CRegs: A New Kind of Memory for Referencing Arrays and Pointers

Henry Dietz
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907
hankd@ee.ecn.purdue.edu
(317) 494 3357

Chi-Hung Chi
Phillips Laboratories
345 Scarborough Road
Briarcliff Manor, NY 10510

Often, pointer and subscripted array references touch memory locations for which there are several possible aliases, hence these references cannot be made from registers. Although conventional caches can increase performance somewhat, they do not provide many of the benefits of registers, and do not permit the compiler to perform many optimizations associated with register references. The CReg (pronounced "C-Reg") mechanism combines the hardware structures of cache and registers to create a new kind of memory structure, which can be used either as processor registers or as a replacement for conventional cache memory. By permitting aliased names to be grouped together, CRegs resolve ambiguous alias problems in hardware, resulting in more efficient execution than even the combination of conventional registers and cache can provide.

This paper discusses both the conceptual CReg hardware structure and the compiler analysis and optimization techniques to manage that structure.

Keywords: cache, register, register-allocation, data-aliasing, compiler-optimization.

1. Concepts

To explain how and why CRegs[1] improve performance, it is first necessary to analyze how and why conventional registers and cache can fail to improve performance (as well as how and why they can improve performance).

1.1. Registers

A register array is a relatively small, fast, local memory residing in an address space separated from that of main memory. The structure of a register memory cell is given in Figure 1.

Figure 1: Register Memory Cell Structure

By placing a value in a register, one can reap at least four benefits:

[1] The fast access time of values in registers reduces latency.

[2] A reference to a register typically does not interfere with references along the path(s) to main memory, thereby effectively increasing usable bandwidth to main memory.

[3] Typically, the predictability of register references aids in compile-time optimization of code and simplifies hardware. Optimizations are aided in that reference times can be known at compile time; hardware is simplified in that register references in most machines cannot cause pipeline bubbles.

[4] Because register names are typically shorter than memory addresses, referencing values in registers actually decreases the required instruction-fetch bandwidth — even though registers typically cannot hold instructions.

The "catch" is that, for most programs, many values cannot benefit from being kept in registers. Although it is true that sometimes a value cannot be kept in a register because the hardware provided too few registers, even given an infinite number of registers, a large fraction of the values computed within any program should not be kept in registers. To understand why some values should not be kept in registers, one must understand a little bit of compiler flow analysis.[2]

Suppose a particular segment of a program refers to two names, one called α and the other called β. If one of α and β is a pointer, or one is a call-by-address argument to this routine and the other is a variable which was accessible in the caller's scope, or both are elements of the same array (such as `a[i]` and `a[j]`), etc., then it is possible that even though α and β look like different names, they refer to the same object. In other words, changing the value of one might change the value of the other, i.e., α and β might be aliases for the same object.

If compile-time analysis can prove that α and β cannot be aliases for the same object, then α and β can each be assigned to a register and each can be kept there indefinitely. Instead, if the compiler can prove that α and β are always aliases for the same object, then α and β are assigned to share a single regis-

[1] A patent application for the CReg invention is currently in progress.

[2] The description given here of the ambiguous alias problem is a gross oversimplification intended only to give an intuitive introduction to the problem. This issue is currently one of the richest research areas within compiler technology; more detailed discussions of this problem appear in [All83], [Bur84], [BuC86], [All86], [Ste86], and [Die87].

ter, and again the object can be kept in a register indefinitely. However, if the compiler isn't sure if α and β refer to the same object, or if α and β only sometimes refer to the same object, we say that α and β are ambiguously aliased to each other.

At this point, it is useful to point out that compile-time analysis techniques for determining if α and β are aliases for each other are, at best, complex to implement and easy to confuse. Confusion results in the "safe" assumption that α and β are ambiguously aliased to each other. In addition, in many cases it is ***theoretically impossible*** for the compiler to determine whether α and β are aliased, in which case the compiler must again assume that they are ambiguously aliased. A good example of such a case is determining whether `a[i]` and `a[j]` are aliased in code like Listing 1.

```
readln(i, j);
b := a[i] + a[j];
```

Listing 1: Compile-Time Unresolvable Aliasing Problem

If the compiler's best "guess" is that α and β are ambiguously aliased, then placing either value in a register will require "flushing" that register whenever either α or β is stored into. This "flushing" is usually needed so often that the cost of referencing α and β from registers is actually greater than the cost of referencing them from main memory, hence, placing α and β in registers would degrade, rather than improve, performance.

1.2. Cache

A cache is a "small" memory holding rapidly accessible copies of values addressed associatively by main memory addresses. The conceptual structure of a cache memory cell is shown in Figure 2.

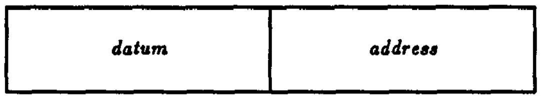

Figure 2: Cache Memory Cell Structure

Of the four benefits listed for placing a value in a register, however, in general, caches only insure [1]: a reduction in access latency. Benefit [2], which is based on the lack of interference between register and main memory data paths, does not hold for a traditional cache, except in that other processors in a multiprocessor system typically would not see interference from cache references on a particular processor's cache.

The predictable reference time for a register reference, benefit [3], is not echoed in cache reference because of the concept of a cache miss. Some would argue that, given a large enough cache, the probability of having a cache miss can be made arbitrarily low; however, we believe this misses the point. One reason we disagree with the very-large cache argument is that the access speed of a memory is related to the size of its address space (e.g., if one can fit the cache on the processor chip, it will probably function much faster than if it is referenced across several chips). Another reason is that the cost of implementing an arbitrarily large cache is also arbitrarily large — it isn't very cost effecitive. In any case, unless the cache is as large as the entire virtual address space of the machine, one will occasionally suffer a cache miss, and this implies that extra hardware/software effort must be made to cope with this situation.

Benefit [4] is based on the reduction of required instruction-fetch bandwidth due to use of short names in referencing values. This cannot be applied to cache because the register correspondence between short names (register numbers) and long names (memory addresses) must be explicitly established by register `Load` and `Store` instructions, whereas the mapping in a cache is unknown to the software. In other words, a compiler cannot tell which cache line of a conventional cache will hold a copy of a particular value it is referencing — hence, it cannot use the cache line number to address the value. The desired value might not even be in the cache, either because it has not yet been placed there or because placing some other entry in cache "bumped" the desired entry out of cache.

The above discussion might lead one to conclude that registers are far better than cache, so why use cache? The answer is simply that while ambiguously aliased values cannot be profitably placed in registers, they can be placed in cache.

1.3. CRegs

Unlike registers, cache-registers (CRegs) may be used to buffer values which may have ambiguous aliases; unlike cache, CRegs provide the ability to use short names for variables instead of addresses (thereby reducing instruction-fetch bandwidth requirements) and also provide for conceptually duplicate entries (many-to-one mapping of short names into addresses). CRegs provide all four advantages of registers; but CRegs provide these advantages for all values, ambiguously aliased or not.

The conceptual structure of a CReg memory cell is a superset of both cache and register cell organizations, as illustrated in Figure 3.

name:
datum
address

Figure 3: CReg Memory Cell Structure

Each CReg is a register which holds both address and data fields. When a CReg is referenced (by CReg number — a short name), an associative search is made to find neighboring CRegs which have the same value in their address field. Any CRegs found by this association *are* aliases for the CReg directly named, and the CReg hardware simply maintains coherence of these entries. This associative function is implemented by hardware very similar to that implementing the associativity of a cache, however, unlike a cache, CRegs avoid making memory references on an aliased `Load` operation by using *duplicate entries in the CReg array.* (The precise operation of CRegs is described in greater detail in the example of the following section.)

Although the STM (generalized Short Term Memory cell) [Sit79] employs a memory cell structure similar to that of a CReg, STMs did not support CReg-like associative function. Likewise, the "rack" desciribed in [StS85], suggests a similar cell structure, but is not associative. The register-addressed stack cache mechanisms of various processors could be argued to provide benefits similar to that of CRegs; however, they do so only for items in the top few stack cells. Since these items are a subset of the items which could have been kept in registers, stack caches also fail to provide a solution to the aliased-item reference problem.

The implementation, and hence the circuit complexity, of a CReg array is virtually identical to that of a similar-size cache; the CReg array is slightly simpler because the hardware is explicitly told where to make each entry (what CReg number), whereas conventional caches use hardware-implemented policies, such as LRU, to decide which line to replace. However, CRegs can be managed as efficiently as registers, hence, unlike cache, very small CReg arrays are quite useful. For example, simply replacing the registers of a conventional processor with CRegs (and, incidentally, not even changing the instruction set) is typically both feasible and effective to the extent that the number of memory references made by the processor can often be halved.

2. An Example

An example clearly illustrates the advantage of being able to use CRegs for all value references, whether ambiguously aliased or not. Consider the FORTRAN code of Listing 2.

```
C
C       A call-by-address subroutine
C
        SUBROUTINE NASTY(I, J, K)
 10     I = J * K
 20     K = J + K
        RETURN
        END
```

Listing 2: FORTRAN Sample Code

The subroutine `NASTY` operates on three arguments, `I`, `J`, and `K`, which are all passed using call by address. Since FORTRAN permits `I`, `J`, and `K` to reference the same cell of main memory, the values of `J` and `K` cannot be blindly placed in conventional registers in the code for line 10 and simply reused in the code for line 20 — to do so would produce incorrect results if `I` refers to the same main memory cell as either `J` or `K`. To place these variables in conventional registers, the compiler would need to know precisely which of the arguments referenced the same cells — the classic ambiguous alias problem discussed earlier. Hence, unless the compiler is permitted to generate multiple encodings of `NASTY` (one for each possible set of variable aliases), none of `I`, `J`, and `K` can be kept in registers.

However, all of these variables can be placed in CRegs. In fact, if this is done, the references in line 20 will *always* be served within CRegs — main memory will not be accessed. Table 1 shows all possible combinations of aliases for `I`, `J`, and `K`, and, for each alias set, where the values of `J` and `K` are to be found.

Table 1: CReg Place of Reference for `J` and `K`

Aliases	Where `J` is	Where `K` is
`I`, `J`, `K` disjoint	CReg for `J`	CReg for `K`
`I` is `J`	Assoc. `I`,`J`	CReg for `K`
`I` is `K`	CReg for `J`	Assoc. `I`,`K`
`J` is `K`	Assoc. `J`,`K`	Assoc. `J`,`K`
`I` is `J` is `K`	Assoc. `I`,`J`,`K`	Assoc. `I`,`J`,`K`

As indicated earlier, the read requests in line 20 for `J` and `K` are always satisfied within the CReg array. The entries in the table which say "CReg for" are simple CRegs acting as ordinary registers (with no associative access); the entries which say "Assoc." are satisfied by the associative memory function of the CRegs. (Notice that the associative function depends on the existence of *duplicate entries* in the CReg array — which would not be permitted in a conventional cache.) Further, since all three variables can be placed in CRegs, the references in line 20 would simply use the CReg names, rather than memory addresses, because a CReg name *implies* a main memory address. This fact also permits *entire instructions to disappear:* `Store` instructions can be implicit, using a "dirty bit" and a "lazy store" mechanism.

It can be argued that the references in line 20 *might* be satisfied in a more conventional cache, thereby avoiding a main memory reference in the same way that CRegs avoid the reference; however, only CRegs can guarantee that this occurs under all circumstances (as we detailed in the previous section). Even accepting that a conventional cache *might* avert the main memory references as CRegs do, CRegs also reduce the *instruction fetch* bandwidth requirements by permiting short CReg names to be used for `I`, `J`, and `K` rather than long main memory addresses.

The next subsection briefly describes a simple RISC register-based processor design and its adaptation to use CRegs instead of registers. In the subsection following that, the above example is encoded and the execution traced for this RISC machine.

2.1. A RISC Processor using CRegs

There are many possible implementation techniques for CRegs. In this section, we discuss a simple modification of a RISC processor design which replaces a register array with a CReg array[3]. The point here is not to propose a RISC architecture, but rather to show the difference between register-based and CReg-based versions of the same architecture.

The example architecture is a 16-register RISC machine whose register Rf (register 15) is also the PC (program counter). Replacing the registers with CRegs, we obtain the programmer's model given in Figure 4.

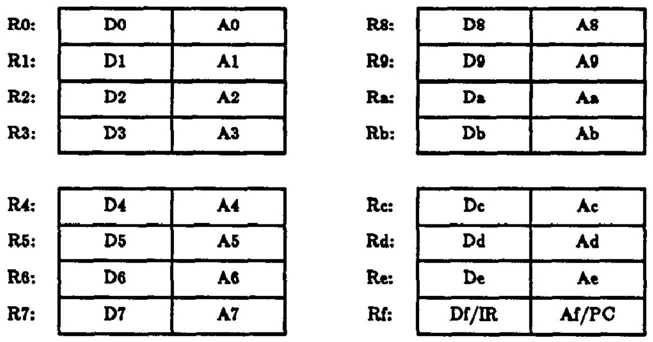

Figure 4: CReg RISC Programmer's Model

The first difference one notices is that each CReg has two fields, the data and address fields. Quite naturally, the PC is the address field of Rf, which implies that the data field of Rf is the IR (instruction register); although it is not a major point of this paper, prefetch of instructions beyond a branch is really just another flavor of ambiguous alias reference, and it too is handled using CReg associativity.

The programmer's model also reflects fragmentation of the 16 CRegs into four sets of four CRegs each. Since it may be difficult to construct cache-like hardware which is more than about 4-way associative [Smi78], the 16-CReg array is broken into associative groups of four CRegs each[4]. For example, if CReg R1 holds D1=5, A1=601 and an instruction attempts to load the contents of memory location 601 into R2, an associative load will occur (without a main memory reference) and R2 will hold D2=5, A2=601. If one tried to load the contents of memory location 601 into R4 instead of R2, since R1 and R4 are in different associative sets, a memory reference would be needed to load R4 and the value loaded might not match the value in R1. Obviously, the "trick" is to always place names which are ambiguously aliased together in the same set . . . compiler techniques to accomplish this are discussed in sections 3.2 and 3.3. If CReg hardware can be built without this segmentation, the only results are that compilation becomes easier and the CRegs become more effective.

The instruction sets of the RISC and CReg RISC machines are identical, hence there are no changes to describe.

2.2. Code for the Example

Using the original (conventional register-based) RISC design, the instruction sequence for the lines 10 and 20 of the FORTRAN program in listing 1 would be as given in listing 2. (To simplify the example, no special handling of delayed loads or other compiler optimizations are assumed.)

```
Ld      R0,@argJ        ;R0 <- addr(J)
Ld      R1,@R0          ;R1 <- J
Ld      R2,@argK        ;R2 <- addr(K)
Ld      R3,@R2          ;R3 <- K
Mul     R4,R1,R3        ;R4 <- R1 * R3
Ld      R5,@argI        ;R5 <- addr(I)
St      @R5,R4          ;mem(R5) <- R4
Ld      R1,@R0          ;R1 <- J
Ld      R3,@R2          ;R3 <- K
Add     R4,R1,R3        ;R4 <- R1 + R3
St      @R2,R4          ;mem(R2) <- R4
```

Listing 3: Register RISC Code

Listing 3 should be compared with the CReg RISC instruction sequence given in Listing 4. (As Listing 3, Listing 4 does not reflect the application of any compiler optimizations.)

[3] This is not implying, for example, that CRegs could not be implemented in any other way — in fact, CRegs can even be implemented by taking a conventional off-chip cache and simply mapping some portion of the global memory address space into literal cache addresses (cache line addresses rather than associative cache line address labels). A conventional processor could then use short offsets from an index register to reference these CRegs by name.

[4] Unlike caches which are grouped into sets by address-space, the CRegs in this machine are grouped into sets by CReg name-space. Alternatively, CReg sets could be time-space partitioned. For example, associative access to CRegs {R0, R1, R2, R3} may complete in two cycles, for CRegs {R4, R5, R6, R7} in three cycles, etc. Provided the number of cycles (time-multiplexed associative sets) is not greater than the main memory reference time, a compiler can consider this cost function in allocating CRegs so as to maximize probable benefit.

```
Ld      R3,@argJ      ;D3 <- addr(J), A3 <- argJ
Ld      R0,@R3        ;D0 <- J, A0 <- addr(J)
Ld      R4,@argK      ;D4 <- addr(K), A4 <- argK
Ld      R1,@R4        ;D1 <- K, A1 <- addr(K)
Mul     R2,R0,R1      ;D2 <- R0 * R1
Ld      R5,@argI      ;D5 <- addr(I), A5 <- argI
St      @R5,R2        ;A2 <- D5, mem(A2) <- D2
Add     R1,R0,R1      ;D1 <- D0 + D1, mem(A1) <- D1
```

Listing 4: CReg RISC Code

The first seven instructions of both register and CReg code sequences appear to serve the same purpose; however, they do not imply the same memory references. Further, in the rest of the code, a single instruction in the CReg version replaces four instructions in the conventional register code.

To make these differences more visible, we will consider the situation which occurs using CRegs when `I` is an alias of `K` (i.e., addr(`I`) is addr(`K`)). Figure 5 shows the progression of CReg contents as the CReg RISC code of Listing 4 is executed. In each of the diagrams of Figures 5a-5h, CReg associative sets which are not involved in the actions caused by this code sequence are not shown. The additional field on each CReg in Figures 5a-5h is the "dirty" bit — a `1` indicates a dirty value, which may be lazily stored back to main memory, reseting the dirty flag. Notice that dirty bits are also set/reset associatively.

R0:	—	—	-	R4:	—	—	-
R1:	—	—	-	R5:	—	—	-
R2:	—	—	-	R6:	—	—	-
R3:	addr(J)	argJ	0	R7:	—	—	-

Figure 5a: CReg Contents after Executing `Ld R3,@argJ`

R0:	J	addr(J)	0	R4:	—	—	-
R1:	—	—	-	R5:	—	—	-
R2:	—	—	-	R6:	—	—	-
R3:	addr(J)	argJ	0	R7:	—	—	-

Figure 5b: CReg Contents after Executing `Ld R0,@R3`

R0:	J	addr(J)	0	R4:	addr(K)	argK	0
R1:	—	—	-	R5:	—	—	-
R2:	—	—	-	R6:	—	—	-
R3:	addr(J)	argJ	0	R7:	—	—	-

Figure 5c: CReg Contents after Executing `Ld R4,@argK`

R0:	J	addr(J)	0	R4:	addr(K)	argK	0
R1:	K	addr(K)	0	R5:	—	—	-
R2:	—	—	-	R6:	—	—	-
R3:	addr(J)	argJ	0	R7:	—	—	-

Figure 5d: CReg Contents after Executing `Ld R1,@R4`

R0:	J	addr(J)	0	R4:	addr(K)	argK	0
R1:	K	addr(K)	0	R5:	—	—	-
R2:	J*K	—	-	R6:	—	—	-
R3:	addr(J)	argJ	0	R7:	—	—	-

Figure 5e: CReg Contents after Executing `Mul R2,R0,R1`

R0:	J	addr(J)	0	R4:	addr(K)	argK	0
R1:	K	addr(K)	0	R5:	addr(K)	argI	0
R2:	J*K	—	-	R6:	—	—	-
R3:	addr(J)	argJ	0	R7:	—	—	-

Figure 5f: CReg Contents after Executing `Ld R5,@argI`

R0:	J	addr(J)	0	R4:	addr(K)	argK	0
R1:	J*K	addr(K)	1	R5:	addr(K)	argI	0
R2:	J*K	addr(K)	1	R6:	—	—	-
R3:	addr(J)	argJ	0	R7:	—	—	-

Figure 5g: CReg Contents after Executing `St @R5,R2`

R0:	J	addr(J)	0	R4:	addr(K)	argK	0
R1:	J+(J*K)	addr(K)	1	R5:	addr(K)	argI	0
R2:	J+(J*K)	addr(K)	1	R6:	—	—	-
R3:	addr(J)	argJ	0	R7:	—	—	-

Figure 5h: CReg Contents after Executing `Add R1,R0,R1`

It is interesting to note that the lazy store mechanism, in the case traced above, would very likely avoid performing the store to main memory which was implied in Figure 5g (since the store implied in 5h makes 5g a "dead store" because `I` and `K` are actually the same object).

3. Compiler Technology

The basic compiler technology needed to make good use of CRegs is very similar to that needed to perform register allocation, however, there are a few complications. The first complication is that names must be grouped according to which other names they are ambiguously aliased with, henceforth called an **alias set**; this is discussed in the next subsection. The subsection after that discusses the problem of reasonably packing alias sets into CReg associative sets and of allocating registers given such a packing.

Throughout this section, our intent is *not* to provide the best possible CReg management, but rather to demonstrate that reasonably good CReg management is not particularly difficult to implement.

3.1. Alias Sets

As discussed above, the fundamental flaw in static analysis of conventional-language programs is that it is not possible to statically determine, for all variables, which ones *are* aliased to which others at each point in the program. The CReg mechanism does not aid in solving this problem; however, it changes the problem into one which can be solved. The alias problem for CRegs is simply finding which items *can be* aliased to each other. We call this problem the construction of **alias sets**.

The basic tools with which alias sets are constructed are the familiar algorithms of compiler flow analysis (including dependence analysis). These tools have been particularly well-honed in pursuit of efficient automatic parallelization. The presentation here is intended merely to provide a brief overview to the analysis involved in creating alias sets.

3.1.1. Names

The first issue to resolve in grouping names into alias sets is the basic question of what constitutes a name. Each variable could be considered a name, however, this is not the most useful definition. The difficulty is rooted in the fact that a variable α may be an alias for a variable β within one region of a program, while α may be an alias for δ in another section of the code. In such a case, considering α to be a name used for grouping into alias sets, it would be necessary either to make the alias set containing α be $\{\alpha,\beta,\delta\}$ or to make the alias set for α be $\{\alpha,\beta\}$ in one region of code and $\{\alpha,\delta\}$ in another. Ideally, names should be chosen so that each name is a member of an alias set whose contents are independent of position in the program, yet where no names are included unnecessarily.

The solution to this naming problem is simply to incorporate control and data flow information in the names: however, the mapping from user variable names into these **aliased-object names** is surprisingly complex. For example, if the user has declared `i` to be an `int` variable and `p` to be an `int *` which is initially set to point at `i` (e.g., `p=(&i);`), then references to both `i` and `*p` use the same aliased-object name: user names are mapped many-to-one into aliased-object names. This means that if the compiler can detect that two user names are unambiguously aliased to each other, these two user names will share a single aliased-object name. The rule is more precisely expressed as:

Definition 1: User-Name Merging
The user-created names α and β can be merged into a single aliased-object name within some region of code *iff* the values associated with the names α and β are known to be the same throughout that region of code.

which also implies that explicitly made copies of values can all share a single aliased-object name (i.e., the compiler can perform copy propagation).

On the other hand, in a code sequence like `i=j; . . . i=k;`, the user name `i` will be mapped into multiple aliased-object names, one for each different value stored into `i`. This rule is best expressed in terms of **D-U chains and U-D chains** [AhS86]:

Definition 2: User-Name Spliting
Let U be the set of uses of (loads from) the user name α. For each use $u_i \in U$, let the U-D chain rooted at u_i be called d_i. If, for any i and j, $d_i \cap d_j \neq \emptyset$, then let $d_i = d_i \cup d_j$ and delete d_j. When no more such merger/deletions can be performed, each of the remaining sets (d_i) can be represented by a separate aliased-object name.

Notice that values which do not have programmer-assigned names, such as intermediate results within an expression, also may be assigned aliased-object names by the above rules.

3.1.2. Formation of Alias Sets

Given the above definitions, it is relatively easy for a compiler to generate a set of names appropriate for grouping into alias sets; but what is an alias set? There are actually several compile-time distinguishable types of aliases:

[1] A name α is a **true alias** of the name β if α is known to always be associated with the same value that is associated with β. (Notice that, if this is so, the two names may be merged by Definition 1 given above.)

[2] A name α is an **intersection alias** of the name β if α and β are known to share some elements of their values, however, perhaps not all elements. For example, if `a` is a `struct` containing members called `b` and `c`, then `a` and `a.b` are intersection aliases. Intersection aliases occur most often in code referring to arrays.

[3] A name α is a **sometimes alias** of the name β if α is known to be a true or intersection alias for β under some circumstances at runtime, however, α is not an alias for β under other circumstances. For example, references to `a[i]` and `a[5]` are sometimes aliases if `i` could be equal to `5`.

[4] A name α is an **ambiguous alias** for β if α is an intersection alias or sometimes alias for β, or if the compiler is unable to determine the relationship between α and β.

[5] A name α is **mutually exclusive** of β if α and β are *not* related by any of the above alias types. If, for all β, α is mutually exclusive of β, then α is **unambiguous**.

For the purpose of CReg assignment, an alias set is a set of names grouped by "closure" of the ambiguous alias relation. In other words, given a name n, the alias set for n consists of n $\cup$ (all names which are ambiguous aliases of n) $\cup$ (all names which are ambiguous aliases of those names) $\cup \ldots$ Notice that these alias sets have several useful properties:

Uniqueness

If α is a name in alias set S, then α is in no other alias set. This assignment is also independent of the region of code in which α is referenced.

Completeness

If α is a name, it is a member of some alias set; if α is mutually exclusive of all other names, then the alias set which contains α is a singleton set containing α.

Relationship to CReg Assignment

The number of elements in an alias set is the maximum possible number of CRegs which that set could use beneficially (i.e., it is the upper bound on CRegs needed, achieved only if all names are simultaneously live [AhS86]). In fact, the elements of an alias set *are* the items which are assigned to CRegs.

3.2. CReg Allocation

Given that the source program has been analyzed and that the collection of alias sets is known, the next step is to assign values to CRegs and to generate code reflecting that assignment. Since CRegs closely resemble registers, it is not surprizing that the allocation schemes for CRegs closely resemble those for register allocation, except for the need to operate on alias sets. If, for example, all alias sets obtained from a program are singleton sets, CReg allocation *is precisely* register allocation.

Due to limitations of hardware circuit complexity, the (simultaneous) assocativity of a CReg array is constrained to be a small number: typically four (just like the associativity of cache). However, it is quite reasonable to have an array much larger than just four CRegs — breaking the CReg array into associative sets as described in the example CReg RISC processor of section 2.1. Consequently, the first and the most important rule of CReg allocation is to put all elements from each particular alias set into the same CReg associative set. At first, this sounds overly constrained, since an alias set containing more than four elements cannot possibly "fit" into a four-element CReg associative set, however, experience with compiler automatic parallelization technology [Ste86] has shown that the average number of *simultaneously active* ("live") names within an alias set is very rarely more than three[5].

Another key issue in CReg allocation is CReg **spilling**. An item is spilled from a register if a register is needed for some other item, yet no registers are empty. Here, the problem is that if a single name from an alias set is to be referenced from a CReg associative set other than that which contains the other elements of the alias set, *all* elements of the alias set must first be flushed from the CRegs. This makes spilling of alias sets highly undesirable: spills defeat the benefits gained from CReg hardware automatically maintaining consistency across multiple names in an alias set.

As an illustration of the above guidelines, the following subsection presents an easily implementable CReg allocation scheme based on usage counts [Fre74]. Although good enough to demonstrate the advantages of CRegs, this CReg allocation scheme is far from optimal (measuring optimality in terms of minimizing the total execution time for all references). An optimal CReg allocation scheme based on machine state transition modeling [ChD87] is currently under investigation within the Compiler-oriented Architecture Research group at Purdue University (CARP).

3.3. Example CReg Allocation Scheme

The main modification to conventional register allocation based on usage counts [Fre74] is that CReg allocation is effectively heirarchical: one first allocates alias sets to CReg associative sets and then allocates individual CRegs within each set.

To describe the algorithm, it is first necessary to define some measures which will be used to define allocation priorities in the algorithm. The **usage count** of an alias set is defined as the total number of references to names within the alias set which appear in the program segment under consideration. For best results, each reference which appears in the program text should be weighted according to its expected frequency of execution relative to other references. Expected execution frequencies can be estimated by examining the program control flow [Die87]; for example, references in the `then` clause of an `if` statement have about one half the execution frequency of those which precede the branch. References inside a loop are weighted by the expected number of times the loop will iterate. The **cost-savings estimate** of an alias set is therefore:

```
Cost = ((Usage Count) *
         ((Cost of Memory Reference) -
          (Cost of CReg Reference))) -
       ((Size of Alias Set) *
        (Cost of CReg Load))
```

[5] *This number makes sense in that code like a[i]=a[j]*a[k] only uses three names — the number three seems to be a side-effect of the dominance of binary operations.*

The **size** of an alias set is equal to the maximum number of simultaneously live values in that set within its live range period (as suggested earlier, the upper bound on this number is the number of names in the alias set, and this is an acceptable approximation). The **live range** of an alias set is set of references during which any name within the alias set is live.

Given these definitions, the CReg allocation scheme is:

[1] Compute the alias sets for references within the program. This was described in section 3.1. Size and cost estimates are associated with each alias set, as described above.

[2] Assign alias sets to CReg sets. The assignment procedes to allocate alias sets in the following way:

[2a] The unallocated alias set with the largest cost-savings estimate is allocated first.

[2b] If multiple alias sets have the same cost-savings estimate, the one with the largest size is allocated first.

[2c] If there is more than one CReg set which can fit the current alias set, the CReg set which is the "best fit" is chosen.

[2d] If there is no CReg set which can fit the current alias set, then the alias set is placed in the CReg set for which the estimated spill cost to make space for the alias set is the lowest. The estimated spill cost for a CReg set to fit a new alias set is computed by summing the costs to remove alias sets (in reverse order of allocation) from that CReg set until the number of free CRegs in the set is $\geq$ the size of the alias set. If an alias set is larger than the number of CRegs in a CReg set, then, for the purpose of allocating CReg sets, the estimated spill cost for a CReg set to fit that alias set is computed by summing the costs to remove all alias sets from that CReg set.

[3] Perform the CReg allocation within each CReg set independently as:

[3a] Keep all CRegs in use as much as possible.

[3b] When CRegs must be spilled, the alias set(s) with the minimum total cost-savings estimate should be chosen.

4. Conclusions

In this paper, the CReg, a new architectural concept, is introduced. The first section argued that registers and cache are inherently unable to provide good efficiency in accessing aliased objects because neither structure embodies the concept of an aliased object in such a way as to allow a compiler to manage aliased references. The second section presented a detailed discussion of the operation of CReg hardware in managing aliased references; an outline of the new compilation technology associated with managing CRegs — in particular, operating on alias sets — is given in section 3.

At the writing of this paper, the performance of CRegs has been examined directly for only a few small examples. Performance for these examples has been very encouraging. Table 2 gives a *static* comparison of using registers only, registers plus an arbitrarily large cache, or CRegs only, for the RISC machine mentioned earlier. The FORTRAN benchmark is that given in Listing 2, considering only the code for the assignment statements; the C benchmark is a typical encoding of quicksort in C, using pointers to mark the start and end of the subarray to be sorted.

Table 2: Static Comparison of CRegs with Registers and Cache

Benchmark	Registers Only	Registers + Cache	CRegs Only
FORTRAN assignments			
Total Instruction Words	14	14	11
Total Memory References (Min)	23	20	12
Total Memory References (Max)	23	23	17
C quicksort			
Total Instruction Words	94	94	50
Total Memory References (Min)	157	124	72
Total Memory References (Max)	157	157	80

Ongoing work at Purdue University includes the construction of a simulator and a compiler so that *dynamic* results can be obtained for a much larger set of benchmarks and so that CReg design tradeoffs can be investigated.

References

[AhS86] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison Wesley, Reading, Massachusetts, 1986.

[All83] J. R. Allen, *Dependence Analysis for Subscripted Variables and its Application to Program Transformations*, Rice University, Ph.D. Thesis, April 1983.

[All86] F. Allen, "The Parallel Translator Project," NASA / ICASE Parallel Languages and Environments Workshop, November 1986.

[BuC86] M. Burke, R. Cytron, "Interprocedural Dependence Analysis and Parallelization," SIGPLAN Symposium on Compiler Construction, 1986, pages 162-175.

[Bur84] M. Burke, *An Interval Analysis Approach Toward Interprocedural Data Flow*, IBM, Yorktown Heights, New York, Research Report RC 10640 (#47724), July 1984.

[ChD87] C. H. Chi and H. Dietz, *Compiler-Driven Cache Policy (Known Reference String)*, Purdue University Technical Report TR-EE 87-21, June 1987.

[Die87] H. Dietz, *The Refined-Language Approach to Compiling for Parallel Supercomputers*, Polytechnic University, Ph.D. Thesis, June 1987.

[Fre74] R. A. Freurghouse, "Register Allocation Via Usage Counts," *Communications of the ACM*, Vol. 17, No. 11, November 1974, pp. 638-642.

[Sit79] Richard Sites, "How To Use 1000 Registers," Caltech conference on VLSI, January 1979.

[Smi78] A. J. Smith, "A Comparitive Study of Set Associative Memory Mapping Algorithms and Their Use for Cache and Main Memory," *IEEE Transactions on Software Engineering*, Vol. 4, No. 2, March 1978, pp. 121-130.

[Ste86] K. Stein, *Refined C Compiler Status Report*, Internal Report, Stevens Institute of Technology, 1986.

[StS85] G. L. Steele Jr. and G. J. Sussman, "The Dream of a Lifetime: A Lazy Variable Extent Mechanism," ACM SIGPLAN order number 552800, 1985, pp. 163-172.