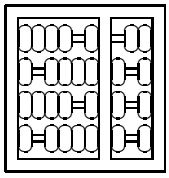

INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Hole Allocation in Spill Code Generation

Wesley Attrot *Guido Araújo*

Technical Report - IC-08-27 - Relatório Técnico

September - 2008 - Setembro

Hole Allocation in Spill Code Generation

Wesley Attrot* Unicamp

Guido Araújo Unicamp

Abstract

Allocating variables to registers is a central task in program code generation. Though considered a well-studied and thoroughly scrutinized problem, register allocation sometimes reveals a new interesting facet. Spill code generation is part of any register allocation algorithm and its efficacy can have a direct impact in the final program performance. In this paper we present a dynamic programming algorithm to generate spill code, which tries to simultaneously reallocate spilled variables into two type of holes left by the register allocator: intervals between live ranges (called dead-holes), and low-density live ranges (called live-holes). As a post-pass optimization, this technique can be used by any register allocation algorithm. Experimental results reveal that a considerable reduction in memory traffic can be achieved.

1 Introduction

Memory access represents a major performance bottleneck in most computer architectures. Even though the combination of large caches and memory transaction buffers have been able to reduce memory traffic [16], the memory wall still poses a challenging problem to contemporary computer design.

Register allocation algorithms have been a central optimization in any modern optimizing compiler [14, 23], as a way to reduce memory traffic. Register allocation must handle situations where the number of live variables is larger than the number of processor registers. In such cases, it must decide which variables to allocate onto memory and also it should generate instructions to access such variables, a process called spill code generation. An efficient spill code generation algorithm should be able to generate as few memory accesses as possible [15].

Typically, a register allocation algorithm is based on the construction of an interference graph [11] or chordal graph [20], using the interference of variable liveness as a starting point. Spill code generation occurs when the register allocation process fails to assign a register to each variable in the interference point under analysis.

After the register allocation algorithm ends its work, a close inspection of the resulting register assignment will reveal a few *holes* among the registers live ranges. These are regions where a particular register was not assigned to any variable in the program. In Figure 1 we

*This work is sponsored by CNPq

have an example of this situation. If instruction (2) does not make a reference to register $R1$, then register $R1$ is not live during this instruction. If the amount of instructions between the end of a live range and the begin of another that shares the same register is of several instructions, we have what we call a *dead hole*, since the register is dead in between these two live ranges.

(1) := **R1**
(2) instr_a
(3) **R1** := ...

Figure 1: Dead Hole

Similar holes can also occur even though a variable has been assigned to a register. For example, consider Figure 2 where a live range has been assigned to register $R1$. It has a definition in instruction (1) and an use in instruction (7). If the instructions from (2) to (6) do not make any reference to register R1, so we have a *hole* where register $R1$ is not referenced at all. This hole is not dead, because $R1$ contains a live value that is not used for 5 instructions. We call such hole a **live hole**, because the value in the register is still live in the *hole*, but the value is not used.

(1) **R1** := ...
(2) instr_a
(3) instr_b
(4) instr_c
(5) instr_d
(6) instr_e
(7) ... := **R1**

Figure 2: Live Hole

Traditional spill code techniques insert a **store** after every definition and a **load** before every use in the spilled live range [8]. This way, many load/store instructions are generated. Some algorithms [5, 16] try to minimize the spill code using the *dead holes* that appear after register allocation. Other technique [9] use the *live holes*, but in a very restricted way. In this paper we propose a technique that tries to maximize the use of *dead* and *dead* holes simultaneously, splitting the spilled live range among these holes and inserting instructions to move the value of the variable among the holes while trying to minimize the number of required move instructions.

For example, consider the situation in Figure 3 where two live ranges have registers and a third live range $l3$ is going to be spilled. In this figure, each black bullet is a reference to a register, i.e., a read or write. Traditional spill code generation would break the spilled live range into several load/store instructions. If the live range $l3$ contains has many instruction, many spill code instructions will be generated.

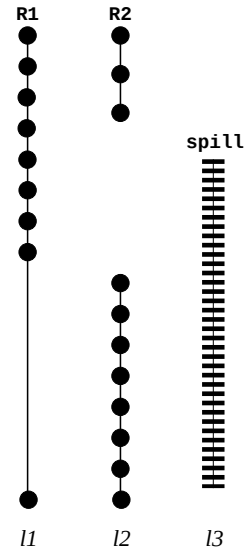

Figure 3: Live ranges for allocation

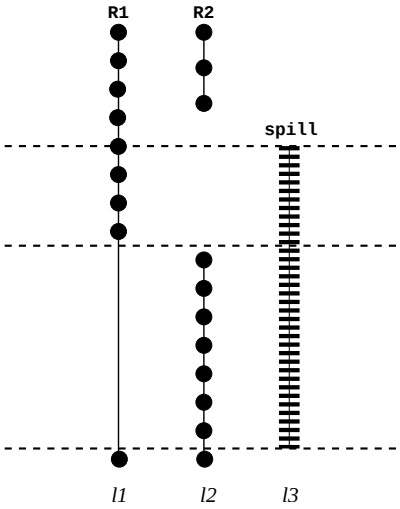

Figure 4: Fitting $l3$ into live/dead holes

The live range $l2$ has a dead hole, but it only covers a portion of $l3$. Notice that, after several references to register R1, live range $l1$, has a live hole, where no access to R1 is performed. Thus, boths hole $l1$ and $l2$ can be simultaneouly used to fit live range $l3$, as shown in Figure 4. The first portion of $l3$ fits into the dead hole and a second portion of $l3$ fits into the live hole in $l1$.

Figure 5 shows how each portion of $l3$ fits on $l1$ and $l2$, after $l3$ is split. To make this technique to work, we need to insert *move* to move register values between live ranges. These instructions are required to avoid destroying the values of R1 and R2, thus keeing the semantics of the program correct.

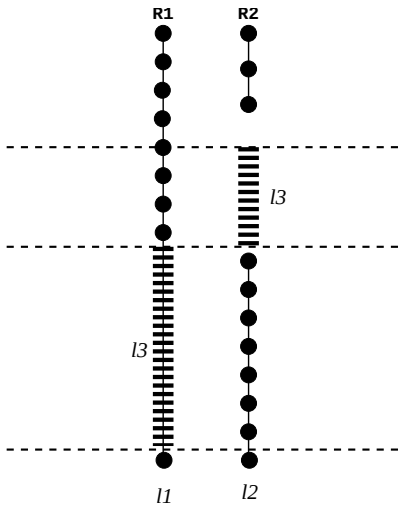

Figure 5: Splitting the live range

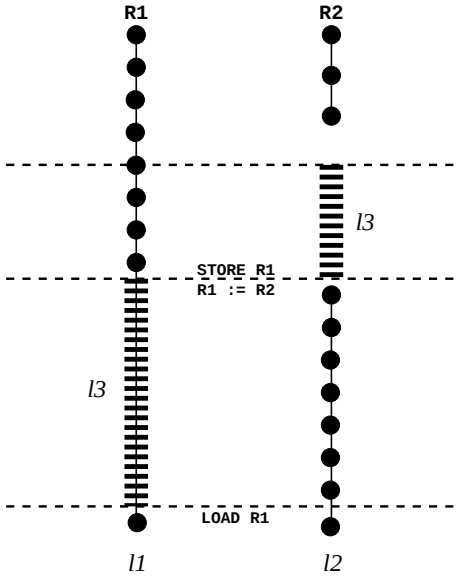

Figure 6: Inserting change instructions

The move instructions are shown in Figure 6. As the first hole used is a dead hole, no change instruction is needed to fit the first portion of $l3$. When $l3$ changes from register R2 to R1, a move instruction is required. Given that R1 is a live hole, we store its contents into memory, thus preserving the value of $l1$, and move the value of $l3$ from R2 to R1, turning R2 free for $l2$ to use, without destroying the value of $l3$. When $l3$ finishes its live range, we restore the value of $l1$ back to R1, inserting a load instruction. Of course, for this technique to work well, the number of load/stoers inserted is this case should be smaller than a traditional live range spilling.

Notice that it does not matter how many instructions $l3$ contains or how many references it makes to the registers that were assigned to it. By splitting part of $l3$ into a dead hole of R1 and a live hole of R2, we only need to insert three move instructions. Regular spill code generation, would break each reference of $l3$ into small pieces and would probably break $l3$ or $l2$ so as to allocate the second portion of $l3$ that interferes with $l1$ and $l2$.

2 Hole Allocation

We call the above described technique *Hole Allocation*. The hole allocation algorithm is not based on the interference among registers, but it is based on minimizing the number of move instructions required to split a live range among the holes of other live ranges. We assume that move instructions are cheaper than memory accesses, a reasonable assumption when considering modern processor architectures, and a solid fact when considering embedded processor architectures.

To minimize the number of move instructions, hole allocation uses a dynamic programming algorithm. To exemplify its working, let's assume an architecture with only 3 registers and the program basic block shown in Figure 7, which contains 4 instructions and 4 live ranges.

For the sake of simplicity, assume that the live range selected for spilling starts and ends in this basic block. This live range interferes with other 3 live ranges that have been assigned to registers R1, R2 and R3.

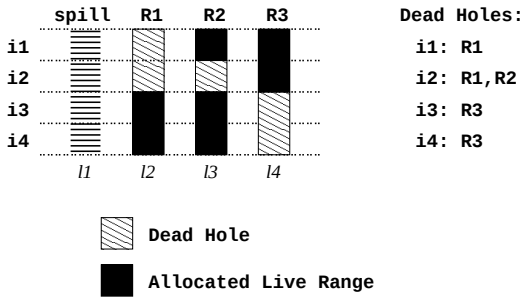

Figure 7: Dead holes in a basic block

For the example in Figure 7, we will also suppose that the instructions where the live ranges are live make at least one reference (use/definition) to R1, R2 and R3.

In Figure 7, dead holes are the empty portions of the live ranges of each register shown. When a register is referenced in a instruction, for use or definition, this register is neither a dead or live hole for such instruction, we thus we say that this register is not available. In Figure 7 we have only dead holes, since we are assuming no live holes for now.

The question now is how to choose the registers in such a way to minimize the number of move instructions resulting from moving the value from one register to another.

The dynamic programming algorithm for hole allocation tries all the possibilities to decide what is the smallest cost. All possibilities for the basic block in Figure 7 are show in the graph in Figure 8.

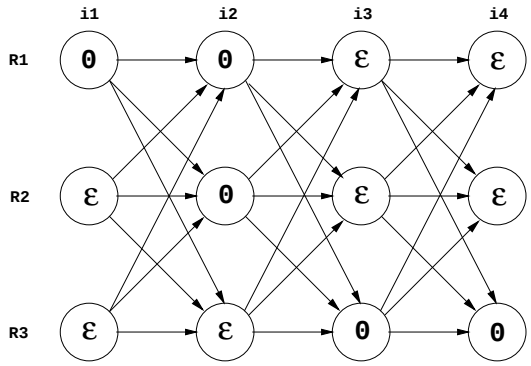

Figure 8: Ways of using dead holes

Figure 8 shows all the possibilities of registers that can be assigned to live range $l1$ at each instruction. For example, at instruction $i1$ we can use register R1 and at instruction $i2$ we can use register R2. The values inside the circles represent the cost of using the register at the instruction. For example, in instruction $i2$ the cost of using register R1 or R2 is 0, but there is no cost in using register R3 in instruction $i2$. These costs are related to live and dead holes. When a register is a dead hole in instruction i, we set the cost of using it as 0, given that R1 has cost zero when assigned to instructions $i1$ and $i2$, R2 has cost zero when used in instruction $i2$ and finally R3 has cost zero when assigned to instructions $i3$ and $i4$.

If a register is a live hole in instruction i, we set the cost as being 1. This is because live holes involve load/store instructions, so using a live hole is more expensive than using a dead hole. When a register is neither a live or dead hole for instruction i, we set its cost to ∞, represented by the symbol ϵ in Figure 8.

Moving from one register to another also has a cost. Moving between the same register has cost 0. The cost of moving between different registers depends on the type of hole that the registers are. The costs are defined in the table 1 and are only applicable when the source and destination registers are different.

Hole Type	Dead Hole	Live Hole	∞
Dead Hole	1	3	∞
Live Hole	3	5	∞
∞	∞	∞	∞

Table 1: Costs from changing of hole type

Moving between dead holes has cost 1 because we only need to insert a MOVE instruction. Moving between live and dead holes has cost 3 because we need to insert a LOAD or STORE and a MOVE instruction. Finally, moving between two live holes has cost 5 since we need to insert a STORE, a MOVE and a LOAD instruction.

The graph in Figure 8 shows all the possible register assignments for the instructions in

live range $l1$.

Moving a live range from one register to another has a cost of adding instructions to the program, so we want to find how to choose the registers in a way to minimize the amount of instructions to be inserted. Take, for example, the example above. In the first instruction we can only use $R1$. In the second instruction, we can use $R1$ or $R2$, keeping the value in $R1$ at no extra cost. But if we choose $R2$ in instruction $i2$, a MOVE from $R1$ to $R2$ is necessary, so it is more profitable to keep the value in $R1$ in instruction $i2$.

As said before, the problem of choosing the right registers can be solved by dynamic programming. At each step the algorithm we tries to make the choice that will lead to the lowest possible cost.

	i_1	i_2	i_3	i_4
R1	0	0	∞	∞
R2	∞	1	∞	∞
R3	∞	∞	1	1

(a)

	i_2	i_3	i_4
P_{R1}	R1	R1	R1
P_{R2}	R1	R1	R1
P_{R3}	R1	R1	R3

$P_f = R3$

(b)

Table 2: Solution for figure 8

Table 2(a) shows the computation of the costs. For instruction $i1$ if we choose register $R1$ the cost is 0, given that for this instruction $R1$ is dead. Registers $R2$ and $R3$ are not available in instruction $i1$, so we can not use them, and thus the cost of choosing these registers is infinity. The next step is to compute the register to be used in instruction $i2$. Starting with $R1$, the cost of moving from $R1$ in $i1$ is 0, because no move is needed. The cost of moving from $R2$ in $i1$ is $\infty + 1 + 0$, where ∞ is the cost of keeping the value in $R2$ in $i1$, 1 is the cost of moving the value from $R2$ to $R1$ and 0 is the availability of $R1$ in $i2$. Since the result is ∞, the best choice, at the moment, is moving from $R1$ in $i1$. The cost of moving from $R3$ in $i1$ to $R1$ in $i2$ is $\infty + 1 + 0$, where ∞ is the cost of keeping the value in $R3$ in $i1$, 1 is the cost of moving the value from $R3$ to $R1$ and 0 is the availability of $R1$ in $i2$. The cost of moving from register $R1$ in instruction $i1$ to register $R1$ in instruction $i2$ is 0, and the cost of moving from register $R2$ in instruction $i1$ to register $R1$ in instruction $i2$ is ∞ and the cost of moving from register $R3$ in instruction $i1$ to register $R1$ in instruction $i2$ is ∞. As we want the lowest cost for register $R1$ in instruction $i2$,we choose to move the value from register $R1$ in instruction $i1$. This choice is represented in table 2(b), by the position addressed by the pair (l_1,$i2$).

This process must be repeated for $R2$ and $R3$ in $i2$, and again for $R1$, $R2$ and $R3$ in

Algorithm 1 Compute Available Registers

```
for each basic block B do
  for each instruction i in B do
    gen[B] = gen[B] ∪ defs[B]
    gen[B] = gen[B] − uses[B]
    kill[B] = kill[B] − defs[B]
    kill[B] = kill[B] ∪ uses[B]
  end for
end for
for each basic block B do
  avail_in[B] = all registers
  avail_out[B] = all registers
end for
while any avail_in change do
  avail_out[B] = ∩ avail_in[P]
  avail_in[B] = gen[B] ∪ (avail_out[B] − kill[B])
end while
```

$i3$ and $i4$. The values and choices are shown in Table 2 (a) and (b). The table 2 (a) has the costs that the dynamic programming algorithm computes at each step of the algorithm. The Table 2 (b) shows the registers selected at each step of the algorithm, showing the **Path** (P_{ri}) starting at each register.

The register used in the last instruction of the basic block will be the one with the lowest cost. By looking to Table 2 (b), we can see that the lowest cost is 1, using $R3$ in instruction $i4$, so $P_f = R3$. The register to be used at each instruction is obtained by the Algorithm 2.

Algorithm 2 Print Registers choosen

```
n ← number of instructions
r ← P_f
print(i_n, r)
for j ← n downto 2 do
  r ← P_r[i_j]
  print(i_{j−1}, r)
end for
```

For Figure 7 the output of Algorithm 2, are the values presented in table 3.

i_4	$R3$
i_3	$R3$
i_2	$R1$
i_1	$R1$

Table 3: Final solution for basic block in figure 7

This approach must be executed for each basic block in the Control Flow Graph (CFG) to produce a complete solution to the whole program. The example in figure 7 showed the solution for a single basic block, so no previous registers were considered. In Figure 9 we compute the solution for basic block $B1$ similarly as for Figure 7. Basic block $B2$ is solved with the output of basic block $B1$, that is, when computing the cost of using the registers in the first instruction of $B2$, we must compute also the cost of moving the value from the last register used in $B1$. If the last register chosen in $B1$ is $R2$, the cost of using $R1$ in $i1$ for $B2$ will be $1+0$ if $R1$ is available or $1+\infty$ if $R1$ if not available at $i1$ in $B2$. The same computation is performed for $R2$ and $R3$ at $i1$ in $B2$.

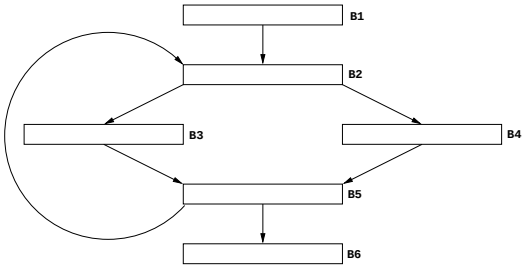

Figure 9: The Control Flow Graph

Blocks $B3$ and $B4$ both take the output of $B2$ to compute the cost for the first instruction. In $B5$ we must analyze the output of $B3$ and $B4$. Suppose that the last instruction in $B3$ uses $R2$ and the last instruction in $B4$ uses $R3$. If $R1$ is available at $i1$ in $B5$, the cost of using it will be $1+1+0$, where the first 1 is a MOVE from $R2$ to $R1$ and the second 1 is a MOVE from $R3$ to $R1$ and 0 is the availability of $R1$. The same computations are done for $R2$ and $R3$ at $i1$ in $B5$.

The back edge from $B5$ to $B2$ is ignored during the solution of finding the lowest costs to split the live range. For traces without back edges, the solution found by the dynamic programming algorithm, is optimal, achieving the lowest number of value movement among the registers. After finding the solution for all basic blocks in the CFG, we must analyze the output register chosen in $B5$ and $B1$ and the input register selected in $B2$ and insert the proper adjust instructions, maybe creating new basic blocks.

When analyzing *live holes* together with *dead holes* some considerations must be done. The instructions to move a register from a *dead hole* to a *live hole* need to access memory (store the value which is live) and a MOVE instruction to change the value of register. If we are moving values between two *live holes*, load instructions (to restore the previous value) may be necessary. Memory access performed by load/store are more expensive than a move, so when computing the costs of moving values between registers, loads and stores can not have the same weight as a move. We define that a load or store have a cost of 2 while the move has the cost of 1. To transfer a value from a *dead hole* to a *live hole* requires 2 instructions (store/move) and has a cost of 3. To transfer a value between two *live holes* has probably a cost of 5, one instruction to store the value of the *live hole* that receives a new value; one move, for moving the value between the live ranges; and one load, to restore the previous value of this *live hole*.

	Hole Allocation			GCC			Improvement	
	Load	**Store**	**Move**	**Load**	**Store**	**Remat**	**Load Reduction**	**Store Reduction**
164.GZIP	15	12	43	53	50	0	71.70%	76.00%
175.VPR	18	38	93	227	188	84	92.07%	79.79%
177.MESA	47	109	281	803	506	14	94.15%	78.46%
181.MCF	7	6	11	16	9	4	56.25%	33.33%
186.CRAFTY	106	101	397	264	170	54	59.85%	40.59%
197.PARSER	11	13	70	68	76	3	83.82%	82.89%
254.GAP	19	21	158	244	160	0	92.21%	86.88%

Table 4: Comparison between Hole Allocation and GCC spill code algorithm

The computation of *dead/live holes* is simple. The *dead holes* can be computed using a data-flow analysis similar to liveness analysis. Actually we are looking for the opposite result of liveness analysis, that we call *available registers analysis*. Algorithm 1 shows how to compute the available registers. The *live holes* are computed by analyzing each instruction once. For each instruction we have the set of all registers available for allocation in the target architecture; from this set we remove the registers referenced by the instruction (definition/use) and remove the registers that are *dead holes* in the instruction. The remaining set is the set of *live holes*, that is, registers that are live but are not being referenced in the current instruction.

This algorithm is not expensive. The computational cost of *dead holes* with available registers has the same complexity as standard liveness analysis. The cost for *live holes* is to visit each program instruction once. This information is only computed once, because after each spilled live range is inserted into the program, the information about live/dead holes is updated by visiting each instruction once. The computation of the register switch for each spilled live range is performed by visiting each program instruction only once, that is, this step has computational complexity of $i * s$, where i is the number of instructions and s is the number os spilled live ranges.

3 Results

We implemented the hole allocation algorithm in GCC 3.3 [1] and used it to generate spill code after register allocation, which was performed by a well-establish algorithm [11]. Programs from SPEC CPU 2000 [2] written in C were used as the testbench, from which we measured the number of of spill code instructions. We compared them with the number of spill code instructions generated by the GCC coloring algorithm.

From Table 4 one can notice that hole allocation generates much less load and store instructions across all programs. We achieve for than 90% reduction of load instructions for 3 programs, while having the 4 remaining above 50%, and more than 80% reduction of store instructions for 2 programs, while the other programs are all above 30%. On the other hand, the number of MOVE instructions is larger than the number of rematerialization instructions

produced by GCC. Since move instructions is very cheap in modern micro-architectures we believe this could eventually be translated to performance speed-up. With hole allocation we reduced the amount of instructions referencing main memory, thus minimizing the overall penalty of the spill code introduced into the program.

Inserting many of MOVE instructions is not a major penalty when compared to inserting load/store instructions, given that data does have to go through the memory hierarchy. If the processor uses a register renaming or alias system, the cost of a MOVE instruction can become almost zero, turning MOVEs instead of load/store much more profitable.

4 Future Work

For the future, we intend to evaluate the Hole Allocation using the speed of the program as the central metric, also we intend to compare it with other register allocation techniques like linear scan [21]. Moreover, by assuming that it starts with a number of dead-hole registers, hole allocation has the potential to become a fast full register allocation algorithm on its own, specially in environments where compiling time is central, as in dynamic binary translation. We intend to explore this variation as well.

References

[1] *GNU Compiler Collection*. `http://gcc.gnu.org`.

[2] *Standard Performance Evaluation Corporation (SPEC)*. `http://www.spec.org`.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[4] Andrew W. Appel and Lal George. Optimal spilling for cisc machines with few registers. *SIGPLAN Not.*, 36(5):243–253, 2001.

[5] Peter Bergner, Peter Dahl, David Engebretsen, and Matthew O'Keefe. Spill code minimization via interference region spilling. *SIGPLAN Not.*, 32(5):287–295, 1997.

[6] D. Bernstein, M. Golumbic, y. Mansour, R. Pinter, D. Goldin, H. Krawczyk, and I. Nahshon. Spill code minimization techniques for optimizing compliers. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 258–263, New York, NY, USA, 1989. ACM.

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

[8] Gregory Chaitin. Register allocation and spilling via graph coloring. *SIGPLAN Not.*, 39(4):66–74, 2004.

[9] Keith D. Cooper and L. Taylor Simpson. Live range splitting in a graph coloring register allocator. In *CC '98: Proceedings of the 7th International Conference on Compiler Construction*, pages 174–187, London, UK, 1998. Springer-Verlag.

[10] Changqing Fu and Kent Wilken. A faster optimal register allocator. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 245–256, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

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

[12] David W. Goodwin and Kent D. Wilken. Optimal and near-optimal global register allocations using 0-1 integer programming. *Softw. Pract. Exper.*, 26(8):929–965, 1996.

[13] Timothy J. Harvey. Reducing the impact of spill code. Technical Report TR98-319, 24, 1998.

[14] Laurie J. Hendren, Guang R. Gao, Erik R. Altman, and Chandrika Mukerji. A register allocation framework based on hierarchical cyclic interval graphs. In *CC '92: Proceedings of the 4th International Conference on Compiler Construction*, pages 176–191, London, UK, 1992. Springer-Verlag.

[15] Priyadarshan Kolte and Mary Jean Harrold. Load/store range analysis for global register allocation. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 268–277, New York, NY, USA, 1993. ACM.

[16] Akira Koseki, Hideaki Komatsu, and Toshio Nakatani. Spill code minimization by spill code motion. In *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, page 125, Washington, DC, USA, 2003. IEEE Computer Society.

[17] Steven M. Kurlander and Charles N. Fischer. Zero-cost range splitting. *SIGPLAN Not.*, 29(6):257–265, 1994.

[18] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205, USA, 1997.

[19] Takuya Nakaike, Tatsushi Inagaki, Hideaki Komatsu, and Toshio Nakatani. Profile-based global live-range splitting. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 216–227, New York, NY, USA, 2006. ACM.

[20] Fernando Magno Quintao Pereira and Jens Palsberg. Register allocation via coloring of chordal graphs. In *CATS '07: Proceedings of the thirteenth Australasian symposium on Theory of computing*, pages 3–3, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc.

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

[22] Michael D. Smith, Norman Ramsey, and Glenn Holloway. A generalized algorithm for graph-coloring register allocation. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 277–288, New York, NY, USA, 2004. ACM.

[23] Omri Traub, Glenn H. Holloway, and Michael D. Smith. Quality and speed in linear-scan register allocation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 142–151, 1998.

[24] Christian Wimmer and Hanspeter Mössenböck. Optimized interval splitting in a linear scan register allocator. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 132–141, New York, NY, USA, 2005. ACM.