Evaluation of Algorithms for Local Register Allocation[⋆]

Vincenzo Liberatore[1] thanks UMIACS, University of Maryland, College Park, MD 20742. E-mail: vliberatore@acm.org. , Martin Farach-Colton[2] [⋆⋆], and Ulrich Kremer[3] [⋆⋆⋆]

[1] UMIACS, University of Maryland, College Park, MD 20742, USA,
vliberatore@acm.org,
WWW home page: http://www.umiacs.umd.edu/users/liberato/

[2] Dept. of Comp. Sc., Rutgers University, New Brunswick, NJ 08903, USA,
farach@cs.rutgers.edu,
http://www.cs.rutgers.edu/∼farach/

[3] Dept. of Comp. Sc., Rutgers University, New Brunswick, NJ 08903, USA,
farach@cs.rutgers.edu,
http://www.cs.rutgers.edu/∼farach/.

Abstract. Local register allocation (LRA) assigns pseudo-registers to actual registers in a basic block so as to minimize the spill cost. In this paper, four different LRA algorithms are compared with respect to the quality of their generated allocations and the execution times of the algorithms themselves. The evaluation is based on a framework that views register allocation as the combination of boundary conditions, LRA, and register assignment. Our study does not address the problem of instruction scheduling in conjunction with register allocation, and we assume that the spill cost depends only on the number and type of load and store operations, but not on their positions within the instruction stream.

The paper discusses the first optimum algorithm based on integer linear programming as one of the LRA algorithms. The optimal algorithm also serves as the base line for our study in terms of the quality of generated allocations. In addition, two known heuristics, namely Furthest-First (FF) and Clean-First (CF), and a new heuristic (MIX) are discussed and evaluated. The evaluation of the quality of the generated allocations is based on a suite of thirteen Fortran programs from the *fmm, Spec, and Spec95X* benchmark suites. An advanced compiler infrastructure (ILOC) was used to generated aggressively optimized, intermediate pseudo-register code for each benchmark program. Each local register allocation method was implemented, and its implementation was evaluated by simulating the

[⋆] 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.

[⋆⋆] Dept. of Comp. Sc., Rutgers University, New Brunswick, NJ 08903. E-mail: farach@cs.rutgers.edu.

[⋆⋆⋆] Dept. of Comp. Sc., Rutgers University, New Brunswick, NJ 08903. E-mail: uli@cs.rutgers.edu.

execution of the generated physical register allocation code on an architectures with N registers and an instruction set where loads and stores are C times as expensive as any other instruction. Experiments were performed for different values of N and C. The results show that only for large basic blocks the allocation quality gap between the different algorithms is significant. When basic blocks are large, the difference was up to 24%. Overall, the new heuristic (MIX) performed best as compared to the other heuristics, producing allocations within 1% of optimum. All heuristics had running times comparable to live variable analysis, or lower, i.e., were very reasonable. More work will be needed to evaluate the LRA algorithms in the context of more sophisticated global register allocators and source level transformations that potentially increase basic block sizes, including loop unrolling, inlining, and speculative execution (superblocks).

1 Introduction

Register allocation can substantially decrease the running time of compiled programs. Unfortunately, register allocation is a hard problem. To overcome the complexity of register allocation and to focus on its most crucial part, we propose a framework that breaks register allocation into a sequence of three distinct phases: *boundary allocation*, *local allocation*, and *register assignment*. Boundary allocation fixes the set of pseudo-registers that reside in physical registers at the beginning and at the end of each basic block. Local register allocation (LRA) determines the set of pseudo-registers that reside in physical registers at each step of a basic block, while previously chosen boundary conditions are respected. Register assignment maps allocated pseudo-registers to actual registers. The proposed framework has the advantage that (1) each phase is more manageable than the whole register allocation process, (2) it allows the integration of improved local register allocation heuristics with any global framework (e.g. Chaitin-Briggs), and (3) it isolates the register assignment phase, which was found to be the easiest phase of register allocation [PF96].

The canonical boundary allocation assumes that registers are empty at the beginning and at the end of a basic block. A more sophisticated boundary allocation extracts the boundary conditions from a state-of-the-art global register allocator. For example, a Chaitin-Briggs style allocator [BCT94] could determine the register contents at the basic block boundaries. Once boundary conditions are fixed, any LRA that respects the boundary conditions can be used. In this paper, we give an optimum algorithm as well as heuristics to perform LRA under any specified set of boundary conditions. Finally, allocated pseudo-registers will be assigned to physical registers. While register assignment could theoretically add significant overhead, it is in fact the easiest step of register allocation [PF96].

A first advantage of our framework is that each phase is more manageable than the whole process. Moreover, our approach allows us to replace the local component of any global allocation (e.g. Chaitin-Briggs) with improved LRA heuristics. Our framework puts emphasis on LRA. The basic block in the inner-most loop should be the most highly optimized part of the program [ASU86]. Moreover, pseudo-registers are easier to reuse in a basic block [PF96]. Finally, LRA algorithms are the first and fundamental step in demand-driven register allocation [PF96]. Note that we do not address the problem of instruction scheduling nor the interplay of register allocation with instruction scheduling. We assume that spill code has a fixed cost that is independent of the position where it is inserted. However, we will account for simple rematerializations.

This paper mostly focuses on the LRA phase. An optimum algorithm and three suboptimal heuristics for LRA are discussed. The optimum algorithm is based on a new integer programming formulation of LRA. The algorithm is a novel branch-and-bound method that exploits a special substructure of the integer program. The resulting allocator returns the best possible local allocation and took less than one minute in almost all our benchmarks. Using integer programming formulations for optimal solutions of NP-hard compiler problems has been discussed by a few researchers, in particular in the context of evaluating the quality of heuristic approaches [RGSL96,Kre97]. In addition to the optimal algorithm, we also propose a new heuristic, called MIX, for the same problem. MIX takes polynomial time, returned allocations that were always within 1% of the optimum, and was always more than ten times faster than the optimum. We also analyze previous heuristic for LRA. Previous heuristics always returned allocations that were worse than MIX's. We also measure the time taken by the heuristics, which is part of the total compilation time. If the heuristic running time is considered in isolation, MIX was substantially slower than previous heuristics. However, if we add the heuristic running time to live range analysis time, all heuristics ran in a comparable amount of time. In other words, the total allocation time was dominated by live range analysis rather than by the heuristic running time, and so no great difference was found in the overall running time. In conclusion, MIX returned better allocations than previous heuristics, while the total elapsed time was not seriously affected.

The paper is organized as follows. In §2, LRA is formally defined, followed by a review of relevant literature. In §3, we describe our experimental set-up. In §4, we give our integer programming formulation, a new branch-and-bound algorithm, and report experimental results for this algorithm. In §5, we define previous and new heuristics for LRA, prove worst-case bounds, and report and discuss experimental results. The paper concludes with a summary of its contributions.

2 Register Allocation

The problem of *Register Allocation* is to assign pseudo-registers to actual registers in a basic block so as to minimize the spill cost. Details are specified in the following section.

2.1 Local/Global Register Allocation

For the purpose of this paper, *register allocation* will operate on sequences of intermediate code instructions. Intermediate code instructions define and use *pseudo-registers*. Pseudo-registers contain temporary variables and constants. No aliasing between pseudo-registers is possible. We assume that a pseudo-register represents only one live range, and thus a pseudo-register is defined at most once. We also assume that each pseudo-register can be stored and retrieved in a designated memory location. We denote by $V = \{t_1, t_2, \ldots, t_M\}$ the set of pseudo-registers that appear in the intermediate code, and so $M = |V|$ is the number of distinct pseudo-registers that appear in the code. An example of intermediate code sequence is found in the leftmost column of Figure 1. In the figure, the instructions are `ADDI, SUB`, etc, and the pseudo-registers are `t0, t1, ..., t7`.

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	t5
	(read,{1,6})			t6	t1	t5
`ADD t1 t6 ⇒ t7`	(write,{7})	`ADD R2 R1 ⇒ R1`	1	t7	t1	t5
		total cost:	11			

Fig. 1. Example of optimal LRA with 3 registers. The first column gives a sequence of intermediate code instructions, the second column its representation in terms of pseudo-register usage, the third column the result of applying an optimum allocation, the fourth column gives the cost per operation assuming the spill cost is $C = 2$ and all other operations are unit cost, and the last columns give the register contents after a step has been executed. In the example, the set of live variables at the end of the segment is $L = \{\mathtt{t7}\}$ and `t2` contains the constant 4.

Register allocation maps pseudo-registers into a set of N actual registers. More precisely, a register allocation is a mapping that specifies which pseudo-registers reside in a register at each step of the program. Formally, a register allocation is a function $ra : V \times \mathbb{N} \to \{\mathsf{True}, \mathsf{False}\}$, where $\mathbb{N}$ is the set of natural integers, $ra(t_i, j) = \mathsf{True}$ if t_i is in a register at step j and $ra(t_i, j) = \mathsf{False}$ otherwise. The register allocation ra function cannot be any mapping $V \times \mathbb{N} \to \{\mathsf{True}, \mathsf{False}\}$, but satisfies the following two additional constraints imposed by register-register architectures:

– If a pseudo-register i is used by an instruction, then i occupies a register immediately before that instruction is executed.
– If a pseudo-register i is defined by an instruction, then i occupies a register immediately after that operation is executed.

There are two issues that are beyond the scope of this paper: instruction scheduling and register assignment. We now show how our functional definition ra of register allocation correctly excludes the two issues. In this paper, we do not consider instruction scheduling, which is the problem of rearranging the order of execution of instructions. Observe that the functional definition of ra excludes any instruction scheduling. We also distinguish between register allocation and *register assignment*: register allocation decides which pseudo-registers reside in actual registers and register assignment maps those pseudo-registers to particular physical registers. Our functional definition for register allocation keeps the two phases distinct. Register assignment could introduce an overhead because the assignment might have to be enforced by swap operations. In particular, at the end of a basic block, different register assignments must be made consistent by means of register swapping.

Since register allocation is, in our definition, a function, we can take its restriction to a subset of its domain. Specifically, we define the *boundary allocation* of a register allocation to be a function that specifies which pseudo-registers occupy actual registers at the beginning and at the end of each basic block. In other words, a boundary allocation fixes register contents at the boundaries of a basic block, and leaves undetermined register contents inside a basic block. We define *Local Register Allocation* (LRA) as the problem of assigning pseudo-registers to registers in a basic block once the boundary allocation is fixed. In other words, LRA is a register allocation for straight-line code that satisfies additional boundary conditions. We can view register allocation as formed by two components: the boundary allocation and the local allocation. We notice that, given a register allocation, its local allocation can be replaced with any other local allocation that satisfies the same boundary conditions.

We will now introduce some notation for LRA. Notice that each intermediate code instruction generates a sequence of reads and writes to pseudo-registers. For example, the instruction `SUB t1 t2 => t3` reads the pseudo-registers `t1` and `t2`, subtracts them, and writes the result into `t3`. We define the *static reference sequence* σ corresponding to the intermediate code to be a sequence of references, each of which is either a read or a write of a subset of V. For example, the instruction `SUB t1 t2 => t3` results in the sequence $((\mathsf{read}, \{\mathtt{t1}, \mathtt{t2}\}), (\mathsf{write}, \{\mathtt{t3}\}))$. Formally, a static reference sequence σ is a sequence of elements of $\{\mathsf{read}, \mathsf{write}\} \times V$. In Figure 1, we give a sequence of intermediate code instructions in the first column and the corresponding static reference sequence σ in the second column. Finally, boundary conditions can be cast in the static reference sequence σ by extending the sequence as follows:

– The first element of σ will be a read of the pseudo-registers that are in a register at the beginning of the basic block.

– The last element of σ will be a read of the pseudo-registers that are in a register at the end of the basic block.

Register allocations impose register contents at each step of program execution. Register allocations have to be enforced by loading the appropriate pseudo-registers into registers. Moreover, register allocations often require pseudo-registers to be stored back into their memory location. We now detail how a register allocation is enforced by load and store operations. If the register allocation specifies that a pseudo-register i is in a register at step j, but i is not in any register immediately before step j, then a load operation is inserted in the code to load i into a register. In turn, some other register i' would have to be evicted from the register file to make room for i. Define the set $S = \{\mathsf{clean}, \mathsf{dirty}\}$ as the set of pseudo-register states. We explain S as follows. If i' is clean, then it can be evicted without executing any store. If i' is dirty and live, then i' must be stored when evicted. If i' is clean, then either i' contains a constant or the value in the register is consistent with the value in the memory location assigned to i'. If i' is dirty, then i' does not contain a constant and the value in the registers is not consistent with the value in the location of i'. A pseudo-register i' is dirty when i' has been defined and its contents are maintained in a real register, but have not been stored to the memory location corresponding to i'. Notice that if a pseudo-register is not live, then we do not need to store it regardless of its being clean or dirty. Figure 1 gives an example of register allocation and assignment. The leftmost column gives a sequence of intermediate code, and we assume that the code is only a basic block in a larger program. The set of live variables at the end of the basic block is $L = \{\mathtt{t7}\}$. The second column reports the static reference sequence σ associated with the intermediate code. The third column gives the final code produced when a register allocation is enforced by (i) interspersing load and store operations in the code and (ii) rewriting pseudo-registers as the assigned actual registers. The last three columns give the register contents immediately after each instruction has been executed.

Since load and store operations are expensive to execute, an objective of register allocation is to find a feasible mapping of pseudo-registers into actual registers so as to minimize the total cost due to loads and stores. Specifically, we assume that load and store operations cost C times as much as any other operation. Notice that load immediates do not involve a memory access and cost $1/C$ as much as a load from memory. Since we assign different costs to different instructions, we will refer to a *weighted* instruction count. In this paper, we assume that the spill cost depends only on the number and type of inserted load and store operations, and not on the position they occupy. An example is given by the fourth column in Figure 1 where the costs of each operation are calculated when $C = 2$. The total cost of this sample allocation is 11, and it is the best possible allocation for this basic block. Different allocations yield different costs. In Figure 2, we report the same intermediate code as in Figure 1, but with a different allocation. Again, $C = 2$ and $L = \{\mathtt{t7}\}$, but this time the total cost is 14.

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			

Fig. 2. Example of FF LRA with 3 registers. As compared to the optimal LRA shown in Figure 1, the FF heuristic results in an overall cost of 14 vs. 11 for the optimal allocation.

2.2 Discussion and Related Work

We study register allocation as the problem of assigning pseudo-registers to registers so as to minimize the spill cost. Other approaches view register allocation as a two step process. Find the minimum number of registers needed to execute the given code without spilling (*register sufficiency*), and if there are less physical registers than needed, introduce spill code, and repeat the previous step. It can be shown that the register sufficiency problem is exactly equivalent to coloring a certain graph, which is called *interference graph* [Cha82]. Several authors put forth compelling arguments against such an approach:

– Some optimizations, like in-line expansion and loop unrolling, complicate the interference graph, and good colorings become hard to find. Hence, the solution to the register sufficiency problem will likely exceed the number of actual registers [HFG89].
– As soon as the the number of registers is exceeded, then spill code must be inserted. Unfortunately, it is hard to decide which pseudo-registers to spill and where to insert spill code [HFG89].
– Coloring addresses the problem of register assignment, but does not deal with the issue of deciding which pseudo-registers should actually be allocated to physical registers [PF96].

We add the following two observations in support of those arguments. First, the number of registers in the target architecture is fixed, while spill code is not. Therefore, registers are fixed resources and spill code corresponds to a cost. Minimizing a fixed resource is only a very indirect way to minimizing the actual spill cost. Moreover, register sufficiency (or, which is the same, graph coloring) is not only an NP-hard problem [GJ79], but also a problem that is very hard to approximate efficiently [Hoc97].

Another approach to register allocation is *demand-driven register allocation* [PF96]. Demand-driven allocation starts from an inner loop LRA and expands it to a global allocation. Our formulation of LRA and our heuristics can be used in demand-driven register allocation. In demand-driven register allocation, the boundary conditions specify that no pseudo-register resides in a register at the basic block boundary. The subsequent global allocation cleans inefficiencies introduced by such boundary conditions.

Several different models have been proposed to formalize LRA. Historically, the first models are simpler and disregard some feature, whereas subsequent model are more complete. Belady considered a model where there are no boundary conditions, no multiple references in one instructions, and stores can be executed for free [Bel66]. Subsequent work counted each load and store as a unit cost [HKMW66,Luc67,HFG89]. Briggs *et al.* give algorithms for global register allocation where each load and store from memory costs $C = 2$ times load immediates or any other operations [BCT94]. Such cost model allows us to keep track of simple rematerializations. In this paper, we also consider the case when one instructions can generate multiple references. Multiple references arise commonly in actual code. For example, the instruction `SUB t1 t2 => t3` requires that both `t1` and `t2` be simultaneously present in registers before the instruction could be executed. Such feature was first considered in [HFG89]. The simplest LRA problem is Belady's, where stores have no cost, loads have unit cost, and there are no multiple references. Belady gave a polynomial-time algorithm for that LRA problem. Subsequently, it was found that, if the cost of a store were zero, the problem would be polynomially solvable even in the presence of different load costs, boundary conditions, and multiple references [FL98], but LRA is NP-hard as soon as stores are counted as having positive cost [FL98]. In conclusion, the hardness of LRA is due to the presence of store operations and not on several other features mentioned above.

We define boundary allocation as a functional restriction of global allocation, and local register allocation as a basic block allocation that respects boundary conditions. To the best of our knowledge, no such formulation had previously been given. The division of register allocation into boundary and local allocations makes possible to integrate an LRA algorithm with any global allocation by simply replacing its local portion. To the best of our knowledge, register allocation and register assignment have been considered as two distinct phases in all previous papers on LRA [Bea74,HKMW66,Luc67,HFG89]. Register allocation is more manageable if it is divided into allocation and assignment. Register assignment could conceivably cause the introduction of a large number of register swap operations. Actually, Proebsting *et al.* report that register assignment could almost always be enforced without swaps, and conclude that more emphasis should be placed on allocation rather than on assignment [PF96].

While we propose a three-phase approach to register allocation, some previous work takes a more compartmentalized approach to register allocation. The register set is partitioned into two sets: one to be used only by pseudo-register live in the basic block and the other only by global pseudo-registers. The former set of register is intended to be used for global allocation, and the latter for local allocation. It will be clear that all LRA algorithms in this paper would work correctly in this framework as well, but we give measurements only for our three-phase approach.

A few heuristics have been proposed for LRA. The oldest is Belady's *Furthest-First* (FF): if no register is empty, evict the pseudo-register that is requested furthest in the future [Bac81]. FF is optimum in the simple Belady's model, which assumes that stores are executed at no cost and that there are no boundary conditions [HKMW66]. FF is also optimum when there multiple references in one step [Lib98], and is a $2C$-approximation algorithm for LRA even in the presence of paid stores and boundary conditions [FL98]. If stores have a positive cost, FF is not necessarily optimal : Figure 2 gives an FF allocation of cost 14, whereas the optimum is in Figure 1 and costs 11. FF's major problems are that it does not take into account the cost of storing and the effects of rematerialization. In the figure, pseudo-register `t1` is stored and later reloaded at a total cost of 4 even though `t2` contains a constant and so it could be evicted for free and reloaded at a unit cost. In this case, FF failed to detect that `t2` could be rematerialized at a small cost. Another problem arises because FF does not distinguish between clean and dirty registers. In order to fix the latter problem, an heuristic called *Clean-First* (CF) has been introduced [FL88]. CF evicts a clean pseudo-register that is requested furthest in the future. If no clean pseudo-register exists, CF evicts a dirty pseudo-register that is requested furthest in the future. We will conduct a theoretical and experimental analysis of CF in §5.2. Finally, Farach *et al.* introduced an algorithm $\mathcal{W}$ that is provably never worse than twice the optimum [FL98].

Since LRA is NP-hard, no polynomial-time algorithm can be reasonably expected to return an optimum solution in all cases. In particular, all the heuristics above fail to return the optimum solution in some cases. As opposed to heuristics, an optimum algorithm is proposed in [HKMW66,Luc67,Ken72]. Such optimum algorithm works only when $C = 1$ and takes exponential time and space in the worst case. The optimum algorithm failed to terminate on a few benchmarks due to lack of memory space; those tests were executed as late as 1989 [HFG89].

3 Experimental Set-up

We performed experiments with ILOC, the Intermediate Language for Optimizing Compilers developed at Rice University[1]. We used several ILOC programs 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,

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

and dead code elimination. The resulting ILOC code is similar to that in the first column of tables 1 and 2: it is a sequence of intermediate code instructions that operate on an infinite set of pseudo-registers[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 assumed we had N integer registers and N double precision registers. In our experiments, floating point operations are assumed to cost as much as integer ones. We remark that this is only a measurement choice, and that all algorithms in this paper would work if floating point operations were attributed a different cost than integer ones. We performed experiments for a number of registers ranging as $N = 16, 32, 64$, and spill cost $C = 2, 4, 8, 16$. We used a SUN UltraSparc1 (143MHz/64Mb) for algorithm timing experiments. All our allocators were written in C, compiled with `gcc -O3`, and take as input the ILOC programs above. We run our allocators to obtain ILOC code that uses at most N physical registers. We also kept track of the *weighted instruction count* of each basic block, that is, we counted the number of instructions in each basic block weighted by a factor of C if they involve memory accesses, as described above. Then, we transformed that ILOC code into C programs with ILOC's i2c tool in order to simulate its execution. In the resulting C program was instrumented to count the number of times each basic block was executed. We ran the resulting C code to obtain a *dynamic weighted instruction count*. The count is dynamic because it is collected by a simulation of code execution and it is weighted because spill code count is multiplied by a factor of C, as described above:

$$\sum_{\text{basic block } \mathcal{B}} (\text{number of times } \mathcal{B} \text{ was executed}) \times (\text{weighted instruction count for } \mathcal{B}) \; .$$

Table 1 describes the static reference sequences used in the experiments. Column 1 gives the name of the benchmark suite and column 2 the program name. Column 3, 4, and 5 report data for the static reference sequences of double precision variables. Column 3 gives the number of basic blocks where there is at least one live double precision variable. Column 4 gives the average length of the corresponding reference sequences. Column 5 gives the average number of distinct pseudo-registers referenced in a basic block. Finally, column 6, 7, and 8 report the same quantities for integer sequences. A measure of the size of the double (integer) LRA problem associated with one benchmark can be obtained by the product (number of double (integer) blocks) $\times$ (average double (integer) length).

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. The program tomcatv has long integer sequences on average, but not long double sequence. Some optimization passes (e.g. loop unrolling) produce long basic block, but no

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

benchmark	prg	Double			Integer		
		blcks	avg len	avg var	blcks	avg len	avg var
fmm	fmin	56	22.93	20	54	4.46	3.70
	rkf45	129	10.85	8.78	132	26.51	23.07
	seval	37	7.81	5	43	19.05	14.44
	solve	96	4.88	3.85	110	27.79	24.14
	svd	214	7.96	6.25	226	38.74	34.77
	urand	10	6.1	4.1	13	18.38	12.62
	zeroin	31	20.10	16.10	30	5.7	4.7
spec	doduc	1898	16.66	12.43	1998	25.59	21.12
	fpppp	433	57.95	44.56	467	60.91	54.47
	matrix300	7	2.57	1.71	62	23.11	17.58
	tomcatv	72	11.68	9.67	73	73.48	68.14
spec95X	applu	493	16.26	10.82	679	55.89	47.32
	wave5X	6444	10.92	7.54	7006	53.25	45.23

Table 1. Characteristics of static reference sequences from optimized code. For each benchmark suite and program, the table gives the number of basic block with references to double (integer) variables, the average number of references to double (integer) variables per basic block, and the average number of distinct double (integer) variables in each block.

such optimization is available in the ILOC system and none has been applied to our benchmarks.

We conducted experiments to compare the FF and CF heuristics, our new heuristic called MIX and a new optimum algorithm. We were mostly interested in two quantities: the *quality of the allocation* each algorithm returns and the *speed* of the algorithm itself. Our purpose was to identify possible trade-offs between compiler speed and the speed of generated code. We measured allocation quality by means of a dynamic weighted instruction count, which we described above. The speed of the allocator itself was measured as follows. We inserted `rusage` routine calls before and after each allocator was actually called. Then, we summed up the user and system time elapsed between the two rusage calls. In this way, we counted only the time needed to execute the LRA allocators, and we disregarded the time for reading the input ILOC files, constructing data structures that represent the associated LRA problems, and performing live range analysis. We also inserted rusage code to estimate the time spent for live range analysis. The live range analysis time was always estimated separately from the allocation time.

LRA performance can be measured only after boundary allocations are fixed. A possible choice is to extract the boundary allocation from a state-of-the-art global allocator. Unfortunately, no such allocator is currently available in the ILOC infrastructure distribution. A *canonical* boundary allocator assumes that all registers are empty at the beginning and end of each basic block [ASU86]. We used this boundary allocation. However, this is only an experimental choice, and that all discussed algorithms would work for any other boundary allocator. More work will be needed to evaluate the different LRA algorithms for other boundary allocators.

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.

4.1 Integer Programming Formulation

Our integer programming formulation is inspired by a recent integer multicommodity commodity flow formulation [FL98]. However, the integer programming formulation is different in some respects from the multicommodity formulation, and we will define it below from first principles for the sake of clarity.

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. 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 C 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 live at step j or if i is live, 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 live 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 live at the entry of the basic block, or the steps after the last reference to i if i is live 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. 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 C 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 live 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 live at step j and that cannot reside in 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

$$\begin{aligned}
&\min \sum_{i=1}^{M} c_i x_i + \sum_{i=1}^{M}\sum_{k=1}^{K(i)} c_{ik} y_{ik} && && (1.1)\\
&\text{s.t. } x_i \geq y_{ik} && i = 1, 2, \ldots, M; k = 1, 2, \ldots, K(i) && (1.2)\\
&\sum_{(i,k)\in K_j} y_{ik} \geq \rho_j && j = 1, 2, \ldots, |\sigma| && (1.3)\\
&x_i, y_{ik} \in \{0, 1\} && i = 1, 2, \ldots, M; k = 1, 2, \ldots, K(i) && (1.4)
\end{aligned}$$

Figure 3 gives an example of formulation (1). In the example, we assume that there are only $N = 2$ actual registers, that pseudo-register 3 contains an immediate and it is the only pseudo-register live at the beginning of the basic block, and that pseudo-registers 1 and 2 are live at the end of the basic block. Pseudo-register 3 can be evicted either during the first two requests, or during the fourth, or after the fifth. However, pseudo-register 3 can be evicted without storing it after the fifth request, so there is no cost in keeping it out of registers. We can safely assume that 3 does not reside in a register after step 5. Therefore, we associate with pseudo-register 3 only two value ranges and the corresponding decision variables are y_{31} and y_{32}, where y_{31} represents the decision to evict 3 during the first two requests and y_{32} during the fourth. The set of variables for the other pseudo-registers is defined analogously.

4.2 Branch-and-Bound

We now describe a branch-and-bound algorithm that efficiently solves the integer program (1). Let $\mathcal{N}$ be the part of the constraint matrix corresponding to the constraints (1.3). Figure 3 gives an example of such matrix. First, notice that $\mathcal{N}$ is a binary matrix, that is, $\mathcal{N}$'s entries are either 0 or 1. The matrix in the example has the following property: in each column, the ones appear in consecutive rows. Actually, such property holds in general for any matrix $\mathcal{N}$ obtained from (1.3), as it is proven by the following proposition.

Proposition 1. *The matrix* $\mathcal{N}$ *has the* consecutive ones property in the columns*, that is, in each column, the ones appear in consecutive positions.*

Proof. Value ranges extend over intervals of the basic block. Therefore, the corresponding y_{ik}'s will appear in consecutive constraints in (1.3). Since $\mathcal{N}$ has a 1 where y_{ik} appears and a 0 otherwise, the 1's will appear in consecutive rows in $\mathcal{N}$.

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

$$
\begin{array}{lcl}
\underline{\quad\sigma\quad} & & \min\ Cx_1 + Cx_2 + Cy_{11} + Cy_{21} + y_{31} + y_{32} \\
(\mathsf{write},\{1\}) & \longmapsto & \text{s.t. } y_{31} \geq 0 \\
(\mathsf{write},\{2\}) & \longmapsto & \quad y_{11} + y_{31} \geq 0 \\
(\mathsf{read},\{3\}) & \longmapsto & \quad y_{11} + y_{21} \geq 1 \\
(\mathsf{read},\{2\}) & \longmapsto & \quad y_{11} + y_{32} \geq 1 \\
(\mathsf{read},\{3\}) & \longmapsto & \quad y_{11} + y_{22} \geq 1 \\
(\mathsf{read},\{1\}) & \longmapsto & \quad y_{22} \geq 0 \\
 & & \quad x_1 \geq y_{11} \\
 & & \quad x_2 \geq y_{21} \\
 & & \quad x_2 \geq y_{22} \\
 & & \quad x_3 \geq y_{31} \\
 & & \quad x_3 \geq y_{32} \\
 & & \quad x_1, x_2, x_3, y_{11}, y_{21}, y_{22}, y_{31}, y_{32} \in \{0, 1\}
\end{array}
\quad \Longrightarrow \mathcal{N} = \begin{pmatrix} 0\,0\,0\,1\,0 \\ 1\,0\,0\,1\,0 \\ 1\,1\,0\,0\,0 \\ 1\,0\,0\,0\,1 \\ 1\,0\,1\,0\,0 \\ 0\,0\,1\,0\,0 \end{pmatrix}
$$

Fig. 3. Sample formulation of LRA as an integer program. The first column gives a representation of σ as read or write references. The second column gives the corresponding integer program. The matrix $\mathcal{N}$ is the matrix corresponding to constraints (1.3).

Consider now the following problem, where b and c are arbitrary vectors and e is the vector of all ones:

$$
\begin{array}{rl}
\min c^T y & (2.1) \\
\text{s.t. } \mathcal{N} y \geq b & (2.2) \\
0 \leq y \leq e & (2.3)
\end{array} .
$$

Since $\mathcal{N}$ has the consecutive ones property, the program (2) can be solved in polynomial time with a *network flow* algorithm and has always an integer solution [AMO93]. We will next show how to exploit this fact to efficiently solve the original problem (1).

First, we will argue that the constraints $y_{ik} \in \{0, 1\}$ can be replaced by the weaker constraints $0 \leq y_{ik} \leq 1$ ($i = 1, 2, \ldots, M$ and $k = 1, 2, \ldots, K(i)$). A constraint substitution of this type is called a *relaxation of integrality constraints*. The significance of such relaxation is the following: if y_{ik} is binary, then we need to try both cases $y_{ik} = 0$ and $y_{ik} = 1$. As a consequence, the number of nodes explored by a branch-and-bound algorithm increases exponentially in the number of y_{ik}'s. However, if we only impose that $0 \leq y_{ik} \leq 1$, no such branching is necessary, and the number of explored nodes decreases. We argue that the relaxation is valid in (1) for the following reason. Once all x_i's have been fixed to a binary value $\bar{x}_i \in \{0,1\}$, then problem (1) becomes:

$$\begin{aligned}
&\min \sum_{i=1}^{M} \sum_{k=1}^{K(i)} c_{ik} y_{ik} && (3.1)\\
&\text{s.t.} \sum_{(i,k)\in K_j} y_{ik} \geq \rho_j \; j = 1,2,\ldots,|\sigma| && (3.2)\\
&y_{ik} \leq \bar{x}_i \qquad i = 1,2,\ldots,M; k = 1,2,\ldots,K(i) && (3.3)\\
&y_{ik} \in \{0,1\} \qquad i = 1,2,\ldots,M; k = 1,2,\ldots,K(i) && (3.4)
\end{aligned}$$

Suppose that we now relax the integrality constraints (1.4) into $0 \leq y_{ik} \leq 1$ ($i = 1,2,\ldots,M; k = 1,2,\ldots,K(i)$). Now, problem (3) has the form (2), and so y_{ik} will be binary in an optimum solution without loss of generality. In other words, we are guaranteed integrality of the y_{ik}'s even if no such integrality condition is explicitly imposed. The relaxation of integrality constraints caused a dramatic reduction of the number of integer variables in our benchmarks. The solid bars in Figures 4 and 5 give the percentage of decision variables that are real (that is, not integer) for each benchmark in our suite. The number of real variables was always more than 95% and it was almost 100% for several benchmarks. In other words, the number of integer variables is less than 5% and often almost 0% of the number of all decision variables. In conclusion, the special structure of the $\mathcal{N}$ implies a special structure (2) of a large subproblem of (1), which in turn implies a reduction of the number of integer variables in the problem formulation. Finally, the reduction of the number of integer decision variables diminishes the amount of branching needed by a branch-and-bound algorithm.

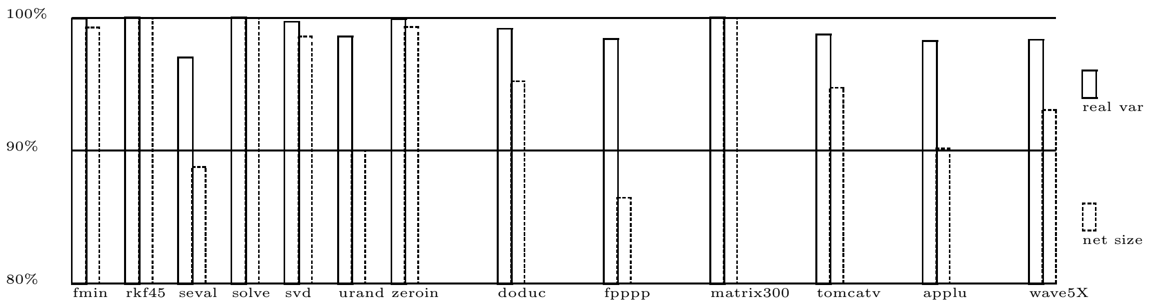

Fig. 4. Characteristics of (1): double reference sequences

There is a second and more subtle change to the branch-and-bound algorithm that allows us to take advantage of the special structure of (2). First, we claim that the constraints (1.3) constitute most of the constraints in (1). The dashed bars in Figure 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

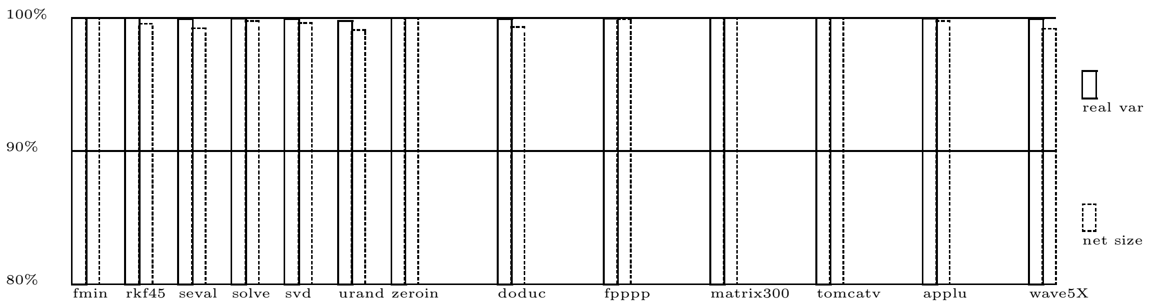

Fig. 5. Characteristics of (1): integer reference sequences

efficiently solved as a network flow problem with side constraints [AMO93]. An essential component to solve such problems is a routine to determine the minimum cost flow in a directed graph. Conversely, any min cost flow algorithm can be employed to solve the initial relaxation of (1). There are several of algorithms to solve network flow problems [AMO93], and there is no reason to choose one over another except for its performance. The worst-case bounds of min cost flow algorithms depend on the characteristics of the network flow instance, and in particular on the number of rows and columns in $\mathcal{N}$. Notice that the number of columns is linear in the number of rows because at most a constant number of value ranges can start at each reference. The best worst-case time bound for this type of problems is given by the *successive shortest path* algorithm. Another alternative that is more commonly used in practice is the *network symplex* algorithm. In some preliminary experiments, we tried the successive shortest path algorithm, but we quickly abandoned it because it was overmatched by the network symplex. All results in this paper are obtained by means of the network symplex algorithm. We remark again that there are several network flow algorithms, and that a theoretical and experimental discussion of all those algorithms is beyond the scope of this paper. We also remark that the minimum cost network flow problem is currently a very active area of research, and it is possible that more efficient algorithms will be discovered in the near future [AMO93]. On the whole, the branch-and-bound algorithm was implemented with calls to the CPLEX[3] callable library. The initial relaxation algorithms was set to the network optimizer, and it solves the initial relaxation as discussed above.

4.3 Experimental Results and Discussion

Table 2 reports the running time of the optimum in seconds for $C = 2$. Experimental results are reported for several values of N in order to ascertain the robustness of our procedure. As we discussed above, the reported times do not include I/O, live range analysis, and the time to set-up the problem matrices.

The optimum takes always less than one minute except for one benchmark and two choices of N. We can compare the branch-and-bound time with the

[3] CPLEX is a trademark of CPLEX Optimization.

benchmark	prg	N							
		2	4	8	16	32	64	128	256
fmm	fmin	0.20	0.18	0.17	0.16	0.15	0.15	0.15	0.15
	rkf45	0.50	0.47	0.44	0.41	0.37	0.37	0.37	0.37
	seval	0.16	0.15	0.13	0.13	0.12	0.12	0.12	0.12
	solve	0.40	0.38	0.36	0.34	0.31	0.31	0.31	0.31
	svd	1.01	0.95	0.90	0.84	0.73	0.70	0.70	0.70
	urand	0.05	0.04	0.04	0.04	0.03	0.04	0.03	0.03
	zeroin	0.11	0.10	0.09	0.09	0.08	0.08	0.08	0.08
spec	doduc	8.99	8.52	8.06	7.48	6.97	6.76	6.28	6.28
	fpppp	23.60	26.10	1490	139	15.9	20.8	11.2	8.42
	matrix300	0.14	0.13	0.12	0.10	0.10	0.10	0.10	0.10
	tomcatv	0.48	0.46	0.45	0.42	0.37	0.29	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 solution run-times in seconds

size of the benchmark (which has been defined in §3 on the basis of table 1). Broadly speaking, we observe that in most cases the optimum takes longer on larger programs. The branch-and-bound running time decreases as N increases for all programs but fpppp. An intuitive explanation is that when more registers are available, most allocation problems should become easier.

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.

5 Heuristics

5.1 Heuristic Definition

We experimented with three heuristics. Each heuristic specifies which pseudo-register is to be evicted if no empty register is available. *Furthest-First* (FF) determines the set S of pseudo-registers that are requested furthest in the future, and evicts a clean pseudo-register in S. If all pseudo-register in S are dirty, an arbitrary element in S is evicted [Bac81,FL98]. *Clean-First (CF)* evicts a clean pseudo-register that is used furthest in the future. If no clean pseudo-register exists, a dirty pseudo-register is evicted that is requested furthest in the future [FL88]. The heuristic *MIX* is based on the algorithm $\mathcal{W}$ [FL98]. While $\mathcal{W}$ is detailed in [FL98], we will also report it here for the sake of completeness.

The algorithm $\mathcal{W}$ has an intuitive explanation in terms of the program (1). Recall that (1) consists of two parts: $\mathcal{N}$, which has a very strong structure, and the side constraints. Therefore, one could wish that the side constraints disappeared. Unfortunately, it is not possible to simply delete a set of constraints without altering the problem at hand. However, it is possible to perform the following three step procedure, called *Lagrangian relaxation*: (1) Insert a penalty in the objective function with the property that the penalty increases if the side constraints are violated, (2) remove all side constraints, and (3) solve the resulting problem. The idea is that, if the penalty is big enough, the side constraints will not be violated even though they are not explicitly impsed. Finally, the resulting problem is defined only by $\mathcal{N}$, and so it can be solved in polynomial-time as a network flow problem. The penalty function is to be chosen appropriately: formulas can be found in [FL98] and are omitted from the present paper.

We modified $\mathcal{W}$ in two respects in order to make it into the faster algorithm MIX. In the first place, MIX will invoke $\mathcal{W}$ only if the static reference trace is not longer than a certain parameter `length_max`, and it will use FF otherwise. We now explain our reason for setting an upper bound on the basic block length. The FF algorithm runs in (almost) linear time, whereas minimum cost network flow algorithms are superlinear. Therefore, FF becomes faster and faster than any min cost flow algorithm as the size of the problem increases. Consequently, FF is much faster than $\mathcal{W}$ for big basic blocks. We reduce MIX's running time by invoking $\mathcal{W}$ only on basic blocks that are not too long. The right choice of `length_max` depends on how much compilation time is valued against the performance of the compiled code. We experimented with four values of `length_max`, specifically `length_max` $= \infty, 1500, 1000, 0$. When `length_max` $= \infty$, we run $\mathcal{W}$ without modifications, when `length_max` $= 0$ we run FF, and the other two cases are an intermediate algorithm between FF and $\mathcal{W}$. Our findings are as follows. When `length_max` $= +\infty$ ($\mathcal{W}$), MIX was roughly 100 times slower than when `length_max` $= 0$ (FF). When `length_max` $= 1000$, MIX did not yield allocations significantly better than FF. Finally, we found that `length_max` $= 1500$ strikes a good balance between the two extremes. In particular, the performance of the compiled code for `length_max` $= 1500$ was on average no worse than .1% than that for `length_max` $= +\infty$ ($\mathcal{W}$), while the algorithm itself was no slower than 3 times `length_max` $= 0$ (FF). In the rest of the paper, we will give results only for `length_max` $= 1500$.

Another idea we implemented is the following. If we could detect that in a basic block there is little chance for optimization, then a simple and fast algorithm should be preferred to a slow algorithm. It is only when we detect the opportunity for optimizations that more complex algorithms can be justified. Define the *maximum dirty pressure* as the number of dirty variables that are simultaneously live in a basic block. If the maximum dirty pressure is zero (and there is no load immediate), then FF is optimum. In this case, no algorithm can outperform FF, and so complicate algorithms are not useful. We took this observation to the following consequence: MIX invokes $\mathcal{W}$ only when the maximum dirty pressure is at least a certain value, and will follow FF otherwise. Specifically, MIX is implemented to run $\mathcal{W}$ only when the maximum dirty pressure is bigger than 150, and it will invoke FF otherwise.

5.2 Worst-Case Analysis

The spill cost of FF is never $2C$ times worse than the optimum, and $\mathcal{W}$'s is never more than twice the optimum [FL98]. Consequently, MIX is never worse than $2C$ the optimum when `length_max` < 1500 or the maximum dirty pressure is less than 150, and it is never worse than twice the optimum in all other cases. By contrast, we will now show that CF does not have a performance guarantee in the worst case. 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 cost greater than $2Cn$. In this case, the optimum strategy is FF that would evict a dirty variable, paying a cost of $3C$. Take n arbitrarily large and, while n grows, the cost ratio of CF over FF tends to infinity.

5.3 Experimental Results

The experimental results can be divided into two broad categories: results for the benchmarks fpppp and tomcatv and results for all other benchmarks. First, we report results for all other benchmarks, and then we give our findings for fpppp and tomcatv. On all benchmarks except fpppp and tomcatv, no heuristic was worse than .5% of the optimum, and, on those instances, FF and MIX took nearly the same time and were faster than CF. Such findings hold with minimal variations for all benchmarks except fpppp and tomcatv. We turn now to fpppp and tomcatv and report our results in the following tables. The time rows represents 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, as discussed above. The cost rows report the total weighted dynamic instruction count in thousands. We make the following observations on the entries in the tables. FF, CF, and MIX were always at least ten times faster than the branch-and-bound algorithm. FF and CF are independent of C, and they took the same time for all values of C. In most cases, an increase of C caused MIX to slow down, but in some cases MIX was actually faster for a larger value of C. CF produced allocations that were as much as 25% worse than FF and MIX. FF produced allocations that were up to 4% worse than the optimum for $N = 32$. Moreover, FF and CF produced quite different allocations even in the case $C = 2$, and the gap increased with C.

MIX produced allocations that are up to .9% worse than the optimum and never took much more than 1 second. MIX cost was always better than FF's or CF's in these experiments. The gap between MIX cost and the other heuristics grows larger with C. The time taken by MIX is often less than the time taken by live-variable analysis, a necessary prerequisite to almost all register allocators. The time for live range analysis was 1.58s on fpppp, .01s on tomcatv.

				FF		**MIX**		**CF**	
C	**prg**		**OPT**		%		%		%
2	fpppp	time	145.3	0.3149	0.2167%	0.5205	0.3582%	0.6929	0.4769%
		cost	193357	194858	0.776%	193713	0.184%	209013	8.1%
	tomcatv	time	0.9069	0.04426	4.88%	0.07608	8.388%	0.09915	10.93%
		cost	413779	416380	0.629%	413779	0%	421582	1.89%
4	fpppp	time	151.2	0.3186	0.2107%	0.5218	0.3452%	0.6957	0.4601%
		cost	251322	254325	1.19%	252259	0.373%	282635	12.5%
	tomcatv	time	0.9138	0.04329	4.738%	0.07477	8.182%	0.1055	11.55%
		cost	484108	489310	1.07%	484108	0%	499714	3.22%
8	fpppp	time	140.6	0.3224	0.2293%	0.815	0.5797%	0.709	0.5044%
		cost	367253	373258	1.64%	369222	0.536%	429879	17.1%
	tomcatv	time	0.9064	0.04328	4.775%	0.0747	8.242%	0.09689	10.69%
		cost	624766	635170	1.67%	624766	0%	655978	5%
16	fpppp	time	144.4	0.3198	0.2215%	0.7655	0.5301%	0.7042	0.4877%
		cost	599115	611125	2%	601868	0.459%	724367	20.9%
	tomcatv	time	0.8915	0.04261	4.779%	0.07321	8.212%	0.09739	10.92%
		cost	906082	926890	2.3%	906082	0%	968506	6.89%

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

				FF		**MIX**		**CF**	
C	**prg**		**OPT**		%		%		%
2	fpppp	time	19.07	0.3056	1.602%	0.5107	2.678%	0.686	3.597%
		cost	167076	169013	1.16%	167633	0.333%	178557	6.87%
	tomcatv	time	0.8496	0.04172	4.911%	0.07345	8.645%	0.09845	11.59%
		cost	343450	343450	0%	343450	0%	343450	0%
4	fpppp	time	19.22	0.3101	1.613%	0.5134	2.671%	0.6907	3.593%
		cost	198815	202689	1.95%	199928	0.56%	221776	11.5%
	tomcatv	time	0.8568	0.04138	4.829%	0.07309	8.53%	0.1048	12.23%
		cost	343450	343450	0%	343450	0%	343450	0%
8	fpppp	time	18.95	0.3101	1.636%	0.9179	4.844%	0.6943	3.664%
		cost	262293	270040	2.95%	264118	0.696%	308215	17.5%
	tomcatv	time	0.676	0.02934	4.341%	0.05801	8.581%	0.06277	9.285%
		cost	343450	343450	0%	343450	0%	343450	0%
16	fpppp	time	19.73	0.3096	1.57%	0.8418	4.267%	0.6954	3.525%
		cost	389248	404743	3.98%	392802	0.913%	481093	23.6%
	tomcatv	time	0.8336	0.04092	4.909%	0.07178	8.61%	0.09635	11.56%
		cost	343450	343450	0%	343450	0%	343450	0%

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

C	**prg**		**OPT**	**FF**		**MIX**		**CF**	
					%		%		%
2	fpppp	time	23.96	0.296	1.235%	0.5014	2.093%	0.6693	2.794%
		cost	150466	151062	0.396%	150502	0.0239%	158519	5.35%
	tomcatv	time	0.773	0.03795	4.91%	0.06959	9.002%	0.09394	12.15%
		cost	343450	343450	0%	343450	0%	343450	0%
4	fpppp	time	24.77	0.2977	1.202%	0.501	2.023%	0.6736	2.72%
		cost	165595	166787	0.72%	165667	0.0434%	181701	9.73%
	tomcatv	time	0.7799	0.03731	4.785%	0.06917	8.869%	0.1008	12.92%
		cost	343450	343450	0%	343450	0%	343450	0%
8	fpppp	time	24.23	0.2993	1.235%	0.9717	4.01%	0.6863	2.832%
		cost	195853	198236	1.22%	195949	0.049%	228065	16.4%
	tomcatv	time	0.7713	0.0372	4.823%	0.06873	8.911%	0.09243	11.98%
		cost	343450	343450	0%	343450	0%	343450	0%
16	fpppp	time	25.2	0.3001	1.191%	0.9845	3.907%	0.6822	2.707%
		cost	256368	261135	1.86%	256752	0.15%	320792	25.1%
	tomcatv	time	0.7578	0.0373	4.922%	0.06758	8.918%	0.09237	12.19%
		cost	343450	343450	0%	343450	0%	343450	0%

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

5.4 Discussion

We found an interesting correlation between the performance of heuristics and the average length of basic blocks (average basic block length is found in table 1). In general, programs with long basic block should be harder to solve due to the NP-hardness of the problem. Indeed, fpppp has the longest average basic block length and the largest gap between the optimum and the heuristics. The program tomcatv has long integer sequences, but not long double sequence. Correspondingly, we observe that heuristics performed better on tomcatv than on fpppp. However, tomcatv still gave rise to substantial performance differences when $N = 16$ and C is large. All other benchmarks have shorter basic blocks, and no significant discrepancy between optimum and heuristics was detected. Long basic blocks are produced by some optimization passes (e.g., loop unrolling, superblock scheduling), but, unfortunately, none is available to us. Since we found that a large average basic block length dramatically increased the difference between heuristics and optimum on our benchmarks, it is easy to conjecture that length-increasing optimizations would widen the heuristic-optimum gap to much more substantial figures than those above for most programs. We notice that the *average static* sequence length affected the hardness of LRA instances. In other words, when we take the average length, we do not weigh the sequence length by the number of times basic blocks were executed. It is not at all obvious *a priori* why a static average should be related to a dynamic instruction count, but we found that this was indeed the case in our benchmarks.

Integer programming is useful for several hard compilation problems; see [Kre97] for a survey. Network flow techniques have been used for intraprocedural register assignment [KF96]. Lagrangian relaxation was proposed by the famous mathematician Lagrange in the eighteen century in the context of nonlinear optimization, and was introduced into discrete optimization by Held and Karp [HK70]. The gist of Lagrangian relaxation is that, given a complicate problem with many types of constraints, hard constraints should be moved into the objective function so that the remaining problem is easy to solve. In compiler optimization, there are examples where several types of constraints are imposed. If only one type of constraints existed, the problem would be easy, but multiple types of constraints complicate the problem solution. It would be interesting to understand whether Lagrangian relaxation could be exploited to solve those problems efficiently.

Finally, we notice that live range analysis takes much longer than any LRA heuristic. As a consequence, live range analysis dominates the time required to perform LRA.

6 Conclusions

In this paper, we have proposed an approach to register allocation that divides an allocator into three successive phases: boundary, LRA, and register assignment. We have studied the problem of local register allocation in the context of four different algorithms. We have given an optimum algorithm and a new heuristic called MIX. The optimum algorithm is based on an integer programming formulation and it was reasonably fast. With the exception of one benchmark program, the optimal solution was always computed in less than a minute. The heuristic MIX combines two previous algorithms FF and $\mathcal{W}$ (the algorithm $\mathcal{W}$ is, in turn, based on a Lagrangian relaxation of the integer program), and returned allocations that were within 1% of the optimum. MIX outperformed FF more significantly when C is larger. All three heuristics, FF, MIX, and CF, computed solutions in less time than typically required for the live variable analysis step within the compiler. For short basic block, the qualities of the generated allocations were comparable across the three heuristics. However, for larger basic block sizes, our findings suggest that MIX should be the best choice for LRA on optimized code, especially when C is large.

More work will be needed to evaluate the LRA algorithms and our three-phase framework in the context of different global register allocators. In addition, since the results suggest that the performance gap between the algorithms will increase with increasing basic block sizes, we are planning to investigate the impact of source level transformations that potentially increase basic block sizes, including loop unrolling, inlining, and speculative execution (superblocks).

Acknowledgments

We gratefully acknowledge helpful conversations with Bill Pugh and 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.

[Bac81] John Backus. The history of FORTRAN I, II, and III. In Richard Wexelblat, editor, *History of Programming Languages*, ACM Monographs Series, pages 25–45. Academic Press, New York, 1981.

[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.

[Bea74] J. C. Beatty. Register assignment algorithm for generation of highly optimized code. *IBM J. Res. Develop.*, 18:20–39, January 1974.

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

[Cha82] G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pages 98–105, 1982.

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

[FL98] Martin Farach and Vincenzo Liberatore. On local register allocation. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 564–573, 1998.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness.* Freeman, San Francisco, 1979.

[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.

[HK70] M. Held and R. Karp. The traveling salesman problem and minimum spanning trees. *Operations Research*, 18:1138–1162, 1970.

[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.

[Hoc97] Dorit S. Hochbaum, editor. *Approximation Algorithms for NP-hard Problems.* PWS Publishing Company, Boston, 1997.

[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.

[KF96] Steven M. Kurlander and Charles N. Fischer. Minimum cost interprocedural register allocation. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1996.

[Kre97] Ulrich Kremer. Optimal and near-optimal solutions for hard compilation problems. *Parallel Processing Letters*, 7(2):371–378, 1997.

[Lib98] Vincenzo Liberatore. Uniform multipaging reduces to paging. *Information Processing Letters*, 67:9–12, 1998.

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

[PF96] Todd A. Proebsting and Charles N. Fischer. Demand-driven register allocation. *ACM Transactions on Programming Languages and Systems*, 18(6):683–710, November 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.