Progressive Spill Code Placement

Dietmar Ebner
Vienna Univ. of Technology
ebner@complang.tuwien.ac.at

Bernhard Scholz
University of Sydney
scholz@it.usyd.edu.au

Andreas Krall
Vienna Univ. of Technology
andi@complang.tuwien.ac.at

ABSTRACT

Register allocation has gained renewed attention in the recent past. Several authors propose a separation of the problem into decoupled sub-tasks including spilling, allocation, assignment, and coalescing. This approach is largely motivated by recent advances in SSA-based register allocation that suggest that a decomposition does not significantly degrade the overall allocation quality.

The algorithmic challenges of intra-procedural spilling have been neglected so far and very crude heuristics were employed. In this work, (1) we introduce the *constrained min-cut (CMC) problem* for solving the spilling problem, (2) we provide an integer linear program formulation for computing an optimal solution of CMC, and (3) we devise a progressive Lagrangian solver that is viable for production compilers. Our experiments with Spec2k and MiBench show that optimal solutions are feasible, even for very large programs, and that heuristics leave significant potential behind for small register files.

Categories and Subject Descriptors

D.3.4 [**Software**]: Programming Languages

General Terms

Algorithms

Keywords

spilling, register allocation, constrained min-cut, SSA form

1. INTRODUCTION

Register allocation is a fundamental optimization in compilers for embedded systems because of the gap between processor speed and memory bandwidth of modern computer architectures. The objective is to assign hardware registers to program variables. Program variables that are not alive at the same time can use the same register. If too many program variables are alive at the same time, some of the variables will not fit in the register file and are thus stored in memory, which we refer to as *spilling*.

Spilling is costly in terms of code size, performance, and energy consumption. Code size is increased because of additional instructions that are issued to transfer values of program variables between memory and registers due to spilling. We refer to these additional instructions as *spill code*. Spilling also has a detrimental impact on the execution time of a program as memory accesses take significantly longer to execute than register instructions and the overall energy dissipation suffers because of the higher workload for the memory system. Thus, whether for general-purpose or for embedded computing, minimizing the spill overhead is of paramount importance.

A large fraction of previous work on register allocation employs the *spill-everywhere* model, i.e., for spilled program variable v the compiler issues store instructions after definitions of v and load instructions before uses of v. This approach may lead to a sub-optimal code quality because in the entire range where the spilled program variable is alive it is assumed that there are more program variables alive then registers available. However, this does not hold in practice.

In this paper, we consider a more flexible model also known as *load-store optimization*, where live ranges of program variables are split arbitrarily. Live ranges of program variables can be split by storing and loading their values to and from memory, respectively. We seek a minimal-cost placement for loads and stores caused by spilling such that at most k program variables are alive at any point in the program where k is the number of available registers in the register file.

Spilling has been mainly considered in the context of a particular register allocation scheme and not as a separate transformation. Hence, the proposed strategies and metrics are often based on the particular program representation that is used by the register allocation algorithm, e.g., live intervals in the case of linear scan allocators or interference graphs for graph coloring based techniques. Usually, spilling is invoked only if the allocation algorithm fails, often leading to backtracking or iteration of the whole process. Our approach employs static single assignment (SSA) form as underlying program representation and complements the recent advances of register allocation for SSA form.

In summary, the key contributions of this paper are:

1. the *constraint min-cut problem,* which is a well-defined combinatorial problem that solves spill-code placement,
2. an integer linear program formulation for the constraint min-cut problem to compute an optimal solution,

3. a Lagrange relaxation algorithm for the constraint min-cut problem whose precision can be traded for run-time making the algorithm very suitable for compilers,

4. an experimental evaluation of a state-of-the-art heuristic, an optimal integer linear programming solution, and a Lagrange algorithm with the Spec2k and MiBench benchmark suite.

The paper is organized as follows: In Sec. 2 we survey the related work. In Sec. 3 we present a motivating example and model the spill code placement as a network problem. In Sec. 4 we introduce the constraint min-cut problem formally. In Sec. 5 we devise a polynomial time algorithm using Lagrangian relaxation. In Sec. 6 we provide experimental results with the Spec2k and MiBench benchmark suite, and in Sec. 7 we draw our conclusions.

2. RELATED WORK

Chaitin et al. [CAC$^+$81] established the connection of register allocation with graph coloring of interference graphs. Nodes of an interference graph represent program variables and an edge between two variables imply that they are concurrently alive at some point in the program. Finding a vertex coloring of an interference graph with at most k colors where k denotes the number of machine registers, is a NP complete problem and a solution cannot be approximated by a constant factor, unless P is equal to NP. The chromatic number of an interference graph can be greater than k and some program variables have to be placed in memory as a consequence, which we refer to as *spilling*. Numerous follow-up papers introduce improvements of Chaitin heuristic for graph coloring on two particular sub-problems: improving colorability and *coalescing* [CH90, BCT94, GA96, PM98]. Coalescing aims to eliminate copy instructions of program variables by assigning both variables the same machine register. What is common to graph coloring based approaches and most techniques [GW96, PS99, KG06] proposed is that both register allocation/spilling and coalescing are considered to be inherent sub-problems that are solved concurrently.

Appel and George [AG01] were among the first who proposed a *two-phase* approach where spilling is performed in a separate phase prior to the actual register assignment problem. Their work demonstrate that the decomposition of register allocation into separate phases does not significantly degrade the overall code quality. In general it is not guaranteed that there is a valid k coloring even if there are no more than k variables simultaneously live at any given point in the program. To overcome this problem, the authors introduced parallel copies of program variables, which are to be removed at a later coalescing phase.

Static Single Assignment (SSA) form [CFR$^+$91] is an intermediate representation of programs spliting existing variables into versions denoted by the original program variable name and a sub-script describing the version. At confluence points, new versions of program variables are introduced that merge several definitions of program variables with so-called Φ-functions non-deterministically. SSA is called strict if all uses are dominated by their definition. Interference graphs of SSA graphs are chordal. This has been discovered independently by several research groups [HG06, BDMS05, HGG06, PP05]. A graph is called chordal if each cycle of length four or more has a chord, which is an edge joining two non-adjacent nodes in the cycle. As chordal graphs are a subset of perfect graphs, they inherit their properties. Most importantly, the chromatic number of perfect graphs equals the size of the largest clique. This property even holds for each induced sub-graph. There is a so-called *perfect elimination order* that allows to color perfect graphs optimally in $\mathcal{O}(n+m)$ where n is the number of vertices and m is the number of edges in the interference graph. At confluence points, Φ-functions introduce swap instructions for variables that are assigned different registers on different incoming paths and the large number of additional copy-related variables caused by Φ-functions complicates coalescing. Recent work [GH07] suggests that the problem can be solved efficiently in practice, and near-optimal polynomial time algorithms are likely to be found in the near future.

One of the first heuristic algorithms directly solving a simple variant of the spilling problem has been proposed by Belady [Bel66] in an operating system context, namely paging of virtual memory with write back. This problem corresponds to the spilling problem for basic blocks. Belady's algorithm is a furthest-first heuristic, i.e., the variable that is used furthest in the future is evicted. Guo et al. [GGP04] empirically showed that this simple heuristic can be both efficient and effective, especially for large basic blocks. Another approach for the local spilling problem was introduced by Hsu et al. [HFG89] that maps the spilling problem to the shortest path problem in a weighted directed graph that grows exponentially with the number of variables and registers. To limit the exponential growth, the authors propose pruning rules to compute problem input sizes with up to 100 nodes. Important theoretical insight into local register allocation and spilling has been contributed by Farach [FCL00]. Farach considers both a simplified version where stores are disregarded (weighted caching) and a more complex variant where both spills and re-loads are minimized. For the first problem, Farach presents an *ILP* with the consecutive ones property. Thus, there is an equivalent minimum cost network flow problem that can be solved in polynomial time. For the more complex variant, the author proves NP completeness and an efficient 2-approximation is presented.

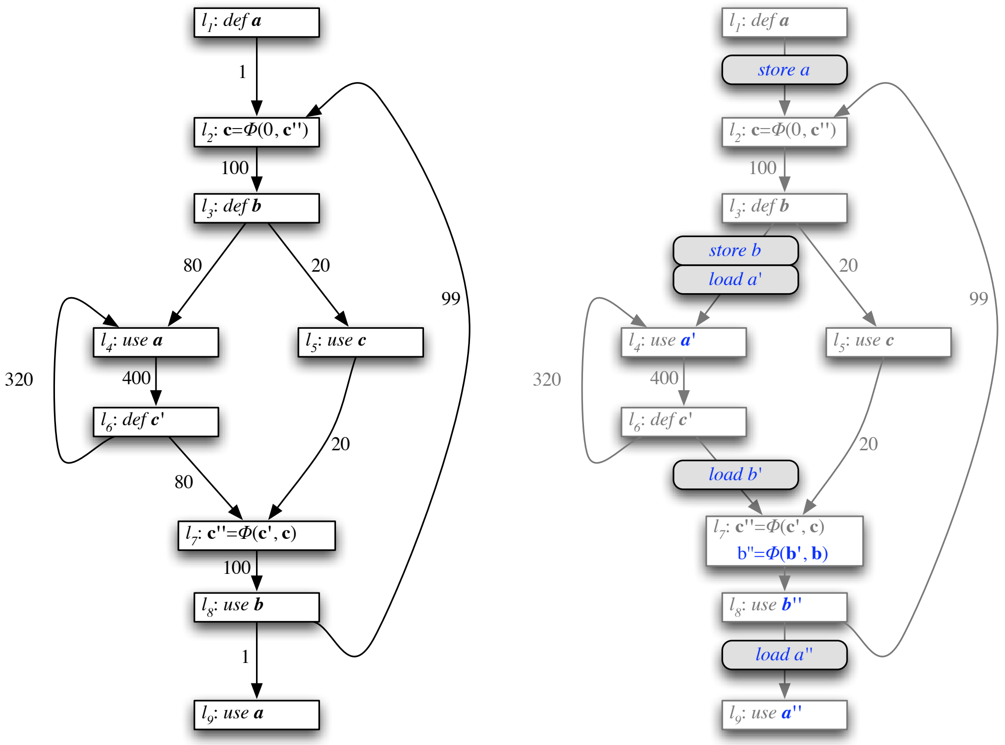
The intra-procedural extension of Belady's algorithm was introduced by Hack [Hac07]. The heuristic is applied separately to each basic block. The notion of "furthest first" is extended to the global scope by recursively computing the minimum over all successor blocks. In a separate step, the partial solutions computed by applying the heuristic locally are combined to obtain a solution for the whole function. This approach can be implemented efficiently for SSA form. Complexity results and heuristics for various variants of the spilling problem are given by Bouchez et al. [BDR07].

3. MODELING

For the spilling problem we use as an input control flow graphs (CFG) in static single assignment (SSA) form [CFR$^+$91]. The SSA form is a program representation in which all variables have a single static assignment and Φ-functions are placed at confluence points to merge definitions of the same program variable. We denote the set of variables of the SSA form by $\mathcal{V}$. Each node in the CFG corresponds to a labeled single instruction ℓ of the form

$$\ell : (d_1, \ldots, d_m) \leftarrow \mathrm{op}(u_1, \ldots, u_n).$$

The sets $\mathcal{D}_\ell \subseteq \mathcal{V}$ and $\mathcal{U}_\ell \subseteq \mathcal{V}$ denote the subset of variables



(a) Input program in SSA form (b) Spilling transformation for two registers

Figure 1: Motivating example

defined and used at ℓ, respectively. For each $v \in \mathcal{V}$, def(v) denotes the unique label defining v. Liveness for strict variables is defined in the usual way: a variable v is *live* at label ℓ, if there is a (possibly empty) path from ℓ to a label ℓ' such that $v \in \mathcal{U}_{\ell'}$ and the path does not include def(v). The set of variables live at label ℓ is denoted by $\mathcal{L}_\ell$. Two variables $v, v' \in \mathcal{V}$ are said to *interfere* iff there exists a label in the program where both are live. For example, the graph in Fig. 1(a) shows a node-labeled CFG in SSA form with variables ***a***, ***b***, ***c***, ***c'***, and ***c''***. For the latter variable, Φ-functions have been inserted that disambiguate multiple reaching definitions. Edge frequencies are denoted along the arcs in the CFG.

In our example, variable ***a*** is live at labels l_2 to l_9 while the live range of variable ***b*** spans across labels l_4 to l_8. Note the special meaning of ϕ-functions in this respect: their arguments are only used if control flow enters a label along the corresponding in-edge, e.g., variable ***c*** is live at label l_5 but not at l_6. For the consideration of liveness, we can treat phi functions as if their arguments are used at the end of the particular predecessor block. We are assuming a RISC architecture where all arguments to an instruction have to reside in a register. Likewise, results are always written into one or more destination registers. Transfers from memory into registers and vice versa can be accomplished using explicit load and store instructions. Consequently, the number of variables live at a particular label ℓ cannot exceed $k - |\mathcal{D}_\ell|$ in order to allow for a coloring with k registers.

The objective of spilling is to insert load- and store-instructions along the edges of the CFG in order to split live ranges such that that the overall costs are minimized. Costs may be constant to minimize for code size, proportional to the edge frequencies to optimize for execution time, or any combination of the two. Furthermore, we can easily add support for re-materialization by justifying the cost-function accordingly. This is most useful for constants or constant expressions such as frame-pointer indirect addressing.

Assuming a total number of two registers, we show a cost-minimal transformation with respect to the given edge-weights for our example in Fig. 1(b). Live ranges for variables ***a*** and ***b*** have been split by storing them to a dedicated location in memory at the point of definition and re-loading them before they are used. In order to maintain SSA form, we have to insert additional Φ-functions and rename references to reflect those changes accordingly [CFR+91].

Many algorithms insert re-load instructions for spilled variables right before they are used. However, this can lead to inefficiencies, e.g., inserting a re-load for variable ***a*** in Fig. 1(b) within the inner loop right before label l_4 increases the costs of the transformation significantly.

Considering a *single* variable v, we can compute the overall costs for reloading its value from memory using a simple min-cut computation. We transform the control flow graph into a weighted network N_v as shown in Fig. 2. For each node in the CFG at which v is live, we generate a node in our min-cut network. Likewise, edges are introduced that reflect the cost of inserting a re-load instruction. The weighted network N_v has dedicated nodes for the source s and sink t, respectively.

For all successors of def(v), we add an additional edge from s to the corresponding node in N_v with weight zero. This reflects the fact that a value is always available right

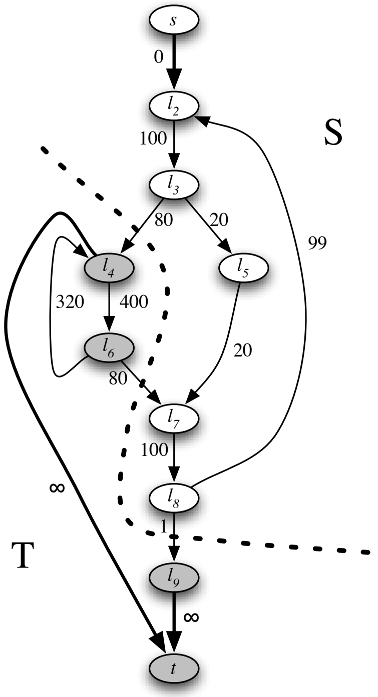


Figure 2: Modeling for a single variable

after its definition. Furthermore, each label where v is used is connected to the artificial t-node with costs ∞. The intention of this transformation is as follows: any s-t cut with weight less than ∞ corresponds to a valid segmentation of the original live range such that re-load instructions are placed on every cut-edge passing from a node in S to a node in T. The weight of the cut reflects precisely the costs of the transformation in the chosen cost model. The cut that has been chosen for the example in Fig. 1 is depicted by the bold dotted line. It consists of the edges (l_3, l_4) and (l_8, l_9) which are the places where we inserted the re-load instructions before.

Solving the min-cut problems isolated for each variable does not lead to a meaningful solution as the model always allows for the trivial solution where the S-partition is constituted by nothing but the s-node. This is the equivalent of assigning a machine register to each variable for the entire live range. In order to account for register constraints, we identify source and sink nodes for each of the generated networks (one per variable). Thus, we obtain a combined network flow problem for the whole function. Nodes in the combined graph are assigned to disjoint partitions. For each label ℓ in the CFG, we define a partition Q_ℓ that includes all the nodes from networks N_v that correspond to label ℓ. Thus, each partition Q_ℓ includes one node per variable live at ℓ. Furthermore, a partition Q_ℓ has *capacity* $k - |\mathcal{D}_\ell|$. Intuitively, the capacity of a partition denotes the number of live ranges that may pass through a particular label without exceeding register constraints.

We can thus reduce the problem where to insert re-loads to the problem of finding a minimum cut in the combined network subject to the conditions $|T \cap Q_\ell| \leq k - |\mathcal{D}_\ell|$ for each partition Q_ℓ. This model allows us to formulate spilling as a well-defined combinatorial optimization problem and has

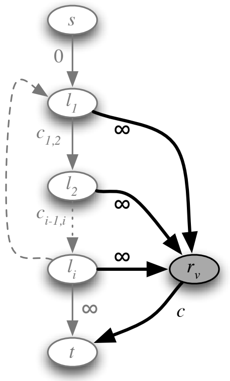


Figure 3: Accounting for store costs

an underlying network flow structure. The model accomodates the design of simple greedy heuristics as it is straightforward to find feasible solutions, e.g., by heuristically assigning $|Q_\ell| - k + |\mathcal{D}_\ell|$ nodes to the S partition.

The proposed model assumes that the value of each definition is always available in memory. It is sufficient but not necessarily optimal to store a value right after its point of definition. As these spill instructions usually do not come for free, it might be desirable to account for their costs within the optimization model. For each variable v, an additional node r_v is inserted. We introduce an additional arc (r_v, t) whose weight corresponds to the costs of storing the particular variable as depicted in Fig. 3. We have to account for those costs only if there is at least one re-load operation. In other words, there is a node other than s that is assigned to the S partition. We can model this constraint by adding additional arcs with weight ∞ from each label other than s and t to the newly introduced node r_v. Those edges imply that $r_v \in S$ if any of the adjacent nodes other than t is in S. For values that allow for re-materialization such as initializations with constants or constant expressions, store costs are usually 0 and the additional node can be safely removed. Otherwise, the same considerations discussed for re-loads apply: the cost function can be used to optimize for any combination of code size and performance.

4. THE CONSTRAINED MIN-CUT (CMC) PROBLEM

In the following, we describe the constrained min-cut problem as a combinatorial problem, as a quadratic and as a linear mathematical program.

Let $G(V, E)$ be a digraph with edge weights $w_{uv} \in \mathbb{Z}^+$, for each edge $(u, v) \in E$. Further, let $s, t \in V$ denote two distinguished vertices and let $P = \{\{s\}, \{t\}, Q_3, \ldots, Q_r\}$ denote a disjoint partitioning of V. Each disjoint set Q_i, for all i, $(1 \leq i \leq r)$, has associated a capacity $c_i \in \mathbb{Z}^+$. Find a separation of V into two disjoint sets S and T, ($s \in S$ and $t \in T$) whose cut set $\sum_{(u,v) \in E, u \in S, v \in T} w_{uv}$ is minimal such that $|T \cap Q_i| \leq c_i$ for all i, $(1 \leq i \leq r)$. We may state the CMC problem as a quadratic integer program as follows,

$$
\begin{array}{lll}
\min & \sum_{(u,v) \in E} w_{uv}(1 - x_u)x_v & \\
s.t & x_t - x_s \geq 1 & \\
 & \sum_{u \in Q_i} x_u \leq c_i & \text{for all } i, 1 \leq i \leq r \\
 & x_u \in \{0, 1\} & \text{for all } u \in V
\end{array}
\tag{1}
$$

The variables x_u are 0-1 integer variables with the interpretation that x_u is zero if vertex u is in S and one if it is in T. By linearizing the above quadratic mathematical program we obtain the following integer linear program,

$$\begin{array}{lll} \min & \sum_{(u,v)\in E} w_{uv} y_{uv} & \\ \text{s.t.} & x_t - x_s \geq 1 & \\ & x_u - x_v + y_{uv} \geq 0 & \text{for all } (u,v) \in E \\ & \sum_{u\in Q_i} x_u \leq c_i & \text{for all } i, 1 \leq i \leq r \\ & x_u \in \{0,1\} & \text{for all } u \in V \\ & y_{uv} \in \{0,1\} & \text{for all } (u,v) \in E \end{array} \tag{2}$$

Disregarding the capacity constraints in line 4 of Eq. 2, the model corresponds to the *s-t* min-cut problem whose constraint matrix is *totally unimodular* [AMO93], i.e., the solution of the relaxed problem gives the optimal integral solution. This implies that a polynomial time solver for linear programming may be employed to solve the problem (e.g. interior-point). Better performing *s-t* min-cut algorithms exist [SW97] or max-flow algorithms can determine the solution of an instance of the *s-t* min-cut problem because the *s-t* min cut is the dual problem of max-flow. Nevertheless, having the capacity constraints of line 4 in Eq. 2 renders the problem NP complete.

Note that the integer linear program in Equation 2 provides already a feasible algorithmic approach to the CMC problem. As our experiments show, mature integer linear programming solver technology can solve large instances within reasonable time limits. However, integer linear programming solvers are too heavy weight in terms of runtime and memory consumption for a compiler setting.

5. LAGRANGIAN RELAXATION

Lagrangian relaxation [AMO93] is a general solution approach for mathematical programs and is quite often applicable for problems with an embedded network structure like CMC. In the following we are applying Lagrangian relaxation to CMC to devise a polynomial-time algorithm with near-optimal solutions, which does not rely on integer linear programming.

Consider the CMC formulation presented in Equation 2 and let X denote the set of solutions that satisfy the constraints of the *s-t* min-cut problem without the additional capacity constraints in Line 4. We may express CMC problem as $z^* = \min \{ax : x \in X, Bx \leq c\}$ where constraints $Bx \leq c$ denotes the capacity constraints for disjoint sets Q_i with capacity vector c. By relaxing the constraints, we obtain the Lagrangian relaxation $L(\mu) = \min \{ax + \mu(Bx - c) : x \in X\}$. We hereby remove the constraints of the mathematical program and transform them to a term of the objective function with associated *Lagrangian multipliers* μ. Hence, a solution of the relaxed problem will not necessarily be a feasible solution of the CMC problem. LP theory states that for any value for the Lagrangian multipliers μ, the value $L(\mu)$ is a *lower bound* on the optimal objective value of the CMC problem. The tightest possible bound is obtained by the *Lagrangian multiplier problem* $L^* = \max_{\mu\geq 0} L(\mu)$ and we have the well-known relations $L(\mu) \leq L^* \leq z^*$.

The Lagrangian multiplier problem provides an optimality test: for any vector μ, if x is a feasible solution to the CMC problem and $L(\mu) = ax$, then $L(\mu)$ is an optimal solution of the Lagrangian multiplier problem and x is an optimal solution for the CMC problem. Furthermore, if for some choice of μ, the solution x^* of the Lagrangian relaxation is feasible for CMC and x^* satisfies the *complementary slackness condition* $\mu(Bx^* - c) = 0$, then x^* is an optimal solution of the CMC problem as well.

Lagrangian relaxation is useful for the design of an algorithm if we can find an efficient algorithm for the Lagrangian multiplier problem. The Lagrangian multiplier problem $L(\mu)$ of CMC may be stated as follows:

$$\begin{array}{lll} \min & \sum_{(u,v)\in E} w_{uv} y_{uv} + \sum_{i=1}^{r} \mu_i \sum_{u\in Q_i} (x_u - c_i) & \\ \text{s.t.} & x_u - x_v + y_{uv} \geq 0 & \text{for all } (u,v) \in E \\ & x_u \in \{0,1\} & \text{for all } u \in V \\ & y_{uv} \in \{0,1\} & \text{for all } (u,v) \in E \end{array} \tag{3}$$

The polyhedron defined by the set of inequalities above already corresponds to a *s-t* min-cut problem. The main difference is the objective function: the Lagrangian term $\sum_{i=1}^{r} \mu_i \sum_{u\in Q_i} (x_u - c_i)$ of the objective function has for each disjoint set Q_i an associated Lagrangian multiplier μ_i, that is multiplied by the number of nodes that are placed in the T-partition.

For a constant μ_i, term $\mu_i c_i$ in the Lagrangian term evaluates to a constant and we reduce the Lagrangian term to $\sum_{i=1}^{r} \mu_i \sum_{u\in Q_i} x_u = \sum_{u\in V} \mu_{\chi(u)} x_u$ where $\chi(u)$ is index i such that $u \in Q_i$. Note that index i is unique because the node set V is disjointly partitioned. For a constant μ vector, solving the Lagrangian function becomes a special case of *Stone's* problem [Sto77], which allocates a set of processes $P = \{p_1, \ldots, p_r\}$ on to processors α and β. Execution times of a process may differ depending on the processor where it is executed and there are communication costs if two communicating processes are mapped to different processors. More formally, let $w_\alpha(p)$ and $w_\beta(p)$ denote the executions time of process p on processors α and β, respectively, and let $w_{\alpha\beta}(p_1, p_2)$ denote the communication costs between processes p_1 and process p_2 when placed on α and β, respectively. The objective of Stone's problem is to find a partitioning of set P into two disjoint sets α and β such that the following objective function becomes minimal:

$$\min \sum_{p\in\alpha} w_\alpha(p) + \sum_{p\in\beta} w_\beta(p) + \sum_{p_1\in\alpha, p_2\in\beta} w_{\alpha\beta}(p_1, p_2) \tag{4}$$

Stone reduces the problem to the *s-t* min-cut problem. Processes are represented by nodes in the network and in the network there are a dedicated source node s and a dedicated sink node t. Execution costs of processor p on α and β are represented as an edge between source s and p with capacity $w_\beta(p)$ and an edge between p and t with capacity $w_\alpha(p)$, respectively. Communication costs between p_1 and p_2 are modeled as an edge with capacity $w_{\alpha\beta}(p_1, p_2)$.

We use Stone's reduction for computing the Lagrangian function $L(\mu)$ of CMC for a given constant μ. The *s-t* min-cut network is augmented with edges of the form (s, u) with capacity μ_i for $u \in V$ as depicted in Fig. 4. Edge (s, u) becomes a cut-edge iff $u \in T$.

Employing Stone's reduction gives us an an efficient algorithmic vehicle to solve the Lagrangian function for a constant μ. However, it remains to be shown how to solve the Lagrangian multiplier problem $L^* = \max_{\mu\geq 0} L(\mu)$. We use a variant of *sub-gradient optimization* that is an adaption of Newton's method for solving systems of non-linear equations. The principle idea is to gradually adapt the vector

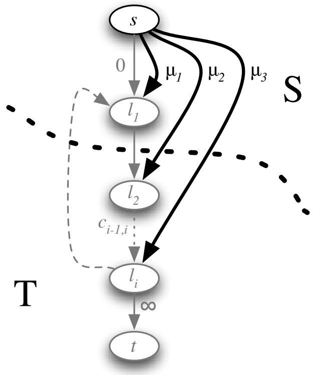


Figure 4: Reduction of the relaxed problem $L(\mu)$ to a s-t min-cut instance.

of Lagrangian multipliers $\mu^{k+1} = [\mu^k + \theta_k(Bx^k - c)]^+$ for an initial choice μ^0. The scalar θ_k denotes the *step length* in the k-th iteration and x^k is a solution vector of the subproblem $L(\mu^k)$. As we are relaxing inequalities, we never consider negative elements in our vector μ. Thus, μ^k is set to zero if the update strategy would cause it to become negative (denoted by $[\ ]^+$). The choice of the step length determines the convergence speed and we use a popular standard heuristic that adapts the step length after every iteration by $\theta_k = \frac{\lambda_k(U-L(\mu^k))}{||Bx^k-c||^2}$ where U denotes the best upper bound to the problem found so far, λ_k is a scalar that is gradually decreased, $||Bx^k - c||^2$ denotes the Euclidean norm of the inner term.

The process allows for an intuitive representation: if for any partition Q_i, the term $(Bx^k - c)_i$ is zero or negative, the capacity constraints imposed on Q_i are satisfied. However, if the term is greater than zero, μ_i serves as a *penalty* that directs the min-cut algorithm to put some of the nodes from the T to the S-partition. The step length directs the algorithm to move quickly towards the optimum at the beginning of the approximation, while we proceed with more care once we are close to the optimal value L^*. A theoretical discussion of convergence criteria and the rational for choosing the step length is beyond the scope of this work. As it is a standard technique in combinatorial optimization, the interested reader is referred to relevant literature, e.g., Ahuja et al. [AMO93].

In general, a solution to the Lagrangian multiplier problem is not necessarily feasible for the CMC problem. A popular approach is to use the values obtained from the relaxation in order to solve the remaining problem using enumeration techniques such as branch-&-bound. The efficiency of those techniques largely depends on the size of the *duality gap*. An interesting result in combinatorial optimization states that the bound obtained from Lagrangian relaxation is always as tight as the bound obtained from an LP relaxation. An even stronger statement for problems satisfying the *integrality property* (such as s-t min-cut) guarantees that both bounds are equal. Thus, solving the Lagrangian multiplier problem is equivalent to solving the LP relaxation but does not rely on linear programming and can often be solved more efficiently.

An alternative technique that can be used to obtain near-optimal solutions is the use of so-called *Lagrange heuristics*. The main idea is that solutions to the Lagrangian multiplier problem are usually very close to the solution of the CMC problem while only a small number of relaxed constraints remain violated. A Lagrange heuristic is an algorithm that resolves violated constraints in a greedy manner, often achieving close to optimal results. An additional advantage is that the computed bound provides us with an *performance guarantee* in respect to the optimal solution.

6. EXPERIMENTAL EVALUATION

We have implemented and evaluated the proposed techniques using LLVM 2.4 – a compiler infrastructure built around an equally named fully typed low level virtual machine [LA04]. All programs have been cross compiled using one core of a Xeon DP 5160 3GHz. The ILP formulation for CMC is solved using ILOG CPLEX(tm) 10.

First, we are interested in the potential for exact spilling compared to heuristics. As a reference, we use the default allocator of LLVM 2.4 – an improved implementation of linear scan register allocation with backtracking in the case of spilling. We compare results for a varying number of available registers with a modified backend where we perform spilling right before the standard register allocator. Register allocation in LLVM is traditionally performed after SSA elimination. Thus, even though we spill a sufficiently large number of registers, the register allocator might insert additional spill code that is not strictly necessary. We present benchmarks for two representative embedded architectures: a 4-way VLIW core for audio-/video-decoding and an ARM processor as an example for an embedded RISC architecture.

The first experiment is based on an OnDemand(TM) CHILI core – a regular 4-way VLIW load/store architecture with 64 general purpose registers and some specialized instruction set extensions for multimedia applications. Execution times have been gathered using a cycle-accurate simulator. We use typical benchmarks reflecting the characteristics of embedded media applications provided by OnDemand(TM), most of them are taken from the freely available MiBench suite. Both the simulator and the compiler have been modified to support a varying number of registers, allowing us to make experiments for several different settings.

We present data for various configurations with 8, 12, 16, and 32 registers in Fig. 5. For small register files, the achieved speedup on top of LLVM is substantial for all benchmarks, ranging from about 8% to almost 30%. Not surprisingly, the potential for improved spilling techniques decreases with increasing number of machine registers. On average, there is an improvement of about 15.5% for the scarcest setting, gradually decreasing to 6.5%, 3%, and 0.9% for 12, 16, and 32 registers respectively. None of the benchmarks showed additional speedup for 64 registers.

Solver times for those benchmarks are definitely within practical limits. Our ILP-based algorithm finished within a few seconds for most of the benchmarks, the most difficult being automotive-susan and video-h263 with 24 and 23 seconds respectively. This includes the time spent on the CMC reduction as well as SSA reconstruction after insertion of spill code.

In order to test our approach on larger benchmarks, we consider the widely-used SPECINT 2000 suite. Running those benchmarks on a bare-metal VLIW is infeasible as they require an underlying operating system and a complete

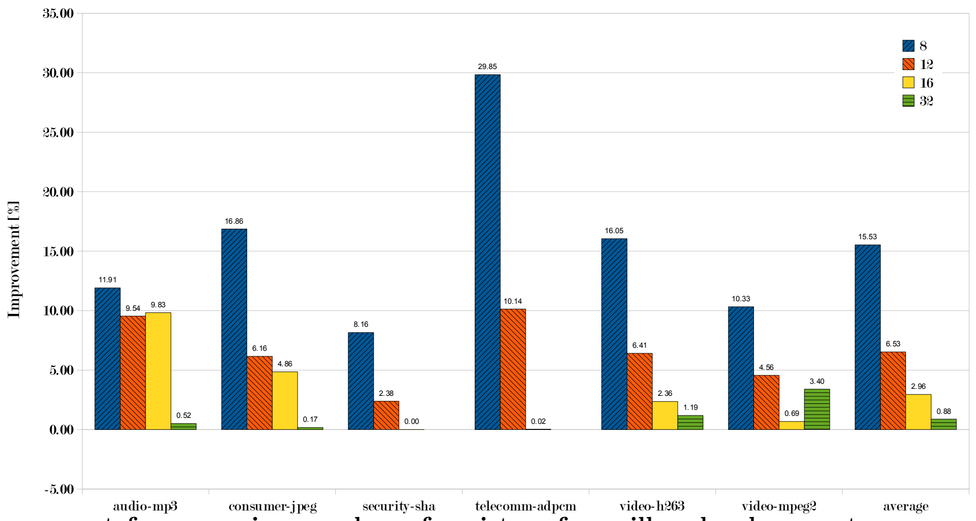


Figure 5: Improvement for a varying number of registers for spill code placement compared to the linear scan allocator of LLVM for a 4-way VLIW processor.

`libc` implementation. Therefore, we use an ARMv7 board (OMAP3 EVM) at 500 MHz with 128MB of main memory running a Linux 2.6.22 kernel. Profiles are obtained using the input set "train" while the reported execution times were gathered using the "test" inputs. Floating point operations are emulated using a IEEE754 softfloat library. We measured execution times using the unix `time` utility considering the best out of 10 runs on an unloaded machine for each configuration.

ARM processors support two different instruction sets: ARM and ARM Thumb. While in ARM mode, the processor fetches 32-bit instructions and has access to the full register file with 16 registers. Thumb mode is a more compressed 16-bit instruction set architecture that allows for smaller code size at a significant performance penalty. In Thumb mode, most instructions can only access the lower half of the register file and only a subset of the addressing modes is available. Hence, the register pressure is much higher than in ARM mode. Note, that some of the architectural registers are special-purpose and cannot be used for generic program variables, e.g., program counter or stack pointer.

Table 1 shows the results for both ARM and ARM Thumb execution. We use (1) a generalization of Belady's algorithm as proposed in [HGG06] (BLY) as a baseline, (2) LLVM 2.4 standard spilling heuristic, and (3) our spill code placement based on CMC using CPLEX to compute an optimal solution. Note that in ARM Thumb mode two benchmarks fail due to bugs in the LLVM backend that we could not yet solve (i.e., 176.gcc and 300.twolf). On average, LLVM is about on par with a gcc cross-compiler at its highest optimization level. The spilling heuristic of LLVM shows an improvement of 4.91 and 4.33 percent for ARM and ARM Thumb mode respectively compared to the generalized Belady heuristic. The algorithm has been implemented as described in [HGG06], i.e., we compute partial solutions for basic blocks that are heuristically combined to obtain a solution for the whole function. The main reason for the performance regressions compared to the other algorithms is that there is no explicit considerations of loop structures and block weight, which often leads to avoidable spill code within inner loops. The speedup obtained with our CMC reduction compared to BLY is about 10.44% on average for ARM Thumb mode and 7.79% for ARM.

Lagrange Relaxation.

To overcome the limitations of ILP solvers, we have shown in Section 5 how the relaxation of capacity constraints leads to a problem that can be solved using efficient generic network flow algorithms. We evaluate the general solution approach for three different greedy Lagrange heuristics with increasing algorithmic complexity. For any partial solution, we visit each partition Q_i in order, while evicting $[|T \cap Q_i| - c_i]^+$ many nodes from the T partition according to one of the following strategies:

- *SIMPLE* Nodes are simply ordered according to their effect on the objective function, preferring those causing a low penalty.
- *REGION* This is basically the same strategy as before with the addition that we also remove nodes in the neighborhood as long as this does not *increase* the overall penalty. As we might well decrease the overall effect on the objective function, this approach can be seen as a simple hill-climbing heuristic.
- *MIN-CUT* This algorithms computes for each node within a partition the optimal set of nodes to be moved from the T to the S partition such that the overall weight of the cut is minimized.

Fig. 6 shows the average ratio of the optimal solution and the solution obtained from each of the greedy heuristics. The x-axis denotes the number of iterations for the sub-gradient approximation algorithm. The average quality for the pure heuristic without the Lagrangian relaxation corresponds to $x = 0$. The two simple strategies perform initially very poor with an average of only 25% and 41% respectively. The graph clearly shows how approximations to the Lagrangian multiplier problem effectively guide the heuristics towards the optimum. After thirty iterations, the average quality for the simple strategies is improved to 63% and 75%. The MIN-CUT heuristic shows initially an average quality of almost 91% and climbs up to more than 95%. However, com-

			ARM			ARM Thumb		
Benchmark	Source [LOC]	CPLEX [sec]	BLY [sec]	LLVM [%]	CMC [%]	BLY [sec]	LLVM [%]	CMC [%]
164.gzip	5615	12.72	12.11	13.07	15.44	14.00	12.00	17.25
175.vpr	11301	103.44	12.85	6.64	6.73	16.50	1.54	0.61
176.gcc	132922	483.39	7.89	8.83	13.20	n/a	n/a	n/a
181.mcf	1494	1.16	1.30	1.56	1.56	1.41	1.44	4.44
186.crafty	12939	75.37	30.31	13.86	12.18	40.30	16.40	12.32
197.parser	7763	19.88	12.18	0.91	3.92	14.14	-5.80	2.54
253.perlbmk	72206	285.81	1.70	0.00	8.97	1.78	1.00	4.71
254.gap	35759	47.15	3.67	-0.81	3.67	4.21	-1.64	34.94
255.vortex	49232	1026.44	33.76	-0.79	-1.86	42.39	6.99	10.30
256.bzip2	3236	18.32	27.83	0.69	6.02	40.35	9.98	10.55
300.twolf	17822	124.74	1.34	11.67	17.54	n/a	n/a	n/a
Mean			13.18	**4.91**	**7.79**	19.45	**4.33**	**10.44**

Table 1: Experimental results using the SPECINT 2000 benchmark suite showing the execution times for both ARM and ARM Thumb mode. We compare three different algorithms: a generalized furthest first heuristic (BLY), the linear scan register allocator from LLVM 2.4 (LLVM), and our improved spill code placement algorithm (CMC). All results are relative to BLY. We use the geometric mean for speedups; for the absolute figures we use the arithmetic mean.

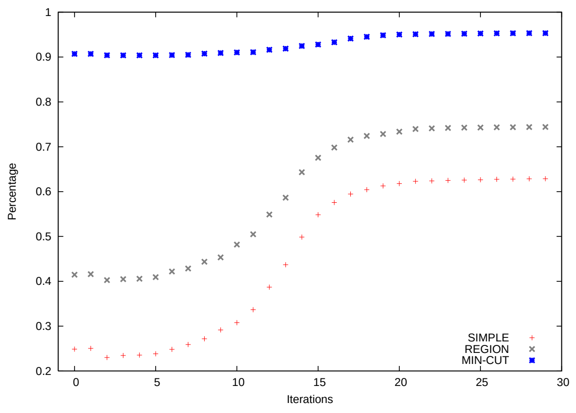


Figure 6: Average quality of three different Lagrange heuristics compared to the precomputed optimal solution over the whole SPECINT 2000 benchmark set. The x-axis denotes the number of iterations for the sub-gradient optimization algorithm.

puting the metric for this heuristic is only feasible for a small number of violated partitions. Initially, solution times might even exceed the time required for the provably optimal ILP approach.

Detailed performance results for the various algorithms are given in Figure 7. Each figure shows the runtime for all non-trivial benchmarks from the SPECINT 2000 suite. The y-axis shows the runtime of the algorithm in seconds. Each of the plots (a) to (e) features a polynomial asymptotic function that has been computed using a least squares approximation. Note that y-axis are drawn with logarithmic scale and that (a) and (b) show a smaller value range.

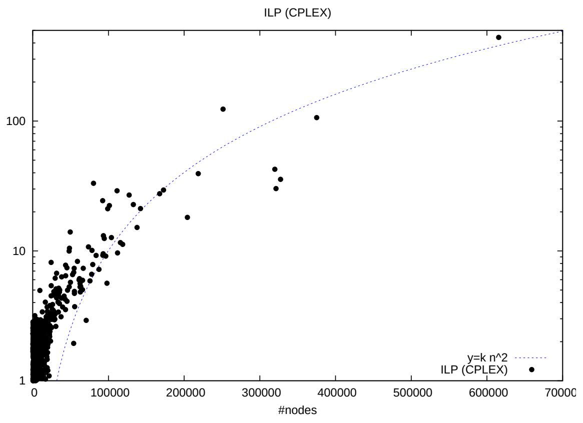
The performance of integer linear programming based algorithms strongly depends on the particular solver. It is very interesting that most of the problems are integral, even if integrality constraints are dropped. Among the whole benchmark suite, only 18 problems have a non-integral solution and require branch & bound. Thus, in practice almost all the time is spent in the simplex algorithm solving the LP relaxation. We compare two different ILP solvers: ILOG CPLEX and the open source GNU linear programming kit (glpk)[1]. Performance results for both solvers are shown in Figure 7 (a) and (e) respectively. While both algorithms indicate an asymptotic quadratic runtime in practice, the constant differs by more than an order of magnitude. In fact, there were 15 instances that could not be solved by glpk within a time limit of half an hour while all benchmarks where solved by CPLEX.

The performance of the algorithms based on Lagrangian relaxation largely depends on the particular Lagrangian heuristic. At most 20 iterations have been used for the Newton approximation. The strategy SIMPLE is very fast and shows an asymptotic linear runtime in practice ($|V|^{1.02}$). Note that the worst-case performance is determined by the maxflow algorithm, which is in the order of $\mathcal{O}(|V||E|\log|V|)$. The more sophisticated Lagrangian heuristics REGION and MINCUT show quadratic behavior, but with a much larger constant.

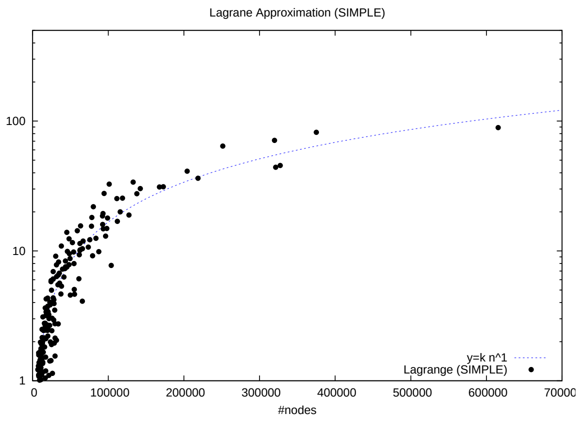
A comparison among all algorithms is given in Figure 7 (f). Note that the CPLEX based algorithm is among the fastest techniques while the GLPK solver performs worst compared to the rest of the field. The MINCUT heuristic delivers solutions that are very close to the optimum, but for significant computational costs.

Apart from Lagrangian heuristics, the proposed relaxation is very tempting for two more reasons. First, it provides us with bounds that can be used to give a provable certificate on the quality of solutions. Second, those bounds are valuable for enumeration schemes such as branch-&-bound in order to prune the search space more effectively. One last advantage we want to point out is that Lagrangian heuristics lead to *progressive* algorithms that deliver quickly feasible solutions which are gradually improved as the algorithm proceeds. Thus, we can effectively trade compile time for code quality.

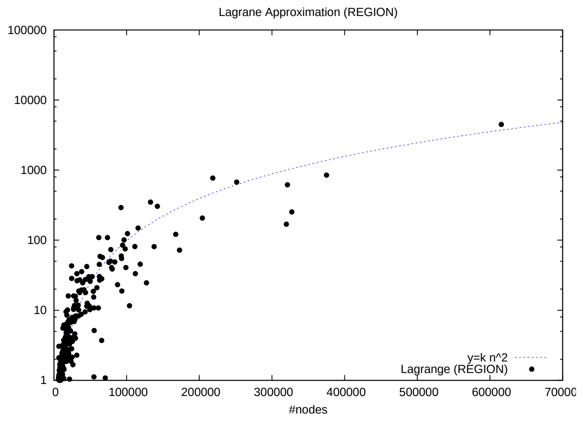
[1]`http://www.gnu.org/software/glpk`



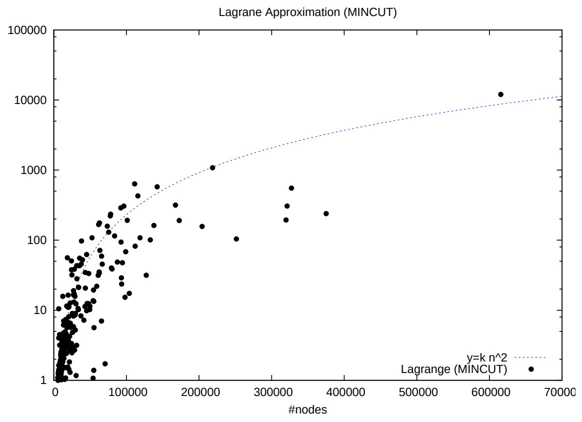
(a)



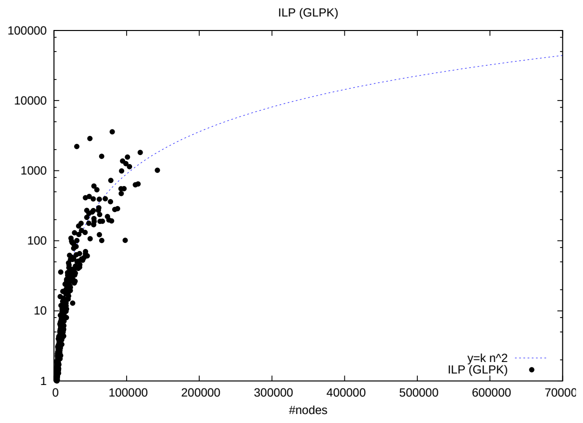
(b)



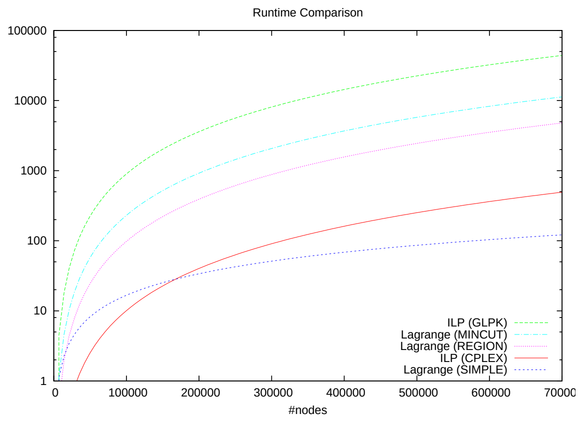
(c)



(d)



(e)



(f)

Figure 7: Runtime comparison for the various CMC algorithms.

7. CONCLUSIONS

Our results show that traditional heuristics perform sufficiently well when the number of machine registers is large, but leave significant potential for improvement on architectures with few registers. The separation of spilling from allocation and coalescing is favorable in several respects. First, it allows a separation of concerns, thereby simplifying the design and implementation of allocation/spilling frameworks for compilers. Second, it allows us to take advantage of efficient algorithms for allocation and coalescing that benefit from the chordality of interference graphs for programs in SSA form. Empirical results show that optimal spill code placement lead to performance improvements of more than 15% on average for machines with few registers.

The main contribution of our work is the reduction of spill code placement to a well-defined combinatorial problem called the *constrained min-cut* problem. The proposed model is interesting as it is based on a generic network flow substructure. We present an ILP formulation that can compute optimal solutions for standard problem sizes. In addition, we devise a Lagrangian relaxation algorithm for the constraint minimum cut problem that computes a solution progressively. The progressive nature of this algorithm allows solve-time to be traded for better code quality.

8. REFERENCES

[AG01] Andrew W Appel and Lal George. Optimal spilling for cisc machines with few registers. In *PLDI*, pages 243 – 253. ACM Press, 2001.

[AMO93] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[BCT94] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst.*, 16(3):428 – 455, 1994.

[BDMS05] Philip Brisk, Foad Dabiri, Jamie Macbeth, and Majid Sarrafzadeh. Polynomial-time graph coloring register allocation. In *International Workshop on Logic and Synthesis*. ACM Press, 2005.

[BDR07] Florent Bouchez, Alain Darte, and Fabrice Rastello. On the complexity of spill everywhere under SSA form. In Santosh Pande and Zhiyuan Li, editors, *LCTES*, pages 103–112. ACM, 2007.

[Bel66] Laszlo A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Syst. J.*, 5:78–101, 1966.

[CAC$^+$81] Gregory J Chaitin, Mark A Auslander, Ashok K Chandra, John Cocke, Martin E Hopkins, and Peter W Markstein. Register allocation via coloring. *Computer Languages*, 6:47 – 57, 1981.

[CFR$^+$91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing ssa form and the control dependence graph. *TOPLAS*, 13(4):451–490, October 1991.

[CH90] Fred C. Chow and John L. Hennessy. The priority-based coloring approach to register allocation. *ACM Trans. Program. Lang. Syst.*, 12(4):501–536, 1990.

[FCL00] Farach-Colton and Liberatore. On local register allocation. *ALGORITHMS: Journal of Algorithms*, 37, 2000.

[GA96] Lal George and Andrew W Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3):300 – 324, 1996.

[GGP04] Jia Guo, Maria Jesus Garzaran, and David A. Padua. The power of belady's algorithm in register allocation for long basic blocks. In *LCPC*, volume 2958 of *LNCS*, pages 374–390. Springer, 2004.

[GH07] Daniel Grund and Sebastian Hack. A fast cutting-plane algorithm for optimal coalescing. In *Compiler Construction*, volume 4420 of *LNCS*, pages 111–125. Springer, 2007.

[GW96] Goodwin and Wilken. Optimal and near-optimal global register allocation using 0-1 integer programming. *Software–Practice and Experience*, 26, 1996.

[Hac07] Sebastian Hack. *Register Allocation for Programs in SSA Form*. PhD thesis, Universität Karlsruhe, October 2007.

[HFG89] W. Hsu, C. Fischer, and J. Goodman. On the minimization of loads/stores in local register allocation. *IEEE Trans. on Softw. Eng.*, 15(10):1252, October 1989.

[HG06] Sebastian Hack and Gerhard Goos. Optimal register allocation for ssa-form programs in polynomial time. *Information Processing Letters*, 98(4):150–155, May 2006.

[HGG06] Sebastian Hack, Daniel Grund, and Gerhard Goos. Register Allocation for Programs in SSA-Form. In *Compiler Construction*, volume 3923, pages 247–262. Springer, March 2006.

[KG06] David Ryan Koes and Seth Copen Goldstein. A global progressive register allocator. *ACM SIGPLAN Notices*, 41(6):204–215, June 2006.

[LA04] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE Computer Society, 2004.

[PM98] Jinpyo Park and Soo-Mook Moon. Optimistic register coalescing. In *PACT*, pages 196–204. IEEE Computer Society, October 1998.

[PP05] Fernando Magno Quintão Pereira and Jens Palsberg. Register allocation via coloring of chordal graphs. In *APLAS*, volume 3780 of *LNCS*, pages 315–329. Springer, 2005.

[PS99] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895 – 913, 1999.

[Sto77] H. S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE TSE*, SE-3(1):85–93, January 1977.

[SW97] Mechthild Stoer and Frank Wagner. A simple min-cut algorithm. *J. ACM*, 44(4):585–591, 1997.