The Store-Load Address Table and Speculative Register Promotion

Matthew Postiff, David Greene and Trevor Mudge
Advanced Computer Architecture Laboratory, University of Michigan
1301 Beal Ave., Ann Arbor, MI 48109-2122
{postiffm, greened, tnm}@eecs.umich.edu

Abstract

Register promotion is an optimization that allocates a value to a register for a region of its lifetime where it is provably not aliased. Conventional compiler analysis cannot always prove that a value is free of aliases, and thus promotion cannot always be applied. This paper proposes a new hardware structure, the store-load address table (SLAT), which watches both load and store instructions to see if they conflict with entries loaded into the SLAT by explicit software mapping instructions. One use of the SLAT is to allow values to be promoted to registers when they cannot be proven to be promotable by conventional compiler analysis. We call this new optimization speculative register promotion. Using this technique, a value can be promoted to a register and aliased loads and stores to that value's home memory location are caught and the proper fixup is performed. This paper will: a) describe the SLAT hardware and software; b) demonstrate that conventional register promotion is often inhibited by static compiler analysis; c) describe the speculative register promotion optimization; and d) quantify the performance increases possible when a SLAT is used. Our results show that for certain benchmarks, up to 35% of loads and 15% of stores can potentially be eliminated by using the SLAT.

1. Introduction

Register allocation is an important compiler optimization for high-performance computing. Access to data stored in machine registers avoids using the memory subsystem, which is generally much slower than the processor. Register promotion allows scalar values to be allocated to registers for regions of their lifetime where the compiler can prove that there are no aliases for the value [3, 4, 5]. The value is *promoted* to a register for that region by a load instruction at the top of the region. When the region is finished, the value is *demoted* back to memory. The region can be either a loop or a function body in this work, though promotion can be performed on any program region. The benefit is that the value is loaded once at the start of the region and stored once at the end, and all other accesses to it during the region are from a register allocated to the value by the compiler.

Unfortunately, imprecise aliasing information and separate compilation conspire to limit the types and amount of data that can be safely allocated to registers. To allow a relaxation of the compiler's conservative nature, we introduce the *store-load address table (SLAT)* and investigate its use in enabling more effective register allocation. We also introduce a new compiler transformation called *speculative register promotion*, which makes use of the SLAT, and evaluate the performance gains it can provide.

The SLAT and speculative register promotion introduce several new opportunities for register allocation. Figure 1 shows the combinations that we consider in this paper. Figure 1(a) is conventional register allocation as done by most compilers. Figure 1(b) shows the result of register promotion, which requires more sophisticated compiler alias analysis. (Throughout the paper we use the term *alias* somewhat loosely to include all possible references to data though mechanisms other than its primary name, including ambiguous pointers and side-effects.) Figure 1(c) requires further compiler support because in order to prove that the global can be allocated to a register for its entire lifetime requires that the whole program be analyzed at once. This allows the compiler to make the determination that the variable `global` is only ever used through its name, and never through a pointer. Previous work has examined this optimization [17, 21].

Figure 1(d) shows another example using default register allocation. This time the loop contains a function call, which means that conventional promotion (with separate compilation of functions) cannot be sure that `foo()` does not access the `global` variable. Thus `global` cannot be promoted to a register. Figure 1(e) shows how the SLAT allows promotion to occur anyway. The compiler promotes `global` as in normal register promotion but uses special opcodes to inform the hardware that the promotion is speculative. Finally, link-time global allocation can be done even under separate compilation when the SLAT is used to protect the `global` variable. In this case, the mapping operation occurs at the start of the program– say at the top of `main()`–and is not shown in the figure. Table 1 gives a summary of these allocation strategies.

The remainder of this paper is organized as follows. Section 2 describes the logical organization of the SLAT. Section 3 introduces the speculative register promotion transformation. In Section 4 we describe our experimental setup, while

```
while () {
  ld r5, global
  add r5, r5, 1
  st global, r5
}
```

(a) Original source

```
ld r5, global
while () {
  add r5, r5, 1
}
st global, r5
```

(b) Register promotion

```
while () {
  add r32, r32, 1
}
```

(c) Link-time global allocation

```
while () {
  ld r5, global
  add r5, r5, 1
  st global, r5
  foo();
}
```

(d) Original source

```
map r5, global
while () {
  add r5, r5, 1
  foo();
}
unmap r5, global
```

(e) SLAT-based promotion

```
while () {
  add r32, r32, 1
  foo();
}
```

(f) SLAT-based link-time global allocation

Figure 1. The results of using different register allocation strategies. (a) The original source code, in a combination of C and assembler notation. It uses the default strategy for allocation, which does not allocate the global to a register. (b) Register promotion moves the load and store outside of the loop. (c) After application of link-time global variable allocation, each occurrence of global is replaced with r32 and unnecessary copies are removed. (d) Another snippet of source code, which includes a function call, rendering the global not promotable by conventional means. (e) The SLAT allows the promotion to occur in spite of the function call. (f) Link-time global variable allocation can also be performed with help from the SLAT even when separate compilation is used.

our experimental results are analyzed in Section 5. Section 6 describes previous work in the areas of memory disambiguation and register allocation. Finally, we discuss our conclusions and directions for future work in Section 7.

2. The Store-Load Address Table (SLAT)

The store-load address table (SLAT) is a hardware structure that allows the compiler to relax some of the conservative assumptions made due to imprecise analysis of memory communication. Logically, the SLAT is a table where each entry contains a logical register number, memory address and some information flags for bookkeeping. Speculative register promotion uses the SLAT to associate a memory address with a register. All references to this address will be forwarded to the register file as long as the address is mapped in the SLAT. Thus, the SLAT is indexed associatively by address.

Special machine instructions are used by the compiler to manage the SLAT. To initialize a speculative promotion, a special map instruction is used. This instruction includes a memory address and a register number. A SLAT entry is created, indicating that the data at the given memory address resides in the given register. A load from memory is also executed to place the desired data in the register. Likewise, an unmap instruction removes an association from the SLAT, sending the data in the register to the memory. The map and unmap operations are essentially just special load and store operations.

After a map instruction has associated a memory address with a register, every subsequent memory operation examines the SLAT, comparing its address operand with those in the

SLAT. When a match (conflict) is detected in the SLAT, the memory operation is redirected to the register file. A load retrieves its value from the SLAT-mapped register instead of from memory; a store uses the mapped register as its destination instead of memory. An unmap instruction at the bottom of the promotion region handles storing the updated register out to memory.

Since the SLAT allows register allocation of potentially aliased variables (including globals that may be used by callee functions) whose scopes may exceed that of a single function, special handling is necessary to close the "gap" between function-scoped machine registers and registers containing mapped data. One example of this problem occurs at function call boundaries. On entry to the callee, all callee-save registers used by the function are first spilled to the stack to preserve existing values for the caller. These registers are restored upon function exit. If one of these callee-save registers is mapped in by the SLAT, the spill instruction must be dynamically modified to store the data to the "home" memory location of the data (the global storage or stack location for an aliased local variable). This home address is available in the SLAT entry for the register being spilled. A reload operation likewise must be modified to load from the home location. These operations require two new memory instructions: spill and reload. These are store and load instructions with special opcodes to indicate their function (saving and restoring of callee-save registers). These instructions must examine the SLAT to see if the referenced register is mapped. Thus the SLAT is also indexed directly by register number. We classify such registers as *callee-update*, analogous to callee-save, because their val-

2

Allocation Strategy	What is Allocated	Region in Register File	Is Whole-Program Information Used?
(a, d) Default	Unaliased local scalars including compiler temporaries.	Local	No.
(b) Register Promotion	Aliased local scalars or global scalars aliased or not. In either case, they are promoted for regions where they are provably unaliased.	Local	Can be used to enhance alias analysis so that extra candidates can be proven safe to promote.
(c) Link-time global allocation	Unaliased global scalars.	Global	Required.
(e) SLAT-based promotion	Aliased local scalars or global scalars. SLAT allows allocation even in aliased regions.	Mappable	Can be used to reduce number of SLAT promotions necessary.
(f) SLAT-based link-time global allocation	Aliased and unaliased global scalars.	Mappable	Can be used to reduce number of SLAT promotions necessary.

Table 1. Various strategies for allocating registers. In our usage, "aliased" means that the variable's address has been taken somewhere in the program or it could be referenced through a function-call side effect. The register file regions are conceptual divisions of the registers into groups based on their function. The "local" region of the register file is the region used for local variables in the function. The global region contains global variables for their entire lifetime. The mappable region contains mapped (speculatively promoted variables). In our experiments, the local and mappable regions are the same. The letters in column 1 correspond to the labels in Figure 1.

ues are automatically updated by any memory accesses in the callee function.

Because the reload instruction must have access to the home memory address for the data, the processor must keep every SLAT entry that is created until an unmap deallocates it. Moreover, an address can be mapped to multiple registers or a single register can be re-mapped to a new address. These cases are simplified by the fact that only one mapping is active for a particular function. The compiler can guarantee that no address or register is mapped twice in the same region. It can do this because it only speculatively promotes directly-named scalar variables.

There are several strategies for dealing with these situations. One possibility is to have a large SLAT with a hardware-controlled overflow spill mechanism, similar to that used in the C-machine stack cache [24]. Another possibility is to require compiler management of the SLAT. Instructions to save and restore SLAT entries can be generated in the same way instructions to save and restore callee-save registers are generated. Our simulations assume an infinite-sized SLAT so that we may evaluate its performance potential.

In addition to callee-save spills and reloads, spill and reload operations are necessary to deal with excessive register pressure within a function. Speculative register promotion can increase the amount of this spilling. Since the spilling effectively negates the benefit of register promotion the compiler may simply reverse the promotion if spilling occurs. Memory access size and overlap must also be considered in the SLAT; the compiler can restrict promotions to ease this problem.

3. Speculative Register Promotion Using the SLAT

This section outlines how the SLAT can be used to allow speculative register promotion. Preliminary exploration into the limitations on static register promotion indicated that a significant number of memory operations cannot be promoted due to ambiguous or unseen memory accesses through function calls. This will be quantified later in the paper. To address this problem, we consider a new optimization called *speculative register promotion* which uses the SLAT to allow promotions in these situations. It does this by providing a fallback mechanism in the case that the promotion was too aggressive, i.e. that there was a conflict where the promoted value was not synchronized with its value in memory. When this occurs, the hardware can provide the current value.

As we saw in Section 2, the SLAT is tailored to solve this problem because the hardware compares each load and store address against those stored in the SLAT. Once a value is promoted to a register with a map instruction, it can be used or defined several times before a conflicting memory load appears. Since the value in memory could be out of date with respect to the value promoted to the register, both load and store operations have to be examined to see if they are attempting to access the value that was promoted to a register.

The register promoter in our C compiler, MIRV, can promote global scalar variables, aliased local scalar variables, large constants, indirect pointer references (we call these *dereferences*), and direct and indirect structure references. It

Instruction	Action
map reg, addr	Add an entry to the SLAT. If there is a pre-existing mapping for the address in the SLAT, the data is forwarded from the previous register to the register currently being mapped. Otherwise, the data is loaded from memory.
unmap reg, addr	Remove an entry from the SLAT. If there is a previously mapped but unspilled entry, store the data from reg to the previously mapped register.
spill	If the register contains a value that was placed there by a previous map instruction, spill the value to the mapped address (home location) instead of the address specified to the stack spill location.
reload	If the previous SLAT on the SLAT stack has a mapping for this register, reload the value from its mapped address. Otherwise, reload from the specified location on the stack.
load	If any entry in the SLAT stack maps the load address, and has not been spilled, then copy from the mapped register to the load's destination register. Increment slatLoadConflicts.
store	If any entry in the SLAT stack maps the store address, and has not been spilled, then copy from the store source register to the register indicated in the SLAT entry. This implements the "callee update" register convention (a modification of "callee save". Increment slatStoreConflicts.
call	Push a new SLAT onto the SLAT stack.
return	Pop current SLAT from SLAT stack.

Table 2. Description of the actions that take place at various points in the SLAT simulator.

can do so over loops or whole functions. The algorithm is described in detail elsewhere [17]. Speculative register promotion was a simple augmentation to the existing promoter. Any directly-named value (global or local) which is not promoted because of aliases can be promoted speculatively (based on simple selection heuristics). This is accomplished by emitting a promoting load (map) and demoting store (unmap) at the boundaries of the region, with additional information indicating these are speculative promotion operations. The backend of the compiler passes this through via annotation bits in the instruction encoding and the simulator treats the map/unmap operation as described in Table 2. Since global and aliased data can reside in registers, the compiler was also restricted from certain kinds of code motion around those accesses.

4. Experimental Setup

All the benchmarks used in this study were compiled with the MIRV C compiler. The compiler takes a list of optimizations to run on the code as well as the number of registers that are available on the architecture. We ran variants of the SPEC training inputs in order to keep simulation time reasonable. Our baseline timing simulator is the default sim-outorder configuration. A description of MIRV, our compilation methodology, and benchmark inputs is presented in the technical report of [12].

All simulations were done using the SimpleScalar 3.0/ PISA simulation toolset [10]. We have modified the toolset (simulators, assembler, and disassembler) to support up to 256

registers. Registers 0-31 are used as defined in the MIPS System V ABI [11] in order to maintain compatibility with pre-compiled libraries. Registers 32-255 are used either as additional registers for global variables or additional registers for local caller/callee save variables.

A modified version of sim-profile was used to simulate the behavior of a program compiled to use the SLAT. The simulator implements an infinite-sized SLAT with ideal replacement. Table 2 shows the actions that are taken at various instructions in the program. While the simulator is idealized and is not particular to an implementation, it allows us to see the potential benefits of the SLAT. Later work will address specific implementation issues.

5. Experimental Evaluation

This section presents our experimental results. Section 5.1 discusses the performance improvements possible with conventional register promotion and shows how it is limited in its applicability. Section 5.2 shows the performance improvement that can be obtained when values can be promoted speculatively.

5.1. Register Promotion

Previous work showed the performance of basic register promotion in the MIRV compiler [17]. That work found that register promotion improves performance from 5% to 15% on some benchmarks. Other benchmarks perform worse with reg-
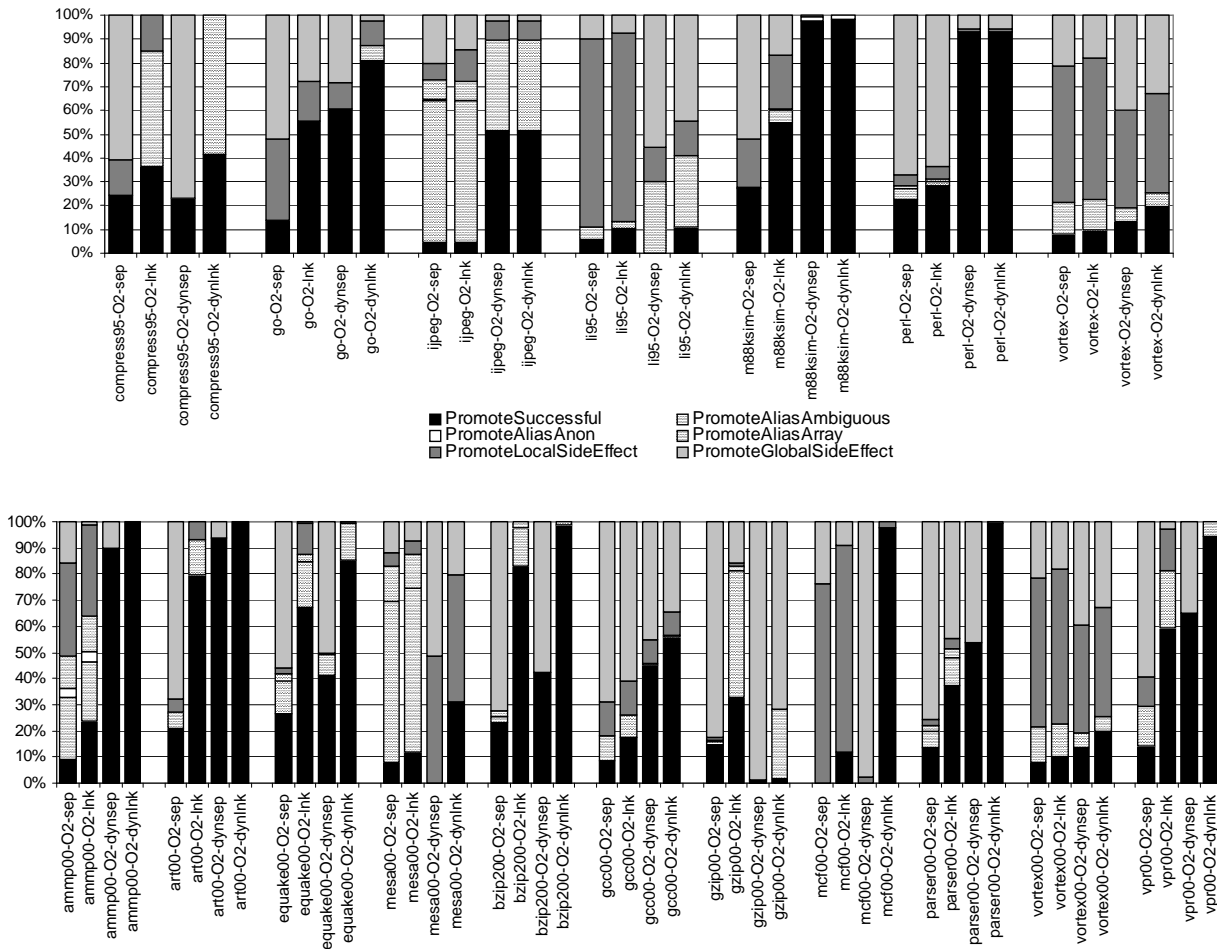
Figure 2. The reasons that scalar candidates could not be promoted. The bars shows the breakdown of reasons that promotion could not occur. All compilations use -O2 optimization. The first bar is separate compilation of modules with no interprocedural alias analysis. The second bar has interprocedural side-effect analysis information annotated at each function call site for improved alias analysis precision. This increased information in turn increases the number of candidates that are provably safe to promote. The percentages shown for these first two bars are percentages of scalar promotion candidates. The third and fourth bars are analogous except that they estimate loss in performance by weighting the counts by dynamic frequency of access to the candidate variables. The legend is explained in Table 3. These numbers are based on compile-time estimates and unlike later figures are only indicative of trends.

ister promotion. This is due to extra register pressure caused by the promotion, which introduces spilling code.

The somewhat lackluster results for many benchmarks led us to evaluate the reasons why promotion is not performing well. The graph in Figure 2 shows statistics kept by the compiler which demonstrate that promotion is often limited by aliasing and side-effects. The figure shows each benchmark (along the X axis) in four different configurations. The first configuration is -O2 with separate compilation of the program's files. The compiler produces the least detailed alias information in this case. The second configuration is similar except that a simple interprocedural side-effect analysis is used to improve the precision of alias analysis at function call sites. This increases the precision of the alias analysis and allows the compiler to determine that more values are safe to

promote. For these two bars, the percentages indicate the number of *static* references that fall into each category.

The third and fourth bars are similar to the first and second except that they estimate the effect of the un-promoted values by weighting each value by the number of load and store executions that would have been saved in a training run of the benchmark if the value had been promoted. Thus the percentages on the Y-axis change meaning for the third and fourth bar, because they indicate the estimated percentage of *dynamic* references that fall into each category.

The bars are divided into portions showing the reason that a promotion could not occur. The legend of the graphs are explained in Table 3. The last two categories–local and global side effects–are of interest in this paper because the SLAT can aid the compiler in promoting those references to registers.

Legend Entry	Explanation
Successful	The transformation was not restricted.
AliasAmbiguous	A possible manipulation of some data through a pointer prevented the transformation. In other words, there is a pointer that *might* point to something the restricts the transformation, but the compiler does not know for sure.
AliasAnon	A manipulation of code involving dynamically allocated memory was restricted by some other possible manipulation of dynamic memory.
AliasArray	A transformation involving an array was restricted by some other use of the array. Because MIRV does not track individual array elements, any reference to an array element is considered to reference the entire array.
GlobalSideEffect	A manipulation of code involving a global variable was prevented by a function call. Usually this is because a function is assumed to define and use all global variables when it is called. If, however, the compiler is performing whole-program analysis, this means that the called function references the global variable somewhere in its body, or in the body of some function further down the call chain.
LocalSideEffect	A possible manipulation of some data through a pointer passed to a function prevented the transformation. In other words, a pointer argument to the call might point to a local variable. The compiler must assume that variable is both used and defined by the call, preventing transformations across the call site.

Table 3. An explanation of the legend in Figure 2.

For example, for the compress95-sep bar, about 25% of static references were promoted. About 15% of values were not promotable because of local side effects, and about 60% of values were not promoted because of a global side effect. Local and global side effects are due to function call sites within the promotion region.

Overall, it is evident that of all promotion candidates, only 20% to 30% of potential promotions are actually performed. Some outliers, such as `li` have almost no promotable values, and some have a large portion of candidates that are promotable. More advanced alias analysis (shown in the second bar of each group) increases the number of promotion successes by 20% in many cases. Still, 20% to 50% of promotion candidates are not promotable using the side-effect analysis.

For the non-promotable candidates, the primary reason is side effects due to function calls (both local and global side effects). This implies that any mechanism that allows promotion in such regions will have to handle call sites very well.

The dynamic estimates in the third and fourth bars of the graphs in Figure 2 show slightly different results because the frequency count of loop bodies is taken into consideration when weighting the effect of a successful or missed promotion. The results are very benchmark dependent, sometimes showing that the static estimate was good, as is the case with `vortex` and `art`. In many cases the dynamic estimate shows that the missed promotion opportunities were not significant factors in performance.

There are a significant number of promotion opportunities that are missed because of poor alias analysis. This obser-

vation led us to develop speculative register promotion, which is evaluated below.

5.2. Speculative Register Promotion using the SLAT

The main shortcoming of register promotion is the number of cases where promotion cannot happen because of aliasing. In this section, we evaluate the performance benefits possible from allowing more promotion via the SLAT.

5.2.1. Loop and Function Promotion. Table 4 shows the improvements possible with the SLAT. The first two numeric columns show improvement possible on top of MIRV at -O2 optimization, which includes only loop-level promotion. The numbers were collected by modifying the register promoter in MIRV to annotate the candidates that could not be promoted. Each such candidate variable reference (load or store) was annotated and each occurrence was counted during the simulation. The numbers in the table are the percentage of all load and store instructions that were thus annotated, meaning that if we had "perfect" register promotion, all of these loads and stores would have been transformed by the compiler into register references. Note that these percentages are different than shown in Figure 2 because those percentages are only of promotion candidates, not *all* load and store operations. One other caveat with regard to these numbers is that they are overly conservative because they count store operations that may not be necessary because the promoted variable is not actually defined in the promotion region. Therefore, several of the

Category	Benchmark	mirvcc -O2		mirvcc -O2 with function level promotion	
		Reduction in Loads %	Reduction in Stores %	Reduction in Loads %	Reduction in Stores %
SPECint95	compress	18.9	12.8	36.6	14.2
	gcc	1.3	-2.7	1.6	-5.5
	go	1.3	-1.8	1.9	-5.2
	ijpeg	0.3	-0.4	0.3	-0.4
	li	6.5	2.2	8.1	2.6
	m88ksim	0.8	0.0	3.8	-0.2
	perl	0.0	0.0	1.5	-0.1
	vortex	-1.7	-3.1	-1.1	-5.8
SPECfp2000	ammp	4.6	-0.1	4.7	-0.1
	art	13.6	12.2	13.6	12.2
	equake	4.6	-0.1	4.7	-0.1
	mesa	0.5	0.0	0.5	0.0
SPECint2000	bzip	5.3	-0.4	7.3	-1.4
	gcc	2.0	-2.5	2.3	-5.0
	gzip	24.2	12.2	31.4	18.1
	mcf	6.8	1.2	6.9	1.2
	parser	14.0	-0.5	16.9	0.5
	vortex	-1.7	-3.1	-1.1	-5.8
	vpr	7.8	-4.4	13.2	-6.3

Table 4. Reductions in dynamic loads and stores possible for missed promotion candidates with the SLAT. The baseline in columns 3 and 4 is compiled with loop-level register promotion The baseline in columns 5 and 6 is compiled with loop- and then with function-level promotion. The percentages give the number of loads and stores that could be removed if the promotion could take advantage of the SLAT.

"improvements" in store instruction counts are actually negative, indicating that more stores were counted after the optimization than before. The actual performance will be better than these numbers show.

Even with those caveats, the compress, art, gzip, parser, and vpr benchmarks all exhibit significant potential for improvement for both load and store instructions–with 10% to 20% reductions possible in several cases. This substantiates our earlier conclusion that conventional promotion is unable to take advantage of many opportunities. The other results are not very significant, which is not a surprise since this optimization is very dependent on the benchmark.

The third and fourth numeric columns show what happens when loops *and* functions are considered as regions. If a variable can be promoted in a loop, it is done first. Then, if the variable is still profitably promotable over the whole function body, this transformation is made. The result is that function-promoted variables are loaded once at the top of the function and stored once before the function exits, and all other references are to a register instead of to memory. Function-level promotion increases the number of candidate loads and stores for the promoter to examine and we see a corresponding increase in the number of loads and stores that could have been

eliminated with speculative promotion, but that were not removed because of aliasing problems. In this case, what has happened is that the pool of promotion candidates has been enlarged by examining the whole function body, but very few of those additional candidates are actually promoted. We verified this by comparing the overall performance of function-level promotion with the base -O2 configuration. There was not any significant difference (less than 1% for all benchmarks). This indicates that while function-level promotion found more candidates it wanted to promote, it could not promote most of them due to aliasing concerns. The SLAT is effective in allowing these promotions to occur.

5.2.2. Whole-Program Global Variable Promotion.
Previous work demonstrates that link-time allocation of global variables to registers is an important performance optimization [17, 21]. The previous work has only considered "un-aliased" global variables, i.e. those whose addresses are not taken anywhere in the program. The SLAT could further improve the performance of link-time global variable allocation by allowing global variables whose addresses are taken to reside in registers for their entire lifetime. If an enregistered

7

global variable is accessed through a pointer, the SLAT will correctly redirect the memory operation to the register file.

Experiments showed that most benchmarks are not generally improved by such a scheme. This result indicates that most global variables (or at least the important ones) do not have their address taken. This is intuitive, since the global variables are directly accessible and thus need not be used through a level of indirection. This is still promising for the SLAT, however, in a separate compilation environment. In such an environment, the compiler cannot determine which globals are aliased and which are not because modules are not visible as in our link-time, whole-program allocation scheme. Therefore, the SLAT can allow us to approach the good performance of link-time global variable allocation (as in [17]) without needing to compile the whole program as a single unit.

5.2.3. SLAT Size Considerations. The next question we examine is how many entries the SLAT needs to achieve the performance improvements above. The simulator keeps track of the current number of SLAT entries in use and also tracks the high water mark of this number, which indicates the most SLAT entries that would ever be in use concurrently. The high water mark results are presented in Table 5. Except for `li` and `vortex`, none of the benchmarks require more than 50 SLAT entries to speculatively promote all aliased variables. These two benchmarks are exceptional because of their deep function call chains (`li` is a recursive descent program). Most benchmarks require less than 30 entries. This indicates that the SLAT should be effective while still very small in size. This is important since the SLAT must be fully associative. As described in the caption of the table, the third column is for loop-based register promotion, while the fourth column adds function-level promotion to the normal loop based promotion. Function-level promotion produces more candidates in the function bodies and, as we found earlier, not many of those are promotable because of alias problems. Thus the number of SLAT entries required to accommodate function level promotion is higher than for loop-level promotion–by a large margin in some benchmarks.

These numbers double-count any overlap that occurs because a variable gets allocated to the SLAT more than once. This can happen for global variables promoted in two different functions which are active at the same time on the procedure call stack. Overlap can also happen if a variable is promoted over a loop region and then the function promoter decides to promote it over the whole function body. If we corrected for this effect, the values in the graph would be even lower, meaning that an even smaller SLAT will provide the benefits we seek from speculative register promotion.

We also tracked the variation in the required size of the SLAT over the benchmark run. The resulting distribution (not included in this paper) showed that 90% of the *instructions* were executed under conditions requiring about 1/2 to 3/4 of the maximum number of SLAT entries to capture most of the benchmark's execution. This gives a tighter bound on the required size of the SLAT, although it does still count duplicates.

At this point it may be questioned why the SLAT would ever need more entries than there are architected registers. This is a valid question because at most only one aliased variable can be allocated to a given register at any given time, so the most active SLAT entries would be equal to the number of registers. However, at any given time, there are more values alive than there are registers because there are multiple functions "alive" on the procedure linkage stack. Each function could have promoted several values. While these values are not in the registers (they have been spilled out by the calling convention) they are nonetheless active in the sense that they will be coming back into registers when the procedure stack unwinds as functions are completed. Some sort of SLAT management (similar to callee/caller save registers) would allow the SLAT to be limited in size but we do not consider that in detail in this work.

Category	Benchmark	SLAT Entries Actually Required	
		-O2	-O2 with function promotion
SPECint95	compress	7	19
	ijpeg	23	27
	m88ksim	11	37
	perl	10	26
SPECfp2000	ammp	2	11
	art	11	19
	equake	16	36
	mesa	4	5
SPECint2000	bzip	23	32
	gzip	11	25
	mcf	5	7
	parser	26	44
	vpr	10	16

Table 5. Summary of SLAT utilization for select benchmarks. The third and fourth columns show the maximum number of SLAT entries ever used concurrently in the benchmark, not accounting for duplicates. The third column (mirvcc -O2) is for register promotion over loop bodies. The fourth column adds promotion over whole function bodies.

5.2.4. Other Considerations. When a load or store finds that its operand is mapped by the SLAT, a conflict has occurred and fixup needs to be performed to retrieve the latest value (on a load) or update the mapped register (on a store). Our simulations showed that for `compress`, `gzip`, and `parser`, this happened roughly 2%, 3%, and 5% of memory operations. The rest of the benchmarks were well under 1%.

6. Background and Related Work

This section reports on a number of proposals that combine software and hardware approaches to disambiguation, allocation, and scheduling.

Several previous proposals have discussed methods to allow register allocation for aliased variables. CRegs solves the aliasing problem by associating address tags with each register [18, 19, 20]. These tags are checked against loads and stores to keep the registers and memory consistent. On a store, an associative lookup must update all copies of the data in the CReg array. Variable forwarding was proposed as an improvement to CRegs [22]. This technique allows the elimination of compiler alias analysis, simplifying the software side of the problem but complicating the hardware because a value can be mapped to any registers in the register file. Chiueh proposed an improvement on both CRegs and variable forwarding [23]. Aliased data items are kept in the memory hierarchy (data cache) and accessed indirectly through registers. The registers contain the address of the value and the compiler specifies a bit on each operand in the instruction to direct the hardware to use that register indirectly.

The weakness of CRegs is that writes must associatively update several registers. The SLAT does not require this associative write-update to the register file because the compiler guarantees that only one copy of the data is mapped to a register within a function. This vastly simplifies register access compared to CRegs.

Nicolau proposed a purely software disambiguation technique where a load could be scheduled ahead of potentially dependent stores [2]. This technique is called runtime disambiguation because the hardware checks conditions at runtime to determine if a conflict has occurred.

The Memory Conflict Buffer (MCB) is designed as an extension of Nicolau's runtime disambiguation. It allows the compiler to avoid emitting explicit (software) checks of address operands [8, 9]. Instead, addresses that need to be protected are communicated to the hardware by special load operations and then special check operations ask the hardware whether a conflict has occurred for the given address. Hardware does the address comparisons instead of software. Like for runtime disambiguation, the goal is to perform code scheduling in the presence of ambiguous memory operations.

The SLAT is different from the MCB in that it must retain information across function calls to be effective–as was shown, this is important because many aliases are due to assumed side effects of function calls, so that SLAT must handle function calls elegantly. The information stored in the MCB is not valid across function calls [8].

The IA64 architecture provides hardware support for compiler-directed data speculation through use of an Advanced Load Address Table (ALAT) [12]. It allows static scheduling of loads and dependent instructions above potentially aliased stores. The compiler is responsible for emitting check and fixup code for the (hopefully rare) event that a conflict occurs.

The SLAT is different than the ALAT in a number of respects. The most notable difference is that the hardware must compare not only store addresses to all SLAT entries (as with the ALAT), but in addition it must compare all load addresses as well. This is because the most current value for the memory location could be housed in a register and any loads that access that memory location need to receive the current value. The ALAT cannot provide this functionality because the hardware only checks the addresses of store instructions with the entries in the ALAT.

Another difference is that the SLAT must retain all the information ever entered into it whereas ALAT entries can be replaced because of overflow, conflicts, or context switches. This is because the ALAT requires an explicit check instruction to determine if the fixup code needs to be run. If an entry is missing from the ALAT, the check instruction runs the fixup code. Thus the ALAT is "safe" even when it loses information. On the other hand, if the SLAT "loses" an entry, load and store instructions could be executed without detection of conflicts, which would produce incorrect program output.

Another difference between the SLAT and ALAT is that SLAT fixup is not initiated at the point of transformation but at the point where the conflict occurs. For the ALAT, fixup is always initiated at the point of the original load (which has been converted to a check load). For the SLAT, since the correct data is in a register, the hardware can forward the data for a load from the register or for a store to the register.

Transmeta Corporation recently introduced a line of processors that is designed to run unmodified x86 programs using dynamic binary translation [1, 6]. Capability similar to the ALAT is provided by special hardware and instructions to allow load and store reordering. Two instructions are necessary for this: `load-and-protect (ldp)` and `store-under-alias-mask (stam)`. The `ldp` instruction "protects" a memory region. The `stam` instruction then checks if it would store to a previously protected region. If it would, it traps so that fixup can be performed. The main purpose of this system is to allow Transmeta's code morphing software to allocate stack variables to host registers.

The SLAT differs from this approach in that it is designed for a static compilation environment, hardware corrects conflicts instead of taking an exception, and memory does not necessarily need to be kept up to date since the latest value is in the register.

7. Conclusions

This paper has described the design of the store-load address table, its use in a new optimization we call speculative register promotion, and the reductions in load and store operations possible when using this optimization. We began by showing that register promotion was often limited by compiler alias analysis. The number of loads and stores can be significantly reduced for several of the benchmarks with the addition of a SLAT and speculative register promotion–up to 35% reduction in loads and 15% reduction in stores. Applying the SLAT to link-time global variable allocation does not produce much benefit for most benchmarks. It is more important in this case to note that the SLAT effectively allows link-time allocation even in the face of separate compilation, so that the SLAT

can achieve most or all of the benefit of link-time allocation while doing so in a separate compilation environment. Finally, we showed that the SLAT can be modestly sized and achieve the benefits reported here.

There are several important avenues of future work. In addition to providing more detailed performance numbers, we will investigate strategies for determining when the compiler should use the SLAT. We will also address specific SLAT hardware implementation issues as well as compiler management of the SLAT storage. Future work will also include investigating other ways the hardware can help the compiler do aggressive, potentially unsafe operations.

Acknowledgments

References

[1] Alexander Klaiber. The Technology Behind Crusoe^{TM} Processors. Transmeta Corporation. January 2000.

[2] Alexandru Nicolau. Run-Time Disambiguation: Comping with Statically Unpredictable Dependencies. IEEE Transactions Computers, Vol. 38 No. 5, pp. 663-678. May, 1989.

[3] Keith Cooper and John Lu. Register Promotion in C Programs. Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI-97), pp. 308-319, June, 1997.

[4] A. V. S. Sastry and Roy D. C. Ju. A New Algorithm for Scalar Register Promotion Based on SSA Form. Proc. ACM SIGPLAN'98 Conf. Programming Language Design and Implementation (PLDI), pp. 15-25, , 1998.

[5] Raymond Lo, Fred Chow, Robert Kennedy, Shin-Ming Liu and Peng Tu. Register Promotion by Sparse Partial Redundancy Elimination of Loads and Stores. Proc. ACM SIGPLAN'98 Conf. Programming Language Design and Implementation (PLDI), pp. 26-37, , 1998.

[6] Malcom J. Wing and Edmund J. Kelly, Transmeta Corporation. Method and apparatus for aliasing memory data in an advanced microprocessor. United States Patent 5926832. http://www.patents.ibm.com.

[7] David Bernstein, Martin E. Hopkins, and Michael Rodeh, International Business Machines Corporation. Speculative Load Instruction Rescheduler for a Compiler Which Moves Load Instructions Across Basic Block Boundaries While Avoiding Program Exceptions. United States Patent 5526499. http://www.patents.ibm.com.

[8] David M. Gallagher, William Y. Chen, Scott A. Mahlke, John C. Gyllenhaal and Wen-mei W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. ACM SIGPLAN Notic-
es, Vol. 29 No. 11, pp. 183-193. Nov. 1994.

[9] Tokuzo Kiyohara, Wen-mei W. Hwu; William Chen, Matsushita Electric Industrial Co., Ltd., and The Board of Trustees of the University of Illinois. Memory conflict buffer for achieving memory disambiguation in compile-time code schedule. United States Patent 5694577. http://www.patents.ibm.com.

[10] Douglas C. Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. University of Wisconsin, Madison Tech. Report. June, 1997.

[11] UNIX System Laboratories Inc. System V Application Binary Interface: MIPS Processor Supplement. Unix Press/Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[12] Matthew Postiff, David Greene, Charles Lefurgy, Dave Helder, Trevor Mudge. The MIRV SimpleScalar/PISA Compiler. University of Michigan CSE Technical Report CSE-TR-421-00. http://www.eecs.umich.edu/mirv.

[13] Intel IA-64 Application Developer's Architecture Guide. May 1999. Order Number: 245188-001.

[14] H. Roland Kenner, Alan Karp, and William Chen, Institute for the Develoment of Emerging Architecture, L.L.C. Method and apparatus for implementing check instructions that allow for the reuse of memory conflict information if no memory conflict occurs. United States Patent 5903749. http://www.patents.ibm.com.

[15] Robert Yung and Neil C. Wilhelm. Caching Processor General Registers. Intl. Conf. Computer Design, pp. 307-312, Oct, 1995.

[16] Robert Yung and Neil C. Wilhelm. Caching Processor General Registers. Sun Microsystems Laboratories Tech. Report. June, 1995.

[17] Matthew Postiff, David Greene, and Trevor Mudge. Exploiting Large Register Files in General Purpose Code. University of Michigan Technical Report CSE-TR-434-00. http://www.eecs.umich.edu/mirv.

[18] H. Dietz and C.-H. Chi. CRegs: A New Kind of Memory for Referencing Arrays and Pointers. Proc., Supercomputing '88: November 14--18, 1988, Orlando, Florida, pp. 360-367, Jan, 1988.

[19] S. Nowakowski and M. T. O'Keefe. A CRegs Implementation Study Based on the MIPS-X RISC Processor. Intl. Conf. Computer Design, VLSI in Computers and Processors, pp. 558-563, Oct, 1992.

[20] Peter Dahl and Matthew O'Keefe. Reducing Memory Traffic with CRegs. Proc. 27th Intl. Symp. Microarchitecture, pp. 100-104, Nov, 1994.

[21] David W. Wall. Global Register Allocation at Link Time. Proc. SIGPLAN'86 Symp. Compiler Construction, pp. 264-275, July, 1986.

[22] B. Heggy and M. L. Soffa. Architectural Support for Register Allocation in the Presence of Aliasing. Proc., Supercomputing '90: November 12--16, 1990, New York Hilton at Rockefeller Center, New York, New York, pp. 730-739, Feb, 1990.

[23] T.-C. Chiueh. An Integrated Memory Management Scheme for Dynamic Alias Resolution. Proc., Supercomputing '91: Albuquerque, New Mexico, November 18-22, 1991, pp. 682-691, Aug. 1991.

[24] David R. Ditzel and H. R. McLellan. Register Allocation for Free: The C Machine Stack Cache. Proc. Symp. Architectural Support for Programming Languages and Operating Systems, pp. 48-56, March, 1982.