An Efficient Static Analysis Algorithm to Detect Redundant Memory Operations[*]

Keith D. Cooper and Li Xu
Department of Computer Science
Rice University
Houston, Texas, USA

ABSTRACT

As memory system performance becomes an increasingly dominant factor in overall system performance, it is important to optimize programs for memory related operations. This paper concerns static analysis to detect redundant memory operations and enable other compiler transformations to remove such redundant operations.

We present an extended global value numbering algorithm to detect redundant memory operations. The key of the new algorithm is a novel SSA-based representation for memory state which allows accurate reasoning about memory state. Using this representation, the algorithm can trace values through memory operations to detect equivalence in the same way that it traces them through register-based scalar operations. Thus it discovers both redundancy involving scalar values and redundancy involving memory operations. The redundancy relation detected by the algorithm can then be used by traditional redundancy elimination transformations to remove those redundant operations.

Experiments using a suite of realistic applications demonstrate the algorithm is powerful and fast. In practice, it has essentially linear time complexity.

1. INTRODUCTION

The rate of improvement in microprocessor CPU speed continues to exceed the rate of improvement in DRAM memory speed, producing an increasing gap between processor and memory performance. It is vital to optimize memory usage to achieve better performance on modern processor architectures. An effective technique is to detect redundant memory operations – loads and stores that will have same effect as earlier memory operations in execution and either remove them or replace them with cheaper scalar operations. Consider the sample C code in Figure 1. For the accesses to `p->x` and `p->y` in line 10 and 11, the compiler generates 4 loads. However, as the values of `p->x` and `p->y` are not changed, the loads in line 11 are redundant. The two loads can be removed and their results from line 10 can be reused.

```
1    struct parm {
2      int x;
3      int y;
4    };
5    struct parm pa = {3, 7};
6    struct parm pb = {2001, 2002};
7
8    void Compute(struct parm *p, int *result)
9    {
10     result[0] = p->x + p->y;
11     result[1] = p->x - p->y;
12   }
13   void Client()
14   {
15     int ra[2], rb[2];
16     Compute(&pa, ra);
17     Compute(&pb, rb);
18     ...
19   }
```

Figure 1: An Example Code Fragment

Redundant memory operations can be classified into three categories: run-time redundancies, partially-static redundancies, and fully-static redundancies. Run-time redundancy is the most general form: loads or stores are redundant if in program execution, they access the same memory address and use the same value as a previous memory operation. This is shown in the upper left corner of Figure 2, where operation P and Q access same memory address and operate on the same value. Sometimes, static analysis can prove that on some control flow path from P to Q, P and Q operate exactly on the same address and value. This case, called partially static redundancy, is shown in the upper center of Figure 2. If static analysis can prove that P occurs on <u>all</u> possible control-flow paths to Q, and that P and Q access same address with same value, then Q is a fully static redundancy, shown in the upper right corner of Figure 2. Generally, for arbitrary P and Q, it is impossible to decide statically whether they will access same address with same value. The Venn diagram at the bottom of Figure 2 shows the relationship between these three categories.

[*]This work was supported by the DOE through the Los Alamos Computer Science Institute.

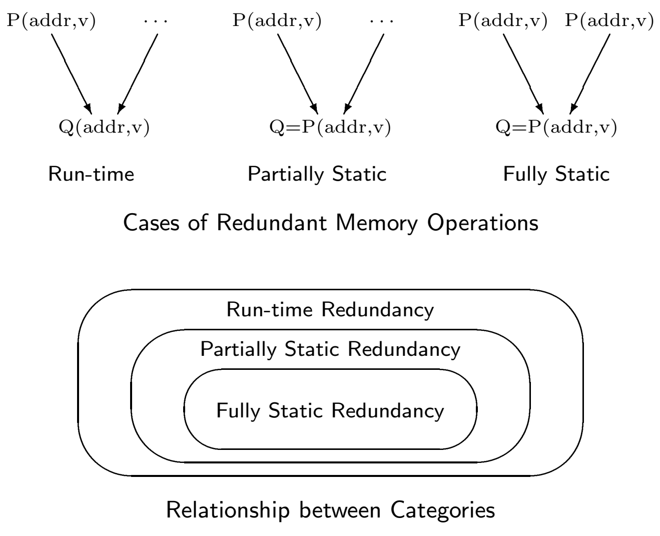

Figure 2: Classifying Redundancies

Both hardware [18, 23, 31, 36] and software techniques [25, 29, 24, 4, 6] have been proposed to detect and remove redundant memory operations. In their dynamic instruction reuse work [31], Sodani et al propose to use hardware lookup tables to store operation input and result. A redundant operation is detected by matching input against lookup table entries. If there is a match, the operation is canceled and earlier result is reused. For loads, memory address can be used as lookup input; for stores, address and store value should be used to match any earlier load and store to determine redundancy. Yang et al describe a more elaborate hardware scheme for dynamic load redundancy removal [36]. Although hardware methods have the advantage that they can see run-time memory addresses and values, they do require significant hardware support. The limited lookup table size also limits the scope over which they can detect redundancy. Thus, they capture a subset of the run-time redundancy suggested in Figure 2. In contrast, software methods target the partially static and fully static redundancies. They remove the redundancies with transformations that rewrite the code. This paper presents a static technique to detect redundant memory operations.

Previous work on static memory redundancy analysis targets the problem in isolation – memory operations are inspected separately from scalar operations. However, as the address and value involved in memory operations are either computed or used by scalar operations, we believe, the scalar and memory redundancy detection should be considered together rather than separately. In this paper, we present a unified global redundancy detection algorithm based on optimistic global value numbering, which is at least as powerful as the isolated approach, and is capable of detecting both scalar and memory redundancy at the same time. This avoids multiple iterations when using separate passes to detect scalar and memory redundancy. Further more, traditional scalar redundancy removal transformations can be easily extended to use the analysis result to remove redundant memory operations in the same way as scalar operations, eliminating the need for a dedicated memory redundancy removal phase.

```
 0   FRAME 0 => r2 r3 [@pa_0 @pa_4 @pb_0 @pb_4 @ra_0 @rb_0]
10   i2i r3 => r4 # get result
10   i2i r2 => r5 # get p
10   i2i r5 => r6
10   iLD r6 => r7 [@pa_0 @pb_0]
10   uADDI 4 r5 => r8
10   iLD r8 => r9 [@pa_4 @pb_4]
10   iADD r7 r9 => r10
10   iST r4 r10 [@ra_0 @rb_0] [ @ra_0 @rb_0]
11   uADDI 4 r3 => r11
11   i2i r2 => r12
11   iLD r12 => r13 [@pa_0 @pb_0]
11   uADDI 4 r2 => r14
11   iLD r14 => r15 [@pa_4 @pb_4]
11   iSUB r13 r15 => r16
11   iST r11 r16 [@ra_0 @rb_0] [@ra_0 @rb_0]
12   RTN
```

Figure 3: ILOC Code for `Compute`

The remainder of the paper is organized as follows: Section 2 introduces the intermediate representation we use to detect program redundancy, with a focus on memory related operations; Section 3 gives an overview of SCC-based value numbering which forms the basis for our global redundancy detection scheme; Section 4 describes a new SSA form for memory operations, and details the key memory value numbering algorithm; Section 5 demonstrates how the widely used scalar common subexpression elimination can be modified to remove those fully static redundant memory operations, and evaluates the redundancy detection and removal algorithms on a suite of realistic benchmarks; Section 6 discusses related work, and Section 7 concludes the paper.

2. INTERMEDIATE REPRESENTATION

Our compiler uses a low-level, RISC-style, three-address intermediate representation, called ILOC. All memory accesses in ILOC occur on load and store operations. The other operations work from an unlimited set of virtual registers. The ILOC code for the example function `Compute` is shown in Figure 3. The line number indicates the source line in the C code of Figure 1.

For static analysis to detect that two memory operations P and Q are equivalent, memory alias information must be considered. (The absolute access address and value are generally not available at compile time.) To account for aliasing effects, an explicit list of memory objects, called an M-list, is associated with each memory operation (loads, stores, and calls). The M-list indicates the possible set of memory objects that the operation may affect. Our algorithm assumes that an external alias analysis is performed to disambiguate memory and compute M-lists for memory operations.

The results reported in this paper were generated with a flow-insensitive, context-insensitive, Andersen-style pointer analysis [3] for C programs.[1] The pointer analysis generates, as its result, the M-lists for loads and stores. In addition, it

[1]The whole program pointer analysis includes C library functions which are summarized in stubs and treated polymorphically. C array objects are treated as a single entity; array elements are not distinguished. For C struct objects, every scalar and array field is treated as a separate object. Heap objects are classified by their allocation sites; all objects from a single site are treated as a single entity.

computes a conservative estimate of the REF and MOD sets, in terms of memory objects, for each C function.

The rules to construct M-list for memory operations are listed as follows. Greek letters (α, β, γ, ...) represent memory objects and virtual register names (r_1, r_2, ..., r_n) represent scalar values.

FRAME frame_size $\Rightarrow$ arg_list[r_1 ...] M-def[α ...]

FRAME is a pseudo-operation that generates no executable code. It records the function's activation record size, its arguments, and its REF and MOD information. The M-def list of FRAME of function f is defined as M-def(FRAME) = REF(f) $\cup$ MOD(f). REF(f) is the set of all memory objects that might be referenced by an execution of f, or indirectly through a procedure called by f. MOD(f) is the set of all memory objects that might be modified by an execution of f, or indirectly through a procedure called by f. These sets are computed by the whole program pointer analysis described above. Intuitively, M-def(FRAME) is the set of memory objects accessed by f, either in REF or MOD.

JSRl *label* arg_list[r_1 ...] $\Rightarrow r_o$ M-use [α ...] M-def[β ...]
JSRr r_f arg_list[r_1 ...] $\Rightarrow r_o$ M-use [α ...] M-def[β ...]

The JSRl operation implements a direct function call. JSRr implements an indirect function call to the address stored in r_f. The arg_list contains function arguments, r_o is the return result (if there is one). For a call to f, M-use = REF(f), and M-def = MOD(f). If the call is ambiguous, these sets include the union of the corresponding set, taken over all possible targets of the JSR.

xLD $r_a \Rightarrow r_v$ M-use[α ...]

xLDs are the family of load operations in ILOC. x designates a specific load instruction. ILOC supports signed and unsigned loads of bytes, half-words, words, and double-words. r_a is the load memory address. r_v is the load result. M-use is the set of memory object names that the load may read. M-use is computed by the pointer analysis.

xST r_a r_v M-use[α ...] M-def[α ...]

xSTs are the family of store operations in ILOC, where x can take on the same values as in a load. r_a is the memory address defined by the store. r_v is the value to be stored at r_a. M-use and M-def are identical. They contain the set of memory objects that the store may define.

The M-lists for memory operations of `Compute` are shown in the brackets in Figure 3. For example, `iLD r6 => r7 [@pa_0 @pb_0]` means the load may read from object @pa_0 or @pb_0 using base address in `r6` and the load result is put in `r7`. Our front end encodes C struct field by its offset, @pa_0 and @pb_0 correspond to `pa.x` and `pb.x` in Figure 1.

3. SCC-BASED VALUE NUMBERING

To prove that memory operation P is redundant in terms of Q, static analysis must decide that the address and value they operate upon are equivalent. However, in the presence of memory aliasing, the address and value can only be determined symbolically. Addresses that differ symbolically may, in fact, produce the same address at run-time. Thus, the analysis must further prove that the memory states before and after execution of P will be equivalent to those surrounding the execution of Q. Our algorithm uses value numbering to discover both address and value equivalence and memory state equivalence. The key property of value numbering is: *If two scalar values are assigned the same value number, they must have same value at run time.* To extend value numbering to memory objects, we must ensure the analogous property for memory objects: *if two memory objects in the M-list are assigned the same value number, they are guaranteed to have the same memory states at run time.*

The new algorithm builds on Simpson's SCCVN algorithm for optimistic global value numbering [30]. It discovers value-based identities (as opposed to lexical identities) [13]. Simpson's algorithm is, arguably, the strongest global technique for detecting redundant scalar values. It finds all the redundancies discovered by the Alpern-Wegman-Zadeck algorithm [2]. It finds a broad class of algebraic identities. It discovers all the constants found by the sparse simple constant algorithm [34]. SCCVN has been implemented in a number of compilers.

The SCCVN algorithm extends Simpson's dominator-based technique (DVNT) to a global scope [7, 30]. It abstracts cycles out in the control-flow graph (CFG) and iterates over each cycle's internal structure to find a fixed-point solution for that cycle. That solution then factors back into the global propagation of value numbers.

The algorithm assumes the presence of both a CFG and the SSA graph. It constructs a reduced CFG by replacing each cycle in the CFG with a single node that represents it. The reduced CFG is acyclic. SCCVN visits all the nodes in the reduced CFG in reverse postorder and value numbers them. When it encounters a node that represents a cycle in the original CFG, it iterates over the basic blocks in the cycle, value numbering as it goes. Figure 4 shows the algorithm.

SCCVN maintains two key data structures: the value table and the operation table. The value table maps each SSA name to a value number. The process that generates value numbers guarantees that if two SSA names have the same value number, they will always have the same value at run time. The operation table maps a tuple, containing an opcode and the value numbers of its operands, into a value number. It is used to discover redundant operations. If the current operation matches an earlier entry in the operation table, then the current operation must be redundant with that earlier operation.

4. VALUE NUMBERING MEMORY

The new algorithm extends SCCVN so that it computes value numbers for memory operations and uses the results to find redundant memory operations. By value numbering memory operations along with scalar operations, the new algorithm can trace value through memory and prove value equality in the global scope. Because it combines knowledge from register-based scalar values and from memory-based values in the same analysis, it can find redundancies

```
ValueNumberOneSCCGroup($SCC$)
{
   do {
      set boolean flag $changed$ = false;
      for each $BLOCK$ in $SCC$ {
         ValueNumberOneBasicBlock($BLOCK$);
         if (value_table has changed during numbering)
            $changed$ = true;
      }
   } while ($changed$);
}

ValueNumberOneBasicBlock($BLOCK$)
{
   for all $\phi$ nodes in $BLOCK$ {
      /* value number $\phi$: $ssa_\phi \Leftarrow \Phi(ssa_1, ssa_2, \cdots)$ */
      ignore $\phi$ args not numbered yet;
      if numbered args have same value number $ssa_v$
         value_table[$ssa_\phi$] = $ssa_v$;
      else
         value_table[$ssa_\phi$] = $ssa_\phi$;
   }
   for all scalar $op$ in $BLOCK$ in execution order {
      /* value number $op$: $r_3 \Leftarrow opcode\ r_1\ r_2$ */
      use tuple $\langle opcode$, value_table[$r_1$], value_table[$r_2$]$\rangle$
         to lookup operation_table;
      if there is a match with result $v$
         value_table[$r_3$] = $v$;
      else
         value_table[$r_3$] = $r_3$;
         add tuple $\langle opcode$, value_table[$r_1$], value_table[$r_2$]$\rangle$
            to operation_table with result $r_3$;
   }
}
```

Figure 4: SCC-based Scalar Value Numbering

that separate analysis of register-based and memory-based analysis would miss.

4.1 Building SSA Form with M-lists

Like SCCVN, the new algorithm works on SSA form. It constructs SSA names for the memory objects in the M-lists as follows: for FRAME, names in M-def are treated as definitions; for JSR operations, names in M-use are treated as uses, and names in M-def are treated as definitions; for xLD, names in M-use are treated as uses; for xST, names in M-use are treated as uses, and names in M-def are treated as definitions. The construction inserts ϕ-functions for memory object names, just as it does for virtual registers.

The uses and definitions for memory operations are defined so that SSA naming of memory objects represents a flow-sensitive description of memory states for those memory operations. In particular, if two memory operations have the same SSA name for one memory object in their uses, then the state of that memory object must be same before the execution of the operations. The M-def set for FRAME can be considered as the initial states of all possible memory objects accessed by the function. Because JSR and xST operations can change the states of memory objects in M-def, the SSA construction conservatively assumes those operations do change the states of those memory objects. Thus, it treats M-def as definition set. Figure 5 shows the SSA form of Compute.

```
 0 FRAME 0 => r2 r3 [@pa_0 @pa_4 @pb_0 @pb_4 @ra_0 @rb_0]
10 i2i r3 => r4 # get result
10 i2i r2 => r5 # get p
10 i2i r5 => r6
10 iLD r6 => r7 [@pa_0 @pb_0]
10 uADDI 4 r5 => r8
10 iLD r8 => r9 [@pa_4 @pb_4]
10 iADD r7 r9 => r10
10 iST r4 r10 [@ra_0 @rb_0] [@ra_0_1 @rb_0_1]
11 uADDI 4 r3 => r11
11 i2i r2 => r12
11 iLD r12 => r13 [@pa_0 @pb_0]
11 uADDI 4 r2 => r14
11 iLD r14 => r15 [@pa_4 @pb_4]
11 iSUB r13 r15 => r16
11 iST r11 r16 [@ra_0_1 @rb_0_1] [@ra_0_2 @rb_0_2]
12 RTN
```

Figure 5: SSA Form for Compute

Of course, in some contexts, those operations will not modify the states of those memory objects, *e.g.*, they might store the same value into the same memory objects. The new algorithm discovers such facts, propagates that knowledge along edges in the SSA graph.

4.2 SCC-based Memory Numbering

To extend SCCVN to handle memory operations, we must modify the basic-block value-numbering algorithm to deal with the M-list on memory operations, and to number the ϕ-functions for memory objects. The modified version is shown in Figure 6. Because the new algorithm extends SCCVN, it inherits the optimistic nature of that algorithm. Like SCCVN, it finds the maximum fixed point for cycles, except that it handles both scalar values and memory states.

The extensions to **ValueNumberOneBasicBlock** are as follows:

FRAME frame_size $\Rightarrow$ arg_list[r_1 ...] M-def[α ...]

For any $\alpha \in$ M-def, set value_table[α]=α.

This creates the needed initial state. The value-numbering algorithm operates on SSA names for both register-based and memory-based objects. Since SSA names are unique, we can use them directly as value numbers.

JSRl *label* arg_list[r_1 ...] $\Rightarrow r_o$ M-use [α ...] M-def[β ...]
JSRr r_f arg_list[r_1 ...] $\Rightarrow r_o$ M-use [α ...] M-def[β ...]

For any $\beta \in$ M-def of JSRl or JSRr, set value_table[β]=β. If there is any return value, set value_table[r_o]=r_o.

This reflects the conservative assumption that executing the call might change the memory state for any object in the M-def list. It also assigns a new value number to r_o, to reflect the return value.

```
ValueNumberOneBasicBlock($BLOCK$)
{
  for all $\phi$ nodes (including memory $\phi$) in $BLOCK$ {
    value number $\phi$ as before;
  }
  for all op in $BLOCK$ in execution order {
    if op is scalar
      value number op as before;
    if op is FRAME
      value number op as in Section 4.2;
    if op is JSRl/r
      value number op as in Section 4.2;
    if op is xLD
      value number op as in Section 4.2;
    if op is xST
      value number op as in Section 4.2;
  }
}
```

Figure 6: Basic Block Memory Value Numbering

xLD $r_a \Rightarrow r_v$ M-use[α ...]

Iloc supports several kinds of load and store operations (different sizes and different address modes). To compare these different operations, the algorithm needs a mechanism that translates an arbitrary load or store into a form where they can be compared. We introduce a function $Norm$ that converts the load/store opcode into a canonical form.

Similarly, a $Value$ function returns the value number for an SSA name or a set of SSA names. The value number of a set of names is just the set of value numbers of the individual SSA names in the set. $Value$(M-use) of xLD and xST represents the memory state before the execution of load and store operations.

For a load op$_1$, "xLD $r_a \Rightarrow r_v$ M-use$_1$", the algorithm constructs a tuple, $\langle Norm(x\mathsf{LD}), Value(r_a)\rangle$, and uses it as a key into the operation table. If there is a matching entry op$_2$, the algorithm tests for the equality of $Value$(M-use$_1$) and $Value$(M-use$_2$). If they are equal, then op$_1$ is redundant, value_table[r_v] = v, where v is the result value of op$_2$. Otherwise, op$_1$ is not redundant, value_table[r_v] = r_v, and a new entry $\langle$ $Norm(x\mathsf{LD})$, $Value(r_a)$, $Value$(M-use$_1$)$\rangle$ is added to the operation table for op$_1$, with result value as r_v.

Intuitively, value numbering for load checks whether the canonical load opcode, generated by $Norm$, load address and M-list object value numbers are matched. If there is a match, the early operation has exactly same memory states and operates on the same memory location, thus the current load is redundant and the previous result value is reused; if there is no match, the canonical opcode, load address and M-list object value numbers, and load result value number are added to the operation table.

xST r_a r_v M-use[α ...] M-def[α ...]

For a store op$_1$, "xST r_a r_v M-use$_1$ M-def$_1$", the algorithm constructs the tuple $\langle Norm(x\mathsf{ST}), Value(r_a)\rangle$ and uses it as a lookup key into the operation table. For any matching op$_2$ with result value as $Value(r_v)$, the algorithm checks whether $Value$(M-use$_1$) = $Value$(M-use$_2$). If they are equal, then op$_1$ is redundant and the algorithm must set

$$\text{value\_table}[\alpha] = \text{value\_table}[\alpha\prime],$$

for $\alpha \in$ M-def$_1$, where $\alpha\prime \in$ M-use$_1$ and α and $\alpha\prime$ are SSA names for the same memory objects.

If $Value$(M-use$_1$) $\neq$ $Value$(M-use$_2$), then op$_1$ is not redundant, and for each $\alpha \in$ M-def$_1$, it sets value_table[α] = α, and a new entry $\langle Norm(x\mathsf{ST}), Value(r_a), Value(\text{M-def}_1)\rangle$ is added for op$_1$, with result value $Value(r_v)$.

The value numbering for store checks the canonical store opcode, generated by $Norm$, the store address, store value, and M-list value numbers against an earlier load or store operation to detect redundancy. If xST is redundant, the state of objects in the M-def set is not changed, and the algorithm propagates its numbering from the M-use set to the M-def set; if xST is non-redundant, objects in the M-def set needs to assign new value number to indicate new memory state.

4.3 Correctness

The value numbering process maintains two invariants with regard to the M-list sets. First, the memory state immediately prior to the execution of a memory operation is represented by $Value$(M-use) for both xLD and xST. Second, the memory state immediately after execution of a memory operation is represented by $Value$(M-use) for an xLD or $Value$(M-def) for an xST. The lookup keys for the operation table are constructed from the combination of canonical opcode, value number of memory address, value number of memory contents, and the value number of the appropriate M-list. The algorithm only considers two operations to be redundant if they have the same key—that is, they perform the same operation on the same address, value, and memory state.

4.4 Time Complexity

The time complexity of the new algorithm is determined by the time needed to manipulate the SSA value table and the operation table. Let N be the total number of operations, M be the total number of SSA names in the function, αM ($0 \leq \alpha < 1$) be the number of memory object SSA names accessed by the function (same as |M-def| of FRAME), and $D(G)$ be the loop connectedness of the control flow graph G [19]. We assume that hashing requires constant time, an assumption justified by our experience with the implementation.[2] Simpson showed that the time complexity attributable to building the SSA value table is $O(M \times D(G))$.

The time attributable to building the operation table can be bounded by determining the time complexity for operation lookup and update. Since each scalar operation has at most 3 operands, the lookup and update is $O(1)$. For load and store operations, as the size of M-list set is $O(\alpha M)$. a lookup is $O(\alpha M)$ and an update is $O(\alpha M)$. Thus the time complexity attributable to the operation table is

[2]If this becomes problematic, the implementor can use multi-set discrimination to guarantee O(1) access [9].

SSA	Value Number
r2	r2
r3	r3
r4	r3
r5	r2
r6	r2
r7	r7
r8	r8
r9	r9
r10	r10
r11	r11
r12	r2
r13	r7
r14	r8
r15	r9
r16	r16

SSA	Value Number
@pa_0	@pa_0
@pa_4	@pa_4
@pb_0	@pb_0
@pb_4	@pb_4
@ra_0	@ra_0
@rb_0	@rb_0
@ra_0_1	@ra_0_1
@rb_0_1	@rb_0_1
@ra_0_2	@ra_0_2
@rb_0_2	@rb_0_2

Figure 7: Value Table

$O(\alpha MN \times D(G))$, and the overall complexity for SCC-based memory value numbering is $O((\alpha MN + M) \times D(G))$. In practice, $D(G)$ is bounded by a small constant; loads and stores only account for a fraction of total operations; and the M-use and M-def set can often be bounded in size by a small constant, so we expect linear complexity of $O(N+M)$ for most applications.

4.5 Discussion

The value table and operation table for `Compute` after value numbering are shown in Figure 7 and Figure 8. For the first and third loads, the algorithm proves they use equivalent addresses (`r2`) and have the same memory states (`@pa_0` and `@pb_0`); thus the third load is redundant. The same holds for the second and fourth loads. As shown in this example, detection of redundant memory operation requires tracing of flow of scalar values (addresses and memory values); on the other hand, load results are used in other scalar computations, and the ability to trace value through memory operations therefore contributes to the detection of scalar redundancy.

Our new algorithm extends the value-based redundancy detection of Simpson's algorithm to handle memory-based values. It relies on value identity, rather than lexical identity. Because it unifies the treatment of register-based values and memory-based values, it can detect equalities that separate analyses cannot find. The power of the underlying algorithm, Simpson's SCCVN algorithm, ensures that it can find as many register-based redundancies as other algorithms. It also finds a class of redundancies involving memory-based values that Simpson's original algorithm misses. An approach that separates register-based redundancy detection from memory-based redundancy detection might discover many of the same equalities by repeating some sequence of *register-based analysis, memory-based analysis, register-based analysis, . . .*, until no further improvements are found. This repetition might be required to trace values from registers through memory and back to registers. Unifying these two analyses eliminates the need to iterate between them.

Opcode	Operand	result
iLD	r2, @pa_0, @pb_0	r7
uADDI	4, r2	r8
iLD	r8, @pa_4, @pb_4	r9
iADD	r7, r9	r10
iLD	r3, @ra_0_1, @rb_0_1	r10
uADDI	4, r3	r11
iSUB	r7, r9	r16
iLD	r11, @ra_0_2, @rb_0_2	r16

Figure 8: Operation Table

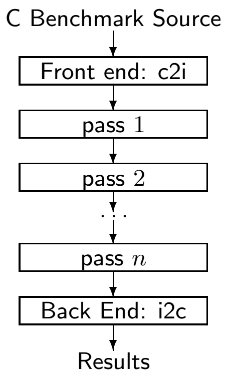

Figure 9: ILOC Execution Model

5. EXPERIMENTAL RESULTS

The execution model for our compiler system is depicted in Figure 9. The C front end (c2i) converts the program into ILOC. The compiler applies multiple analysis and optimization passes to the ILOC code. Finally, the back end generates executable in C form to emulate ILOC code.

5.1 Value-based CSE

Our new algorithm discovers redundancies through value numbering. Because it uses value equality and ignores control flow, it requires a separate phase to discover which operations can actually be removed. This follow-on transformation must incorporate information about control flow that allow it to distinguish between a partially-redundant and a fully-redundant operation. Two candidates for this redundancy removal phase are traditional common subexpression elimination (CSE) [13, 1] or partial redundancy elimination [26, 20]. For this work, we implemented the classic CSE framework, reworked to reflect the fact that the equations are operating on an SSA-based name space where kills cannot occur. With information of redundant memory operations, it is now capable to remove the fully static memory redundancy as shown in Figure 2.

Figure 10 shows the equations used for value-based CSE. (Notice the absence of kills.) To identify fully redundant memory operations, for each memory operation in the operation table, we assign it a unique ID number (not overlapped

$$\text{AVLOC}_i = \text{computed locally as in Section 5.1}$$

$$\text{AVIN}_i = \begin{cases} \emptyset & \text{if } i \text{ is the entry block;} \\ \bigcap\limits_{j \in pred(i)} \text{AVOUT}_j & \text{otherwise.} \end{cases}$$

$$\text{AVOUT}_i = \text{AVIN}_i \cup \text{AVLOC}_i$$

Figure 10: CSE Data Flow Equation System

Benchmark	Description	Input	func	SCC	block	scalar op	call op	load op	store op
adpcm	16-to-4 bit voice encoding	clinton.pcm	2	12	55	69	5	6	4
g721	CCITT G.721 voice encoding	clinton.pcm	14	180	197	637	27	59	58
gsm	GSM speech encoding	clinton.pcm	57	769	1125	3833	162	664	288
epic	Pyramid image encoding	test_img.pgm	36	523	1099	1569	190	307	121
pegwit	Elliptic curve public key encryption	news.txt	96	1089	1474	6363	456	1371	361
mpeg2dec	MPEG-2 video decoding	child.mpg	113	1645	2465	5080	660	1212	516
181.mcf	Combinational optimization	test input	24	313	611	1205	81	392	213
164.gzip	Compression	test input	60	969	1649	3841	303	970	496
256.bzip2	Compression	test input	62	718	1626	3695	302	796	293
175.vpr	FPGA placement and routing	test input	255	3095	5610	14428	1903	4632	1001

Table 1: Test Benchmarks and ILOC Statistics

Benchmark	Total Instruction Count			Load Count			Store Count		
	v_0	v_1	v_1/v_0	v_0	v_1	v_1/v_0	v_0	v_1	v_1/v_0
adpcm	8,091,930	8,091,930	100%	443,004	443,004	100%	74,056	74,056	100%
g721	283,191,957	281,696,433	99.5%	27,609,466	26,113,942	95%	4,936,714	4,936,714	100%
gsm	213,363,814	212,579,040	99.6%	33,325,763	32,613,964	98%	4,150,355	4,150,355	100%
epic	54,548,643	54,478,173	99.9%	6,612,912	6,542,419	99%	608,135	608,135	100%
pegwit	1,683,188	1,545,808	92%	346,052	208,672	60%	91,539	91,539	100%
mpeg2dec	158,374,708	157,488,222	99.4%	20,294,127	19,477,705	96%	3,087,227	3,087,226	100%
181.mcf	182,417,210	177,782,197	97%	43,121,895	40,101,387	93%	7,046,815	7,046,815	100%
164.gzip	2,971,447,783	2,821,169,722	95%	587,721,573	439,438,112	75%	192,368,544	192,368,539	100%
256.bzip2	10,787,948,463	10,652,916,923	98%	1,888,455,044	1,773,172,428	94%	1,267,983,097	1,267,983,097	100%
175.vpr	807,194,517	698,340,504	87%	251,385,602	177,868,051	71%	45,495,883	45,495,883	100%

Table 2: Dynamic Instruction Count

with any scalar value number); for each memory operation not in the operation table, we assign it the ID of the memory operation with which it is redundant. Using the value table and memory ID assigned to memory operations, the AVLOC$_i$ set for block i is computed as follows: 1) for a scalar operation s in block i, if s defines value r_v, then $r_v \in$ AVLOC$_i$; 2) for a memory operation in block i with ID m, $m \in$ AVLOC$_i$, furthermore, if it is an xLD with result value r_v, then $r_v \in$ AVLOC$_i$.

When the equations in Figure 10 are solved, the AVIN$_i$ set contains the available value and memory ID at the entry of block i. Fully redundant operations can be detected and removed by scanning the operations in block i in execution order as follows: (1) if scalar operation s computes $r_v \in$ AVIN$_i$, s is redundant and removed, otherwise, add r_v to AVIN$_i$; and (2) if memory operation with ID $m \in$ AVIN$_i$, m is redundant and removed, otherwise, add m to AVIN$_i$, furthermore, for load, add result value r_v to AVIN$_i$. In the example for `Compute`, the third and fourth loads have same ID with the first and second load, and thus are removed by CSE. Figure 11 shows the code for `Compute` after CSE.

```
0   FRAME 0 => r2 r3
10  iLD r2 => r7
10  uADDI 4 r2 => r8
10  iLD r8 => r9
10  iADD r7 r9 => r10
10  iST r3 r10
11  uADDI 4 r3 => r11
11  iSUB r7 r9 => r16
11  iST r11 r16
12  RTN
```

Figure 11: `Compute` after Redundancy Removal

5.2 Benchmarks

We integrated the new algorithm followed with CSE as a single pass on ILOC, referred to as v_1. For comparison, we also implemented the scalar SCCVN with CSE, referred to as v_0. We tested the two passes on 10 benchmarks, 6 from Mediabench [22], 4 from SPEC2000 CPU integer benchmarks [32]. The benchmarks and the statistics of the compiled ILOC intermediate form for the applications are listed in Table 1.

The applications are first translated from C into ILOC, and a sequence of sparse conditional constant propagation [34], dead code elimination, control flow restructuring (to remove empty and unreachable basic blocks), copy coalescing passes are applied to the ILOC. After that, whole program pointer analysis is run to get the annotated ILOC with point-to and function REF and MOD information. The v_0 and v_1 pass works on the annotated ILOC after pointer analysis. The output ILOC code is then run through dead code elimination, peephole optimization, copy coalescing, and passed to the back end to generate final executables.

5.3 Results

The dynamic total instruction count and instruction count of loads and stores are shown in Table 2. The difference of total instruction count and memory instruction count are shown in Table 3.

The dynamic instruction counts show that all the benchmarks except *adpcm* had significant reductions in run-time memory operations from doing CSE based on memory value numbering (v_1), when compared to CSE with scalar value numbering only (v_0). The column labeled $\mathcal{I}-\mathcal{M}$ shows that six of the benchmarks (*gsm*, *mpeg2dec*, *181.mcf*, *164.gzip*, *256.bzip2*, *175.vpr*) have a larger difference in total instruction count than in memory operation count. This suggests that, for those applications, detection of memory redun-

Benchmark	Difference in Op Counts		$\mathcal{I} - \mathcal{M}$
	Total Ops $\mathcal{I}$ $(v_0 - v_1)$	Memory Ops $\mathcal{M}$ $(v_0 - v_1)$	
adpcm	0	0	0
g721	1,495,524	1,495,524	0
gsm	784,774	711,799	+72,975
epic	70,470	70,493	-23
pegwit	137,380	137,380	0
mpeg2dec	886,486	816,423	+70,063
181.mcf	4,635,013	3,020,508	+1,614,505
164.gzip	150,278,061	148,283,466	+1,994,595
256.bzip2	135,031,540	115,282,616	+19,748,924
175.vpr	108,854,013	73,517,551	+35,336,462

Table 3: Dynamic Instruction Count Difference

dancy also leads to detection of more scalar redundancy.

To understand why memory value numbering finds no opportunity in *adpcm*, we inspected its source code. Interestingly, in the key encoding function `adpcm_coder`, the programmer stores the values of frequently used global variables into local variables which the compiler maps to virtual registers. In the kernel computation loop, the values in the local variables are used rather than loading the values from the global variables. In this way, the programmer has manually rewritten the function to perform exactly the transformation that the new algorithm would do: find redundant memory operations and reuse the results in registers.

As the data from Table 3 show, for *g721* and *pegwit*, the entire difference in instruction counts occurs in memory operations. This suggests that the redundant memory instructions do not lead to discovery of more redundant scalar instructions. For *epic*, the memory instruction count difference is slightly larger than total instruction count difference, as some eliminated memory operations are replaced with scalar operations.

For all six other benchmarks, the total instruction count difference is significantly larger than the memory instruction count difference. This suggests detection of redundant memory operations leads to discovery of additional redundant scalar operations. In these cases, it would take the separate approach multiple iterations to detect the same set of scalar and memory redundancy.

Our execution model does not model the microarchitecture. As memory operations become more expensive relative to arithmetic, reduction in memory operations should produce more run-time performance improvement. (We have run the smaller codes on a version of the SimpleScalar simulator, which confirms the results stated here in terms of raw operations. If the committee desires, the full paper can include SimpleScalar results. We omitted them because of space limitations and the concern that they easily lead to comparisons against other SimpleScalar simulations with different parameters, such as memory latency.)

The data in Table 2 and 3 also show that most redundant memory operations discovered by our technique are loads. In fact, across all benchmarks, only *mpeg2dec* and *164.gzip* had redundant stores (one and five, respectively). Tracing

```
From mpeg2dec.c:
712 static void Clear_Options()
713 {
714     Verbose_Flag = 0;
715     Output_Type = 0;
716     Output_Picture_Filename = " ";
717     hiQdither = 0;
718     Output_Type = 0;
719     ...

From inflate.c:
153 #define flush_output(w) (wp=(w),flush_window())
    ...
946 flush_output(wp);
947 ...
```

Figure 12: mpeg2dec and 164.gzip Source Code

back to the source, we discovered the redundant stores are mapped to the source code of line 718 of `mpeg2dec.c` in *mpeg2dec* and line 946 of `inflate.c` in *164.gzip* as shown in Figure 12. A careful code review would have caught these cases[3], however, it does highlight the need for automatic analysis to detect and remove such inefficiencies.

5.4 Algorithm Run Time Analysis

To assess the impact of memory value numbering on compile time, we measured the running time of the SSA construction, SCC value numbering, and CSE removal phase of v_0 and v_1. Because the absolute running time is very fast, we intentionally measured the times on a relatively slow machine – a lightly loaded SUN Ultra-1 workstation with 140MHz clock and 240MB memory. The results are shown in Table 4. These results show, that the key phases of memory value numbering are fast.

Time (second)	SSA		Value Numbering		CSE	
	v_0	v_1	v_0	v_1	v_0	v_1
adpcm	0.00	0.00	0.00	0.00	0.00	0.00
g721	0.00	0.00	0.00	0.00	0.01	0.00
gsm	0.00	0.01	0.04	0.08	0.09	0.12
epic	0.01	0.00	0.04	0.08	0.06	0.06
pegwit	0.00	0.02	0.03	0.05	0.02	0.02
mpeg2dec	0.01	0.01	0.02	0.06	0.06	0.08
181.mcf	0.00	0.00	0.01	0.03	0.02	0.02
164.gzip	0.01	0.04	0.04	0.37	0.11	0.10
256.bzip2	0.00	0.00	0.04	0.11	0.06	0.07
175.vpr	0.02	0.04	0.13	0.54	0.24	0.23

Table 4: Running Times for Component Phases

We also studied the average number of SCC iterations and the average number of table lookup and update steps for scalar and memory operations. The results in Table 5 show that the SCC iterations reach a fixed point very quickly for all applications (within 2 iterations on average). The average of table lookup and update for memory operations are also quite low. To understand how large the M-list sets are for real applications, we also collected the set size for memory related operations. The results are shown in Table 6.

[3]The *164.gzip* case is more difficult, the same macro expansion is used in several other places which do not cause redundancy.

Average	SCC iter		scalar lookup		scalar update		load lookup	load update	store lookup	store update
	v_0	v_1	v_0	v_1	v_0	v_1	v_1	v_1	v_1	v_1
adpcm	1.33	1.42	3.25	3.45	1.10	1.10	2.50	1.17	1.75	1.75
g721	1.07	1.08	1.26	1.26	0.55	0.55	1.34	0.76	1.34	1.34
gsm	1.14	1.14	1.91	1.91	0.84	0.84	1.88	0.78	1.45	1.45
epic	1.20	1.22	2.32	2.51	0.94	0.95	2.11	0.68	2.36	2.36
pegwit	1.11	1.12	1.38	1.40	0.57	0.57	1.38	0.81	1.55	1.55
mpeg2dec	1.10	1.10	1.79	1.86	0.74	0.72	1.67	0.68	1.52	1.52
181.mcf	1.26	1.27	2.47	2.53	0.80	0.77	2.70	0.90	2.54	2.54
164.gzip	1.18	1.19	2.06	2.19	0.72	0.70	2.17	0.78	1.92	1.92
256.bzip2	1.25	1.26	2.70	2.82	0.72	0.70	2.82	0.74	2.82	2.76
175.vpr	1.16	1.17	2.28	2.43	0.69	0.60	2.42	0.65	2.30	2.29

Table 5: SCC Iterations, Table Lookup and Update

Avg Size	ref	mod	func M-def	call M-use	call M-def	load M-use	store M-def
adpcm	6.00	3.50	6.50	1.40	0.80	1.00	1.00
g721	5.07	3.79	5.86	2.74	1.70	1.02	1.00
gsm	14.14	6.25	15.49	5.19	2.20	1.03	1.06
epic	9.42	2.72	9.64	2.97	0.92	1.08	1.11
pegwit	26.07	16.25	27.15	14.31	8.95	4.60	4.71
mpeg2dec	38.19	22.50	45.28	16.46	9.05	1.45	1.40
181.mcf	12.71	6.96	13.46	4.22	2.38	1.99	1.65
164.gzip	30.82	16.60	32.70	15.72	7.68	1.09	1.07
256.bzip2	24.81	8.31	25.71	10.89	3.13	1.01	1.02
175.vpr	30.40	8.78	31.55	9.32	2.50	1.23	1.28

Table 6: M-list Set Size

The data shows that, for load and store operations, the M-use and M-def set size is very small. These measurements explain why the new algorithm runs so quickly. As the data suggest, it has essentially linear time complexity in practice.

6. RELATED WORK

Program redundancy detection and removal have long been studied in literature [1, 13, 26, 20, 5]. However, most algorithms only deal with unambiguous scalar values. Aliased memory operations cause the algorithms to use worst case assumptions, *i.e.*, all scalar values related to memory would be killed by an aliased store. On the other hand, work on register promotion focuses solely on memory redundancy [25, 24, 29]. As our experimental results show, these two kinds of redundancy interact, so detecting one often allows the compiler to detect the other. The published algorithms for register promotion work from a lexical notion of identity, rather than from a value-based identity. The value-based approach taken in our algorithm should reveal a larger set of equivalent expressions. In [4, 6], Bodik *et al.* use a value numbering based path-sensitive analysis to detect both scalar and memory redundancy based on value equality. However, the algorithm mainly targets array-oriented applications where array subscripts can be represented as linear algebraic expressions and can be analyzed symbolically. They also treat aliased store unsafely (values will not be killed even the store does change those values), and depend on data speculation support from hardware to maintain program correctness. In comparison, our algorithm targets general purpose applications and the analysis is conservative and safe.

Other researchers have also proposed using SSA form to represent aliasing information [14, 11]. These extended SSA representations separate the aliased memory object information from the memory operation itself. Thus they require significant numbers of new SSA names to model the possible alias effects, and the authors propose various solutions to resolve the problem. In contrast, in our M-list representation, the aliased memory object information is directly associated with memory operation it affects, there is no need to invent additional SSA names. The aliasing effect is handled by taking memory object state as additional operand in the value numbering process, which allows our algorithm to not only detect redundant memory operation, but also propagate unchanged state along SSA edges. The value numbering process in [11] does not detect and propagate unchanged states.

Our algorithm assumes the existence of an interprocedural pointer analysis pass. The implementation uses a version of Andersen's algorithm [3]. Nothing in the work relies directly on that algorithm; other styles of pointer analysis would work in this framework [28, 21, 10, 27, 35, 15, 33].

7. CONCLUSION AND FUTURE WORK

This paper presents a powerful, unified approach finding redundancy in both register-based, scalar operations and memory-based operations. By using a SSA-based M-list representation, our algorithm captures accurate information about the state of memory. It can then detect redundant memory operations alongside with scalar redundancies. It can discover all the scalar and memory redundancies found by separate approaches. Our experiments show that interaction exists between scalar and memory redundancies. Our algorithm finds them both in a single analysis phase.

The redundancy relation detected by the new algorithm can then be used to drive a redundancy-removal transformation. We showed that traditional scalar common subexpression elimination can be easily adapted to remove fully static redundant memory operations. A similar approach should adapt partial redundancy elimination to use this information [26, 20] In particular, with the M-list representation, analysis can be developed to identify loop invariant loads and stores, and move them outside the loop. We are building a redundancy-removal transformation based on lazy-code motion to demonstrate the use of memory value numbers in that transformation.

8. ACKNOWLEDGMENTS

This work was performed in the research compiler built over the years by the Massively Scalar Compiler Group at Rice. Many people have contributed to that effort with time, implementation, and insight. All these people deserve our thanks. This effort was funded through the Los Alamos Computer Science Institute, as part of the project on Compilation Issues for High-performance Microprocessors.

9. REFERENCES

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Conference Record of ACM Symposium on Principles of Programming Languages (POPL)*, pages 1–11, 1988.

[3] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, University of Copenhagen, 1994.

[4] Rastislav Bodik and Sadun Anik. Path-sensitive value-flow analysis. In *Conference Record of ACM Symposium on Principles of Programming Languages (POPL)*, pages 237–251, 1998.

[5] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. Complete removal of redundant expressions. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 1–14, 1998.

[6] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. Load-reuse analysis: Design and evaluation. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 64–76, 1999.

[7] Preston Briggs, Keith D. Cooper, and L. Taylor Simpson. Value numbering. *Software Practice and Experience*, 27(6):701–724, June 1997.

[8] Preston Briggs, Keith D. Cooper, and L. Taylor Simpson. Value numbering. *Software–Practice and Experience*, 27(6):701–724, June 1997.

[9] Jiazhen Cai and Robert Paige. Using multiset discrimination to solve language processing problems without hashing. *Theoretical Computer Science*, 145(1&2):189–228, 1995.

[10] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of ACM Symposium on Principles of Programming Languages (POPL)*, 1993.

[11] F. Chow, S. Chan, S. Liu, R. Lo, and M. Streich. Effective representation of aliases and indirect memory operations in ssa form. In *Compiler Construction*, pages 253–267, 1996.

[12] Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(2):181–196, March 1995.

[13] John Cocke. Global common subexpression elimination. *SIGPLAN Notices*, 5(7):20–24, July 1970. In *Proceedings of a Symposium on Compiler Construction.*

[14] Ron Cytron and Reid Gershbein. Efficient accomodation of may-alias information in ssa-form. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 36–45, 1993.

[15] Manuvir Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 35–46, 2000.

[16] Alain Deutsch. Interprocedural May-Alias analysis for pointers: Beyond *k*-limiting. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 230–241, 1994.

[17] Andrei P. Ershov. On programming of arithmetic expressions. *Communications of the ACM*, 1(8):3–6, August 1958. *The figures appear in volume 1, number 9, page 16.*

[18] S.P. Harbison. An architecture alternative to optimizing compilers. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 57–65, 1982.

[19] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, January 1976.

[20] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy code motion. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, pages 224–234, 1992.

[21] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, pages 235–248, 1992.

[22] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.

[23] M.H. Lipasti, C.B. Wilkerson, and J.P. Shen. Value locality and load value prediction. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, 1996.

[24] Raymond Lo, Fred Chow, Robert Kennedy, Shin-Ming Liu, and Peng Tu. Register promotion by sparse partial redundancy elimination of loads and stores. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 26–37, 1998.

[25] John Lu and Keith Cooper. Register promotion in c programs. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, pages 308–319, 1997.

[26] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Commun. ACM*, 22(2):96–103, 1979.

[27] Erik Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 13–22, 1995.

[28] Barbara G. Ryder, William A. Landi, Philip A. Stocks, Sean Zhang, and Rita Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(2), March 2001.

[29] A. Sastry and Roy Ju. A new algorithm for scalar register promotion based on ssa form. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 15–25, 1998.

[30] Taylor Simpson. *Value-Driven Redundancy Elimination*. PhD thesis, Rice University, 1996.

[31] Avinash Sodani and Gurindar S. Sohi. Dynamic instruction reuse. In *International Symposium on Computer Architecture*, pages 194–205, 1997.

[32] SPEC. http://www.specbench.org/osg/cpu2000/cint2000/.

[33] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Conference Record of ACM Symposium on Principles of Programming Languages (POPL)*, pages 32–41, 1996.

[34] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, 1991.

[35] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for c programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 1–12, 1995.

[36] J. Yang and R. Gupta. Load redundancy removal through instruction reuse. In *International Conference on Parallel Processing*, pages 61–68, 2000.