Register Spilling and Live-Range Splitting for SSA-Form Programs

Matthias Braun[1] and Sebastian Hack[2]

[1] Institut für Programmstrukturen und Datenorganisation
Universität Karlsruhe (TH)
braun@ipd.info.uni-karlsruhe.de
[2] Computer Science Department
Saarland University
hack@cs.uni-sb.de

Abstract. Register allocation decides which parts of a variable's live range are held in registers and which in memory. The compiler inserts *spill code* to move the values of variables between registers and memory. Since fetching data from memory is much slower than reading directly from a register, careful spill code insertion is critical for the performance of the compiled program.

In this paper, we present a spilling algorithm for programs in SSA form. Our algorithm generalizes the well-known furthest-first algorithm, which is known to work well on straight-line code, to control-flow graphs.

We evaluate our technique by counting the *executed* spilling instructions in the CINT2000 benchmark on an x86 machine. The number of executed load (store) instructions was reduced by 54.5% (61.5%) compared to a state-of-the-art linear scan allocator and reduced by 58.2% (41.9%) compared to a standard graph-coloring allocator. The runtime of our algorithm is competitive with standard linear-scan allocators.

1 Introduction

The register allocation phase of a compiler maps the variables of a program to the registers of the processor. Usually, the register pressure (i.e. the number of simultaneously live variables at an instruction) in a program is much higher than the number of available registers. Thus, the compiler has to generate so-called spill code that moves the contents of the variables between memory and registers. Since accessing memory is much slower than accessing a register, the amount of executed spill code has to be minimized.

The key to good spill-code generation lies in splitting the live-range of a variable at the right places: Consider a loop with excessive register pressure and a variable that is defined before the loop and used afterwards. Ideally, a compiler would store (spill) the variable in front of the loop and load (reload) the variable after the loop. If the variable was reloaded inside the loop, the reload would be executed in each loop iteration. Another example is a variable that is used in a loop but has already been spilled before the loop. Reloading this variable directly before its use in the loop will cause memory traffic in each loop iteration. Thus, it is preferable to put the reload in front of the loop.

Register allocation is often formulated as a NP-hard problem. In such a setting, the actual register demand can exceed the maximum register of a program and become NP-hard to determine. Therefore, compilers allocate registers heuristically (e.g. using graph coloring [1, 2] or linear scan [3–5]). If this heuristic runs out of registers, variables are spilled until enough registers have been freed and the heuristic can resume its work. In such a situation, the generation of spill code is driven by the failure of the allocation heuristic instead of the program's structure. In extreme cases [1, 4], such a failure results in spilling the whole live range of a variable: Stores will be put after each definition and loads in front of each use, regardless of their location in the program.

Recent results show that if the program is in SSA form, its register demand equals its maximum register pressure (see [6–8]). This allows for decoupling spill code generation and register assignment: Once the maximum register pressure in the program is lowered to the number of available registers, registers can be assigned *optimally* using a linear-time algorithm that provably does not cause further spill code.

In this paper, we propose a program transformation that limits the maximum register pressure of an SSA-form program by inserting efficient spill code. Its main features are:

- It extends the well-known Min algorithm [9], which has proven to be very successful [10] in straight-line code register allocation, to control-flow graphs.
- Our algorithm retains the SSA form. Hence, it is ideal for the use in SSA-based register allocation.
- It is effective. Our algorithm meets the requirements described above. It is sensitive to the structure of the program by splitting live-ranges around loops. Our experiments show a reduction of executed reload instructions by 54.5% (executed spills by 61.5%) compared to one of the most sophisticated live-range splitting algorithms available [5].
- It is efficient. Our algorithm consists of two passes: An enhanced liveness analysis and a single sweep over the program. Required analysis information is a loop tree and def-use chains; both are usually available during the backend phase of a modern compiler. Furthermore, we do not build large or complex data structures (such as an interference graph).

Structure of this paper. The next section recaps the Min algorithm and its use in register allocation of straight-line code. In Section 3 we discuss, by way of examples, how the Min algorithm can be generalized to code with branches. Section 4 presents our algorithm in detail. We evaluate our algorithm experimentally in Section 5. The last two sections discuss related work and conclude.

2 The Min Algorithm and Local Register Allocation

The original Min algorithm was developed as a page replacement strategy in operating systems. Its basic idea is: If a memory page has to be removed to

Algorithm 1 The Min algorithm

```
def limit(W, S, insn, m):
    sort(W, insn)
    for v $\in$ W[m:$-$1]:
        if v $\notin$ S $\wedge$ nextUse(insn,v) $\neq \infty$:
            add a spill for v before insn
        S $\leftarrow$ S $\setminus$ {v}
    W $\leftarrow$ W[0:m]
```

```
def minAlgorithm(block, W, S):
    for insn $\in$ block.insnuctions:
        R $\leftarrow$ insn.uses $\setminus$ W
        for use $\in$ R:
            W $\leftarrow$ W $\cup$ {use}
            S $\leftarrow$ S $\cup$ {use}
        limit(W, S, insn, k)
        limit(W, S, insn.next,k$-$|insn.defs|)
        W $\leftarrow$ W $\cup$ {insn.defs}
        add reloads for vars in R
        in front of insn
```

swap in a new one, remove the page whose next use is farthest in the future. If the Min algorithm knew the future and thus always knew whose page's use is farthest away, it would perform the minimum number of replacements (see van Roy [11] for a proof). The Min algorithm has often been applied to straight-line register allocation and has shown to be very effective [10] in this setting.

Let us now review the Min algorithm in the setting of register allocation for a single basic block. For the rest of this paper, we assume that the program is in SSA form. The *next-use distance* of a variable v at an instruction I is the number of instructions between I and the next use of v in the block. Especially I itself can be the next user, leading to distance 0. If there is no further use in the block, the distance is ∞.

The content of the register file is reflected in a set W containing the variables currently available in a register. Initially W is empty. We traverse the basic block from entry to exit, updating W according to the effects of each instruction. Assuming a conventional load/store architecture, each instruction

$$I : (\underbrace{y_1, \dots, y_m}_{\text{defs}_I}) \leftarrow \tau(\underbrace{x_1, \dots, x_n}_{\text{uses}_I})$$

requires that its operands x_i are available in registers and writes its results y_i to registers. At each program point, W must not contain more than k (the number of available registers) variables. Hence, the effects of an instruction I on W are as follows:

1. All variables in $\text{uses}_I \setminus W$ have to be reloaded in front of I. Thus, they have to be added to W. If W has not enough room, $|W| + |\text{uses}_I \setminus W| - k$ variables in W have to be spilled.
2. None of the variables in defs_I can be in W directly in front of I since all of these variables are dead there. Hence, we need $|\text{defs}_I|$ free registers. If there is not enough room in W to hold $|\text{defs}_I|$ variables, $|W| + |\text{defs}_I| - k$ variables have to evicted from W.

Algorithm 1 shows the Min algorithm for straight-line code. minAlgorithm performs the steps 1 and 2 on each instruction of the block. limit takes the set W, sorts it according to the next-use distance from instr, and evicts all variables but the first m. Note carefully that for an instruction I we call limit twice. The first time, to make room for the operands and the second time to provide registers for the result variables of I. In the latter case, the next-use distance is measured from the instruction *behind* I because the uses of I do no longer matter when I writes its results.

Furthermore, minAlgorithm takes a *subset* S of W. Because the program is in SSA form, each variable has only one definition and thus needs to be spilled at most once. If a variable is evicted multiple times, a spill has to be placed only at the place of the first eviction. The set S records all variables in W for which a spill has already been inserted. Updating S is easy: Whenever a variable is reloaded, it must have been spilled before. Hence, we add it to S. When evicting variables from W, we only create spills for variables not in S whose next use is not ∞.

3 Overview

Guo et al. [10] empirically showed that the Min algorithm gives very good results on straight-line code. The intuitive explanation for this is that evicting the variable with the furthest next use frees a register for the longest possible time. In this paper however, we are not only interested in spill-code generation for single basic blocks but for a whole control-flow graph (CFG).

To this end, let us first investigate how the straight-line version would perform on single *execution traces* of a CFG. Consider the CFG in Figure 3. The ↯n sign denotes regions with high register pressure where at most n variables can live through in registers. Consider the following the execution traces of the CFG:

1. S, L, E: The register pressure is critical at the end of S and either x or y have to be evicted from W. The farthest next use is the one of y in block E. Thus, y is evicted.
2. S, B, H, H, H, E: As in the previous example, y is evicted in S. The register pressure in B is so high that x has to be evicted from W. Thus, x is reloaded upon its first use in the first execution of H. The register pressure in H is uncritical. Hence, x can remain in a register for the second and third execution of the loop body H.

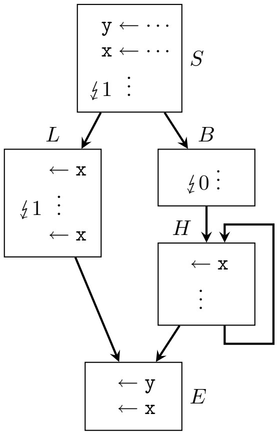


Fig. 1: Example CFG

Let us go back to the static setting where we consider a CFG, not single traces. To develop an *effective* spilling algorithm, we have to place the spill and reload instructions in the CFG such that the straight-line Min algorithm is emulated as good as possible *for every possible* trace. In principle, this can be achieved by applying the Min algorithm to each block separately. However, reviewing the trace examples above, we have to consider each block in its context:

- When reaching the end of S, we have to decide whether x or y is evicted. In a naive per-block application of Algorithm 1 both, x and y have a next-use distance ∞ at the end of S because they have no further use in S. However, to choose y, as in example 1 above, we need a CFG-global next-use view: The earliest possible next use of x is closer than the one of y (blocks L and B in contrast to E).

- Algorithm 1 assumes that the register set W is empty at the entry of the block. This implies that every live-in variable is reloaded on its first use in the block.

  Considering both trace examples above, we observe that x is in registers at the end of L and H. Thus, it is also in registers at the entry of E. Consequently, applying the Min algorithm to all predecessors of a block before applying it to the block itself, makes the state of the register set at their exits available. The initialization of W can then be chosen accordingly.
- Consider the second trace above: x was spilled in B. In the first execution of H it was reloaded and used from a register in the following two executions. However, we cannot have one version of H with the reload and one without! Placing a reload in H is not a good solution since this reload would be executed needlessly in every iteration except for the first. It is much better to place the reload in front of the loop at the exit of B. Therefore, we need to know that H is a loop head and initialize W with the variables used next in the loop. This has the effect of hoisting the reloads out of the loop.

4 A Min Algorithm for CFGs

The following outline summarizes the presented algorithm:

1. Compute liveness and global next uses (Section 4.1):

   We present a modification to the standard data-flow formulation of liveness analysis to compute CFG-global next-use values for each variable.
2. For each block B in *reverse post order* of the CFG:
   (a) Determine initialization W_B^{entry} of register set (Section 4.2):

   We compute a set of variables, which we assume to be in registers at the entry of the block.

   (b) Insert coupling code at the block entry (Section 4.3):

   Depending on the state of the register file at the exit of the predecessor of B, we add spill and/or reload code on B's incoming control-flow edges to ensure that all variables in W_B^{entry} are indeed in registers at the entry of B.

   (c) Perform Min algorithm on B (Section 2)
3. Reconstruct SSA (Section 4.4):

   Inserting reloads for a variable creates multiple definitions for this variable. This clearly violates the static single assignment property. We describe, how SSA is reconstructed after spill/reload code insertion.

4.1 Global Next-Use Distances

The next-use distances beyond a single block are computed by augmenting a standard liveness analysis. Instead of computing live-in and live-out sets, we compute maps that associate variables with next-use distance; instead of unifying live sets at control-flow splits, we merge the maps by taking the minimum next-use distance per variable. This modelling actually entails liveness information: If the next-use distance of a variable is smaller than ∞, the variable is live, otherwise it is dead.

To reflect the dynamic behavior of the program, each control-flow edge (P, Q) is assigned a *length* $\ell_{P,Q}$. Edges leading out of loops are assigned a very high length M[3]; all other edges have length 0. When computing the next-use distances, the length of the edges are added to the next-use distances of the variables that live over the edge. The effect is that the distances of uses behind loops are larger than the distances of all uses inside the loop.

Formalism. Let us now briefly discuss the next-use analysis formally. For an introduction to data-flow analysis, we refer to Nielson et al. [12]. Our domain is the set

$$\mathbb{D} = \mathbf{Var} \to \mathbb{N} \cup \{\infty\}$$

of maps from variables to natural numbers (augmented by a value ∞). The join of two maps $a, b \in \mathbb{D}$ is defined by taking the minimum of the variables' next-use distances:

$$a \sqcup b := \lambda v.\, \min\{a(v), b(v)\}$$

$\langle \mathbb{D}, \sqcup \rangle$ is a *join semi-lattice* that satisfies the *ascending chain condition*[4].

The transfer function f_B for a block B takes the next-use distances at the exit of the block and computes the next-use distance at the entry of the block. (Just like liveness analysis takes the set of live variables at the exit and computes the set of live variables at the entry.) There are two cases:

1. If a variable v has at least one use in B that is not preceded by the definition of v, the distance to v's next use is the length of the block ℓ_B[5] plus the distance $\nu_B(v)$ from the entry of the block to the first use of v in B that is not preceded by the definition of v.
2. If v has no such use in B, the distance from B's entry to v's next use is the sum of ℓ_B, the length $|B|$ of B, and the distance from B's exit to the next use of v.

[3] M has to be larger than the number of instructions on the longest path through the loop. Hence, in practice, a value like 100000 works nicely.

[4] The proof is straightforward and is omitted here for the sake of brevity.

[5] Using the standard formalization of data-flow analyses (see also [12]), we cannot incorporate information on control-flow edges in the transfer function. As we assume critical edges to be split, the length of an edge can be uniquely attributed to some block.

This yields

$$f_B(a) = \lambda v.\ \ell_B + \begin{cases} \nu_B(v) & \text{if } \nu_B(v) \neq \infty \\ |B| + a(v) & \text{otherwise} \end{cases}$$

Finally, the initial value $\imath$ of each block maps each variable to the distance of its first local use:

$$\imath_B := \lambda v.\, \infty$$

We chose $\sqcup$ as the minimum in order to ensure the convergence of the data-flow analysis. A more appropriate choice for the spilling problem would be to compute the next-use distance as a weighted sum of the successor's distances, using execution frequencies as weights. However this would violate the laws for a proper lattice join operation and the theoretical framework of data-flow analysis could no longer be used soundly. The practitioner however may just iterate the analysis long enough to obtain sufficiently precise information.

4.2 Initialization of the Register Set

For each block B we compute the set W_B^{entry} of variables, which we require to be in registers at the entry of B. As discussed in Section 3, the choice of W_B^{entry} is essential for the effectiveness of the algorithm. According to the examples of Section 3, W^{entry} is computed differently for loop headers and normal blocks.

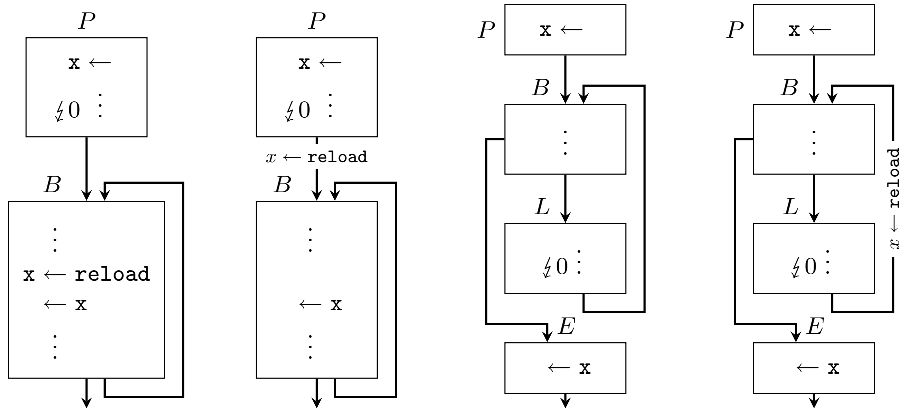
Normal blocks. Let B be a non-loop-header block. As we process the nodes in reverse postorder, every predecessor of B has already been processed. Let W_P^{exit} denote the set W after the Min algorithm has been applied to block P. Furthermore, let

$$\mathit{all}_B = \bigcap_{P \in \mathit{pred}(B)} W_P^{\text{exit}} \qquad \mathit{some}_B = \bigcup_{P \in \mathit{pred}(B)} W_P^{\text{exit}}$$

The variables in all_B are in registers on every incoming edge at B. Thus, we can assume them to be in registers at the entry of B. The variables in $\mathit{some}_B \setminus \mathit{all}_B$ are available in registers at some of the predecessors. They are sorted according to their next-use distance and put into the remaining slots of W_B^{entry}. initUsual in Algorithm 2 shows the pseudocode for computing W^{entry} of normal blocks.

Loop headers. Now, let B be a loop header. Consider some variable v that is live-in at and used in B. Furthermore, assume that v has been spilled in some block P outside B's loop, like variable x in Figure 2a. In this example, $\mathtt{x} \notin W_P^{\text{exit}}$. If we determined W_B^{entry} by looking at the contents of W_P^{exit} (as we would do for normal blocks), x would not be contained in W_B^{entry}. This would cause the insertion of a reload of x inside the loop; something that has to be avoided at all costs. It is much better, to allocate x to W_B^{entry} so that the reload is put on the edge from P to B, as shown in Figure 2b.

But there are also variables that should not be put in W_B^{entry}: Consider Figure 2c. Variable x lives throughout the loop but is not used inside. Inside the loop, the register pressure is critical such that x cannot "survive" the loop in



(a) Reload in the loop: bad

(b) Load before the loop: good

(c) Variable living throughout the loop

(d) Needless reloads for live-through variable

Fig. 2: Reloads in loops

a register. By allocating x to W_B^{entry} we definitely require x to be in a register at the entry of B. Hence, a reload has to be put on the loop backedge (see Figure 2d). When determining W_S^{entry}, we already processed H. As the register pressure in H is uncritical, x is in W_S^{exit}. Thus, x is also included in W_S^{entry} and x can be used from a register. This reload is executed in every loop iteration. However, only in the last iteration that exits the loop, the reloaded value will be actually used (in block E). In this case, the reload should clearly be put at the entry of E.

So which variables should be put into W_B^{entry}? Following the discussion above and in Section 3, we ignore the predecessors of B. Let I_B be the set of variables that are live-in at B as well as defined by ϕ-functions in B. The candidates for W_B^{entry} are all variables in I_B used within B's loop $\mathcal{L}$. The variables are sorted according to their next-use distance and the first k are allocated to W_B^{entry}.

If there is room left in W_B^{entry}, we consider the set $T_B \subseteq I_B$ of variables that are not used in $\mathcal{L}$. The example Figure 2c shows that such a variable should only be assigned to W_B^{entry} if we are sure that the variable will survive the loop without being evicted. We determine how many variables of T_B can be kept in registers throughout $\mathcal{L}$ heuristically: Consider the *maximum register pressure* $p_{\mathcal{L}}$ of the loop. The difference $p_{\mathcal{L}} - |T_B| =: t$ gives an estimate on the register pressure caused by variables used *inside* the loop[6]. If t is smaller than k, we conclude that $k - t$ variables can survive the loop in registers. In this case, the remaining slots in W_B^{entry} are filled with at most $k - t$ variables from T_B.

[6] If $\mathcal{L}$ is a single-exit loop or T_B only consists of variables defined outside $\mathcal{L}$ this estimation is exact, else it might be an under-approximation.

The maximum loop register pressure $p_{\mathcal{L}}$ can easily be computed during the liveness analysis presented in the last subsection. As we have to traverse the instructions of each block anyways, we can also keep track of the maximal register pressure inside each block. $p_{\mathcal{L}}$ is then simply computed by taking the maximum over the maximum register pressures of the blocks of $\mathcal{L}$.

Algorithm 2 Initialization of W

```
def initLoopHeader(block):
    entry ← block.firstInstruction
    loop ← loopOf(block)
    alive ← block.phis ∪ block.liveIn
    cand ← usedInLoop(loop, alive)
    liveThrough ← alive \ cand
    if |cand| < k:
        freeLoop ← k − loop.maxPressure
            + |liveThrough|
        sort(liveThrough, entry)
        add ← liveThrough[0:freeLoop]
    else:
        sort(cand, entry)
        cand ← cand[0:k]
        add ← ∅
    return cand ∪ add

def initUsual(block):
    freq ← map()
    take ← ∅
    cand ← ∅
    for pred in block.preds:
        for var in pred.Wend:
            freq[var] ← freq[var] + 1
            cand ← cand ∪ {var}
        if freq[var] = |block.preds|:
            cand ← cand \ {var}
            take ← take ∪ {var}
    entry ← block.firstInstruction
    sort(cand, entry)
    return take ∪ cand[0:k−|take|]
```

4.3 Connecting a block to its predecessors

When applying the Min algorithm to a block B, we need to insert coupling code at B's borders. For example, a variable that we require to be in W_B^{entry} might not be in W_P^{exit} of some predecessor P. For this variable, a reload on the way from P to B has to be inserted.

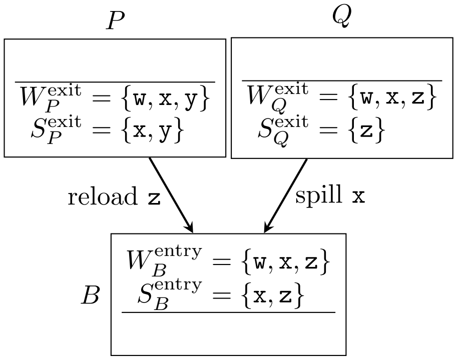

P
Q
$W_P^{\text{exit}} = \{\mathtt{w},\mathtt{x},\mathtt{y}\}$
$S_P^{\text{exit}} = \{\mathtt{x},\mathtt{y}\}$
$W_Q^{\text{exit}} = \{\mathtt{w},\mathtt{x},\mathtt{z}\}$
$S_Q^{\text{exit}} = \{\mathtt{z}\}$
reload z
spill x
B
$W_B^{\text{entry}} = \{\mathtt{w},\mathtt{x},\mathtt{z}\}$
$S_B^{\text{entry}} = \{\mathtt{x},\mathtt{z}\}$

Fig. 3: Coupling code at block borders

Additionally, we have to provide a sensible initialization S_B^{entry} for the set S that records which variables in W have already been spilled. The invariant for S is: v is in S at instruction I iff v was spilled on all paths from the CFG root to I. To avoid redundant spills, S_B^{entry} is set to all variables in W_B^{entry} that are spilled on some path to B:

$$S_B^{\text{entry}} := \left[\bigcup_{P \in \text{pred}(B)} S_P^{\text{exit}} \right] \cap W_B^{\text{entry}}$$

The coupling code for a predecessor P of B has to be inserted as follows:

- All variables in $W_B^{\text{entry}} \setminus W_P^{\text{exit}}$ need to be reloaded on the edge from P to B.
- All variables in $(S_B^{\text{entry}} \setminus S_P^{\text{exit}}) \cap W_P^{\text{exit}}$ need to be spilled from P to B.

The example in Figure 4.3 shows a block B, its predecessors P and Q which have already been processed, and the inserted code for $W_B^{\text{entry}} = \{\mathtt{w}, \mathtt{x}, \mathtt{z}\}$.

In the preceding paragraphs of this subsection, we assumed that all predecessors have already been processed. Let us now consider a loop header B and a predecessor P of B that has not yet been processed. Thus, S_P^{exit} and W_P^{exit} are not available. When processing B, we simply ignore P and add the corresponding spills and reloads as soon as P has been processed.

4.4 Retaining the SSA Form

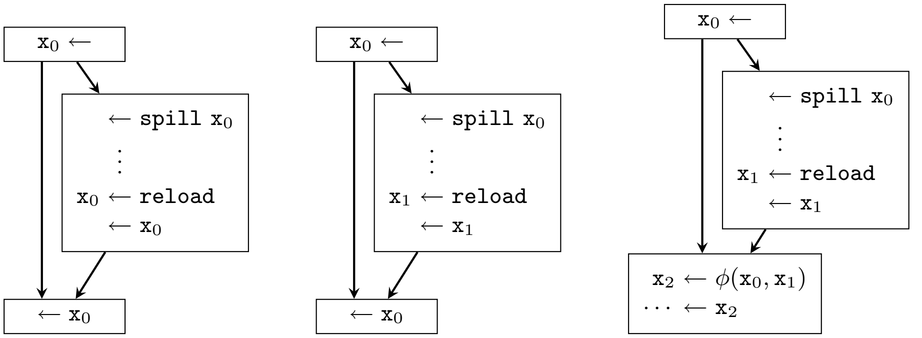
In this section, we briefly discuss the interdependency of spill-code generation and the SSA form. Due to space limitations, we only give a brief overview; a more in-depth discussion can be found in [13]. Let us first consider the requirements on the input program and then sketch how SSA is retained during the algorithm.

Requirements on the input program. In a non-SSA-form program, each variable is assigned one spill slot (i.e. the memory location where the spilled values of that variables are written to and read from). In SSA form, we *need* to assign all variables of a ϕ-congruence class (cf. Sreedhar et al. [14]) the same spill slot. Else, spilled ϕ-functions result in memory copy instructions. To this end, we demand that each ϕ-congruence class is free of interference, i.e. the CFG is in *conventional* SSA form [14].

Producing SSA output. Inserting a reload for an SSA variable creates a second definition of that variable. Consider the example in Figure 4a. There are two definitions for $\mathtt{x}_0$; the original one and the reload. Creating a new variable $\mathtt{x}_1$ for the reload and renaming the following use, re-establishes the single assignment property. However, the use of $\mathtt{x}_0$ at the lower block is then no longer correct: Coming from the left block, $\mathtt{x}_0$ holds the right value, while coming from the right, the variable to use is $\mathtt{x}_1$. Hence, we have to place a ϕ-function in the lower block that selects over $\mathtt{x}_0$ and $\mathtt{x}_1$ and defines a new variable $\mathtt{x}_2$. Thus, spilling a variable can cause new ϕ-functions to be inserted.

All in all, we need to record all inserted reload operations per variable and *reconstruct* SSA for those variables. This can be achieved with an efficient algorithm by Sastry and Ju [15].

In short, the algorithm takes the original definition of a variable v, a set of new definitions of v (in our case the inserted reloads) and the list of uses of v. Then, for each use, the dominance tree is walked upwards. The first found definition is responsible for that use. When passing an iterated dominance frontier (see Cytron et al. [16]), the algorithm lazily inserts ϕ-functions and wires their operands to suitable definitions. As a side effect, dead definitions are never reached by this search process and can thus be eliminated.



(a) Original program

(b) SSA reconstructed

(c) Second definition of $\mathtt{x}_0$, ϕ-function inserted

Fig. 4: Adding a reload causes a ϕ-function to be created

5 Evaluation

Our experimental evaluation consists of two parts: First, we briefly discuss the compile-time behavior of our algorithm. Second, we assess the quality of the produced code.

Setup. We implemented the presented spilling algorithm in the libFirm [17] compiler. This compiler produces code for the x86 architecture and features a completely SSA-based register allocator as presented in [13]. All measurements were conducted on the integer part CINT2000 of the CPU2000 benchmark [18]. The program 252.eon was not compiled because the used compiler is not able to process C++. The compile-time measurements were taken on a Core 2 Duo 2GHz PC with 2GB RAM running Linux with kernel version 2.6.22. All presented data only considers the 7 general-purpose registers of the x86. The low number of available registers emphasizes the importance of spill-code generation.

5.1 Runtime of the Algorithm

Figure 5 shows the time spent in the spilling phase in relation to the size of the compiled function. The granularity limit of the time values is 1ms resulting in discrete looking values. The linear regression is also drawn in the diagram to indicate that the time spent on spilling scales roughly linear with the number of instructions. The average throughput is 430 instructions per millisecond.

5.2 Code Quality

We compare our algorithm to the spill code generated by a standard Chaitin/Briggs [1, 2] allocator (IFG) and the Wimmer & Mössenböck [5] variant of

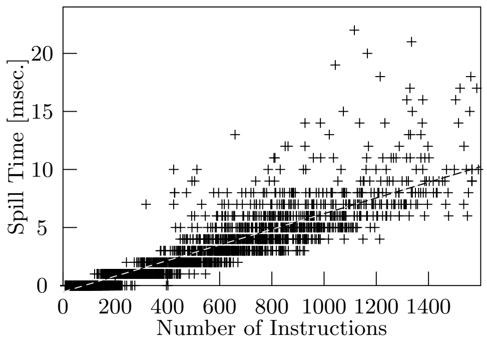


Fig. 5: Time spent on spilling

Bench.	IFG Spilling		Linear Scan	
	Spills	Rel.	Spills	Rel.
gzip	2.39	3.39	1.63	2.91
vpr	1.78	2.83	0.79	1.34
gcc	2.06	2.54	1.69	2.86
mcf	2.01	3.29	15.62	5.87
crafty	1.43	2.04	1.23	1.61
parser	1.54	1.92	1.17	1.38
perlbmk	0.93	1.28	0.77	1.22
gap	1.71	2.78	1.04	1.79
vortex	1.58	2.15	1.62	1.93
bzip2	2.01	2.21	1.83	2.05
twolf	1.53	1.85	1.21	1.25
Average	1.72	2.39	2.60	2.20

Table 6: Ratio of spill and reload instructions compared to SSA spilling

Benchmark	SSA Spilling (this paper)			IFG Spilling [1, 2]			Linear Scan [5, 19]		
	Insns.	Spills(%)	Rel.(%)	Insns.	Spills(%)	Rel.(%)	Insns.	Spills(%)	Rel.(%)
164.gzip	329	4.2	5.9	374	9.0	17.5	347	6.6	16.2
175.vpr	196	5.2	9.4	223	8.2	23.5	176	4.6	14.1
176.gcc	158	3.9	5.5	171	7.4	12.9	172	6.1	14.5
181.mcf	51	0.5	3.1	53	1.0	9.7	60	7.0	15.3
186.crafty	208	6.7	7.4	219	9.1	14.2	198	8.7	12.5
197.parser	322	4.6	7.6	339	6.8	13.9	304	5.7	11.1
253.perlbmk	397	11.1	9.8	392	10.5	12.7	357	9.6	13.3
254.gap	252	3.7	4.2	263	6.1	11.1	215	4.5	8.7
255.vortex	321	5.0	5.3	341	7.4	10.6	340	7.6	9.6
256.bzip2	287	4.2	6.8	313	7.7	13.8	320	6.9	12.5
300.twolf	297	3.8	5.7	306	5.7	10.2	293	4.7	7.2
Average	256	4.82	6.41	272	7.16	13.65	253	6.53	12.27

Table 7: Executed instructions (billions), percentage of spills and reloads

linear-scan implemented in LLVM version 2.3 [19]. Instead of re-implementing the linear-scan allocator in our framework, we decided to directly compare to the fine-tuned implementation within LLVM. Of course, comparing two different compilers is always problematic. However, the backend passes in LLVM and libFirm are quite similar and LLVM's middle-end is usually more powerful than libFirm's. We verified this by manual inspection of the code generated for some important inner loops.

To assess the quality of the produced code as general as possible, we count the executed spill and reload instructions of the benchmark programs. We deliberately do not compare runtimes of the produced code as they are biased by all sorts of microarchitectural effects like caching, out-of-order execution and the like. Those influences can greatly vary among different processor architectures and even different implementations of the same architecture. To count the reload and spill instructions, we modified the code generators of libFirm and LLVM to add a special NOP instructions in front of each spill and reload. We then executed all benchmarks with the Valgrind [20] machine-code instrumentation tool and used a home-made plugin to count the marked spill and reload instructions.

The dynamic instruction counts (in billions) are shown in Table 7 along with the percentage of executed spill and reload instructions. Our algorithm is labeled "SSA Spilling". "IFG Spilling" is the graph-coloring spilling and "Linear Scan" shows the results of the code generated by LLVM. Table 7 shows that 3.5% to 20% of the executed instructions are spill and reload instructions. Hence, the spilling heuristic has a significant impact on code quality.

Table 6 shows the ratio of executed spill instructions compared to the results of our algorithm. Our algorithm produces better code than the IFG and the linear-scan algorithm in almost all of the cases. We constantly produce less executed reloads, sometimes even less than the half. On average, the linear-scan (IFG) approach performs 2.60 (1.72) times as many spills and 2.20 (2.39) times as many reloads.

6 Related Work

This paper's approach of separating register allocation from register assignment is in line with Proebsting and Fischer [21]. Their algorithm also globally allocates on top of local information. However, Proebsting and Fischer calculate for each use a probability that this use can be made from a register. This probability needs to be propagated through *all possible* paths to that use. Finally, whenever their algorithm decides to allocate a global variable to a register, the probabilities of all remaining variables need to be updated, which renders the algorithm quadratic in the number of (global) variables where we visit each variable only once.

Morgan [22] proposes to handle some of the spilling before main register allocation in order to improve spill code placement. He describes an algorithm that identifies variables that live throughout a loop but are not used inside it. These variables are spilled in front of and reloaded behind the loop. This improves the situation for a common class of variables but still leaves most spilling decisions to the register allocator. Our algorithm yields a program for which an SSA-based register allocator does not need to insert additional spill code. In our technique the complex analysis of Morgan is replaced by the CFG-global modelling of next-use distances, especially the length assignment loop-exiting edges.

Guo et. al. [10] apply the Min algorithm as described in Section 2 to long basic blocks. They are able to considerably improve the runtime of their benchmarks compared to a standard MIPSPro or GCC compiler. However, their improvements are mostly visible in basic blocks with long live ranges resulting from extensive loop unrolling. Their good results lead us to investigate the applicability of the Min algorithm in global register allocation.

Farach and Liberatore [23] prove that the spill-problem is NP-hard for basic blocks. They also prove that the Min algorithm algorithm gives a $2C$-approximation to the local spilling problem. This supports the good experimental results by Guo et al.

The main motivation for this work were recent results in SSA-based register allocation (see [6–8]). If the maximum register pressure in the program is lowered to the number of available registers, a linear-time algorithm can assign the registers *optimally* without adding further spill code. Up to now, a good and fast heuristic to lower the register pressure was missing in this context.

Wimmer and Mössenböck [5] perform live-range splitting while allocating registers in a linear-scan allocator. In the tradition of linear-scan allocators, the CFG is flattend to linear code. In this setting, a list of use-points is constructed per variable. For points with high register pressure the variables with the furthest next-use (in the flattened code!) are spilled first. They present a technique for moving the split positions for spills and reloads to earlier points to move spills and reloads in front of loops. The linear order of the basic blocks however is too restrictive: At control-flow splits, blocks are forced into an arbitrary order which often unnecessarily prohibits hoisting spills or reloads. The original linear-scan allocator (Poletto & Sarkar [4]) introduced CFG flattening to deliver the best possible compile-time performance by avoiding any expensive analysis and additional passes.

To improve code quality, several extensions like Wimmer and Mössenböck [5], Traub et al.[3], and Sarkar & Barik [24] were developed over the years. They successively left the rapid linear-scan paradigm by adding liveness analysis, various other analyses, and fix-up passes. Many of these extensions implicitly rely on the CFG although all algorithms still use the flattened view. We demonstrated that flattening the CFG is not necessary for efficient high-quality spill-code generation.

7 Conclusions

We presented an efficient and effective approach to spill-code generation and live-range splitting. Unlike most existing techniques, our approach is not entangled with a register allocator: It is a program transformation that limits the register pressure of an arbitrary SSA-form program to a given number. While this is useful as a pre-spill phase in any compiler, our technique is predestined for the use in SSA-form register allocation: If a SSA-form program has a maximum register pressure of k, a SSA-based allocator can find an optimal register allocation without introducing further spills in linear time.

Our algorithm is most sensitive to the structure of the program: It carefully splits live ranges around loops to avoid reload instructions in loops where possible. Our evaluation on the CINT2000 benchmark suite shows that our approach reduced the number of executed reload instructions by 54.5% compared to the state-of-the-art linear-scan allocator and by 58.2% compared to a standard graph-coloring allocator. At the same time, the compile-time overhead is competitive with popular linear-scan allocators: We perform liveness analysis and one sweep over the program's CFG.

8 Acknowledgements

We thank Michael Beck, Alain Darte, Gerhard Goos, Daniel Grund, Christoph Mallon, Fabrice Rastello, Jan Reineke, and Christian Würdig for several insightful discussions. Furthermore, we thank the anonymous reviewers for their valuable comments.

References

1. Chaitin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., Hopkins, M.E., Markstein, P.W.: Register allocation via graph coloring. Journal of Computer Languages **6** (1981) 45–57
2. Briggs, P., Cooper, K.D., Torczon, L.: Improvements to graph coloring register allocation. ACM Trans. Program. Lang. Syst. **16**(3) (1994) 428–455
3. Traub, O., Holloway, G., Smith, M.D.: Quality and speed in linear-scan register allocation. In: PLDI '98: Proceedings of the Conference on Programming Language Design and Implementation, New York, NY, USA, ACM Press (1998) 142–151
4. Poletto, M., Sarkar, V.: Linear scan register allocation. ACM Trans. Program. Lang. Syst. **21**(5) (1999) 895–913
5. Wimmer, C., Mössenböck, H.: Optimized interval splitting in a linear scan register allocator. In: VEE '05: Proceedings of the 1st international conference on Virtual execution environments, New York, NY, USA, ACM (2005) 132–141
6. Bouchez, F., Darte, A., Guillon, C., Rastello, F.: Register Allocation: What does the NP-Completeness Proof of Chaitin et al. really prove? or revisiting Register Allocation: Why and How? In: 19th International Workshop on Languages and Compilers for Parallel Computing (LCPC'06), New Orleans, USA (Nov 2006)
7. Brisk, P., Dabiri, F., Jafari, R., Sarrafzadeh, M.: Optimal Register Sharing for High-Level Synthesis of SSA Form Programs. IEEE Trans. on CAD of Integrated Circuits and Systems **25**(5) (2006) 772–779
8. Hack, S., Grund, D., Goos, G.: Register allocation for programs in SSA-form. In: Compiler Construction. Volume 3923., Springer (March 2006)
9. Belady, L.A.: A study of replacement algorithms for a virtual-storage computer. IBM Systems Journal **5**(2) (1966) 78–101
10. Guo, J., Garzarn, M.J., Padua, D.: The power of beladys algorithm in register allocation for long basic blocks. In: The 16th International Workshop on Languages and Compilers for Parallel Computing. (2003)
11. Roy, B.V.: A short proof of optimality for the min cache replacement algorithm. Inf. Process. Lett. **102**(2-3) (2007) 72–73
12. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer (1999)
13. Hack, S.: Register Allocation for Programs in SSA Form. PhD thesis, Universität Karlsruhe (TH) (October 2007)
14. Sreedhar, V.C., Ju, R.D.C., Gillies, D.M., Santhanam, V.: Translating out of static single assignment form. In: SAS '99, London, UK, Springer-Verlag (1999) 194–210
15. Sastry, A.V.S., Ju, R.D.C.: A new algorithm for scalar register promotion based on SSA form. In: PLDI '98: Proceedings of the conference on Programming language design and implementation, New York, NY, USA, ACM (1998) 15–25
16. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadek, F.K.: Efficiently computing static single assignment form and the control dependence graph. Transactions on Programming Languages and Systems **13**(4) (October 1991) 451–490
17. Firm: The libFirm Compiler. http://www.libfirm.org
18. Standard Performance Evaluation Corporation: SPEC CPU2000 V1.3. http://www.spec.org/cpu2000/
19. LLVM: The LLVM Compiler Infrastructure. http://www.llvm.org
20. Valgrind: Valgrind Instrumentation Framework for Building Dynamic Analysis Tools. http://www.valgrind.org
21. Proebsting, T.A., Fischer, C.N.: Probabilistic register allocation. SIGPLAN Not. **27**(7) (1992) 300–310
22. Morgan, R.: Building an Optimizing Compiler. Digital Press (1998)
23. Farach, M., Liberatore, V.: On local register allocation. In: SODA '98: Proceedings of the ninth annual symposium on Discrete algorithms, Philadelphia, PA, USA, Society for Industrial and Applied Mathematics (1998) 564–573
24. Sarkar, V., Barik, R.: Extended linear scan: An alternate foundation for global register allocation. In: CC. (2007) 141–155