Scalar Program Performance on Multiple-Instruction-Issue Processors with a Limited Number of Registers

Scott A. Mahlke *William Y. Chen* *Pohua P. Chang* *Wen-mei W. Hwu*

Center for Reliable and High-Performance Computing
University of Illinois
Urbana, IL 61801

Abstract

In this paper the performance of multiple-instruction-issue processors with variable register file sizes is examined for a set of scalar programs. We make several important observations. First, multiple-instruction-issue processors can perform effectively without a large number of registers. In fact, the register files of many existing architectures (16–32 registers) are capable of sustaining a high instruction execution rate. Second, even for small register files (8–12 registers), substantial performance gains can be obtained by increasing the issue rate of a processor. In general, the percentage increase in performance achieved by increasing the issue rate is relatively constant for all register file sizes. Finally, code transformations designed for multiple-instruction-issue processors are found to be effective for all register file sizes; however, for small register files, the performance improvement is limited due to the excessive spill code introduced by the transformations.

1 Introduction

The design of multiple-instruction-issue processors provides a computer architect with a large variety of features to consider. An important parameter is the size of the register file. By increasing both the number of data dependencies and the amount of spill code, a small register file reduces the number of concurrently executable instructions available to a multiple-instruction-issue processor. Therefore, the performance of a multiple-instruction-issue processor may be severely restricted if an insufficient register file is used. However, architects who are extending a family of processors are constrained by an existing register file configuration in order to maintain instruction set compatibility. Many of these existing processors have small register files. Also, in the design of new multiple-instruction-issue processors, cost and design constraints may restrict the size of the register file.

In this paper, the performance of multiple-instruction-issue processors with small and moderate numbers of registers is analyzed for a set of scalar programs. Several important considerations are addressed. First, are moderate sized register files (16–32 registers) capable of sustaining the performance of a multiple-instruction-issue processor? Second, for processors with small register files (8–12 registers), does an increase in the issue rate result in significant performance improvements? Finally, are code transformations designed for multiple-instruction-issue processors effective for a processor with a limited register file? In this paper, only the integer register file is evaluated, however the issues discussed are applicable to both integer and floating point register sets.

1.1 Related Work

Previous researchers have studied the effect of register file size and structure on the performance of RISC processors. Chow and Hennessy propose and evaluate priority-based graph coloring to perform global register allocation [1]. They also evaluate the register file configuration of a RISC processor and conclude that a split configuration of caller-saved and callee-saved registers provides the best performance. Goodman and Hsu show that an integrated code generation strategy in which code scheduling is performed along with register allocation outperforms simple postpass scheduling for scientific programs on a single-issue processor [2]. Bradlee, Eggers, and Henry study the effect of code generation strategy on register file requirements for a single-issue processor [3]. They find that an integrated strategy requires fewer registers than simple postpass scheduling, and that with the integrated strategy little performance gains result from register files larger than 32 registers. Chang, Lavery, and Hwu show the effectiveness of prepass code scheduling for integer scalar programs on multiple-instruction-issue processors [4].

There has also been significant research evaluating other architectural features for multiple-instruction-issue processors. Butler *et al.* investigated the exploitable parallelism under varying hardware constraints including the dynamic window size, the number of functional units, and the branch prediction strategy [5]. Sohi and Vajapeyam examined tradeoffs in the instruction format design for VLIW processors [6]. Cohn *et al.* studied the resource performance tradeoffs for the iWarp processor [7]. Chang *et al.* investigated the effect of the scheduling model on the performance of multiple-instruction-issue processors [8].

1.2 Organization of the Paper

The remainder of this paper consists of three sections. In Section 2, three performance issues are examined for multiple-instruction-issue processors with a limited number of registers. In Section 3, an experimental evaluation of the performance of multiple-instruction-issue processors with varying register file configurations is presented. Finally, some concluding remarks are made in Section 4.

2 Register File Size Issues

Many existing processors contain a small number of integer registers. Table 1 describes the integer register files for several common processors. With the exception of the AMD 29k processor, the total number of registers ranges from 8 to 32, but the number of registers available to the register allocator is actually much less. For example, for the MIPS R2000, of the 32 integer registers, 7 are reserved for special purposes (e.g., stack pointer, constant values, operating system use, etc.) and 5 are reserved for parameter passing across subroutine calls.[1] Therefore, only 20 unrestricted registers are available for use by a register allocator. The other processors have similar restrictions.

Three problems that a small register file introduces are increased data dependencies, a large amount of spill code, and reduced effectiveness of compiler optimizations. Each of these problems and their impact on performance will be discussed in the remainder of this section.

2.1 Effect of Data Dependencies

Data dependencies limit the amount of instruction reordering a code scheduler can perform. For single-instruction-issue processors, a scheduler inserts independent instructions after instructions requiring multiple cycles to execute to reduce pipeline interlocks. For multiple-instruction-issue processors, a scheduler must identify independent instructions that can be concurrently executed in addition to reducing interlocks. High issue rate processors provide performance improvements only when the scheduler is able to find sufficient instructions to concurrently execute.

When the number of registers a processor contains is limited, additional data dependencies are introduced when registers are recycled. For smaller register files, recycling occurs frequently, thereby significantly increasing the number of data dependencies. The number of additional data dependencies introduced can be reduced if code scheduling is performed before register allocation [2] [4] [3]. However, one should expect the performance of higher issue rate processors to suffer more performance loss when there are a large number of data dependencies.

[1] For many of these processors, register allocators can often utilize parameter registers between function calls and several of the reserved registers as additional caller save registers.

2.2 Effect of Spill Code

Spilling, in general, occurs when the number of overlapping live variables (program variables and compiler temporaries) exceeds the total number of available registers. A small register file will result in a large number of spills in program segments with moderate numbers of live variables. A register is spilled by temporarily storing its contents into memory, thereby freeing the register to hold other values. When the spilled variable is required for use, it is reloaded into a register from memory. For single-instruction-issue processors, spill code almost always results in performance loss.

For multiple-instruction-issue processors, however, it may be possible to hide much of the spill code in empty instruction slots. The ability of multiple-instruction-issue processors to hide spill code is dependent on two issues. First, if there are no available instruction slots to place the spill code, increasing the issue rate of the processor can create the necessary empty slots to hide the spill code. For spill code of this type, higher issue rate processors will tolerate more spill code. Second, if the spill occurs for a variable which is accessed on a region's critical path, additional cycles will be introduced to the program execution. In this case, the spill code cannot be hidden by a multiple-instruction-issue processor. For spill code of this type, the total cycle count of all issue rates will be affected relatively equally for a given amount of spill code.

2.3 Effectiveness of Compiler Optimizations

Many compiler optimizations increase the register file requirements of a program [9]. Some optimizations, such as common subexpression elimination, extend the live range of a variable by preserving the result of a previous expression for a longer period in order to avoid the recomputation of that expression. Other optimizations such as loop invariant code removal, move the computation of an expression to a less frequently executed program region. As a side effect of this movement, the live range of the register holding the result of the computation is extended. Optimizations designed for multiple-instruction-issue processors increase the register file requirements of a program more rapidly. For example, a combination of loop unrolling and induction variable expansion will create $N-1$ new induction variables (assuming the loop is unrolled N times) for each induction variable originally in the loop.

Much of the performance improvements achieved by compiler optimizations may be lost if the optimizations increase the number of live variables such that excessive spilling occurs. Multiple-instruction-issue optimizations, in particular, may be very ineffective for machines with a small number of registers.

Processor	*Registers*	*Comments*
MIPS R2000	32	7 reserved, 5 parameter passing, 11 caller save, 9 callee save
SPARC	32	Fixed size window 2 reserved, 14 parameter passing, 8 local; 8 global
Intel i860	32	5 reserved, 11 parameter passing, 12 callee, 4 caller
Intel i486	8	1 reserved, 7 general purpose
Motorola 68k	16	1 reserved, 8 data, 7 address
Motorola 88k	32	8 reserved, 8 parameter passing, 16 general purpose
AMD 29k	192	128 in a variable overlapping window; 43 reserved, 21 general purpose

Table 1: Integer register file configurations for existing processors.

Benchmark	*Size*	*Benchmark Description*	*Input Description*
eqntott	3461	boolean equation minimization	5 files of boolean equations
espresso	6722	truth table minimization	20 original espresso benchmarks
lex	3316	lexical analyzer generator	5 lexers for C, lisp, pascal, awk, pic
xlisp	7747	lisp interpreter	5 gabriel benchmarks
yacc	2303	parser generator	10 grammars for C, pascal, pic, eqn

Table 2: Benchmarks.

3 Experiments

In this section, an empirical evaluation of the performance of multiple-instruction-issue processors with varying number of registers and varying level of compiler optimization is presented. Each experiment consists of compiling a set of integer benchmarks with a particular optimization level for a target architecture.

3.1 IMPACT-I C Compiler

We have developed IMPACT-I, a prototype retargetable C compiler with classical and multiple-instruction-issue optimization capability. The IMPACT-I C compiler contains a complete set of classical local, global, and loop optimizations [9] [10], profile-guided inline function expansion [11], profiled-guided instruction placement [12], and profile-guided branch prediction [13]. Code is generated using a variant of integrated prepass scheduling [2] [4], in which prepass scheduling is performed, followed by register allocation, and followed lastly by postpass scheduling. Register allocation is performed using a variant of priority-based graph coloring [1] [14]. Profile information and static loop analysis are used to identify the most frequently accessed variables to the register allocator.

In order to calibrate the quality of the classical optimizations, we compare the execution times of the code generated by our compiler and a leading commercial compiler, the MIPS C compiler[2], on a DEC 3100 workstation. The benchmarks used in this study are shown in Table 2. The *Size* column specifies the size of each program in number of lines of C source code. Some of the compiler optimizations require profile information. The inputs used for profiling each benchmark are also described in Table 2. Table 3 shows the speedup we obtain over the MIPS C compiler using its highest degree of optimization. The evaluations presented in this paper are thus based on very efficient code.

The IMPACT-I C compiler also contains a set of multiple-instruction-issue optimizations which are designed to expose more concurrently executable instructions [15]. These optimizations include superblock formation, loop unrolling, loop peeling, branch target expansion, induction variable expansion, accumulator expansion, and register renaming. A superblock is the basic scope for optimizations. Superblock formation consists of first combining basic blocks which execute in sequence into a trace [16], and then performing code duplication to eliminate all side entrances from the trace.

Loop unrolling replicates the body of a loop several times. Loop peeling fully unrolls loops with small numbers of iterations. Branch target expansion copies the target superblock of a frequently taken branch into its fall-through path. Induction variable expansion removes the dependencies between induction variables in unrolled copies of a loop body. Accumulator expansion removes the flow dependencies between accumulator variables in unrolled copies of a loop body. Register renaming is used to remove anti and output dependencies between instructions in a superblock.

3.2 Processor Architecture

The code generator takes as input a machine description file that characterizes the instruction set, the microarchitecture (including the number of instructions that can be issued in a cycle and the instruction latencies), the number of registers, and the code scheduling model. The underlying microarchitecture is assumed to have in-order ex-

[2]MIPS CC release 2.1

Benchmark	*MIPS -O4*	*IMPACT*
eqntott	1.0	1.04
espresso	1.0	1.02
lex	1.0	1.01
xlisp	1.0	1.13
yacc	1.0	1.01

Table 3: Speedup comparison.

Instruction Class	*Latency*
integer ALU	1
barrel shifter	1
integer multiply	3
integer divide	25
branch (correctly predicted)	1
branch (incorrectly predicted)	2
memory load	2
memory store	-
FP ALU	3
FP conversion	3
FP multiply	4
FP divide	25

Table 4: Instruction latencies.

ecution and deterministic instruction latencies (Table 4). The instruction set is a RISC assembly language which is a superset of the MIPS R2000 instruction set. The microarchitecture uses a squashing branch scheme with profile-based branch prediction. One branch slot is allocated by the compiler for each predicted-taken branch.

The basic processor is assumed to support general code percolation scheduling [8]. General code percolation requires the processor to contain a complete set of non-trapping instructions. In this manner, the compiler can move independent loads, divides, and floating point instructions above branches. General code percolation has been shown to provide significant performance improvements beyond a restricted code percolation scheme that does not allow instructions which may cause traps to be moved above branch instructions [8].

3.3 Results

The experiments in this section evaluate the performance of multiple-instruction-issue processors with varying register file size. The performance with two levels of compiler optimization, level 1 (lv 1) and level 2 (lv 2), is presented. Level 1 optimization includes all classical optimizations previously discussed along with superblock code scheduling. Level 2 optimization includes all other previously discussed multiple-instruction-issue optimizations applied in addition to the level 1 optimizations. A base machine configuration with an issue rate of 1, 4 registers, and level 1 compiler optimization will be used.

The number of registers represents the total number of registers seen by the register allocator. Some processors have additional integer registers reserved to hold constant values, for the frame pointer, for parameter passing, and many other things. Registers for these purposes are not included in the number of registers. The register allocator reserves 4 of the total registers to handle spilling. Therefore for the base machine which has only 4 registers, all of the registers are used to handle spills. The remaining registers are divided into two equal groups of caller-saved and callee-saved registers.

For each compilation, the program execution time and the number of dynamic memory accesses (assuming a 100% cache hit rate) are calculated. In order to calculate this information, the program is re-profiled using an input different from those it was originally profiled with. The execution time is derived by summing the worst case execution times of each superblock. The worst case is due to long instruction latencies that protrude from one superblock to another. The execution time for each benchmark B and machine configuration is reported as a speedup relative to benchmark B executing on the base machine configuration.

The number of dynamic memory accesses is the sum of the execution frequencies of all loads and stores in the program. It is reported as the *memory access ratio* (MAR), which is defined as the number of dynamic memory accesses for benchmark B for a particular machine configuration divided by the number of dynamic memory accesses for benchmark B executing on the base machine configuration. The MAR is an indication of the demands placed upon the memory system.

Figures 1 – 20 show the effect of varying the register file size on the speedup and the MAR for each benchmark described in Table 2. The number of registers is varied from 4 to 48 and the issue rate is varied from 1 to 8. By issue rate we mean that the processor can issue up to that many instructions per cycle. No limitation has been placed on the combination of instructions that can be issued in the same cycle.

Effect of Register File Size

From the figures it can be seen that for all issue rates the speedup and the MAR stabilize at a relatively small number of registers. For all benchmarks, little performance improvement is observed for register files larger than 24 registers. Two of the benchmarks, *eqntott* with optimization levels 1 and 2 and *lex* with optimization level 1, achieve little improvement even beyond 12 registers. Our results indicate that multiple-instruction-issue processors can perform effectively without a large register file. In fact, the register files of many existing processor architectures are capable of supporting a high instruction execution rate.

Increased data dependencies do significantly reduce the performance for all issue rates for register files smaller than 12 registers. However, processors which are con-

strained to a very small register file can still obtain substantial performance improvements by extending to multiple-instruction-issue. The speedup gain due to increasing the issue rate, in general, is independent of the register file size. For example consider *espresso* with level 2 optimization (Figure 7), increasing the issue rate from 1 to 4 with 8 registers results in a 63% gain in speedup, while increasing the issue rate from 1 to 4 with 32 registers results in a 66% gain in speedup. Similarly, for *yacc* with level 2 optimization (Figure 19), increasing the issue rate from 1 to 4 with 8 registers results in a 78% gain in speedup, while increasing the issue rate from 1 to 4 with 32 registers results in a 86% gain in speedup. The overall performance of a multiple-instruction-issue processor with a larger number of registers is higher than that with a small number of registers, however the percentage increase in performance achieved by increasing the issue rate is relatively constant for all register file sizes.

Effect of Spill Code

It can also be seen that for each benchmark and optimization level the speedup and the MAR stabilize at approximately the same number of registers for all issue rates. The spill code, therefore, is not effectively hidden in empty instruction slots by multiple-instruction-issue processors. Most of the spilled variables, in fact, occur on the critical path of a region, adding to the total execution cycles. Thus, the performance for all issue rates and a given register file size is similarly restricted by the spill code.

Effect of Multiple-Instruction-Issue Optimizations

The effect of multiple-instruction-issue optimizations can be seen by comparing the speedup and MAR curves for level 1 and level 2 optimization for each benchmark. The most dramatic change occurs for *lex*. The maximum speedup for *lex* increases from (3, 4, 4) to (5, 9, 12) for issue rates (2, 4, 8), respectively. However, as a side effect of the multiple-instruction-issue optimizations, the number of registers required to stabilize the performance increases from 12 to 24. This behavior could be anticipated, though. The multiple-instruction-issue optimizations create significantly more overlapping live variables, therefore the performance of processors with small register files will be limited due to the excessive spilling. The increase of spill code is shown clearly by comparing the MAR in Figures 10 and 12. Similar behavior is also observed for *yacc*. The behavior of *xlisp* is completely opposite to that of either *lex* or *yacc*. The multiple-instruction-issue optimizations reduce the MAR for issue 4 and 8 and effectively reduce the number of registers required to stabilize performance. Level 2 optimization has little effect on either the speedup or the MAR for *eqntott* and *espresso*.

The results, however, do not indicate that multiple-instruction-issue optimizations should not be performed for processors with small register files. As an example, for *yacc* with 8 registers and issue 4, a 14% gain in speedup results from level 2 optimization, and with 16 registers and issue 4, a 13% gain results from level 2 optimization. The maximum speedup resulting from multiple-instruction-issue optimizations for a small register file is limited due to the excessive spill code introduced, however, performance improvements can still be obtained with the compiler optimizations.

The set of multiple-instruction-issue optimizations considered in this paper is not complete. Additional optimization techniques can further increase performance; however, they are also likely to increase the register file requirements. The performance gain of multiple-instruction-issue, though, is already substantial. For example, with 24 registers and an issue rate of 4, all benchmarks execute on average over 2 instructions per cycle. Thus, although the set of optimization techniques considered is not complete, the register file configurations reported in this paper are capable of supporting high performance multiple-instruction-issue processors.

Issue Rate Versus Register File Size

The tradeoffs in speedup between issue rate and the number of registers is also interesting to examine. In general, an increase in issue rate results in larger performance gains than an increase in register file size. For large enough register files (e.g., larger than 24 registers) this trend can be easily seen since the speedup changes very little as the number of registers increases. However, this behavior also occurs for small register files. For example consider *eqntott* with level 1 optimization (Figure 1), increasing the number of registers for issue 1 from 8 to 48 results in a 44% gain in speedup, while increasing the issue rate from 1 to 2 with 8 registers results in a 54% gain in speedup and increasing the issue rate from 1 to 4 with 8 registers results in a 103% gain in speedup. Yet this trend does not hold for all cases. Consider *espresso* with level 2 optimization (Figure 7). An increase in issue rate from 2 to 4 with 8 registers results in a 11% gain in speedup, while increasing the number of registers from 8 to 16 for issue 2 results in a 43% gain in speedup. For all benchmarks, though, increasing both the issue rate and the number of registers will result in the largest performance gains. However, to achieve a certain level of speedup a designer can tradeoff the effects of register file size and issue rate.

4 Conclusions

The performance of multiple-instruction-issue processors with a limited number of registers for a set of integer scalar programs has been examined. Our results indicate that multiple-instruction-issue processors can perform effectively without a large number of registers. This enables many existing architectures with moderately sized register files to achieve substantial performance improvements by extending to multiple-instruction-issue. For those architectures with small register files, an extension

to multiple-instruction-issue can also provide significant speedup. Our results show that in general the increase in speedup due to increasing the issue rate is independent of the register file size. We also evaluate the effectiveness of a set of code transformations for multiple-instruction-issue processors with varying sized register files. The transformations are found to be effective for all register file sizes. However, the performance improvement for small register files is limited due to the excessive spill code introduced.

In this paper, significant performance improvements were observed for multiple-instruction-issue processors with small and moderate sized register files over their single-instruction-issue predecessors. A large amount of work in the area of multiple-instruction-issue processor performance with limited register file sizes still needs to be done. We are currently examining the performance implications of the floating point register file, additional classes of benchmarks, and further compiler optimization techniques.

Acknowledgements

The authors would like to thank Nancy Warter, Roger Bringmann, Richard Hank, Tokuzo Kiyohara, and all members of the IMPACT research group for their support, comments, and suggestions. This research has been supported by the National Science Foundation (NSF) under Grant MIP-8809478, Dr. Lee Hoevel at NCR, the AMD 29K Advanced Processor Development Division, the National Aeronautics and Space Administration (NASA) under Contract NASA NAG 1-613 in cooperation with the Illinois Computer laboratory for Aerospace Systems and Software (ICLASS).

References

[1] F. C. Chow and J. L. Hennessy, "The priority-based coloring approach to register allocation," *ACM Transactions on Programming Languages and Systems*, vol. 12, pp. 501–536, October 1990.

[2] J. R. Goodman and W. C. Hsu, "Code scheduling and register allocation in large basic blocks," in *Proceedings of the 1988 International Conference on Supercomputing*, pp. 442–452, July 1988.

[3] D. G. Bradlee, S. J. Eggers, and R. R. Henry, "The effect on risc performance of register set size and structure versus code generation strategy," in *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 330–339, May 1991.

[4] P. P. Chang, D. M. Lavery, and W. W. Hwu, "The importance of prepass code scheduling for superscalar and superpipelined processors," Tech. Rep. CRHC-91-18, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, May 1991.

[5] M. Butler, T. Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow, "Single instruction stream parallelism is greater than two," in *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 276–286, May 1991.

[6] G. S. Sohi and S. Vajapeyam, "Tradeoffs in instruction format design for horizontal architectures," in *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 15–25, April 1989.

[7] R. Cohn, T. Gross, M. Lam, and P. S. Tseng, "Architecture and compiler tradeoffs for a long instruction word microprocessor," in *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 2–14, April 1989.

[8] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "Impact: An architectural framework for multiple-instruction-issue processors," in *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 266–275, May 1991.

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

[10] S. A. Mahlke, "Design and implementation of a portable global code optimizer," Master's thesis, University of Illinois, Urbana, IL, 1991.

[11] W. W. Hwu and P. P. Chang, "Inline function expansion for compiling realistic c programs," in *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, pp. 246–257, June 1989.

[12] W. W. Hwu and P. P. Chang, "Achieving high instruction cache performance with an optimizing compiler," in *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pp. 242–251, May 1989.

[13] W. W. Hwu, T. M. Conte, and P. P. Chang, "Comparing software and hardware schemes for reducing the cost of branches," in *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pp. 224–233, May 1989.

[14] G. J. Chaitin, "Register allocation and spilling via graph coloring," in *Proceedings of the ACM SIGPLAN 82 Symposium on Compiler Construction*, pp. 98–105, June 1982.

[15] P. Chang, *Compiler Support for Multiple Instruction Issue Architectures*. PhD thesis, University of Illinois, Urbana, IL, 1991.

[16] J. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, vol. c-30, pp. 478–490, July 1981.

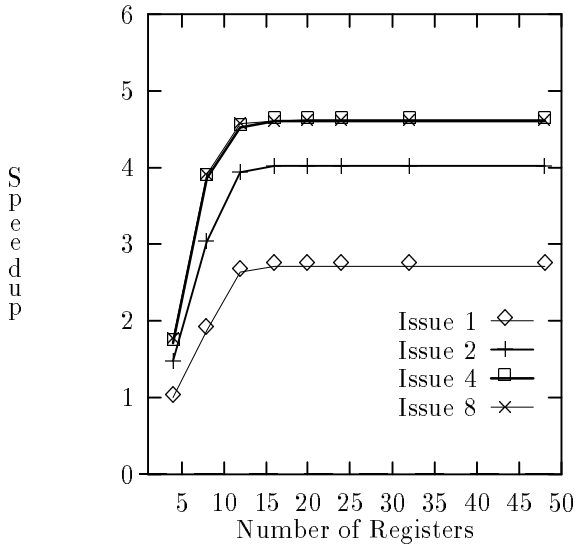

Figure 1: Speedup for *eqntott* (lv 1 opt).

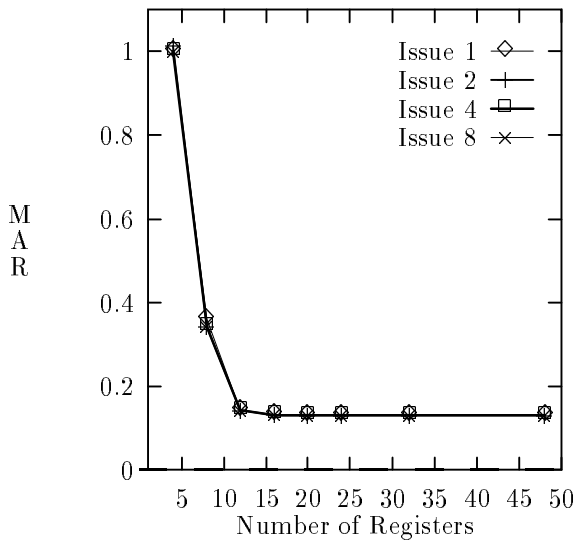

Figure 2: MAR for *eqntott* (lv 1 opt).

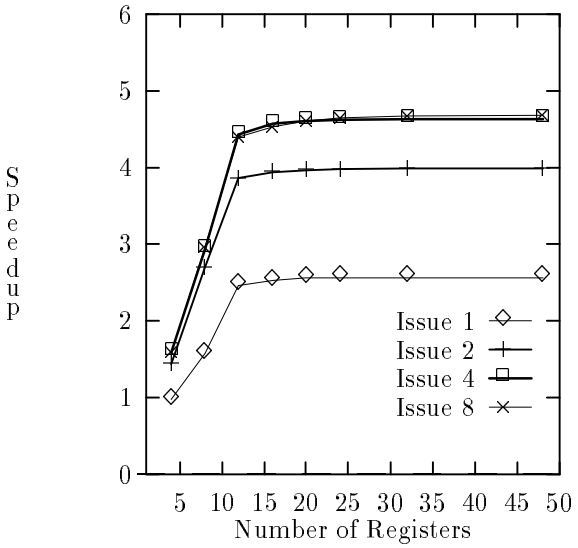

Figure 3: Speedup for *eqntott* (lv 2 opt).

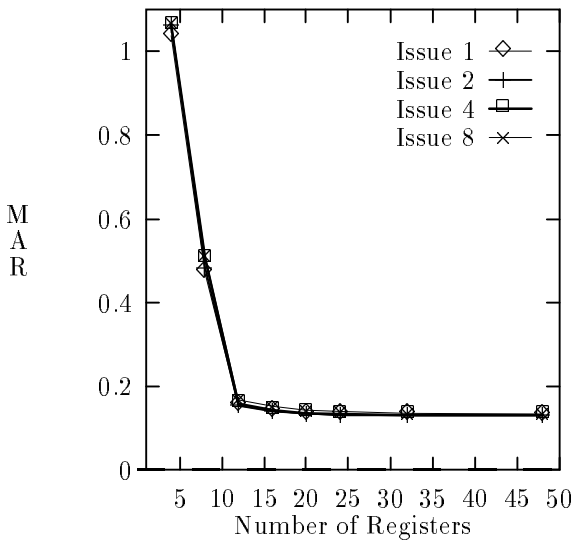

Figure 4: MAR for *eqntott* (lv 2 opt).

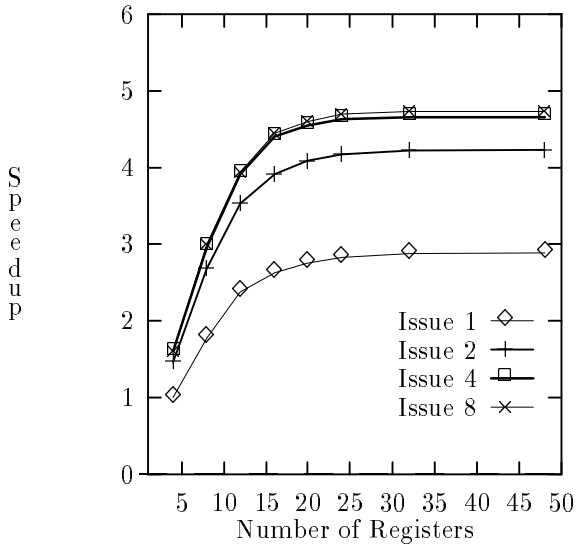

Figure 5: Speedup for *espresso* (lv 1 opt).

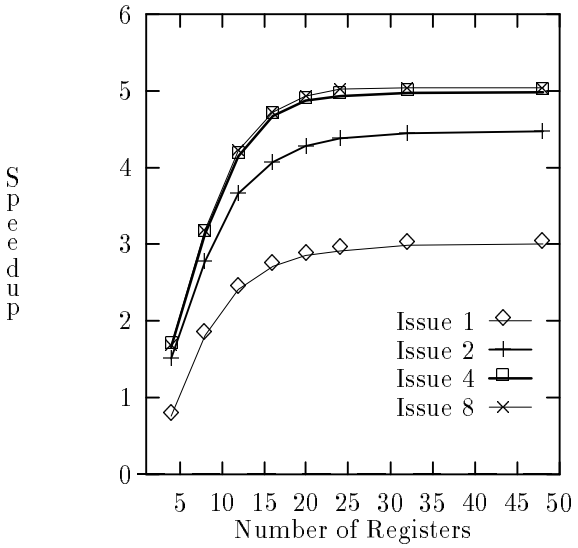

Figure 7: Speedup for *espresso* (lv 2 opt).

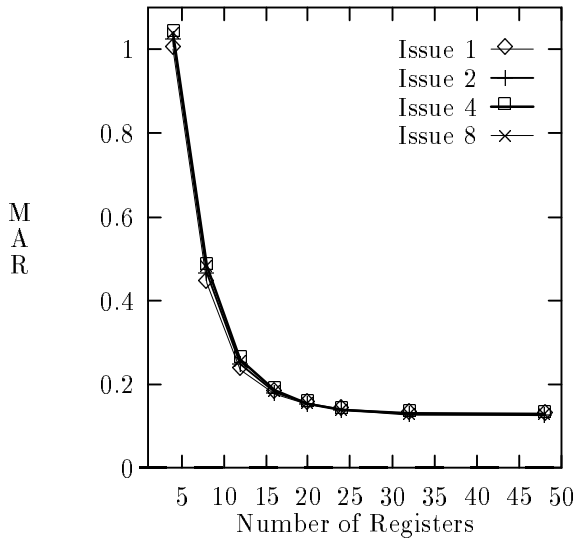

Figure 6: MAR for *espresso* (lv 1 opt).

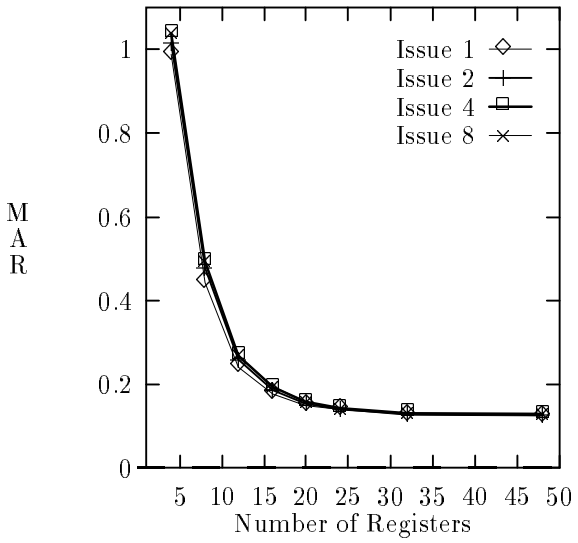

Figure 8: MAR for *espresso* (lv 2 opt).

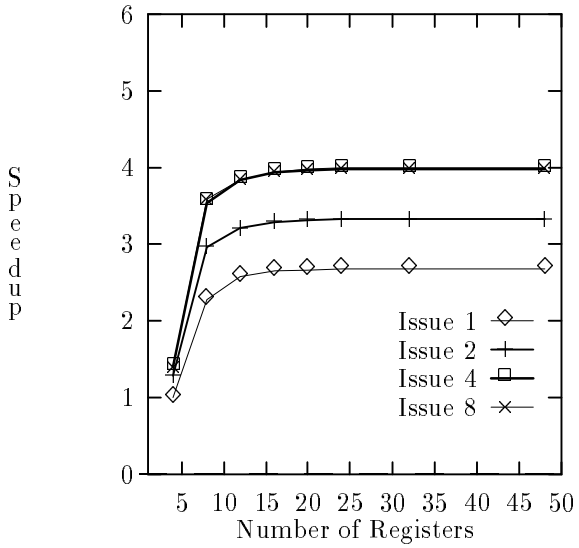

Figure 9: Speedup for *lex* (lv 1 opt).

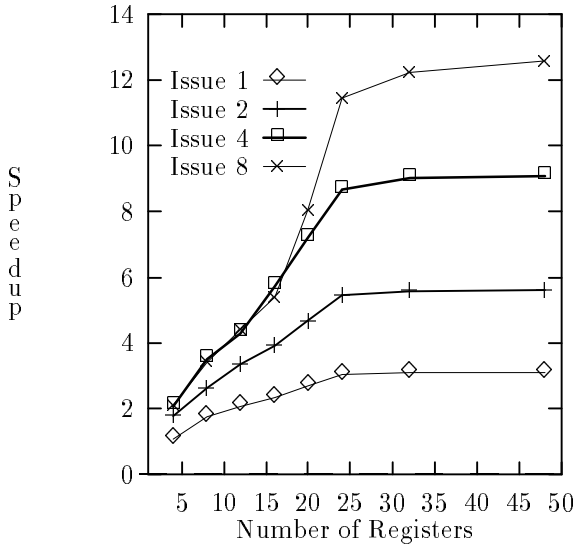

Figure 11: Speedup for *lex* (lv 2 opt).

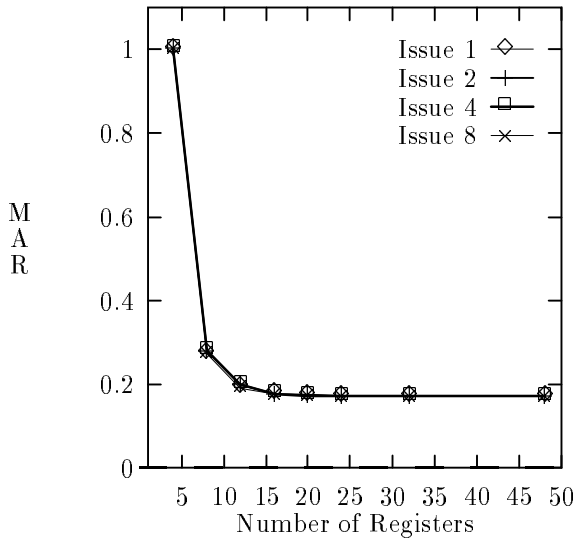

Figure 10: MAR for *lex* (lv 1 opt).

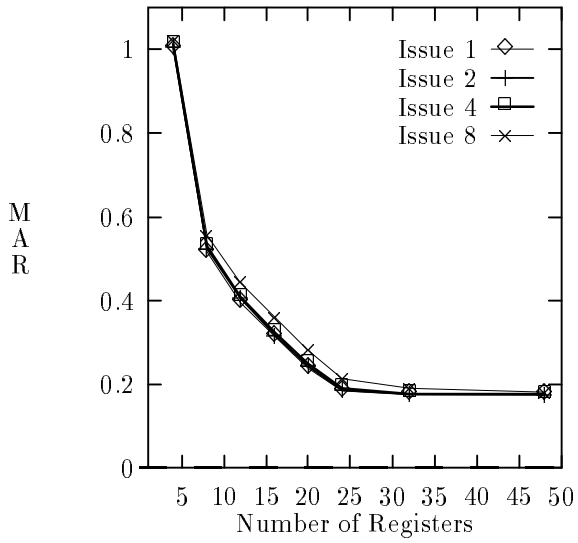

Figure 12: MAR for *lex* (lv 2 opt).

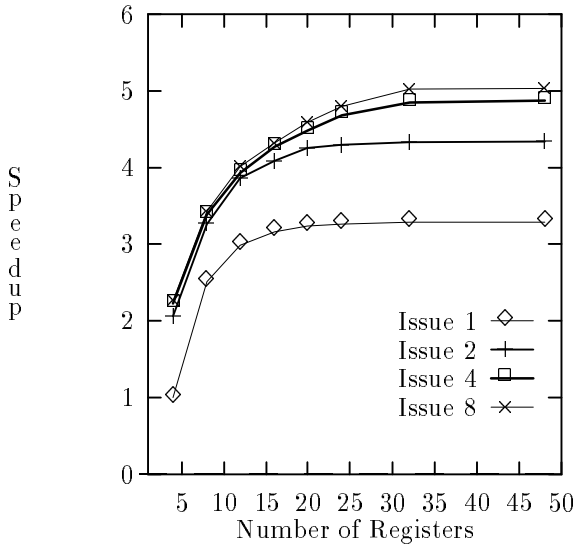

Figure 13: Speedup for *xlisp* (lv 1 opt).

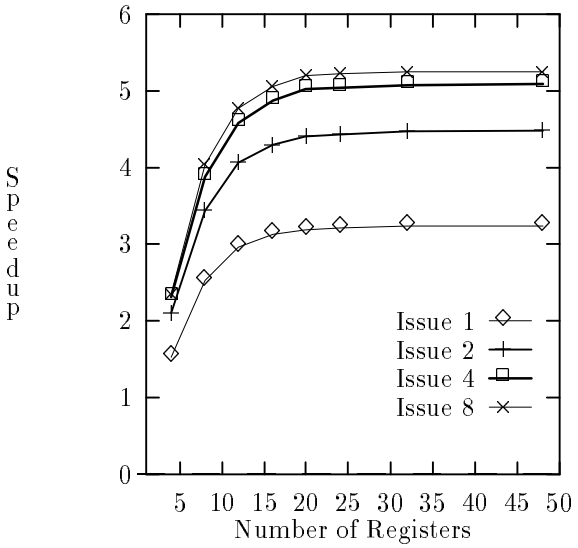

Figure 15: Speedup for *xlisp* (lv 2 opt).

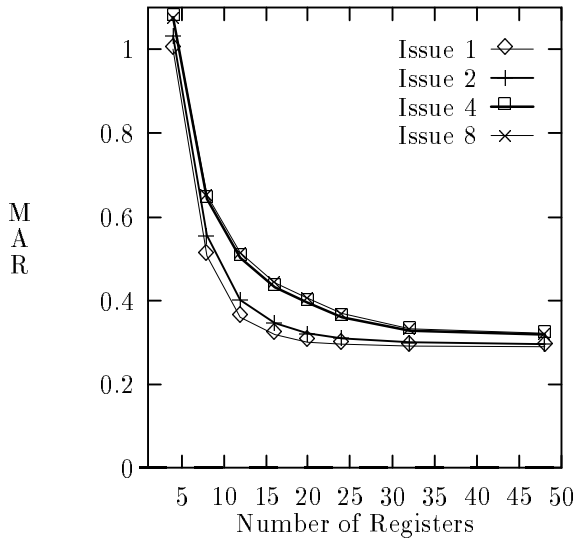

Figure 14: MAR for *xlisp* (lv 1 opt).

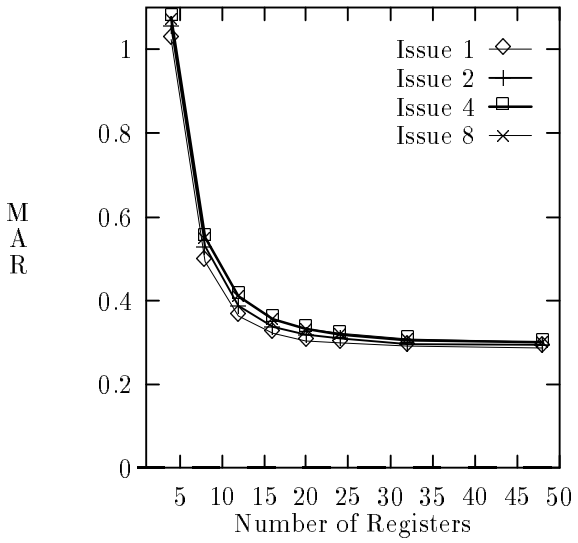

Figure 16: MAR for *xlisp* (lv 2 opt).

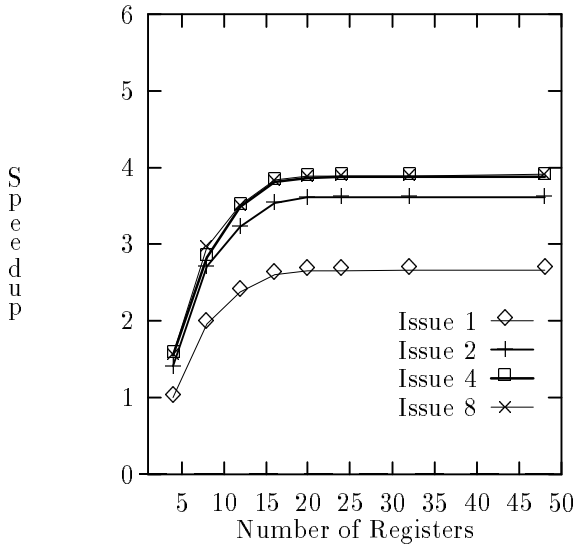

Figure 17: Speedup for *yacc* (lv 1 opt).

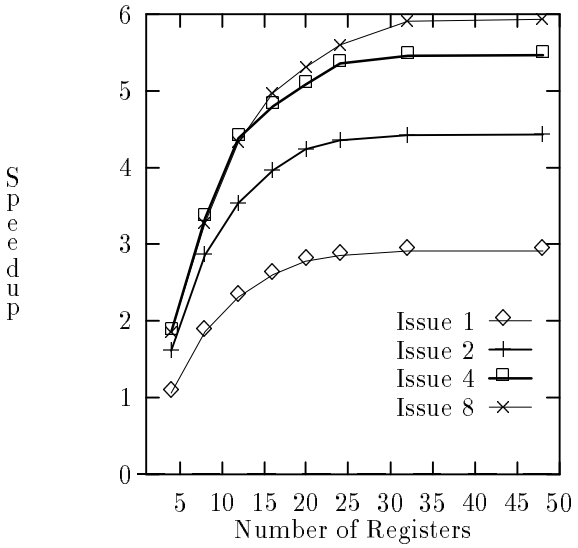

Figure 19: Speedup for *yacc* (lv 2 opt).

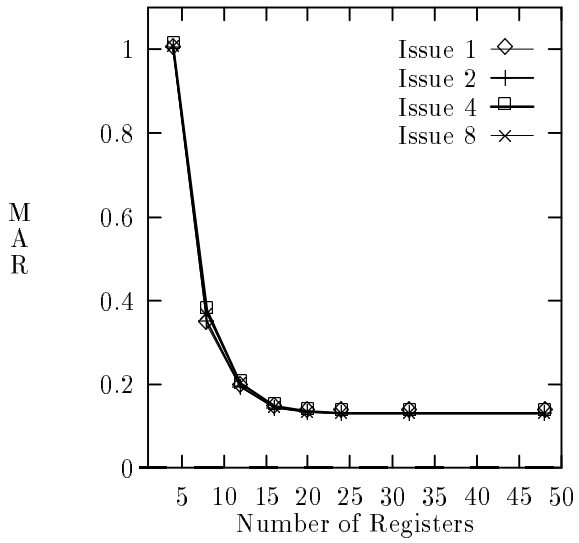

Figure 18: MAR for *yacc* (lv 1 opt).

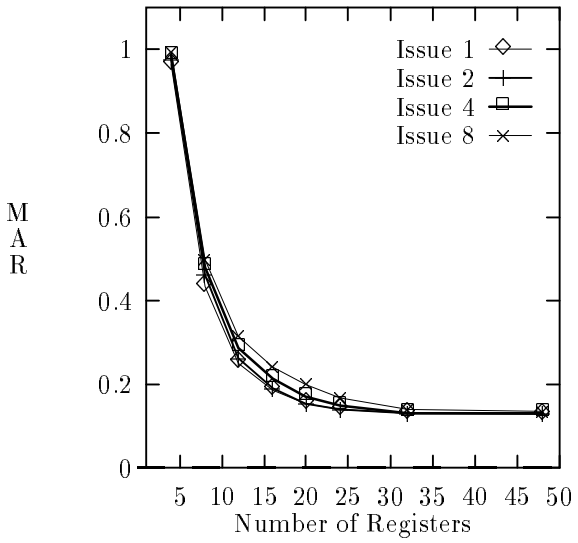

Figure 20: MAR for *yacc* (lv 2 opt).