Architectural Support for Register Allocation in the Presence of Aliasing*

Ben Heggy
Mary Lou Soffa
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260

Abstract

High performance computer architectures use registers to provide high speed access to data operands, to provide short names for operands, and to reduce memory traffic for accesses to these operands. The possibility of aliasing in a program segment reduces the quality of code that a compiler can produce by necessitating that memory and register copies of variables that have been allocated to registers be kept consistent. A hardware support mechanism is presented that permits all classes of data objects, including dynamically allocated objects and array elements, to be held in registers without consideration of possible aliases and without requiring that the generated code maintain consistency between register and memory copies of variables. Use of this approach permits programs to benefit from the speed advantages and reduced memory traffic associated with register storage, obviates the need to collect aliasing information for use in register allocation, and reduces instruction traffic by eliminating code used solely to maintain register-memory consistency. The support hardware can be implemented using known hardware technology and without increasing the cycle time of the processor.

Introduction

High performance computer architectures usually include fast general purpose registers designed to act as storage for frequently accessed data. Register storage can improve the performance of a computer system by providing substantial improvements in access time, by providing short names for operands, and by reducing the number of requests for memory access. These factors increase throughput for simple uniprocessors and can have even greater impact on pipelined processors and multiprocessors where memory access contention is a significant consideration. While the hardware advantages of registers are clear, their effectiveness can be severely reduced if compilers are not capable of using them effectively. The presence of aliases in a program negatively affects the ability of a compiler to perform effective register allocation. These negative effects are accentuated when interprocedural register allocation is performed, both due to the increased difficulty in obtaining accurate aliasing information and due to the number of aliases present.

An *alias* in a program arises whenever there are two or more distinct ways to refer to the same storage location. [1] When aliases are present, accesses to variables through names that appear to be independent may actually interfere because they refer to the same storage location. Aliases are introduced through the use of arrays, pointers, and call-by-reference parameters and thus occur in most programs in most programming languages. Aliasing introduces *back-door* access paths to variables that must be considered at compile-time in order to generate code that executes as expected. The possibility of back-door accesses forces a compiler to update the memory copies of variables held in registers to assure that accesses to the memory copies retrieve the current values of the variables. The possibility of back-door updates forces the compiler to re-load register copies from memory before each use due to the effects of aliased updates of the memory copy. The instructions generated to perform memory updates and re-load registers increase both the static and dynamic instruction count of the executable program and substantially increase memory traffic.

When a variable has been allocated to a register, it is only safe to remove the code that guards against back-door accesses when all code that uses (or modifies) the variable has been identified and changed to refer to the register. This identification implies a need to detect all possible names for a storage location; hence all possible aliases must be detected for each variable. Unfortunately, this presents two problems that force the compiler to retain the guard code. First, depending on the ways aliases may be introduced in the programming language, and the level of precision desired from aliasing information, the determination of all possible aliases can

* This work was supported in part by the National Science Foundation under grant CCR-8801104 to the University of Pittsburgh.

CH2916-5/90/0000/0730/$01.00 © IEEE

be an NP-complete problem or can produce a solution set of exponential size, and thus is not effectively solvable. Second, as different aliasing relationships may be established while the program runs, appropriate modification of code to expect a variable in a register may depend on the context from which the code was invoked. In either of these circumstances, it is not possible to eliminate all back-door accesses and therefore it is not possible to omit the guard code. The presence of aliases in a program thus reduces the quality of code that can be generated.

Several software techniques have been developed that conservatively estimate the aliases in a program. [4,5,6] These techniques reduce the negative effects of aliasing on the generated code by determining that some variables cannot be aliased, thus permitting them to be maintained in registers without the need to maintain consistency between their memory and register copies. These approaches usually produce an improvement in the code because they provide estimates of the actual aliasing patterns of a program instead of the worst-case assumption (that all type-compatible variables are aliases). However, because they produce only estimates of aliasing information, these techniques can be ineffective in worst-case scenarios. Even where these approaches are effective, they still require all variables that may exhibit aliasing to be maintained in memory.

Another approach to reducing the difficulties associated with register allocation in the presence of aliasing is to eliminate registers altogether. Some processor architectures, such as the Intel iAPX432, use only memory-to-memory instructions and do not provide explicit user-accessible registers.[7] A cache is used to improve operand access time. The elimination of registers does obviate the need to perform alias analysis for register allocation. However this approach suffers from decreased utilization of high-speed memory because of the online, block oriented nature of the cache management problem.

Another technique that has appeared in the literature suggests the use of hardware support in the form of "CRegs." [2] CRegs are banks of four registers which have an associative memory used to check for back-door accesses. When a request is issued to load a register using a memory address that is associated with one of the registers in the same CReg set, the request is satisfied from the register. When two or more CRegs have the same associated address, they are accessed in parallel in order to cause consistency among the aliased values. It is this parallel associative access technique that limits the CReg set size to 4. This technique still requires alias analysis in order to assure that possible aliased variables are always allocated to registers in the same set. The technique cannot directly handle any variable with greater than four possible aliases and must spill all elements of an alias set whenever it is necessary to access an element of the alias set in a different CReg associative set. The technique also requires that the register allocation scheme be cognizant of the organization of the machine registers into CReg associative sets.

The work presented in this paper approaches the problems associated with register allocation in the presence of aliasing by utilizing hardware support to eliminate the possibility of back-door accesses to the memory copies of variables. The use of this approach, which does not require alias analysis, permits a compiler to generate improved object code by eliminating the need to maintain consistency between register and memory copies of variables. Additionally, this approach reduces the overhead of runtime accesses to variables that are aliased by short-circuiting accesses using memory addresses and directing them to registers. When back-door accesses are removed from the runtime environment, it becomes permissible to freely allocate variables to registers without consideration of their possible aliasing relationships. The elimination of back-door accesses is accomplished by providing hardware to transparently note and recall the address associated with the value stored in each general purpose register and to monitor the addresses generated for memory references, redirecting references to the memory copy of variables to the register copy. We refer to this approach as *variable forwarding*.

This approach is preferable to estimating the aliases in a program and denying aliased variables promotion to registers because it allows even variables that have aliases to be allocated to registers safely. This approach maintains the problem of register allocation in a compile-time setting thus providing the possibility of better utilization of high-speed memory than can be achieved with cache management. A compiler targeted at a processor with variable forwarding hardware does not need to collect any aliasing information in order to perform register allocation. The elimination of instructions inserted to maintain consistency of register and memory copies of variables reduces both the static and dynamic instruction counts of a program and thus reduces memory traffic for both instructions and data. The added hardware maintains the table of addresses transparently, without added instructions in the code and does not extend the processor cycle time.

Before presenting our approach in greater detail, we consider the ways that aliases can be introduced into a program, the algorithms and heuristics available to estimate aliasing, and the effects of aliasing on register allocation and code generation.

Aliases

Aliases are introduced into a program in a variety of ways. When two or more pointer variables reference the same object in memory, the names for the fields of the referenced objects are aliases. Subscript expressions for references to the elements of an array may differ in structure and content yet may evaluate to the same value. The use of pass-by-reference parameters can introduce aliases in several ways. The same variable may be passed to a procedure at several positions in the argument list. If at least two of these positions are pass-by-reference parameters, then the corresponding formal parameters are names for the same variable. If a variable which is non-local to a procedure but which is visible to that procedure under the scoping rules of the language is passed to the procedure as a pass-by-reference parameter, the non-local name and parameter are aliases. If a variable that has aliases is passed as a pass-by-reference parameter, it carries its aliases along to each formal parameter to which it can be bound. While the C programming language does not have an explicit pass-by-reference parameter mechanism, the use of pointer parameters introduces similar difficulties.

Approximating Aliasing

The parameter aliasing problem can be solved in an imprecise way using a simple deterministic polynomial time algorithm. A precise solution to this problem can also be achieved using a deterministic algorithm, but because the size of the solution set is potentially exponential in the number of nodes in the program's call graph, this approach is generally too costly to be considered. The only way to make use of a precise solution to this problem would require a separate copy of each procedure, tailored for each possible aliasing circumstance, in order to accommodate varied aliasing circumstances on each distinct path through the call graph (infinite paths can be truncated safely). When only one copy of each procedure is used, an imprecise solution to the parameter aliasing problem is required in order to assure the correctness of the program. The merged information of an imprecise solution may yield results that are excessively pessimistic for a particular path and may include paths that are infeasible. Regardless of whether multiple copies of procedures are used with precise information or single copies are used with imprecise solutions, aliased variables must be maintained in memory, thus reducing the efficiency of the generated code.

Other types of aliasing can be estimated as well. The complexity of performing these analyses increases with the generality of the aliasing problems presented by the language, while the quality of the solution decreases. Determining aliasing in a program with arrays or pointers is an NP-complete problem. [5,6] Several techniques are known that consider the relationship between the values of program variables and subscript expressions in order to detect cases in which array accesses occur independently. These techniques, generally known as *array reference disambiguation*, are based on algebraic properties of the subscript expressions and can eliminate a class of expressions that occur when an array is being processed in a linear or systematic way. Processing orders based on random numbers, inputs, and contents of the array elements themselves cannot usually be disambiguated. Additionally, most disambiguation schemes can be led astray by the presence of infeasible paths.

The problem of determining whether or not two pointers may point to the same object is NP-complete. [5] There are some heuristics that may be applied to this problem but these generally require substantial analysis time, while still producing only rough estimates of the actual aliasing present in the program.

Effects of Aliasing on Generated Code

Without register forwarding hardware, the possibility that an aliased reference may occur to the memory copy of a variable while the variable is loaded into a register must be taken into account by the register allocation strategy. There are two basic approaches to this end that may be safely adopted: 1) deny potentially aliased variables promotion to registers and 2) maintain consistency between the register and memory copies of a variable. Both of these approaches forfeit some, if not all, of the advantages of storing variables in registers.

If variables that may have aliases are denied promotion to registers, most operations will require at least one memory access. If the compiler uses worst-case aliasing assumptions, the only variables that will be stored in registers are temporaries.

If registers are used, the memory and register copies of the variable must be kept consistent at all times, implying that the register copy of a variable must be reloaded from memory before each use that follows any instruction that could have written to memory. This assures that the value in the register reflects any updates that may have occurred using an aliased name, directly reaching the memory copy. Additionally, any modification of the value stored in a register must be immediately followed by a write to the memory copy of the variable so that any direct use of the memory copy through an aliased name will retrieve the most recent value of the variable.

Both of these approaches produce code that uses registers in a very inefficient manner and generates a

significant number of memory access requests. As an example of this situation, consider Figure 1. In Figure 1a, we present a small Pascal-like program fragment. In Figure 1b, we present one possible intermediate code sequence for the fragment. Figure 1c presents code generated using registers, with the insertion of `loads` and `stores` as required to maintain consistency of the register and memory copies of all variables. Figure 1d presents code that maintains variables in memory.

```
...
for i := 1 to 10
      T := T + i;
      A := A * i;
...
```

(a) Source Fragment

```
      ...
      i := 1
L1:   i cmp 10
      jpgt L2
      T := T + i
      A := A * i
      i := i + 1
      jp L1
L2:   ...
```

(b) Intermediate Code

```
      ...
      load r0,#1
      store r0,i
L1:   cmp r0,#10
      jpgt L2
      load r1,T
      add r0,r1
      store r1,T
      load r0,i
      load r2,A
      mul r0,r2
      store r2,A
      load r0,i
      inc r0
      store r0,i
      jp L1
L2:   ...
```

(c) Consistent Copies

```
      ...
      mov #1,i
L1:   cmp i,#10
      jpgt L2
      add i,T,T
      mul i,A,A
      add i,#1,i
      jp L1
L2:   ...
```

(d) Memory Only

Figure 1
Example Code Generation with Aliases

When code is generated placing `loads` and `stores` around variable uses and definitions, a total of 72 `load` and `store` instructions will be executed thus generating 72 memory operations (31 writes and 41 reads). In the case where variables are not promoted to registers, no `load` or `store` operations are executed, but implicit accesses in other instructions generate 81 memory operations (31 writes and 50 reads).

Variable Forwarding

The architectural features of variable forwarding assure that accesses to the memory copy of a variable retrieves the most recent value, even if that value is stored in a register. These features also assure that modification of the memory copy of a variable that is loaded into a register will affect the value stored in the register. These features permit the register allocator and code generator to make best-case assumptions about aliasing in the program without any danger of producing a program that does not operate correctly.

```
      ...
      load r0,i
      mov #1,r0
L1:   cmp r0,#10
      jpgt L2
      load r1,T
      add r0,r1
      store r1,T
      load r2,A
      mul r0,r2
      store r2,A
      inc r0
      store r0,i
      jp L1
L2:   ...
```

(a) Local Allocation

```
      ...
      load r0,i
      mov #1,r0
      load r1,T
      load r2,A
L1:   cmp r0,#10
      jpgt L2
      add r0,r1
      mul r0,r2
      inc r0
      jp L1:
L2:   store r0,i
      store r1,T
      store r2,A
      ...
```

(b) Global Allocation

Figure 2
Example Code Generation with Variable Forwarding

In Figure 2, we present the code that could be generated for a processor with variable forwarding hardware for the example program presented in Figure 1. Figure 2a presents code generated using only local register allocation. This code segment will execute 52 `load` and `store` operations resulting in 52 memory operations (31 writes and 21 reads). Figure 2b presents code generated using global register allocation, that will move `load` and `store` operations to the beginning and end of spans, and thus out of the loop body. This code will perform only 6 `load` and `store` operations resulting in only 6 memory accesses. Even from this simple example, it is clear that the ability to perform register allocation on the assumption that there are no aliases can provide for significant improvement of the generated code and significant reduction of both instruction and data memory traffic.

Forwarding Hardware

Figure 3 presents a schematic layout of the register forwarding hardware. The variable forwarding technique has two parts: memory forwarding, that detects back-door accesses to memory copies of variables and substitutes accesses to their register copies, and register forwarding that is necessary to permit potentially aliased variables to be allocated to different registers. The lower half of the figure shows the modifications to the memory access unit that perform memory forwarding. The upper half of the figure presents the modifications to the register select logic that perform register forwarding.

Effects on Memory Access

The usual action of the memory control unit is to buffer and synchronize accesses between the internal and external data, address, and control busses. Variable forwarding augments the memory control unit with additional hardware designed to monitor each memory access and detect occasions where memory forwarding is needed. This hardware compares the address being requested from memory with the address associated with each register through the use of an associative memory. This memory is shown in the lower left of Figure 3 and contains one entry for each general purpose register. As memory accesses occur, the addresses requested are applied to this memory. When a hit occurs, the result of the memory access in progress is ignored (or the access may be aborted) and a signal is sent to the register select logic to complete the request with the register causing the hit. It is important, both for the simplicity of the forwarding mechanism itself and for the simplicity of the associative memory design that each unique address appears only once in the associative memory, thus guaranteeing that only one entry will match per operation. This fact eliminates the need for shift registers and added logic needed to process multiple hits and permits the hit indication signals to be directly used in addressing the corresponding register.

The use of an associative memory to perform the parallel comparison of memory addresses with register tags permits this operation to complete in less time than the general memory subsystem requires to perform the actual memory operation requested. This means that if the two operations are initiated in parallel, the result of the associative memory access will be available in time for the memory control unit to substitute a register access whenever necessary. Because the comparison occurs in parallel with the memory request, the presence of the variable forwarding hardware does not increase the general memory access time. In fact, when a hit occurs and a register access is substituted for a memory access, the access time is reduced to a value somewhere between the normal memory access time and the register access time. When a request for a value from memory is satisfied by a forwarded register, a memory operation has been initiated that is not necessary for the instruction to complete. The results of such requests can be ignored, without concern for reversing side effects because all accesses to the memory locations in question will be forwarded to the appropriate register. Depending on the nature of the memory subsystem, it may be beneficial to abort memory requests that were initiated, but later found to be unnecessary. Figure 4 presents a detailed algorithm for the actions performed during a memory read or write operation.

Similar associative memory technology is currently used in most virtual memory systems to provide the translation of virtual addresses into the physical addresses to which they are mapped. The output of translation lookaside buffers (TLB's) is usually a set of words indicating the physical addresses associated with the matching entries. The associative memory used in register forwarding can be made less general than those used in virtual address translation as it does not need to accommodate the possibility of multiple hits, nor does it need to return a data value for the entry causing a hit, but merely the select signal of the matching entry. The size of an associative memory is usually limited by the amount of chip real estate available. This limitation should not pose a problem as each cell of the required associative memory is simpler than the cells used in translation lookaside buffers, and because the number of entries in the memory is the same as the number of general purpose registers, which is usually relatively small. The dedication of space to the variable forwarding hardware is justified when the improved runtime performance is considered. Additionally, recent results [8] have suggested that the use of register windowing hardware is unnecessary because a suitably designed compiler can achieve equally good results. The space formerly allocated to the register windowing hardware can be utilized by the register forwarding associative memories.

```
MemoryAccess:
        EA := effective address of operand
        initiate memory read/write request for EA
        /* these checks are performed in parallel */
        apply EA to register associative memory yielding R
        if R is not null then
                submit a read/write request for register R
                ignore or abort result of memory access
        else
                wait for memory access to complete
end MemoryAccess
```

Figure 4
Operation of a Memory Read or Write Operation

Effects on Register Access

The usual function of the register select logic is to operate as a simple address decoder. Variable forwarding augments the register select logic to perform register forwarding. This type of forwarding is needed when the same memory location has been `loaded` into several different registers (when aliased variables have been allocated to different registers). Any access to one of the registers associated with a single memory location must be forwarded to a single register in order to maintain consistency. We refer to the set of registers associated with a single memory location as a *forwarding group*. A single register from the forwarding group holds the actual

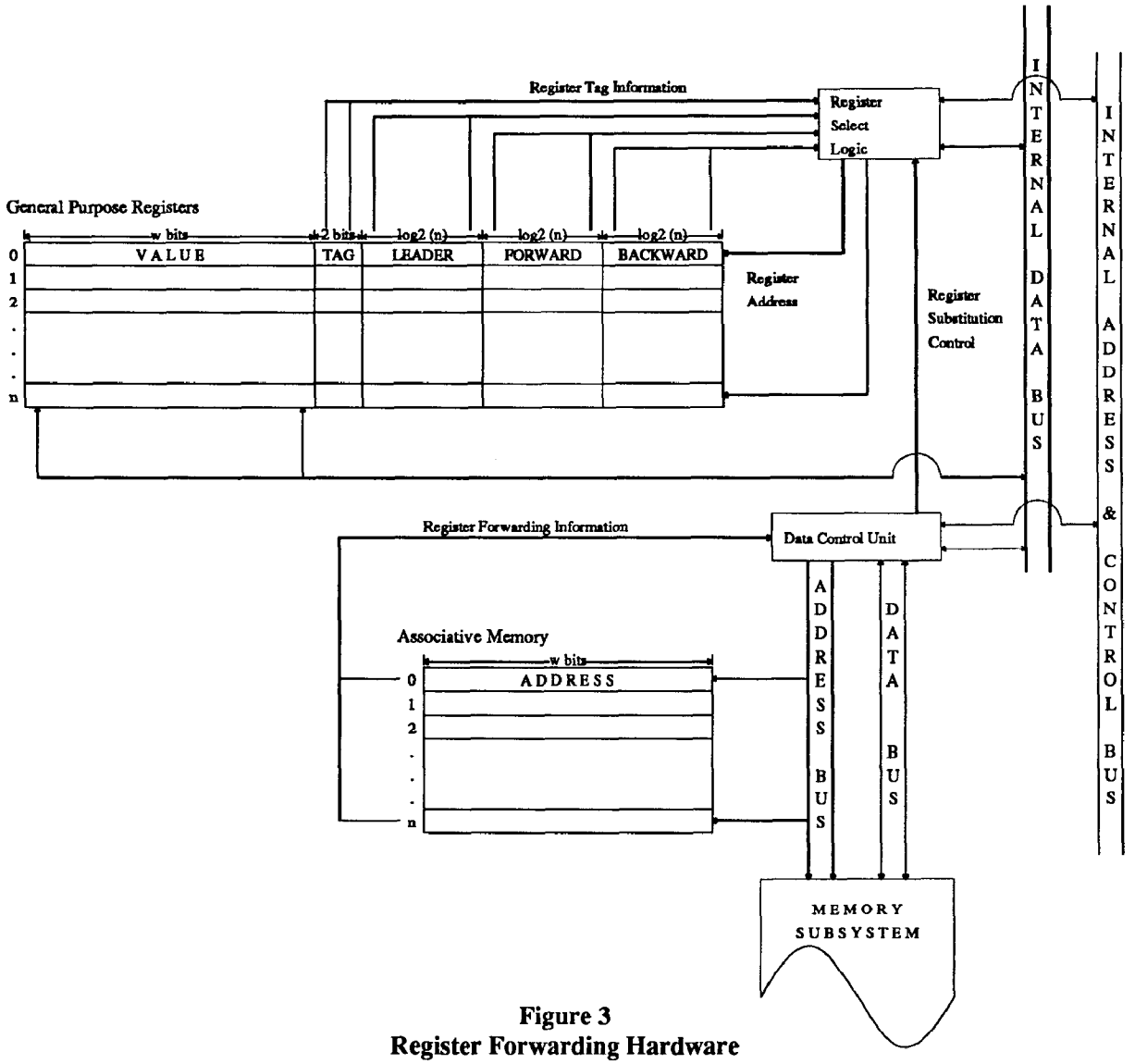

Figure 3
Register Forwarding Hardware

value of the variable. This register is referred to as the *leader*. The leader is the only register in the forwarding group that has its address field set to trap accesses to the memory copy of the variable and hence is the register that will be selected by any operation initiated by the memory forwarding system. All accesses to other registers in the forwarding group are also forwarded to the leader.

The knowledge of register forwarding information is represented by a tag and three link fields, forward, backward, and leader, that are stored with each register. Each time a register is selected, its tag field is checked to determine whether or not the register is part of a forwarding group. If the register is part of a group, but is not the leader, the leader link field is used to select the leader register of the group. This can result in a one cycle extension of the time required to access an operand in a register. However, this increased time is due to the need to retrieve the correct register and not due to any overhead associated with testing an individual register's tag field. Thus, the time required to access a register which is not being forwarded is not extended over the register access time of a processor without register forwarding. The forward and backward link fields are used to construct a doubly linked list of the registers in the forwarding group. This list is used to find other registers in the group in order to update the leader field when the current leader is `stored` and thus removed from the group.

The `load` instruction must be augmented to determine if the address of a variable that is being `loaded` into a register is already associated with another register, and when it is, to add the register to the existing forwarding group. This is achieved by adding a check of

the associative memory to the load operation before setting the tag field entry of the target register. Figure 5 presents a detailed algorithm of the actions performed in the register read and write operations.

```
RegisterAccess:
        RA := requested register
        Apply RA to register bank
        wait for Result
        if Result.Tag == FORWARDED then
                RA := Result.Leader
                Apply RA to register bank
        signal operation complete
end RegisterAccess
```

Figure 5
Operation of a Register Read or Write Operation

Effects on Instructions

As part of the variable forwarding technique, the load and store instructions are augmented to update the forwarding information. The augmentation of these instructions is such that no auxiliary instructions are needed to maintain the register-address mappings. This section presents the details of the operation of these instructions along with several new instructions designed to optimize register management.

At machine initialization, the associative memory entries are all initialized to 0, where it is assumed that no valid data object will ever be assigned this address. The register tags are all set to EMPTY, indicating that the registers are not being forwarded and have not been loaded with any values. The register forward, backward, and leader link entries may be left uninitialized because they are only accessed after the tag and link fields have been modified. Like register contents and TLB's, register forwarding tags and the contents of the variable forwarding associative memory must be saved and restored at context switches.

The load instruction is used to move a variable from memory into a register. The instruction updates the register address tag for the register being loaded and creates forwarding groups whenever an attempt is made to load a register with the contents of a memory location that is already associated with another register. To begin association of a register with a variable from memory, but giving the variable an initial value instead of retrieving its value from memory, a special instruction is provided: init. The operands of init specify the address of the variable, the register, and the initial value to be loaded into the register. Additionally, we provide a move instruction that does not perform the usual initialization of the register address field. This instruction is useful for accessing memory mapped devices without forwarding future references to a register, thus losing access to the actual device control or status word. Figure 6 presents the detailed operation of the load instruction.

```
Load Register, Value:
/* set up new tag and address */
EA := Effective Address of Operand.
Request Fetch of memory operand at EA
/* these checks are performed in parallel with the memory request */
Leader := TLBcheck(EA)
        if Leader <> FAIL then
                Register.Address := 0
                Register.Backward := Leader
                Register.Leader := Leader
                Register.Tag := FORWARDED
                if Leader.Tag == LOADED then
                        Register.Forward := Register
                        Leader.Tag := LEADER
                else
                        Register.Forward := Leader.Forward
                        Leader.Forward.Backward := Register
                Leader.Forward := Register
                ignore result of memory read
        else
                Register.Tag := LOADED
                Register.Address := EA
                Register.Forward := Register
                Register.Backward := Register
                wait for result of memory read
                Register.Value := result
signal operation complete
```

Figure 6
Operation of the LOAD Instruction

The store instruction dissociates the mapping of a variable and register so that the register may be used for other purposes. The instruction removes the register from its forwarding group and clears the register's addressing information. If the register was a leader then a new leader is selected from the group and the value from the register is copied to the new leader. If the register was the only register in the group, the value is copied to the associated memory location and the group is destroyed. If the value in the register is not live, there is no reason to perform the save. A special instruction, kill, is provided that clears the address information without storing the register contents. Figure 7 shows the detailed operation of the store instruction. The kill instruction performs the same actions with the exception of the memory write operation.

Changes to Code Generation

The altered semantics of the load and store instructions and the presence of the forwarding hardware affect the way the code generator should operate, both in terms of correct instruction selection and in terms of register allocation.

When generating code for a three address statement of the form, Dest := Src1 op Src2, the code

generator should always assure that the address associated with the register holding the value corresponds to the address of `Dest`. If `Dest` is the same as `Src1` or `Src2`, the code generator should generate a sequence that begins by `loading` `Dest` into a register and then performing the operation with the register as the result of the operation. If `Dest` is disjoint from `Src1` and `Src2`, then the code generator should emit an `init` of the register with the address of `Dest` and either `Src1` or `Src2` as the initial value followed by the operation with the result in the register.

```
Store Register:
case Register.Tag of
    LEADER:
        NewLeader := Register.Forward
        NewLeader.Value := Register.Value
        NewLeader.Address := Register.Address
        if NewLeader.Forward == NL then
            NewLeader.Tag := LOADED
        else
            R := NewLeader.Forward
            while R.Forward != R do
                R.Leader := NewLeader
                R := R.Forward
            R.Leader := NewLeader
    LOADED:
        initiate & wait for completion of transfer
        of Register.Value to Register.Address
    FORWARDED:
        Register.Backward.Forward := Register.Forward
        Register.Forward.Backward := Register.Backward
        if Register.Forward == Register then
            if Register.Backward.Tag == LEADER then
                Register.Backward.Tag := LOADED
Register.Tag := EMPTY
Register.Address := 0
signal operation complete
```

Figure 7
Operation of the STORE Instruction

After a value has been computed into a register, the result should either be `stored` to memory, if it is live, or `killed`, if it is no longer needed, before the register is used for another computation.

Operations that reference values through pointers or addresses can also be compiled to place the values into registers. For example, a pass-by-reference parameter can be `loaded` from memory into a register, operated upon, and then `stored` before the procedure exits. Similarly, fields of dynamically allocated objects can be `loaded` into registers and operated on, followed by a `store` instruction.

Example

This section presents an example program that demonstrates the operation of the variable forwarding hardware in a variety of aliasing circumstances. The example program which contains a procedure to buffer a list of outputs into several columns, is written in a Pascal-like pseudo-code and is presented in Figure 8. The main program has three calls to the procedure, each of which creates a different aliasing circumstance. The first call does not introduce any aliases. The second call introduces an alias between the global variable CurrentRow and the formal parameter Value. As both of these variables are allocated to registers, this call demonstrates the operation of register forwarding. The last call introduces an alias between an element of the buffer array, Page, and the formal parameter PageNum. As only the parameter is allocated to a register, this call demonstrates the operation of memory forwarding. The pass-by-reference parameter PageNum would have likely been coded as a global variable in a real program, but is included as a parameter to demonstrate the improved code that can be generated for accesses through pointers.

```
Main Program
Constants:
    MAXCOLS = 5;
    MAXROWS = 60;
Variables:
    Page:array[1..MAXCOLS]
    of array [1..MAXROWS] of
        integer;
    CurrentRow:integer=1;
    CurrentCol:integer=1;
    PageNum:integer=1;
    I:integer=4;
Begin
    MultiColOut (PageNum,I);
    MultiColOut (PageNum,
        CurrentRow);
    MultiColOut (Page[I][I],
        PageNum);
End.
```

```
Procedure MultiColOut (
    var PageNum:integer;
    var Value:integer);
begin
Page[CurrentCol][CurrentRow] :=
    Value;
CurrentRow := CurrentRow + 1;
if (CurrentRow > MAXROWS) then
    CurrentRow := 1;
    CurrentCol := CurrentCol + 1;
    if (CurrentCol > MAXCOLS )
        then
            OutputPage();
            CurrentCol := 1;
            PageNum :=
             PageNum + 1;
end;
```

Figure 8
Example Program Source

In the first call, the register configuration during the body of the procedure will be:

r7	t0(PageNum)	r10	t3
r8	t1(Value)	r11	CurrentRow
r9	t2	r12	CurrentCol

The only memory access that will occur (other than `loads` and `stores`) will be to an element of the array Page, and the pass-by-reference parameter PageNum. As these are not aliased to any of the values in the registers, there will not be any associative memory hits during the execution of the procedure, and thus no forwarding will occur.

```
Main:   push    #I              ; stack parameters
        push    #PageNum
        call    #2,MultiColOut  ; first call
        push    #CurrentRow     ; stack parameters
        push    #PageNum
        call    #2,MultiColOut  ; second call
        push    #PageNum        ; stack first parameter
        init    r0,t1,I         ; t1 := I
        dec     r0              ; t1 := t1 -1
        init    r1,t2,#Stride   ; t2 := Stride
        mul     r0,r1           ; t2 := t1 * t2
        mul     #ElemSize,r0    ; t1 := t1 * ElemSize
        add     r1,r0           ; t1 := t1 + t2
        add     #Page,r0        ; t1 := t1 + Addr(Page)
        push    t1              ; stack computed parameter
        call    #2,MultiColOut  ; third call
        halt    ; terminate program
MultiColOut:
        pop     r7              ; retrieve address of PageNum
        load    r7,(r7)         ; associate with r7, get initial value
        pop     r8              ; retrieve address of Value
I0:     load    r8,(r8)         ; associate with r8,  get initial value
I1:     load    r11,CurrentRow  ; Put CurrentRow in r11
        init    r9,t2,r11       ; set address of t2 for r9, initialize
        dec     r9              ; t2 := CurrentRow -1
        mul     #ElemSize,r9    ; t2 := t2 * ElemSize
        load    r12,CurrentCol  ; Put CurrentCol in r12
        init    r10,t3,r12      ; set address of t3 for r10, initialize
        dec     r10             ; t3 := CurrentCol -1
        mul     #Stride,r10     ; t3 := t3 * Stride
        add     r10,r9          ; t2 := t2 + t3
        kill    r10             ; t3 no longer needed
        add     #Page,r9        ; t2 := t2 + Addr (Page)
I2:     move    r8,(r9)         ; store t1 at address computed in t2
I3:     kill    r8              ; Value has no further uses
        kill    r9              ; nor does t2
        inc     r11             ; CurrentRow := CurrentRow + 1
        cmp     r1,#MAXROWS
        jple    L1
        move    #1,r11          ; CurrentRow := 1
        inc     r12             ; CurrentCol := CurrentCol + 1
        cmp     r12,#MAXCOLS
        jple    L2
        call    #0,OutputPage
        move    #1,r12          ; CurrentCol := 1
        inc     r7              ; PageNum := PageNum + 1
L1:L2:  store r7                ; PageNum is live, store
        store   r11             ; CurrentRow is live, store
        store   r12             ; CurrentCol is live, store
        ret
```

Figure 9
Pseudo-Machine-Code for Example Program

In the second call, the register configuration during the procedure will be the same as during the first call, but in this case, the access to Value is also an access to CurrentRow. The instruction at label I1 will attempt to fetch CurrentRow into r11 and set its address field accordingly. Because of the alias of CurrentRow and Value, the address of CurrentRow is already associated with the register holding Value (r8) due to the instruction at label I0. Thus this `load` instruction will create a forwarding group with r8 as the leader. Should there be an access to the memory location, r8's tag would cause an associative memory hit and redirection would be to r8. Should r11 be accessed, the forwarding tag would cause a redirection to the leader of the group, r8. The instruction at label I3 performs a `kill` on r8. This removes the leader of the forwarding group. The `kill` operation will copy the value of r8 into r11 and set r11's address field to cause a trap on the address of CurrentRow.

In the final case, a similar sequence occurs, however the address associated with r8 will be the address of an element of the page array. When execution conditions are such that the address computation for Page[CurrentRow][CurrentCol] yields the same address as associated with r8, the `move` instruction at label I2 will be redirected to write to r8.

Multiprocessor Architectures

While the discussion to this point has centered on a uniprocessor architecture, the variable forwarding technique is a useful addition to other processor configurations as well. Variable forwarding can be directly applied in any message passing processor architecture where individual processors cannot directly access the local memories of other processors.

A shared memory multiprocessor using variable forwarding exhibits a phenomenon similar to the cache coherency problem and may be managed with similar techniques. When a memory reference occurs that is not forwarded by the local hardware, the forwarding associative memory of all other processing units must be checked to assure that the value is not loaded into a register in another unit. This is similar to the situation that arises when a block has been loaded into one processing unit's cache and another processing unit generates a request for the same block.

One possible approach to avoiding inconsistencies among processors is to build an associative memory that contains the addresses loaded into all of the registers in the multiprocessor. Register tags must also be extended to permit forwarding chains to cross processing units. This memory could then be updated and consulted by each processing unit for each memory reference as in the uniprocessor case, but with hits resulting in either a local access or a request for access to a register in another processing unit. The associative memory could also be replicated in each processing unit, with each unit monitoring the memory requests of all processing units in order to update its local view of the multiprocessor's complete register set. While replication eliminates contention for a single associative memory, it quickly becomes impractical as the number of processing units increases because each unit must have an associative

memory large enough to represent all registers in the complete multiprocessor.

An alternative organization that uses the same basic approach but does not enlarge the individual processing unit's associative memories can be used instead. Each processing unit has an associative memory reflecting the contents of its own register set. Each unit must monitor all memory accesses by all units, checking each against its own associative memory. Updates to the local associative memory are triggered only by local `load` and `store` instructions. When a hit occurs on an address generated by another processor, an interrupt is sent to that processor, signaling that the value must be fetched from the register copy held by the interrupting processor. As in the other approaches, register tag fields must contain a field to permit forwarding chains to cross processing units and a communication protocol for accesses to other processing units' registers.

Further techniques for applying variable forwarding to a shared memory multiprocessing environment may be patterned after existing cache coherency strategies.

Instruction scheduling must take into account the possibility that access to a potentially aliased variable (subscripted variables, pointers, reference parameters) that is loaded into a register may require an added register cycle time to complete. This added time must be scheduled but may not be needed if the execution of the program is such that the variable does not actually have an alias.

Conclusions

The use of register forwarding hardware can substantially improve the quality of code that can be generated for a program containing aliasing. This approach obviates the need for aliasing analysis when performing register allocation. This technique cannot eliminate the need for aliasing analysis for application of optimizations.

Code generated for use with this hardware is substantially improved over code that must accommodate aliases directly. Global register allocation is more effective because `loads` and `stores` can be moved to the extremes of spans without concern for register-memory consistency. The static and dynamic numbers of `loads` and `stores` are reduced. The access time for variables which are held in registers, but which are referenced by their corresponding memory address is shorter than the normal memory cycle, thus improving execution speed over code for variables with known aliases. The normal register access time is not extended, but register forwarding may extend register access time for aliased variables. The number of instructions that can be eliminated through removal of guard code far outweighs this negative effect. The classes of values that can be allocated to registers are expanded to include elements of arrays, objects referenced through pointers, and pass-by-reference parameters.

The hardware required to implement register forwarding is based on known technology and can be incorporated into a processor without extending its cycle time. The nature of the variable forwarding problem allows the use of a simplified associative memory due to the information needed when a matching entry is found and due to the guarantee of only single matching entries. The chip real estate needed for this hardware can be recovered from the area often used by register window hardware.

Further analysis of the effectiveness of this technique is being carried out through implementation of a simulator for a RISC processor equipped with variable forwarding hardware and a compiler designed to perform interprocedural register allocation and to produce code that takes advantage of this hardware.

Acknowledgement

We would like to thank one of the Supercomputing '90 referees for comments that were helpful.

References

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

[2] H. Dietz, C.H. Chi, "CRegs: A New Kind of Memory for Referencing Arrays and Pointers", *Supercomputing '88*, pp. 360-367, 1988.

[3] J.R. Ellis, *Bulldog: A Compiler for VLIW Architectures*, MIT Press, Cambridge, Mass., 1986.

[4] K.D. Cooper, K. Kennedy , "Fast Interprocedural Alias Analysis", *Conference Record of Sixteenth ACM Symposium on Principles of Programming Languages*, pp.49-59 , January 1989.

[5] J.R. Larus, P.N. Hilfinger, "Detecting Conflicts Between Structure Accesses", *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 621-34, 1988.

[6] E.W. Myers, "A Precise Inter-procedural Dataflow Algorithm", *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages*, pp. 219-230, January 1981.

[7] G.J. Myers, *Advances in Computer Architecture*, Wiley Interscience, John Wiley & Sons, New York, 1982

[8] D.W. Wall, "Register Windows vs. Register Allocation", *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 67-78, 1988