Register Deprivation Measurements

Manuel E. Benitez
Jack W. Davidson

Computer Science Report No. CS-93-63
November 15, 1993

Register Deprivation Measurements

Manuel E. Benitez
meb1u@virginia.edu
(804) 982-2296

Jack W. Davidson
jwd@virginia.edu
(804) 982-2209

Department of Computer Science
University of Virginia
Charlottesville, VA 22903

Abstract

The development of register deprivation measurements was motivated by the desire to study the effect that register demand has on code improvement and register allocation strategies. In addition to the obvious application of testing the spill mechanism used by the compiler's register allocator, the register deprivation strategy can be used to determine the relationship between the number of allocable registers and the effectiveness of a code improver as a whole or its optimization phases individually, enhance the coverage of validation suites, evaluate the register demands of benchmark suites and help machine designers determine the optimal number of registers needed to support existing compiler technologies. This paper contains a description of register deprivation techniques, presents some of their most useful applications, and discusses issues that must be addressed in order to incorporate this technique into a compiler. Also included are register deprivation measurement results obtained using a modest set of benchmarks that provide interesting and somewhat unexpected insights pertaining to optimizations, benchmark programs and architectures.

1 Introduction

1.1 Motivation

Optimizing compilers generate code by invoking a comprehensive set of optimizations. Since most code improvement transformations increase the average number of registers that are simultaneously needed within a program, good register allocation is essential to producing high-quality code. Currently, graph coloring [CAC+81] is the paradigm of choice for performing register allocation. The typical compiler that uses graph coloring allocation performs optimizations using an infinite set of pseudo-registers that are bound to hardware registers just prior to code generation. When the allocator is unable to assign to each pseudo-register to an actual register, it reduces the register pressure by introducing spills, which essentially assign pseudo-register values to memory location instead of a hardware registers.

Good graph coloring register allocation strategies strive to minimize spills and, when they are unavoidable, spills the values that reduce the quality of the resulting code as little as possible. Regardless of how slightly a spill impacts the quality of the code, it invalidates the assumption that accessing a pseudo-register item is roughly equivalent to accessing a hardware register and can cause the optimizer to produce code that is less efficient than the original code. The need to produce efficient spill code also complicates an otherwise simple register allocator.

A typical consequence of postponing register assignment decisions is that register resources are over-committed and produce many spills when the code provides many opportunities to perform code improvement transformations or the target machine lacks sufficient registers. Even priority-based coloring allocation strategies [CH84] are vulnerable under these circumstances because, although they do not explicitly spill, they cannot provide registers to all of the transformations that expected to get them. While these problems are often manageable, they do limit the range of architectures over which a compiler can be retargeted and the number of register-consuming optimization phases that the compiler can effectively accommodate.

These concerns motivated the development of new graph-coloring register allocation strategies for the retargetable *vpcc/vpo* optimizing compiler [BD88]. The most difficult part of this task was not developing these strategies, but devising a technique to evaluate their efficacy. Thus, the register deprivation measurement techniques presented here were initially developed to satisfy this need. Later, these techniques were found to have applications beyond their original purpose and to be simple enough incorporate into existing compilers.

1.2 Applications

As the previous section implies, register deprivation measurements can be used to evaluate register allocation algorithms. The reason for this is that the increased competition for the registers caused by global and inter-procedural optimizations or a diminutive register set can be simulated even on an architecture with a prodigious set of registers by reducing the number of register available to the compiler. Register deprivation techniques can also be used to design better register allocation mechanisms because they provide insights about the behavior of the allocator as the demand on the register resources varies. In addition, register deprivation measurements can also be used to gauge the effectiveness of and gain insights into the entire code improvement system as well as the individual optimization phases within the compiler.

During the course of a register deprivation measurement, spill mechanisms and register allocation heuristics are exercised more frequently and under more severe conditions than even a thorough compiler validation test suite is likely to produce. This is not a novel use of a register deprivation-like technique, since Fraser and Hanson utilized a similar strategy to test a local register allocator for the *lcc* compiler [FH92]. What is surprising about the register deprivation process presented here, however, is that it uncovered defects in areas of the compiler that were not associated with the register allocator and thought to be very reliable.

Register deprivation measurements can also be used to determine if a benchmark suite fails to exercise certain architectural elements. These evaluations can be used to select new code for both validation and benchmark suites. As marketing and design decisions are increasingly made on the basis of benchmark test results, tools that can be used to locate inadequacies in benchmark suites become more valuable.

Existing compiler technology is not capable of making effective use of *large* numbers of registers. Even link-time register allocation techniques [Wal86] have limits on the number of registers that they can successfully utilize. Register deprivation measurements can be used to probe these limits.

Hardware designers are often forced to make trade-offs without the guidance of empirical data. Register deprivation techniques are well suited to the task of providing architects with the kinds of feedback that can be used to ensure that an architecture complements the compiler technology that supports it. How many registers should be provided by a machine and how many stages should be included in a processor’s pipeline are just two or the issues that register deprivation measurements can help to resolve.

1.3 Organization

The intent of this paper is to describe register deprivation measurements, discuss some useful applications of this technique and discuss the issues that must be addressed before incorporating it in an existing compiler. A description of the register deprivation process and how the results of the measurements are presented graphically is given in Section 2. Section 3 elaborates on the various applications of the register deprivation technique. Section 4 presents the issues that have to be addressed in order to incorporate the register deprivation process into a compiler system. Section 5 concludes this paper with a brief summary.

2 Register Deprivation Measurements

2.1 Trials and probes

A register deprivation *trial* consists of a sequence of *probes*. A probe is a collection of data obtained by compiling and executing a suite of benchmark programs. Each successive probe is identical to the previous one in every respect except that the compiler is prohibited from using one or more of the registers available to the preceding probes.

The data collected during a trial can be any combination of measurements that give an indication of the quality of the code generated for each program in the benchmark suite. Potential measurements include execution times, instruction and memory reference counts, cache performance measurements, hardware monitor timing information and page replacement traces. The register deprivation results presented here measure the number of instructions, the number of memory references and the amount of processor time required to execute a set of benchmark programs.

The number of probes that make up a complete register deprivation trial depends on the target architecture. In the first probe, the complete set of allocable registers is available to the compiler, while in the final probe, only the absolute minimum set of registers needed to successfully compile and execute the benchmark suite is used. The total number of probes is the difference between the number of allocable registers available to these two probes plus one. Register deprivation experiments that vary only a subset of the registers and *register augmentation* measurements, which can be performed by simulating the effects of having more registers than are actually available on the target machine [DW91b], are also possible.

Table 1 shows the instruction execution counts obtained from a sample register deprivation trial consisting of six probes taken on a mythical machine with a three program benchmark suite. These results indicate that the system used to compile the benchmark programs uses each additional allocable register available in a manner that reduces the number of instructions that the benchmark programs execute. Although these values indicate the net effect of each additional register on the number of instructions that the benchmark suite executes, it does not provide the information needed to determine how each individual optimization phase or groups of phases in the compiler are affected. To obtain this information, a pair of register deprivation trials must be performed.

2.2 Comparative measurements

To separate the impact that each additional register has on the global optimization phases of the optimizer from any side effects that they might have on the code generated for each benchmark program, a *base* trial must be performed in which the compiler produces code without performing any global optimization transformations. Unlike the optimized trial, the results of the base trial shown in Table 2 reveal few improvements as new allocable registers are

Benchmark	Number of allocable registers in the probe					
	4	5	6	7	8	9
prog1	158,507,316	135,235,161	114,342,566	103,482,410	99,286,348	98,637,175
prog2	97,258,149	88,617,097	80,234,234	77,982,381	76,321,685	75,795,145
prog3	63,196,050	56,069,413	54,453,682	53,297,184	52,943,230	52,109,342

Table 1: Instruction execution counts from a register deprivation trial

added. These slight improvements result from a reduction in the amount of spill code produced by the local register allocator.

Benchmark	Number of allocable registers in probe					
	4	5	6	7	8	9
prog1	160,452,720	160,173,064	160,148,155	160,147,579	160,147,532	160,147,532
prog2	98,753,703	98,692,100	98,682,457	98,678,981	98,678,981	98,678,981
prog3	64,093,511	63,981,342	63,978,106	63,971,423	63,971,367	63,971,288

Table 2: Instruction execution counts from a register deprivation base trial

To illustrate the relationship between the reduction in the number of instructions executed that are attributable exclusively to performing global optimizations and the number of allocable registers available, each result obtained in the optimized register deprivation trial is compared against its corresponding base trial result. This process eliminates the impact that the size of the allocable register set has on the components that are not being measured, regardless of whether it is beneficial or detrimental. The following formula, given in Hennessy and Patterson [HP90], is used to obtain the percent improvement of the initial optimized trial over the base trial:

$$\text{percent improvement} = \frac{\text{instructions executed}_{\text{base trial}} - \text{instructions executed}_{\text{optimized trial}}}{\text{instructions executed}_{\text{optimized trial}}} \times 100$$

Table 3 shows the results obtained by applying this formula to each corresponding pair of values in the sample trials. These improvement values indicate that global optimizations produce only a slight reduction in the number of instructions executed by the benchmark programs when the minimum number of allocable registers is used. As addi-

Benchmark	Number of allocable registers in probe					
	4	5	6	7	8	9
prog1	1.23%	18.44%	40.06%	54.76%	61.30%	62.36%
prog2	1.54%	11.37%	22.99%	26.54%	29.29%	30.19%
prog3	1.42%	14.11%	17.49%	20.03%	20.83%	22.76%

Table 3: Results of comparing two register deprivation trials

tional registers are made available, the global optimization transformations are able to perform transformations that substantially reduce the number of instructions executed by the benchmark programs. The rate of improvement tapers off quickly after sufficient registers are provided to perform the most beneficial transformations. Once this point is reached, each successive register enables the optimizer to make only modest improvements to the benchmark suite.

Instead of displaying the results of a comparative register deprivation measurement in tabular form, the measurement results are plotted with the number of allocable registers available to each probe on the *X* axis and the percent improvement value along the *Y* axis. These plots display all of the individual improvement values produced by each benchmark program along with a solid line that indicates the average performance improvement over each of the programs in a trial. The corresponding plot of the sample register deprivation experiment results is shown in Figure 1.

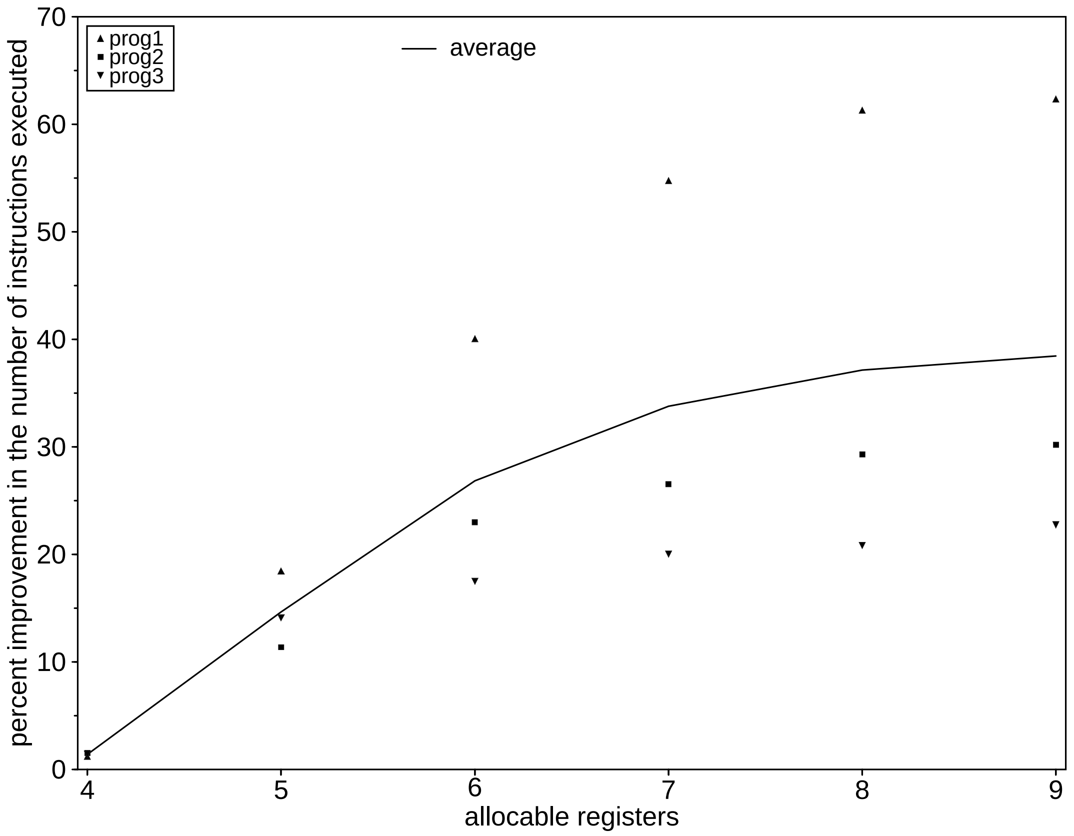
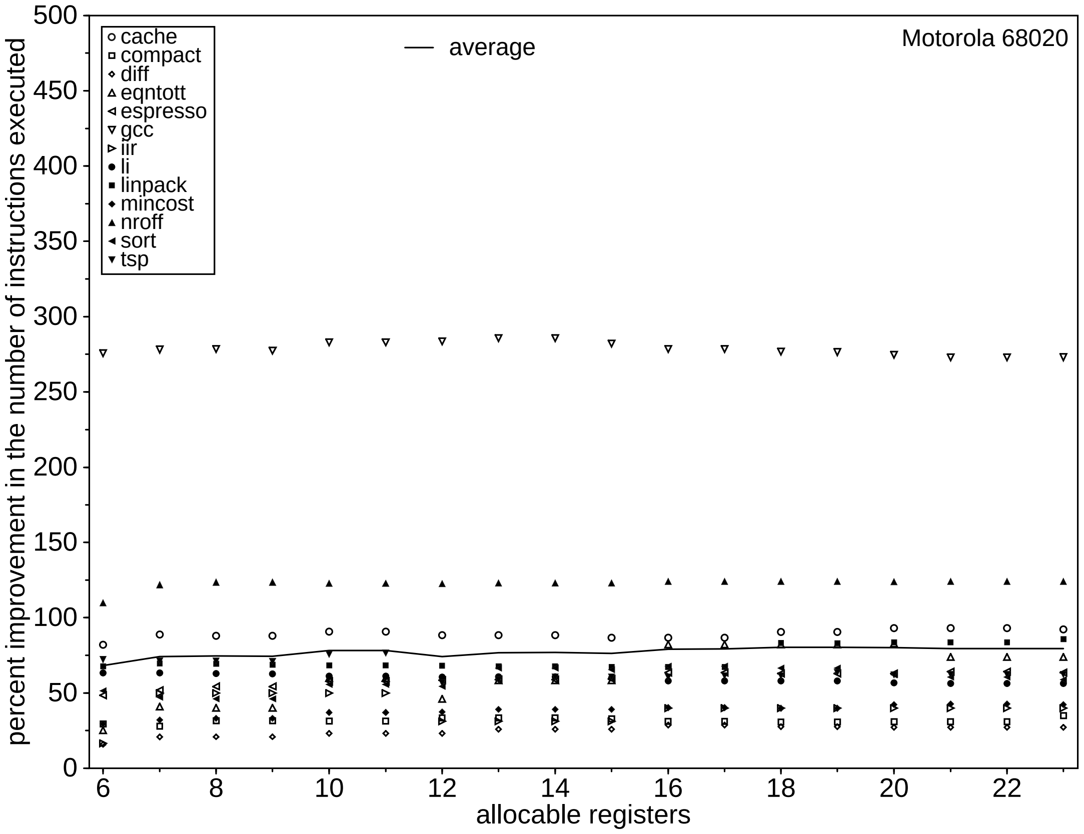


Figure 1: Results of a sample register deprivation experiment

3 Applications

3.1 System performance measurements

The process of comparing a register deprivation base trial during which the compiler foregoes all optimizations against a trial that applies the entire set of optimizations can be used to determine the impact that the number of allocable registers has on the entire code improvement system. Figure 2 shows how the number of allocable registers affects the number of instructions executed by the object code produced by the *vpcc/vpo* system for the Motorola 68020 [Mot85] CISC and the MIPS R3000 [Kan87] RISC processors. These results show that the number of instructions executed by the code produced by *vpcc/vpo* decreases as the number of allocable registers increases. It is inter-



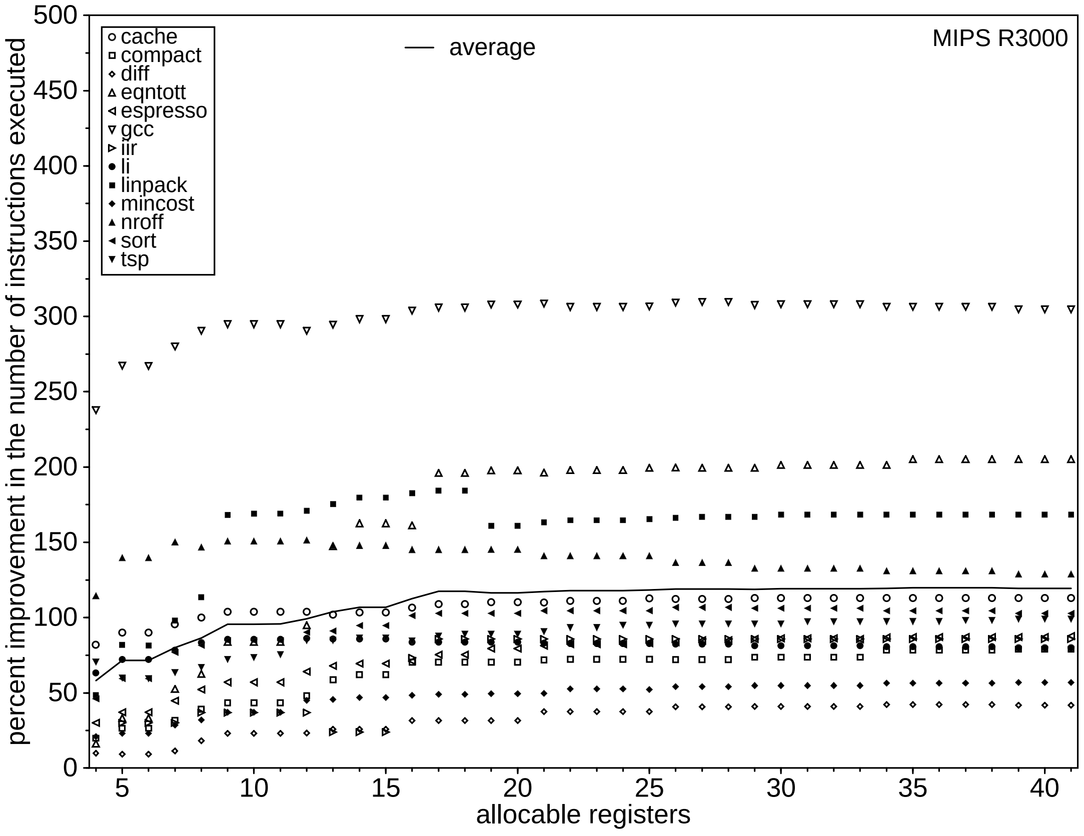


Figure 2: System performance results (instruction counts)

esting to note that the improvements obtained on the CISC processor are small compared to those on the RISC processor. This effect can be attributed to the fact that most of the code-improvement transformations performed by *vpcc/vpo* replace memory references with register references. On a CISC machine like the Motorola 68020, memory references are integrated into the instructions that use their values, so the total number of instructions remains fairly constant. On most RISC architectures, replacing a memory reference with a register reference eliminates an entire load or store instruction, and these reductions become evident when the number of instructions executed is measured.

It is important to emphasize that the results in Figure 2 show only the effect of the register set size on the number of instructions executed, which is only one measure of code quality. Attempting to draw general conclusions about the overall code quality exclusively from this metric is inappropriate. To obtain further insight on the impact that the number allocable registers available has on *vpcc/vpo*, a pair of register deprivation trials measuring the number of memory references executed were performed. The results of comparing these two trials are given in Figure 3.

The memory reference comparison results suggest that the register set has a significant impact on the number of memory references that the code produced by *vpcc/vpo* performs. These results support the claim that the most effective code-improvement transformations performed by *vpcc/vpo* primarily replace memory references with register references, because they illustrate that each additional allocable register produces a substantial decrease in the number of memory references executed. This relation holds true for both CISC and RISC architectures.

While instruction and memory reference counts can provide useful insights concerning the relationship between the number of allocable registers and the quality of the code produced by the compiler, execution time measurements are more indicative of code quality and overall system performance. Given that the compiler generates correct code, most users are ultimately concerned with the speed at which the improved code executes. Since neither instruction execution nor memory reference counts alone or in concert can provide this information, register deprivation trials must also be performed to explicitly measure execution times. The impact of varying the number of allocable registers on the overall execution time performance of the code generated by *vpcc/vpo* is shown in Figure 4.

These results indicate that while there is some correlation between the instruction and memory reference counts and the execution time performance, it is difficult to predict the magnitude of the execution time performance improvements from the instruction and memory reference counts. The instruction and memory reference counts do not, for example, indicate the magnitude of the impact that the code-improvements have on the performance of the memory system's cache or the processor's instruction pipeline. Also, the instruction and memory reference counts are unaffected by factors such as the amount of time spent performing I/O and the other essential tasks performed by the operating system on behalf of the program. In addition, these counts are entirely reproducible while the execution time measurements, because of an interaction between the multi-tasking environment and the low-resolution clock provided by the system on which they are obtained, have a substantial margin of error. To increase the level of confidence of the execution time register deprivation measurements each individual measurement is repeated five times so that the highest and lowest execution time results can be discarded and the remaining three are averaged to provide the final execution time value. In spite of these precautions, it is not unusual for two identical execution time trials to vary by a few percent.

Despite the unfortunately wide margin of error, the execution time register deprivation measurements show that *vpcc/vpo* is able to improve the quality of the code generated for both CISC and RISC architectures even when few allocable registers are available. The positive average improvement obtained when only the minimal number of regis-

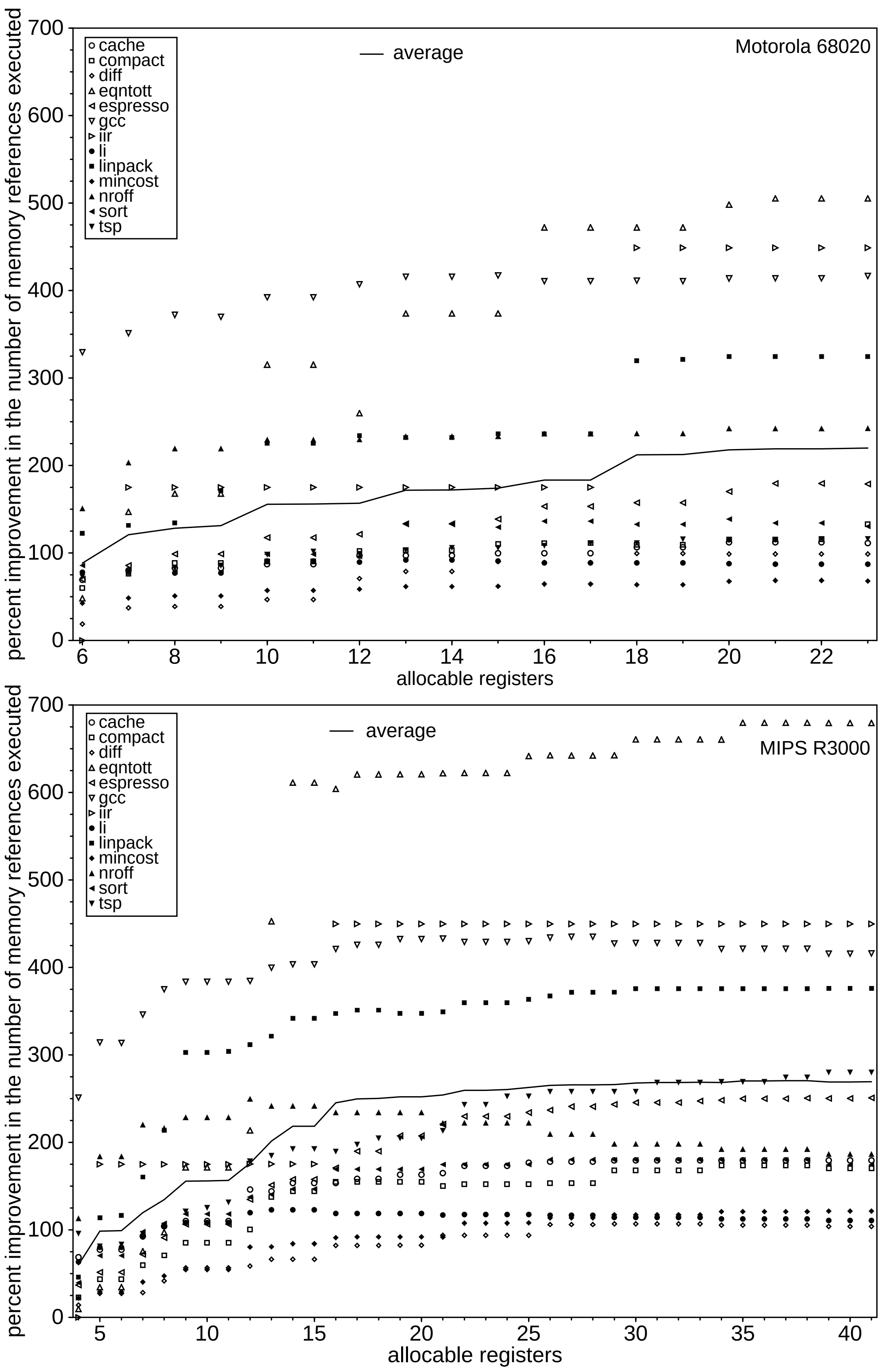
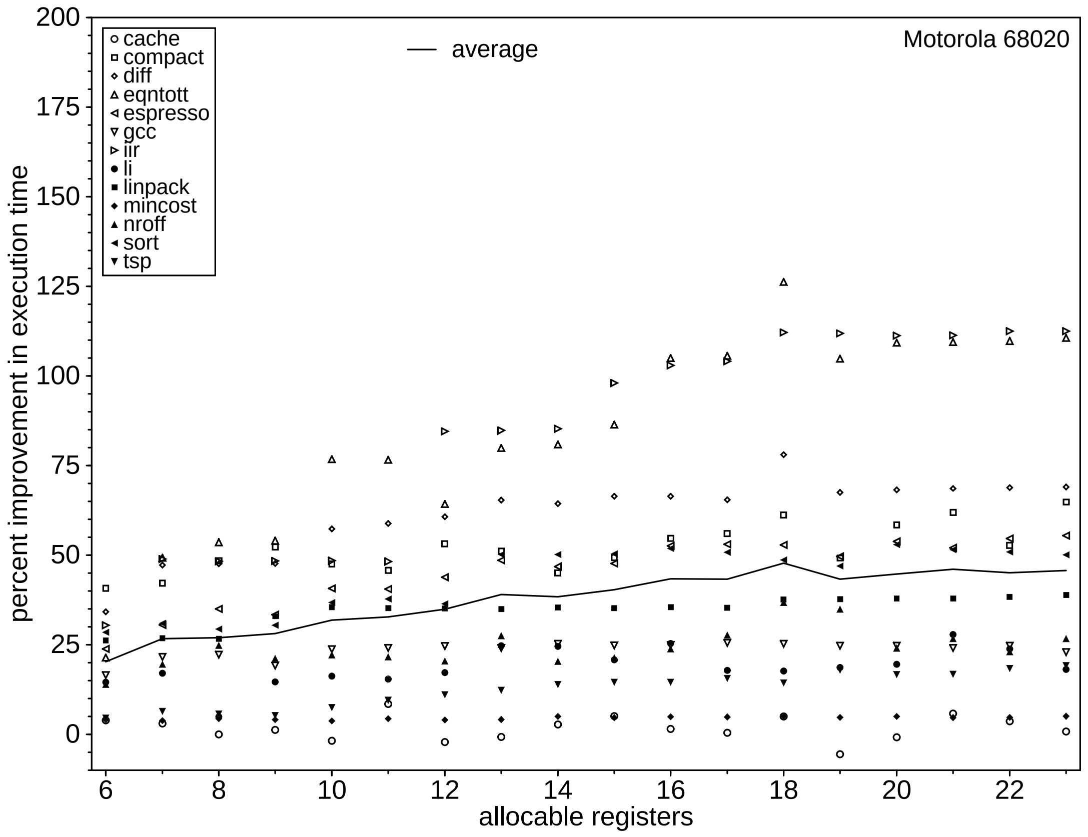


Figure 3: System performance results (memory reference counts)



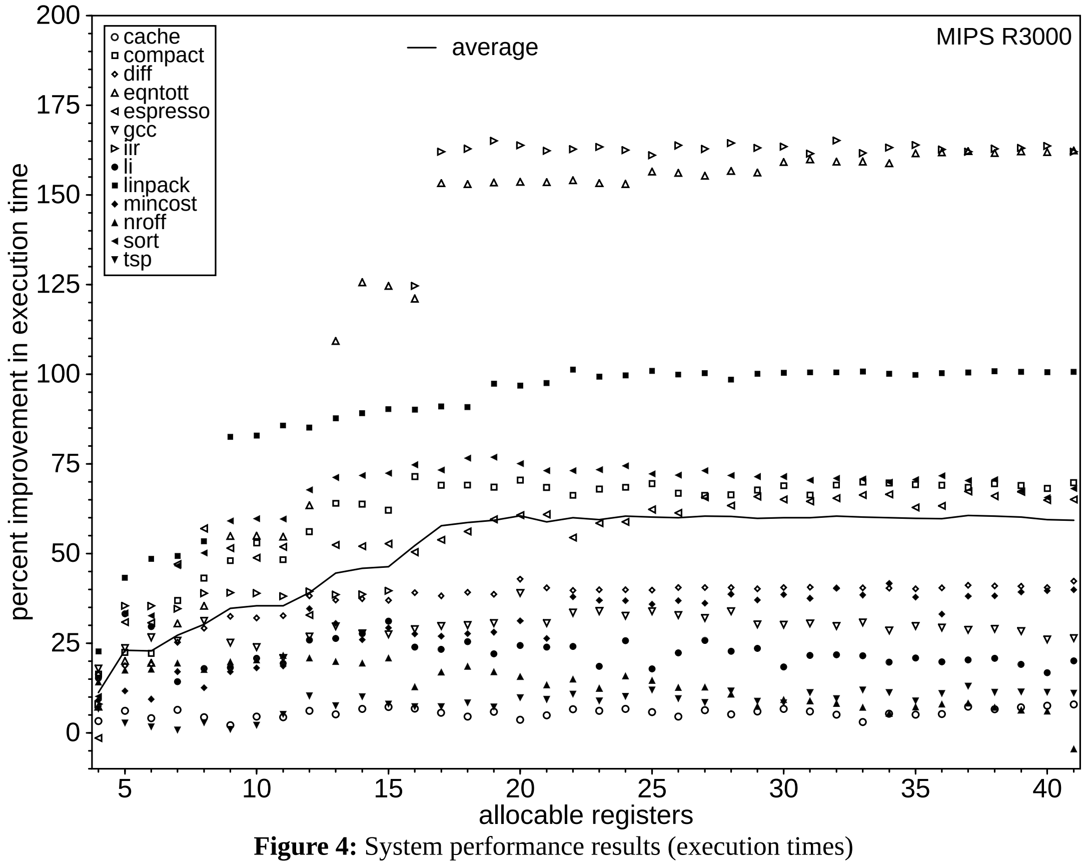


Figure 4: System performance results (execution times)

ters are available suggests that *vpcc/vpo* does not over-commit register resources. This suggests that it is not limited exclusively to architectures that provide large sets of registers, but that it is of some utility even on machines that provide few registers.

3.2 Phase performance measurements

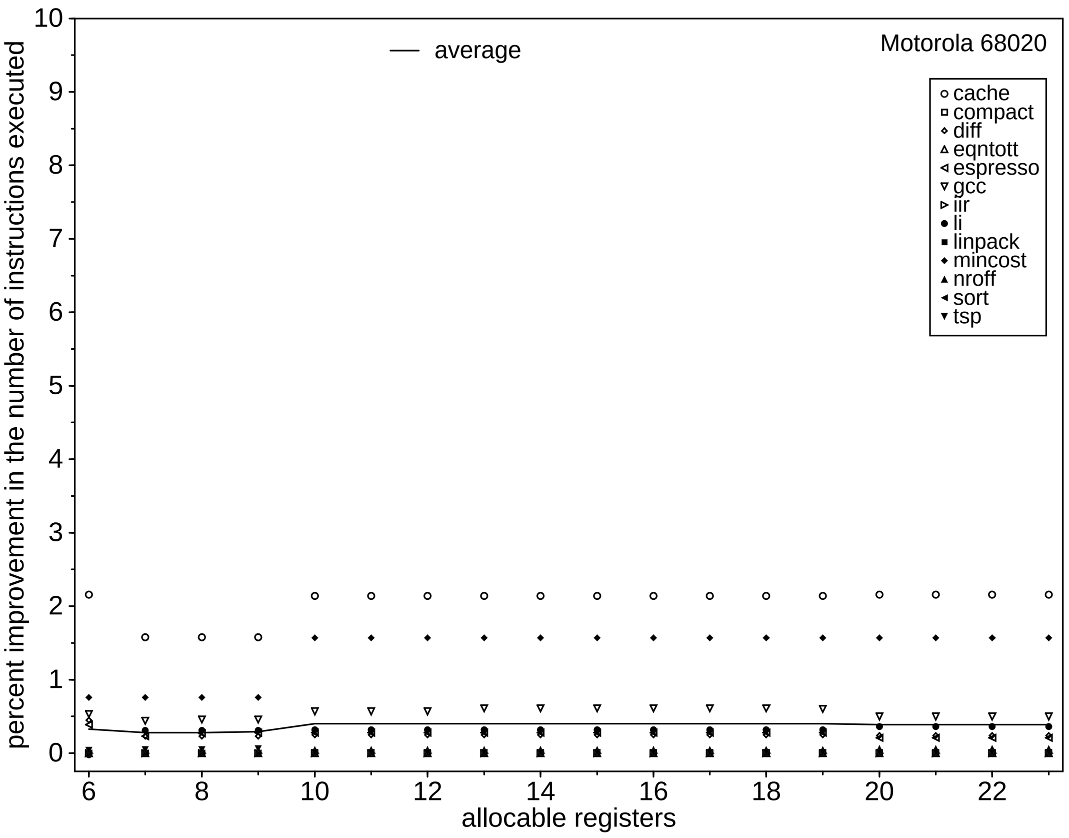
Register deprivation experiments can also reveal the effect that the size of the register set has on the individual optimization phases in a language translation system. This is done by comparing a register deprivation trial in which the phase whose impact is to be measured is invoked against a base trial in which it is not. This technique isolates the effect that the size of the register set has on the phase measured from the effect that the register set has on the other optimization phases. It is important to note, however, that the interactions between the measured phase and the other phases influence the results of the comparison. In most cases, capturing the effect of these interactions along with the direct impact on the measured phase is desirable because it best represents the overall effect that the measured phase has on the optimizer. In other instances, these interactions can be minimized by not invoking any of the other optimization phases in both of the register deprivation trials performed.

Figures 5 and 6 show how the size of the register set affects the reduction in the dynamic instruction and memory reference execution counts produced by *vpcc/vpo*'s evaluation order determination phase [Dav86]. Recall that the purpose of evaluation order determination is to reduce the number of registers required to assign register to hold temporary values. These results where obtained by comparing a base trial in which only control-flow optimizations and instruction selection were performed against a trial in which control-flow optimizations, instruction selection and evaluation order determination were invoked. The figures show a nearly constant average improvement in the number of instructions and memory references executed on the Motorola 68020. On this system, the set of volatile registers is small even when there are many allocable registers (see Figure 8), so most of the improvements appear as a reduction in the number of instructions and memory references needed to preserve the values of the non-volatile registers across external function calls. Also, for reasons already mentioned, reductions in spill code make little difference in the CISC instruction counts but show up to some extent on the memory reference counts. These two factors are responsible for the nearly constant improvements exhibited by the evaluation order determination phase on this architecture.

On the MIPS R3000, however, the instruction and memory reference counts initially show dramatic improvements as some allocable registers become available and then remain flat for the remainder of the experiment. The significant improvements at the low end of the scale are a combination of two factors. First, since volatile registers are plentiful when there are more than just a few allocable registers, the reduction in the number of registers assigned to temporary expression values does not affect the number of save and store instructions in the function prologues and epilogues. Second, since spills due to inefficient expression sequences occur only when very few allocable registers are available, reductions in the amount of spill code generated appear only when allocable registers are scarce. These results suggest that the ratio of volatile to non-volatile registers can have a significant impact on the effectiveness of some optimizations and that there exists optimizations that become less effective as the number of allocable registers increases.

3.3 Benchmark suite evaluation

Register deprivation measurements can be used to determine if a suite of test programs lacks sufficient amounts of some types of code. An interesting characteristic of the MIPS R3000 register deprivation results shown in Figure 2 is the presence the "flat" areas between probes 5 and 6, 9 and 11, and 14 and 15. These probes add only floating-point



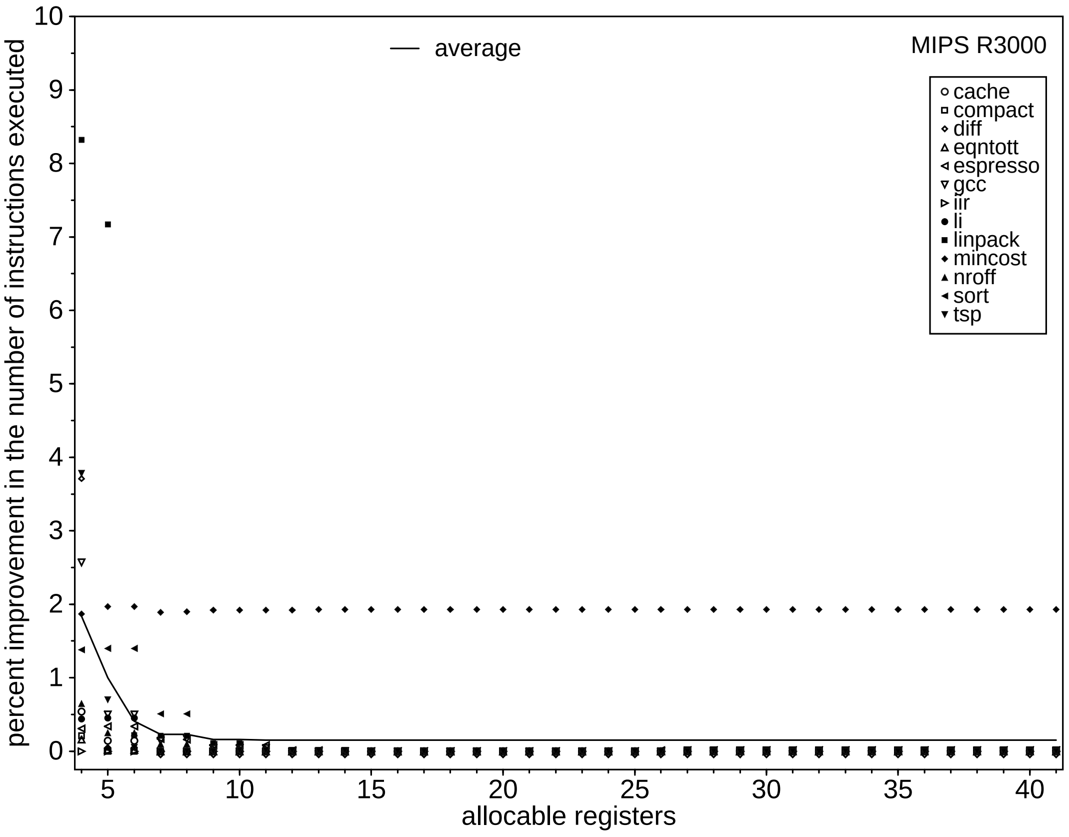


Figure 5: Evaluation order determination performance results (instruction execution counts)

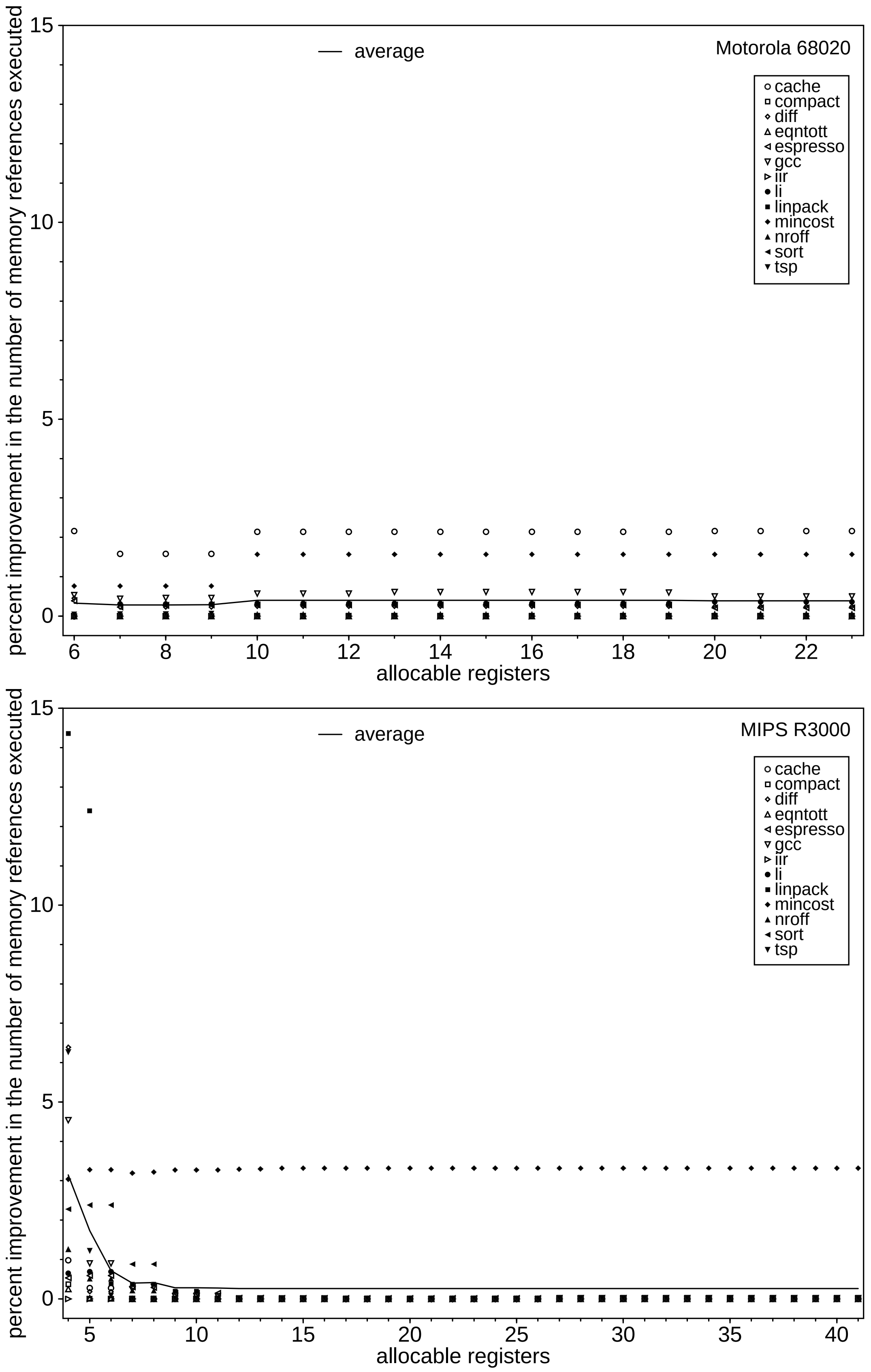


Figure 6: Evaluation order determination performance results (memory reference counts)

registers to the set of allocable registers available to the compiler (see Figure 7), suggesting that either the benchmark suite lacks sufficient floating-point codes or that the compiler cannot make use of more than a small number of floating-point registers effectively. Which of these hypotheses is correct can be determined by adding more floating-point codes to the benchmark suite. If these “flat” areas become less pronounced, one can conclude that the suite was heavily biased towards integer codes. The absence of flat areas in a register deprivation experiment does not necessarily indicate that the benchmark suite has an appropriate balance of integer and floating-point codes, but the ability to detect an imbalance under some circumstances is a useful property.

In addition to providing insight into overall trends, the register deprivation results also yield useful information about individual benchmark programs and the way they interact with the compiler. One example is the *nroff* utility, which, as a cursory examination of the MIPS processor results shown in Figure 2 reveals, executes more instructions as more registers become available. The reason for this decrease in code quality is that this program spends much of its time calling a function to obtain the next character from the input file. This function consists of a loop that skips over “noise” characters. Since most input files contain few such characters, the code in this loop is, for all essential purposes, straight-line code. The optimization phases that perform transformations on loops, however, fail to determine this at compile time, and invest register resources assuming that the loop body is executed more frequently than the code that immediately precedes the loop is. Thus, as each new register is made available, new code is inserted to compute loop-invariant expressions outside of the loop and load and store instructions are added to ensure that the values in the non-volatile registers used by the function are properly maintained. Since these transformations fail to improve the code, each additional register merely increases the magnitude of the mistake.

Another interesting insight that was gleaned from the register deprivation results is the tendency for benchmarks to improve abruptly rather than gradually as the number of allocable registers is increased. One instance of this can be seen in the performance results for the MIPS 3000 shown in Figure 4. These results show a significant improvement in the execution time of the *iir* benchmark between 15 and 17 registers, which comes from applying recurrence optimization [BD91] to the main loop of the benchmark. This transformation is very effective, but consumes a trio of registers and the mechanism that prevents *vpcc/vpo* from over-committing register resources prevents the transformation from taking place until there are sufficient registers available to support it. For this reason, it is not unusual to see sudden improvements in the execution times of individual programs instead of the gradual improvement that is more characteristic of the average performance values.

One final example of how register deprivation measurements can be used to detect benchmarks that behave in extraordinary ways is evident in the MIPS R3000 results shown in Figures 5 and 6. Contrary to the general trend, the improvements made to the *mincost* benchmark by the evaluation order determination phase are constant regardless of the number of allocable register available. Unlike the rest of the benchmark programs, *mincost* contains an expression that uses the value returned by an external function call and is written so that *vpcc* emits much of the expression before the external function is called. Because of this arrangement, a number of non-volatile registers are used to hold subexpression values. When evaluation order determination is applied, however, the expression is reordered so that the external function call is executed first. This eliminates the need to use non-volatile registers and the overhead involved in saving and restoring these register’s values in the prologue and epilogue of the procedure that contains this expression.

3.4 Architecture design

Register deprivation measurements can be used to provide useful architecture design information. For example, when attempting to determine whether to sacrifice potential instructions in favor of additional registers, a register deprivation experiment performed on a suitable simulator can be used to determine if the compiler will be able to use the additional registers effectively. Similar techniques can be employed to determine the number of pipeline stages, processing units and the amount of on-chip cache to provide. For example, to determine the effect of registers on a pipeline, the machine would be simulated and the number of interlocks would be plotted against the number of registers available. Alternately, the number of register could be left constant while varying the number of pipeline stages, instead.

An example of the insights that register deprivation measurements provide can be seen in the Motorola 68020 results shown in Figure 2. These results indicate that most significant memory reference reductions occur between probes 6 through 8, 11 and 12, 14 and 15, 17 and 18 and 22 and 23. Surprisingly, all of these probes increase the number of data registers available to the compiler (see Figure 8). This information suggests that partitioning the integer register set into registers that handle address expressions and registers that process all other integer expressions might not result in the best possible utilization of the register set.

3.5 Validation suite enhancement

The process of implementing and obtaining register deprivation measurements exposed deficiencies in parts of *vpcc/vpo* that were thought to be free of defects. In particular, the code responsible for inserting spill code in the local register assigner and transfer code in the global register allocator was more thoroughly exercised by the register deprivation process than by any of the substantially larger validation suites previously employed. In general, spill mechanisms are most thoroughly tested by the register deprivation probe that uses the fewest number of registers.

Interestingly, register deprivation measurements uncovered deficiencies in parts of the compiler that are not associated with the register allocator. These bugs were not always uncovered by the probe with the minimum number of registers. One such example involved a problem with the aliasing mechanism used by the common subexpression elimination phase in *vpcc/vpo*. As the number of available registers decreased, a common subexpression previously held in a register had to be re-computed and, because the re-computation was improperly handled by the aliasing mechanism, the effects of a memory update were misrepresented. This caused a register value to be incorrectly used in lieu of the updated memory value in a subsequent instruction. This problem manifested itself only in some of the intermediate probes because the register containing the incorrect value was no longer available in the later probes and the correct value was fetched from the appropriate memory location instead.

Deficiencies in other parts of the optimizer were also uncovered, although these are not easy to characterize. The improved testing capability is derived from the fact that each probe in a register deprivation experiment effectively presents the optimization phases with a slightly mutated version of the original benchmark program. While not as effective as a completely different validation program, a mutated test has the ability to uncover problems that the original version may not. A mutated test also has the advantage of being quite similar to the original test, which is helpful when attempting to locate the point of failure. This capability allows the register deprivation technique to be used as an effective test tool. The nature of the deficiencies uncovered during register deprivation measurements suggest that register augmentation measurements, where the register set of a machine is artificially augmented, might provide similar testing capabilities.

4 Implementation

4.1 Compiler

Implementing register deprivation testing requires a compiler that can be specifically tailored to generate code that uses only a subset of the registers available on a target machine. For exhaustive register deprivation testing, this usually entails more than just reducing the number of register used by the register allocation algorithm. Modifications must also be made to the calling convention that determines the code sequences used to perform external function calls and to the code generator. Since the register deprivation process can be lengthy and repeated often, it should be automated. This is best accomplished by implementing the mechanism that controls the available registers so that it can be adjusted either with an option given to the compiler through the command line or a constant that can be easily changed before the compiler (or at least the modules that control register allocation) are compiled.

We implemented register deprivation measurement technique using the *vpcc/vpo* compiler. The *vpcc* front-end accepts traditional C [KR78] and generates stack-based, intermediate-language code, which is used directly by the code generator to generate machine-level code in the form of register transfer lists (RTLs). The back-end, *vpo*, improves this code to produce high-quality assembly-language code for the target machine. Most of the traditional optimizations, including local variable promotion, code motion, strength reduction, induction variable elimination, common subexpression elimination, evaluation order determination, dead variable elimination, constant folding and various control-flow optimizations [ASU86] are performed. Instruction scheduling and branch delay-slot filling, which is essential on some pipelined architectures, and recurrence optimization [BD91], which can significantly improve some scientific and digital signal processing codes, are also included.

4.2 Profile Tool

The object code produced by *vpcc/vpo* is instrumented to determine the number of instructions and memory references executed using *ease* [DW91b]. When *ease* is employed, the object code produced by the compiler is instrumented with execution counters. After the instrumented code executes, the values of these counters are written to a file along with instruction information for each basic block. A report generator uses this information to provide a detailed execution profile that includes the total number of instructions and memory references executed by the object code.

To facilitate the task of obtaining execution time measurements, *ease* provides a facility for measuring the amount of processor time spent executing a program. This facility operates on the same principle as the */bin/time* utility provided with most UNIX systems, but allows execution time results to be directed into a separate file other than the execution log file so that they can be more easily read by the graph generation utility program. Because the execution time information provided by UNIX systems can be affected by factors that are not easily controlled, a number of precautions have been taken to minimize the impact of these factors. First, to minimize the errors introduced by the fairly long interval between system clock ticks, all benchmarks were invoked with input data sets so that they executed for at least 30 seconds. To eliminate clock drift and other external effects, each program was executed five times, the highest and lowest times were discarded and the remaining three times were averaged to provide a single time value. Finally, an effort was made to identify and reduce the external factors that affect the times. These efforts included running the time trials on lightly-loaded machines and avoiding the use of dynamic-link library functions that could be overwritten by the system and whose reload time might be partially charged to the user. While these precautions increase the accuracy of the execution time measurements, the resolution provided cannot reveal performance differences of less than one or two percent. In such cases, the instruction and memory reference counts are the only accurate measures of performance.

4.3 Registers

Careful consideration must be given to determining which registers are eliminated from each successive probe in a register deprivation test. This task is complicated by the fact that register sets commonly provide more than one register type (e.g. general-purpose and floating-point registers). If the intent of the register deprivation testing is to examine only integer codes or an optimization that operates only on loops that manipulate floating-point values, then registers of one type only need to be considered. Since we are interested in measuring the effects of register deprivation in general, our implementation maintains the integer to floating-point register ratio as close to the original value as possible.

4.4 Calling Convention

On systems where the calling convention dictates the use of a caller-saves strategy, the registers are partitioned into those whose values may be arbitrarily changed across an external function call (volatile) and those whose values must not be affected by an external function call (non-volatile). Since the ratio of volatile to non-volatile registers can have a significant impact on code quality [DW91a], our register deprivation testing implementation also maintains the volatile to non-volatile register ratios as close to the original values as possible.

Many calling conventions pass actual parameter values in a subset of the volatile register set. Even though the number of registers reserved for parameter passing is usually small, the register deprivation trials will eventually reach the point where enough volatile registers are withheld that the remaining registers cannot implement the original calling convention. Because of this, the ability to modify the calling convention is essential to performing a thorough register deprivation experiment. The register deprivation experiments performed on the MIPS R3000, which passes argument values across functions in registers, adjust the number of registers used to pass argument values so that the ratio of the number of argument passing registers to total allocable registers remains fairly constant.

4.5 Probes

In order to maintain the register ratios discussed in the previous two sections, we start with the original set of allocable registers available to the compiler. This number is usually slightly less than the number of registers available on the target machine, since some registers are reserved for special functions (e.g., stack pointer, frame-pointer, etc.). To determine which register will be removed from each successive probe, a set of potential trials is considered where, for each type of register available, one of the volatile registers and one of the non-volatile registers are removed. The trial with the type-to-type and volatile-to-non-volatile ratios that are closest of the original values for these ratios is chosen. The process continues with the next successive probe until no additional registers can be withheld from the compiler. On most architectures, this point is usually reached when two registers for each register type remain, which is the minimum number of registers required to perform a binary operation. On machines where binary operations can be performed on memory locations, this limit may be lower.

Figure 7 shows which registers are available for each of the probes in the register deprivation experiments performed on a MIPS R3000. Of the forty-eight general-purpose and floating-point registers, only the forty-one registers that are available for local and global allocation by the compiler are shown. The remaining registers are reserved by

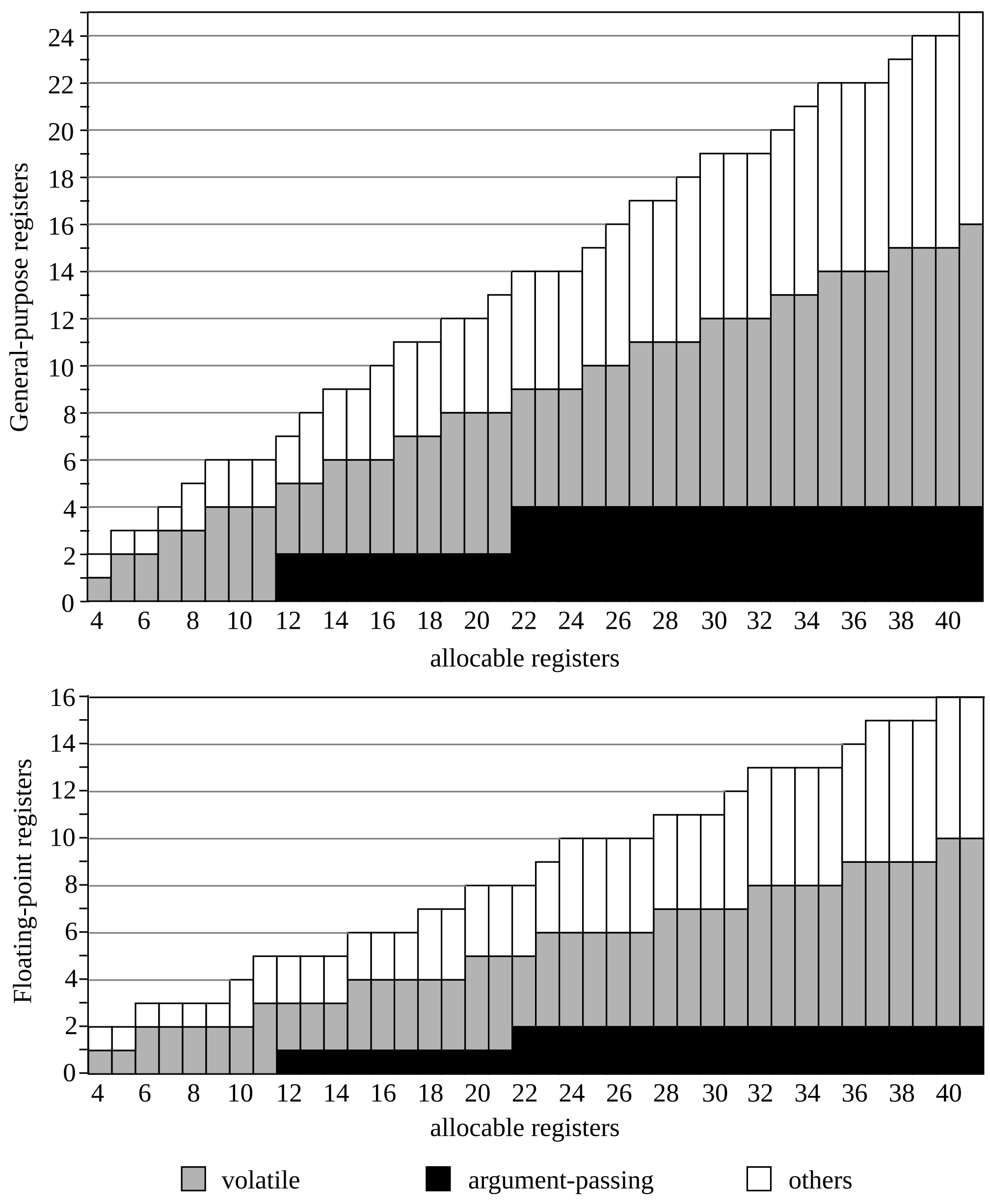


Figure 7: Registers available on each MIPS R3000 probe

the assembler or the operating system, are special in that they always return a fixed constant value, or are used exclusively as argument pointers or frame pointers.

Figure 8 shows the registers available for each of the probes in the register deprivation experiments performed on the Motorola 68020. This architecture provides three different classes of registers: address, data, and floating-point. Of the twenty-four total registers in these three classes, one of the address registers is reserved as both stack pointer and frame pointer. Because a pair of registers from each of the classes is required to generate code, the minimum number of registers used is six. Unlike the volatile to non-volatile register ratios for the address and floating-point register classes, the ratios for the data register class are not kept fairly constant because the calling convention uses the volatile data registers to return function values. In the final six register probe, it is important to note that although

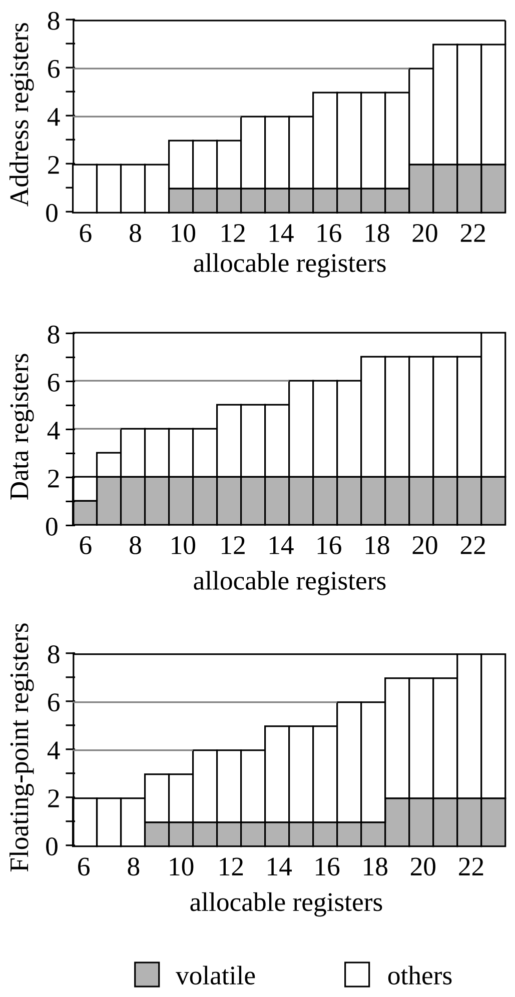


Figure 8: Registers available on each Motorola 68020 probe

there is only one allocable volatile data register, the functions that return floating-point values will actually use an additional volatile register in the return sequence. Because these code sequences make up a minute portion of the instructions executed in each probe, their effect on the outcome of the experiments are negligible.

4.6 Library Functions

On systems where the source code to the library functions is available, changes in the calling convention can be implemented by simply re-compiling the library. When the source code is either not available, or large portions are written in a language that does not allow the calling convention to be easily modified, a mechanism must be implemented to allow the use of the existing libraries. This mechanism entails generating special code before each external function call that passes actual parameters to a library function that expects argument values to arrive in specific registers. On some systems, it may not be possible to differentiate between calls to library functions and calls to user functions so that special interface code is needed before every function call. This special code introduces three problems:

- the code must be annotated so that it is not counted as part of the instructions or memory references that are executed by the program,

- the library code is not modified to reflect the code that would be generated if the register limits of the probe were fully enforced and
- the execution time of the programs is affected so that wall-clock execution times are not entirely accurate.

The first problem is trivially handled by *ease*. In order to prevent the second problem from affecting the results of instruction and memory reference count register deprivation test, none of the library codes are instrumented so that the number of instructions and memory references executed by the library functions are not counted in any of the probes in the register deprivation measurements. The last problem affects only the experiments that measure execution time, but this factor is not overwhelmingly significant because the benchmarks programs selected for these measurements spend only a small portion of their time executing library functions.

When generating code for a probe that limits the number of parameter passing registers, actual parameters that would normally be placed into these registers are placed in the memory locations normally reserved for actual parameter values that cannot be passed in registers. Special interface code, consisting of a series of uninstrumented load instructions to transfer actual parameter values to the registers prescribed by the standard calling convention, is generated between the point where these values are stored to memory and the external function call. The library function code is satisfied because it expects these parameter values to be in the registers that the special interface code has put them in. The register deprivation measurements that measure instruction and memory reference counts remain accurate because the execution counts are identical to those produced with a completely modified calling convention. The experiments that measure execution times are slightly affected by this approach.

Figure 9(a) shows a sample of the code that is generated on the MIPS R3000 for a probe with only four allocable registers. The source code consists of a simple call to a function that accepts two integer parameter values. Normally, the calling convention would pass the first actual parameter value in general-purpose register four and the second actual parameter value in general-purpose register five. Since both of these registers are withheld from the compiler in this probe, the calling convention is modified so that these actual parameter values are passed in a special location in the caller’s activation record. In the event that the function called is a library function still adhering to the original calling convention, the actual parameter values would not be passed properly. The special interface code, highlighted by the gray box, is generated to ensure that any function still following the original calling convention receives the actual parameter values correctly. Since the special interface code is not instrumented by *ease*, the execution counts generated when the code is executed reflects the instructions generated to satisfy the new calling convention. The execution time measurements, however, are not entirely accurate because they include the time required to execute the interface code.

In the C library, there are functions that call user-defined functions. An example of such a library function is *qsort()*. This function sorts an arbitrary array of data and one of the parameters passed to it is a pointer to a user-defined function that accepts a pair of pointers to elements in the array and returns a value that indicates which of the two items should be ordered first. If the library functions continue to rely on the original calling convention, the actual parameter values may be available in the registers specified by the original calling convention rather than in the memory locations from which the new code must load them. The solution once again is to introduce special interface code similar to the code required prior to an external function call. Figure 9(b) shows a sample of the interface code that is generated by the four register probe on the MIPS R3000. This code is placed at the entry point of functions that accept parameters in order to transfer the actual parameter values from the registers dictated by the original calling convention to memory to satisfy the requirements of the calling convention used by the probe.

The following C function call:

```
      max(15, 19)
```

Produces the following assembly code:

```
      li    $2,15
      li    $16,19
      sw    $2,0($sp)
      sw    $16,4($sp)
      lw    $4,0($sp)
      lw    $5,4($sp)
      jal   max
```

(a)

The following C function declaration:

```
      int max(a, b)
      int a, b;
      {
      }
```

Produces the following function prelude code:

```
max:  subu  $sp,$sp,8
      sw    $16,4($sp)
      sw    $4,8($sp)
      sw    $5,12($sp)
```

(b)

Figure 9: Calling convention interface code

4.7 Test Suite

The 13-program benchmark suite described in Table 4 was used to perform all of the register deprivation experiments shown here. These benchmarks were chosen to represent the kinds of codes that consume most of the cycles in a professional development and educational environment. The register deprivation results shown in Figure 2 took approximately 34 hours to collect on a DECstation 5000/125. Most of this time was spent compiling the test suite and the portions of *vpo/vpcc* that perform register allocation for each of the 37 probes in the trial. The remainder of the time was spent executing the test programs and collecting execution information. The process was completely automated using shell scripts to avoid human error.

Name	Description	Source	Type	Lines of C code
cache	Cache simulation	User code	I/O, Integer	820
compact	Huffman coding file compression	UNIX utility	I/O, Integer	490
diff	Text file comparison	UNIX utility	I/O, Integer	1,800
eqntott	PLA optimizer	SPEC benchmark	CPU, Integer	2,830
espresso	Boolean expression translator	SPEC benchmark	CPU Integer	14,830
gcc	Optimizing compiler	SPEC benchmark	CPU, Integer	92,630
iir	Infinite impulse response filter	Kernel benchmark	CPU, Integer	50
li	LISP interpreter	SPEC benchmark	CPU, Integer	7,750
linpack	Floating-point benchmark	Synthetic benchmark	CPU, Floating-point	930
mincost	VLSI circuit partitioning	User code	CPU, Floating-point	500
nroff	Text formatting	UNIX utility	I/O, Integer	6,900
sort	File sorting and merging	UNIX utility	I/O, Integer	930
tsp	Traveling salesperson problem	User code	CPU, Integer	450

Table 4: Benchmark suite

5 Summary

This paper has presented *register deprivation measurements*, a technique that can be used to gauge the effects of the register set on an optimizer as a whole or on a single optimization phase. It can be used to design and then test techniques to improve the performance of an optimizer on machines having small register sets or in circumstances where there is a high demand for registers. Our experience shows that register deprivation experiments enhance the ability of validation suites to reveal deficiencies in compiler systems. Register deprivation measurements can also be helpful in determining how many registers an optimizer can utilize effectively and provides information to a machine designer who may be interested in designing an architecture that will be supported primarily by an existing compiler. Compiler designers interested in developing new optimizations that take advantage of large register files can use register deprivation tests to determine how effectively each additional allocable register is utilized.

6 References

[ASU86] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, Reading, MA, 1986.

[BD88] M. E. Benitez and J. W. Davidson, A Portable Global Optimizer and Linker, *Proceedings of the SIGPLAN '88 Symposium on Programming Language Design and Implementation*, Atlanta, GA, June 1988, 329-338.

[BD91] M. E. Benitez and J. W. Davidson, Code Generation for Streaming: an Access/Execute Mechanism, *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, April 1991, 132-141.

[CAC+81] G. J. Chaitin, M.A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins and P. W. Markstein, Register Allocation via Coloring, *Computer Languages 6*, 1, January 1981, 47-57.

[CH84] F. C. Chow and J. L. Hennessy, Register Allocation by priority-based coloring, *ACM SIGPLAN Notices, 19*, 6, June 1984, 222-232.

[Dav86] J. W. Davidson, A Retargetable Instruction Reorganizer, *Proceedings of the SIGPLAN '86 Symposium on Programming Language Design and Implementation*, Palo Alto, CA, June 1986, 234-241.

[DW91a] J. W. Davidson and D. B. Whalley, Methods for Saving and Restoring Register Values across Function Calls, *Software—Practice and Experience 21*, 2, February 1991, 149-165.

[DW91b] J. W. Davidson and D. B. Whalley, A Design Environment for Addressing Architecture and Compiler Interactions, *Microprocessors and Microsystems*, 15, 9, November 1991, 459-472.

[FH92] C. W. Fraser and D. R. Hanson, Simple Register Spilling in a Retargetable Compiler, *Software—Practice and Experience 22,* 1, January 1992, 85-99.

[HP90] J. L. Hennessy and D. A. Patterson, *Computer Architecture, A Quantitative Approach*, Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[Kan87] G. Kane, *MIPS RISC Architecture*, Prentice-Hall, Englewood Cliffs, NJ, 1987.

[KR78] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1978.

[Mot85] Motorola, *MC68020 32-Bit Microprocessor User's Manual*, Prentice-Hall, Englewood Cliffs, NJ, 1985.

[Wal86] D. W. Wall, Global Register Allocation at Link-time, *Proceedings of the SIGPLAN '86 Symposium on Programming Language Design and Implementation*, Palo Alto, CA, June 1986, 264-275.