Runtime Register Allocation

Kemal Ebcioğlu and Vivek Sarkar
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598, USA
{kemal,vsarkar}@us.ibm.com

Kartik Agaram
University of Texas at Austin
1 University Station C0500
Austin, TX 78712-0233, USA
akkartik@cs.utexas.edu

Abstract

The use of registers instead of memory operands is an effective performance enhancement as well as a power saving mechanism, but compilers still cannot allocate many of the memory references of a program into registers, for two fundamental reasons unrelated to the size of the register file: (1) dynamically varying load/store operand addresses, and (2) inherent limitations of compile time alias analysis. Previously proposed architectural features such as CRegs [2] or IA-64 ALAT [1] mechanisms have only addressed the aliasing portion of the two fundamental impediments.

In this paper, we have designed an ideal "limit register allocation machine," as a theoretical tool to gain insight into the essence of the register allocation problem, and to address both of the fundamental impediments. The limit machine uses only register operations in its ISA (performs memory operations only for automatic register spills and fills) and has a unique capability to modify the register fields of instructions at run time. Based on the experience with this limit machine, we have also proposed:

(1) A software emulation method for the limit machine, which, after performing optimizations such as partial redundancy elimination, can result in efficient register allocation of memory references for loads/stores with dynamically varying addresses, which could not be register-allocated with any of the previous techniques.

(2) A hardware implementation of concepts from the limit machine, by adding a dynamically changing operand location prediction field ("Predicted Register Number") to each load/store instruction, which can improve the performance of data L1 caches by speculatively accessing the predicted operand location in the D-L1 cache array directly (as if it were a "register file"), as soon as the instruction is available, without first going through the base register read, address arithmetic, address translation, and associative search phases.

1 Introduction

Register allocation enables us to place the contents of a memory address `X` into a register `r` and allows us to refer to the contents of `X` (perhaps a 32 or 64 bit address, obtained by accessing a base register, possibly performing address arithmetic, performing virtual to real translation and then accessing an associative cache) by its current physical location `r` (a short, perhaps 5-8 bit, register number, directly encoded inside the instruction). By placing frequently referenced memory variables into registers, and in effect compressing the address trace of a program by replacing full-width addresses of load/store operands with references to register numbers, a compiler can achieve significant performance benefits, as well as power reduction, compared to using a standard memory hierarchy accessed through loads and stores.

But precisely how does a compiler decide when the contents of a memory location `X` can be allocated in a register `r`? Although enumerating the many alternative register allocation research approaches of the past, e.g., [3, 4, 5], will likely be too long for our purposes, we will present here what we believe to be a useful "canonical abstraction" of the register allocation problem.

To separate the basic feasibility of register allocation from the restrictions coming solely from the size of the register file, we will first assume that we have an unlimited number of registers. Suppose the compiler has somehow determined that a *candidate set* of static[1] load-store instructions in a code fragment will refer to a fixed memory address `X` during the execution of the code fragment. `X` is in general not a constant address, but it is a simple function of the initial contents of the machine registers, before the code fragment starts execution. For example, `X` could be "the value of the expression `r3+4` in the starting state," which will stay fixed during a particular execution, but which may indicate different addresses in different executions, depending on the initial value of `r3`. Below, we will describe a code transformation to keep the contents of such a memory address in a register, throughout the entire code fragment.

The canonical register allocation code transformation. *Given a single-entry code fragment, a memory address* `X` *(a constant or register expression, to be evaluated at the starting state), and a static set of load/stores inside the code fragment (the* candidate set*), we define* canonical register allocation *as the following code transformation:*

- *Pick a currently unused register* `rX`*. Change the references to memory in the candidate set of load/store instructions (*`L rt=..., ST rt,...`*) into references to the register* `rX` *(the loads and stores in the candidate set thus becoming* `LR rt=rX, LR rX=rt`*, respectively*[2]*.)*
- *Load the contents of* `X` *into* `rX` *just before entering the transformed code fragment, and store it back to address* `X` *just after exiting the transformed code fragment*[3]*.*

As a concrete example, consider the C source code in Figure 1, which we will use as a running example throughout the paper, and the corresponding original assembly code in Figure 2. Assuming that, for the sake of this example, it was determined that `X` and `Y` will never refer to the same location, the canonical register allocation technique we just described can change the original code in Figure 2 into that of Figure 3, by keeping the contents of address `X` inside register `rX` throughout the code fragment. Here the candidate set of load/store instructions are marked as "candidate" in the comments. Starting from the version of the code in Figure 3, the compiler can then perform copy propagation (or just convert the program to SSA form and back) to eliminate all or most of the `LR` statements. Hence, very efficient three-register operations can be obtained.

```
sub(int *X, int *Y) {
  do {
        if (...) { Y++;}
        *X= (*X) + (*Y);
  } while(...);
}
```

Figure 1. Original C code

Note that if `Y` and `X` do point to the same location during some iteration of the while loop, this transformation is not correct, since the remaining load `(*Y)` in the loop will not see the updates to address `X` in memory and

[1] A static set of load-store instructions is just the set of instruction addresses of the loads-stores in the binary code generated by the compiler. The instruction address uniquely defines the load/store instruction.

[2] Here, `LR rt=rX` means: copy register `rX` into register `rt`.

[3] Note that optimizations can later remove these initial loads and final stores, when they are not needed. If the reference to `X` occurs inside a conditional statement within a code fragment, such as `"if (X!=NULL) L rt=X"`, the initial load and final store of `X`, can lead to exceptions that were not occurring the the original code. Analysis techniques to identify cases where there will be no extra exceptions despite the speculative code motion, have been described in [6], in the context of speculative code motion out of loops. In this paper, we will propose *speculative loads and stores* for accomplishing the initial load and final store of the canonical register allocation transformation, which preserve the exception behavior of the original program. Speculative loads from a non-existent address yield the special $\perp$ value without causing an exception (similar to the $\perp$ value (33rd bit) in [9] or `NaT` value in IA-64 [1]), and speculative stores to a non-existent address are treated as no-ops. When the memory address is valid, the speculative loads and stores behave as normal ones. Also `LR rt=rX` or `LR rX=rt` operations for a speculatively loaded `rX`, must also first check if `rX` is $\perp$, and cause an exception if so. Speculative loads and stores can be used either sparingly, when the possibility of introducing extra exceptions cannot be disproved, or all the time, the latter case for mimicking the exception behavior of the original program exactly, for all input states. On existing machines such as the IA-64, speculative loads and stores and trapping `LR's` can be emulated by macro-expansion, and then optimized as usual, all in software. For the rest of the paper, we will assume an exception-free environment, where all memory addresses are accessible, to simplify the formalism.

```
Loop:   ...
        BC      L1
        A       r3=r3,4   // Y++
L1:     L       r12=(r6)  // refers to X -- candidate
        L       r4=(r3)   // (*Y) refers to ?
        A       r12=r12,r4 // (*X)+(*Y)
        ST      r12,(r6)  // refers to X -- candidate
        ...
        BC      loop
```

Figure 2. Initial assembly code

```
        L       rX=X // (L rX=(r6); LR r6'=r6)
Loop:   ...
        BC      L1
        A       r3=r3,4
L1:     LR      r12=rX
        L       r4=(r3)    // (*Y) refers to ?
        A       r12=r12,r4
        LR      rX=r12     // can be optimized as A rX=rX,r4
        ...
        BC      loop
        ST      rX,X // (ST rX,(r6'))
```

Figure 3. After allocating contents of memory location X in register rX

will load the incorrect, old contents of `X` (the updates are done only to the register `rX` in the transformed loop). Also, if there is an instruction in the loop that alters the base register `r6` of the instructions `L r12=(r6)` and `ST r12,(r6)`, making `r6` point to an address different than `X` during some loop iterations, the transformation is again incorrect: this is because a given register such as `rX` can represent (i.e., cache) the contents of only one memory address `X` throughout the execution of the loop, not the contents of multiple memory addresses at the same time. Otherwise, the transformation is correct.

The requirements for performing the canonical register allocation transformation correctly are given below.

Conditions for canonical register allocation. *For any starting state, the execution of the code fragment must satisfy the following:*

1. *All members of the candidate set of loads and stores must refer only to* `X`*, and*
2. *No load or store outside of the candidate set (but inside the code fragment) must ever refer to* `X`

These two conditions are in fact necessary and sufficient conditions for correct canonical register allocation, when one does not consider register allocation optimizations that rely on any special property of a code fragment (a property that is not shared by all code fragments)[4]. In the appendix, we provide a proof of the necessity and sufficiency of these conditions.

In order to link these conditions to the terminology of fundamental impediments described in the abstract, we can say that violations of condition (1) correspond to the "dynamically varying load/store operand addresses" and that the violations of condition (2) correspond to the "inherent limitations of the compile time alias analysis."

Scalar variables of a procedure allocated on the stack (automatic variables), whose address is not taken, trivially satisfy conditions (1) and (2), and this method could be used for traditional register allocation as done by compilers (which can of course be done by many other, more direct methods). Also, when the compiler can prove these two conditions for a set of loads/stores by sophisticated interprocedural alias analysis, e.g. as in [7], many other kinds of variables (e.g., on the heap or in the static external variables area), can be allocated in registers as well. Note that the scope of the code fragment will in general be changed while making register

[4] We are not considering some special case optimizations that do not apply to all code fragments, such as (1) letting a load outside of the candidate set refer to `X` during the code fragment, when `X` is read-only throughout the code fragment, or (2) allowing a member of the candidate set to refer to two distinct addresses `X` and `Z` when addresses `X` and `Z` have equal contents.

allocation decisions for a procedure body. For example, register allocation of an array element `A[i]` in an inner loop `for(i=1..N){...A[i]...A[i]...A[i]}` can be achieved by considering only the for loop body as the code fragment, while the entire procedure body can be used as the code fragment for register allocation of an automatic scalar variable, whose address is not taken.

When condition (1) is guaranteed, since, e.g., each member of the candidate set uses the same base register that is unchanged throughout the code fragment, condition (2) can be relaxed by using hardware techniques such as CRegs[2] or the IA-64 ALAT [1] mechanism. These techniques tolerate references to `X` from outside the candidate set of loads and stores, if these references are infrequent, and can therefore perform speculative register promotion of the candidate set, in the presence of pointers which cause static alias analysis to fail.

To date, we are not aware of any register allocation method that is able to overcome both of the conditions (1) and (2) simultaneously. The present paper aims to propose novel hardware and software techniques to overcome both of the conditions (1) and (2).

In section 2, we introduce a new dynamic view of the register allocation problem by defining a "limit register allocation" machine, as a new theoretical tool for gaining insight into register allocation. This machine is able to rewrite register fields of the binary program at run-time, and force both of the conditions (1) and (2) to become true as a result of the binary rewriting, as soon as they cease to be true as a result of dynamically changing operand addresses in loads/stores, or inherent limitations of static alias analysis. For example, when a load instruction that was referring to `Y` suddenly starts referring to `X`, the corresponding `LR rt=rY` instruction in the transformed program (where all loads and stores have been converted to `LR` operations) suddenly starts using register `rX` (containing the memory operand at address `X`) instead of register `rY` (containing the memory operand at address `Y`), by binary rewriting. Inspired from this ideal machine, we then propose a new software emulation technique and a new hardware technique for aggressive register allocation: Section 3 describes the software emulation technique and section 4 describes the hardware technique. Section 5 discusses the related work, and section 6 concludes the paper. The appendix provides a proof that conditions (1) and (2) are both necessary and sufficient, with the given assumptions.

2 The limit register allocation machine

Having defined the two necessary and sufficient conditions for register allocation, we now define a limit register allocation machine that continually tracks the program and exploits all register allocation opportunities.

We first change each original load (`L`) and store (`ST`) in the program to new instructions `L*, ST*`, that also contain a new register number field called `ncr` (a *Named Cache Register*), indicating an entry in a new `Named Cache Register` (`NCR`) file we are introducing to the architecture, as shown in Figure 4.

```
Original load/stores:
L rt=(rb)  //R[I.rt]=M[R[I.rb]] --I:instruction, R:registers, M: memory
ST rt,(rb) //M[R[I.rb]]=R[I.rt]
   ==>
New instructions:
L* rt,(rb),ncr  //R[I.rt]=NCR[I.ncr].data, if (R[I.rb]==NCR[ncr].addr)
ST* rt,(rb),ncr //NCR[I.ncr].data=R[I.rt], if (R[I.rb]==NCR[ncr].addr)
              //NCR: Named Cache Register file
```

Figure 4. Transformation of `L, ST` instructions to `L*, ST*`.

The precise semantics of `L*, ST*` are shown in the pseudo-code in Figure 5.

Here we are adding to the architecture a new register file structure `NCR` - or Named Cache Register file - each of whose entries have the three fields: data, address, and dirty bit. For now, assume that all that memory accesses are of word size[5]. The `NCR` is a collection of single-entry direct-mapped caches, that one can refer to by name (`ncr` field) in an instruction (hence the name Named Cache Register). When the access to the single-entry direct-mapped cache named by the `ncr` field causes a "cache miss," then entries in `NCR` need to searched by address,

[5]Partword memory accesses, overlapping operands, and different virtual addresses aliasing to the same memory location, are implementation issues that can be added to the basic `NCR` mechanism. We will not clutter the current presentation with them.

```
Limit machine instruction execution loop:
switch(I) { //I is the next instruction to execute
case L* rt,(rb),ncr :
        if (R[I.rb]!=NCR[I.ncr].addr) {I.ncr=accessNCR(false,R[I.rb]);}
        // here R[I.rb]==NCR[I.ncr].addr
        R[I.rt]=NCR[I.ncr].data; ...; break;
case ST* rt,(rb),ncr :
        if (R[I.rb]!=NCR[I.ncr].addr) {I.ncr=accessNCR(true,R[I.rb])}
        // here R[I.rb]==NCR[I.ncr].addr
        NCR[I.ncr].data = R[I.rt]; NCR[I.ncr].dirty=true; ...; break;
default: <execute instruction as usual>; }

registerNumber accessNCR(boolean isWrite, address y) {
        //return an ncr representing the contents of the memory address y
        if ((Exists registerNumber ry) (NCR[ry].addr==y)) {return ry;}
        registerNumber ry=chooseEvictable(NCR); // find an ncr to evict
        if (NCR[ry].dirty) M[NCR[ry].addr]=NCR[ry].data; // write back if dirty
        if (!isWrite) NCR[ry].data=M[y]; //if a load, read initial value
        NCR[ry].addr=y; NCR[ry].dirty=false;
        return ry; }

Initialization:{for(I: all L*,ST*) I.ncr=rNULL;}
//NCR[rNULL].addr does not match any real load/store address

Wrap-up: for(i:0..NCR'last){if (NCR[i].dirty) M[NCR[i].addr]=NCR[i].data;}
```

Figure 5. Semantics of the limit register allocation machine with `L*,ST*`

similar to an associative cache (the associativity is an implementation issue which also depends on the replacement policy). One observation we could make is that, when there is a hit in the single-entry direct mapped cache named by the `ncr` field, i.e. `NCR[I.ncr].addr==R[I.rb]`, the instruction `L* rt,(rb),ncr` performs a register copy `R[I.rt]=NCR[I.ncr].data`, which we can abbreviate as `LR rt=ncr`, and the instruction `ST* rt,(rb),ncr` performs a register copy `NCR[I.ncr].data=R[I.rt]`, which we can abbreviate as `LR ncr=rt`.

Optionally, for the purpose of having a single uniform register file (more suitable for additional optimizations such as copy propagation) one could map the normal general registers `R` into a portion of the `NCR` register file structure, such that `R[i]==NCR[i].data, i=0..31`, and where `L*` and `ST*` instructions use `ncr` numbers greater than or equal to 32 (assuming there are 32 general purpose registers).

The semantics given in Figure 5 is perhaps best explained by an example, namely our running example in Figure 1. The limit register allocation machine will first initialize all `L*, ST*` ncr fields to `rNULL`. Let `X0, Y0` be the initial values of pointer variables `X, Y`, respectively, before the while loop is entered. Suppose initially the memory address `Y0` points to the word just before `X0`. I.e., assume `X0==Y0+1` in terms of C address arithmetic. Assume that during the first two iterations of the while loop, `Y` stays the same (equal to `Y0`), but in the third iteration, `Y` gets incremented in the instruction `A r3=r3,4`, thus becoming equal to `X0`.

During iteration 1, assume that `ncr` number `rX` is chosen to represent the memory location `X0` by the `accessNCR` routine, when the first `L*` instruction (I1) is executed. Also assume that `rY` is the `ncr` number that is chosen to represent memory address `Y0`, during the execution of the second `L*` (I2). The store instruction (I3) (which also refers to `X0`) will discover that ncr number `rX` already represents address `X0` and will start using `rX` as its ncr. The state of the binary program and the contents of the `NCR` register file entry numbers `rX` and `rY` are shown in Figure 6, after iteration 1.

Iteration 2 gets executed uneventfully, where the `ncr` entries `rX` and `rY` are re-used by the `L*, ST*` instructions I1,I2,I3, without going to memory. (`NCR[rX].data` is updated by the `ST*` instruction I3). But in iteration 3, a change occurs as shown in Figure 7. Since the base register `r3` of original second load instruction (I2) has been incremented (becoming `X0` instead of `Y0`), and since the instructions I1, I3 have already started referencing `X0` and `(*X0)` is already in an `NCR` register `rX`, i.e., `(NCR[rX].data==(*X0))`, the `L*` instruction I2 is rewritten to refer to `rX` instead of `rY`. As long as base register of the second load I2 does not change again, the iterations can keep accessing registers only, without needing to go to memory.

When the program (while loop in this case) finishes, `NCR[rX].data` is written into the corresponding address

```
Loop:   ...
        BC      L1
        A       r3=r3,4
L1:(I1) L*      r12,(r6),rX (LR r12=rX) //NCR[rX].data=(*X0),NCR[rX].addr=(X0)
   (I2) L*      r4,(r3),rY  (LR r4=rY)  //NCR[rY].data=(*Y0),NCR[ry].addr=(Y0)
        A       r12=r12,r4              //optimize as A rX=rX,rY
   (I3) ST*     r12,(r6),rX (LR rX=r12) //NCR[rX].data=R[r12]
        ...
        BC      loop
```

Figure 6. Dynamic snapshot of code after iteration 1

```
loop:   ...
        BC      L1
        A       r3=r3,4
L1:(I1) L*      r12,(r6),rX (LR r12=rX)
   (I2) L*      r4,(r3),rX  (LR r4=rX) //NCR[rX].data=(*X0), NCR[rX].addr=(X0)
        A       r12=r12,r4             //optimize as A rX=rX,rX
   (I3) ST*     r12,(r6),rX (LR rX=r12)//NCR[rX].data=R[r12]
        ...
        BC      loop
```

Figure 7. Dynamic snapshot of code after iteration 3

`NCR[rX].addr` since it has been overwritten (dirty). Notice that the limit machine has successfully done register allocation in our difficult code example (which is not register allocatable via existing techniques), using binary rewriting.

2.1 How the limit register allocator always meets the two conditions

It is interesting to discuss the relationship between the limit register allocator machine, and the two conditions described in the introduction. At a given point in the execution trace, for a memory address `X`, let `C(X)` be the set of static load and store instructions defined as {`I|NCR[I.ncr].addr==X`}. These instructions are the static loads and stores in the program whose last executions in the execution trace referred to the address `X`. For a given memory address `X`, let $\{t_0, t_1, ..., t_k\}$ be the set of points (dynamic instruction sequence numbers) in the execution trace, where a change occurs to `C(X)`. Note that during any trace fragment starting at instruction sequence number t_i in the trace and ending just before instruction sequence number t_{i+i}, where `C(X)` does not change, (1) all loads stores in the program that refer to `X` (the members of `C(X)`), refer only to `X`, and (2) no loads stores outside of `C(X)` refer to `X`. When these conditions are violated by the instruction with sequence number t_{i+1} (e.g. an instruction that was referring to `X` starts referring to `Y`, or an instruction that was referring to `Y` starts referring to `X`), the limit machine rewrites the binary program (e.g., by changing the ncr `rX` to `rY`, or `rY` to `rX` in the offending instruction), to make the conditions (1) and (2) true again, for the next trace fragment, t_{i+1} to t_{i+2}. Hence for any given address `X`, the "limit register allocation" machine in fact continuously forces the conditions (1) and (2) (mentioned in the introduction) to be true, by dynamic binary rewriting. Thus, the limit machine exploits all register allocation opportunities for `X` in the execution trace.

2.2 The opportunity for register allocation

Based on a simulation of the limit register allocator machine on sampled PowerPC traces coming from 11 SPECInt2000 benchmarks[6], the opportunity for promoting loads/stores to register accesses (precisely, the percentage of dynamic `L*/ST*` where `NCR[I.ncr].addr==R[I.rb]` was immediately true at the beginning of instruction execution, i.e., where the load/store operand was found immediately in the ncr entry named in the instruction) is between 15% and 90%, as shown in Figure 8. Figure 9 shows results with a finite number of `NCR` entries. Again, the same metric is reported (percentage of dynamic `L*/ST*` where there was an immediate hit in the ncr entry named

[6]The "parser" trace was not available to us at this time

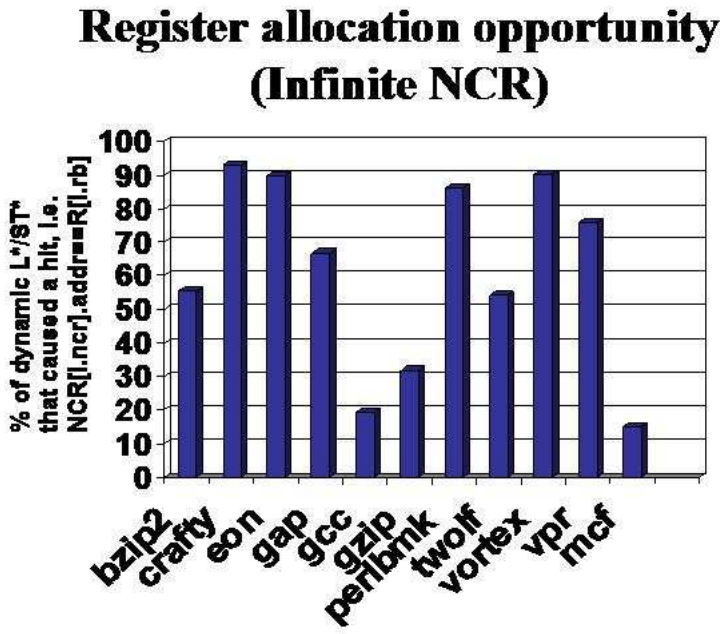

Figure 8.

in the instruction). But the percentages are lower than the infinite `NCR` case, since even if a load or store uses the same address that it used during its last execution, its ncr entry may have been evicted from the `NCR` since the last execution of this load/store, because of the small size of the `NCR`. The results show that 2048 registers with LRU replacement are almost as good as an infinite number of registers, for this particular metric. The Belady OPT replacement approach interestingly does very well, even with 128 registers.

3 Software emulation of the limit machine

Here we will describe a compilation technique that will, in principle, simply emulate the limit register allocation machine algorithm by replacing each load/store by the macro expansion of `L*,ST*` defined by the algorithm of Figure 5, and then optimizing the resulting code using first principles.

The data and address fields of each `NCR` entry can be represented as discrete symbolic registers to the compiler. Firstly, there is no advantage in simulating the actual writing of the `ncr` field of an instruction, and performing an address comparison first to the instruction's ncr field. Instead, in the macro expansion of a load or store operation, the memory address must be compared with the address field of all applicable `NCR` entries, starting from the entries that are more likely to match (perhaps based on profiling feedback). If there is a match, the data field of the matching `NCR` entry (another symbolic register) can be read or written. When no address match is found among the applicable `NCR` entries, the macro expansion code needs to access memory and simulate the eviction of an existing `NCR` entry. It is better to choose an `NCR` entry to evict, which is known at compile time, and avoid replacement policies that find such entries dynamically: This approach will cut down on the number of `NCR` entries to compare against, later. A dirty bit is not needed among the fields of the `NCR` entries, since the compiler will know which `NCR` entries may have been modified at a given point. Also, it makes sense to do the optimization in modest sized code fragments where the applicable `NCR` entries are initialized at the beginning of the code fragment, and are then flushed back to memory at the end, if overwritten.

To make the compilation result practical, the number of address comparisons should be minimized. We assign a unique "home ncr" to each group of loads/stores, where the operand address is equal within the group, but may

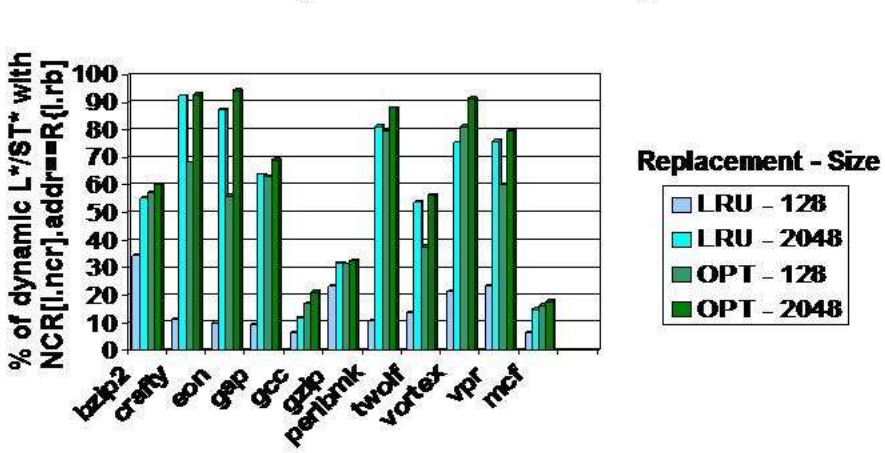

Figure 9.

differ among the groups (the distinct symbolic address expressions normally derived by an ILP compiler for each load/store for alias analysis can be used for representing each distinct group). When all applicable ncr comparisons fail and the `L*,ST*` emulation has to go out to memory, the home ncr will be chosen for eviction. A base register of a given load/store should be compared only with the home ncr's of loads/stores which (1) may have preceded this one on some execution path (2) may be aliased with this one. If there is an address expression which is definitely equal to the current load/store address, and its home ncr is known to be assigned by now, no further address comparison is needed; the data is already available in that ncr.

The general approach to compilation, is to start with a small number of comparisons for a load or store, and gradually apply optimizations that reduce the address comparisons until (hopefully) there are none. The following optimizations are among those that are useful:

- Advanced static alias analysis. This cuts down the address comparisons.
- Assertion propagation to aid detection of redundancies. E.g., propagating assertion `(x==y)` in the context of `x=y;...;` or `if (x==y) {...}`. These are important for deducing that an address comparison is redundant.
- Loop peeling to expose redundant or loop invariant computations. Often an assertion such as `(x==y)` that would enable us to remove an address comparison, is true inside a loop but not on the incoming edges to the loop. The loop can then be peeled just enough to make the assertion true on the incoming edges (without exceeding a code size budget). This can be done on "loops" with multiple entries as well.
- Moving loop invariant conditional branches out of loops. if a test such as `(x==y)` is loop invariant, it can be moved out of the loop by creating two copies of the loop, one which behaves as if `(x==y)` and another which behaves as if `(x!=y)`, and selecting the right loop version to enter at run time, based on the comparison `(x==y)`. The "loop" can have multiple entries. This optimization also helps with the removal of redundant address comparisons.
- Limited tail duplication on time-critical code fragments. Similarly, an assertion that would allow an optimization such as removing an address comparison or copy operation may be true on some incoming edges

but not all. In that case, it makes sense to do just enough tail duplication to make the assertion true on all incoming edges (so that the redundant operation can be eliminated in at least one copy of the code). One can even tail-duplicate cyclic code by treating the cyclic code as an atomic basic block, all subject to the code size budget.

- Partial redundancy elimination.
- Copy propagation/elimination. Constant propagation.

In this section we will apply the compilation procedure suggested here to the same running example, described in Figure 1. Notice that while existing compiler techniques cannot register allocate this example, our technique can, with the resulting code behaving just like the limit register allocator machine.

3.1 Initial code

Figure 10 describes the initial assembly version of the program, with just the above-mentioned optimizations involving "home ncrs" to cut down on the number of address comparisons, and assertion propagation/copy elimination/PRE.

```
rY.addr=NULL; rX.addr=NULL; //initalize the NCR entries to be used
Loop: ...
BC L1
A r3=r3,4
L1: // L r12=(r6) home ncr=rX
If (r6==rX.addr) {} else {rX.data=LOAD(r6); rX.addr=r6;}
// (rX.addr==r6) is true here but not at the loop entry--peeling will help
// L r4=(r3) home ncr=rY
If (r3==rY.addr) {r4=rY.data;} else if (r3==rX.addr) {r4=rX.data;} else { r4=rY.data=LOAD(r3); rY.addr=r3;}
A rX.data=rX.data,r4 //copy prop of LR r12=rX.data;A r12=r12,r4; LR rX.data=r12
// ST r12,(r6) home ncr=rX (code eliminated through copy elim.)
...
BC Loop
// at the end of a procedure simulate flushing all the dirty NCR entries
STORE(rX.addr,rX.data)
```

Figure 10. Initial Code

3.2 Applying optimizations

Figure 11 describes the effects of loop peeling, which makes `(r6==rX.addr)` true throughout the loop, and hence eliminates the `(r6==rX.addr)` comparison.

Figure 12 isolates the frequently executed cycle of the inner loop where the conditional `(r3==rX.addr)` is loop invariant. Figure 13 shows the result of moving the invariant condition out of the loop.

In `Loopy` the load may be executed only on the first iteration, and subsequently the data in `rY.data` will be used (since `(r3==rY.addr)` will be true). To expose this redundancy, loop peeling is appropriate as in Figure 14. Also, additional copy propagation has been done in Figure 14.

After this point, code would be copied from branch targets, to reduce the number of branches being executed.

The aggressive software emulation technique described here, will rely on a cost-performance analysis (based on profiling feedback), and will focus on optimizing the most frequently executed parts of the code, until various budgets are exceeded. Address comparisons created by this technique can be implemented through Huffman-encoded trees that check the most frequently matching addresses first, or with predicated execution features on ILP machines.

```
rY.addr=NULL; /*rX.addr=NULL;*/...  //rX.addr=NULL dead
BC L1a       // peeled partial body of loop
ADD r3=r3,4
//r6==rX.addr false before this statement and true after it
L1a: /*if(r6==rX.addr){} else*/ {rX.data=LOAD(r6); rX.addr=r6;}
B  peelentry0

Loop: ...
BC L1
A r3=r3,4
L1: /*if(r6==rX.addr){} else{...}*/  //r6==rX.addr true here, rX definitely assigned
peelentry0:
if (r3==rY.addr) {r4=rY.data;} else if (r3==rX.addr) {r4=rX.data;}
  else {r4=rY.data=LOAD(r3); rY.addr=r3;}
A rX.data=rX.data,r4 // copy prop. of r12=rX.data;A r12=r12,r4;rX.data=r12;
...
BC Loop
Exit:
STORE(rX.addr,rX.data)
```

Figure 11. Loop peeling makes `(r6==rX.addr)` true inside loop-eliminates address comparison.

```
Loop: ...
BCnot L2
L1:
If (r3==rY.addr) {r4=rY.data;} else if (r3==rX.addr) {r4=rX.data;} else {r4=rY.data=LOAD(r3); rY.addr=r3;}
A rX.data=rX.data,r4
...
BC Loop
B exit

L2:
A r3=r3,4
peelentry0:
B L1
```

Figure 12. Isolating the part of the loop where `(r3==rX.addr)` is invariant

4 Runtime register allocation using hardware

This section outlines a proposal to do register allocation in hardware at runtime. We observe that the operation of the limit machine from Section 2, is very similar to an associative cache structure, when the data is not found immediately in the entry directly given by the `ncr` field. This leads us to propose a level-1 data cache structure that can also be accessed as a register file. The advantages of this approach are as follows: (1) The register replacement policy and cache replacement policy are unified into one piece of hardware (thus benefitting from state-of-the-art cache organizations), and (2) When some loads/stores do use ncr's while some others use a plain L1 data cache, coherence between the two conceptually separate memory hierarchies is automatically achieved. Ordinary loads/stores can thus be mixed with loads/stores that try to make use of an `ncr`.

In our new hardware approach, which attempts to mimic the limit machine described in the introduction, which dynamically rewrites its own register fields, all `L/ST` instructions are extended with a Predicted Register Number (`PRN`) field

The `PRN` field has three subfields: `<set number, way number within set, offset within line>` which supply the information needed to directly locate a load/store operand inside a traditional set-associative cache.

First, the `PRN` is sent to the D-L1 cache. The access is speculatively completed quickly, as if the D-L1 were a register file, and subsequent operations that may depend on the load data are also started speculatively, as soon as possible.

Then the normal address is also sent to the cache, after the normal register access and `TLB` delays have elapsed, and the speculative access is checked for correctness.

```
Loopx: ... // this loop behaves as if (r3==rX.addr) were true
BCnot L2
L1x:
r4=rX.data; //(r3==rY.addr) is false since r3==rX.addr is true and  ncr's have distinct addresses
A rX.data=rX.data,r4
...
BC Loopx
B exit

Loopy: ... // this loop behaves as if (r3==rX.addr) were false
BCnot L2
L1y:
if(r3==rY.addr) {r4=rY.data;} else {r4=rY.data=LOAD(r3); rY.addr=r3;}
// (r3==rY.addr) true inside loop but not on entry-- loop peeling will help
A rX.data=rX.data,r4
...
BC Loopy
B exit

L2:
A r3=r3,4
peelentry0:
//select correct loop to enter
if (r3==rX.addr) {goto L1x;} else {goto L1y;}
```

Figure 13. Moving the invariant condition (`r3==rX.addr`) out of the loop

If the L/ST operand is already in the cache array location denoted by `PRN`, (I.e. the cache line indicated by the set number and way number within set subfields of the `PRN` has a valid tag, which equals the upper bits of the real operand address of the L/ST, and the offset within line subfield of the `PRN` was equal to the offset within line subfield of the L/ST address), then there is nothing to be done, the access was correct.

Otherwise, first, the speculatively started operations that may depend on the L/ST are squashed. The set associative D-L1 cache is accessed as usual, using the load/store real address. If there is a cache miss, the lower level cache(s) are accessed as usual, and an existing line in D-L1 is evicted for replacement (casting it out to L2 if it was dirty/overwritten).

The choice of the line to be evicted can vary, according to the replacement policy.

The load/store instruction is then completed with the correct operand in the D-L1 cache. Also, the correct current location of the load/store operand is written into the `PRN` field of the Load/Store instruction that caused the register number misprediction.

There are two special invalid values of the `PRN` field, which force register number mispredictions

Non-sticky invalid value: All load/stores are initialized to use the non-sticky invalid value when a program is first loaded in memory. When the load/store first executes, it will mispredict. The current location of the operand is then written into the `PRN` field of the load/store.

Sticky invalid value: This `PRN` value also forces mispredictions, but cannot be overwritten. So the load/store will behave like an ordinary load/store that does not use the `PRN` prediction mechanism. Software or hardware algorithms could identify suitable loads that mispredict often. Such loads could be scheduled by a compiler in a in-order issue machine or by the hardware in an out-of-order issue mechanism, by using a longer load-to-use latency.

5 Related Work

The IA-64 ALAT mechanism [1], while originally designed for scheduling speculative loads for instruction level parallelism, can also be used for register promotion of load instructions in the presence of pointer stores. As long as the probability of overlap is very low, load instructions can be speculatively promoted to registers. CRegs [2] is another mechanism designed specifically for register promotion in the presence of pointer accesses. Transmeta has also described a similar technique. However, these techniques all attempt to overcome requirement (2) described

```
Loopx: ... // this loop behaves as if (r3==rX.addr) were true
BCnot L2
L1x:
/*r4=rX.data;*/ //(r3==rY.addr) is false since r3==rX.addr is true and  ncr's have distinct addresses
A rX.data=rX.data,rX.data;//copy prop. of r4=rX.data;A rX.data=rX.data,r4
...
BC Loopx
B exit

L1y': // peeled partial body of Loopy
if(r3==rY.addr) {/*r4=rY.data;*/} else {/*r4=*/rY.data=LOAD(r3); rY.addr=r3;}
// here rY.addr==r3
B peelentry1;

Loopy: ... // this loop behaves as if (r3==rX.addr) were false
BCnot L2
/*r4=rY.data;*/ // since r3==rY.addr
peelentry1:
A rX.data=rX.data,rY.data//copy prop. of r4=rY.data;A rX.data=rX.data,r4
...
BC Loopy
B exit

L2:
A r3=r3,4
peelentry0:
//select correct loop to enter
if (r3==rX.addr) {goto L1x;} else {goto L1y';}
```

Figure 14. Peeling `Loopy` to expose redundancy, copy propagation of `r4`

in the introduction (difficulty of alias analysis at compile time), and not requirement (1) (Dynamically varying load/store operand addresses). To our knowledge this paper's method is the first one to provide a solution for overcoming both (1) and (2). Also for the case of overlapping pointers, our method does not show performance degradation, whereas approaches such as the IA-64 ALAT mechanism suffer significant performance loss.

The ncr fields in the limit register allocation engine described in Figure 5 is similar to way-prediction in associative caches. However, in our case, way prediction information is stored as part of the instruction (the ncr field), not per cache set, as in the typical "MRU" policies for way prediction [8]. Also, because the `NCR` register file is accessed as soon as an instruction is available, a shorter pipeline can be achieved compared to the traditional load/store pipeline. The shorter pipeline can lead to smaller branch misprediction penalties.

6 Conclusions

In this paper, we have analyzed the two fundamental impediments a compiler faces (aliasing, and dynamically varying load/store addresses) when making a register allocation decision, and have designed an ideal "limit register allocator machine," as a theoretical tool to gain insight into the essence of the register allocation problem, and to address both of the fundamental impediments. We have also proposed:

(1) A software emulation method for the limit machine. This leads to successful register allocation of code fragments that could not be handled by other means before.

(2) A hardware implementation of concepts from the limit machine, by adding a dynamically changing operand location prediction field ("Predicted Register Number") to each load/store instruction, which improves the performance of data L1 caches by speculatively accessing the predicted operand location in the D-L1 cache array directly (as if it were a "register file").

Based on the observations in this paper, we can see an interesting continuum of "cache-like" register file structures, ranging from ordinary register files, our proposed technique, and CRegs/ALAT approaches.

Further experimental results will be provided in the final version of this paper. Here are also some further (speculative) variations on the ideas, some of which will be part of future work related to the present paper:

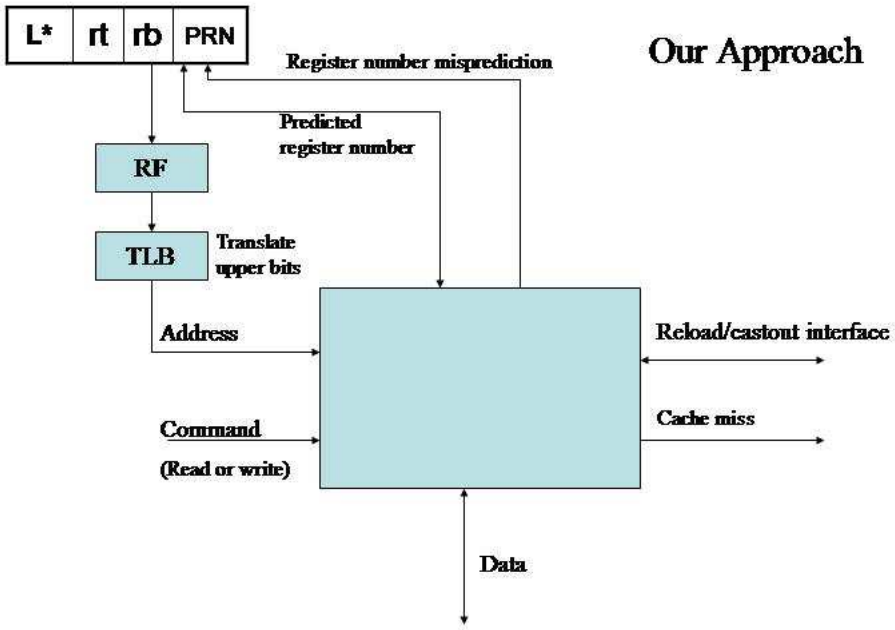

Figure 15.

- Using more than one ncr field per instruction, to increase chances of a match.
- Hiding the details from an existing ISA, by keeping the `PRN` field only the icache hierarchy and not in memory.
- Using `PRN`'s pervasively at all levels of the memory hierarchy, as general purpose location prediction bits, as a way to speed up any cache or memory access.
- Anticipating ncr changes ahead of time (using both simple and sophisticated prediction mechanisms) and prefetching the desired data into the next ncr to be used by an instruction. Doing the ncr field binary rewriting ahead of time.
- Merging N ncr registers that represent N different addresses into one physical ncr, when the contents of the addresses are equal.
- Organizing the `NCR` with long lines. Packing words from noncontiguous but related operands contiguously, in a single long line of the `NCR`.

Acknowledgments

This work has been supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCHC020056. We are grateful to Marc Auslander for observing that the `ncr` field of the limit register allocator of Figure 5 can be stored in an instruction cache, for use as a data-cache way prediction field per instruction.

Appendix: Necessity and sufficiency of conditions (1) and (2)

Definition. *A* state *is a mapping from memory addresses and machine registers to bit-string values.*

Definition. *A code transformation is* correct*, if for all starting states: (a) the transformed code terminates if and only if the original code terminates, and (b) when it terminates, the transformed code produces the same final state as the original code (except possibly for "dead" memory addresses and registers in the final state, whose values will never be referenced before being overwritten, and hence do not matter).*

Theorem. *Assume that register allocation optimizations that take advantage of any special property of a given code fragment (that is not shared by all code fragments) are not to be considered*[7]*. Then, given a single-entry code fragment, a candidate set of load/store instructions in the code fragment, and a memory address* `X` *(a register or constant expression, to be evaluated in the starting state), the canonical register allocation of memory address* `X` *as defined in the Introduction will be a correct code transformation, if and only if, for every starting state, both of the conditions (1) and (2) (defined in the Introduction) hold during the execution of the code fragment.*

Proof. Sufficiency: Suppose that for all starting states, both (1) and (2) hold during execution. Then the transformed version of the code after the canonical register allocation, clearly yields intermediate states identical to the original code after each instruction execution, except that all references to memory address `X` are replaced by references to register `rX`, which always has the same value as the contents of memory address `X` in the original code. If the original code fragment does not terminate for this starting state, neither does the transformed version. Otherwise, `rX` is stored back into `X` at the end of the code fragment in the transformed version, hence, the final state of the general registers and memory in the original and transformed code are identical. The only potentially different register in the transformed code, `rX`, is dead at the final state. Hence, the transformation is correct.

Necessity: Assume the contrary. Suppose we have performed canonical register allocation for a candidate set and memory address `X`, and the code transformation is correct, but there is a starting state which leads to an execution which violates either (1) or (2).

If condition (1) does not hold, a member of the candidate set is referring to a different address `Y` instead of `X` during execution. This will still yield a correct final result only if (a) `X` and `Y` have equal contents, or if (b) the incorrect intermediate state in the transformed program, still leads to the same final result as in the original program (or the same non-terminating behavior as in the original program). But, this means that the optimization has relied on a special property of the given code fragment (a property that is not shared by all code fragments). Contradiction.

If condition (2) does not hold, then a load or store instruction outside of the candidate set but inside the code fragment refers to `X` during execution. Then, either (a) the contents of memory address `X` and the value of register `rX` are identical at the instant after executing the offending load or store, or (b) the incorrect intermediate state in the transformed program, still leads to the same final result as in the original program (or the same non-terminating behavior as in the original program). But this means the optimization has relied on a special property of the code fragment (a property that is not shared by all code fragments). Contradiction. □

References

[1] Jin Lin, Tong Chen, Wei Hsu, Pen-Chung Yew, Roy Ju. "A Compiler Framework for Speculative Analysis and Optimizations" in Proceedings of the SIGPLAN'03 Conference on Programming Language Design and Implementation, San Diego, June, 2003.

[2] Peter Dahl and Matthew O'Keefe. Reducing memory traffic with CRegs. Proceedings of the 27th IEEE/ACM annual international symposium on Microarchitecture. pp. 100-104, 1994 .

[3] G. J. Chaitin et. al,"Register allocation via coloring", in Computer Languages, 6:47-57, 1981

[4] Fred C. Chow, John L. Hennessy. The priority-based coloring approach to register allocation, ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 12, Issue 4 (October 1990), pp. 501-536.

[5] Massimiliano Poletto, Vivek Sarkar. Linear Scan Register Allocation (1999). ACM Transactions on Programming Languages and Systems, Volume 21, Issue 5, pp. 895-913.

[7] This "rule of the code transformation game" makes our approach suboptimal, but it provides great conceptual economy by avoiding special cases

[6] Kemal Ebcioglu, Randy D. Groves, Ki-Chang Kim, Gabriel M. Silberman, Isaac Ziv. VLIW compilation techniques in a superscalar environment, Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation, Orlando, Florida, pp. 36-48, 1994.

[7] Rakesh Ghiya and Laurie J. Hendren. "Putting Pointer Analysis to Work," in Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,, San Diego, California, pp. 121-133, January 1998.

[8] Michael D. Powell, Amit Agarwal, T. N. Vijaykumar, Babak Falsafi, Kaushik Roy. Reducing set-associative cache energy via way-prediction and selective direct-mapping. Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture. pp. 54-65, 2001.

[9] Kemal Ebcioğlu. Some Design Ideas for a VLIW Architecture for Sequential Natured Software. in Proceedings of IFIP WG 10.3 Working Conference on Parallel Processing, pp. 3-21, M. Cosnard et al. (eds.), North Holland, 1988.