The Need for Large Register Files in Integer Codes

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

Abstract

Register allocation is an important optimization for high performance microprocessors but there is no consensus in the architecture or compiler communities as to the best number of registers to provide in an instruction set architecture. This paper discusses reasons why this situation has occurred and shows from a compiler perspective that, compared to the conventional 32-register file, 64 or more registers enables performance improvements from 5% to 20%. This is demonstrated with existing advanced compiler optimizations on the SPECint95 and SPEC2000 benchmarks. This work also documents that the optimizations eliminate cache hit operations, converting common-case cache hits to faster register accesses. Finally, this work provides additional measurements for the proper number of registers in a high-performance instruction set and shows that most programs can easily use 100 to 200 registers when multiple active functions are considered for simultaneous allocation to the register file.

1. Introduction

Large register files have many advantages. If used effectively they can: 1) reduce memory traffic by removing load and store operations; 2) improve performance by decreasing path length; and 3) decrease power consumption by eliminating operations to the memory hierarchy. Their merit derives from the fact that they require few address bits and act as a very high-speed cache.

On the other hand, large, multi-ported register files can become the critical path that sets the cycle time of the processor. There have been a number of proposals to circumvent this problem by implementing the register file either by physically splitting it, or providing a cache of the most frequently used registers and having a large backing store for the full architected set of registers. [7, 25, 26, 27]. The caching technique is particularly useful in a register renaming processor, where the architected (logical) register file can be much larger than the physical register file. In such a case, the physical register file acts as a cache for the larger architected register set. In this paper, we assume that the implementation issues can be solved using such techniques.

Instead of considering implementation issues of large register files, we focus on their performance advantages. In scientific and DSP code, there are well-known techniques for exploiting large register files. However, a common view has been that large files are difficult to use effectively or are not necessary in general purpose code. In this paper we will show that this is not indeed the case by applying several existing advanced compiler optimizations separately and then in combination. The results show that at least 64 registers are required for highest performance, and many more if the scope of register allocation is increased beyond the function boundary to include multiple functions which are active simultaneously on the call stack.

Architecture Type	Architecture/Family	Register Configuration
Embedded	SHARC ADSP-2106x [29]	16 primary; 16 alternate; 40 bits wide
	TI TMS320C6x [32]	2 files of 16 registers; 32 bits wide; 1 Mbit on-chip program/cache/data memory
	Philips TriMedia TM1000 [36]	128 registers; 32 bits each
	Siemens TriCore [33]	16 address; 16 data; 32 bits wide
	Patriot PSC1000 [39]	52 general purpose registers; 32 bits wide
Workstation	Intel IA32 [34]	8 int; 8 float in a stack; 32 bits wide
	Transmeta Crusoe TM3120, TM5400 [40]	64 int; 32-bits wide
	Intel IA64 [35]	128 int, rotating and variable-size windows; 128 float; 64 bits wide; 64 1-bit predicates
	DEC Alpha 21x64 [31]	32 int; 32 float; 64 bits wide
	Sun Sparc V9 [30]	8 globals, 16-register window with 8 ins and 8 locals, as well as access to 8 outs which are the next window's ins, all 32-bits. Number of fixed-sized windows from 3 to 32. 32 64-bit floating point regs.

Table 1. Register configurations for a number of embedded and workstation processors.

For control-intensive integer codes, the kind we are focusing on in this work, previous research to determine the best number of registers has not arrived at a clear consensus. For example, one study suggests that the number of processor registers that can be effectively used is limited to a couple dozen [1]. Others have suggested that existing compiler technology *cannot* make effective use of a large number of registers [2]. Studies on the RISC I architecture refer to earlier work which shows that 4 to 8 windows of 22 registers each (for a total of 80 to 144 registers) is sufficient to house the locals and parameters for over 95% of function calls [3, 4]. In addition there has been much effort in the area of optimizing spill code; this is indicative of a need for more registers.

Floating point and DSP codes are generally thought to be able to take advantage of a large number of processor registers if the compiler utilizes advanced transformations such as loop unrolling, register renaming, accumulator and induction variable expansion, and software pipelining. One study confirmed this by showing that for a set of loop nests from the PERFECT and SPEC suites, register utilization increases by a factor of 2.6 after all ILP transformations were applied. While most optimized loops required fewer than 128 registers, the average register usage was about 70 for the loops studied [5]. However, other work which used some similar benchmarks found that after 32 registers, adding registers produced only a marginal performance improvements, particularly if sophisticated code generation techniques are used [6]. Finally another study showed that optimal performance for a number of Livermore kernels requires 128 to 256 registers [7].

This lack of agreement in the architecture and compiler research communities is mirrored in commercial processor designs. Table 1 shows the register configuration of several embedded and workstation-class processors. While it is true that the register set size on some of these machines was constrained by backward compatibility and cost concerns, it is interesting to note the wide variety of register configurations. Incidentally, many of the high performance machines have larger physical register files to implement out-of-order execution. These additional registers are not available to the compiler.

There are several possible reasons for these mixed results, at least as far as integer codes are concerned. First, a small number of registers is suggested by the fact that a typical programmer does not usually have a large number of scalar variables in each function. Second, temporary variables are short-lived and can be packed into a small number of registers. Third, many data references are made through structure or array variables whose components cannot be easily allocated to registers. Fourth, variables local to a function are typically the only candidates considered for permanent register allocation. Though some global variables could be allocated to registers for their entire lifetime, most compilers leave them in memory and shuttle them in and out of registers with load and store operations. Fifth, local variables cannot generally live in the register file while a called function is executing unless a register windowing approach is used to house the variables from multiple functions in different regions of the register file. Sixth, compiler optimizations are often deliberately throttled in order to avoid introduction of spill code. Thus 32 registers might appear to be sufficient because the optimizations have been artificially restricted to that level.

On the other hand, sophisticated compiler optimizations can significantly increase the number of variables by performing loop optimizations. These *optimization temporaries* can create greater register pressure than arithmetic and other simple temporaries since they live over longer ranges. The pool of register allocation candidates can also be enlarged by including array elements and scalar global variables. Furthermore, global variable register allocation can increase the number of registers that are required. Finally, variables that are aliased can be allocated to registers for some portions of their lifetime, further increasing the number of candidates for register allocation.

Many of the earlier studies did not have access to the advanced compiler transformations used today, either because the transformations were not known, or were not available in the compilers used for the studies. The lack of freely available optimizing compilers suitable for research studies has had a limiting effect on earlier research.

This paper makes the case that for integer codes, a large architecturally-visible register file is necessary for high performance, and such large files can easily be utilized by current compiler technology. In fact, if the compiler does not use the right optimizations, the *apparent* register requirements of the program can be significantly lower than reality. To this end, we explore the performance effects of four sets of optimizations in this paper. The first is traditional function-level optimizations. These increase register pressure by adding temporary scalar values and extending scalar lifetimes [8, 9]. The second optimization allocates global scalar variables and addresses to registers for their lifetime. The third is register promotion for aliased variables and global scalars. The fourth is function inlining, which effectively combines the live registers of two functions into one allocation pool. Each of these techniques increases the pool of register allocation candidates and depending on the benchmark can significantly impact performance. All of these optimizations have been examined in previous work but this study examines them in combination in order to present the register requirements of a modern optimizing compiler.

The presentation will proceed as follows. Section 2 describes the compilation environment and includes descriptions of the optimizations that we report on in this paper. Section 3 describes our simulation setup. In Section 4 we present our experimental results. Each transformation is analyzed for its effects on overall performance, register pressure, and other effects. In addition, an incremental analysis is provided to examine how much benefit the non-traditional transformations provide over classical optimization. Section 5 presents some more evidence that a large number of registers is necessary for integer codes. Section 6 describes some previous work on register architecture. We summarize our conclusions in Section 7 and provide directions for future research.

2. Compilation Environment

This section describes the optimizations that were applied to increase register usage. We use the MIRV C compiler. MIRV is an ANSI C compiler which targets the Intel IA32 and SimpleScalar PISA architectures; for this work, the PISA architecture is used because it offers the ability to use up to 256 registers. The baseline optimization level is -O1, which includes classical optimizations. The compiler performs graph-coloring register allocation and does scheduling both before and after allocation. The number of registers available for allocation is passed as a command-line argument to the compiler. A complete description of the MIRV compiler, the types and order of optimizations, and a comparison against GCC is outside of the scope of this paper. This information can be found in our technical report [28].

We use MIRV to transform the C files of the SPECint95 and SPEC2000 benchmarks into the MIRV high-level intermediate representation (IR). The IR files for each module are then linked together, producing a single "linked MIRV" file for the benchmark. Once linking is done, the MIRV compiler can apply the optimizations that are used in this work to show that general purpose code can take advantage of a larger register set. The following subsections briefly describe these optimizations.

2.1 Register Promotion

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. 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 our compiler. 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 come from a register allocated to the value by the compiler.

Constant values that cannot fit into an instruction can also be promoted to a register in order to eliminate load-immediate instructions. Finally, indirect pointer references can also be promoted to a register. The register promoter in MIRV can promote global scalar variables, aliased local scalar variables, constants, and indirect pointer references (we call these *dereferences*). For the remainder of this paper, we will only consider loop regions.

2.2 Link-time Global Variable Register Allocation

One shortcoming of register promotion is that promoted global scalars are moved into and out of registers each time the loop (promotion region) is invoked. This causes unnecessary load and store instructions. Instead, MIRV can allocate global variables to registers for their entire lifetime. This eliminates the need to move the globals in and out of registers at region boundaries.

Post-link allocation can be used separately from register promotion, in which case it achieves much the same effect, in addition to avoiding the movement of values at region boundaries. It could also be applied after register promotion. In our experiments, we consider its application apart from promotion to isolate its benefit.

2.2.1 Allocating Global Variables

Instead of allocating globals into the set of local registers, we allocate them to a separate area of the register file. This is similar to the global registers in the SPARC and IA-64 architectures [22, 23]. After linking the program's intermediate representation into one file and doing the appropriate coalescing of global variable declarations and definitions, the compiler determines if the global's address is ever taken and annotates that into the global's declaration in the intermediate representation.

Next we run the MIRV SimpleScalar/PISA backend and select the global variables to allocate to registers. Because the usage of registers 0-31 is fixed by the System V ABI, MIRV allocates global variables to registers 32..32+N, where N is set by a compiler command-line option. A global variable must satisfy two conditions before it is considered a candidate for permanent allocation to a register. First, its address cannot be taken anywhere in the program. This ensures that we can allocate the variable to a register for its entire lifetime without being concerned about aliasing conditions. Second, the global cannot be an imported variable. The second condition ensures that we only allocate user variables to registers—we have opted not to modify the C library code and thus do not attempt to allocate imported library variables to registers.

MIRV uses three heuristics to determine which global scalars to put into registers. The first is a simple first-come-first served algorithm. It selects the first N allocatable global variables and allocates them. The second heuristic is based on a static frequency count estimate. It is determined by the following formula:

$$\text{static frequency count} = \sum_{\substack{\text{all uses and} \\ \text{definitions}}} (\text{loop nest level ? 4 << loop nest level : 1}) \qquad \textit{(Eq. 1)}$$

That is, a reference to a variable within the top level of a function is given weight 1. A use within a loop is given weight 4 << 1 = 8. A use within a doubly-nested loop is given weight 16, and so on. Candidates are sorted by frequency count with the highest frequency variables getting preference for registers.

The third and final heuristic is based on a dynamic frequency-of-use. The program is instrumented with counters which are used to keep track of the execution frequency of the block statements in the program. The program is then run on a training data set and the final counter values are used to compute how many times a given global variable was accessed. This data is more accurate than the static frequency count estimate.

We only consider the static frequency count heuristic in this paper since the first-come-first-serve is dependent on variable declaration order and the dynamic heuristic does not provide appreciable performance improvements when there are more than a few registers available for globals. We call the configurations based on the static heuristic "statX" where X is the number of registers in the machine. In our experiments, X minus 32 of those registers are permanently set aside for global variables; the original 32 registers are left to behave according to the ABI specification.

Once the candidates are chosen, they are allocated to a register by the MIRV backend. If the variable being allocated has an initial value, initialization code is inserted into the beginning of `main()`.

We do not use lifetime ranges to compact more global variables into the given number of registers. We think that trying to compact globals into fewer registers will not yield much appreciable benefit unless the machine has very few registers. Of course, if an existing ISA is used, then there are very few registers in general so it would be beneficial to try to compact them. Compacting globals by sharing registers requires movement of data between memory and registers. Register promotion does exactly that, so it is evident that these two optimizations are related.

2.2.2 Allocating Global Constants

Global arrays are responsible for a large portion of the computation in some programs. Each time a global array is referenced, its base address is formed and then used as part of an indexing calculation. This address formation (or materialization) takes two instructions on the PISA (MIPS IV) ISA because of the range limit on immediate constants. Even with loop invariant code motion that moves these address computations out of loops, many such instructions are executed.

We optimize this case by creating a new global scalar variable pointer which is initialized to point to the base of the global array. This pointer is then used in all array indexing computations instead of materializing the address. The new pointer variable is allocated to a global register for the entire program so that instead of materializing it at each point of use, a single register reference is all that is required. This reduces the number of load-immediate and add instructions that are executed. We call this optimization *address promotion*. Address promotion should not be performed if the pointer cannot be allocated to a register because rematerializing the address is probably cheaper than loading it from memory.

Loop invariant code motion can move many of these instructions out of local loops because they compute an invariant base address. Promoting the base address into a register removes all such instructions entirely instead of just moving them outside of the loops, which is important for functions which are themselves called within a loop. This optimization may seem wasteful at first because it is allocating a constant value (label) to a register. However, the frequency of these operations and the ease with which register allocation can memoize them make these operations prime candidates for optimization. Thus address promotion eliminates redundant computation by memoization. This effect is present in the go benchmark as will be mentioned later.

Local arrays cannot benefit from this optimization since their base address is already essentially in a register (at a constant offset relative to the frame pointer).

2.3 Inlining

MIRV inlines two kinds of functions: those that are called from one call site and those that are “small.” Functions that are invoked from one call site are called *singletons*. There are a surprising number of singleton functions in the SPEC benchmarks as shown in the fifth numeric column of Table 2. From 17% to 63% of functions are called from a single call site. These can be inlined profitably without causing code expansion because the original function can be removed after inlining.

Singleton functions are inlined in MIRV if the estimated register pressure is below a certain bound. Otherwise, if inlining were performed, there might be extra spill code introduced. For example, if there are 3 live registers at the call site and a maximum of 8 live registers in the singleton callee, then inlining will require 11 registers whereas before inlining, 8 registers was sufficient because of the prologue spills and epilogue reloads present in the singleton callee. As more registers are made available to the inliner, this quickly becomes a non-problem.[1] The bound is set to the total number of registers available to the compiler for the specific simulation (either 32, 64, or 256 in this paper). Small functions are those with fewer than 10 C statements. No register pressure measurements are examined for small function inlining.

Category	Benchmark	Total Functions	% Functions called from 0 or 1 call sites	
SPECint95	compress	30	23%	40%
	gcc	2,046	15%	26%
	go	383	0%	63%
	ijpeg	475	46%	29%
	li	382	49%	17%
	m88ksim	291	22%	30%
	perl	349	4%	26%
	vortex	959	29%	33%
SPECfp2000	ammp	206	23%	42%
	art	41	12%	34%
	equake	36	11%	53%
SPECint2000	gcc	?	?	?
	gzip	140	16%	31%
	mcf	39	8%	54%
	parser	349	6%	36%
	vortex	959	29%	33%
	vpr	301	5%	43%

Table 2. Functions in SPEC benchmarks that are called from 0 or 1 call sites (-O1 optimization). These are compile-time measurements and do not include functions never called dynamically.

Small functions are currently inlined until the code is expanded to a user-defined limit. The limit we use for this paper is 10%, which means that we do not allow inlining to expand the static code size by more than 10%. This is not a problem in practice. For example, inlining all the small functions in *go* at every call site only results in a 7% code expansion. Once a singleton is determined to be suitable for inlining based on our register pressure heuristic, the function is inlined without regard for code expansion because the original copy of the function will be deleted after inlining. The net code expansion will be *about* zero.

A potential negative effect of inlining is that of poor code layout. Inlining a function can make performance worse if functions are aligned in such a way that they conflict with each other

1. Some have disagreed with our approach. It is true that there are no more live *values* before inlining than after. However, combining the allocation candidates of the two functions together can cause spill code which performs worse than the original prologue/epilogue code. Our heuristic for this is to look at register pressure to determine if more spilling code will be introduced.

or if the working set size of the program has been made large enough that it does not fit into the cache. These issues are not examined in this paper but are described elswhere in the literature [42].

3. Simulation Environment

All simulations were done using the SimpleScalar 3.0 simulation toolset [24]. 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 [10]. Registers 32-255 are used either as additional registers for global variables OR additional registers for local caller/callee save variables.

We ran variants of the SPEC training inputs in order to keep simulation time reasonable. Table 3 shows the data sets used for these experiments. Our baseline timing simulator is the default sim-outorder configuration; it is detailed in Table 4. Detailed information is available elsewhere [28].

Category	Benchmark	Input
SPECint95	`compress`	30000 q 2131
	`gcc`	regclass.i
	`go`	9 9 null.in
	`ijpeg`	specmun.ppm, -compression.quality 25, other args as in training run
	`li`	boyer.lsp (reference input)
	`m88ksim`	ctl.lit (train input)
	`perl`	jumble.pl < jumble.in, dictionary up to ’angeline’ only
	`vortex`	250 parts and 1000 people, other variables scaled accordingly
SPECfp2000	`ammp`	test input modified so that numstp is 1, short.tether has the first 162 lines, and all.init.ammp has a subset of the statements in the original
	`art`	-scanfile c756hel.in -trainfile1 a10.img -stride 2 -startx 134 -starty 220 -endx 139 -endy 225 -objects 1 (test input)
	`equake`	< inp.in (test input)
SPECint2000	`gcc`	first 1442 lines of test input cccp.i
	`gzip`	input.compressed 1 (test input)
	`mcf`	inp.in (test input)
	`parser`	2.1.dict -batch < test.in
	`vortex`	250 parts and 1000 people, other variables scaled accordingly
	`vpr`	net.in arch.in place.in route.out -nodisp -route_only -route_chan_width 15 -pres_fac_mult 2 -acc_fac 1 -first_iter_pres_fac 4 -initial_pres_fac 8 (test input)

Table 3. Description of benchmark inputs.

SimpleScalar parameter	Value
fetch queue size	4
fetch speed	1
decode, width	4
issue width	4 out-of-order, wrong-path issue included
commit width	4
RUU (window) size	16
LSQ	8
FUs	alu:4, mult:1, memport:2, fpalu:4, fpmult:1
branch prediction	2048-entry bimod, 4-way 512-set BTB, 3 cycle extra mispredict latency, non-spec update, 8-entry RAS
L1 D-cache	128-set, 4-way, 32-byte lines, LRU, 1-cycle hit, 16KB
L1 I-cache	512-set, direct-mapped 32-byte line, LRU, 1-cycle hit, 16KB
L2 unified cache	1024-set, 4-way, 64-byte line, 6-cycle hit, 256KB
memory latency	18 for first chunk, 2 thereafter
memory width	8 bytes
Instruction TLB	16-way, 4096 byte page, 4-way, LRU, 30 cycle miss penalty
Data TLB	32-way, 4096 byte page, 4-way, LRU, 30 cycle miss penalty

Table 4. Simulation parameters for sim-outorder (the defaults).

4. Experiment Results

The results of our experiments are presented in the following subsections. The baseline in all cases is MIRV -O1 with 32 registers. In each set of graphs in the following subsections, the first two bars are the MIRV -O1 results with 64, and 256 registers. The graphs show simulated cycles for each of the SPECint and SPEC2000 benchmarks described in the previous section.[1] Each figure shows the performance improvement of the respective optimization compared to -O1 with 32 registers so that it is clear what the individual optimization is doing to performance. Section 4.1 shows -O1 versus -O1 with register promotion. Section 4.2 shows -O1 versus -O1 with link-time allocation. Section 4.3 shows -O1 versus -O1 with inlining. Section 4.4 shows the combination of all 4 sets of optimizations. The final subsection, Section 4.5, shows some information on the cache effects of the optimizations.

Note that -O1 performance can actually become worse when the number of local registers is increased because of extra spill code in the prologue and epilogue to handle callee save registers.

4.1 Register Promotion

Our first set of results shows the performance increases due to register promotion. In addition to the two -O1 bars, each graph has three bars for register promotion, for 32, 64, and 256 reg-

1. Note to the reviewer: There are several empty entries in the graphs and tables that follow. These simulations will be completed for the final paper.

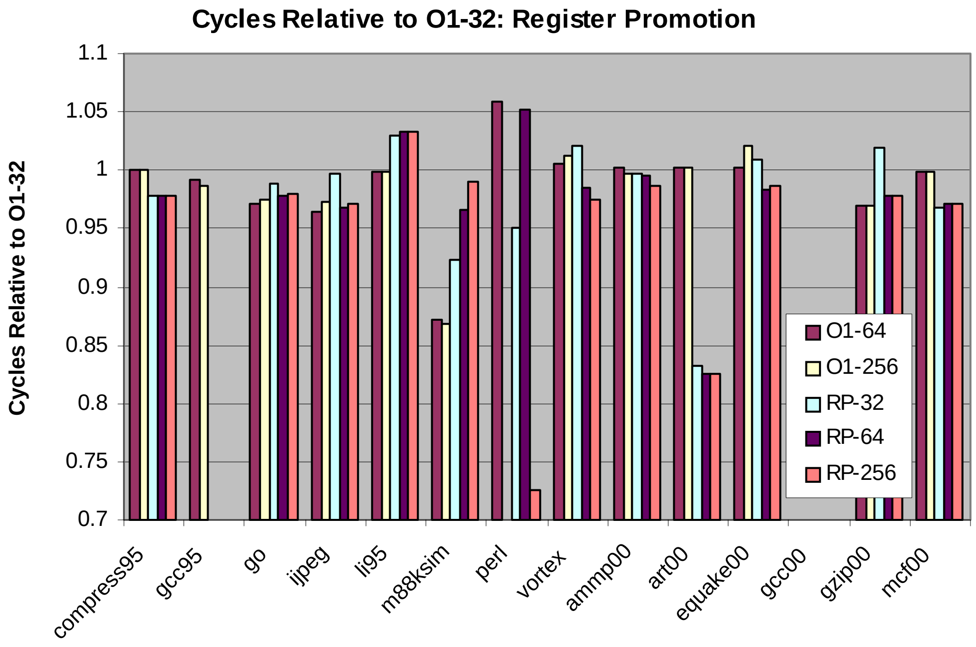


Figure 1. Performance of register promotion.

isters. Register promotion improves performance from 5% to 15% on some benchmarks. Other benchmarks, most notably `li`, actually get worse with register promotion. This is due to spilling caused by the promotion.

The art benchmark was helped a great deal by register promotion. Examination of the benchmark source reveals five integer global variables which are used heavily in loops for comparison and array indexing operations. In addition, there are several global pointers into arrays and other data structures that are promoted. Promotion provides a significant performance win by reducing the number of loads by 46%. A small reduction in store instructions (2.3%) indicates that these variables are not modified often by the program. These variables may not require full promotion but can use a weaker form that only considers load instructions.

As the number of registers is increased for promotion, results generally improve slightly. Some benchmarks, such as `m88ksim` and `vortex`, show a performance degradation. This is due to the extra register save and restore operations at function call boundaries. This overhead outweighs the benefit of promotion. While the reduction in the number of memory operations for `go`, `ijpeg` and `equake` is significant (10%-15% of each category), performance only improves by about 5%-8%.

4.2 Link-time Global Variable Register Allocation

Figure 2 shows the performance results when link time allocation is applied. The results reported are for 32 and 224 registers specifically set aside for global variables. Because of precompiled libraries, we could not permanently allocate any global variables to registers 0-31. This optimization reduces the number of load instructions anywhere from 0% to 50%. Some benchmarks, such as `ijpeg`, `vortex` and `equake` see little or no reduction in memory operations. For those benchmarks that do see a significant reduction, performance improves up to 15%.

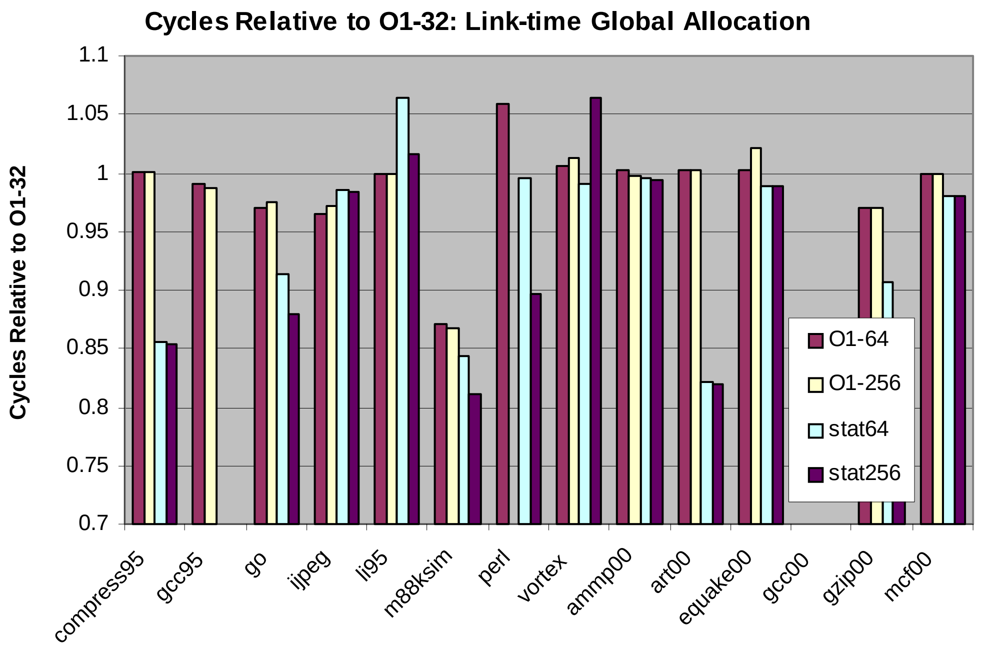


Figure 2. Performance of post-link allocation.

Performance actually degrades in some cases when the number of registers set aside for global variables is increased from 32 to 224 (the stat64 and stat256 configurations, respectively). This is not intuitive and indeed is due to another effect: code layout. For example, for `vortex` compiled with -O1 optimizations, the instruction cache miss rate is 7%. For the same program compiled with link-time allocation, the instruction cache miss rate is over 9%.

The effect of allocating base addresses is significant for some benchmarks, particularly `go`, which accesses a lot of global arrays. In this case, loads and stores are reduced as well as address arithmetic. Essentially, the base address has been memoized into a register. This reduces the amount of redundancy in computation that is seen during program execution.

`Compress` is not a particularly good benchmark to study in this context because most of the benefit of link-time allocation comes from two global pointer variables which index into I/O buffers. These are used in two small helper functions called `getbyte()` and `putbyte()`, which were introduced when the program was made into a benchmark. Link-time allocation helps to remove this inefficiency and eliminates almost 60% of the load instructions and nearly 50% of the store instructions in that benchmark.

Many of the benchmarks show that adding registers for global variables is much more important than adding registers for local variables. For example, the `compress` benchmark does not benefit at all at the -O1 optimization level with 64 or 256 registers. However, setting aside 16 registers for globals vastly improves the performance. This is true for `go`, `m88ksim`, `art`, `gzip`, and `vpr`, and to a lesser extent some of the other benchmarks.

Adding registers for locals does impact performance by 5% or more for `go`, `m88ksim`, and `equake`. Sometimes variables are aliased for a small initial portion of their overall lifetime and could be allocated permanently to a register afterwards. For example, we noticed that in the `compress` benchmark, the global `seedi` integer could be allocated to a register after it is ini-

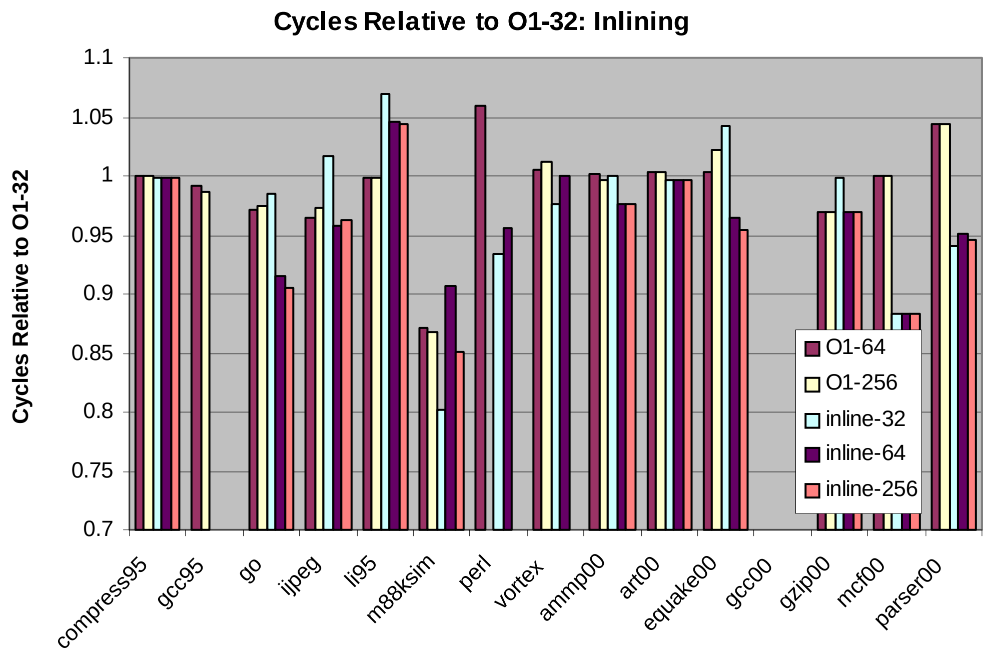


Figure 3. Performance of -O3 (inlining).

tialized through a pointer. However, this yielded only a marginal performance improvement, so we did not investigate this further.

Link-time allocation is much more effective than register promotion at improving performance. This points out the importance of permanently allocating globals into registers. Promotion can achieve some of this benefit by removing loads and stores from loop bodies, but it still shuttles data back and forth to memory at region boundaries. If the promotion itself occurs within a function that is called from a loop, the promotion and demotion are still within a loop. Link-time allocation avoids all such unnecessary data movement.

4.3 Inlining

Figure 3 shows the experiment results when inlining is applied to the benchmarks. The results suggest that with inlining, register pressure increases significantly. This is particularly true for `go` and `equake`, as adding more registers results in performance improvements. Other benchmarks like `compress` and `art` do not make use of extra registers.

Inlining increases register pressure in two ways. First, the local variables for the caller and callee are combined and allocated in a single graph coloring phase. Both sets of variables compete for the same set of registers. The function call overhead that takes care of shuttling these two sets of variables in and out of registers has been eliminated. Second, inlining also increases the scope of optimization. Because the optimizer has an increased amount of program context, it can perform more transformations. As a general rule, transformations increase register pressure by adding temporaries and extending value lifetimes. As a result, inlining increases register pressure indirectly by opening up more opportunities for program transformation.

Without extra registers, `gzip` perform worse when inlining is performed. This is mainly due to register spilling. Inlining introduces too many simultaneously live variables which must subsequently be spilled. Often this spill code is more expensive than the function call register save and restore because it occurs in multiple places (everywhere the variable is referenced).

4.4 Combined Results

The graphs in Figure 4 show what happens when we combine the optimizations of the last three sections: register promotion, link-time allocation, and inlining. The third bar shown is what we call “best16”. This includes all the aforementioned optimizations. The ‘16’ in the name indicates that we allowed 16 extra registers for local variables (for promotion and inlining) and 16 extra registers for global link-time allocation, for a total of 32 extra registers. Similarly for best112, where a total of 224 extra registers are allowed, half for local variables and half for link-time allocation. The number of cycles required to execute the program is generally by 10% to 20% in almost every case.

In particular for the `go` benchmark, when best112 is applied, the execution time and number of dynamic instructions has been reduced by 20% over the baseline -O1 optimization level. The number of loads has been reduced by over 30% and the number of stores by over 50%.[1]

Each of the optimizations has increased register usage and takes advantage of extra registers. For example, the extra 32 registers available in the best16 configuration demonstrate significant performance improvements over -O1 optimization. In some cases, like `compress`, `go`, `m88ksim`, and `vpr`, the best112 configuration (a total of 224 extra registers) shows 3% to 7% performance improvement over the best16 configuration. This configuration serves to illustrate the point that more than 32 registers are required to achieve the best possible performance in our compilation environment. Ideally, an architecture should have at least 64 registers to allow for the maximum benefit for register promotion, global variable allocation, and inlining. With more aggressive alias analysis, optimizations, and scheduling, we are confident that the compiler could use even more registers than reported here.

4.5 Cache Effects

The number of data cache accesses for the best configurations is shown in Figure 5. The graph shows the number of data accesses made to the cache. This is an important metric because it shows the effectiveness of register allocation techiques in shielding the memory hierarchy from traffic. The combination of optimizations produce substantial reductions in cache accesses for `go`, `art`, `gzip`, `vpr`, and to a lesser extent the other benchmarks.

There are several potential benefits of reducing the number of cache accesses. For example, fewer demand accesses to the cache by the processor might allow the memory hierarchy to do more effective prefetching since it has more time to query the cache to see if prefetch candidates already reside in the cache. Another important benefit is power savings.

In Table 5, we show the reductions in data cache hits and misses for the best112 configurations. The number of cache *misses* is not reduced substantially for any benchmark. On the other hand, the number of cache *hits* has been reduced in most cases, sometimes as much as 30% to 60%. In effect, the compiler has moved items from the data cache into the register file: those items that were already hits in the data cache are now hits in the register file. This is a good optimization because the register file will likely be much faster to access than the cache, in addition to

1. Due to space limitations we cannot show the graphs of the counts of instructions, loads, and stores.

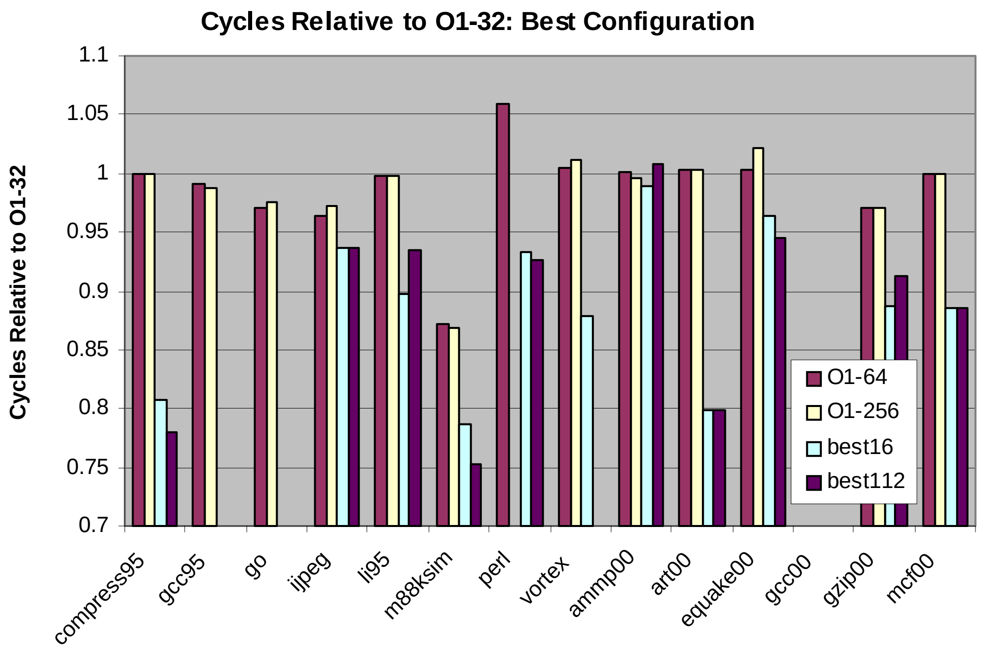


Figure 4. Performance of the “best” configurations.

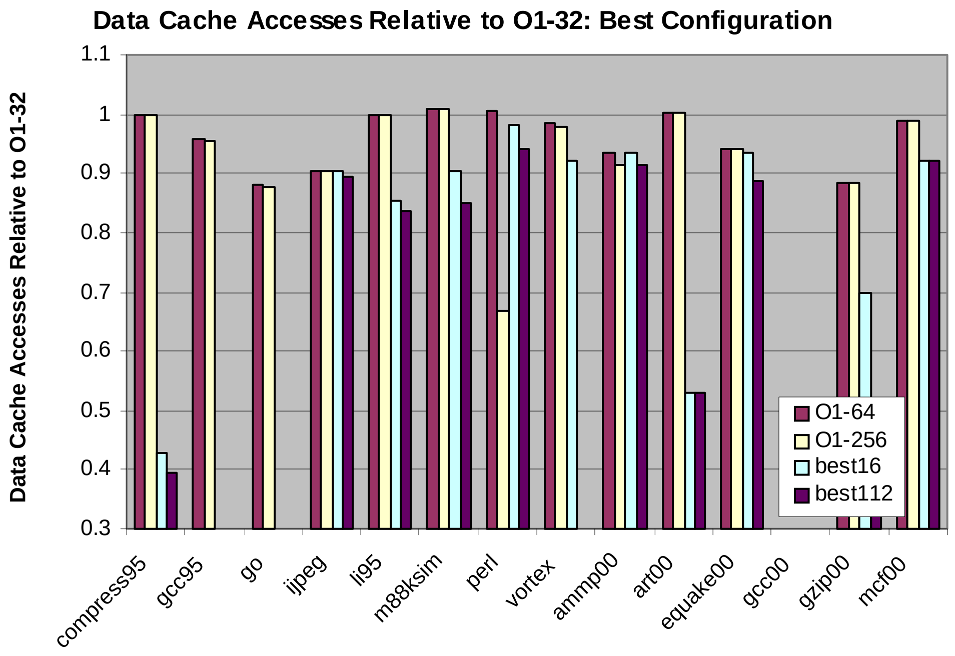


Figure 5. Data cache accesses in O1 and best various configurations.

having a much higher bandwidth because of the larger number of ports typically available on a register file.

This is an important illustration of how it is not always most important to optimize the primary “bottleneck” in the system. In the case of memory accesses, it is intuitive to focus on reducing the cache miss latency or miss ratio because misses are expensive in terms of cycles. Such is the focus of work on prefetching and smarter cache replacement policies. Instead, our research optimizes the common case, cache hits, and effectively reduces the cache hit time by eliminating the cache access altogether. From our data, it seems that techniques which reduce cache miss latency or miss ratio can be applied on top of our optimizations to further improve performance. Processors with larger caches which have two or more cycle access time would benefit even more from optimizing the cache hit time this way.

For example, if a load-use instruction pair executes in back-to-back cycles on a conventional microarchitecture, changing the load into a register access can improve performance by eliminating the load instruction (one cycle) and also allowing the consumer instruction to start one cycle earlier, for a savings of two cycles. Compiler scheduling may reduce this benefit by separating the load from its use. On the other hand, the situation may be worse if the load has to wait for a free port on the data cache. Allocating the variable to a register instead solves two problems at once: it reduces path length by elimination of load and store instructions, and it solves the bandwidth problem because the register file will typically be highly ported.

Category	Benchmark	Misses in best112 Relative to Baseline	Hits in best112 Relative to Baseline
SPECint95	compress	100%	36%
	gcc	?	?
	go	96%	60%
	ijpeg	99%	89%
	li	102%	84%
	m88ksim	101%	84%
	perl	96%	94%
	vortex	?	?
SPECfp2000	ammp	?	?
	art	101%	38%
	equake	100%	89%
SPECint2000	gcc	?	?
	gzip	100%	44%
	mcf	100%	91%
	parser	100%	66%
	vortex	?	?
	vpr	99%	73%

Table 5. Data cache misses of the best configurations relative to O1-32.

In Figure 6 we show how inlining and our "best" optimization impacts instruction cache performance. Instruction cache performance is not significantly impacted by our optimizations in most cases. Inlining often results in a better cache hit rate than -O1 optimization. The exceptions are `li95`, `m88ksim`, and `vortex`, where the instruction cache miss rate increases by 1% or more. When the "best" optimizations are turned on, the situation is remedied because the best configuration does not do inlining as aggressively since some registers are reserved for link-time

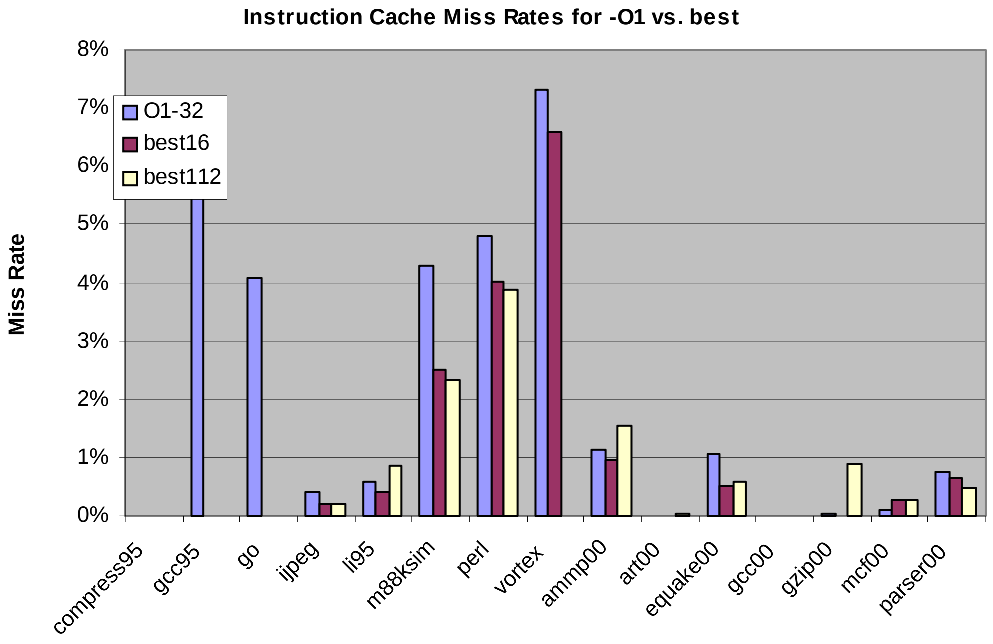


Figure 6. Instruction cache miss rates for the -O1-32 and best configurations.

global allocation and thus are not available for inlining. Alternative code layouts were not considered, although our experience indicates this could improve performance by several percent.

5. Theoretical Register Requirements

This subsection examines several measurements to determine what the most appropriate number of registers would be for an architecture that processes general purpose code.

5.1 Variable Count Arguments

The first measurement we show is a simple counting argument pointing out that there are many candidates for register allocation. Additional intraprocedural optimizations, such as our -O2 configuration, generally increase the number of candidates for allocation. Inlining in the -O3 column shows a significant reduction in the number of candidates. This is because inlining eliminates parameter variables and allows propagation optimizations. At the same time, inlining increases the number of long-lived variables that are allocated to registers.

Table 7 shows the number and types of global variables in the benchmarks. The fourth column in the table shows the percentage of global variables that are not considered as candidates to be placed into a register. This percentage is computed by linking together the benchmark at the MIRV IR level and processing the MIRV IR with a filter that determines whether the global variable's address is ever taken. If it is, then we say the variable cannot go into a register (even though it may be able to be enregistered for parts of its lifetime).

The integer benchmarks fall into two categories. For the majority of the benchmarks, most global variables could be allocated for their entire lifetime. Two interesting cases are `go` and

Category	Benchmark	-O1 Local Variables	-O2 Local Variables	-O3 Local Variables	-O4 Local Variables
SPECint95	compress	117	120	49	49
	gcc	24842	?	?	?
	go	3435	3490	1948	1948
	ijpeg	3321	3330	2251	2251
	li	1703	1713	1506	1506
	m88ksim	1859	1887	1298	1298
	perl	4062	4125	3764	3764
	vortex	13690	13778	7859	7860
SPECfp2000	ammp	2332	2364	1855	1855
	art	154	254	174	174
	equake	191	218	84	84
SPECint2000	gcc	?	?	?	?
	gzip	690	714	398	398
	mcf	218	218	76	76
	parser	1924	2001	1296	1296
	vortex	13690	13777	?	7860
	vpr	2244	2388	1639	1639

Table 6. Statistics on local variables under different optimizations levels.

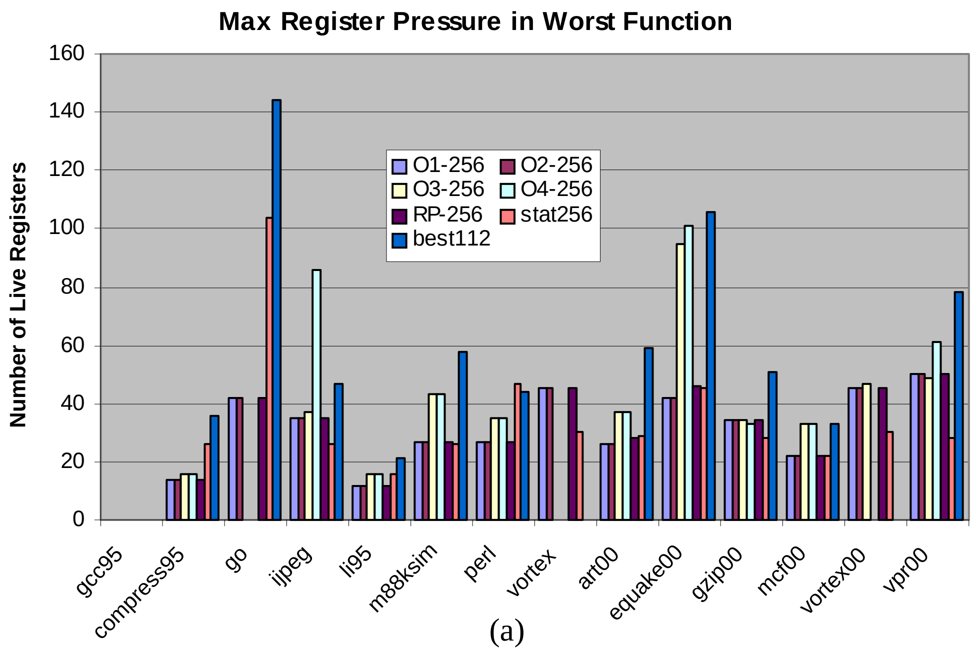
`vortex`, where there is a high percentage of globals that are aliased. In both benchmarks, most of the variables in question are passed by address to some function which changes the global's value. The number of call sites where this happens is usually fairly small for any given variable. In vortex, a heavily used variable called Theory is modified through a pointer in memory management code. It appears that this use through the pointer is for initialization only (so register promotion could promote the global to a register after initialization is over). The floating point benchmark equake shows similar behavior.

Overall, there are a significant number of global variables in these benchmarks which may benefit from register allocation. The global aggregates (shown in the fifth column of Table 7) are the number of arrays and structures found at the global scope. The base addresses of these structures are candidates for register allocation as well.

Figure 7 shows two more counting arguments. Figure 7(a) shows an estimate of the maximum register pressure in the "worst" function in each of the benchmarks. Typically under 40 registers are required by such functions, but occasionally the number climbs well above 50, particularly under the -O4 optimization level. A similar situation is seen in Figure 7(b), where the maximum live at any function call site is shown. For example, there are 40 simultaneously live values at the "worst" call site in the go benchmark under -O1 optimization.

5.2 Instruction Count Argument

The last section showed measurements of the "worst" functions in the benchmarks but said nothing about the average register requirements. This section shows data on the register requirements of the `go` benchmark over four different optimization levels in Figure 8.[1] The X-



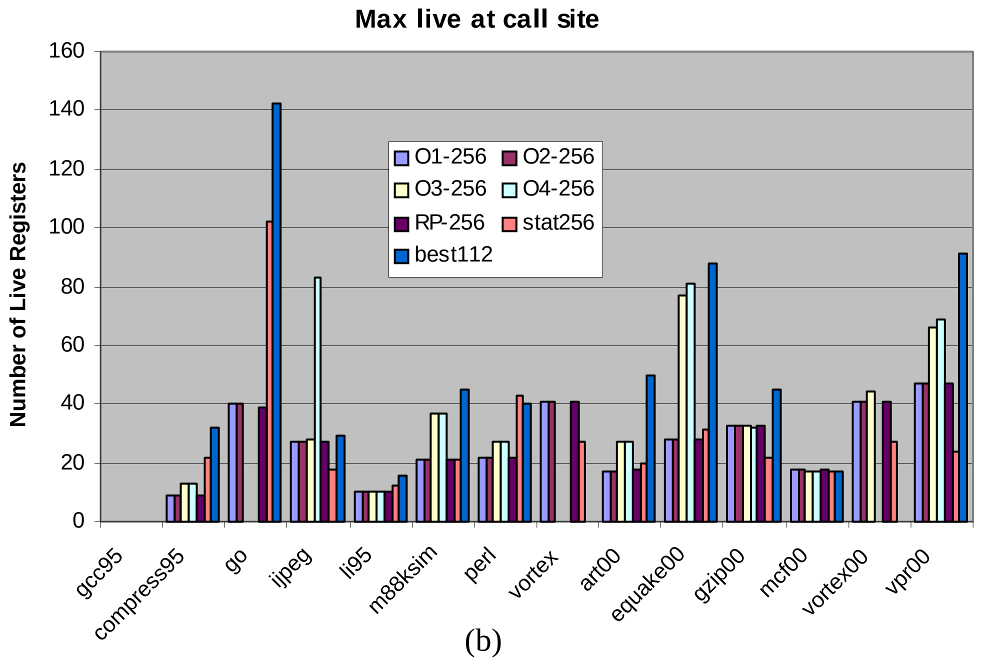


Figure 7. Other estimates of register pressure in the benchmarks. Note the change in y-axis scale.

axis of each graph is the number of registers in a register bin. The Y-axis shows the percentage of dynamic instructions that execute in functions which required that number of registers. For example, in Figure 8(a), 36% of the instructions come from functions where 13-16 registers are desired by the function. This does not count special registers such as the frame pointer, stack pointer, global pointer, zero, and assembler and kernel reserved registers. We will assume that there are 8 such reserved registers for the sake of argument. For these instructions, a register file size of 13+8=21 to 16+8=24 is the right number.

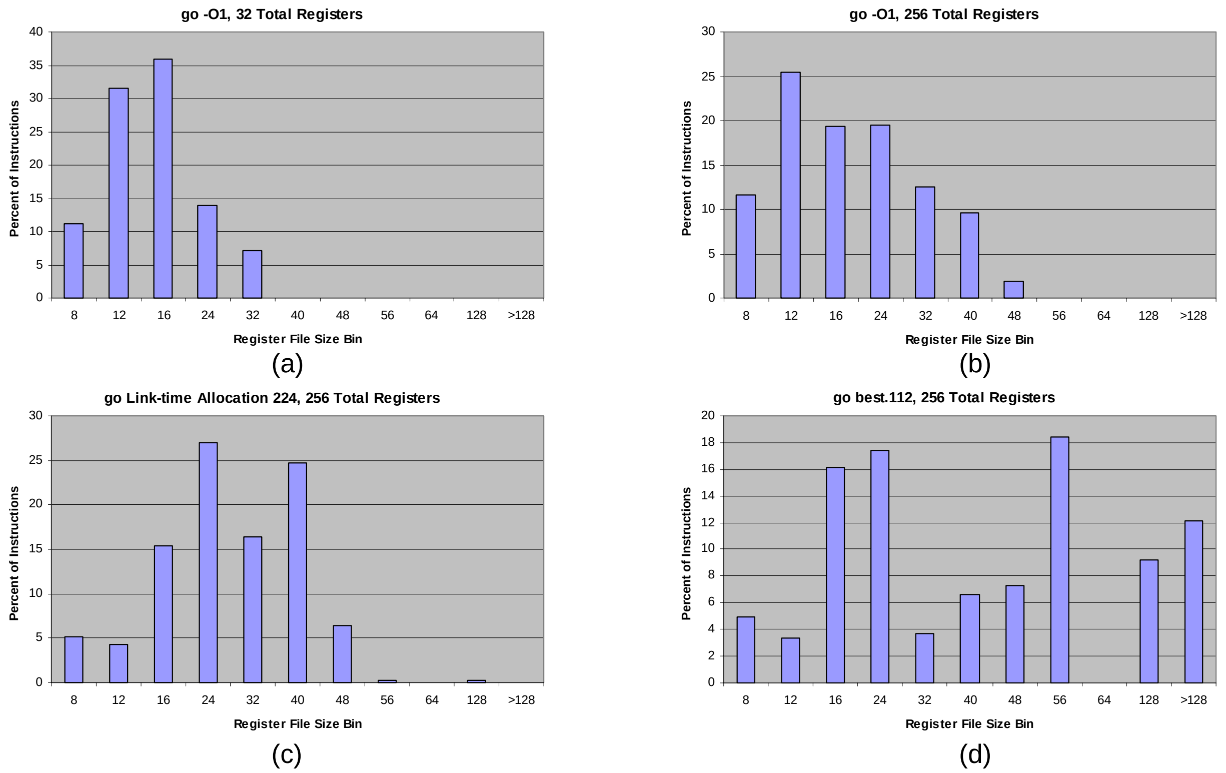


Figure 8. Comparison of register requirements for the go benchmark across several optimization levels. The X axis is a register file size bin, where 8 represents 0-8 registers, 12 represents 9-12, 16 represents 13-16, etc.

These numbers are produced by annotating the each function with a *maxlive* attribute which tells how many registers are simultaneously live in the function. This *maxlive* number comes directly from the live variable analysis phase of the backend's register allocator. Its final value is computed immediately before the assembly code is generated, so it includes the effect of spilling code to reduce register pressure for a given register configuration. This information is used along with a profile that tells how many instructions were executed from each function. Note that the *maxlive* number is not how many colors were required to color the graph, but instead is the maximum number of values that are simultaneously live. It would take at least this many registers to color the graph without any additional spilling code. Therefore, *maxlive* is a lower bound on the number of registers required for maximum performance.

The executable that was used to produce Figure 8(a) was compiled assuming 32 registers were available on the machine. Figure 8(b) shows that when this assumption is changed, to 256 registers, the profile looks significantly different. In this case, 32 registers are required over 10% of the time and 40 registers another 10%. This points out that when the compiler is limited to 32 registers, the register-usage profile appears to be very conservative (because of the spilling code inserted during register allocation). Only when the compiler is loosed from the restriction of a small register file can a true measurement of register requirements be made.

Thus far, we have shown that with traditional optimizations such as CSE and LICM, the MIRV compiler could easily and profitably use a machine with 40 or more registers.

Figure 8(c) shows what happens when link-time allocation is applied to go. Even more registers are required–25% of the time, a register file size of 32-40 is appropriate. Figure 8(d) shows what happens when all of the optimizations in this paper are turned on. A large shift to the

Category	Benchmark	Global Variables	Un-allocatable Globals	Global Aggregates
SPECint95	compress	27	3.7%	14
	gcc	1018	7.1%	298
	go	80	15%	200
	ijpeg	29	3.4%	35
	li	79	0.0%	3
	m88ksim	106	1.9%	8.7
	perl	208	2.9%	42
	vortex	515	41.7%	105
SPECfp2000	ammp	48	6.3%	4
	art	28	7.1%	4
	equake	34	35.3%	4
SPECint2000	gcc	?	?	?
	gzip	99	2.0%	47
	mcf	6	0.0%	3
	parser	83	22.9%	92
	vortex	515	41.7%	105
	vpr	96	6.3%	10

Table 7. Statistics on global variables under -O1 optimization.

right of the graph occurs, and many instructions are executed in a context that requires 56 or more registers. From this data it is clear that current compiler technology can utilize a large number of registers. At least 64 registers are required for the benchmarks shown here.

5.3 Cross-Function Argument

The previous sections considered register allocation in the conventional sense, where the allocator is limited to candidates within a single function. A running program, however, typically has several functions “in progress” on the call stack. Each of these has some number of live values which should be summed to find out the best number of registers for the architecture. Since this is a hard number to compute, in this section we provide a couple of estimates to show the utility of a cross-function allocation scheme.

The first is simply the call graph depth, shown in Figure 9(a). This graph shows the maximum call depth reached by the benchmark. Most benchmarks only traverse 10 to 20 levels deep in the call graph. This supports the classic argument for register windows, which is that windows will allow the machine to capture most of the dynamic call stack without stack overflows and underflows, which necessitate memory operations to manage the stack.

The graph in Figure 9(b) shows our estimate of the register requirement of each of the benchmarks. This estimate is produced by applying the following formula to the dynamic call graph:

$$CGmaxLive(\text{caller}) = MAX\begin{pmatrix} CGmaxLive(\text{callee}) + maxLiveAtCallSite(\text{caller}), \\ maxLive(\text{caller}) \end{pmatrix} \quad \textit{(Eq. 2)}$$

The formula is applied during a run of the benchmark where the call graph is annotated with two pieces of information. At each call site, we have placed an integer for the number of live variables at the call site minus the number of parameters. We call this number *maxLiveAtCallSite*(caller) and do not include parameters under the assumption that *CGmaxLive*(callee) will include those. The other number is annotated into the function itself and is the most number of live variables at any point of the caller function. We call this *maxLive*(caller).

As an example of how this formula works, consider the following situation. Function A has *maxLive*(caller) = 11 and *maxLiveAtCallSite*(caller) = 3. If we call a function B whose *CGmaxLive* is 6, then the simulator says that for this sequence of function calls, max(3+6, 11) = 11 registers is plenty. If B's *CGmaxLive* is 9, then the 12 registers is about right.

The graph in Figure 9(b) shows this estimate across our benchmark. A very large number of registers is suggested. When all optimizations are turned on, the "best" number of registers is estimated to be about 100 or more for most benchmaks. Some require several hundred registers. The outlying point is the `li` benchmark. This is a recursive descent program has a very complicated dynamic call graph for which it would be difficult to keep all values in registers.

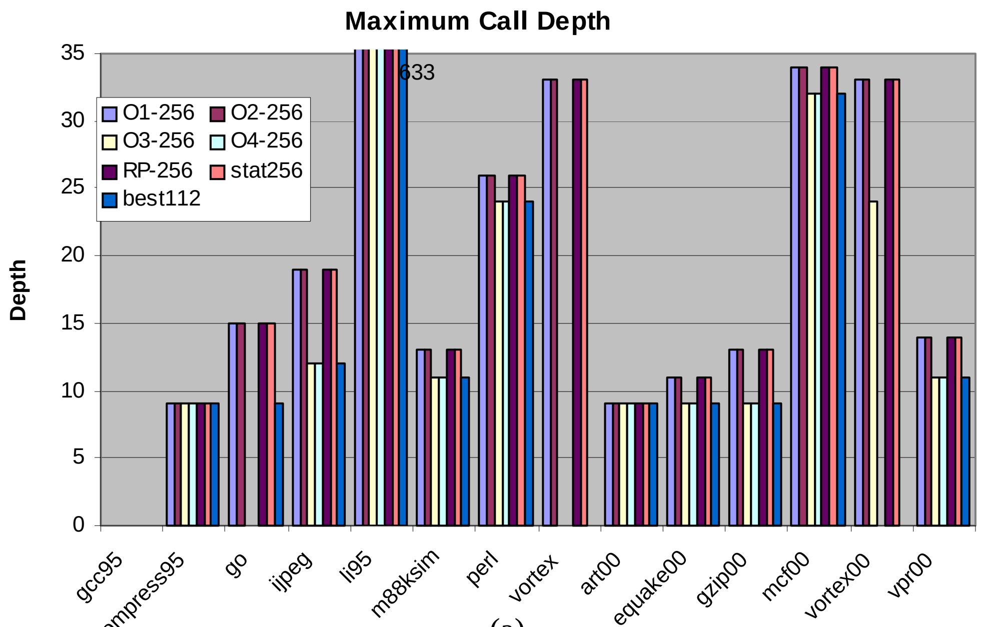
6. Related Work

6.1 Intra-Procedural Allocation

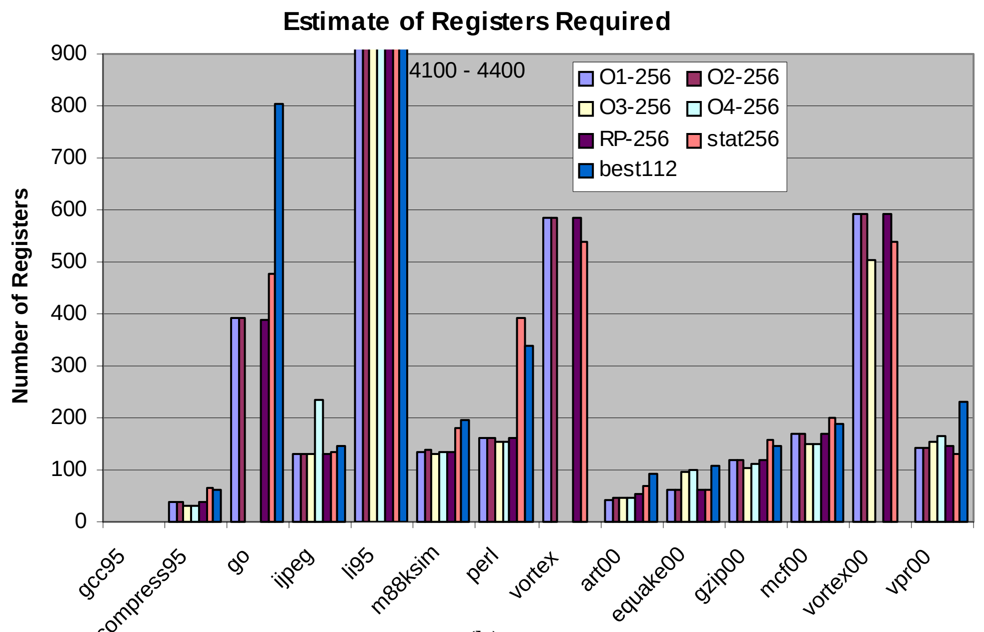
Mahlke et. al. examined the trade-off between architected register file size and multiple instruction issue per cycle [1]. They found that aggressive optimizations such as loop unrolling and induction variable expansion are effective for machines with large, moderate, and even small register files, but that for small register files, the benefits are limited because of the excessive spill code introduced. Additional instruction issue slots can ameliorate this by effectively hiding some of the spill code. This work noticed little speedup or reduction in memory traffic for register files larger than about 24 allocatable registers (often fewer registers were required). The compiler used in the study was not able to take advantage of more registers because of the conventional application binary interface [10] and lack of optimizations in the compiler such as those described here.

Benitez and Davidson study the effect of register deprivation on compiler optimizations [2]. The technique proposed in that paper is to study the effect of a series of probes where the compiler is deprived of more registers in each successive probe. This allows the compiler writer to examine the effect of optimizations on register utilization. The paper suggests that current technology cannot utilize a large number of registers.

The large body of research into optimizing spill code indicates the prevalence of spill operations in modern programs and highlights the importance of having a sufficient number of registers. For example, Cooper and Harvey propose the compiler-controlled memory which combines hardware and software modifications to attempt to reduce the cost of spill code [11]. The hardware mechanism proposed is a small compiler-controlled memory (CCM) that is used as a secondary register file for spill code. The compiler allocates spill locations in the CCM either by a post-pass allocator that runs after a standard graph-coloring allocator, or by an integrated allocator that runs with the spill code insertion part of the Chaitin-Briggs register allocator. A number of routines in SPEC95, SPEC89, and various numerical algorithms were found to require significant spill code, but rarely were more than 250 additional storage locations required to house the spilled variables. Potential performance improvements were on the order of 10-15% on a processor with



(a)



(b)

Figure 9. An estimate of register pressure based on the dynamic call graph.

a 2-cycle memory access time, but effects from larger traditional caches, write buffers, victim caches, or prefetching were not modeled. These results show the potential benefit of providing a large number of architected registers–not only simplifying the compilation process, but also reducing spill code and memory traffic.

6.2 Inter-Procedural Allocation

Wall described a link-time approach that allocates local and global variables as well as labels into registers [17]. The linker can optionally do a global register allocation pass which builds the call graph of the program and groups functions based on whether they can be simultaneously active or not. For functions that can never be simultaneously active, their local variables can be allocated to the same registers without needing to save and restore them across call boundaries. Functions that can be simultaneously active have their locals allocated to separate regions of the register file, while functions that cannot be live at the same time can share registers. Global variables as well as constants such as array base addresses take part in the link-time allocation as well. Once a global is allocated to a register, it resides in that register for its entire lifetime. The most frequently used variables are selected to be allocated into registers. This is done either with static estimates or with profiling data. Once it is determined which variables will be allocated to registers, the linker rewrites the object code under the direction of compiler-provided annotations. Compared to the baseline—the original code produced by the compiler which has only the 8 temporary registers—the link-time technique improves executable speed by 10 to 25% on a configuration with 52 registers. Link-time allocation (without dataflow analysis or coloring at all) does better than traditional graph coloring such as is done in a Briggs-Chaitin allocator [19, 20, 21]. Graph coloring can be added in Wall's system in order to allow variables which do not conflict within a procedure to share a register. This local coloring is especially important when there are few registers in the machine but does not improve performance much with 52 processor registers.

Our post-link allocation strategy is closely modeled after an earlier one described by Wall. First, we compare our results against a more aggressive set of baseline compiler optimizations than the previous work. Second, we only consider global variables because we do not have the ability to modify any library code as in the previous study; local variables are allocated with traditional graph coloriong techniques within the constraints of the application binary interface. Third, we examine the benefits of global variable register allocation over a wider range of machine configurations; in particular, the number of registers is a variable in this study but was fixed at 52 in the previous work. Fourth, we choose a simpler frequency-based allocation scheme that does not build the program's control flow graph. Finally, our link-time allocation transforms the high and low-level IRs of the MIRV compiler instead of working at a binary level.

The Sparc architecture's register windows are a hybrid register/memory architecture intended to optimize function calls [3, 22]. Each subroutine gets a new window of registers, with some overlap between adjacent register windows for the passing of function arguments.

Because Wall's link-time allocator splits the registers across active functions, it is a software approximation of the behavior of register windows [18]. Wall concludes that using profile information to guide the link-time allocator always produces better performance than register windows. Variable-sized windows are slightly better than fixed-size windows, and it is more important to add a set of registers for global variables than adding more registers for the window buffer. This points out the importance of allocating global variables to registers.

Chow describes a similar interprocedural approach to register allocation where the available functions are visited in depth-first order and allocation made from the bottom of the call graph towards the top [41]. He calls this the "depth first interprocedural register allocator." In the ideal case, if function A calls function B, then B will have been allocated before A is visited. Then when A is being allocated, it can see the registers that are in use by B and avoid using them. This allows the compiler to omit prologue and epilogue code to manage callee save registers. In effect, this pushes the callee save spills upwards in the call graph, eliminating them from the functions closer to the leaves. Like Wall's allocator, this approximates the behavior of hardware register windows.

6.3 Register Promotion

Cooper and Lu examined promotion over loop regions. Their algorithm is most similar to what is presented here [14], though our alias analysis is a form of MOD/REF analysis which is somewhat simpler than used in any of the previous work. Sastry and Ju examine promotion over arbitrary program intervals using an SSA representation [15]. Lo and Chow use a variant of partial redundancy elimination to remove unnecessary loads and stores over any program region [16].

All of the previous work shows substantial reductions in the number of dynamic load instructions executed and varying reduction in the number of stores eliminated. Typically 0-25% of loads are removed and up to 40% of stores are removed, depending on the program. Explicit load/store instructions are needed for register promotion because the global variables share registers which are local in scope. Cooper and Lu's results indicate that the main benefit of promotion comes from removing store operations. The other previous work shows that loads are improved more than stores. This disparity is primarily because the baseline compiler optimizations are not reported in any detail in any of the papers, which makes it difficult to perform a fair comparison. Two of the papers only counted the improvement compared to the total number of *scalar* load and store instructions [15, 16]. While this shows the improvement of one aspect of the memory bottleneck, it does not show how effective promotion is at removing overall memory operations. One of the papers did not consider the effect of spilling because it simulated with an infinite symbolic register set before register allocation [16]. Spill code is inserted by the allocator if too much promotion occurs in a region. Each of these papers reported on the improvement in dynamic load and store counts after their promotion algorithm was applied. They did not report on results from a cycle simulator as we do here.

7. Conclusions

This work first demonstrated the lack of consensus regarding register set size in research and commercial products. Research has been contradictory, ranging from statements like "compiler technology cannot utilize a large register file," to the other extreme that register sets of 100 or more registers are necessary for best performance. For commercial parts, register set sizes from 8 to 128 are seen in both the embedded and high performance realms.

While this is the case, the majority of high-volume processors have 32 or fewer registers. This can be attributed to a number of causes, primarily related to human programming style and the variables that are considered as candidates for allocation.

We have demonstrated in this paper that existing advanced compiler optimizations can easily make use of 64 or more registers in general purpose codes. A reasonably aggressive optimizing compiler can make use of registers by allocating global variables, promoting aliased variables (in regions where it is accessed through one name only), as well as inlining, to improve performance up to 20%. These performance improvements were possible even when the number of cache misses did not decrease appreciably when optimizations were applied. This demonstrates two additional important conclusions. First, the allocation techniques used here are effectively moving data from the data cache into the register file. Second, these optimizations eliminate cache accesses through this movement, effectively speeding up access to data that would otherwise reside in the cache. Since cache hits are the common case, this is an important optimization. Other techniques that reduce cache miss penalty such as prefetching, data layout, and smarter cache replacement policies, are orthogonal to the work here.

Finally, this work demonstrates that we can use even more registers if the compiler can allocate registers such that active functions share the register file. In this case, 100 to 200 registers are necessary to satisfy the register requirements of most benchmarks.

We note that there are other candidates for register allocation that we have not explored in this work. Aliased data items cannot reside in registers for their *entire* lifetime or in regions where they are aliased unless support is added to the hardware. We will examine this in future work.

Acknowledgments

This work was supported by DARPA grant DABT63-97-C-0047. Simulations were performed on computers donated through the Intel Education 2000 Grant. The primary author is supported on a University of Michigan Rackham Graduate School Predoctoral Fellowship. We would also like to thank Vijayalakshmi Srinivasan and anonymous reviewers for their comments, which significantly improved the quality of the paper.

References

[1] Scott A. Mahlke, William Y. Chen, Pohua P. Chang and Wen-mei W. Hwu. Scalar Program Performance on Multiple-Instruction-Issue Processors with a Limited Number of Registers. Proc. 25th Hawaii Intl. Conf. System Sciences, pp. 34-44, Jan 6-9, 1992.

[2] Manuel E. Benitez and Jack W. Davidson. Register Deprivation Measurements. Department of Computer Science, University of Virginia Tech. Report. Nov. 1993.

[3] Yuval Tamir and Carlo H. Sequin. Strategies for managing the register file in RISC. ACM Transactions Computer Systems, Vol. 32 No. 11, pp. 977-988. Aug. 1983.

[4] D. A. Patterson and C. H. Sequin. Risc 1: a Reduced Instruction Set VLSI Computer. 8th. Ann. Symp. Computer Architecture, Vol. 32 No. CS-93-63, pp. 443-457. Nov. 1981.

[5] S. A. Mahlke, W. Y. Chen, J. C. Gyllenhaal, W. W. Hwu, P. P. Chang and T. Kiyohara. Compiler code transformations for superscalar-based high-performance systems. Proc. Supercomputing '92: Minneapolis, Minnesota, November 16-20, 1992, pp. 808-817, Aug. 1992.

[6] David G. Bradlee, Susan J. Eggers and Robert R. Henry. The Effect on RISC Performance of Register Set Size and Structure Versus Code Generation Strategy. Proc.18th Intl. Symp. Computer Architecture, pp. 330-339, May. 1991.

[7] John A. Swenson and Yale N. Patt. Hierarchical Registers for Scientific Computers. Proc. Intl. Conf. Supercomputing, pp. 346-353, July 1988.

[8] Steven S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, 2929 Campus Drive, Suite 260, San Mateo, CA 94403, USA, 1997.

[9] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman. Compilers--Principles, Techniques, and Tools. Addison-Wesley, Reading, MA, USA, 1986.

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

[11] Keith D. Cooper and Timothy J. Harvey. Compiler-Controlled Memory. Eighth Intl. Conf. Architectural Support for Programming Languages and Operating Systems, pp. 100-104, Oct, 1998.

[12] R. M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. IBM Journal Research and Development, Vol. 11 No. 1, pp. 25-33. Jan, 1967.

[13] Robert M. Keller. Look-Ahead Processors. ACM Computing Surveys, Vol. 7 No. 4, pp. 177-195. Dec, 1975.

[14] 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.

[15] 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.

[16] 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.

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

[18] David W. Wall. Register Windows vs. Register Allocation. ACM SIGPLAN Notices, Vol. 23 No. 7, pp. 67-78. July 1988.

[19] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins and Peter W. Markstein. Register Allocation Via Coloring. Computer Languages, Vol. 6 No. 1, pp. 47-57. unknown month, 1981.

[20] G. J. Chaitin. Register Allocation and Spilling via Graph Coloring. Proc. SIGPLAN ‘82 Symp. Compiler Construction, pp. 98-105, unknown month, 1982.

[21] Preston Briggs. Register Allocation via Graph Coloring.. Rice University, Houston, Texas, USA Tech. Report. unknown month, 1992.

[22] David L. Weaver and Tom Germond. The SPARC Architecture Manual, Version 9. Sparc International and PTR Prentice Hall, Englewood Cliffs, NJ, 1994.

[23] Intel. IA-64 Application Developer’s Architecture Guide. Intel Corporation, May 1999. Available at http://developer.intel.com/design/ia64/devinfo.htm.

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

[25] Linley Gwenapp. Digital 21264 Sets New Standard. Microprocessor Report, Vol. 10, No. 14. October 28, 1996, pp. 11-16.

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

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

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

[29] Analog Devices. ADSP-2106x SHARCTM User’s Manual. Second Edition (7/96). Analog Devices, Inc. 1997.

[30] David L. Weaver and Tom Germond. The SPARC Architecture Manual, Version 9. Sparc International and PTR Prentice Hall, Englewood Cliffs, NJ, 1994.

[31] Richard L. Sites. The Alpha Architecture Reference Manual. Digital Press, Burlington, MA, 1992.

[32] Texas Instruments. TMS320 DSP Development Support Reference Guide and Addendum. August 1997. Literature Number SPRU226.

[33] Siemens. TriCore Architecture Overview Handbook. Version 1.1. September 17, 1997.

[34] Intel Corporation. Penium Pro Family Developer's Manual. Volume 2: Programmer's Reference Manual. Order Number 242691. 1996.

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

[36] Philips Electronics. TM1000 Preliminary Data Book. TriMedia Product Group, 8111 E. Arques Avenue, Sunnyvalue, CA 94088. 1997.

[37] Steven S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, 2929 Campus Drive, Suite 260, San Mateo, CA 94403, USA, 1997.

[38] K. D. Cooper and K. Kennedy. Interprocedural Side-Effect Analysis in Linear Time. ACM SIGPLAN Notices, Vol. 23 No. 7, pp. 57-66. July 1988.

[39] http://www.ptsc.com/psc1000/overview.html.

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

[41] Fred C. Chow. Minimizing Register Usage Penalty at Procedure Calls. SIGPLAN '88 Conf. Programming Language Design and Implementation, pp. 85-94, July 1988.

[42] W. Y. Chen, P. P. Chang, T. M. Conte and W. W. Hwu. The Effect of Code Expanding Optimizations on Instruction Cache Design. Center for Reliable and High-Perform ance Computing, University of Illinois, Urbana-Champaign Tech. Report CRHC-91-17. May 1991.

Appendix A: Register Promotion Implementation

The following is a simplified explanation of how the algorithm proceeds. Each function in the module is treated separately, except for the alias information that is shared between functions.

1. Collect global scalar variables and constants into *globalCandidateList*
2. Collect aliased scalar local variables into *localCandidateList*
3. Walk through the function and find each variable or constant reference, or suitable dereference. If it is a direct reference is to a *globalCandidate* or *localCandidate*, store this reference into the *candidateReferenceList*. There is a *candidateReferenceList* for each promotion region. If it is a suitable dereference, also add it to the *candidateReferenceList*.
4. For each promotion region, remove aliased candidate references from the *candidateReferenceList*. An alias exists if the alias analyzer has determined there is a potential definition or use of the candidate which is not through the candidate's primary name.
5. For each promotion region, promote all candidates that remain in the *candidateReferenceList* for the region. Do not promote something in an inner loop if it can be promoted in an outer loop.

In MIRV, dereferences are only promoted in loop regions. A "suitable dereference" is a load or store operation which satisfies a number of conditions. For this work, these are:

1. The base address is a simple variable reference.
2. The dereference is at the top level block of the loop, i.e. not in a conditional clause. Otherwise promoting it might cause a fault.

```
for (i=0;i<DIM_X;i++) {
  B[i] = 0;

  for (j=0;j<DIM_Y;j++)
    B[i] += A[i][j];


}

Original Code
```

```
for (i=0;i<DIM_X;i++) {
  B[i] = 0;
  t1 = &B + i*4;
  for (j=0;j<DIM_Y;j++)
    *t1 += A[i][j];


}

After Optimization
```

```
for (i=0;i<DIM_X;i++) {
  t1 = &B + i*4;
  t2 = 0;
  for (j=0;j<DIM_Y;j++)
    t2 += A[i][j];

  *t1 = t2;
}

After Register Promotion
```

Figure 10. Example of promotion of an indirect memory reference. After optimization, the invariant address computation is moved out of the inner loop, leaving a dereference of t1. This constitutes a load and store operation. Note that the address arithmetic is now explicit (no C scaling). After register promotion, the dereference is removed from the loop [14].

3. The loop must be a do-while loop so that it is guaranteed to execute at least once. This is often satisfied because MIRV performs loop inversion to reduce the number of branches in the loop, which turns while loops into do-while loops.
4. The variable constituting the base address must be loop invariant and there must be no other definitions of it in the loop.

Step 4 in the above algorithm depends on the alias analysis used in the compiler. MIRV does flow-insensitive, intra-procedural alias analysis. Our alias analysis is based on the presentation in Muchnick’s book, where he also notes that flow-insensitivity is not usually problematic [37]. The lack of inter-procedural alias analysis implies that pointers passed in function calls result in the conservative assumption that anything reachable through the pointer is both used and defined in the callee. To mitigate the effects of function calls, the compiler performs a simplistic side-effect analysis [38]. For each function, the compiler records which global objects may be used and defined in the function. Pointer arguments are assumed to point to any global object of compatible type. We have observed that this process opens up many more chances for promotion than would otherwise be possible.

The MIRV promoter can also promote dereferences of invariant pointers. For example, the loop in Figure 10 has an array reference with a loop-invariant index. The array element can be promoted to a register as shown in the figure.