Reducing Memory Traffic with CRegs

Peter Dahl
dahl@ee.umn.edu

Matthew O'Keefe
okeefe@ee.umn.edu

Department of Electrical Engineering
University of Minnesota, Minneapolis, MN 55455

Abstract

Array and pointer references are often ambiguous in that compile time analysis cannot always determine if distinct references are to the same object. Ambiguously aliased objects are not allocated to registers by conventional compilers due to the cost of the loads and stores required to keep register copies consistent with memory and each other. There are several hardware and software strategies that can be used to solve the ambiguous alias problem; we have implemented one such scheme called CRegs in a compiler and instruction level simulator. We present a modification to Briggs' Optimistic Coloring Algorithm that allows us to allocate local and parameter arrays to CRegs. The CRegs register file operation and instruction set modifications required to implement this scheme are discussed. Underlying hardware issues such as pipeline impact and chip area are briefly discussed. Several benchmarks are compared in terms of dynamic instructions executed for two CReg set sizes. The measured reduction in memory operations is significant, averaging 23% for the benchmarks shown.

Keywords: CRegs, ambiguous alias, register allocation, graph coloring, live range.

1 Introduction

An ambiguous alias occurs when two names may or may not refer to the same memory location. In the example shown in Figure 1, if the relationship between `i` and `j` is not known at compile time or varies (denoted `i ? j`), the references may depend on each other and the load in line 3 is required. If the relationship between `i` and `j` can be determined (`i` $=$ `j` or `i` $\neq$ `j`) then a load can be eliminated because `a[i]` can be allocated to a register. In general, ambiguously aliased objects are not allocated to the register file because of the loads and stores required to maintain correctness.

CRegs is a hardware scheme that performs a run time consistency check of effective addresses, in this manner aliases are detected and values are propagated to ambiguously aliased objects in other CRegs [10]. With CReg hardware maintaining addresses in the register file, the compiler can allocate pointer values and array elements to CRegs safely without fear of using stale data, reaping the benefits of fast local memory for these objects. A performance gain is expected due to the associated reduction in memory bandwidth.

A CReg is a register that has an additional address field used to perform associative matches with other CRegs and that is capable of updating other CRegs under certain circumstances. The scope of the associativity is limited by the CReg set size; the underlying hardware must be capable of writing the value fields of the matching CRegs in the set simultaneously. Values in different sets are not updated even if their addresses match.

Alias sets are used to group live ranges for allocation to CReg sets. An alias set is composed of live ranges that are ambiguously aliased at some point in their live range [1]. It is assumed that names that are always aliased have been appropriately renamed and that objects that are never aliased are placed in different alias sets. Formal parameters are analyzed for aliasing and placed into alias sets using an algorithm developed by Cooper [8].

There are many solutions to the ambiguous alias problem [7, 11, 6] but we focus only on CReg hardware and compiler techniques in this paper. The remainder of the paper covers CReg instruction set modifications and hardware design (Section 2), CReg compilation techniques (Section 3), experimental results and analysis (Section 4), and a summary (Section 5).

2 CReg Hardware

In this section we explain basic CReg operation and propose modifications to a typical load/store instruction set architecture to support CRegs. In addition, we show how it can be integrated into an existing superscalar microprocessor design.

All CReg address matching and value update occurs in loads and stores; these instructions are the primary means for maintaining the mapping between memory and registers. Non-memory instructions are modified to clear the address field of their destination CReg. An address is only bound to the value once the value is loaded or stored with a CReg load or store. The range

This work was supported by contract No. DAAL02-89-C-0038 between the Army Research Office and the University of Minnesota for the Army High Performance Computing Research Center, Office of Naval Research Grant No. N00014-93-1-0426, and National Science Foundation Grant NSF CCR-9110261.

	code	i ? j	i $\neq$ j	i = j
1)	... = a[i] + ...	load r2←a[i]	load r2←a[i]	load r2←a[i]
2)	a[j] = ...	store a[j]←r3	store a[j]←r3	store a[j]←r3
3)	... = a[i]	load r4←a[i]	use r2	use r3

Figure 1: Ambiguous Alias Example

of address matching is limited to a CReg set which is responsible for propagating values between CRegs whose addresses match. A *maximal CReg set* is where the addresses associate over the entire register file.

2.1 Original Instruction Set

The original load and store instructions remain unchanged except that they now clear the address field of the CReg they are loading or storing. Hence, a compiler that is unaware of CRegs can safely use existing loads and stores effectively treating CRegs as registers. Our compiler uses these instructions when address matching and value update is not required, *e.g.* when saving and restoring registers at subroutine boundaries.

2.2 CReg Matching Instructions

A load and match address instruction is added to the instruction set to load aliased objects into CRegs. It calculates the effective address[1] as a normal load does, but performs an associative search with other CReg addresses within the set. If a match is found, the memory operation is "squashed" (*i.e.*, not performed) and a copy from the matching CReg is made. The load still takes time in the processor pipeline but no memory operation is initiated.

A store and match address instruction calculates the effective address and associatively searches for other CRegs with the same address. If it finds some, it copies the value to be stored to the other CRegs. Unlike the load, the store always performs a memory operation, keeping objects in memory updated with the current value. Multiple writes can occur within the register file; potentially all the CRegs in the set may need to be written. In practice there are rarely more that two CRegs updated within the set.

There are two variants on the store and match address instruction. The first updates matching CRegs but does not write the address of the CReg being stored. Our compiler uses this when the left hand side of an assignment statement is aliased and spilled therefore the store address should not be written to the source register. The second form of the store updates the address of the CReg being stored as well as the values of matching CRegs. This overwrites any address that was there, effectively allocating the object to the CReg. Our compiler uses this after a DEF to start the live range of an aliased object allocated to a CReg.

2.3 Address Clear Instructions

Care must be taken when live ranges are live across subroutine boundaries. We identify two cases. In the first case, CRegs containing values from one subroutine's stack frame could cause inadvertent squashed loads in following subroutines if their stack frames occupy the same memory space (at different times). The second case involves an object that may not be involved in the subroutine call, but the CReg it is allocated to (or another in the same set) is used by the subroutine. We force a break in the live range for these objects and clear their address fields. This isolates the caller and callee by making the ambiguously aliased objects reload at the first USE following the subroutine call. Alternatively, both the address and value field could be saved and restored at the subroutine boundary.

2.4 Context Switching

More aggressive register isolation is required for context switches, exceptions, and interrupts. The CReg register file must contain the same contents after the event as before. This is accomplished with special stores and loads that move both the value and address field of the CReg to and from memory.

2.5 Example

Figures 2 and 3 illustrate CReg operation. Values in CRegs or memory are referred to by name (`r5`, `a[i]`); addresses are referred to with an "&" prefix (`&r5`, `&a[i]`). Loads and stores involving CReg updates and memory operations are depicted as a box surrounding the operations they perform. For this example, `i` and `j` are not known at compile time, the array references are ambiguously aliased. Assume that the live ranges for `a[i]` and `a[k]` are allocated and that the live range for `a[j]` is spilled. Furthermore, assume the live range for `a[k]` starts with the USE in line 4.

Line 2 of the example shows a DEF of an array element; the store that follows this DEF allocates `a[i]` to `r4` by writing the address of `a[i]` into `&r4`. It also copies the value of `r4` to any CRegs in the same set with matching addresses and writes the value to memory. There is no load required for `a[i]` in line 4. Note that if code in line 3 changed the value of `a[i]` through another name, `r4` would be updated via CRegs address matching. The load of `a[k]` in line 4 is necessary to start the live range for `a[k]` and may or may not result in a memory operation. If the effective address of `a[k]` matches the address field of a CReg in the same set, the value will be copied from that CReg instead of being loaded from memory. Finally, the store to `a[j]` writes the new value in `r6` to memory and to other CRegs with the same address but does not update `&r6`. The address field of `r6` remains null (it was set to null by the add instruction in line 4) because `a[j]` is spilled.

[1]Virtual addresses are used if virtual memory is supported.

```
1)  input i,j
2)  a[i] = 7.0


3)  ...
4)  a[j] = a[i] + a[k]
```

Figure 2: Simple Code Example

```
1)  r1 ← i,  r2 ← j
2)  r4 ← 7.0; &r4 ← null
    | copy r4 to CRegs matching &a[i]           CReg |
    | store r4 to &a[i]                         store |
    | &r4 ← &a[i]                                    |
3)  ...
4)  (no load of a[i], it's allocated to r4)
    | if (&a[k] = &r4)  r5 ← r4 (squash)        CReg |
    | else              load r5 from &a[k]      load |
    r6 ← r4 + r5; &r6 ← null
    | copy r6 to CRegs matching &a[j]           CReg |
    | store r6 to &a[j]                         store |
```

Figure 3: Underlying Operations

2.6 CRegs Implementation

An earlier study showed that CRegs could be integrated into a simple RISC processor pipeline using cache-like circuitry with a small chip area increase and no clock cycle time impact [12]. For superscalar processors [9] CReg stores cannot be dual-issued with an integer ALU operation since the store may write a source register for the integer ALU instruction. Additional forwarding logic is needed, a load followed by a store of the same CReg needs the loaded value and address to avoid stalling. A CReg implementation will require additional register file ports. In addition, each CReg requires an address field approximately doubling the size of the register file. Given increasing chip sizes and the small area required for registers relative to caches, functional units, and the data path, the increased chip area may not be a problem.

3 Allocating Objects to CRegs

Graph coloring register allocators use nodes to represent live ranges of values and arcs between nodes to specify that the two live ranges cannot be allocated to the same register. The nodes of the interference graph are colored representing an allocation of live ranges to specific registers.

3.1 Live Range Construction

A variety of objects are eligible for allocation to CRegs. The following simple rules describe our live range strategy. We first organize the local and formal parameter array references according to the name of the array and the index calculation involved. Any variables in the index calculation are encoded with their value number to annotate the reference uniquely. Live ranges are built for array elements in the same manner as scalars, treating each unique index for a given array as a different element.

Globals and pointers are not allocated and so are "pre-spilled" which means that for each USE/DEF there is spill code responsible for loading/storing the value. Live ranges for scalar locals and formal parameters are created as described by Briggs [3].

3.2 Alias Analysis

This section describes how we group objects into alias sets in preparation for allocating them to CReg sets. If ambiguously aliased objects were not grouped in alias sets, they might be allocated across multiple CReg sets and stale data could occur in some CRegs.

The alias analysis we use for local variables is minimally simple. If an object A is ambiguously aliased with object B at some point in the program, we put both A and B in the same alias set. If another object C is aliased with B at a later point, it is also placed in the alias set. The set implies that A and C are aliased; this may or may not be true but is conservatively assumed to be true. Obviously there is room for improvement in the construction of these sets; note however that A, B, and C all must be allocated to the same CReg set and so enhanced alias analysis may not improve the allocation process.

An alias set is represented in the interference graph as a circularly linked list of live range nodes. For local arrays, the alias set consists of all the live ranges for the elements of that array[2]. Scalars have a null pointer for their alias set.

Two basic operations on alias sets find the range of CRegs that are allowed during color selection. If an alias set has a member that has been colored to a CReg set, the remaining alias set members are limited to colors within the CReg set. If no members of the alias set have colors, a CReg set is chosen in a round-robin manner. The first operation is finding the color of any member of an alias set given an uncolored member of the set (`find_alias_color(node)`). If no members of the set have been colored yet, a color from a CReg set is chosen in a round-robin manner. If the node is not a member of an alias set, an illegal color is returned. The second basic operation uses the resulting color and finds the first and last CReg in the corresponding CReg set (`find_first_last(color)`). If the color given is not a valid color, it returns the first and last CReg of the entire register file. If the target architecture supports a

[2]This is conservative; if two local array live ranges do not interfere, they can be placed in different alias sets.

maximal CReg set, `find_first_last()` always returns the entire register file as the range.

3.3 Modifications to Briggs' Optimistic Coloring Algorithm

Briggs' Optimistic Algorithm delays spill decisions until it knows there are no colors available in the color selection phase [4]. We control the selection of colors such that nodes in an alias set are limited to CRegs in one CReg set. The color selection stage scans through the range of colors, checking them against the colors of the node's neighbors. This loop is originally limited to the available colors, we further limit it based on the CReg set to which the node should be colored. For scalar objects or allocation to a maximal CRegs set, the limits remain the entire register file. If the node cannot be colored within the range of colors specified, spill code is inserted and the algorithm iterates.

Chaitin-style allocators (including Briggs') have heuristics that limit the insertion of spill code. For example, if all the USEs of the live range are "close"[3] to the DEF, the live range will not have spill code inserted. We override this in the case of DEFs of ambiguously aliased objects where a store always follows to keep memory updated.

During color selection, we originally found that several alias sets frequently had one member allocated to a single CReg set. The remaining members of the alias sets often were not allocated because the CReg set was full. This is why `find_alias_color()` returns colors from CReg sets in a round-robin manner if the alias set is not fixed to a CReg set yet. This allows alias sets a better chance to be completely allocated to a CReg set. Multiple alias sets can still be allocated to one CReg set if they will fit.

Our augmentation to Briggs' Algorithm is independent of the cost function making it compatible with other improvements [2].

3.4 Dependence and Pointer Analysis

CReg hardware is complementary to compile time techniques to reduce ambiguous aliasing such as dependence and pointer analysis. These techniques are powerful but cannot always succeed in disambiguating all relevant references. When static techniques fail, CReg hardware provides a mechanism to detect aliasing at run time, retaining the advantages of register storage for aliased objects. For this study, no dependence or pointer analysis was implemented. For a maximal CReg set, pointer values can be allocated without analysis. If the CRegs are divided into sets, analysis is required to group references into alias sets.

4 Experimental Results

Our test suite consists of several floating point SPEC '89 and '92 benchmarks, Livermore Loops, and a hydrodynamics code[4].

The optimizing compiler, known as `ccc`, was developed at the University of Minnesota. The simulator can support a conventional or CReg register file for the Alpha architecture [9]. It is an instruction level simulator capable of giving dynamic instructions counts, but does not simulate the processor pipeline or cache and so cannot give actual cycle counts or an indication if a CReg reference would have been a cache hit or miss.

To calculate the number of loads reduced by CRegs, two compilations and simulations are performed. The first compilation is targeted toward a conventional register file. When this is simulated on a conventional architecture, a certain number of instructions are executed and counted in categories (loads resulting in a memory operation/load instructions/stores/all instructions) as a baseline for comparison. The second compile allocates objects to CRegs and then is simulated with a CReg register file. The dynamic instruction counts are compared to get the reduction percentage.

4.1 Analysis

We observed the following for the benchmarks shown in Figure 4:

- The dynamic load reduction ranged from 0 to 75%.
- The average reduction in memory operations with CRegs was 23%.
- Small CReg sets cause only a slight increase in memory operations compared to a maximal CReg set.
- The dynamic instruction count reduction varied from -0.6% to 13.8%.
- Register pressure increased.

Dynamic load reduction is due to squashed loads and direct removal of spill loads. For squashed loads, the memory operation does not occur (*i.e.*, it is not counted as a load), but the instruction still counts in the total instructions executed. Fewer loads occur in the CReg code because more objects are allocated.

The data shows that having a maximal CReg set gives only a slightly better reduction in loads. This means that the register allocator is not overly constrained by allocating alias sets to CReg sets.

The reduction in total instructions executed is slightly positive meaning that the reduction in loads for objects allocated to CRegs more than offsets the increased subroutine isolation code[5].

The maximum number of writes and matches was measured by the simulator for all test cases. It never exceeded two for the benchmarks shown but theoretically it could be as large as the set size. We attribute this to the low number of simultaneously live ambiguously aliased objects.

With a CRegs implementation, execution time is reduced for the following reasons:

- There are generally fewer instructions executed.

[3]Two mentions of a live range are "close" if no other live range goes dead between them [5].

[4]The **ppm** code was obtained from Dr. Paul Woodward at the University of Minnesota.

[5]More CRegs than registers are used within procedures making the number of loads and stores performed by the callee greater.

	Maximal CReg Set				Set Size 4			
Benchmark	mem loads	loads	stores	instrs.	mem loads	loads	stores	instrs.
nasa7/vpenta	36.62	36.60	-0.01	8.52	33.03	32.96	-0.01	7.65
tomcatv	21.06	20.80	0.00	7.11	19.24	17.32	0.00	6.09
matrix300	0.11	0.11	0.00	-0.03	0.11	0.11	0.00	-0.03
doduc/debico	74.44	66.60	-0.12	13.75	75.33	65.35	-0.12	13.49
doduc/dcoera	9.00	0.00	0.00	-0.61	4.38	-1.46	-1.79	-1.51
doduc/dyeh	4.21	0.00	0.00	0.00	1.05	0.00	0.00	0.00
doduc/yeh	11.22	0.00	-0.03	0.00	4.08	0.00	0.00	0.00
Livermore Loops	8.39	4.62	-0.02	1.45	7.92	4.16	-0.07	1.23
ppm	21.30	14.04	-1.42	4.63	19.78	13.74	-1.57	4.47

Figure 4: Percent Reduction Summary for Register File Size of 32

- There are fewer memory operations executed.
- Overall data access latencies are reduced since more references are directly to registers.

5 Summary and Future Work

A register file composed of CRegs is one solution to the ambiguous alias problem. We have shown an instruction level implementation and explained the operation of the memory operations involved. Our CReg compiler works in conjunction with the CReg hardware to produce code that contains fewer load instructions and squashes some memory references dynamically. Live ranges are constructed for local and formal parameter array elements similar to those for scalars. These live ranges are grouped into alias sets using interprocedural alias analysis and local alias information. A simple modification to Briggs' Optimistic Coloring Algorithm allows an interference graph augmented with alias information to be allocated to CRegs. The change involves limiting which colors are available to a node during the color selection phase of the algorithm. A round-robin scheme helps distribute alias sets among the CReg sets and averts deadlock situations which spill alias set members.

In the future, we will be implementing better alias analysis; more aggressive algorithms will keep the alias set size small so that pointer values can be allocated without causing significant increases in spill code. The complexity of pointer analysis may limit this solution to an architecture with a maximal CReg set. One way to do this would be to add alias edges to the interference graph; an alias edge is present between two nodes if the nodes must be allocated to CRegs in the same CReg set.

Acknowledgments

We would like to thank John Mejia for help with our Alpha simulator, Terence Parr for assistance with the PCCTS compiler construction tool, Peter Bergner for his help with the graph coloring algorithms, and James Smith from Cray Research for feedback on an early draft of this paper.

References

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

[2] D. Bernstein, D. Goldin, M. Golumbic, H. Krawczyk, Y. Mansour, I. Nahshon, and R. Pinter. Spill code minimization techniques for optimizing compilers. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 258–263. ACM, June 21–23 1989. Portland, OR.

[3] P. Briggs. *Register Allocation via Graph Coloring.* PhD thesis, Rice University, April 1992.

[4] P. Briggs, K. Cooper, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation.* ACM, June 21–23 1989. Portland, OR.

[5] G. Chaitin. Register allocation and spilling via graph coloring. *SIGPLAN Notices*, 17(6):98–105, June 1982. *Proceedings of the 1982 SIGPLAN Conference on Compiler Construction.*

[6] C. Chi and H. Dietz. Unified management of registers and cache using liveness and cache bypass. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 344–355. ACM, June 1989. Portland, OR.

[7] T. Chiueh. An integrated memory management scheme for dynamic alias resolution. In *Proceedings of Supercomputing '91*, pages 682–691, November 1991. Albuquerque, NM.

[8] K. Cooper. Analyzing aliases of reference formal parameters. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 281–290. ACM, January 1985.

[9] DEC. *Alpha Architecture Handbook.* Digital Equipment Corporation, 1992.

[10] H. Dietz and C. Chi. CRegs: A new kind of memory for referencing arrays and pointers. In *Proceedings of Supercomputing '88*, pages 360–367, November 1988. Orlando, FL.

[11] B. Heggy and M. Soffa. Architectural support for register allocation in the presence of aliasing. *Proceedings of Supercomputing '90*, pages 730–739, 1990.

[12] S. Nowakowski and M. O'Keefe. A CRegs implementation study based on the MIPS-X RISC microprocessor. In *Proceedings of the 1992 International Conference on Computer Design*, pages 558–563, October 1992. Boston, MA.