Register Windows vs. Register Allocation

David W. Wall

December, 1987

Western Research Laboratory 100 Hamilton Avenue Palo Alto, California 94301 USA

Abstract

A large register set can be exploited by keeping variables and constants in registers instead of in memory. Hardware register windows and compile-time or link-time global register allocation are ways to do this. A measure of the effectiveness of any of these register management schemes is how thoroughly they manage to remove loads and stores. This measure also must count extra loads and stores that had to be executed because of window overflow or conflicts between procedures.

By combining profiling, instrumentation, and in-line simulation, we measured the effectiveness of several register management schemes. These included compile-time and link-time schemes for allocating 52 registers, and register window schemes using 128 registers organized into fixed-size or variable-sized windows. Link-time allocation based on profile information was the clear winner in some cases and did about as well as windows in the rest. Even link-time allocation based on an estimated profile was about as good as windows. Variable-sized windows sometimes did better than fixed-sized windows, but the difference was usually small.

Register windows require additional logic in the data path, which may slow the machine cycle slightly, and sometimes use more chip real estate for additional registers. Proponents of windows suppose that they trade these drawbacks for a reduction in the number of memory references that must be made. Our results show that this tradeoff should be made the other way: keep the hardware simple, because a link-time register allocator can effectively duplicate the improvement in memory reference frequency, and the cycle time can therefore be kept as small as possible, resulting in faster programs overall.

This paper also appears in *Proceedings of the SIGPLAN ’88 Conference on Programming Language Design and Implementation,* June 1988.

1. Introduction

Recent machines [1,2,8,10,11] commonly have large register sets. There are several ways of keeping variables and constants in these registers. Some are *register window* approaches, in which the hardware maintains windows of fixed or variable size. Some are *register allocation* approaches, in which the compiler system allocates registers from a flat register set. In each case the goal is to take advantage of a large register set by keeping some or all of the variables in registers rather than in memory, so that fewer memory references are made.

The worth of a register management scheme depends on two things. The first is how thoroughly it keeps variables and constants in registers. The second is how many spills and reloads it must perform in the process. Without a register management scheme, all variables and constants reside in memory. We must therefore perform some number M_0 of loads and stores to move them between their memory homes and the registers where we can operate on them.* If we have a lot of registers and some scheme to manage them, most of these loads and stores can be omitted, leaving M_1 that must still be executed. Thus the ratio of M_1 to M_0 is one measure of the effectiveness of a register management scheme. On the other hand, all register management schemes add some loads and stores as well. For example, a register window scheme must spill some registers when it runs out of windows after a long chain of calls, and a register allocation scheme must spill and reload registers around a recursive call. If the register management scheme requires us to perform S loads and stores that we did not need to perform without it, a more precise effectiveness measure is the ratio $(M_1+S)/M_0$. We will call this measure the *miss ratio* for the register management scheme.

* Other loads and stores, to access array elements or to follow pointer values, must be done no matter how well we manage registers; M_0 does not include these.

A miss ratio of zero is the best possible. It means that all variables resided in registers and no extra spills and reloads were required. A miss ratio greater than unity would mean that the register management scheme had actually made things worse.

2. Link-time code modification

We want to compute the miss ratio for a wide spectrum of register management schemes, including allocation schemes and window schemes. Our technique of link-time code modification [13] was an effective way to do this. To explain why, we must describe how our register allocator works.

In our system, register allocation consists of three steps. The compiler generates code assuming that all variables and constants reside in memory, loading them into a small set of temporary registers as needed. The object modules are annotated with a description of how each instruction uses the variables and procedures of the program. At link time, the allocator constructs a summary of the entire program. It uses this summary to decide, in some manner, which variables and constants should reside in registers rather than in memory. As the modules are linked together, the linker rewrites them based on this register allocation, removing loads and stores associated with variables that now reside in registers. This is easy because of the annotations provided by the compiler.

The rewriting of the code works no matter how the choice of register variables is made. The allocator tries to choose the variables that are used the most, and it can base this decision either on estimates computed by the compiler or on actual counts obtained by profiling. Similarly, we can allocate registers *cooperatively* or *selfishly*. In cooperative allocation, each procedure may not get all of its locals in registers, but procedures in the same call chain use different registers. Spills and reloads are therefore needed only for recursive calls or indirect calls through procedure variables. In selfish allocation, we allocate registers for the procedures independently, so that each procedure is allowed to use all of the registers. However, this means that the registers it uses must be spilled at procedure entry and reloaded at procedure exit.

To change between cooperative and selfish mode, we merely re-link the program and use different meanings for the annotations concerning procedure calls and returns. No recompilation is needed.

3. Computing the miss ratio

Whether we use cooperative or selfish allocation, it is easy to compute our miss ratio. We can build a *use profile* for a program, which lists each variable and the number of loads or stores that can be removed if this variable resides in a register. To build this profile we insert counting code at the beginning of each basic block. We then combine the resulting dynamic counts with static information about each block, gathered by the compiler. Then for any allocation, we can look up the register variables in the use profile to determine the dynamic number of loads and stores we have removed. The code rewriter can help us determine the number of spills and reloads executed by inserting counting code wherever it inserts spills and reloads. Together these give us the miss ratio.

We can even use this approach to compute the miss ratio of a register management scheme based on hardware windows. If we allocate registers in selfish mode, the result is that we remove loads and stores as we would if we used a window machine. However, we must not count the spills and reloads that the rewriter inserts around each procedure. These spills and reloads are required for our machine, but they would not be present in a window machine. On the other hand, a window machine *would* perform spills and reloads when there was an overflow or underflow of the window buffer. The rewriter can count these by inserting code to simulate the state of the window buffer at each procedure entry and exit.

4. Register management schemes

We examined several kinds of register allocation and hardware windows. For each, the compiler did a liveness analysis and register coloring [3] for the local variables within each procedure.* This allowed it to partition the procedure's locals into groups such that no two locals in the same group are ever simultaneously live. All the locals in one group could then be kept in the same register, and each group looks like just a single local for the purposes of register allocation. When we talk about allocating a register to a local, we really mean allocating a register to a group of non-conflicting locals.

Most of the schemes are guided by the frequencies with which variables and constants are used. Depending on the scheme, we may want these frequencies to cover the entire execution of the program or to cover a single call of a given procedure. The frequencies can be compile-time estimates, but we can usually improve the miss ratios by using actual profile information rather than estimates.

4.1. Compile-time allocation

To simulate compile-time allocation, we allocate registers selfishly. Each procedure gets a register for each of its locals, and performs spills at procedure entry and reloads at procedure exit. We have only 52 registers available†, but because of coloring this turns out to be more than enough.

Globals are not kept in registers; intermodule analysis would be needed to do this safely. Locals referenced from a nested procedure are also not kept in registers. Finally, locals used less than twice per call are not kept in registers, because the resulting savings would be less than the expense of spilling and restoring the registers to keep them in. We identify locals used less than twice per call by looking at the compile-time estimates or actual use profile.

4.2. Link-time allocation

Our system does cooperative allocation at link time. The allocator builds an entire call graph at link time, neglecting recursive calls and calls made indirectly through procedure variables. It traverses this call graph from the leaves toward the root, allocating an unlimited number of ‘‘pseudo-registers’’ to the locals. By using the call graph, we allow locals of procedures in different call chains to claim the same pseudo-register. The allocator also allocates one pseudo-register to each global. At this point each variable and constant has been assigned to some pseudo-register. The allocator then selects the 52 pseudo-registers that are used most

* To be more specific, for each procedure the compiler builds a *conflict graph.* The conflict graph has a vertex for each local variable and an edge between any two locals that are simultaneously live. This conflict graph is then colored with a linear-time heuristic algorithm. The compiler makes no attempt to break a single variable up into disjoint *live ranges.* Memory locations used as long-term internal temporaries are included in the locals.

† Our machine, the Titan, has 64 registers. Four are reserved for bookkeeping and eight are reserved for temporary values, including values of variables not resident in registers.

frequently, based on the compile-time estimates or on a use profile. These 52 pseudo-registers become actual registers; the rest stay in memory. Spills and reloads must be inserted around recursive and indirect calls; to count these, we direct the allocator to insert counting code wherever it inserts spills and reloads.

4.3. Mixed-strategy allocation

Steenkiste's scheme [12] is a mixed strategy. It builds the call graph as before, and allocates registers cooperatively starting at the leaves. The use frequencies of the variables are not considered in this process, so there is no need to allocate pseudo-registers first. When it is high enough in the call graph that no more registers are available, Steenkiste's scheme switches to selfish allocation, doing spills and reloads at each call. The idea is that most of a program's time is spent near the leaves of the call graph, and calls in that region will be cheap because no spills are needed. Steenkiste's scheme will always have a miss ratio that is no worse than compile-time, because it is the same as the compile-time scheme except near the leaves.

When Steenkiste's system finds procedures that call each other in a recursive cycle, it handles them with selfish allocation, and treats the entire cycle as a single node when it does the cooperative allocation.

To approximate Steenkiste's approach, we allocate registers selfishly for the whole program. Our system inserts spills and reloads around each procedure, where Steenkiste's would do so only for some procedures. This means we must count the spills and reloads executed only if they correspond to spills and reloads that would in fact be needed. To determine this, the allocator analyzes the call graph the same way Steenkiste's system would, by finding the strongly connected components, reducing them to single nodes, and traversing the graph starting from the leaves. When it encounters a procedure that makes an indirect call, it reverts to selfish allocation above that point even if it is still near the leaves. Procedures that need spills keep each local in a register if it is used at least twice per call, balancing the cost of the spill and reload. Procedures that do not need spills keep each local in a register if it is used at all. In each case the decision may be made based either on estimates or on a profile. Globals are not kept in registers.

4.4. Register windows

Each register window scheme assumes a set of registers organized as a circular buffer. When a procedure is called, the tail of the buffer is advanced in order to allocate a new window of registers that the procedure can use for its locals. On overflow, enough registers are spilled from the head of the buffer to make room for the new window at the tail. These registers are not reloaded until a chain of returns makes it necessary. We assume[*] that spills and reloads have no cost except the loads and stores required, so there is no penalty for spilling as little as possible each time. Globals are not kept in registers; intermodule analysis would be needed to avoid aliasing problems, and avoiding this analysis is a major motivation for register windows. Local variables that are referenced from a nested procedure are also not kept in registers.

We simulate a register window scheme by instructing the register allocator to allocate registers selfishly. Each procedure gets to use all the registers available, as if it had a window

[*] Generously, giving window systems the benefit of doubt.

of registers untouched by other procedures. The procedure spills these registers on entry and reloads them on exit. Because these spills and reloads would not be needed on a window machine, we do not count them in our miss ratio. However, we also instruct the allocator to insert code at procedure entry and exit that simulates the state of a window machine. This code keeps track of the current head and tail of the simulated windows, and the number of spills and reloads that would be needed for overflow and underflow. These spills and reloads do not appear in our own code, but they would be executed in a window machine, and so they are counted in the miss ratio. Locals that do not seem to be used at all are not kept in registers.

We simulate fixed-size windows of w registers by limiting the register allocator to use only w registers total; in selfish allocation it uses them for each procedure independently. We simulate variable-sized windows by not limiting the number of registers; each procedure gets one register for each of its locals. Since many procedures have only a few locals, especially after coloring, this scheme makes better use of the registers, and needs to spill or reload less often.

5. Results

We experimented with variations of these schemes using five synthetic benchmarks and seven real programs in use at DEC Western Research Lab. These twelve programs are summarized in Figure 1. Yacc, rsim, and mx are written in C, the Boyer benchmark is written in Scheme, and the rest are written in Modula-2. The Scheme compiler is recent and has not been tuned as extensively as the others.

	lines	*vars*	*procs*	*remarks*
Livermore	268	166	1	Livermore loops
Whetstones	462	254	13	Floating-point
Linpack	814	214	12	Linear algebra [5]
Stanford	1019	402	49	Hennessy's suite [7]
Boyer	600	700	22	Lisp theorem-proving [6]
mx	18000	2600	444	Mouse-based editor
yacc	1856	665	48	Compiler-compiler
ccom	9591	3155	370	C compiler front end
metronome	4287	1395	137	Timing verifier
rsim	3003	811	97	Logic simulator
grr	5883	2290	286	PCB router
eco	2721	1031	132	Recursive tree comparison

Figure 1. The twelve test programs.

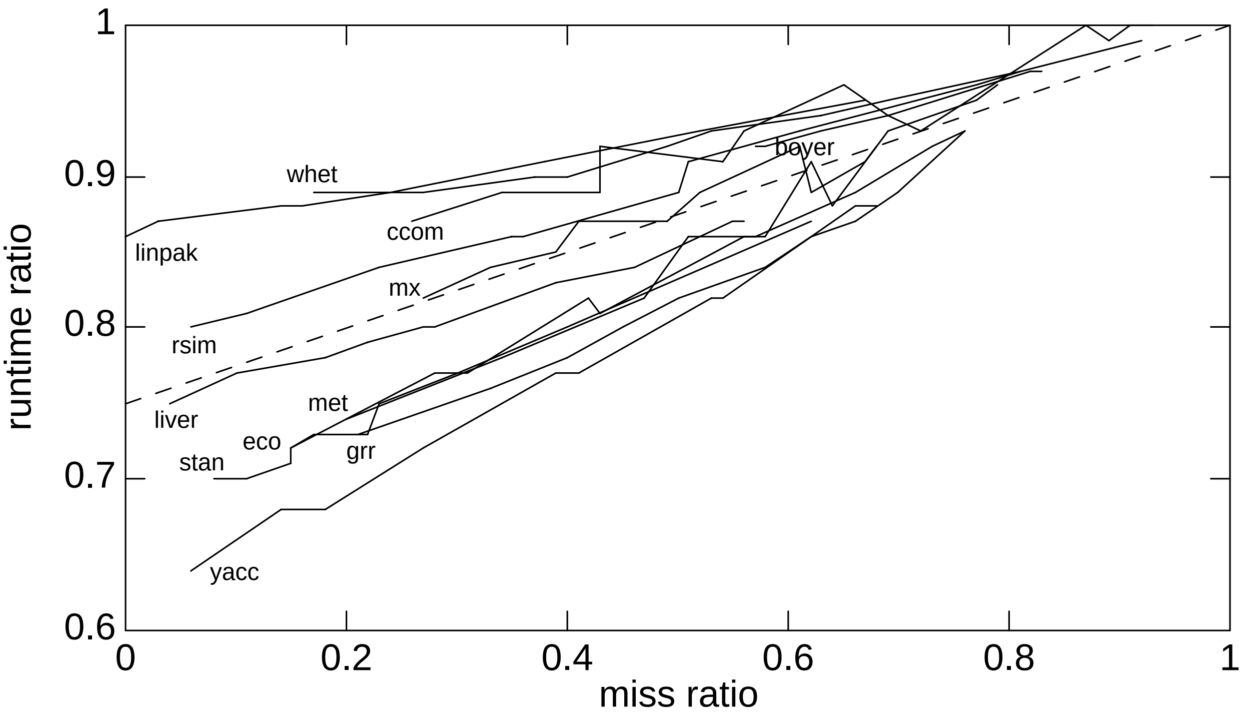

Figure 2. Effect of miss ratio on execution time.

5.1. How does miss ratio affect execution time?

Our machine has a one-cycle data cache, so loads and stores take slightly over one cycle on the average. Instruction-level profiling reveals that around one fourth of the instructions executed are loads and stores of variables and constants. We would therefore expect to get a 25% improvement in execution time if we achieved a miss ratio of zero. This improvement will normally be smaller by an amount proportionate to the actual non-zero miss ratio.

We confirmed this experimentally by limiting our allocator to different numbers of registers. In each case we plotted the miss ratio against the ratio of execution time after register allocation to execution time for unimproved code. Figure 2 shows the results. Each program shows a roughly linear relationship between miss ratio and time ratio. The average slope is indeed about .25.

The slope would be larger if the data cache were slower. Around one sixth of the executed instructions are loads and stores accessing data structures. These would be affected by a slower data cache, as would the removable loads and stores of scalars. In general, the slope would be

$$\frac{N/4}{7/12 + N/6 + N/4}$$

if we had an N-cycle data cache. If N=2, this is about .35; if N=3, it is about .41.

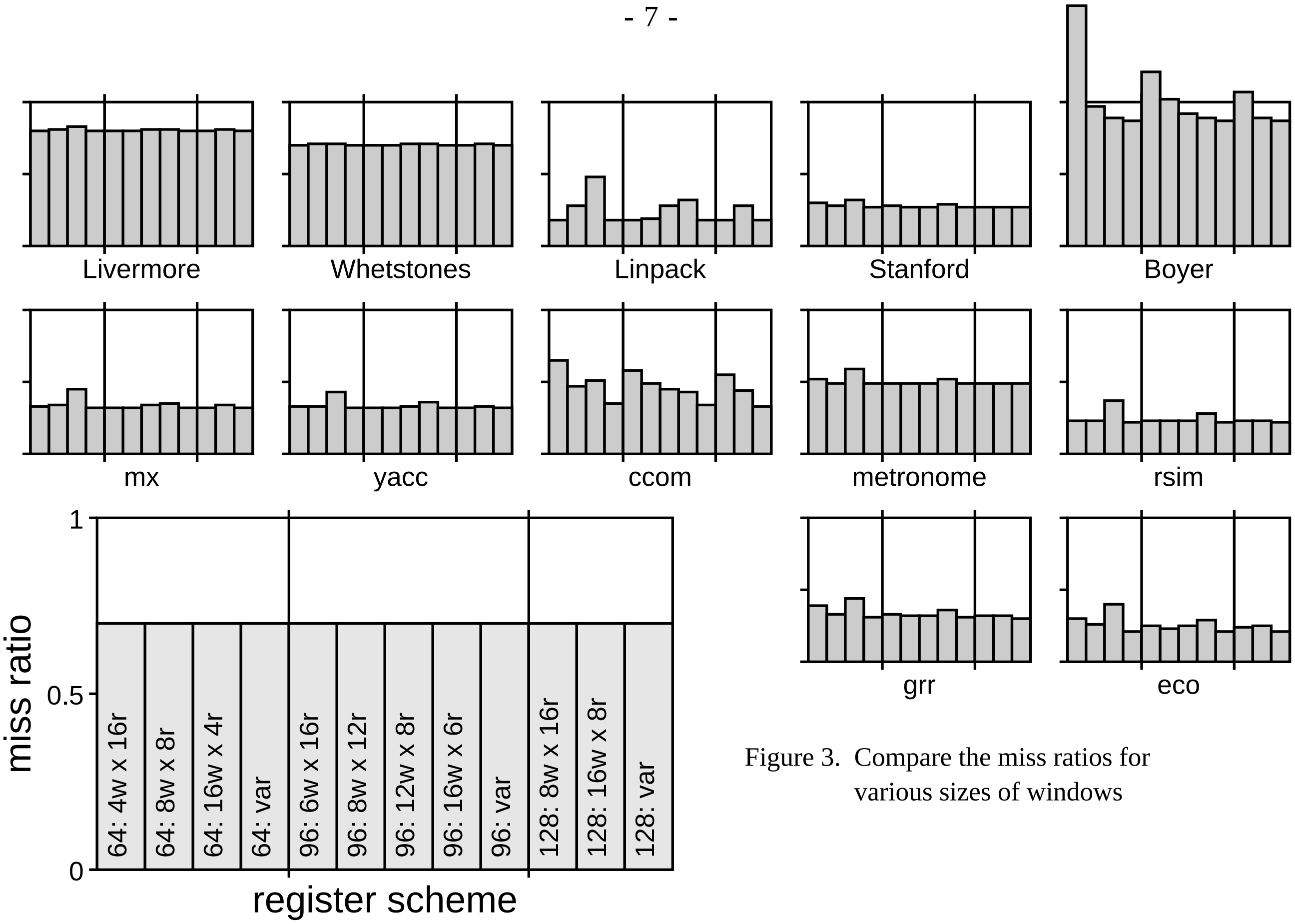

Figure 3. Compare the miss ratios for various sizes of windows

5.2. What kind of windows work best?

We simulated machines with 64, 96, and 128 registers. We used the registers as fixed-sized windows with different window sizes, and also as variable-sized windows. Figure 3 shows the results.

We got the best results for fixed-sized windows when each window had eight registers. It was better to add more windows than to increase the window size. However, adding more windows seldom helped much—the miss ratio was about the same no matter how many eight-register windows we had. Adding more made the results less sensitive to the choice of window size, but had little other effect.

Variable-sized windows were sometimes noticeably more effective than the best fixed-sized windows, but the difference is often surprisingly small. Variable-sized windows did occasionally require two or three orders of magnitude fewer spills and reloads than the same number of registers organized as fixed-sized windows. However, even with fixed-sized windows, the spills and reloads were usually insignificant compared to the normal loads and stores of variables and constants not in registers.

Nothing comes free. Adding more registers or the hardware to allow variable-sized windows must cost something. The cycle time may have to increase, and some chip real estate is lost to more productive uses. This implies that a window machine will probably be better off if it simply uses eight fixed-sized windows of eight registers. To try to get more locals in registers by adding more registers or by organizing them as variable-sized windows is not productive.

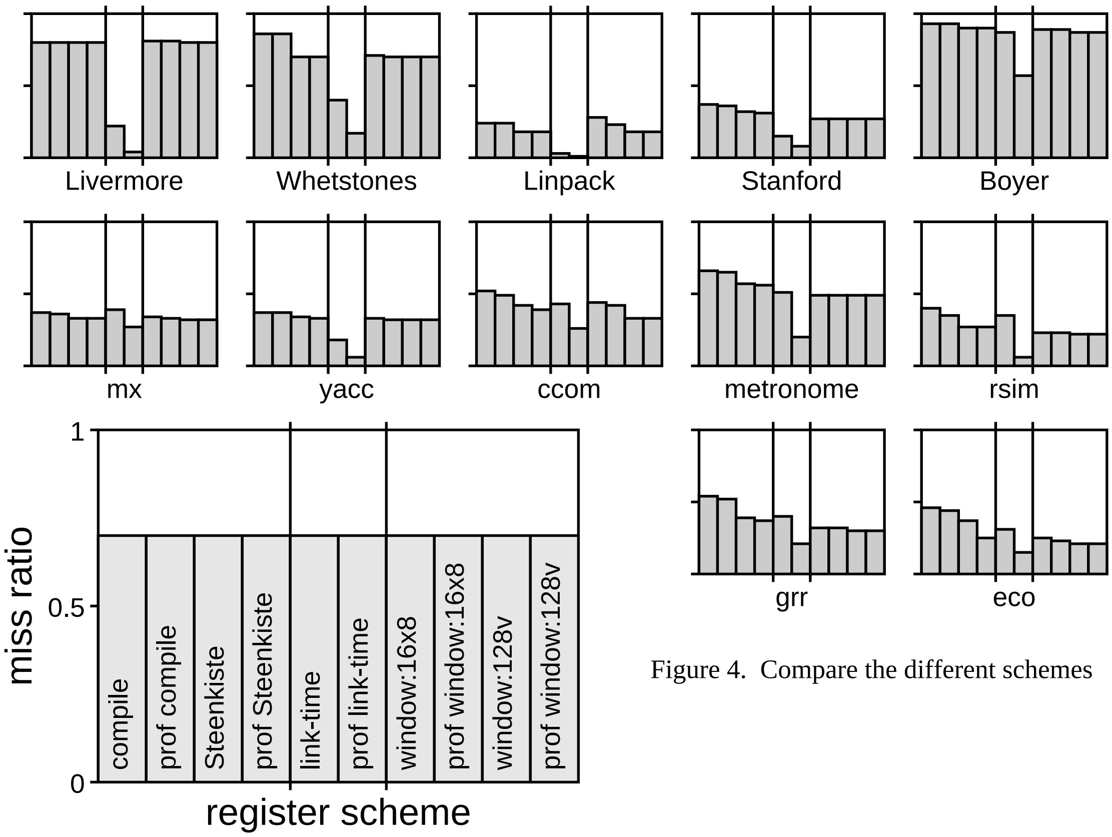

Figure 4. Compare the different schemes

5.3. Do software or hardware schemes work best?

We tested five register management schemes: compile-time, Steenkiste's mixed strategy, link-time with use estimates, 16 fixed-sized windows of 8 registers, and variable-sized windows using 128 registers. The compile-time and link-time schemes used only 52 registers; our experimental framework prevents us from giving them more. This gave the window schemes more than twice as many registers as the compile-time and link-time schemes, but this may be fair because windows can use more registers without requiring extra bits in instruction operands. We also applied each scheme using a profile of variable and constant references instead of the compile-time estimates. These profiles were from a previous run using the same input data, so the improvement is as large as possible. The results are shown in Figure 4.

Compile-time allocation almost always does poorly. This is no surprise; although each procedure gets all the registers it needs, it must spill and reload them on entry and exit. Steenkiste's scheme does better than compile-time allocation, and occasionally slightly better than our link-time scheme.

As we saw before, variable-sized windows sometimes do better than fixed-sized windows, but the difference is often disappointing. With 128 registers available, even fixed-sized windows rarely need to spill.

Our scheme of frequency-based link-time register allocation does much better on the synthetic benchmarks than the other schemes do.* Much of the difference disappears when we

* Synthetic benchmarks can be quite misleading. The main reason for including them in this paper was to show how different their behavior is from real programs.

look at the real programs. However, link-time allocation based on estimates still does considerably better than compile-time allocation, and about as well as the window schemes. The miss ratio of a window scheme was sometimes smaller by as much as 0.12. This corresponds to a difference in execution time of about 3% even with the same cycle time. Since a window machine requires more hardware, its cycle time is probably larger. Even an additional inverter in a twenty-level data path would increase the cycle time by 5%.

Profile-based link-time allocation always did better than the other schemes. In fact, link-time allocation was the only scheme that was able to profit significantly from a use profile. This is understandable. A use profile lets the compile-time scheme and Steenkiste's scheme ignore any local whose uses are outweighed by the expense (if any) of spilling and reloading its register. Similarly, a use profile lets the fixed-sized window scheme allocate each procedure's limited window registers to the procedure's most frequently used locals. In most cases, however, estimates alone are good enough for these purposes. The variable-sized window scheme did not improve at all.

The mx editor makes extensive use of call-back procedures [9], and the eco tree comparison utility is heavily recursive [4]. Cooperative register allocation should require a lot of spills and reloads for these two, but in practice they do not seem to behave differently from the other tests. We suspect that even when the programmer uses a recursive or object-oriented style, most procedure calls are nevertheless direct nonrecursive calls to known procedures.

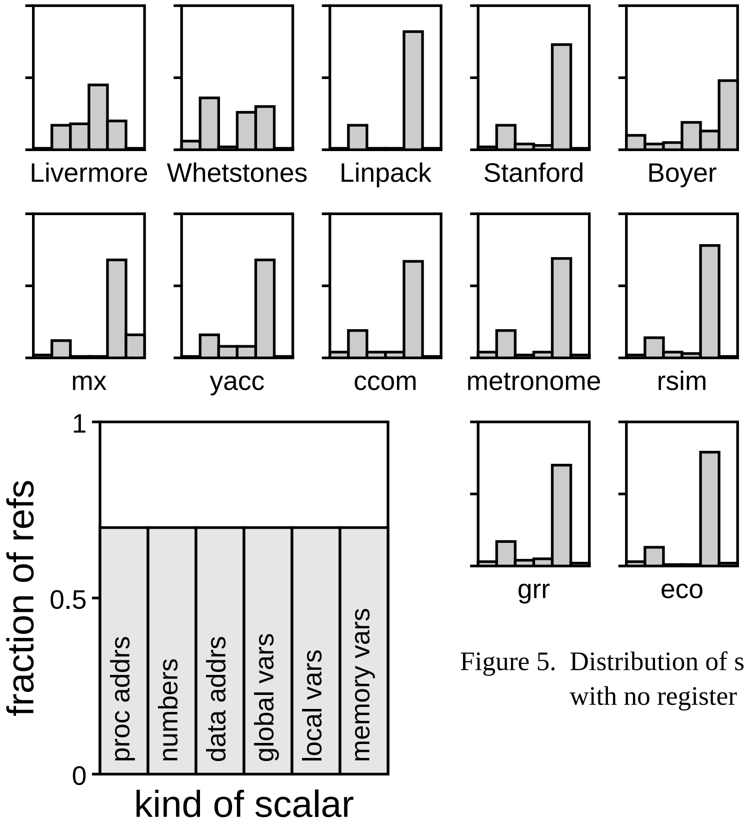

Figure 5. Distribution of scalar references with no register management scheme

5.4. What about globals?

The main advantage of the link-time scheme is the ability to keep globals in registers. There are four kinds of globals: genuine global variables, numeric constants, addresses of procedures, and addresses of global arrays or records. Figure 5 shows how the scalar memory references are distributed if no register management scheme is used.* Getting the locals into registers is a big step, but it is not the whole story. Global variables and constants have their own share of references. Global variables are hard to keep in registers, because we need global analysis to do it safely. However, techniques do exist for removing references to numeric constants and procedure addresses.

One technique for removing references to numeric constants is to allow immediate operands in instructions. Our machine allows immediate operands between 0 and 63 in some instructions, and so the number of memory references in unimproved code is already smaller than it might be. Another technique is to allocate a handful of global registers (which even window machines often have) to numeric constants like 0 or 1 that are common in all programs.

To remove references to procedure addresses, some machines have a special procedure-call instruction with a two-bit opcode and a 30-bit word address destination. This allows a procedure address to be treated as a long immediate operand. The effect on the miss ratio is

* ‘‘Memory variables’’ are scalar variables that cannot reside in registers because they are sometimes accessed indirectly via their addresses.

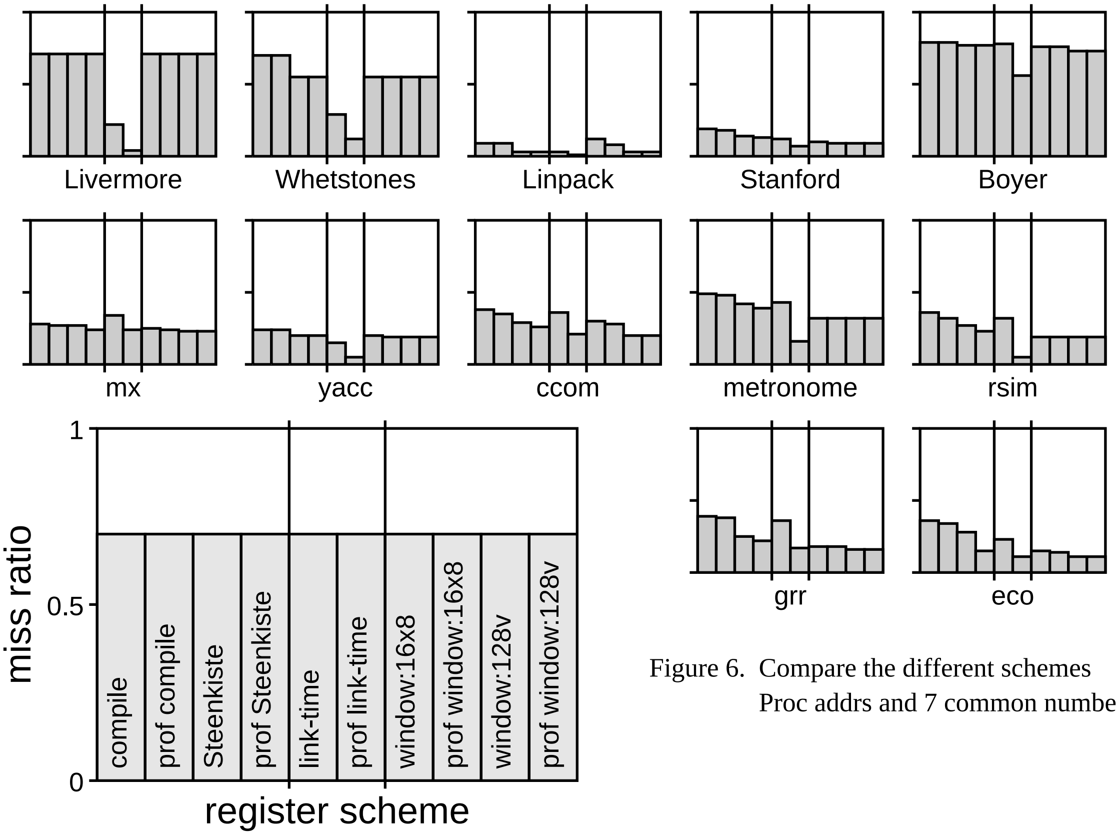

Figure 6. Compare the different schemes
Proc addrs and 7 common numbers free

the same as if we somehow had enough extra global registers to hold all of the procedure addresses. Figure 5 shows us, however, that this will rarely make a big difference.

How would the different schemes compare if we dedicated a few registers to common numeric constants, and if a procedure could be called without loading its address? We answered this question by forcing procedure addresses and certain numeric constants to stay in memory, so that each register management scheme would use its registers as well as possible for other things.* Then we gave them credit for removing references to those procedure addresses and constants, as if they *had* been given registers.

The numeric constants treated this way were 0, 1, 2, 64, 76, 204, and -1. Each of these (and no others) accounted for more than 0.1% of the variable and constant references made by at least four of the twelve programs. They were also the only numbers that accounted for more than 0.1% of the references in at least half of the seven real programs. Except for the integer 76, they were all in the top ten numbers referenced for at least two of the twelve programs.

The results are shown in Figure 6. These extensions have not helped the frequency-based link-time schemes much; the constants and procedure addresses that really matter were kept in registers already. The other schemes all improved by the same amount, depending on the program. This allowed one or both of the window schemes to do as well as profiled link-time allocation in a few cases, but by no means in all of them. The improvement from these

* In fact, this did not affect the allocation made for the compile-time and window schemes, which do not consider globals anyway.

extensions came primarily from the numeric constants rather than the procedure addresses, which is consistent with the distribution we saw in Figure 5.

When neither one used profiles, Steenkiste's scheme managed to beat our link-time scheme about half the time. This suggests that more aggressive pursuit of globals might make it consistently better than ours when no profile is available.

The main competition for profiled link-time allocation is variable-sized windows. Why does allocation still do so much better sometimes? To answer this question we looked at the unremoved references that contribute to the miss ratio. These references fall into categories according to the kind of scalar referenced: procedure addresses, numeric constants, local variables, and so on. Figure 7 shows the contribution of each category to the miss ratio, for both schemes. For example, variable-sized windows have a .71 miss ratio on the Livermore benchmark, of which .08 is references to numeric constants. The *total W* column is the sum of the rest of the *W* columns (except for rounding errors).

			data		*global*		*local*		*memory*		*spills/*			
	numbers		*addrs*		*vars*		*vars*		*vars*		*reloads*		*total*	
	W	*A*	*W*	*A*	*W*	*A*	*W*	*A*	*W*	*A*	*W*	*A*	*W*	*A*
Livermore	.08	.01	.18	.01	.45	.02		.01	.00	.00			.71	.04
Whetstones	.26	.10	.02	.00	.26	.00		.01	.01	.01			.55	.12
Linpack	.03	.00	.00	.00	.00	.00		.00	.00	.00			.03	.00
Stanford	.02	.01	.04	.00	.03	.00		.00	.00	.00	.00	.05	.09	.07
Boyer	.01	.00	.05	.00	.19	.00		.00	.48	.48	.00	.08	.73	.56
mx	.05	.03	.01	.00	.01	.01		.04	.16	.16		.00	.23	.24
yacc	.03	.02	.08	.01	.08	.01		.01	.00	.00		.00	.19	.05
ccom	.10	.09	.04	.01	.04	.01	.00	.04	.01	.01	.01	.03	.20	.21
metronome	.06	.02	.02	.01	.04	.01	.18	.06	.02	.02		.04	.32	.16
rsim	.12	.01	.04	.00	.03	.00		.01	.00	.00	.00	.02	.19	.05
grr	.05	.02	.04	.02	.05	.02		.07	.02	.02	.00	.02	.16	.17
eco	.06	.04	.01	.01	.01	.00		.03	.02	.02	.00	.01	.11	.11

Figure 7. Distribution of the miss ratio by categories for variable-sized windows (W) and profiled allocation (A), with procedure addresses and seven common numbers free. Empty entries are exactly 0; non-zero entries less than .005 are rounded to .00.

The window scheme usually removed more references to locals than allocation did, but this was not always true: The metronome program has several locals that are referenced nonlocally from a nested procedure, which allocation can handle but windows cannot. Allocation usually requires more spills and reloads—the portion of the miss ratio due to spills and reloads for variable-sized windows was never more than 0.01—but the truth is that neither scheme does much spilling. Allocation does not cover local variables as thoroughly as windows do, but mostly compensates by removing references to global variables and data structure addresses.

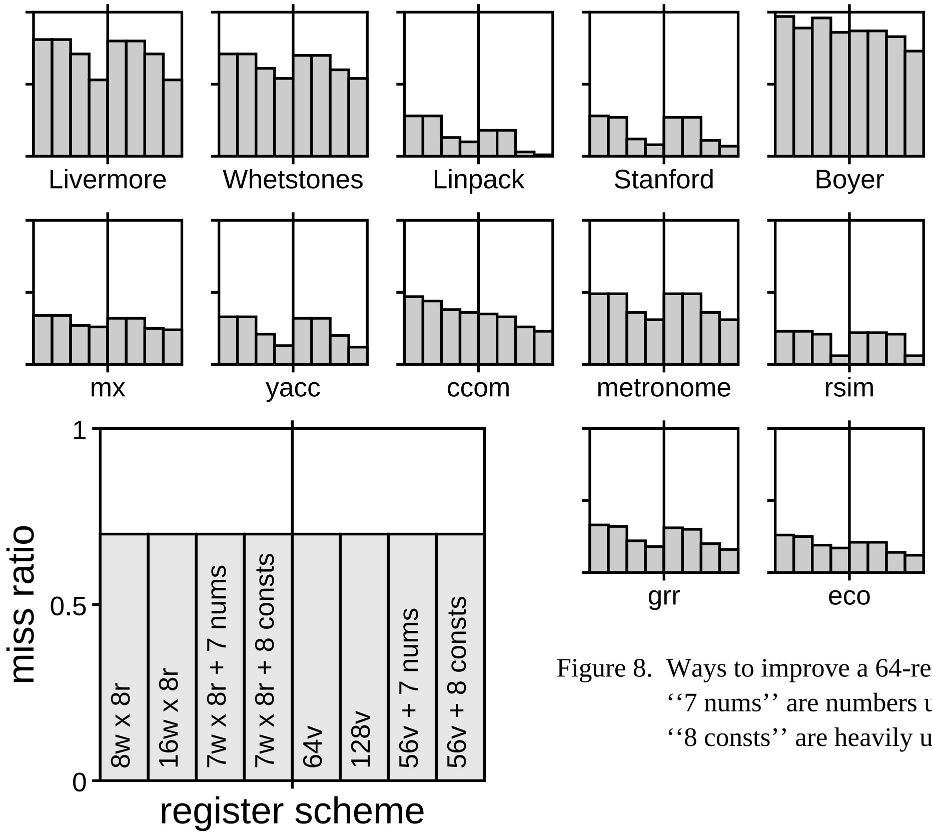

Figure 8. Ways to improve a 64-register window scheme
''7 nums'' are numbers used by most programs
''8 consts'' are heavily used by a given program

5.5. More globals or more windows?

We have seen that there is little difference between a 64-register window machine and a 128-register window machine. We have also seen that both get good improvements by having a few global registers in which they keep important constants. In fact, we can improve a 64-register window machine more by *taking away* eight registers and using them for constants than we can by doubling the number of registers used for windows.

Figure 8 shows the miss ratios for 64 and 128 window registers, and for 56 window registers plus 7 or 8 global registers. The window registers were organized as eight-register fixed sized windows and as variable-sized windows. We chose the constants kept in the global registers by two different methods.

In one case we chose the same constants for all of the test programs. The constants we chose were the seven commonly used numbers (0, 1, 2, 64, 76, 204, and -1, as explained above).

In the other case we chose different constants for each program. The constants we chose were the eight used most often by the program, including procedure and data addresses. Selecting register constants in this way would require some kind of profiling ability. However, it would not require the complete aliasing analysis that selecting true global variables would.

The results show that taking away a few registers, if they are put to better use, is considerably better than adding a lot more registers. And we can see that constants individually selected from a specific program are better than numbers selected because they are used by a lot of programs. Neither is a big surprise.

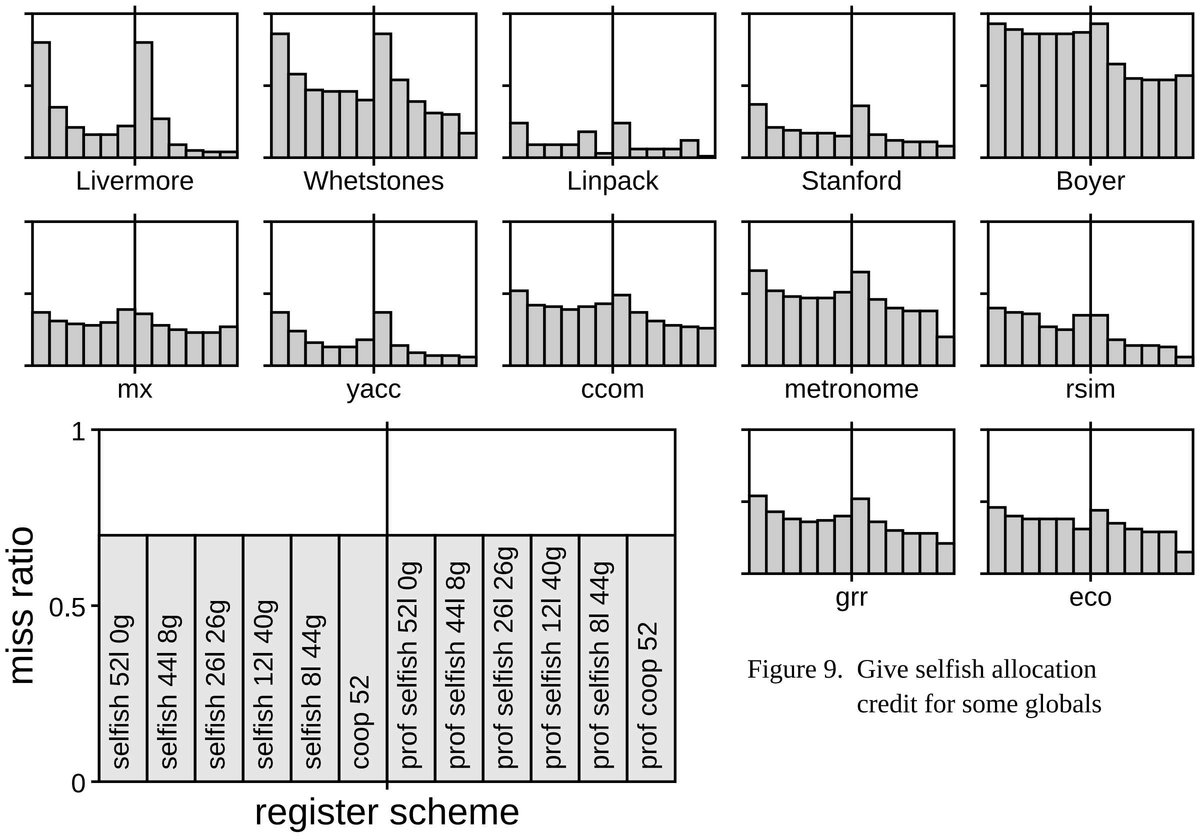

Figure 9. Give selfish allocation credit for some globals

5.6. Selfish allocation or cooperative allocation?

We used selfish allocation, in which each procedure allocates registers independently and spills and reloads them at entry and exit, to simulate compile-time allocation. Augmented by a few global registers for constants, this compile-time scheme was sometimes as good as the cooperative link-time scheme, at least if no profile was used. We might also ask whether selfish or cooperative allocation is the better technique to use in a link-time system. Suppose that at link-time we partition the available registers between a set to be used for selfish allocation and a set to be used for globals. The globals would be selected by estimate or by profile just as our cooperative link-time scheme selects them. This technique would keep more locals in registers, and possibly more globals as well, but it would do more spilling and reloading. Would this technique do better or worse than cooperative allocation?

We tried this for various partitions of the 52 available registers. The results are shown in Figure 9. Selfish allocation does best when most of the registers are reserved for globals. This is consistent with our discovery that windows of more than eight registers are not very useful. When no profile is used, selfishly allocating twelve registers and using the rest for globals works better than cooperative allocation in most cases. On the other hand, using a profile gives cooperative allocation a substantial advantage.

The closeness of the the two approaches was a surprise. We had believed that the advantage of link-time allocation was that spills and reloads were not needed. In fact the advantage is that globals can be kept in registers. How you arrange to cover locals doesn't matter so much, as long as you somehow get the important ones.

6. Conclusions

Our technique of link-time register allocation through code rewriting was a valuable testbed for comparison of register management schemes. Even window machines, which required simulation rather than merely instrumentation, were handled simply by changing the link-time system, and required no recompilation and no machine simulator.

Link-time register allocation based on estimated use frequencies removes memory references at least as well as compile-time register allocation, and nearly as well as register window schemes. Link-time register allocation based on *profile* frequencies does best across the board. Other schemes do not benefit as much from profiling because they do not expend any effort trying to *decide* which variables to keep in registers. Variable-sized windows occasionally do noticeably better than fixed-sized windows, but the two are usually nearly indistinguishable.

The advantage of link-time analysis is that it can keep a few important globals in registers. The most important such globals are numeric constants; by keeping common ones in global registers, much of the advantage of link-time allocation can be removed. Window schemes do little spilling with as few as 64 registers, and it is better to add an ample group of global registers than to add more registers to the window buffer. An instruction that allows us to call a procedure without loading its address is somewhat useful, but its usefulness is small.

A register management scheme that is thorough about keeping globals in registers must do two kinds of intermodule analysis. It must do aliasing analysis to determine which global variables are safe to keep in registers. It must do frequency analysis to decide which variables and constants are used enough to be *worth* keeping in registers. These analyses come free with our sort of link-time allocation.

The choice of a register management scheme depends on several factors. One factor is its effect on the cycle time of the machine. Making the data path even slightly longer can slow programs down more than the improvement in miss ratio speeds them up. Another factor is the cost of lost opportunity. Chip real estate that implements a hardware window scheme, including the larger register set such a scheme usually assumes, is unavailable for other uses. Adding more hardware to make up the difference only slows the machine down further. In contrast, programmer effort spent designing a compile-time or link-time scheme is not as thoroughly lost: These schemes are not difficult and can be implemented at any time, and they mesh well with classical optimization techniques. Though register windows are an elegant idea, they are not the best way to get fast programs.

Acknowledgements

Joel Bartlett, Jeremy Dion, and John Ousterhout made countless useful suggestions regarding the presentation of results. Alan Eustace, Norm Jouppi, and Forest Baskett helped me with the assertions about the hardware consequences of register windows. John Hennessy kindly explained how I had misunderstood Steenkiste’s technique, just in time to correct this paper. My thanks to all.

References

[1] Advanced Micro Devices. *Am29000 Streamlined Instruction Processor User's Manual.* Advanced Micro Devices, Inc., 901 Thompson Place, P. O. Box 3453, Sunnyvale, CA 94088.

[2] Russell R. Atkinson and Edward M. McCreight. The Dragon processor. *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 65-69. Published as *Computer Architecture News 15* (5), *Operating Systems Review 21* (4), *SIGPLAN Notices 22* (10), October 1987.

[3] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages 6:* 47-57, 1981.

[4] Jeremy Dion. Personal communication.

[5] Jack J. Dongarra. Performance of various computers using standard linear equations software in a Fortran environment. *Computer Architecture News 11* (5): 22-27, December 1983.

[6] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems,* pages 116-135. The MIT Press, 1985.

[7] John Hennessy. Stanford benchmark suite. Personal communication.

[8] John L. Hennessy, Norman P. Jouppi, Steven Przybylski, Christopher Rowen, and Thomas Gross. Design of a high performance VLSI processor. In Randal Bryant, editor, *Third Caltech Conference on Very Large Scale Integration,* pages 33-54. Computer Science Press, 11 Taft Court, Rockville, Maryland.

[9] John Ousterhout. Personal communication.

[10] David A. Patterson. Reduced instruction set computers. *Communications of the ACM 28* (1): 8-21, January 1985.

[11] George Radin. The 801 minicomputer. *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems,* pages 39-47 (March 1982). Published as *SIGARCH Computer Architecture News 10* (2), March 1982, and as *SIGPLAN Notices 17* (4), April 1982.

[12] Peter Steenkiste. *Lisp on a Reduced-Instruction-Set Processor: Characterization and Optimization.* PhD thesis, Stanford University. Available as Stanford Computer Systems Laboratory Technical Report CSL-TR-87-324. March 1987.

[13] David W. Wall. Global register allocation at link-time. *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction.* Published as *SIGPLAN Notices 21* (7): 264-275 (July 1986).