Hardness and Algorithms for Local Register Allocation[1]

Vincenzo Liberatore[2] Martin Farach[3] Ulrich Kremer[4]

July 18, 1997

[1]Research partly supported by NSF Career Development Award CCR-9501942, NATO Grant CRG 960215, NSF/NIH Grant BIR 94-12594-03-CONF and an Alfred P. Sloan Research Fellowship.

[2]Department of Computer Science, Hill Center, Rutgers University, New Brunswick, NJ 08903. E-mail: liberato@cs.rutgers.edu. URL: http://www.cs.rutgers.edu/~liberato/.

[3]Department of Computer Science, Hill Center, Rutgers University, New Brunswick, NJ 08903 and Information Sciences Research Center, Bell Labs, 700 Mountain Avenue, Murray Hill, NJ 07974. E-mail: farach@cs.rutgers.edu. URL: http://www.cs.rutgers.edu/~farach/.

[4]Department of Computer Science, Hill Center, Rutgers University, New Brunswick, NJ 08903. E-mail: uli@cs.rutgers.edu. URL: http://www.cs.rutgers.edu/~uli/.

Abstract

We study the problem of local register allocation (LRA): assign pseudo-registers to actual registers in a basic block so as to minimize the spill cost. Our version of LRA does not allow instruction scheduling, but only the insertion of spill code. We assume that the spill cost depends only on the number and type of load and store operations, but not on their position.

Although LRA has been intensively studied, very little was known about it. LRA could be solved exactly by an algorithm that took exponential time and space. On the other hand, LRA was not known to be NP-hard. LRA could be solved by heuristics, but the relative performance of those heuristics was unclear.

In this paper,

- We prove that LRA is NP-hard.
- We describe the first space- and time-efficient optimum algorithm.
- We propose a new heuristics for LRA which we call Conservative-Furthest-First (CFF).
- We conduct detailed experiments on common benchmarks. We compare the optimum algorithm, the heuristics Furthest-First (FF), Clean-First (CF), and CFF. We obtain statistics on the running time of the allocators and on the dynamic instruction count of the compiled code. CFF is the best suboptimal heuristic under both performance measures.

A famous political maxim states that "All politics is local", and we believe that much the same is true for register allocation. *Todd A. Proebsting*

1 Introduction

In this paper, we study the problem of local register allocation (LRA): assign pseudo-registers to actual registers in a basic block so as to minimize the spill cost. For the purpose of this paper, LRA does not allow instruction scheduling, but only the insertion of load and store operations. We postpone the complete definition of the problem to section 2.

Common wisdom asserts that local register allocation is an easy problem, hardly worth studying. This paper challenges this and several other beliefs on local register allocation. We begin the introduction with a "fallacies and pitfalls" section.

1.1 Fallacies and Pitfalls

Fallacy: Most state-of-the-art compilers perform a combined instruction scheduling and register allocation pass.

Instruction scheduling is essential to exploit instruction level parallelism (ILP). However, an instruction schedule for high ILP might increase the traffic between the register file and the memory system. For example, an ILP-optimal schedule might extend the live ranges and causes a high spill cost. Therefore, instruction scheduling and register allocation should arguably be carried out simultaneously.

Unfortunately, both instruction scheduling and register allocation are hard problems, and several compilers do not fully integrate the two phases. In such compilers, instruction scheduling is performed so as to strike a balance between high ILP and the presumed behavior of the register allocator. Once a compromise instruction scheduling has been fixed, the register allocator proper is invoked to obtain a good allocation under the given schedule. For example, in the MIPSpro compiler [RGSL96], the instruction scheduling is followed by the invocation of the Chaitin-Briggs algorithm [BCT94]. Another example is the ACAPS compilers [NG93] that performs a scheduling phase to find a time-optimal scheduling with minimal buffer allocation. Then, the compiler executes a hierarchical register allocation pass. In both compilers, register allocation remains an independent pass and a good register allocator results in good compiled code.

Register allocators of the past often operated under a schedule of instruction that strived to achieve minimum register sufficiency [ASU86, SU70]. Consequently, the register allocator was helped by the instruction scheduling to achieve minimum spill cost. When register allocation operates under a compromise ILP/register-pressure scheduling, the allocator cannot rely any longer on the scheduling to produce an easy problem. Today's register allocators face critical problems that never appeared before.

Fallacy: Local register allocation is well understood.

Local register allocation has been emphasized in several compilers over the decades. Local register allocation precedes global allocation in the FORTRAN IV compiler for a HITAC-5020 [Nak67] and in the FORTRAN H compiler for the IBM System/360 [LM69]. At the other chronological extreme, the ACAPS hierarchical allocator extends a local allocation into a global allocation [HGAM93].

Local register allocation has been a crucial task for optimizing compilers for more than two decades and, in the classification of [HP96], for at least two computer generations.

The objective of local register allocation is to minimize the total traffic between the register file and the memory system. Therefore, it reflects actual performance issues better than register sufficiency and graph coloring. It is then no surprise that local register allocation has retained its fundamental rôle over the years. A compelling case for local register allocation is made in the paper by Hsu, Fischer, and Goodman [HFG89].

Local register allocation has been extensively studied, but it was not quite well understood yet. Two major problems in local register allocation were open before this paper: the intrinsic complexity of the problem, and the quality and the speed of the heuristics.

Fallacy: The complexity of local register allocation is know.

It was known that an optimum local register allocation could be found in exponential time. However, no lower bound established that a better algorithm does not exist. We present the first NP-hardness proof for local register allocation.

Pitfall: The exact optimum for local register allocation is very hard to obtain even with a small register file.

Previous to this paper, the only known algorithm was based on dynamic programming and failed to terminate even with only four registers due to exponential memory requirements [HFG89]. In this paper, we present a new integer programming formulation and a new branch-and-bound algorithm. The branch-and-bound algorithm takes only a polynomial amount of space, never ran out of space on our benchmarks, and always reported an optimum solution. Furthermore, the branch-and-bound algorithm returns the optimum very quickly on most benchmarks even for 256 registers and long basic blocks (more than a thousand instructions). The speed of our optimum algorithm is mainly due to the new integer programming formulation and to the careful engineering of the branch-and-bound algorithm.

Fallacy: Heuristics such as furthest-first (FF) and clean-first (CF) perform well on common benchmark.

The speed of the code produced by FF can be compared with the speed of the code produced by CF. A better performance measure is the speed gap between the code produced by the heuristics and that produced by the optimum. The cost difference between the heuristic and the optimum gives a definite measure of how much is lost by resorting to a fast method rather than to an exponential algorithm. The relative performance of FF and CF compared to the optimum was known before this paper only for small register files and small basic blocks because no efficient optimum algorithm was known. In this paper, we present the first comparison among FF, CF, and the optimum even on large instances. FF is close to the optimum on average. CF is close to the optimum on most benchmarks, but shows a quite erratic behavior.

We will show that both heuristics can perform arbitrarily poorly on some basic block. We propose a simple modification of FF that we call CFF and that offers a worst-case performance guarantee. We show that CFF is also an excellent algorithm on average on all benchmarks.

Pitfall: Finding a local register allocation with a short and straightforward code.

We did not find in the literature any explicit mention of the running time of FF or CF, nor any reference to a careful implementation of those heuristics. However, we did find an implementation of FF with enough details to establish that no refined data structure is used [FH92]. We implemented CFF with binary heaps. The code is somewhat longer than that in the previous straightforward implementation of FF, but CFF is roughly eight times faster on SPEC benchmarks. The algorithm CFF is the first algorithm that presents definite and quantifiable advantages over the classical heuristics FF and CF.

1.2 Contents

We summarize the theoretical and experimental contributions of this paper:

- In Section 2, we will describe the local register allocation problem. We present a more realistic model than that used by previous authors. In particular, we allow for rematerializations, different instruction costs, and multiple references per instruction. We also give the first NP-hardness theorem for local register allocation.
- In Section 3, we describe the instances that we solve in this paper.
- In Section 4, we present a novel integer programming formulation and branch-and-bound algorithm. The branch-and-bound algorithm is the first algorithm to actually return the optimum value on large basic block and large register files.
- In Section 5, we analyze the behavior of the heuristics FF, CF, and CFF, and, in particular, we show that for CFF
  - beside its worst-case performance guarantee, the quality of the allocation is close to the optimum on common benchmarks.
  - CFF is implemented to run roughly two to eleven times faster than other heuristics on SPEC benchmarks.

2 Local Register Allocation

Our model for local register allocation follows mostly Hsu, Fischer, and Goodman [HFG89] and it is consistent with other previous works [HKMW66, Ken72, Luc67]. The model is based on architectures with general purpose registers. Our model extends previous formulations and reflects more realistically actual architectures and instruction sets.

Let $V = \{1, 2, \ldots, M\}$ be a set of pseudo-registers. Pseudo-registers will contain temporary variables and constants. No aliasing between pseudo-registers is possible. We will assume that a pseudo-register is defined at most once, and thus it represents only one live range of a variable. Register allocation assigns pseudo-registers to a set of N actual registers.

We assume that each pseudo-register can be stored and retrieved in a designated memory location. We assign different costs to different instructions and refer to *weighted* instruction count. We assume that loads and stores from memory cost twice as much as any other operation. Notice that load immediates do not involve a memory access and cost half as much as a load from memory. We assume that the spill cost depends only on the number and type of inserted load and store

intermediate code	σ	opt LRA code	cost	Registers		
				R1	R2	R3
				-	-	-
	(read,{0})	LOAD &t0 ⇒ R1	2	t0	-	-
ADDI 3 t0 ⇒ t1	(write,{1})	ADDI 3 R1 ⇒ R2	1	t0	t1	-
	(read,{1,2})	LOADI 4 ⇒ R3	1	t0	t1	t2
SUB t1 t2 ⇒ t3	(write,{3})	SUB R2 R3 ⇒ R1	1	t3	t1	t2
	(read,{3,4})	LOAD &t4 ⇒ R3	2	t3	t1	t4
MUL t3 t4 ⇒ t5	(write,{5})	MUL R1 R3 ⇒ R3	1	t3	t1	t5
	(read,{2,5})	LOADI 4 ⇒ R1	1	t2	t1	t5
SUB t2 t5 ⇒ t6	(write,{6})	SUB R1 R2 ⇒ R1	1	t6	t1	t2
	(read,{1,6})			t6	t1	t2
ADD t1 t6 ⇒ t7	(write,{7})	ADD R2 R1 ⇒ R1	1	t7	t1	t2
		total cost:	11			

Figure 1: Sample LRA with 3 registers. $L = \{7\}$ and t2 contains the constant 4.

operations, and not on their placement. Our cost model is similar to that in [BCT94] and allows us to keep track of simple rematerializations.

In this paper, floating point operations are assumed to cost as much as integer ones. However, the cost model can be modified to include different operation costs. The augmented cost model enjoys most of the theoretical properties described below [FL97].

The set $S = \{\mathsf{clean}, \mathsf{dirty}\}$ is the set of register states, which are explained as follows. Suppose that a pseudo-register i is in the actual register h. If the register state of h is clean, then the value in h is consistent with the value in the memory location assigned to i, otherwise it might differ. A register h is dirty when a pseudo-register i has been defined and its contents are maintained in the real register h, but have not been stored to the memory location corresponding to i.

Each operation issues a sequence of reads and writes to the pseudo-registers. For example the instruction `SUB r69 r68 => r70` reads the pseudo-registers `r69` and `r68`, subtracts them, and stores the result into `r70`. For the purpose of register allocation, a basic block individuates a sequence of references, each of which is either a read or a write of a subset of V. In other words, a *static reference sequence* σ is a sequence of elements in $\{\mathsf{read}, \mathsf{write}\} \times 2^V$.

If a pseudo-register i is either read or written at step j, then a real register has to be reserved to it. If i is already in a register, nothing happens. Otherwise, we insert a load instruction of i into a register. In turn, we might have to evict another pseudo-register i' to make room for i. If i' is dirty and alive, then we have to store i' back to its memory location. The objective of local register allocation is to find a feasible schedule of pseudo-registers into real registers for a basic block so as to minimize the total cost due to loads and stores. Figure 1 gives an example of LRA with $N = 3$.

Notice that if a pseudo-register is not alive, then we do not need to store it. We will denote with L the set of pseudo-registers alive at the end of the basic block. The set L can be determined by live-variable analysis or we can simply assume that $L = V$ [ASU86]. Our theoretical results hold for both choices of L.

Our model of local register allocation is more realistic than that in previous papers. First, previous works counted each load and store as a unit of cost. In this paper, we allow for a more structured cost model that better reflects the cost of spilling. Second, we allow for multiple references in one step. The example above shows that multiple references arise commonly in actual code.

intermediate code	σ	FF LRA code	cost	Registers		
				R1	R2	R3
				-	-	-
	(read,{0})	LOAD &t0 ⇒ R1	2	t0	-	-
ADDI 3 t0 ⇒ t1	(write,{1})	ADDI 3 R1 ⇒ R2	1	t0	t1	-
	(read,{1,2})	LOADI 4 ⇒ R3	1	t0	t1	t2
SUB t1 t2 ⇒ t3	(write,{3})	SUB R2 R3 ⇒ R1	1	t3	t1	t2
		STORE R2 ⇒ &t1	2	t3	t1	t2
	(read,{3,4})	LOAD &t4 ⇒ R2	2	t3	t4	t2
MUL t3 t4 ⇒ t5	(write,{5})	MUL R1 R2 ⇒ R2	1	t3	t5	t2
	(read,{2,5})			t3	t5	t2
SUB t2 t5 ⇒ t6	(write,{6})	SUB R3 R2 ⇒ R1	1	t6	t5	t2
	(read,{1,6})	LOAD &t1 ⇒ R2	2	t6	t1	t2
ADD t1 t6 ⇒ t7	(write,{7})	ADD R1 R2 ⇒ R1	1	t7	t1	t2
		total cost:	14			

Figure 2: Sample Furthest-First LRA with 3 registers. $L = \{7\}$ and t2 contains the constant 4.

A few heuristics have been proposed for LRA. The oldest is probably *Furthest-First* (FF): if no register is empty, evict the pseudo-register that is requested furthest in the future [Bel66]. FF is not necessarily optimum — compare figure 2 with figure 1. Unlike FF, the optimum LRA keeps into account both distance to the next reference and eviction costs. FF and other heuristics are discussed in section 5.

The complexity of LRA is established by the following

Theorem 2.1 *The decision problem corresponding to local register allocation (i.e. given a basic block and two positive integers N and z, decide whether there is a local register allocation whose cost is no more than z) is NP-complete.*

Proof Sketch. The reduction is from set cover. The involved argument is found in [FL97]. □

3 Experimental Setup

We performed experiments with ILOC, the Intermediate Language for Optimizing Compilers developed at Rice University[1]. We used several ILOC routines from the fmm and SPEC benchmarks. The benchmarks have been heavily optimized by the following passes: reassociation, lazy code motion, constant propagation, peephole analysis, dead code elimination, strength reduction, followed by a second pass of lazy code motion, constant propagation, peephole analysis, and dead code elimination[2]. We did not have any part in the coding of the benchmark, in the choice of optimization passes, nor in the selection of the input to those benchmarks.

We strived to isolate as much as possible the effects of local register allocation from global choices. We assumed that the register file is empty at the beginning of each basic block. We conducted live range analysis and stored all live pseudo-registers at the end of each basic block. The additional load and store operations at the beginning and at the end of a basic block might

[1] URL: `http://softlib.rice.edu/MSCP/MSCP.html`

[2] The ILOC benchmarks can be obtained from Tim Harvey (`harv@cs.rice.edu`).

benchmark	prg	Double		Integer	
		blcks	avg len	blcks	avg len
fmm	fmin	56	22.929	54	4.463
	rkf45	129	10.845	132	26.508
	seval	37	7.8108	43	19.047
	solve	96	4.875	110	27.791
	svd	214	7.9579	226	38.739
	urand	10	6.1	13	18.385
	zeroin	31	20.097	30	5.7
spec	doduc	1898	16.663	1998	25.592
	fpppp	305	76.256	336	76.167
	matrix300	7	2.5714	62	23.113
	tomcatv	72	11.681	73	73.479
spec95X	applu	493	16.262	679	55.894
	wave5X	6444	10.917	7006	53.253

Table 1: Characteristics of static reference sequences from optimized code

degrade the overall performance. However, our objective was not to obtain the best possible global allocation, but to compare local allocation strategies once the global choices are fixed.

We assumed we had N integer registers and N double precision registers. We performed experiments for a number of registers ranging as $N = 2, 4, 8, \ldots, 256$. We used a SUN UltraSparc1 (143MHz/64Mb) and the `gcc -O3` compiler.

Table 1 describes the reference sequences used in the experiments. Column 1 gives the name of the benchmark suite and column 2 the program name. Column 3 and 4 report data for the static reference sequences of double precision variables. Column 3 gives the number of basic block where there is at least one live double precision variable. Column 4 gives the average length of the corresponding reference sequences. Finally, column 5 and 6 report the same quantities for integer sequences.

We notice that the program fpppp is quite different from the other benchmarks. First, fpppp has on average the longest double and integer reference sequences. Moreover, fpppp contains the longest sequence among all our benchmarks: the basic block `_.fpppp_` generates a double precision reference sequence of length 6579 — nearly 5 times longer than any other sequence. Therefore, we expect fpppp to be a difficult case in view of the NP-hardness of LRA.

4 An Integer Program

In this section, we will give an integer programming formulation of LRA, present a corresponding branch-and-bound algorithm, and examine its performance. The algorithm is slower than some heuristics, but it returns the best possible local allocation. The optimum algorithm is used in this paper as a local allocator and as a definite point of comparison for faster heuristics. Several other applications of optimum algorithms are discussed for example in [Kre98].

4.1 Integer Programming Formulation

The first time a dirty pseudo-register i is evicted, it has to be stored. If i is subsequently reloaded and evicted, i is clean and no store has to be inserted again. In other words, since a pseudo-register

models one live range, at most one store can be attributed to it. So, the decision of evicting i is a binary decision variable $x_i \in \{0,1\}$ $(i = 1,2,\ldots,M)$, where $x_i = 1$ if i is ever stored, 0 otherwise. The cost c_i of x_i is 2 if a store to memory has to be inserted to evict i, 0 otherwise.

Notice that it is legal to keep the pseudo-register i out of the register file at step j if either i is not alive at step j or if i is alive, but is not requested at step j. Define a *value range* of a pseudo-register i to be a maximal sequence of steps where i is alive and it is legal to keep it out of the register file. A value range of i might consist of the steps strictly between two consecutive references to i, the steps before the first reference to i if i is alive at the entry of the basic block, or the steps after the last reference to i if i is alive at the end of the basic block. The number of value ranges of i will be denoted by $K(i)$. Again, if we ever evict i along its kth value range then we might have to reload i at the end of the value range no matter where the eviction occurred. So, the decision of evicting i during its kth value range is a binary variable y_{ik}, where $y_{ik} = 1$ if i is evicted along its kth value range, 0 otherwise. The cost c_{ik} of y_{ik} is 2 if i has to be reloaded at the end of the value range by a load from memory, 1 if i is reloaded by a load immediate, and 0 otherwise.

We have now introduced all the variables we will use and we turn to the definition of constraints. The condition: (i is evicted during its kth value range) entails (i is evicted) becomes the constraint $x_i \geq y_{ik}$ $(i = 1,2,\ldots,M$ and $k = 1,2,\ldots,K(i))$.

Let L_j be the number of pseudo-registers alive at step j. Define the register pressure $\rho_j = \max\{0, |L_j| - N\}$. The pressure ρ_j gives the number of pseudo-register that are alive at step j and that have to reside outside the register file at step j. Let K_j as the set of pairs (i,k) such that the kth value range of i contains step j. Then, for all steps j we will introduce a constraint

$$\sum_{(i,k)\in K_j} y_{ik} \geq \rho_j \ ,$$

which forces at least ρ_j of the live pseudo-registers to be out of the register file at step j.

The integer program for a reference sequence σ is on the whole

$$\begin{array}{lll}
\min & \sum_{i=1}^{M} c_i x_i + \sum_{i=1}^{M} \sum_{k=1}^{K(i)} c_{ik} y_{ik} & (1.1) \\
\text{s.t.} & x_i \geq y_{ik} \qquad i = 1,2,\ldots,M; k = 1,2,\ldots,K(i) & (1.2) \\
& \sum_{(i,k)\in K_j} y_{ik} \geq \rho_j \qquad j = 1,2,\ldots,|\sigma| & (1.3) \\
& x_i, y_{ik} \text{ binary} \qquad i = 1,2,\ldots,M; k = 1,2,\ldots,K(i) & (1.4)
\end{array}$$

Figure 3 gives an example of formulation (1).

A branch-and-bound algorithm solves the integer program (1) with only a polynomial amount of space. The previous optimum algorithm was based on dynamic programming [HKMW66, Luc67, HFG89], took an exponential amount of space in the worst case, and failed to terminate on a few benchmarks due to the lack of memory space. The branch-and-bound algorithm is space-efficient. It also becomes time-efficient once it is modified as described below.

4.2 Branch-and-Bound

Let $\mathcal{N}$ be the part of the constraint matrix corresponding to the constraints (1.3).

Proposition 4.1 *The matrix $\mathcal{N}$ has the consecutive ones property in the columns, that is, in each column, the ones appear in consecutive positions.*

Assumptions: $N = 2$, the pseudo-register 3 contains an immediate.
Boundary conditions: only 3 alive on entry, $L = \{1, 2\}$.

σ			
		min	$2x_1 + 2x_2 + 2y_{11} + 2y_{21} + y_{31} + y_{32}$
(write,{1})	$\longmapsto$	s.t.	$y_{31} \geq 0$
(write,{2})	$\longmapsto$		$y_{11} + y_{31} \geq 0$
(read,{3})	$\longmapsto$		$y_{11} + y_{21} \geq 1$
(read,{2})	$\longmapsto$		$y_{11} + y_{32} \geq 1$
(read,{3})	$\longmapsto$		$y_{11} + y_{22} \geq 1$
(read,{1})	$\longmapsto$		$y_{22} \geq 0$
			$x_1 \geq y_{11}$
			$x_2 \geq y_{21}$
			$x_2 \geq y_{22}$
			$x_3 \geq y_{31}$
			$x_3 \geq y_{32}$
			$x_1, x_2, x_3, y_{11}, y_{21}, y_{22}, y_{31}, y_{32} \in \{0, 1\}$

$$\Longrightarrow \mathcal{N} = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Figure 3: Sample formulation

Proof Sketch. Value ranges extend over intervals of the basic block. □

Matrices with the consecutive ones property are totally unimodular and correspond to network flow problems. Therefore, those matrices define problems that can be solved efficiently and that always yield an integral solution [AMO93].

The total unimodularity of $\mathcal{N}$ implies that the constraints that the y_{ik}'s be binary can be substituted with the much weaker continuous constraints $0 \leq y_{ik} \leq 1$ ($i = 1, 2, \ldots, M$ and $k = 1, 2, \ldots, K(i)$). Indeed, once the x_i's have been fixed to their optimum integer value, then also the variables y_{ik}'s will be integer in an optimum basic solution by total unimodularity of $\mathcal{N}$. Then, the number of integer variables in the problem is dramatically reduced. The solid bars in the figure 4 and 5 give the percentage of decision variables that are real. The reduction of the number of integer decision variables diminishes the amount of branching of the branch-and-bound algorithm and results in a shorter execution time.

There is a second and more subtle change to the branch-and-bound algorithm that leads to a fast procedure. First, we claim that the constraints (1.3) constitute most of the constraints in (1). The dashed bars in pictures 4 and 5 give the percentage of constraints due to the network matrix $\mathcal{N}$ as a percentage of the total number of constraints. Since $\mathcal{N}$ constitutes a large fraction of constraints, the initial linear relaxation of (1) can be efficiently solved as a network flow problem with side constraints [AMO93].

Finally, we expect the running time of the branch-and-bound algorithm to decrease as N increases. Indeed, more basic blocks can be scheduled without spilling as N increases. In turn, less instances have to be solved by actual branch and bound.

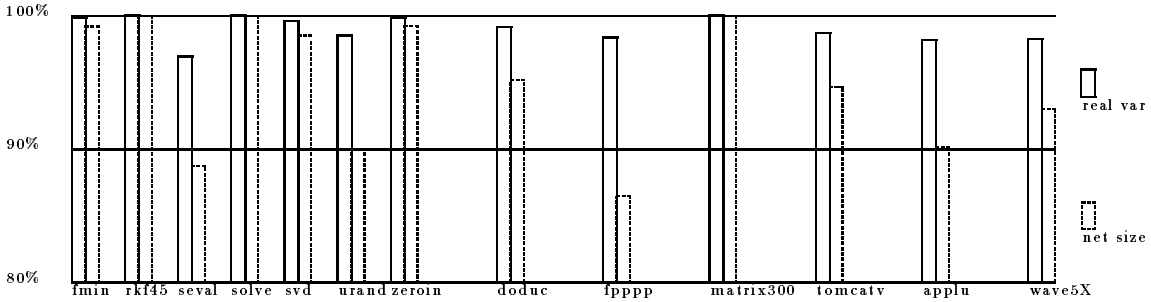

Figure 4: Characteristics of (1): double reference sequences

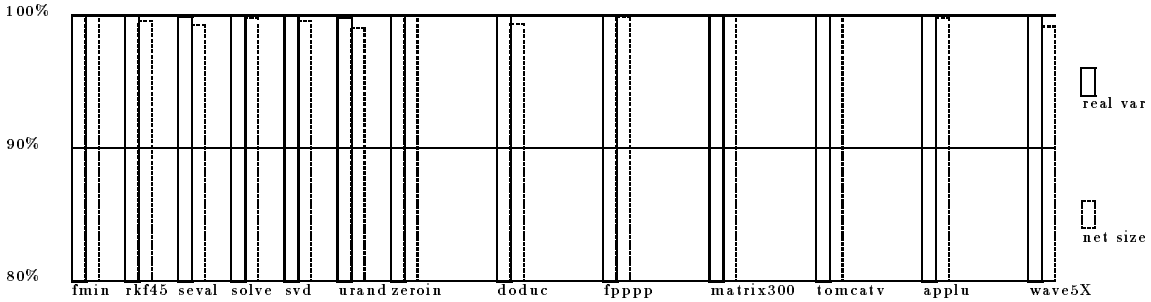

Figure 5: Characteristics of (1): integer reference sequences

benchmark	prg	N							
		2	4	8	16	32	64	128	256
fmm	fmin	0.196	0.178	0.17	0.157	0.149	0.148	0.148	0.148
	rkf45	0.497	0.468	0.441	0.406	0.373	0.372	0.372	0.371
	seval	0.159	0.145	0.132	0.123	0.119	0.119	0.119	0.119
	solve	0.4	0.381	0.364	0.34	0.313	0.306	0.306	0.305
	svd	1.01	0.95	0.901	0.837	0.73	0.699	0.698	0.699
	urand	0.0519	0.0391	0.037	0.035	0.0346	0.035	0.0348	0.0348
	zeroin	0.111	0.0969	0.0921	0.0851	0.0835	0.0831	0.0828	0.0828
spec	doduc	8.99	8.52	8.06	7.48	6.97	6.76	6.28	6.28
	fpppp	23.6	26.1	1.49e+03	139	15.9	20.8	11.2	8.42
	matrix300	0.138	0.126	0.121	0.103	0.101	0.101	0.101	0.101
	tomcatv	0.484	0.464	0.45	0.418	0.367	0.291	0.27	0.27
spec95X	applu	4.23	3.85	3.87	3.45	3.03	2.57	2.34	2.3
	wave5X	43.3	40.6	38.2	34.9	30.7	26.8	25.4	25.2

Table 2: Branch-and-bound run-times in seconds

4.3 Experimental Results

We implemented the branch-and-bound algorithm with calls to the CPLEX[3] callable library. Table 2 reports the user plus system running time of the optimum in seconds. The reported times do not include I/O and live range analysis.

The branch-and-bound running time decreases as N increases for all programs but fpppp. The optimum takes always less than one minute except for one benchmark and two choices of N. Broadly speaking, we observe that the optimum tends to take longer on larger programs.

The fpppp benchmark has a different behavior. We found that fpppp could be explained in terms of its longest sequence of references. The branch-and-bound algorithm does not generate any node for that sequence for all $N \neq 8, 16$, but it visits 12807 nodes for $N = 8$ and 1034 nodes for $N = 16$. Correspondingly, the running time jumps from 26 seconds to 25 minutes! No other basic block exhibits such an extreme behavior. The running time is exposed to the "NP-noise", which the long basic block dramatically amplifies.

In the next section, we will compare the optimum with the heuristics.

5 Heuristics

We experimented with three heuristics. Each heuristic specifies which pseudo-register is to be evicted if no empty register is available.

Furthest-First (FF) evict a pseudo-register that is used furthest in the future [Bel66].

Clean-First (CF) evict a clean pseudo-register that is used furthest in the future. If no clean pseudo-register exists, evict a dirty pseudo-register that is requested furthest in the future [FL88].

[3]CPLEX is a trademark of CPLEX Optimization, Inc.

Conservative-Furthest-First (CFF) determine the set S of pseudo-registers that are requested furthest in the future. Evict a clean pseudo-register in S. If all pseudo-register in S are dirty, evict an element of S arbitrarily.

CFF is a combination of FF and CF: it is similar to FF, with ties resolved by CF. So, FF is not properly speaking one algorithm, but a family of algorithms, and one of the members of this family is CFF. We now specify the tie-breaking rule for FF that has been implemented in an existing compiler: break ties by evicting the pseudo-register that occupies the lowest number register [FH92]. The heuristic adopting this rule will be denoted as FFi, or, when no confusion can arise, simply as FF.

5.1 Worst-Case Analysis

We will examine the worst-case performance of the heuristics under the following stringent non-compulsory load cost model. Suppose that a register i is alive at the entry of the basic block, but it is not initially present in a register. The first time i is read, i is loaded into a real register. However, the first load is a cost that every allocation is forced to pay if the initial register contents are fixed. No allocation can improve on these *compulsory loads* once the initial register contents are given.

The non-compulsory spill cost of CFF is never 4 times worse than the optimum [FL97]. By contrast, we will now show that neither FF nor CF have a performance guarantee in the worst case.

First, consider CF. Construct a reference sequence block that fills all the register but one with dirty variables. Then, ask repeatedly two clean variables for $n+1$ times: $((\mathsf{read}, \{1\}), (\mathsf{read}, \{2\}))^{n+1}$. CF repeatedly evicts and reloads 1 and 2 and pays a total non-compulsory cost of $4n$. In this case, the optimum strategy is FF that would evict a dirty variable, paying a cost of 2. Take n arbitrarily large and, while n grows, the cost ratio of CF over FF tends to infinity. In view of CF's worst instance, we now return to discuss the store operations inserted at the basic block boundaries. CF will not likely incur its worst case on our benchmarks because the boundary stores introduce many clean variables. Nonetheless, CF might still occasionally face a worst-case reference sequence especially on long basic blocks, and we will discover that CF might produce much poorer allocations than FF or CFF.

We now turn to FF. Here the argument is more tricky. Indeed, it can be shown that FF is never 4 times worse than the optimum if both compulsory and non-compulsory loads are accounted for. However, if we account only for compulsory loads, then FF can be arbitrarily worse than the optimum even on a very small reference sequence. Let $\sigma = (\mathsf{write}, \{1\}), (\mathsf{read}, \{2\}), (\mathsf{read}, \{3\})$, $L = \{1, 2, 3\}$, and $N = 2$. The pseudo-register 2 is alive on entry and it will be clean whenever loaded into an actual register. Then, FF evicts the lowest number register 1 for a non-compulsory cost of 2, while the optimum evicts 2 for no non-compulsory cost. The ratio of FF and the optimum is now infinity. CFF is similar to FF, but it breaks ties carefully. In this example, CFF does not pay any non-compulsory cost. In general, it can be shown that CFF is never 4 times worse than the optimum even if we account only for non-compulsory loads and stores [FL97].

5.2 Implementation Issues

A current implementations of FF searches for a register to evict by repeatedly scanning the register file and the basic block [FH92]. Henceforth, we will assume that FF uses this strategy.

We propose an implementation of CFF with a priority queue. A priority queue is a data structure that supports the following operations on a collection of items:

```
for all j's and l's do
    next-ref(j,l) ← the next step with a reference to the lth pseudo-register requested at step j
done
for j = 1 to r do
    f ← the number of pseudo-registers requested at step j, but not in the register file
    f ← f− (the number of empty registers + the number of registers taken by dead pseudo-registers)
    for l = 1 to f do
        evict the pseudo-register DEL-MAX()
    done
    for l ← 1 to the number of pseudo-registers requested at step j do
        i ← lth pseudo-register requested at step j
        if i is not in a register then load i else DELETE(i) end if
        x ← next-ref(j,l)
        if i is dirty then x ← x + 1/2 end if
        INSERT(i,x)
    done
done
```

Figure 6: Conservative Furthest First

INSERT(i,x): insert item i in the collection with key x
DELETE(i): delete item i from the collection
DEL-MAX(): return the item with the largest key from the collection and delete it

A priority queue can be implemented with a binary heap to run in $O(\log n)$ time per operation. Recently, a new priority queue has been proposed with $O(1)$ expected time per operation [Ram96], but it is not clear yet if that data structure is also efficient in practice, so we did not use it. The algorithm CFF is depicted in figure 6.

The running time of CFF is $O(r \log N)$, where r is the number of references to pseudo-registers in the basic block and N is the number of registers. The traditional implementation of FF takes $O(r^2 N)$.

We also implemented CF with binary heaps with an algorithm analogous to that in figure 6.

5.3 Experimental Results

CFF is an improvement over FF in the worst-case relatively both to the quality of the allocation and the running time. We performed experiments to ascertain if the improvement holds also on average on common benchmarks. The experimental setup is the same as that described in section 3. We transformed the ILOC code into C programs with the i2c tool. We then inserted code to count the number of times each basic block was executed. The dynamic weighted instruction count is given by $\sum_{\text{basic block } \mathcal{B}}$ (number of times $\mathcal{B}$ was executed) $\times$ (weighted instruction count for $\mathcal{B}$). The experimental results are given in the following tables for $N = 16, 32, 64$, which are representative cases of today's register file sizes [HP96]. The time is the time taken by the heuristics to run in seconds. It does not include the time to read the input file nor the time to perform live range analysis. The cost is the total weighted dynamic instruction count in thousands.

The fmm suite appears to be too small to provide insight on the performance of the algorithms.

Both CFF and CF are faster than the branch-and-bound algorithm. FF is often slower. CFF

and FF return allocations of nearly the same cost and close to the optimum on all benchmarks. The most critical case for the heuristics is fpppp. In particular, CF can be much worse than CFF and FF on fpppp. Moreover, CF spill code is 30% to 57% worse than the optimum on the long basic block in fpppp. We conclude that CF is worse than FF and CFF on programs with long basic blocks, where little help is given by boundary spill code.

CFF runs 8 (10, 11) times faster than FF on (geometric) average for $N = 16$ (32, 64) across the SPEC benchmarks and twice as fast as CF. FF is often slower than the branch-and-bound algorithm.

In conclusion, CFF offers worst-case performance guarantee relatively both to running-time and the quality of the allocation. Moreover, it is quite stable across all our benchmark, offers allocations close to the optimum, and it is very fast. Therefore, we recommend CFF for LRA.

Acknowledgments

We gratefully acknowledge helpful conversations with Barbara Ryder. We thank Keith Cooper, Tim Harvey, and Taylor Simpson from the Massively Scalar Compiler Group at Rice University for providing us with the ILOC software and benchmark codes.

References

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows.* Prentice-Hall, Englewood Cliff, NJ, 1993.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[BCT94] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

[Bel66] L. A. Belady. A study of replacement algorithms for a virtual storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[FH92] Christopher W. Fraser and David R. Hanson. Simple register spilling in a retargetable compiler. *Software — Practice and Experience*, 22(1):85–99, January 1992.

[FL88] Charles N. Fischer and Richard J. LeBlanc, Jr. *Crafting a Compiler.* Benjamin/Cummings, Menlo Park, CA, 1988.

[FL97] Martin Farach and Vincenzo Liberatore. On local register allocation. Technical Report TR97-33, DIMACS, 1997.

[HFG89] Wei-Chung Hsu, Charles N. Fischer, and James R. Goodman. On the minimization of load/stores in local register allocation. *IEEE Transactions on Software Engineering*, 15(10):1252–1260, October 1989.

[HGAM93] Laurie J. Hendren, Guang R. Gao, Erik R. Altman, and Chandrika Mukerji. A register allocation framework based on hierarchical cyclic interval graphs. ACAPS Technical Memo 33, McGill University, February 1993.

[HKMW66] L. P. Horwitz, R. M. Karp, R. E. Miller, and S. Winograd. Index register allocation. *Journal of the Association for Computing Machinery*, 13(1):43–61, January 1966.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, second edition, 1996.

[Ken72] Ken Kennedy. Index register allocation in straight line code and simple loops. In Randall Rustin, editor, *Design and Optimization of Compilers*, pages 51–63. Prentice-Hall, Englewood Cliffs, NJ, 1972.

[Kre98] Ulrich Kremer. Optimal and near-optimal solution for hard compilation problems. *Parallel Processing Letters*, 1998. To appear.

[LM69] Edward S. Lowry and C. W. Medlock. Object code optimization. *Communications of the ACM*, 12(1):13–22, January 1969.

[Luc67] F. Luccio. A comment on index register allocation. *Communications of the ACM*, 10(9):572–574, September 1967.

[Nak67] Ikuo Nakata. On compiling algorithms for arithmetic expressions. *Communications of the ACM*, 10(8):492–494, August 1967.

[NG93] Qi Ning and Guang R. Gao. A novel framework of register allocation for software pipelining. In *Proceedings of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 29–42, January 1993.

[Ram96] Rajeev Raman. Priority queues: Small, monotone and trans-dichotomous. In *Proc. ESA '96*, number 1136 in Lecture Notes in Computer Science, pages 121–137, 1996.

[RGSL96] John Ruttenberg, G. R. Gao, A. Stoutchinin, and W. Lichtenstein. Software pipelining showdown: Optimal vs. heuristics methods in production compilers. In *Proc. SIGPLAN '96 Conf. on Programming Language Design and Implementation*, pages 1–11, May 1996.

[SU70] Ravi Sethi and J. D. Ullman. The generation of optimal code for arithmetic expressions. *Journal of the Association for Computing Machinery*, 17(4):715–728, October 1970.

benchmark	prg		OPT	CFF		FF		CF	
					%		%		%
fmm	fmin	time	0.16	0.02	12.9%	0.03	16.54%	0.04	27.98%
		cost	2	2	0%	2	0%	2	0%
	rkf45	time	0.41	0.05	12.73%	0.10	23.64%	0.11	28.25%
		cost	385	385	0%	385	0%	385	0%
	seval	time	0.12	0.01	12.13%	0.02	13.47%	0.03	25.53%
		cost	1	1	0%	1	0%	1	0%
	solve	time	0.34	0.04	11.97%	0.09	25.67%	0.09	26.24%
		cost	2	2	0%	2	0%	2	0%
	svd	time	0.84	0.09	11.07%	0.31	36.67%	0.21	24.86%
		cost	9	9	0%	9	0%	9	0%
	urand	time	0.035	0.004	12.05%	0.004	11.02%	0.009	26.37%
		cost	2	2	0%	2	0%	2	0%
	zeroin	time	0.09	0.01	12.91%	0.01	13.92%	0.02	28.27%
		cost	2	2	0%	2	0%	2	0%
spec	doduc	time	7.47	0.83	11.08%	3.5	46.81%	1.71	22.87%
		cost	13208	13238	.23%	13238	.23%	13228	.15%
	fpppp	time	139.4	0.30	0.214%	14.23	10.21%	0.66	0.48%
		cost	193356	194858	.78%	194858	.78%	209013	8.08%
	matrix300	time	0.10	0.02	17.25%	0.02	17.22%	0.03	28.48%
		cost	4222	4222	0%	4222	0%	4222	0%
	tomcatv	time	0.418	0.039	9.53%	0.46	110.8%	0.09	22.8%
		cost	413778	416379	.63%	416379	.63%	421581	1.89%
spec95	applu	time	3.451	0.3239	9.385%	4.331	125.5%	0.6939	20.11%
		cost	613716	615261	.25%	615461	.28%	616486	.45%
	wave5X	time	34.92	3.421	9.797%	32.32	92.56%	7.531	21.57%
		cost	615109	616697	.26%	616721	.26%	616225	.18%

Table 3: Performance for $N = 16$. Time is algorithm running time. Cost is weighted dynamic instruction count. Percentage time is fraction of optimum. Percentage cost is variation over optimum.

benchmark	prg		OPT	CFF	%	FF	%	CF	%
fmm	fmin	time	0.1489	0.01995	13.39%	0.01984	13.32%	0.0436	29.28%
		cost	2	2	0%	2	0%	2	0%
	rkf45	time	0.3734	0.05053	13.53%	0.0552	14.79%	0.1131	30.29%
		cost	385	385	0%	385	0%	385	0%
	seval	time	0.1193	0.01446	12.12%	0.01385	11.61%	0.03126	26.21%
		cost	1	1	0%	1	0%	1	0%
	solve	time	0.3134	0.03936	12.56%	0.05672	18.1%	0.08753	27.93%
		cost	2	2	0%	2	0%	2	0%
	svd	time	0.7304	0.0885	12.12%	0.1712	23.44%	0.2035	27.87%
		cost	9	9	0%	9	0%	9	0%
	urand	time	0.0346	0.004359	12.6%	0.003692	10.67%	0.009069	26.21%
		cost	2	2	0%	2	0%	2	0%
	zeroin	time	0.08345	0.01083	12.98%	0.01073	12.86%	0.02379	28.51%
		cost	2	2	0%	2	0%	2	0%
spec	doduc	time	6.965	0.8068	11.58%	4.443	63.79%	1.696	24.35%
		cost	12687	12723	.29%	12723	.29%	12699	.09%
	fpppp	time	15.87	0.2925	1.843%	25.13	158.4%	0.6527	4.113%
		cost	167076	169013	1.16%	169013	1.16%	178557	6.87%
	matrix300	time	0.1012	0.01704	16.84%	0.01336	13.2%	0.0292	28.86%
		cost	4222	4222	0%	4222	0%	4222	0%
	tomcatv	time	0.3674	0.03916	10.66%	0.5251	142.9%	0.0951	25.89%
		cost	343449	343449	0%	343449	0%	343449	0%
spec95	applu	time	3.026	0.3154	10.42%	6.162	203.6%	0.6929	22.9%
		cost	591098	591098	0%	591098	0%	591098	0%
	wave5X	time	30.72	3.306	10.76%	43	140%	7.471	24.32%
		cost	568015	568532	.09%	568532	.09%	568088	.01%

Table 4: Performance for $N = 32$. Time is algorithm running time. Cost is weighted dynamic instruction count. Percentage time is fraction of optimum. Percentage cost is variation over optimum.

benchmark	prg		OPT	CFF	%	FF	%	CF	%
fmm	fmin	time	0.15	0.02	13.42%	0.02	15.21%	0.044	29.46%
		cost	2	2	0%	2	0%	2	0%
	rkf45	time	0.37	0.05	13.62%	0.06	16.51%	0.11	30.48%
		cost	385	385	0%	385	0%	385	0%
	seval	time	0.12	0.01	12.26%	0.01	12.95%	0.03	26.22%
		cost	1	1	0%	1	0%	1	0%
	solve	time	0.31	0.04	12.83%	0.05	15.17%	0.09	28.57%
		cost	2	2	0%	2	0%	2	0%
	svd	time	0.70	0.09	12.41%	0.12	16.81%	0.20	28.95%
		cost	9	9	0%	9	0%	9	0%
	urand	time	0.03	0.004	12.11%	0.004	11.78%	0.009	25.69%
		cost	2	2	0%	2	0%	2	0%
	zeroin	time	0.08	0.01	13.27%	0.01	14.29%	0.02	28.26%
		cost	2	2	0%	2	0%	2	0%
spec	doduc	time	6.76	0.80	11.78%	5.9	87.44%	1.7	24.91%
		cost	12305	12310	0.04%	12310	0.04%	12310	0.04%
	fpppp	time	20.79	0.28	1.363%	51.73	248.8%	0.64	3.073%
		cost	150466	151062	0.40%	151062	0.40%	158519	5.35%
	matrix300	time	0.10	0.02	16.83%	0.02	15.61%	0.03	28.78%
		cost	4223	4223	0%	4223	0%	4223	0%
	tomcatv	time	0.29	0.04	12.26%	0.19	64.05%	0.09	31.36%
		cost	343450	343450	0%	343450	0%	343450	0%
spec95	applu	time	2.57	0.30	11.6%	6.75	262.9%	0.67	26.28%
		cost	577709	577709	0%	577709	0%	577709	0%
	wave5X	time	26.8	3.16	11.79%	41.08	153.3%	7.32	27.33%
		cost	543432	543432	0%	543432	0%	543432	0%

Table 5: Performance for $N = 64$. Time is algorithm running time. Cost is weighted dynamic instruction count. Percentage time is fraction of optimum. Percentage cost is variation over optimum.