Register Allocation Deconstructed

David Ryan Koes
Carnegie Mellon University
Pittsburgh, PA
dkoes@cs.cmu.edu

Seth Copen Goldstein
Carnegie Mellon University
Pittsburgh, PA
seth@cs.cmu.edu

Abstract

Register allocation is a fundamental part of any optimizing compiler. Effectively managing the limited register resources of the constrained architectures commonly found in embedded systems is essential in order to maximize code quality. In this paper we deconstruct the register allocation problem into distinct components: coalescing, spilling, move insertion, and assignment. Using an optimal register allocation framework, we empirically evaluate the importance of each of the components, the impact of component integration, and the effectiveness of existing heuristics. We evaluate code quality both in terms of code performance and code size and consider four distinct instruction set architectures: ARM, Thumb, x86, and x86-64. The results of our investigation reveal general principles for register allocation design.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Code generation, Compilers, Optimization

General Terms Algorithm, Design, Languages, Performance

Keywords Register Allocation

1. Introduction

Register allocation is a critical part of any optimizing compiler. The register allocator is responsible for finding a desirable assignment of program variables to hardware registers and memory locations. The quality of register allocation has a substantial impact upon final code quality, both when optimizing for performance and when optimizing for code size. When optimizing for performance, an effective register allocation minimizes memory traffic and decreases the sensitivity of a program to the processor-memory gap. When optimizing for code size, an effective register allocation minimizes the amount of overhead introduced by effectively using addressing modes and managing data movement instructions. Despite decades of study, register allocation remains a complex and challenging problem for which no completely satisfactory solutions exist.

In this paper we deconstruct the register allocation problem. We decompose the register allocation problem into distinct components: coalescing, spilling, move insertion, and assignment. The coalescing component of register allocation attempts to eliminate existing move instructions by allocating the operands of the move instruction to identical locations. The spilling component attempts to minimize the impact of accessing variables in memory. The move insertion component attempts to insert move instructions, splitting the live ranges of variables, with the goal of achieving a net improvement in code quality by improving the results of the other components. The assignment component assigns those program variables that aren't in memory to specific hardware registers. Existing register allocators take different approaches in solving these various components. For example, the central focus of a graph-coloring based register allocator is to solve the assignment problem, while the spilling and coalescing components are integrated as extensions to this assignment-focused model.

In this paper we use an optimal register allocation framework to study each component of register allocation. We evaluate both the individual impact of each component and the synergistic impact of fully integrating components upon code quality. We consider both code performance and code size metrics of code quality and target four distinct instruction set architectures: ARM, Thumb, x86, and x86-64.

What is the greatest benefit that can be realized by developing a fully integrated allocation algorithm? Conversely, what is the maximum penalty incurred if the register allocation problem is decomposed into (hopefully) easier to solve subproblems? What is the individual impact of each component of register allocation on code quality? How far from optimal are existing heuristics and where are the most fruitful areas for researchers to focus their attention? The goal of this study is to answer these questions and derive general principles for register allocation design.

The main contributions of this paper are:

- The first comprehensive empirical investigation of the importance, impact, and interaction of the various components of register allocation.
- A set of design principles extrapolated from this investigation useful in guiding the construction of future register allocators.

In the next section we further describe the register allocation problem and describe previous work. In Section 3 we describe the optimal register allocation framework we use. Section 4 contains the methodology we use to perform our study. In Section 5 we present and analyze the results of our investigation, and in Section 6 we extrapolate several general guidelines of register allocation design from the results of our study.

2. Background

Register allocation is an extensively studied problem. Graph-coloring based algorithms are the dominant technique for performing global register allocation. These approaches construct an interference graph and attempt to find a coloring of the graph where the colors correspond to registers [6, 9]. The basic graph-coloring algorithm has been extended in attempts to integrate additional components of register allocation, such as spill code generation [2, 3, 7], move insertion [10], and coalescing [7, 9, 12, 24]. However, at its core the graph-coloring approach is focused on solving the register assignment problem and treats the spilling problem as a secondary concern.

Linear scan allocators [25, 26, 29, 30] are an alternative to graph-coloring approaches. Linear scan allocators were originally developed for dynamic and just-in-time compilation. Initial linear scan allocators were extremely fast and scalable, but produced low quality code. Recently, the extended linear scan algorithm [26] has been shown to produce higher quality code than a graph-coloring allocator while maintaining the scalability advantages of linear scan algorithms.

Most register allocation algorithms attempt to integrate the various components (move insertion, coalescing, spill code generation, and register assignment) of register allocation into a single algorithm. However, several approaches separate spill code generation and register assignment [1, 22, 26]. In particular, Appel and George [1] use an integer linear programming (ILP) formulation to solve the spill code optimization problem and then use heuristics to find a valid assignment. Their framework views assignment as a coalescing problem. Parallel move instructions are inserted at every program point making the assignment problem trivial; the challenge becomes eliminating as many move instructions as possible by assigning the operands of the move to the same register.

It has been shown that inserting moves at every program point is unnecessary; it is sufficient that the program be in SSA form and the assignment problem becomes polynomial [5, 8, 15, 23]. Additional work has embraced the coalescing view of register assignment [4, 13, 16, 31, 32]. These approaches perform spill code generation as a separate phase, insert sufficient moves to render the assignment problem trivial (or expect the code to be in SSA form), and then attempt to remove as many move instructions as possible. Although these approaches primarily use coalescing algorithms, the general problem they are solving is the register assignment problem. In this paper we distinguish between the register assignment problem, which may generate move instructions, and the coalescing problem, which removes existing move instructions. We view coalescing based register assignment techniques as an integration of these two components.

Several register allocators that solve all or part of the register allocation problem optimally have been implemented. Optimal techniques, primarily based on ILP representations, have been presented that solve the register allocation problem in its entirety [11, 20], the spill code optimization problem [1], and the integrated assignment-coalescing problem [1, 13]. However, no study has used an optimal register allocation framework to thoroughly analyze the individual and collective impact of the various components of register allocation.

3. Optimal Register Allocation Framework

The heart of our investigation of register allocation is an optimal register allocation framework derived from [19]. This framework models the register allocation problem using multi-commodity network flow (MCNF) and exactly represents the spill code optimization, move insertion, and register assignment components of register allocation. We extend the model to exactly represent the coalescing component by adding side constraints. These side constraints, which do not fit directly into the network flow formulation, model the benefit of allocating the two operands of a move instruction to the same location. We formulate the resulting model of register allocation using an integer linear programming (ILP) model. The ILP is then solved using ILOG CPLEX 10.0 [17]. It is important to note that this optimal register allocation framework is not practical for production use; finding an optimal solution for a single allocation problem can take as long as several days. However, because the results are optimal, this framework is an ideal tool for investigating the essential features of register allocation.

We consider a register allocation to be optimal if, for a given code quality metric, the allocation has the best code quality that is achievable by assigning registers and inserting loads, stores, and moves into a given instruction stream. Our model is essentially optimal, but there are a few minor limitations. One such limitation is an implicit assumption that it is never beneficial for the same value to be in two different registers at the same program point. For architectures with uniform register sets, this assumption holds as long as copy propagation has been applied to the code. In addition, we do not consider the impact of increasing the size of the stack (to hold spilled values) on code quality. Finally, note that we consider instruction scheduling to be beyond the scope of the register allocator and do not reorder instructions.

Our model correctly models the persistence of a value in memory. That is, a value need only be stored to memory

Register Allocation Components			
Move Insertion	Coalescing	Spilling	Assignment
full: move instructions may be inserted at any program point	**integrated optimal:** the benefits of move coalescing are exactly represented as part of an ILP model of register allocation	**integrated optimal:** the costs of spill code generation are exactly represented as part of an ILP model of register allocation	**integrated optimal:** the costs of register assignment are exactly represented as part of an ILP model of register allocation
limited: move instructions may be inserted only at the entry and exit of basic blocks	**integrated optimal ignoring uncoalescable:** the benefits of coalescing only those move instructions that can be identified as coalescable prior to register allocation are incorporated into the ILP model	**separate optimal:** the spill code generation problem (reducing max liveness to meet register availability) is solved optimally as a standalone problem	**graph heuristic:** a graph-coloring based heuristic is used to assign registers to the results of spill code generation; move instructions may be inserted to improve colorability
none: no register-to-register move instructions are generated by the allocator	**separate optimal:** an ILP model is used to eliminate the maximum number of coalescable moves prior to register allocation	**separate heuristic:** the spill code generation problem is solved as a standalone problem using a heuristic algorithm	**linear scan heuristic:** a linear scan based heuristic is used to assign register to the results of spill code generation; move instructions may be inserted to improve colorability
	separate aggressive: a greedy heuristic aggressively eliminates coalescable moves prior to register allocation		
	none: no coalescing is performed		

Configurations

Table 1. The components of register allocation and the various configurations evaluated in this study.

once with no cost incurred by future evictions. Our model takes a very fine-grained view of spilling; this is not a "spill-everywhere" approach. In our full model we allow moves between registers at every program point. In order to simplify the model, we only permit load and store instructions to be inserted at basic block boundaries and before/after instructions that use/define the corresponding variable. This simplification does not effect the optimality of the result. Constant values may be rematerialized instead of being spilled to memory.

We consider two code quality metrics: code size and code performance. Optimizing for code size is straightforward as the immediate impact of any transformation can be exactly determined at compile time. The complex nature of modern architectures makes an exact metric for performance unattainable. Instead we model code performance using a weighted sum of loads, stores, and move instructions where instructions that are judged to execute more frequently have a proportionally higher weight.

We use our full model of register allocation to find an optimal allocation. This optimal allocation serves as a baseline in our investigation of the importance of the different components of register allocation. In order to perform our investigation, it is necessary to modify the model so that we can analyze each component separately as well as investigate the impact of integrating different components. We consider four distinct components of register allocation: move insertion, coalescing, spilling, and assignment.

Next, we describe how we modify the model to investigate the various configurations of these four components (Table 1).

3.1 Move Insertion

Inserting move instructions during register allocation is typically done to split the live range of a variable in order to avoid spilling the variable, to take advantage of some register preference, or to make the assignment problem easier. As an entirely separate phase, move insertion is pointless since inserting moves can only increase code size and decrease performance. Instead, the nature of move insertion necessitates that it be integrated with at least a register assignment phase. Therefore we do not attempt to analyze the separability of move insertion. Instead we consider the impact move insertion has on code quality by both disabling and limiting the ability of our full model to insert moves. With move insertion disabled in the full model, the optimal allocator never generates register to register moves. If a variable is defined or loaded into a register, it stays in that register until the variable is no longer live or the variable is evicted to memory. As a middle ground between full move insertion (where register to register moves can be inserted at any program point) and

no move insertion we also evaluate limiting move insertion to basic block boundaries. That is, the register allocator will only generate register to register moves at the entry and exits of basic blocks.

3.2 Optimal Coalescing

Move coalescing occurs when the register allocator can allocate the source and destination operand of an existing move instruction to the same location, resulting in the elimination of the move instruction. As a separate phase, the coalescing problem is to eliminate as many coalescable moves as possible. A move is coalescable if the source operand does not interfere with the destination operand. For instance, the source operand cannot be live after the instruction. When performed independently from register assignment, move coalescing merges the live ranges of the two operands of the move instruction into a single new variable.

There are three apparent disadvantages to performing coalescing as a separate pass prior to assignment. The first is that overly aggressive coalescing may make the register assignment problem harder. However, if the assigner effectively utilizes move insertion, this drawback can be overcome. The second disadvantage of decoupling coalescing from the rest of the allocator is the inability to remove uncoalescable moves. For example, in the context of the full register allocation problem, it may be possible to remove a move instruction where the source is live after the instruction if the source value is available in memory. In this case, future users of the source operand value will have to first load the value from memory. Finally, standalone coalescing cannot eliminate move instructions where one operand is a preallocated machine register; these move instructions must be removed by the register assignment phase.

In order to investigate the impact of coalescing on register allocation, we formulate the standalone coalescing problem as an ILP using a graph-coloring representation similar to [12] where we do not restrict the number of colors. This pass identifies the maximum number of coalescable move instructions and then removes these instructions by assigning the operands to identical virtual registers. The resulting instruction stream is passed to a version of our full model that is missing the side constraints that represent the benefits of coalescing. In addition, in order to evaluate the benefit of an integrated allocator being able to remove uncoalescable moves, we consider a version of the full model where coalescing side constraints are only added for those moves that are coalescable.

3.3 Optimal Spilling

Optimal spill code generation minimizes the impact of memory accesses generated by the register allocator on code quality. The goal of spill code generation when run as a separate standalone pass is to legally modify the input instruction stream so that at every program point sufficient registers are available to hold all allocable live variables. The spill code generator reduces the number of allocable variables at a program point by spilling (moving to memory) a variable over some range of program points. The result of spill code generation is provably assignable [16] as long as the assigner is capable of inserting move and swap instructions.

In order to represent the standalone optimal spilling problem we use a simplified version of our full model. In the MCNF based model of register allocation used in the full model, each variable is a flow through a network of nodes where each node represents memory or a specific register. In order to represent the spilling problem, we simplify this network so that instead of having a unit capacity node for each register, we have a single node for each register class with capacity equal to the class size. For example, if there are eight integer registers, these will constitute a single register class that is represented by a node with capacity eight. The resulting model is substantially smaller and less complex than the full model. It does not model register to register moves or coalescing constraints and has considerably fewer nodes and edges. We optimistically model register usage preferences. That is, if there is some benefit to allocating a variable to a specific register at a program point, then in the simplified model, that variable achieves the same benefit by being allocated to the corresponding register class.

3.4 Optimal Assignment

Given the results of spill code generation, optimal register assignment finds an assignment of registers to variables that maximizes code quality. The assigner may have to insert move or swap instructions in order to generate a legal assignment. Since in our optimal framework we allow for only the generation of move, load, and store instructions, we implement swaps using an additional memory location. The register assigner maximizes code quality not only by minimizing the impact of inserted move and swap instructions, but also by exploiting any register preferences. For example, if a variable is moved into a hardware register, allocating that variable to that hardware register will eliminate the corresponding move instruction.

We implement optimal assignment as a standalone problem by constraining the full model to observe the results of spill code generation. For example, if spill code generation creates a load of a variable at a particular program point, then in the constrained full model we force that variable to be loaded by constraining the corresponding variables in the ILP appropriately.

3.5 Heuristics

In addition to developing optimal algorithms for the components of register allocation we have implemented several heuristic algorithms within the parameters of our framework. We have implemented an aggressive coalescer that simply considers all coalescable move instructions in program order and greedily coalesces. Our heuristic spiller is an implementation of the heuristic solver of [19] applied to the simplified,

spill code optimization, version of the MCNF based model. We also use this heuristic solver to solve the full model (ignoring side constraints).

We implement two assignment heuristics to represent the two most prevalent approaches: a linear scan based algorithm similar to [26] and an algorithm that uses graph simplification [9] to find an initial partial assignment that is finalized with a move insertion pass. Note that the assignment heuristics assume that spill code generation has been performed (the maximum number of live variables does not exceed the number of available registers at any point). Therefore, the assignment heuristics do not insert any spill code to reduce register pressure. However, move or swap instructions may have to be inserted at basic block boundaries in order to find a valid assignment. In our implementation, swap instructions are implemented using a temporary memory location. Both heuristics attempt to exploit register preferences by assigning a variable its preferred register if it is available.

4. Methodology

We have implemented our optimal register allocation framework within the LLVM 2.4 [21] compiler infrastructure. We target four distinct instruction set architectures:

- **x86** The venerable Intel x86 32-bit instruction set [18] has variable length instructions, supports direct access to memory in most instructions, and has a limited register set (8 integer, 8 floating point). We consider this ISA to be representative of CISC architectures with limited register resources.
- **x86-64** The Intel x86 64-bit instruction set [18] also has variable length instructions and support for memory operands, but has an extended register set (16 integer, 16 floating point). We consider this ISA to be representative of CISC architectures with sufficient register resources.
- **ARM** The ARM instruction set [27] has four-byte RISC-like instructions and supports 16 integer registers. We target an ARMv6 core with software floating point. We consider this ISA to be representative of RISC architectures with sufficient register resources.
- **Thumb** The Thumb instruction set [27] is an alternative instruction set for ARM processors optimized for code size. It has two-byte RISC-like instructions and can only efficiently access 8 integer registers. For our investigation we restrict the allocator to only allocate to these 8 efficiently accessed registers to make the Thumb target representative of architectures with limited register sets. We target an ARMv6 core and do not generate Thumb-2 instructions. We consider this ISA to be representative of RISC architectures with limited register resources.

We believe that these four architectures, in addition to constituting a substantial portion of market share in the desktop, server, and embedded spaces, are also generally representative of most existing architectures. Our observations on the impact of register allocation on code size and performance can therefore be readily extrapolated to similar architectures.

4.1 Code Size Evaluation

The immediate impact of register allocation on code size is straightforward to measure since the compiler can exactly calculate the size of a function. Since we are concerned with the direct impact register allocation has on code size and wish to avoid any potential noise from downstream optimizations, all reported code size results reference the code size immediately after performing register allocation. We only report the size of executable code; constants and other data are ignored.

Since code size is principally of interest to the embedded community, we use MiBench [14], a commercially representative embedded benchmark suite in our evaluations. We omit five of the 20 benchmarks (mad, tiff, sphinx, ghostscript, and ispell) due to dependencies on libraries not available on our non-native ARM and Thumb targets.

In order to generate results in a reasonable timeframe, we impose a time limit of 10 minutes per a function upon the CPLEX solver. In comparing allocators, we consider only those functions for which an optimal solution is found for *all* allocator configurations and architectures under consideration. The imposition of a time limit introduces a self-selecting bias to our results. However, we find that this time limit is sufficient to select more than 70% of the functions in the considered benchmarks. Furthermore, we observed no significant qualitative change in results as we increased the time limit.

In reporting code size results we compare the total code size over all functions relative to the fully optimal result. That is, a result ratio of one is the best possible result and larger ratios represent a code size increase.

4.2 Performance Evaluation

Code performance is the predominant code quality metric in the desktop and server spaces. We evaluate code performance on the highly relevant x86 and x86-64 targets. We run our performance experiments on an Intel Core 2 Quad (Q6600) processor running at 2.4GHz with 4GB of main memory. We evaluate performance using a subset of the SPEC2006 [28] benchmark suite. Unfortunately, it isn't feasible to solve every function of every benchmark to optimality. In order to make the evaluation practical we consider only those C/C++ benchmarks where, based on profile data, 85% of the execution time is spent in 10 or fewer functions. When compiling these benchmarks we find an optimal register allocation only for these key functions. We further limit the number of benchmarks by only considering those benchmarks where an optimal solution could be obtained for every key function for all configurations of the allocator within a time limit of 1000 minutes. These selection criteria result

in a subset of six benchmarks for x86 (429.mcf, 433.milc, 462.libquantum, 470.lbm, 473.astar, and 482.sphinx) and four benchmarks for x86-64 (433.milc, 462.libquantum, 473.astar, and 482.sphinx). Note that since different benchmark sets are used for the two architectures, they are not directly comparable.

When optimizing for performance, our register allocation model requires some estimate of the execution frequencies of every basic block. We extract exact execution frequencies from a profile run of the train data set and use these exact frequencies within the model. Since we are interested in finding an optimal allocation, we eliminate any noise from differences in data sets by also using the train data set to compute our performance numbers.

In reporting code performance results, unless stated otherwise, we compare a geometric mean across the selected benchmarks of the execution time *increase* relative to the fully optimal model. Since the fully optimal result is only optimal with respect to a weighted sum of executed loads, stores, and moves, the result is not necessarily optimal with respect to all aspects of processor performance. Furthermore, post-allocation optimizations may add noise. As a result, it is possible to observe a performance ratio less than one, indicating a performance improvement relative to the optimal allocation. In addition to reporting execution time, we use performance counters to collect dynamic load, store, and instruction counts. These counts are representative of the simple code performance metric used by our optimal model. Indeed, there are cases where the optimal solution according to our performance metric does not yield the fastest code, implying there are important aspects of processor performance that are not captured by our metric.

5. Analysis

In this section we use our optimal allocation framework to empirically evaluate the importance and impact of the various components of register allocation. We investigate the impact of solving these components separately, but optimally, examine the individual contribution of each component to the overall code quality, and evaluate the performance of heuristics relative to an optimal allocator.

5.1 Move Insertion

Code size and performance results for two different move insertion strategies are shown in Figure 1. In both cases, any pre-existing move instructions are aggressively coalesced prior to register allocation. That is, the allocator cannot simply split a live range by choosing not to coalesce; a move must be inserted. Surprisingly, disabling move insertion altogether, which should necessitate the introduction of additional spill code, has only a marginal impact on code quality. Code size grows by less than 0.25% on all targets. The limited move insertion strategy, which only allows move instructions to be inserted by the allocator at basic block boundaries, performs even better demonstrating miniscule or nonexistent increases in code size.

Both strategies result in essentially equal performance compared to the optimal allocator. Disabling move insertion does result in more memory operations, but these extra memory operations do not have any meaningful effect on performance. In contrast, the limited move insertion strategy is virtually indistinguishable from the optimal allocator.

Several register allocators utilize move insertion in order to generate a better allocation [1, 19, 20] or to simplify the assignment problem [13, 16, 31, 32]. The results of our empirical study strongly suggest that supporting move insertion has a surprisingly minimal impact on final code quality and that supporting full move insertion, where move instructions can be inserted at every program point, is not necessary to achieve high quality code.

5.2 Coalescing

Code size and performance results for four different coalescing strategies are shown in Figure 2. After performing coalescing using one of the shown strategies, both spill code optimization and register assignment are solved optimally as a single integrated pass. As expected, not performing any coalescing results in significant increases in code size as well as significantly increasing the number of instructions executed and degrading performance.

The three remaining coalescing strategies have remarkably little impact on code quality. Code size grows by less than 0.15% on all targets and the difference in performance is negligible. There is a small code size advantage of optimal coalescing over aggressive coalescing, but in practice the greedy heuristic aggressive coalescer usually eliminates the same number of move instructions as the optimal coalescer. When the fully integrated optimal allocator is configured to ignore any potential benefit from eliminating uncoalescable move instructions (that may become coalescable with the introduction of spill code), a small code size increase is observed. In all cases, the integrated coalescer slightly improves upon the optimal allocator implying that the decision of which move instructions are coalesced, not just the number of coalesced move instructions, can influence the final code quality.

These results suggest that, when viewed as a move elimination problem distinct from register assignment, coalescing can be effectively implemented as a separate pass using a simple greedy heuristic. However, when code size is of paramount importance and the target is register limited, improving coalescing and integrating coalescing into the spill code optimizer may yield a small improvement.

5.3 Spilling

The impact on code size and performance of both optimal and heuristic spill code optimization performed as a separate pass is shown in Figure 3. After executing the spill code optimization pass, coalescing and register assignment are per-

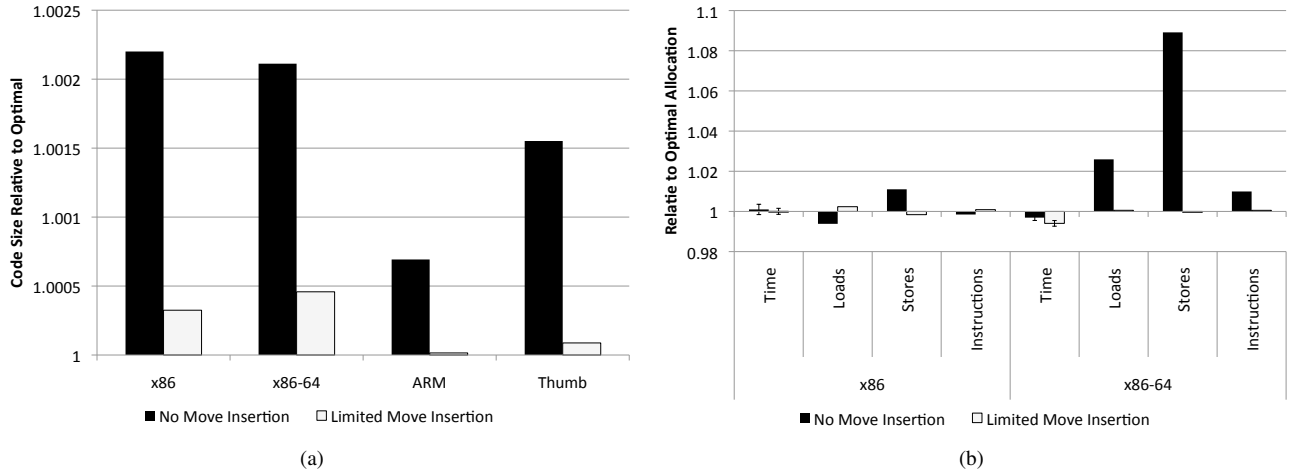

Figure 1. The impact of move insertion on code quality when targeting code size (a) and code performance (b). Taller bars indicate larger/slower code. Error bars are shown for execution time measurements.

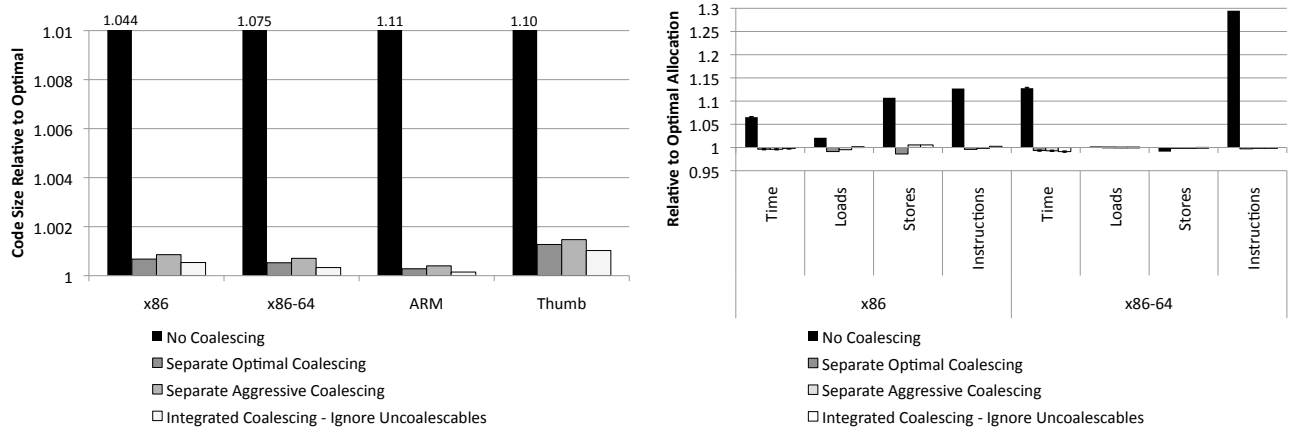

Figure 2. The impact of coalescing on code quality when targeting code size (a) and code performance (b). Taller bars indicate larger/slower code. Error bars are shown for execution time measurements.

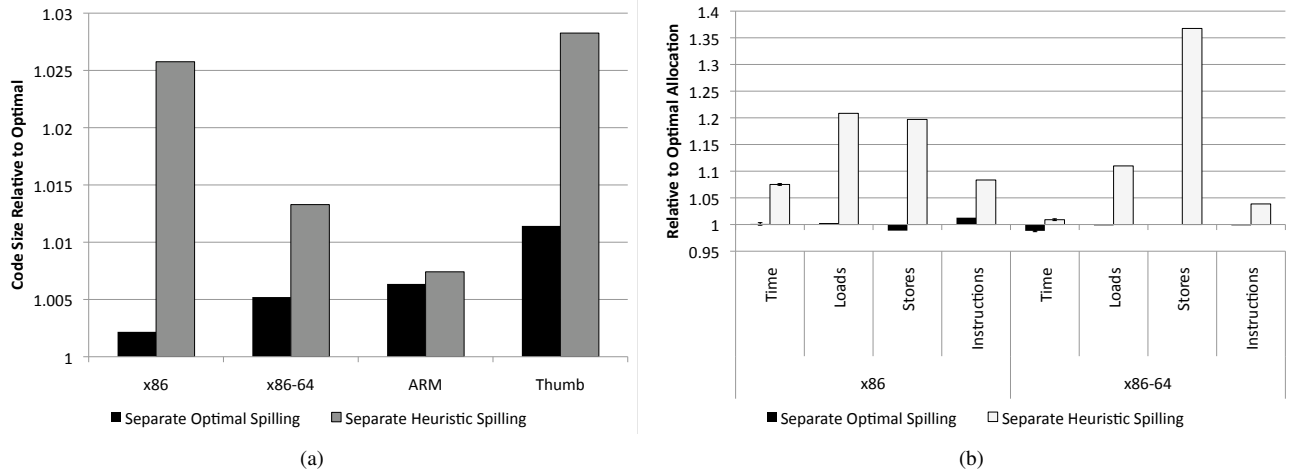

Figure 3. The impact of spill code optimization on code quality when targeting code size (a) and code performance (b). Taller bars indicate larger/slower code. Error bars are shown for execution time measurements.

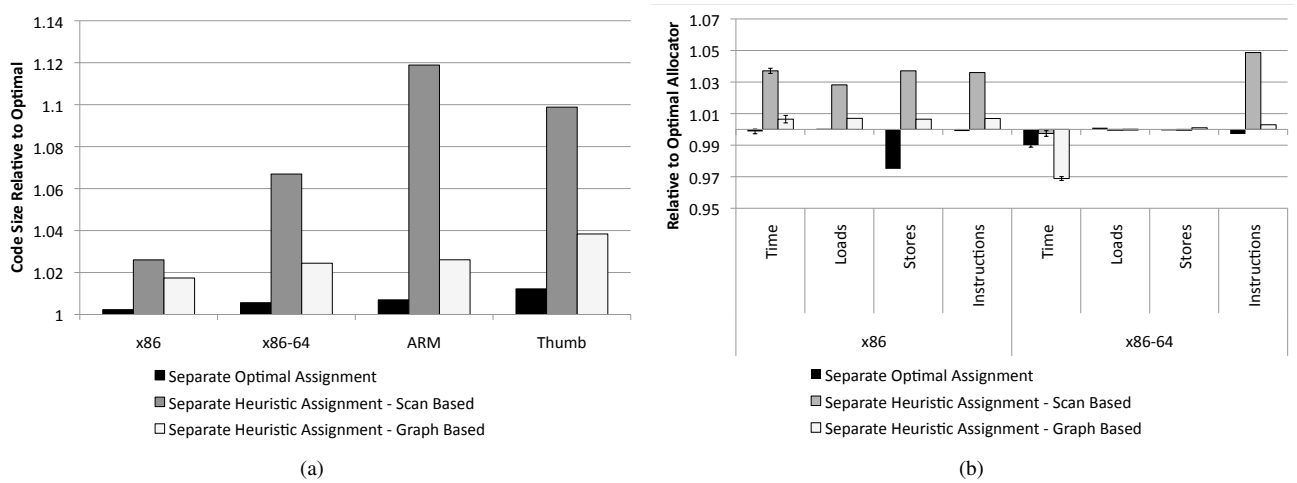

Figure 4. The impact of register assignment on code quality when targeting code size (a) and code performance (b). Taller bars indicate larger/slower code. Error bars are shown for execution time measurements.

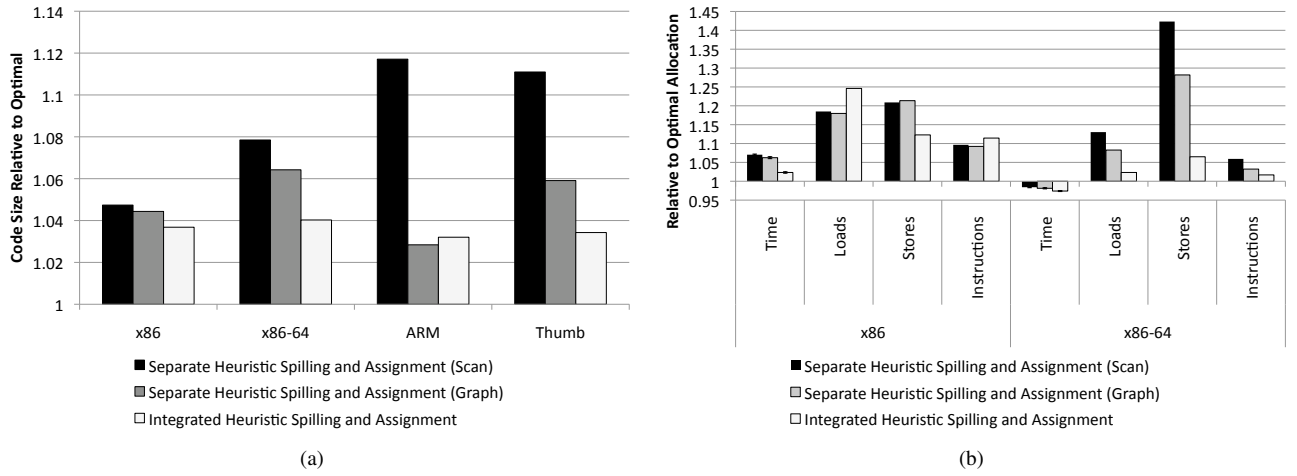

Figure 5. The effectiveness of heuristic allocators when targeting code size (a) and code performance (b). Taller bars indicate larger/slower code. Error bars are shown for execution time measurements.

formed as a single optimal pass. Particularly for the register limited architectures, there is a definite degradation of code quality when a heuristic spiller is used. The register limited x86 architecture exhibits a 20% increase in dynamic memory operations that results in a 7.5% increase in execution time. Interestingly, although the number of dynamic memory operations increases substantially on the x86-64 architecture, this increase does not correspond to a significant increase in execution time.

Performing spill code optimization separately, but optimally, has no discernible impact on performance; however, there are notable increases in code size, ranging from an increase of 0.22% on x86 to 1.1% on Thumb.

These results suggest that, when optimizing for performance, spill code optimization can be implemented as a separate pass without loss of code quality. However, when optimizing for code size, spill code optimization is not so readily decoupled from the rest of register allocation. When targeting architectures with sufficient register sets, a heuristic spill code optimizer can approach the quality of an optimal optimizer, but there remains room for improvement, particularly when targeting register-limited architectures.

5.4 Assignment

Code size and performance results for three different assignment algorithms are shown in Figure 4. Prior to assignment, aggressive coalescing and optimal spill code generation are performed. The optimal assigner does not explicitly optimize for coalescing, but does explicitly optimize for register preferences. Unsurprisingly, given the spilling results, there does not seem to be any penalty for performing assignment as a separate pass when optimizing for performance.

However, an increase in code size is observed. Both heuristic assigners generate poorer quality code than the optimal assigner, but the graph-based assigner generates substantially better code than the scan-based assigner.

The graph-based assigner was especially effective when optimizing for performance. On the x86 architecture, the graph assigner resulted in a small increase in the number of memory operations and instructions executed, while on the x86-64 architecture only a slight increase in the total number of instructions executed was observed. Memory operations are introduced by the assigner when a swap operation is needed. The greater number of registers on the x86-64 architecture apparently made such operations less necessary and only simple move instructions were generated.

The graph-based assigner first uses graph simplification to find a partial coloring of the interference graph. If any variables remain uncolored after this pass then these variables are split into colorable live ranges. If the first pass is successful at finding a color, no move or swap instructions are generated, otherwise move and swap instructions are only generated for the uncolored variables. In practice, the first pass often successfully colors the interference graph. For example, on x86, more than 75% of the compiled functions have their interference graph fully colored while more than 90% of the functions end up with one or fewer uncolored nodes. As a result, few move and swap instructions are generated. Since swap instructions are implemented using a temporary memory location in our framework, the graph-based assigner is especially effective when optimizing for performance.

The results of our empirical study suggest that there remains ample room for improving assignment algorithms and a poor register assignment can negatively impact performance (especially if a register swap requires a memory access). More advanced techniques such as graph recoloring [16] may narrow the gap between heuristic and optimal solutions. However, in order to achieve maximum code quality, particularly when optimizing for code size, some integration between the spill code optimizer and the assigner is necessary.

5.5 Heuristic Comparison

Code size and performance results for three different purely heuristic register allocators are shown in Figure 5. All three heuristics first perform aggressive coalescing since coalescing has been shown to by highly separable and aggressive coalescing is very effective while being compile-time efficient. We consider two heuristics that treat spill code optimization and assignment separately. They both use the same heuristic spill code optimizer, but use our two different assignment algorithms. The third heuristic is an integrated heuristic that attempts to solve the spill code optimization and assignment problems simultaneously. Unsurprisingly, when optimizing for code size the graph-based assigner outperforms the scan-based assigner and both are outperformed by the integrated heuristic. However, there remains a significant gap between the optimal allocation and the best heuristic solution.

In all cases the heuristics increase the number of executed memory operations and instructions. On the x86 architecture, increases of execution time are also observed. Interestingly, these increases do not translate into performance degradations on the x86-64 architecture.

6. Conclusions

In this paper we have presented the first of its kind, comprehensive empirical investigation of the importance, impact, and interaction of the various components of register allocation. Extrapolating from the results of our empirical study, we conclude with a set of design principles useful in guiding the construction of future register allocators and empirically justifying the design of current allocators:

- The ability to insert moves (that is, split live ranges or undo coalescing) has surprisingly little impact on code quality. What benefit there is can primarily be obtained by allowing move insertions only at basic block boundaries. Thus, register allocators need not be designed to explicitly optimize move insertions, although allowing insertions may simplify the design of algorithms for other components.
- Coalescing, when viewed as a move elimination problem separate from register assignment, is highly separable: it can be performed as a standalone pass without materially degrading code quality. Furthermore, a simple greedy heuristic is nearly as effective as an optimal algorithm.
- When optimizing for performance on a modern processor, spill code optimization is of paramount importance. Furthermore, the various components of register allocation can be treated separately without significantly impacting performance. Therefore, when targeting processor performance, new register allocator designs should focus on solving the spill code optimization problem as the coalescing, move insertion, and register assignment problems are adequately solved using existing heuristics.
- Both spill code optimization and register assignment are important contributors when optimizing for code size. If code size is of paramount importance, then integrating these two phases can result in an additional small improvement. Therefore, when targeting code size, new register allocator designs should focus on solving both the spill code optimization and register assignment problems, possibly in an integrated framework.

Acknowledgments

We would like to thank Mike Ryan and Pillai Padmanabhan for their help and support using the Open Cirrus testbed at Intel Research Pittsburgh. This research was sponsored in part by the National Science Foundation under grant CCF-0702640.

References

[1] A. W. Appel and L. George. Optimal spilling for CISC machines with few registers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 243–253. ACM Press, 2001.

[2] P. Bergner, P. Dahl, D. Engebretsen, and M. O'Keefe. Spill code minimization via interference region spilling. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 287–295, New York, NY, USA, 1997. ACM Press.

[3] D. Bernstein, M. Golumbic, Y. Mansour, R. Pinter, D. Goldin, H. Krawczyk, and I. Nahshon. Spill code minimization techniques for optimizing compliers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–263. ACM Press, 1989.

[4] F. Bouchez. *A Study of Spilling and Coalescing in Register Allocation as Two Separate Phases*. PhD thesis, École Normale Supérieure de Lyon, France, December 2008.

[5] F. Bouchez, A. Darte, and F. Rastello. Register allocation: What does the NP-completeness proof of Chaitin et al. really prove. In *Workshop on Duplicating, Deconstructing, and Debunking*, 2006.

[6] P. Briggs. *Register allocation via graph coloring*. PhD thesis, Rice University, Houston, TX, USA, 1992.

[7] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst.*, 16(3):428–455, 1994.

[8] P. Brisk, F. Dabiri, J. Macbeth, and M. Sarrafzadeh. Polynomial time graph coloring register allocation. In *14th Internat. Workshop on Logic and Synthesis*. ACM Press, 2005.

[9] G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 98–101. ACM Press, 1982.

[10] K. D. Cooper and L. T. Simpson. Live range splitting in a graph coloring register allocator. In *Proceedings of the 7th International Conference on Compiler Construction*, pages 174–187, London, UK, 1998. Springer-Verlag.

[11] C. Fu, K. Wilken, and D. Goodwin. A faster optimal register allocator. *The Journal of Instruction-Level Parallelism*, 7:1–31, January 2005.

[12] L. George and A. W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3):300–324, 1996.

[13] D. Grund and S. Hack. A Fast Cutting-Plane Algorithm for Optimal Coalescing. In S. Krishnamurthi and M. Odersky, editors, *Compiler Construction*, Lecture Notes In Computer Science. Springer, March 2007. Braga, Portugal.

[14] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. Mibench: A free, commercially representative embedded benchmark suite. *IEEE International Workshop on Workload Characterization*, pages 3–14, December 2001.

[15] S. Hack and G. Goos. Optimal register allocation for ssa-form programs in polynomial time. *Information Processing Letters*, 98(4):150–155, May 2006.

[16] S. Hack and G. Goos. Register Coalescing by Graph Recoloring. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 227–237, New York, NY, USA, 2008. ACM.

[17] ILOG CPLEX. `http://www.ilog.com/products/cplex`.

[18] Intel 64 and ia-32 architectures software developer's manual. `http://www.intel.com/products/processor/manuals/`.

[19] D. R. Koes and S. C. Goldstein. A global progressive register allocator. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 204–215, New York, NY, USA, 2006. ACM Press.

[20] T. Kong and K. D. Wilken. Precise register allocation for irregular architectures. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 297–307. IEEE Computer Society Press, 1998.

[21] The LLVM compiler infrastructure. `http://llvm.org`.

[22] C. R. Morgan. *Building an Optimizing Compiler.* Butterworth, 1998.

[23] J. Palsberg. Register allocation via coloring of chordal graphs. In *Proceedings of the thirteenth Australasian symposium on Theory of Computing*, pages 3–3, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc.

[24] J. Park and S.-M. Moon. Optimistic register coalescing. *ACM Trans. Program. Lang. Syst.*, 26(4):735–765, 2004.

[25] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913, 1999.

[26] V. Sarkar and R. Barik. Extended linear scan: an alternate foundation for global register allocation. In *Proceedings of the 2007 International Conference on Compiler Construction*, March 2007.

[27] D. Seal. *ARM Architecture Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[28] SPEC CPU2006 benchmark suite. `http://www.spec.org`.

[29] O. Traub, G. Holloway, and M. D. Smith. Quality and speed in linear-scan register allocation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 142–151, New York, NY, USA, 1998. ACM Press.

[30] C. Wimmer and H. Mössenböck. Optimized interval splitting in a linear scan register allocator. In *Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments*, pages 132–141, New York, NY, USA, 2005. ACM Press.

[31] T. Zeitlhofer and B. Wess. Optimum register assignment for heterogeneous register-set architectures. In *Proceedings of the 2003 International Symposium on Circuits and Systems*, volume 3, pages III–252–III–244, May 2003.

[32] T. Zeitlhofer and B. Wess. A comparison of graph coloring heuristics for register allocation based on coalescing in interval graphs. *Proceedings of the 2004 International Symposium on Circuits and Systems*, 4:IV–529–32 Vol.4, May 2004.