

Dynamic Register Reassignment in Strata

Michele Co and David Larochele
Department of Computer Science
University of Virginia
{mc2zk,drl7x}@cs.virginia.edu

Abstract

Register assignment is the phase of register allocation that decides what values will be put into which registers. Because optimal register assignment has been shown to be an NP-complete problem, heuristics are used.

Register assignment is usually done statically at compile time, where there is no knowledge of actual execution patterns. During actual execution, there is more information about frequently executed paths which may benefit from dynamic register reassignment.

In this project we implement a naive linear-scan register reassignment algorithm for fragments. We evaluate the opportunity for optimization, the algorithm's overhead and the execution time improvement on eight of the SPEC2000 integer benchmarks. Naive fragment register liveness calculation functionality is added as well.

We find that the opportunity for register reassignment with our simple algorithm is very small, roughly less than 20% of all moves considered for reassignment. Of the moves which can be eliminated, on average more than 50% were from fragments which contained inlined functions across all optimization levels surveyed. This matches our expectation that most opportunity lies at function callsites. We also observed that on average less than 20% of the moves in inlined code can actually be removed.

However, these results are for the test input set and may not be representative of actual program composition. Most fragments do not contain inlined function calls. In addition, our method of determining whether a function is inlined within a fragment may be flawed. Further exploration with more representative input sets would be more revealing.

1 Introduction

Register assignment is the phase of register allocation that decides what values will be put into which registers. Because optimal register assignment has been shown to be an NP-complete problem, heuristics are used. Depending on the compiler used, register assignment can be done at different phases of compilation, potentially more than once. This has an effect on the ability of the compiler to perform instruction scheduling. Register assignment at any phase can introduce false dependences between instructions, and therefore affects instruction scheduling.

Since register assignment is usually done at compile time, there is no knowledge of actual execution patterns (without profiling). During actual execution, there is more information about frequently executed paths. With this in mind, it may be feasible to reduce the number of register moves within and/or between fragments. If the fragment is used often, and the time needed to perform this optimization is reasonably small, a significant speedup in execution time should be observed.

The goal of this project is to implement a linear-time heuristic for dynamically reassigning registers at the fragment level and to measure the opportunity available, the overhead incurred and any improvement in execution time that might be observed. In order to perform our experiments, we will modify the Strata [5] infrastructure to include a dynamic register reassignment optimization option for the sparc-fast target. In addition, we hope to identify any characteristics that may make a program more suitable for this type of optimization.

2 Background and Related Work

Register allocation and register assignment are optimizations performed by most optimizing compilers. These optimizations seek to minimize traffic between the CPU registers and the remainder of the memory hierarchy [2].

However, since these optimizations are performed on compilation units, and often separate compilation is supported, the optimizations have to be applied conservatively due to lack of sufficient information. In particular, compilers must be conservative at callsites. If a software dynamic infrastructure such as Strata is in place, it is possible that more usage information is available. In the case of Strata, calls are partially inlined when a fragment is formed, so there is a potential for register moves to be eliminated.

There are many types of register allocation and assignment algorithms which generally fall into two categories: graph coloring [1, 2] and linear scan [6, 4]. Both categories of algorithms make use of liveness information but differ in speed and complexity.

Graph coloring allocators make use of liveness information represented as an interference graph. Allocation is performed by heuristically k -coloring this graph. These types of allocators are known to yield very good results. However, this approach's space and time complexity is dominated by the space and time required to construct the interference graph (potentially quadratic), which can be quite expensive.

Linear scan allocators view liveness as a *lifetime interval* [6, 3]. A lifetime interval [3] is a segment of an instruction stream $[m, n]$ where a variable v is first live at instruction m and is last live at instruction n . A linear scan allocator makes assignments with potentially only a one pass over the lifetime interval list. Thus, a linear scan allocator has the advantage of being relatively simple and fast compared to graph coloring allocators.

Since this project involves dynamic register reassignment at the fragment level, the speed of evaluation and reassignment is critical. Therefore, for this project we consider an algorithm similar to the linear scan algorithm used by Poletto and Sarkar [4] in which assignment is performed through a single linear-time scan of liveness ranges. We chose not to implement the second-chance binpacking algorithm suggested by Traub *et al.* [6]. Even though the paper suggests that second-chance binpacking produces better assignment, it is more complex than the one suggested by Poletto *et al.*. This could potentially be evaluated in the future. We expect that the opportunity for eliminating moves will be at callsites.

3 Methodology

3.1 Framework

For this project, we are using the Strata software dynamic translation infrastructure [5] ported to the sparc-fast target. Our implementation is limited in that our optimization algorithm is fairly crude, but simple. In addition, our optimization code currently does not consider floating point code, which we originally thought would not affect our algorithm, since we were testing on SPEC's integer benchmarks.¹

3.1.1 Register Reassignment Algorithm

The algorithm for our simple optimization can be summarized in Figure 1. Basically, for each fragment we search for a move instruction. For safety, a move instruction is not considered if it is a delay slot instruction. If we find an eligible move instruction, we search through the fragment to see if we can replace uses of the move's destination register with a use of the move's source register.

If any unsafe condition is encountered, we stop propagating the copy instruction and note whether this stop condition will allow us to eliminate the move instruction. Figure 2 describes and explains the stop conditions. The last two entries of the table are not exactly the same. From our liveness calculation, we may know that the destination register is not live before we see it set. We were extremely conservative, but possibly incomplete in determining when we could propagate a move.

If it is possible to eliminate the move instruction we rewrite it as a *nop* for simplicity.² To handle the fact that there might be more than one move in the fragment, if we managed to eliminate a move, we recursively invoke *register_reassign()* to look for more copy instructions.³

¹This hole in our coverage may have affected our ability to get SPEC running correctly for *twolf*, *vpr*, and *perl*. We include results for *twolf* because the output compared to the reference was extremely close.

²We chose to change the copy instruction to a NOP to avoid having to potentially rewrite other pieces of the fragment or move code around in the fragment cache. Optimization is attempted at fragment build time, and on any instance where the fragment is patched. Otherwise, no further attempts are made to optimize.

³This could be done more efficiently without recursion, but we ran out of time.

For each fragment:

```
register_reassign()
{
    find_move_instruction();
    if(eligible_move_found)
    {
        propagate_copy_instruction();
    }
    if(possible_to_eliminate_move)
    {
        rewrite_move_to_NOP();
        register_reassign();
    }
}
```

Figure 1: Steps for the dynamic register reassignment algorithm.

Stop Condition	Eliminate Move?	Reason
Instruction is a branch	No	Not sure how to handle possible looping behavior
Instruction is target of a branch	No	Not sure of live out of branches which target this instruction
Source of move inst is written	No	End of the source register's live range
Unknown instruction	No	We don't know what the instruction is, better be safe
Destination register is written	Yes	End of the destination register's live range
Destination register is not live	Yes	End of the destination register's live range

Figure 2: Stopping conditions for copy propagate algorithm.

```

calculate_fragment_liveness()
{
    while(change)
    {
        cur_liveness=all_maybes;

        /* traverse frag backwards */
        while(PC >= frag->fPC)
        {
            if(is_branch)
            {
                if(target_outside_fragment)
                {
                    lookup_fragment();
                    if(fragment_found)
                        cur_liveness=union(cur_liveness, fragment->live_in);
                    else
                        cur_liveness=union(cur_liveness,all_maybes);
                }
            }
            else
            {
                mark_branch_target();
                cur_liveness=union(cur_liveness, target.live_in);
            }
            adjust_register_window_as_necessary();
            if((register[i].SET==YES)|| (register[i].SET==MAYBE))
                cur_liveness.register[i]=NO;
            if((register[i].USED==YES)|| (register[i].USED==MAYBE))
                cur_liveness.register[i]=YES;
            save_cur_liveness();
            PC-=4;
        }
        if(cur_liveness!=frag->live_in)
            change=TRUE;
        frag->live_in=cur_liveness;
    }
    register_reassign();
    deallocate_liveness_arena();
}

```

Figure 3: Register liveness calculation algorithm.

3.1.2 Register Liveness Analysis

In order to implement our optimization, a method for calculating liveness at each instruction and for each fragment was needed. To accomplish this, functionality for disassembling an instruction and returning the list of registers used and set was added to the sparc-fast target. This information was then used to determine liveness at any given instruction. Figure 3 describes the steps in determining register liveness.

During the course of a fragment’s life, it will carry its liveness information with it.⁴ Liveness is first calculated when a fragment is built, and if the fragment is updated, liveness information is also adjusted based on any new branch target information. Liveness at the instruction-level is maintained during liveness calculation in a special arena. This arena is deallocated upon exiting the liveness calculation function. Only a fragment’s *live in* information persists during the lifetime of the fragment.

The basic algorithm for determining liveness for a fragment initially assumes that all registers are potentially live, designated by a value of **MAYBE**. To be more precise, the liveness calculation assumes that at the bottom of the fragment, its notion of liveness for the register set is its *live out* information. For this project we did not manage to calculate *live out* for fragments, so these remain their initial value of all **MAYBE**.

As the algorithm traverses the fragment from bottom to top, it gathers register usage information for each instruction. If a branch is encountered, and the branch target is outside of the fragment, a lookup is performed to search for the target. If the target is a fragment and is found its *live in* values are unioned with the current liveness calculation. If the target cannot be found, a conservative guess of all **MAYBE**’s is unioned with the current liveness calculation. If the instruction encountered is a **save** or a **restore**, the liveness information is rotated as appropriate to account for the register window shift. If the instruction is not a branch, its register uses and sets are determined. If a register is marked as either definitely or possibly set, it is marked as *not live*. If a register is marked as either definitely or possibly used, it is marked *live*. All sets are handled before any uses. When the top of the fragment is reached if there has been a change in the liveness information, the fragment is traversed until liveness information stabilizes. This accounts for branches with targets within the fragment that may loop.

After the above algorithm was implemented it was determined that *live out* was too difficult to calculate. Liveness information at any given instruction is sufficient to determine the stopping condition for an optimization which will reassign registers because it gives us information about the register’s live range.

4 Experimental Methodology and Analysis

4.1 Experimental Setup

All experiments were run on relatively low-load SunBlade 100’s: tharsis, elysium, and uranius. Eight SPEC2000 benchmarks were used: twolf, gzip, gcc, vortex, mcf, parser, gap, and bzip2. The test input set was used.⁵

A comparison study was done by collecting some baseline numbers. We collected SPEC numbers for SPEC alone, SPEC instrumented with Strata without register reassignment, and SPEC instrumented with Strata using register reassignment.⁶ Data was collected for the -xO0, -xO2, and -xO5 optimization levels for each of these cases. The rationale for these optimization level choices is that -xO0 and -xO5 represent the extremes of the optimization scale, and bound the limits of opportunity for dynamic register reassignment. According to the online manual page -xO2 is cc’s default level of optimization if no optimization level is chosen, which might be representative of the average optimization level of compiled code.

In addition, we instrumented Strata to gather these additional statistics:

- Number of times a fragment’s liveness is calculated
- Number of register reassignment attempts
- Number of mov instructions changed to nop
- Number of times a copy propagation is possible but move elimination is not possible

⁴A fragment’s liveness information is considered to be its *live in* values at the start of the fragment. This information is part of the fragment, and resides in the fragment cache.

⁵Due to time pressure, we were forced to run on the test inputs rather than the training or reference inputs.

⁶Since the test input set was used, no SPEC numbers, rather wall clock time was calculated from the output files.

The last three statistics were gathered in two categories by fragment type: fragment with inlined function call and other fragment type.⁷ This allows us to compare overall register reassignment opportunity versus opportunity available for fragments which contain inlined function calls.

We would have liked to determine how often a fragment is executed, but this doesn't seem feasible as fragments jump to other fragments in the fragment cache without Strata directly monitoring them. Instead, we use the number of fragments created as an approximate frame of reference.

4.2 Analysis of Results

Since we used the test input set for SPEC2000⁸, a comparison of running times is unfair due to the fact that most of the execution time is spent in building fragments with little opportunity to use them. Therefore, we have not included an execution time comparison in our report.⁹

Our experiments show that there is not much opportunity for move elimination. Overall, level of optimization of the binary didn't seem to matter much. eliminated were in fragments which contained inlined function code.

Figure 9 shows that less than 20% of register reassign attempts were made on fragments which contained inlined function code. Figure 6 shows that more than 50% of moves that were eliminated were from fragments which contain inlined function code. This matches our intuition that the most likely place for eliminating a move is at a callsite. Figure 7 shows that moves that cannot be eliminated are fairly evenly divided between fragments with and without inlined function calls. Figure 8 shows the percentages of copies that were propagated by whether the copy was allowed to be removed. We see that there is more opportunity for copy propagation than there is for move elimination. Figure 11 shows that less than 20% of moves in fragments with inlined function calls can be removed. Figure 10 shows that the opportunity is even less for fragments without inlined function calls.

One thing that we noticed in debugging our code is that fragments that contain code to save some registers so that Strata can use them often contain moves that can be eliminated. We also noticed that fragments which contain hardware *umul* instructions also contain moves which can be eliminated.

In terms of time overhead, our algorithm is linear-time. It attempts to execute at fragment build and patch time, and examines a fragment in linear time in the size of the fragment to determine liveness. Another pass traverses the fragment potentially the total number of moves in the fragment to determine the possibility of a move elimination. The results show that for the test inputs there is not much opportunity for safe move elimination given our criteria for safeness. Therefore, the overhead of implementing this optimization may not be worthwhile.

When comparing Strata with and without the dynamic register reassigner, we find that our code does not increase fragment cache flushes. However, we have allocated a potentially large arena to contain liveness information during our liveness calculation pass. The data in this arena is deallocated at the end of the calculation pass.

5 Conclusions and Future Work

5.1 Conclusions

For this project, we have implemented a dynamic linear-scan register reassigner. In the process, we have also added a rudimentary method for register liveness calculation. The opportunity for optimization, the overhead of our implementation, and comparison runtime cost have been evaluated.

We find that the opportunity for eliminating moves seems small given our extremely conservative algorithm. A better, smarter algorithm could be developed. In addition, our experiments were run on unrepresentative data sets, which may have affected our findings.

5.2 Future Work

There are several ideas for future exploration of this topic:

- We rewrite register copies that could be eliminated as *nops*. These register copies can be actually eliminated to leave more room in the fragment cache for other code.
- The register reassignment algorithm implemented in this paper is extremely conservative, choosing to bail out of any reassignment attempt whenever a potentially confusing condition occurs (discussed in Section ??).

⁷Since Strata does not have an inlined function call fragment type, we approximated this by setting a flag in the fragment if (**TI.xlate_call*) function was called during the building or patching of the fragment.

⁸*twolf* did not run completely correctly, but was included in the data points because the difference in the output from the reference was very small. In addition, some data points are missing due to some runspec difficulties, and time constraints.

⁹Data is included in the Excel file containing our graphs.

A more aggressive algorithm could be implemented, possibly including a better fragment register liveness calculation algorithm.

- Our algorithm only considers *or* instructions of a register with *%g0* as register moves. Obviously, there are more ways to represent a move between registers, such as *adds*. These could also be considered.
- Although we attempted to keep the division between target-dependent and target-independent code clear, the code we implemented does not have a clear separation. The code could be cleaned up to make the separation clear.
- The code we wrote isn't efficient, with many functions performing duplicate work and other functions performing little work other than calling other functions. It could be modified to be more efficient and streamlined.
- In addition, the code we implemented did not do any decoding or handling of floating point code. We think that this is a potential reason for why we could not get *twolf*, *perl*, and *vpr* working correctly. Disassembler functions for the floating point instructions as well as any other instructions missing from the distributed Strata disassembler could be implemented, and these instructions could also be considered by the reassignment algorithm.
- We ran our experiments on the test inputs for the 7 SPEC2000 benchmarks. Statistics could be gathered when running the same experiments on the training and reference inputs and for more benchmarks.

References

- [1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*, chapter 9, pages 514–545. Addison-Wesley Publishing Company, 1988.
- [2] S.S. Muchnick. *Advanced Compiler Design Implementation*, chapter 16, pages 481–528. Morgan Kaufmann Publishers, 1997.
- [3] Massimiliano Poletto, Dawson R. Engler, and M. Frans Kaashoek. *tcc: a system for fast, flexible, and high-level dynamic code generation*. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Design and Implementation (PLDI '97)*, pages 109–121, Las Vegas, Nevada, June 1997.
- [4] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, 1999.
- [5] K. Scott and J.W. Davidson. Strata: A software dynamic translation infrastructure. Technical report, University of Virginia, 2001.
- [6] Omri Traub, Glenn H. Holloway, and Michael D. Smith. Quality and speed in linear-scan register allocation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 142–151, 1998.

Appendix

Our additions to the existing Strata infrastructure are detailed in Table 1. In addition, we have added two new files to the Strata framework to handle liveness analysis and register reassignment. `liveness.c` is added to the `/src` directory, as it is target-independent and `dis_live.c` is added to `/src/posix/sparc-fast` as the code it contains is target-dependent. The main functions contained in `liveness.c` are described in Table 2. Much of the code in `dis_live.c` is similar to the code in `dis.c` with minor modifications to collect and return register usage information rather than a string representation of the instruction. Most of the functions in `dis_live.c` are helper functions are not described here in the appendix, as they focus mainly on decoding instructions to gain particular bits of information.

Code Options

Our code allows a verbose trace of our algorithm. To view this trace, simply compile our distribution of Strata with `-DDEBUG` set in the appropriate makefiles. In addition, our distribution of the Strata code has the ability to include or exclude the use of our algorithm. To use our algorithm, compile Strata with `-DLIVE` set in the appropriate makefiles.

Filename	Addition	Description
strata.h	typedef register_list	List of registers for holding SET or USE information (an array of chars)
strata.h	#define YES, NO, MAYBE	Values used to determine USE and SET
strata.h	#define TRUE, FALSE	Defined to make sure they are what we expect
strata.h	strata_fragment augmented to include 2 register_lists	Holds live_in and live_out information (live_out currently not maintained)
strata.h	LIVENESS arena added	Arena to hold liveness data per fragment instruction
strata.h	NARENAS changed to 4	To support LIVENESS arena
alloc.c	first and *arena updated	Include initialization of LIVENESS
frag.c	void liveness_init(strata_fragment *frag);	Initialize a location in the LIVENESS arena
frag.c	void liveness_create(strata_fragment *frag);	Allocate memory in LIVENESS arena
frag.c	void liveness_destroy();	Release LIVENESS arena
frag.c	strata_fragment * strata_lookup_fragment_by_fPC(iaddr_t fPC)	Fragment lookup function by fragment PC rather than program PC
frag.c	strata_begin_fragment augmented	calls to liveness_create() and liveness_init()
targ-build.c	void print_insn_reg(iaddr_t PC,inst_t insn)	Debug function for printing collected register information
targ-build.c	void targ_disassemble_fragment_reg(strata_fragment *frag)	Disassemble function modified to return USE and SET information for registers
dis.c	decoder typedef moved to dis_live.h	Moved to allow sharing between dis.c and dis_live.c

Table 1: Changes to the existing Strata infrastructure.

Test Cases

Our algorithm works on small test cases as well as some of the SPEC2000 benchmarks. To run small test cases, first compile Strata with `-DDEBUG` and `-DLIVE` to view the verbose output. Then in the `src/test` directory do `make all` to make all the tests. Run the appropriate binary. To validate output only, Strata can be compile with `-DLIVE` only.

SPEC configuration files are located in the `specconfs` directory. These can be used with the `runspec` tool to validate program output and algorithm behavior.

Code Location

The tarball for our code will be located at: `/home/mc2zk/register_reassign.tar.gz`. A full strata distribution is included in the tarfile. Also included in the tarfile are `paper` and `specconfs` directories. `paper` contains all the code, graphs and data used to write this paper. `specconfs` contains the SPEC2000 configuration files we used to run our experiments.

Function Name	Description
void register_reassign(strata_fragment *frag, register_list *branch_target_info, char *is_branch_target)	Reassigns registers as possible
bool propagate_copy(strata_fragment *frag, iaddr_t startPC, unsigned rs, unsigned rd, register_list *branch_target_info, char *is_branch_target)	Propagates copies
iaddr_t find_move_inst(strata_fragment *frag, iaddr_t startPC, unsigned *rs, unsigned *rd)	Finds eligible move instruction, returns move address, source and destination register names
register_list register_list_union(register_list *a, register_list *b)	Unions two register lists
register_list * allocate_branch_info_list(strata_fragment *frag)	Allocates liveness information lists for a fragment
void calculate_fragment_liveness(strata_fragment *frag)	Traverses fragment from bottom to top to calculate liveness

Table 2: Functions contained in `liveness.c`.

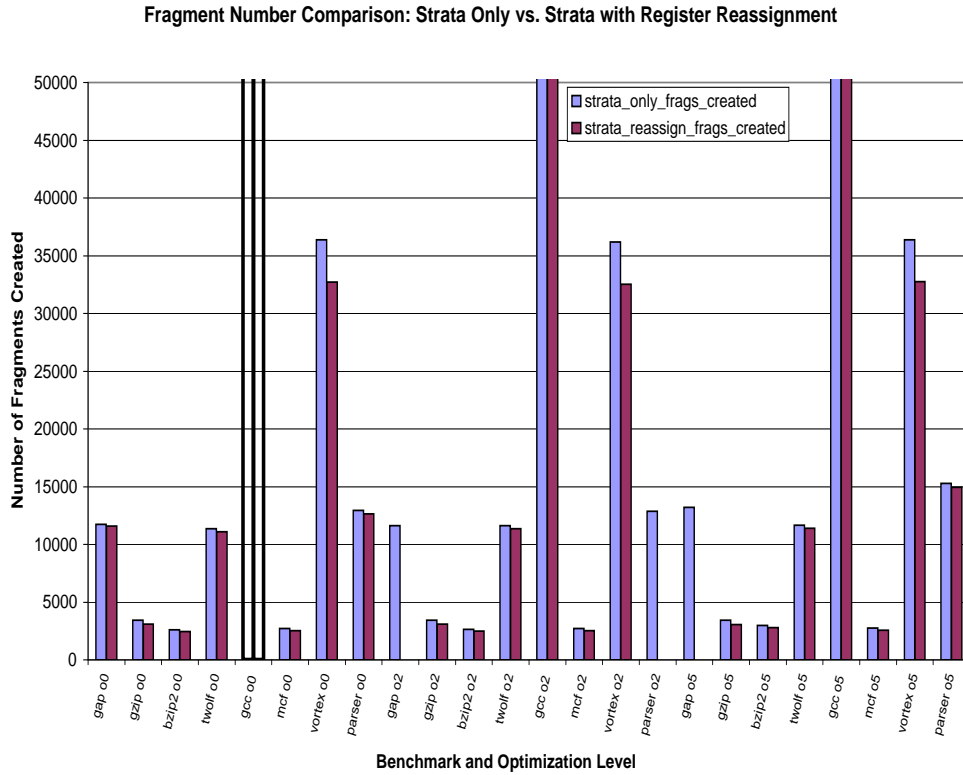


Figure 4: Comparison of number of fragments created: Strata only vs. Strata with register reassignment.

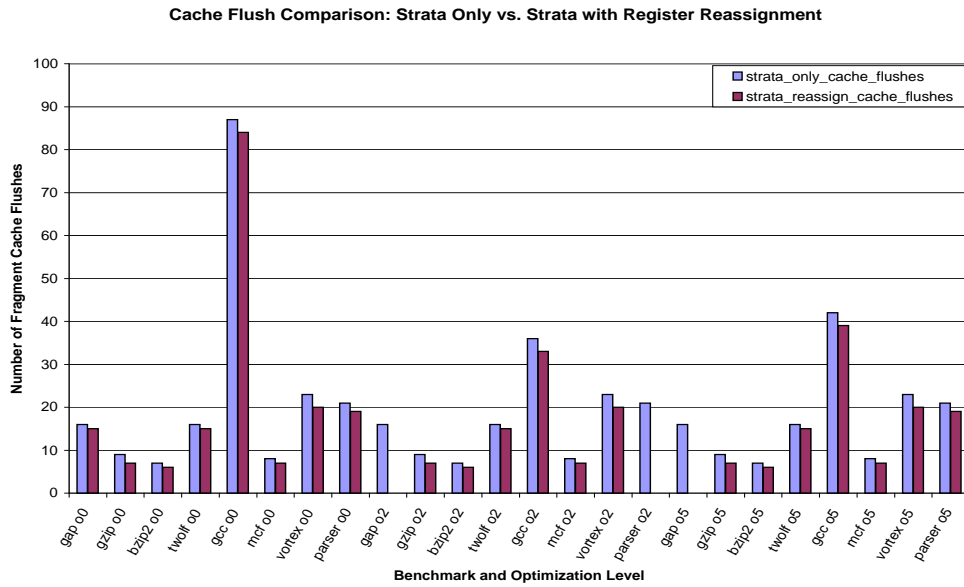


Figure 5: Fragment flush comparison: Strata only vs. Strata with register reassignment

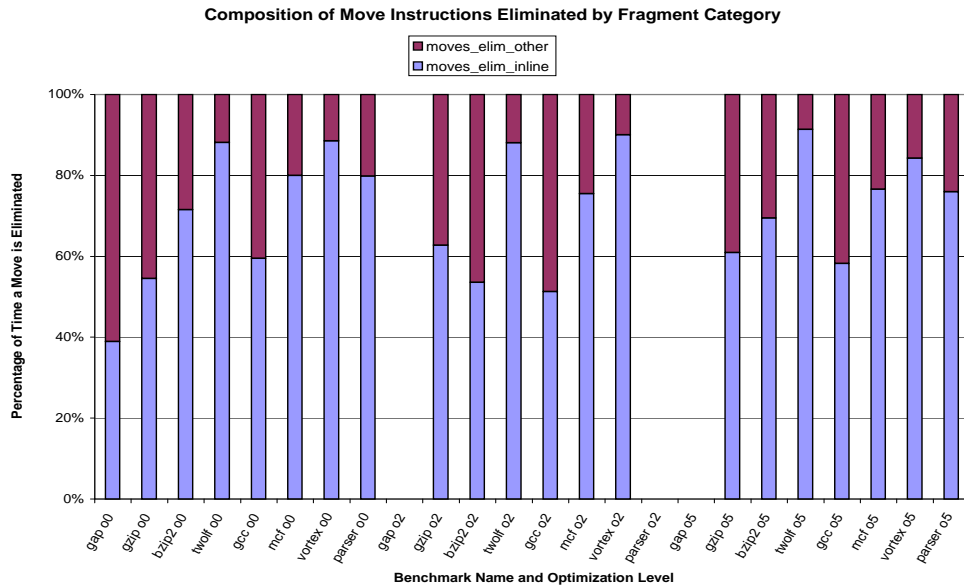


Figure 6: Percentage of move instructions eliminated from fragments with inlined function calls and other fragment types.

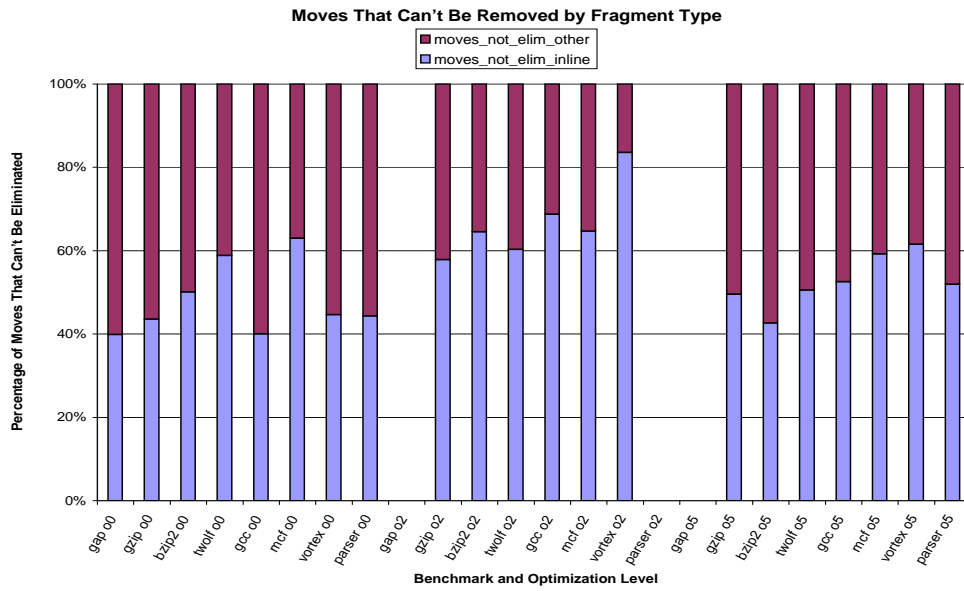


Figure 7: Percentage of move instructions that can not be eliminated by fragment category.

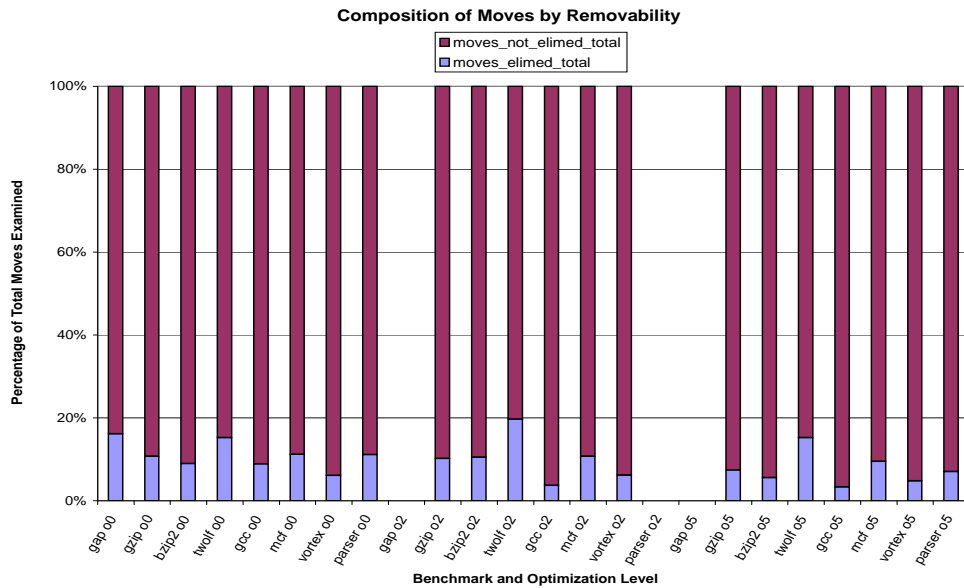


Figure 8: Composition of fragments with move instructions: with inlined function calls and without inlined function calls.

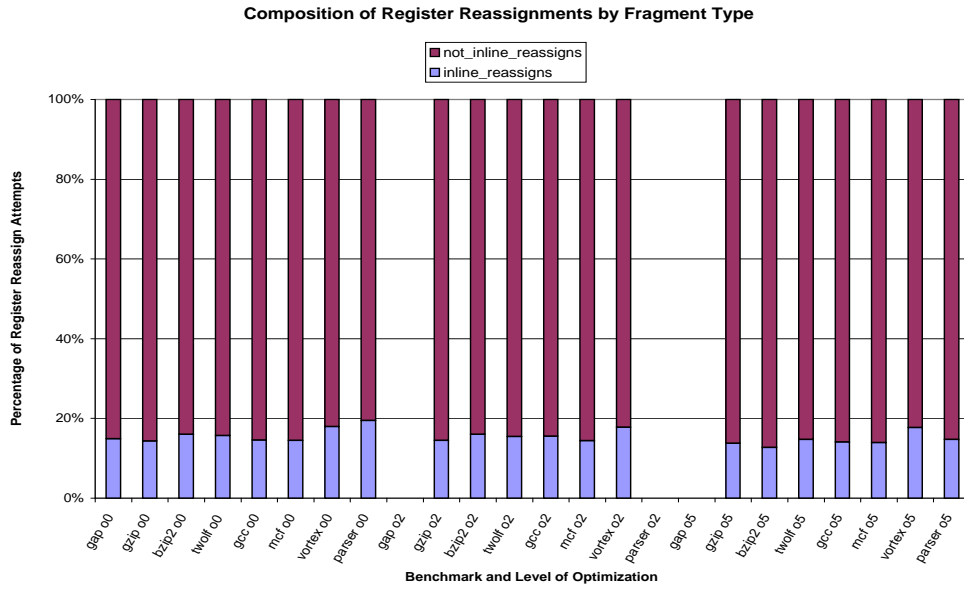


Figure 9: Percentage of register reassignment attempts by fragment category.

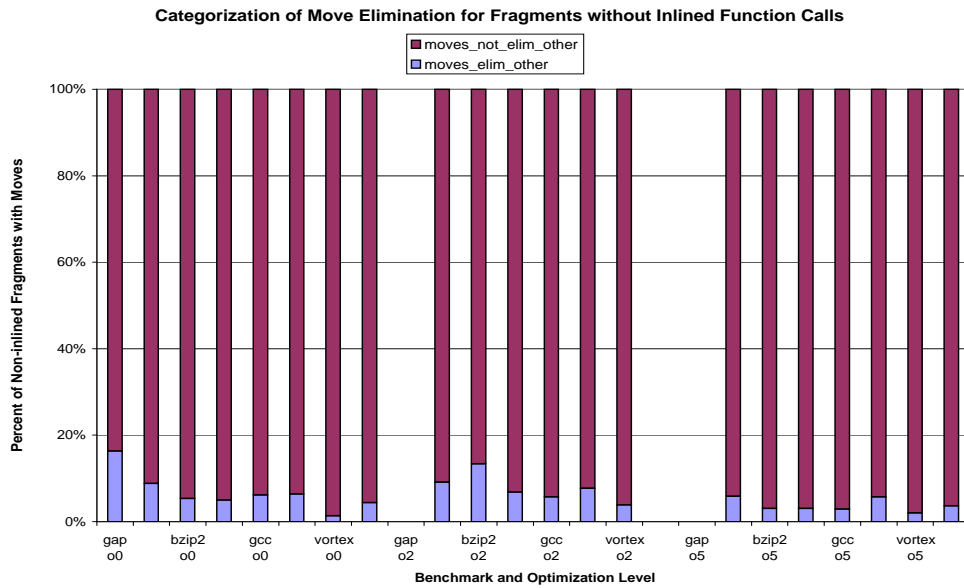


Figure 10: Percentage of moves in fragments without inlined function calls that can be removed.

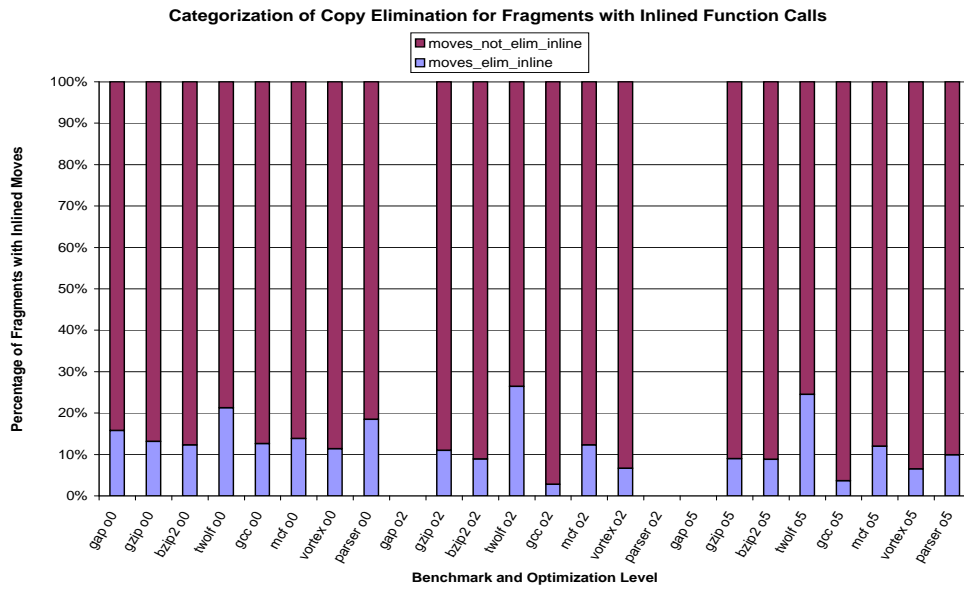


Figure 11: Percentage of moves in fragments with inlined function calls that can be removed.