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 lines 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.

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 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 access same memory address and operate on the same 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 all 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 1: An Example Code Fragment
Both hardware [18, 23, 32, 37] and software techniques [25, 30, 24, 4, 6] have been proposed to detect and remove redundant memory operations. In their dynamic instruction reuse work [32], 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 [37]. 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.

The remainder of the paper is organized as follows: Section 2 introduces the intermediate representation used 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. The pointer analysis generates, as its result, the M-lists for loads and stores. In addition, it
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 ($\alpha, \beta, \gamma, \ldots$) represent scalar values.

$$\text{FRAME frame_size } \Rightarrow \text{arg_list[r1 ...] M-def}[\alpha ...]$$

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.

$$\text{JSRI label arg_list[r1 ...] } \Rightarrow r_0 \text{ M-use}[\alpha ...] \text{ M-def}[\beta ...]$$

$$\text{JSRr } r_f \text{ arg_list[r1 ...] } \Rightarrow r_0 \text{ M-use}[\alpha ...] \text{ M-def}[\beta ...]$$

The JSRI 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, $r_0$ 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.

$$x\text{LD } r_a \Rightarrow r_v \text{ M-use}[\alpha ...]$$

$x\text{LDs}$ 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.

$$x\text{ST } r_a \text{ r_v M-use}[\alpha ...] \text{ M-def}[\alpha ...]$$

$x\text{STs}$ 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-def and M-use sets are both called M-list for FRAME, JSR/r, xLD, xST, and they play the key role in our new memory redundancy detection algorithm to reason about memory states and detect equivalence of load and store operations. The M-lists for memory operations of Compute are shown in the brackets in Figure 3. For example, $i\text{LD r6 } \Rightarrow r7 \ [ \text{pa}_0 \text{ pb}_0 ]$ means the load may read from object $\text{pa}_0$ or $\text{pb}_0$ using base address in $r6$ and the load result is put in $r7$. Our front end encodes C struct field by its offset, $\text{pa}_0$ and $\text{pb}_0$ correspond to $\text{pa.x}$ and $\text{pb.x}$ 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 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 [31]. 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 [35]. SCCVN has been implemented in a number of compilers.

The SCCVN algorithm extends Simpson’s dominator-based technique (DVNT) to a global scope [7, 31]. 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 mem-
ValueNumberOneSCCGroup(\textit{SCC})
{
  do {
    set boolean flag \textit{changed} = false;
    for each \textit{BLOCK} in \textit{SCC} {
      ValueNumberOneBasicBlock(\textit{BLOCK});
      if (value_table has changed during numbering)
        \textit{changed} = true;
    }
  } while (\textit{changed});
}

ValueNumberOneBasicBlock(\textit{BLOCK})
{
  for all \(\phi\) nodes in \textit{BLOCK} {
    /* value number \(\phi\): \textit{ssa}_{\phi} \leftarrow \Phi(\textit{ssa}_{1}, \textit{ssa}_{2}, \ldots)\)
    */
    ignore \(\phi\) args not numbered yet;
    if numbered args have same value number \textit{ssa}_{\phi}
      value_table[\textit{ssa}_{\phi}] = \textit{ssa}_{\phi};
    else
      value_table[\textit{ssa}_{\phi}] = \textit{ssa}_{\phi};
  }
  for all scalar \textit{op} in \textit{BLOCK} in execution order {
    /* value number \textit{op}: r_{3} \leftarrow \textit{opcode} r_{1} r_{2}\)
    */
    use tuple \(\langle \textit{opcode}, \textit{value}_{\text{table}}[r_{1}], \textit{value}_{\text{table}}[r_{2}] \rangle\)
      to lookup operation_table;
    if there is a match with result \(v\)
      value_table[r_{3}] = \(v\);\)
    else
      value_table[r_{3}] = \textit{r}_{3};
      add tuple \(\langle \textit{opcode}, \textit{value}_{\text{table}}[r_{1}], \textit{value}_{\text{table}}[r_{2}] \rangle\)
        to operation_table with result \textit{r}_{3};
  }
}

Figure 4: SCC-based Scalar Value Numbering

Figure 5: SSA Form for Compute

4.2 SCC-based Memory Numbering

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

The extensions to \textit{ValueNumberOneBasicBlock} are as follows:

\textbf{FRAME} frame\_size \Rightarrow \text{arg\_list}[r_{1} \ldots] \text{ M\_def}[\alpha \ldots]

For any \(\alpha \in \text{M\_def}\), set \textit{value}_{\text{table}}[\alpha] = \alpha.

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

\textbf{JSR} \textit{r}_{f} \textit{arg\_list}[r_{1} \ldots] \Rightarrow r_{o} \text{ M\_use}[\alpha \ldots] \text{ M\_def}[\beta \ldots]

For any \(\beta \in \text{M\_def}\) of \textit{JSR} or \textit{JSR}, set \textit{value}_{\text{table}}[\beta] = \beta. If there is any return value, set \textit{value}_{\text{table}}[r_{o}] = r_{o}.

This reflects the conservative assumption that executing the call might change the memory state for any object in the \textit{M}-

0 FRAME 0 => r2 r3 [\text{fps}_0 \text{ fps}_4 \text{ fps}_b \text{ fps}_4 \text{ fps}_0 \text{ fps}_b \text{ fps}_0 \text{ fps}_0]
10 i2i r3 => r4 # get result
10 i2i r4 => r5 # get p
10 i2i r5 => r6
10 iLD r6 => r7 [\text{fps}_0 \text{ fps}_b]
10 uADDI 4 r6 => r8
10 iLD r8 => r9 [\text{fps}_4 \text{ fps}_b]
10 iADD r7 r9 => r10
10 iST r4 r10 [\text{fps}_0 \text{ fps}_b] [\text{fps}_4 \text{ fps}_b] [\text{fps}_b] [\text{fps}_0]
11 uADDI 4 r3 => r11
11 i2i r2 => r12
11 iLD r12 => r13 [\text{fps}_0 \text{ fps}_b]
11 uADDI 4 r2 => r14
11 iLD r14 => r15 [\text{fps}_4 \text{ fps}_b]
11 iSUB r13 r15 => r16
11 iST r11 r16 [\text{fps}_0 \text{ fps}_b] [\text{fps}_4 \text{ fps}_b] [\text{fps}_b] [\text{fps}_0]
12 RTN

execution of the operations. The \textit{M-def} set for FRAME can be considered as the initial states of all possible memory objects accessed by the function. Because JSR and \textit{xST} operations can change the states of memory objects in \textit{M-def}, the \textit{sca} construction conservatively assumes those operations do change the states of those memory objects. Thus, it treats \textit{M-def} as definition set. Figure 5 shows the \textit{sca} form of \textit{Compute}.

Of course, in some contexts, those operations will not modify the states of those memory objects, \textit{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 \textit{sca} graph.
ValueNumberOneBasicBlock(BLOCK)

{ }
for all φ nodes (including memory φ) in BLOCK 

{ value number φ 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 JSR/l/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

def list. It also assigns a new value number to \( r_o \), to reflect the return value.

\( xLD \ r_o \Rightarrow \ r_v \ M-use[\alpha ...] \)

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 \( \text{Norm} \) that converts the load/store opcode into a canonical form.

Similarly, a \( \text{Value} \) 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. \( \text{Value}(M\text{-use}) \) of \( xLD \) and \( xST \) represents the memory state before the execution of load and store operations.

For a load \( \text{op}_1 \), “\( xLD \ r_o \Rightarrow \ r_v \ M-use_{\alpha} \)”, the algorithm constructs the tuple \( \langle \text{Norm}(xLD), \text{Value}(r_o) \rangle \), and uses it as a key into the operation table. If there is a matching entry \( \text{op}_2 \), the algorithm tests for the equality of \( \text{Value}(M\text{-use}_{\alpha}) \) and \( \text{Value}(M\text{-use}_{\beta}) \). If they are equal, then \( \text{op}_1 \) is redundant, and a new entry \( \langle \text{Norm}(xLD), \text{Value}(r_v), \text{Value}(M\text{-use}_{\alpha}) \rangle \) is added to the operation table for \( \text{op}_1 \), with result value as \( r_v \).

Intuitively, value numbering for load checks whether the canonical load opcode, generated by \( \text{Norm} \), 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.

\( xST \ r_o \ r_v \ M-use[\alpha ...] \ M-def[\alpha ...] \)

For a store \( \text{op}_1 \), “\( xST \ r_a \ r_v \ M-use_{1 \ M-def} \)”, the algorithm constructs the tuple \( \langle \text{Norm}(xST), \text{Value}(r_a) \rangle \) and uses it as a lookup key into the operation table. For any matching \( \text{op}_2 \) with result value as \( \text{Value}(r_v) \), the algorithm checks whether \( \text{Value}(M\text{-use}_1) = \text{Value}(M\text{-use}_2) \). If they are equal, then \( \text{op}_1 \) is redundant and the algorithm must set

\[
\text{value}_table[\alpha] = \text{value}_table[\alpha'],
\]

for \( \alpha \in M\text{-def}_1 \), where \( \alpha' \in M\text{-use}_1 \) and \( \alpha \) and \( \alpha' \) are SSA names for the same memory objects.

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

The value numbering for store checks the canonical store opcode, generated by \( \text{Norm} \), 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 \( \text{Value}(M\text{-use}) \) for both \( xLD \) and \( xST \). Second, the memory state immediately after execution of a memory operation is represented by \( \text{Value}(M\text{-use}) \) for an \( xLD \) or \( \text{Value}(M\text{-def}) \) 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, \( \alpha M \) (0 ≤ \( \alpha < 1 \)) 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

\(^2\)If this becomes problematic, the implementor can use multi-set discrimination to guarantee \( O(1) \) access [9].
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 (opa_0 and opb_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.

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

\[ AVLOC_i = \text{computed locally as in Section 5.1} \]

\[ AVIN_i = \begin{cases} \emptyset & \text{if } i \text{ is the entry block;} \\ \bigcap_{j \in \text{pred}(i)} AVOUT_j & \text{otherwise}. \end{cases} \]

\[ AVOUT_i = AVIN_i \cup AVLOC_i \]
memory operations, for each memory operation in the operation table, we assign it a unique ID number (not overlapped 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 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 ∈ AVLOC_i; 2) for a memory operation in block i with ID m, m ∈ AVLOC_i, furthermore, if it is an ALD with result value r_v, then r_v ∈ 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 are detected and removed by scanning the operations in block i in execution order as follows: (1) if scalar operation s computes r_v ∈ AVIN_i, s is redundant and removed, otherwise, add r_v to AVIN_i; and (2) if memory operation with ID m ∈ 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 the same ID with the first and second load, and thus are removed by CSE. Figure 11 shows the code for Compute after CSE.

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 sccv with CSE, referred to as v_0. We tested the two passes on 10 benchmarks, 6 from Media-bench [22], 4 from SPEC2000 CPU integer benchmarks [33]. The benchmarks and the statistics of the compiled ILOC in-termediate 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 [35], 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 I−M shows that


<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Difference in Op Counts</th>
<th>I − M</th>
<th>Memory Ops</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>I (v₀ − v₁)</td>
<td>V (v₀ − v₁)</td>
<td>M (v₀ − v₁)</td>
</tr>
<tr>
<td>adpcm</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>g721</td>
<td>1,495,524</td>
<td>1,495,524</td>
<td>+72,975</td>
</tr>
<tr>
<td>gzip</td>
<td>784,774</td>
<td>711,799</td>
<td>-72,975</td>
</tr>
<tr>
<td>epic</td>
<td>70,470</td>
<td>70,493</td>
<td>-23</td>
</tr>
<tr>
<td>pegwit</td>
<td>137,380</td>
<td>137,380</td>
<td>0</td>
</tr>
<tr>
<td>mpeg2dec</td>
<td>886,486</td>
<td>816,423</td>
<td>+70,063</td>
</tr>
</tbody>
</table>

Table 3: Dynamic Instruction Count Difference

six of the benchmarks (gsm, mpeg2dec, 181.mcf, 164.gzip, 256.bzип2, 175.vpr) have a larger difference in total instruction count than in memory operation count. This suggests that, for those applications, detection of memory redundancy 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 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 would like to explore the memory performance issues using sophisticated microarchitecture simulators and study the effects how the compile-time memory operation reduction optimization will interact with different configurations of processor resources and cache system.

The data in Table 2 and 3 also show that most redundant memory operations discovered by our technique are loads.

From mpeg2dec.c:

```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:

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

Figure 12: mpeg2dec and 164.gzip Source Code

In fact, across all benchmarks, only mpeg2dec and 164.gzip had redundant stores (one and five, respectively). Tracing 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 cases³, 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 v₀ and v₁. 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.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Time (second)</th>
<th>SSA v₀ v₁</th>
<th>Value Numbering v₀ v₁</th>
<th>CSE v₀ v₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>g721</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
</tr>
<tr>
<td>gzip</td>
<td>0.00</td>
<td>0.00</td>
<td>0.04</td>
<td>0.08</td>
</tr>
<tr>
<td>epic</td>
<td>0.01</td>
<td>0.04</td>
<td>0.08</td>
<td>0.06</td>
</tr>
<tr>
<td>pegwit</td>
<td>0.00</td>
<td>0.03</td>
<td>0.05</td>
<td>0.02</td>
</tr>
<tr>
<td>mpeg2dec</td>
<td>0.01</td>
<td>0.02</td>
<td>0.06</td>
<td>0.08</td>
</tr>
<tr>
<td>181.mcf</td>
<td>0.00</td>
<td>0.01</td>
<td>0.03</td>
<td>0.02</td>
</tr>
<tr>
<td>164.gzip</td>
<td>0.01</td>
<td>0.04</td>
<td>0.37</td>
<td>0.11</td>
</tr>
<tr>
<td>256.bzип2</td>
<td>0.00</td>
<td>0.04</td>
<td>0.11</td>
<td>0.06</td>
</tr>
<tr>
<td>175.vpr</td>
<td>0.02</td>
<td>0.13</td>
<td>0.54</td>
<td>0.24</td>
</tr>
</tbody>
</table>

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

³The 164.gzip case is more difficult, the same macro expansion is used in several other places which do not cause redundancy.
for real applications, we also collected the set size for memory related operations. The results are shown in Table 6. 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, 30]. 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 [29, 21, 10, 28, 36, 15, 34]. Although our algorithm needs the alias analysis result to build M-list, however, the relation between the effectiveness of our algorithm and the accuracy of alias analysis is subtle: it is more profitable to have empty intersection between M-lists (so more memory states might be preserved as unchanged), rather than to keep the M-list size smaller which has been used as dominant criterion to measure accuracy of alias analysis. Our experience suggests flow-insensitive, context-insensitive pointer analysis works well with the new algorithm. As C structure is widely used in applications, it is important to distinguish field accesses to aggregate data structures. We use ad-hoc method in the Andersen algorithm to distinguish C struct fields. The aggregate decomposing algorithm described in [27] would be useful if it can be targeted for the C-like pointer-intensive languages.

### 7. CONCLUSION AND FUTURE WORK

<table>
<thead>
<tr>
<th>Average</th>
<th>SCC iter</th>
<th>scalar lookup</th>
<th>scalar update</th>
<th>load lookup</th>
<th>load update</th>
<th>store lookup</th>
<th>store update</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm</td>
<td>1.33</td>
<td>1.42</td>
<td>3.25</td>
<td>3.45</td>
<td>1.10</td>
<td>1.10</td>
<td>1.75</td>
</tr>
<tr>
<td>g721</td>
<td>1.07</td>
<td>1.08</td>
<td>1.26</td>
<td>1.26</td>
<td>0.55</td>
<td>0.55</td>
<td>1.34</td>
</tr>
<tr>
<td>gsm</td>
<td>1.14</td>
<td>1.14</td>
<td>1.91</td>
<td>1.91</td>
<td>0.84</td>
<td>0.84</td>
<td>1.88</td>
</tr>
<tr>
<td>epic</td>
<td>1.20</td>
<td>1.22</td>
<td>2.32</td>
<td>2.51</td>
<td>0.94</td>
<td>0.95</td>
<td>2.11</td>
</tr>
<tr>
<td>pegwit</td>
<td>1.11</td>
<td>1.12</td>
<td>1.38</td>
<td>1.40</td>
<td>0.57</td>
<td>0.57</td>
<td>1.38</td>
</tr>
<tr>
<td>mpeg2dec</td>
<td>1.10</td>
<td>1.10</td>
<td>1.79</td>
<td>1.86</td>
<td>0.74</td>
<td>0.72</td>
<td>1.67</td>
</tr>
<tr>
<td>181.mcf</td>
<td>1.26</td>
<td>1.27</td>
<td>2.47</td>
<td>2.53</td>
<td>0.80</td>
<td>0.77</td>
<td>2.70</td>
</tr>
<tr>
<td>164.gzip</td>
<td>1.18</td>
<td>1.19</td>
<td>2.06</td>
<td>2.19</td>
<td>0.72</td>
<td>0.70</td>
<td>2.17</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>1.25</td>
<td>1.26</td>
<td>2.70</td>
<td>2.82</td>
<td>0.72</td>
<td>0.70</td>
<td>2.82</td>
</tr>
<tr>
<td>175.vpr</td>
<td>1.16</td>
<td>1.17</td>
<td>2.28</td>
<td>2.43</td>
<td>0.69</td>
<td>0.60</td>
<td>2.42</td>
</tr>
</tbody>
</table>

### Table 5: SCC Iterations, Table Lookup and Update

<table>
<thead>
<tr>
<th>Avg Size</th>
<th>ref mod Func M-def call M-use</th>
<th>call M-def</th>
<th>load M-use</th>
<th>store M-def</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm</td>
<td>6.00</td>
<td>3.50</td>
<td>6.50</td>
<td>1.40</td>
</tr>
<tr>
<td>g721</td>
<td>5.07</td>
<td>3.79</td>
<td>5.86</td>
<td>2.74</td>
</tr>
<tr>
<td>gsm</td>
<td>14.14</td>
<td>6.25</td>
<td>15.49</td>
<td>5.19</td>
</tr>
<tr>
<td>epic</td>
<td>9.42</td>
<td>2.72</td>
<td>9.64</td>
<td>2.97</td>
</tr>
<tr>
<td>pegwit</td>
<td>26.07</td>
<td>16.25</td>
<td>27.15</td>
<td>14.31</td>
</tr>
<tr>
<td>mpeg2dec</td>
<td>38.19</td>
<td>22.50</td>
<td>45.28</td>
<td>16.46</td>
</tr>
<tr>
<td>181.mcf</td>
<td>12.71</td>
<td>6.96</td>
<td>13.46</td>
<td>4.22</td>
</tr>
<tr>
<td>164.gzip</td>
<td>30.82</td>
<td>16.60</td>
<td>32.70</td>
<td>15.72</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>24.81</td>
<td>8.31</td>
<td>25.71</td>
<td>10.89</td>
</tr>
<tr>
<td>175.vpr</td>
<td>30.40</td>
<td>8.78</td>
<td>31.55</td>
<td>9.32</td>
</tr>
</tbody>
</table>

### Table 6: M-list Set Size
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. We also thank the anonymous reviewers for their helpful suggestions. 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


