, integrating these two dynamic optimizations can further boost the performance.

5 Conclusions

In this paper, we evaluated the effect of aggressive compile-time memory disambiguation on memory access optimizations performed in the IMPACT compiler. This is the first practical study on this topic using significant benchmarks like SPEC. The experimental results demonstrate that advanced compile-time memory disambiguation technology can provide significant performance improvements. Using the memory disambiguation information generated by the proposed interprocedural pointer analysis to guide register promotion and load/store scheduling, the static methods alone can generate peak speedup of 1.58 over intraprocedural memory disambiguation techniques, with an average speedup of 1.19.

From the comparison between static and dynamic memory access optimization techniques, it shows that aggressive static methods have advantages over purely dynamic methods, but dynamic methods still can complement static methods. This initial study also indicates that address prediction can work constructively with speculative load-reuse mechanisms to extend the coverage of load latency reduction techniques.

We would like to continue this study in two ways. From the perspective of the compiler, we want to improve the coverage of memory access optimizations without sacrificing feasibility. From the perspective of the architecture, we want to select the most appropriate speculative mechanism for different types of loads. We believe the integrated approach should provide broader applicability to tackle the memory access optimization problem.

References

[1] W. Landi and B. G. Ryder, "A safe approximate algorithm for interprocedural pointer aliasing," in *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 235–248, June 1992.

[2] J. D. Choi, M. G. Burke, and P. Carini, "Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects," in *Proceedings of the 20th ACM Symposium on Principles of Programming Languages*, pp. 232–245, January 1993.

[3] M. Emami, R. Ghiya, and L. J. Hendren, "Context-sensitive interprocedural points-to analysis in the presence of function pointers," in *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pp. 242–256, June 1994.

[4] M. Shapiro and S. Horwitz, "Fast and accurate flow-insensitive points-to analysis," in *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 1–14, January 1997.

[5] R. Hasti and S. Horwitz, "Using static single assignment form to improve flow-insensitive pointer analysis," in *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pp. 97–105, June 1998.

[6] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu, "Dynamic memory disambiguation using the memory conflict buffer," in *Proceedings of 6th International Conference on Architectual Support for Programming Languages and Operating Systems*, pp. 183–193, October 1994.

[7] A. Moshovos and G. S. Sohi, "Streamlining inter-operation memory communication via data dependence prediction," in *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pp. 235–245, December 1997.

[8] G. S. Tyson and T. M. Austin, “Improving the accuracy and performance of memory communication through renaming,” in *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pp. 218–227, December 1997.

[9] M. Golden and T. N. Mudge, “Hardware support for hiding cache latency,” tech. rep., University of Michigan, February 1993.

[10] T. M. Austin and G. S. Sohi, “Zero-cycle loads: Microarchitecture support for reducing load latency,” in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pp. 82–92, December 1995.

[11] R. J. Eickemeyer and S. Vassiliadis, “A load-instruction unit for pipelined processors,” *IBM Journal of Research and Development*, vol. 27, pp. 547–564, July 1993.

[12] B. Cheng, D. A. Connors, and W. W. Hwu, “Compiler-directed early load-address generation,” in *Proceedings of the 31st Annual International Symposium on Microarchitecture.*

[13] M. H. Lipasti and J. P. Shen, “Exceeding the dataflow limit via value prediction,” in *Proceedings of the 29th International Symposium on Microarchitecture*, pp. 226–237, December 1996.

[14] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools.* Reading, MA: Addison-Wesley, 1986.

[15] R. P. Wilson and M. S. Lam, “Effective context-sensitive pointer analysis for c programs,” in *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pp. 1–12, June 1995.

[16] L. O. Andersen, *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994.

[17] B. Steensgaard, “Points-to analysis in almost linear time,” in *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 32–41, January 1996.

[18] S. Yong, S. Horwitz, and T. Reps, “Pointer analysis for programs with structures and casting,” in *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pp. 91–103, May 1999.

[19] K. D. Cooper and J. Lu, “Register promotion in c programs,” in *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, pp. 308–319, June 1997.

[20] R. Bodik, R. Gupta, and M. L. Soffa, “Load-reuse analysis: Design and evaluation,” in *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pp. 64–76, May 1999.

[21] B. Cheng and W. W. Hwu, “A practical interprocedural pointer analysis framework,” Tech. Rep. CRHC-99-01, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, April 1999.

[22] B. Cheng and W. W. Hwu, “An empirical study of function pointers using spec benchmarks,” Tech. Rep. CRHC-99-02, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, May 1999.

[23] D. M. Gallagher, *Memory Disambiguation to Facilitate Instruction-Level Parallelism Compilation.* PhD thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.

[24] Hewlett-Packard Company, Cupertino, CA, *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, 1990.

[25] B. Cheng, “A profile-driven automatic inliner for the impact compiler,” Master's thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1997.

[26] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, “The Superblock: An effective technique for VLIW and superscalar compilation,” *The Journal of Supercomputing*, vol. 7, pp. 229–248, January 1993.