Journal of Algorithms **37**, 37–65 (2000)
doi:10.1006/jagm.2000.1095, available online at http://www.idealibrary.com on IDEAL®

On Local Register Allocation[1]

Martin Farach-Colton[2]

Department of Computer Science, Hill Center, Rutgers University, New Brunswick, New Jersey 08903
E-mail: farach@cs.rutgers.com

and

Vincenzo Liberatore[3]

UMIACS, A. V. Williams Bldg., University of Maryland, College Park, Maryland 20742
E-mail: vliberatore@acm.org

Received May 6, 1998

In this paper, we consider the problem of local register allocation (LRA): given a sequence of instructions (basic block) and a number of general purpose registers, find the schedule of variables in registers that minimizes the total traffic between CPU and the memory system. Local register allocation has been studied for more than 30 years in the theory and compiler communities. In this paper, we give a 2-approximation algorithm for LRA. We also show that a variant of the known further-first heuristic achieves a good approximation ratio. © 2000 Academic Press

1. INTRODUCTION

Register allocation is the problem of deciding which variables occupy the register file at each step of a program. The objective of register

[1] 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. Travel support provided by DIMACS. DIMACS is a partnership of Rutgers University, Princeton University, AT & T Labs-Research, Bell Labs, Telcordia Technologies (formerly Bellcore), and NEC Research Institute. DIMACS is an NSF Science and Technology Center, funded under Contract STC 91-19999. DIMACS is also supported by the New Jersey Commission on Science and Technology. A preliminary version of this paper appeared in the *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithm*, pp. 564–573.
[2] URL: http://www.cs.rutgers.edu/~ farach/.
[3] URL: http://www.umiacs.umd.edu/users/liberato/.

allocation is to minimize the total traffic between the CPU and the memory system. Register allocation has a substantial impact on execution times because of the large speed gap between processors and memories. In fact, it adds the largest single performance improvement to compiled programs [17]. Register allocation has received widespread attention in academic and industrial research in the past few decades [1, 13, 33].

1.1. *Local Register Allocation*

In this paper, we will focus on local register allocations (LRA) for which we provide negative and positive results. Local register allocation assigns registers to variables in *basic blocks*, which are maximal branch-free sequences of instructions. Global register allocation assigns registers to variables throughout the program. The local register allocation problem is general enough to model off-line paging with write-backs [6, 10] and off-line weighted caching [24, 35]. A local allocation schedules the loading of values from memory into registers and the storing from registers into memory, given a fixed number of registers. The main difficulty in finding an optimal local register allocation stems from the trade-off between the cost of loads and the cost of stores.

Local register allocation is the first step in *demand-driven register allocation* [28]. Subsequently, a demand-driven allocator will combine local allocations into a global one. Although the second phase (LRA combination) can cause significant overhead in the worst case, in practice it can be done efficiently [28]. Approaches to extend an LRA to a global allocation are also given in [5, 8, 16, 21–23, 26]. The objective of the local register allocation phase is to minimize the total traffic between CPU and memory. Consequently, it is more accurate and effective than heuristics based on graph coloring, in which the goal is to minimize the number of registers needed to satisfy a sequence of requests. The advantages of local register allocation over graph coloring are summarized, for example, in [19].

1.2. *Previous Work*

Local register allocation has been performed since the first Fortran compiler [3, 4]. To the best of our knowledge, local register allocation was first considered formally in a 1966 paper by Horwitz, Karp, Miller, and Winograd [18]. In that paper, an algorithm was presented to produce an optimal allocation through dynamic programming. The algorithm accurately captures the index register architecture used at the time. However, after more than 30 years, the index register model does not reflect the costs of a modern architecutre with general purpose registers. There are two major differences between index and general purpose registers. First, modern architectures can use the values of more than one variable in one step. Second, recent architectures would decompose the old write operation into a sequence of one read and one write. Nonetheless, the algorithm was implemented as recently as 1989 [19], followed by heuristics to fix the points where the allocation was infeasible for new architectures. As such, the dynamic programming algorithm no longer guarantees an optimal solution on modern architectures. The original algorithm also takes an exponential amount of space and time as a function of the number of registers and of the length of the basic block. The space and time requirements were not a problem in early applications, as only two registers and a short basic block were considered [21]. However, the algorithm fails to terminate in more recent implementations due to excessive memory space demands [19]. LRA has been recently proved to be NP-hard [9]. The problem of finding a "good" local register allocation is left as an open question in the "Dragon Book" [1, section 9.6, function `getreg`, step 3] and in other standard textbooks on compilers [13, 33].

Some heuristics have been proposed to quickly produce a local register allocation [1, 13, 19, 33]. For example, the *furthest-first* algorithm (FF) evicts the program variable that will be referenced furthest in the future [3, 6, 15, 16, 27]. Related heuristics other than FF have been proposed in the context of *spilling*, for whose definition and algorithms we refer the reader to the textbooks [1, 13, 33]. Some known LRA heuristics will be discussed in Section 2.3.

1.3. *New Results*

In this paper, we present a simpler NP-hardness proof for the local register allocation problem. We also provide a 2-approximation algorithm $\mathscr{W}$ for the local register allocation problem. Actually, the exact approximation ratio is $2 - 1/K$, where K is a parameter that will be specified later and that is related to the number of accesses to any given program variable in a basic block. Previously, no algorithm had a performance guarantee bounded by a constant. We consider the furthest-first algorithm that is used in existing compilers [15, 16]. We propose a version of the furthest-first algorithm which we call conservative-furthest-first (CFF). We analyze the behavior of CFF and prove a performance guarantee. Our analysis is tight. We study weighted off-line caching, which was known to be polynomially solvable [35]. We present an algorithm that runs in (almost) quadratic time.

1.4. *Contents*

In Section 2, we will define the local register allocation problem, compare it with paging, discuss previous heuristics, and state our results. In Section 3, we will discuss the NP-hardness of local register allocation. In

Section 4, we will present the approximation algorithms $\mathscr{W}$ and CFF. In the Appendix, we will give an example to contrast our algorithms.

2. PRELIMINARIES

In this section, we formulate the local register allocation problem on architectures with general purpose registers. First, we define the off-line paging problem with write-backs. Then, we will explain how to generalize paging to capture the problem of local register allocation.

2.1. *Paging with Write-Backs*

In the paging problem, a set of M pages has to be maintained. However, only N pages can be accommodated in fast memory at any given step. Remaining pages will reside in slow memory. If a page is requested and it is not in fast memory, a page fault occurs and the requested page is brought into fast memory. In turn, a slot in fast memory has to be cleared for the newly requested page, and a page has to be evicted. If the evicted page is not consistent with its version on slow memory, then the evicted page has to be written back to slow memory. A paging strategy aims at minimizing the total number of page faults and write-backs. Paging with write-backs is a more realistic problem than traditional paging with only page faults because it captures the total traffic between fast and slow memory [10]. Off-line paging with write-backs was first formulated in the 1966 paper by Horwitz *et al.* [18] in the context of LRA. Write-back paging is similar to the problem of local register allocation. Both are problems of minimizing the traffic between two adjoining levels of the memory hierarchy, and once pages are substituted by program variables and memory slots by registers, write-back paging is almost identical to local register allocation. However, write-back paging differs from local register allocation in three respects.

First, write-back paging has been studied both as an on-line and as an off-line problem. In the on-line version, future page references are unknown. In the off-line version, the whole sequence of page references is known in advance. Off-line paging algorithms serve as a theoretical [20, 25, 30] and experimental [12, 31] point of comparison for on-line algorithms and stimulate researchers to find on-line approximations [29]. Local register allocation is inherently an off-line problem because the whole sequence of references is completely found in the compiled code.

The second difference is the following. In the paging problem, only one page reference is generated in one step. In local register allocation, multiple references can be generated in one step. For example, consider the ILOC [7] instruction `iADD t0 t1 => t2` which adds the variables `t0` and `t1` and stores the results in `t2`. The variables `t0` and `t1` have to be read at the same time and fed simultaneously to the ALU unit. Multiple simultaneous references arise also in VLIW architectures [14, 17].

Finally, in the paging problem, a fault always accounts for one unit of memory traffic. In local register allocation, different variables are fetched at different costs. Indeed, many variables are loaded from memory, but others are not. For example, suppose that `t0` is a program variable whose value is equal to a constant. Then, `t0` can be loaded in a register by a *load immediate* operation, which does not involve a memory access. Loading `t0` is therefore cheaper than fetching a variable from memory. Some authors charge a load immediate the cost of half a memory access [7].

2.2. *Local Register Allocation*

We now formulate the problem of local register allocation. For ease of presentation, we will continuously refer the reader to the example in Fig. 1, to Table 1, which summarizes the most common definitions, and to Table 2, which gives the meaning of most symbols used in the text. The model captures the problem of register allocation in architectures with general purpose registers. Moreover, the model specializes to the off-line paging problem with write-backs when $\alpha = C = 1$.

LRA operates on sequences of intermediate code instructions without branches. The leftmost column of Fig. 1 gives an example of such a

Intermediate code	Step	σ	Cost	Register Allocation
				$\emptyset$
	1	(read,{0})	(2)	{ (0, clean) }
`ADDI 3 t0 ⇒ t1`	2	(write,{1})		{ (0,clean), (1,dirty) }
	3	(read,{1,2})	(1)	{ (0,clean), (1,dirty), (2,clean) }
`SUB t1 t2 ⇒ t3`	4	(write,{3})		{ (1,dirty), (2,clean), (3,dirty) }
	5	(read,{3,4})	(2)	{ (1,dirty), (3,dirty), (4,clean) }
`MUL t3 t4 ⇒ t5`	6	(write,{5})		{ (1,dirty), (3,dirty), (5,dirty) }
	7	(read,{2,5})	1	{ (1,dirty), (2,clean), (5,dirty) }
`SUB t2 t5 ⇒ t6`	8	(write,{6})		{ (1,dirty), (5,dirty), (6,dirty) }
	9	(read,{1,6})		{ (1,dirty), (5,dirty), (6,dirty) }
`ADD t1 t6 ⇒ t7`	10	(write,{7})		{ (1,dirty), (5,dirty), (7,dirty) }
		capacity cost:	1	

FIG. 1. An example of an optimal LRA with three registers. The first column gives a sequence of intermediate code instructions, the second and third columns its representation as a basic block, the fourth column gives the cost per operation assuming that $S_2 = 1$ and all other S's are two (so that the maximum spill cost is $C = 2$), and the last column gives the register configuration after each step has been executed. In the example, the set of live variables at the end of the basic block is $L = \{5, 7\}$. Compulsory cost is given in parentheses.

TABLE 1
Summary of the Definitions of Some Frequently Used Terms
(Complete Definitions Are Found in the Text)

Term	Short definition
Allocation strategy	A rule that associates a register allocation to a basic block
Basic block	A sequence of references that satisfy the write-once condition
Capacity miss	A reference (other than the first) to a variable that is not in a register
Clean variable	A variable that is not dirty
Dirty variable	A variable that has not been stored after its definition
Live range	Interval from point of definition to last point of use
Live variable	A variable whose value will be needed in the future
Live on exit	A variable whose value will be needed after the end of the basic block
Reference	A use or definition of a set of at most α variables
Register allocation	A sequence of register configurations, one per basic block step
Register configuration	Variables in register and their state (clean/dirty)
Register pressure	Number of live unallocated variables at step j
Spill cost	Cost to load and store a variable
Value range	Interval between consecutive uses of a live variable

TABLE 2
Summary of the Definitions of Some Frequently Used Symbols
(Complete Definitions Are Found in the Text)

Symbol	Short definition
α	Maximum number of variables in a reference
C	Maximum spill cost
K	Maximum number of value ranges per variable
$K(i)$	Number of value ranges of variable i
$K_L(I)$	Number of value ranges of variable i that require a reload
L	Set of variables live on exit
L_j	Set of variables with live range over step j
M	Number of program variables
n	Sum of reference sizes
N	Number of available registers
ρ_j	Register pressure at step j
r	Basic block length
S_i	Spill cost of variable i
σ	A basic block
V	Set of program variables
W	Set of written program variables

sequence in a RISC-like language. Each instruction executes basic operations over a set of variables. In the example, the operations are additions, subtractions, and so on, and the set of variables is $\{\texttt{t0}, \texttt{t1}, \dots, \texttt{t7}\}$. For simplicity of notation, we will denote a variable simply by an index number, so that we define $V = \{1, 2, \dots, M\}$ to be the set of *program variables*. Each intermediate code instruction generates one or more read and write accesses to the program variables. For example, the instructions `SUB t1 t2 => t3` reads simultaneously the values of `t1` and `t2` in order to execute an arithmetic operation on them. Then, it will write the result into `t3`. Define a read or write request to a program variable as an element of $\{\texttt{read}, \texttt{write}\} \times 2^V$. Thus, `SUB t1 t2 => t3` generates the sequences $((\texttt{read}, \{1, 2\}), (\texttt{write}, \{3\}))$. Figure 1 gives the sequence of read and write requests corresponding to a sequence of intermediate code instructions. In practice, only a small number of variables can be read or written at the same time. For example, a typical RISC instruction reads no more than two variables and writes no more than one. We will denote by α an upper bound on the number of variables that are accessed simultaneously. In typical applications, $\alpha = 2$, but we do not fix an *a priori* bound on α. In write-back paging, $\alpha = 1$. We will say that a *reference* is an element $(s, a) \in \{\texttt{read}, \texttt{write}\} \times 2^V$ where $|a| \leq \alpha$. A first input to the local register allocation problem is a *basic block*, which is defined as a sequence of references of some positive length r. The third column in Fig. 1 gives the basic block corresponding to a certain sequence of intermediate code instructions. In the parlance of compilers, the sequence of instructions is usually called a basic block, but, from the viewpoint of register allocation, the sequence of references is, in some sense, the basic block [19]. We will use the following notation: if σ is a basic block, then the jth element of σ is denoted by σ_j and σ is written as $\sigma = (\sigma_1, \sigma_2, \dots, \sigma_r)$. Define $n = \sum_{j=1}^{r} |a_j|$ for $\sigma_j = (s_j, a_j)$. We can take n to represent the total size of the LRA instance. We will restrict our attention to a proper subset of reference sequences defined by the following *write-once condition*: if a program variable i is written at step σ_j, then i is not written at any other step and i is not read before σ_j. The write-once condition means that the value in i is defined at most once and that the value of i cannot be accessed before the step where i is defined. The write-once condition will be shown to hold without loss of generality when we introduce the cost model. Program variables can be partitioned into those that are written at some step of the basic block and variables that are never written and only read. Define W as the set of variables $W \subseteq V$ that are written at some basic block step. In the example, $W = \{1, 3, 5, 6, 7\}$. In general, we assume that the value of a variable $i \notin W$ can be loaded from a memory location or is a constant that can be determined at compile-time. We will assume for simplicity and without loss of generality that all program variables are referenced; that is, for all $i \in V$, there is a $\sigma_j = (s, a)$ $(1 \leq j \leq r)$ with $i \in a$.

Notice that while a program variable $i \in W$ is written at some basic block step, we do not impose that i be read anywhere else. In Fig. 1, the variable 7 is written in the very last step and never read. However, we cannot assume that variable 7 contains a useless value. Indeed, program variables might hold values that are computed in the current basic block and used in subsequent basic blocks. Although those variables are not currently needed, their value must be maintained for future reference. A program variable is said to be *live on exit* if its value must be maintained after the end of the basic block. The set of variables live on exit is denoted by L. The variables live on exit are determined by a live-variable analysis procedure [1, 13, 33]. We will also need the concept of live range and of live variables. Intuitively, the notion of liveness models basic block intervals where a certain variable value is useful. The *live range* of a variable i is an interval of the basic block σ. The starting point of the live range is the step where i is written if $i \in W$ and it is step 1 otherwise. The ending point of the live range is step r if $i \in L$ and is the last step where i is read otherwise. A variable i is said to be *live* throughout all the steps of its live range. For example, in Fig. 1, variable 1 is live from steps 2 to 9, variable 2 is live from steps 1 to 7, and variable 5 is live from steps 6 to 10.

We turn now to model the memory device that is used to execute the basic block. The processor has N general purpose *registers*. If a program variable i is read and it is not in a register, it has to be loaded at cost S_i. In turn, a program variable i' might have to be evicted to make space for the new program variable i. The eviction of i' could require that i' be stored in main memory at a cost $S_{i'}$. The main difficulty of the local register allocation problems originates from the trade-off between minimizing the cost of loads and the cost of stores. We now detail the cost of loading and storing variables. If a variable i is not live, then it can be evicted from a register without storing it because the value of i will never be needed again. For example, variable 3 can be evicted for free at step 7 because 3 is dead. Consider now live variables. A variable $i \in W$ is said to be *dirty* at step j if it has not been stored into main memory between the step when it was written and step j. All other variables are *clean*. The *state* of a variable is an element of {clean, dirty}. Roughly speaking, dirty live variables need to be stored for correctness of the resulting code. A clean live variable can be evicted from a register without storing it. Consider, for example, step 5 in Fig. 1 and assume that after step 4, variables 1, 2, and 3 are in registers. Step 5 requires the value of 3 and 4. Variable 3 is already in a register, and we only need to load variable 4. If there are only $N = 3$ registers, then either variable 1 or variable 2 must be evicted. If variable 1 is evicted, then the value of 1 must be stored in main memory before step

5. Indeed, we need the value of 1 at step 9 and if we did not store variable 1, we would lose the value. However, variable 2 is already available in memory (or is a constant) and so we do not need to store it. Define a *register configuration* Q as a subset of $V \times \{\texttt{clean}, \texttt{dirty}\}$ of size at most N. A register configuration represents the program variables that are currently in a register and their state. We will assume that a variable is present in at most one register; that is, if $(i, s) \in Q$, then $(i, t) \notin Q$. Such an assumption holds without loss of generality because of the cost model that we use and define below. We will always assume without loss of generality that $N \geq \alpha$, because a request for more than N variables cannot be satisfied with only N registers. Figure 1 gives examples of register configurations.

A *register allocation* for a basic block σ of length r is a sequence $\mathscr{Q} = (Q_0, Q_1, \ldots, Q_r)$ with the properties that

- If $\sigma_j = (\texttt{read}, a)$ and $i \in a$, then either $(i, \texttt{clean}) \in Q_j$ or $(i, \texttt{dirty}) \in Q_j$;
- If $\sigma_j = (\texttt{write}, a)$ and $i \in a$, then $(i, \texttt{dirty}) \in Q_j$ and Q_h contains neither $(i, \texttt{clean})$ nor $(i, \texttt{dirty})$ for all $h = 0, 1, \ldots, j - 1$.

The condition on Q_h states that the value of i cannot be used before it is defined. We define an *allocation strategy* as a rule that associates a register allocation to a basic block σ starting from the initial configuration Q_0. Incidentally, $Q_0 = \emptyset$ in paging as well as in the local allocation problems generated by demand-driven register allocation [28]. However, other register allocators fix a possibly nonempty Q_0 in a global phase preceding local allocation [7, 23]. We now describe our cost model. Each variable i has a *spill cost* S_i. Assume without loss of generality that the costs are normalized so that $\min\{S_i : 1 \leq i \leq M\} = 1$, and define $C = \max\{S_i : 1 \leq i \leq M\}$. In write-back paging, $C = 1$. In compiler applications, $C \geq 2$ typically [7, 23]. We now detail the cost of all load and store operations. The cost of changing the contents of one register from $(i, \texttt{dirty})$ to $(i, \texttt{clean})$ at step j is the cost of storing i, which is S_i if i is live and zero otherwise. The cost of changing the contents of one register from (i, s) to (h, t) ($h \neq i$ and $s, t \in \{\texttt{clean}, \texttt{dirty}\}$) at step j is the sum of the cost for storing (i, s) plus the cost for loading (h, t). The cost for storing (i, s) is S_i if i is live and $s = \texttt{dirty}$, and it is zero otherwise. The cost for loading (h, t) is zero if h is being written at step j. The cost for loading (h, t) is S_h if h is not being written at step j. For example, in step 5 we evict variable 2 at no cost because variable 2 is clean. Then, we load variable 4 for a cost $S_4 = C = 2$. Had we evicted variable 1 instead, we would have had to store it because variable 1 is live and dirty. Hence, the cost would have been $S_1 + S_4 = 4$.

The set L of variables live on exit contributes to the determination of live variables and consequently to the determination of spill cost.

Suppose that a program variable $i \notin W$ is not in the initial register configuration Q_0. Then, i has to be loaded at least once by any allocation strategy. The first load of a program variable has been variously called a *compulsory miss*, a cold start miss, or a first reference miss [17]. Compulsory misses cause a fixed cost that any allocation has to pay. Therefore, compulsory misses can be disregarded while seeking an optimum allocation. Henceforth, compulsory misses will be ignored, and we will consider only noncompulsory or, in the terminology of [17], *capacity misses*. The cost for processing capacity misses will be termed the *capacity cost*. We will use $c(\mathscr{A})$ to denote the capacity cost of an allocation $\mathscr{A}$.

EXAMPLE. The cost of loading 0 and 4 is compulsory. The cost of loading variable 2 for the first time is also compulsory, but not the cost of loading variable 2 for the second time.

An *optimum register allocation* $\mathscr{A}$ from Q_0 is the one whose first configuration is Q_0 and that minimizes $c(\mathscr{A})$. In this paper, we consider the following local register allocation problem:

Instance: A set V of program variables, a spill cost S_i for each $i \in V$, a subset $L \subseteq V$ of variables live on exit, a basic block σ over V, a positive integer N (the number of registers), and an initial register configuration Q_0

Solution: An optimum register allocation for σ with N registers

We remark that local spill cost is only one of the contributing factors to the performance of compiled programs and that overall performance depends also on the effectiveness of other compiler phases, such as, for example, global allocation. As a case in point, consider the operations of a demand-driven allocator. Demand-driven allocators first perform local allocation for each basic block independently under the assumption that $Q_0 = \emptyset$ and with no regard for the final register configuration Q_r. Then the allocator combines the local allocations into a global allocation. As a result, the overall spill cost depends not only on the capacity cost of each single allocation, but also on the effects of the recombination phase. Although the global allocation phase could in principle contribute significantly in addition to the local spill costs, it is claimed that it can be done efficiently in practice [28]. Consequently, LRA affects actual performance more than the global recombination pass and is thus identified as one of the two steps where efforts should be concentrated [19, 28].

We will now justify the write-once condition. Replace a program variable i with a collection of newly introduced program variables, each of which is live between two consecutive writes to i. The newly introduced variables have cost S equal to that of the original variable and belong to L if and only if the original variable does. For example, in

$$((\ldots(\texttt{write},\{1\}),\ldots,(\texttt{read},\{1\}),\ldots,(\texttt{read},\{1\}),\ldots,$$
$$(\texttt{write},\{1\}),\ldots(\texttt{read},\{1\}),\ldots),$$

we will obtain

$$((\ldots(\texttt{write},\{2\}),\ldots,(\texttt{read},\{2\}),\ldots,(\texttt{read},\{2\}),\ldots,$$
$$(\texttt{write},\{3\}),\ldots,(\texttt{read},\{3\}),\ldots),$$

where 2 and 3 are newly introduced variables, $S_2 = S_3 = S_1$, and $2, 3 \in L$ if and only if $1 \in L$. Clearly, no program variable is written more than once. A simple induction shows that the new optimization problem is equivalent to the original one. Incidentally, some compilers [7] produce intermediate code instructions that define a variable only once, so that no further transformation is needed.

2.3. *Previous Heuristics*

In this section, we will discuss two LRA heuristics that have been proposed in the literature. The two heuristics are *furthest-first* (FF) and *clean-first* (CF). Both heuristics assign program variables to empty registers if there is one. The heuristics differ if no empty register exists and a variable has to be evicted from a register. FF evicts the variable that will be requested furthest in the future [3]. CF evicts the clean variable requested furthest in the future if there is one, otherwise it evicts the dirty variable requested furthest in the future [13, 19].

The FF rule has a long and interesting story. Apparently, it was rediscovered several times. FF is commonly attributed to Belady [6], who introduced it in the context of off-line paging. Actually, Belady cites the work by Horwitz *et al.* [18], where FF is implicitly proven to be optimal for paging without write-backs. FF was first implemented by Sheldon Best in 1956 as a part of the first Fortran compiler [3]. Apparently, Best had also an (unpublished) proof of optimality as early as 1955. An FF algorithm gives rise to ties among variables whose next reference is equally further in the future, a case that occurs when more than one variable is referenced in a basic block step ($\alpha > 1$) or if several variables are live on exit, but never referenced again in the basic block. FF can be regarded as a family of algorithms, where each algorithm follows a different rule to break ties. All FF algorithms are optimal for off-line paging without write-backs, which is the LRA problem for $C = \alpha = 1$ and $W = \emptyset$. However, FF is not optimum for write-back paging, where we only dispense with $W = \emptyset$ and retain $C = \alpha = 1$. Moreover, some versions of FF do not achieve any multiplicative performance guarantee. Suppose, for example, that ties are broken against the variable with the smallest index number; that is, if two variables are requested equally further in the future, the one with the smallest index is the first to be evicted. Let $Q_0 = \emptyset$, $\sigma = ((\texttt{write}, \{1\}), (\texttt{read}, \{2\}), (\texttt{read}, \{3\}))$, $L = \{1, 2\}$, and $N = 2$. All load costs are compulsory, an eviction for 2 is at no cost because 2 is clean, and an eviction for 1 costs $S_1 = C = 1$. Then, the cost of FF is 1, while the optimum cost is 0. In Section 4.6, we will show that a simple version of FF achieves a $2C$ performance guarantee for the general LRA problem. The example above also suggests that FF should take into account variable state while breaking ties. The CF heuristic is an attempt to introduce state considerations in the FF rule. Unfortunately, CF can be arbitrarily worse than the optimum for the problem of write-back paging. Construct a basic block that fills all registers but one with dirty variables. Then, ask repeatedly for two clean variables for $x + 1$ times: $((\texttt{read}, \{1\}), (\texttt{read}, \{2\}))^{x+1}$. CF repeatedly evicts and reloads 1 and 2 and pays a capacity cost of $2x$. In this case, the optimum strategy is FF that would evict a dirty variable, paying a cost of 1. Take x arbitrarily large and, while x grows, the cost ratio of CF over FF tends to infinity.

3. HARDNESS OF OFF-LINE PAGING

In this section, we give a new proof of the NP-hardness of LRA. First of all, we will show that the problem is NP-hard for $\alpha = N$ and then we will show the result for $\alpha = 1$. Our proofs hold even for the special case $C = 1$. Previously, NP-hardness was known [9] in the original model of index register architectures [18]. That construction can be adapted to the LRA problem that we consider in this paper. However, we will present a simpler reduction directly from set cover.

LEMMA 3.1. *The decision problem corresponding to local register allocation (i.e., given a basic block σ, an initial configuration Q_0, and two positive integers N and z, decide whether there is a register allocation for N registers that costs no more than z) is NP-complete.*

Proof. The reduction is from set cover: given a collection $\mathcal{S} = \{S_1, S_2, \dots, S_m\}$ of subsets of $S_0 = \{1, 2, \dots, p\}$, find a subset $\mathcal{S}' \subseteq \mathcal{S}$ with the properties that $|\mathcal{S}'| \leq z$ and that every element in S_0 belongs to at least one set in $\mathcal{S}'$. We will assume without loss of generality that for every $j \in S_0$ there is an $S \in \mathcal{S}$ with $j \in S$.

We will have program variables numbered from 1 to m corresponding to the sets $S_1, S_2, \ldots, S_m$ and define $X = \{1, 2, \ldots, m\}$. We introduce a program variable $v_j = m + j$ for each $j \in S_0$, so that the total number of variables is $M = m + p$. Let $X_j = \{i : j \notin S_i\} \cup \{v_j\} \subseteq X \cup \{v_j\}$ for $j \in S_0$. The basic block is

$$\begin{aligned}\sigma \quad = \quad & ((\texttt{write}, \{1\}), (\texttt{write}, \{2\}), \ldots, (\texttt{write}, \{m\}),\\ & (\texttt{write}, \{v_1\}), (\texttt{read}, X_1), (\texttt{read}, X),\\ & (\texttt{write}, \{v_2\}), (\texttt{read}, X_2), (\texttt{read}, X),\\ & \vdots\\ & \big(\texttt{write}, \{v_p\}\big), \big(\texttt{read}, X_p\big), (\texttt{read}, X)\big).\end{aligned}$$

Finally, $Q_0 = L = \emptyset$ and $N = m$. We now highlight some properties that we will use in the next proof. Since j is contained in at least one set from $\mathscr{S}$, we have $|X_j| \leq m$. The maximum size of a requested set is the maximum of $|X| = m$ and $\max\{|X_j| : 1 \leq j \leq p\} \leq m$ and so $\alpha = m = N$. Moreover, no more than one variable is written at any step in σ. Clearly, the size of the LRA instance is polynomial in the size of the input.

We will show that every set cover of size z corresponds to an allocation of cost $z + p$ and that conversely an allocation of cost $z + p$ corresponds to a cover of size at most z. Let $\sigma^{(j)}$ be the subsequence of σ defined as $\sigma^{(j)} = ((\texttt{write}, \{v_j\}), (\texttt{read}, X_j))$ for $j \in S_0$. First, notice that $v_j \notin X_h$ when $h \neq j$ and $v_j \notin X$. It follows that v_j is live only during $\sigma^{(j)}$. Therefore, the set of live variables during $\sigma^{(j)}$ is $X \cup \{v_j\}$. Then, there are exactly $m + 1 = N + 1$ live variables during $\sigma^{(j)}$. It follows that σ can be satisfied by evicting during $\sigma^{(j)}$ one program variable in the set $(X \cup \{v_j\}) - X_j = \{i : j \in S_i\}$. Given a set cover $\mathscr{S}'$ of size z, we construct an allocation as follows. We store all the program variables i with $S_i \in \mathscr{S}'$, and for every $\sigma^{(j)}$, we choose to evict a page i with $j \in S_i \in \mathscr{S}'$ in any arbitrary way. The cost due to stores is exactly z. The evicted variable is requested in $(\texttt{read}, X)$ after $\sigma^{(j)}$ and has to be reloaded, so that the cost due to loads is exactly p. Hence, the total cost of the allocation is $z + p$. Conversely, suppose that there is an allocation $\mathscr{Q}$ with $c(\mathscr{Q}) = z + p$. Notice that any allocation reloads all X after X_j and evicts at least one program variable at each request sequence $\sigma^{(j)}$. In fact, we can assume without loss of generality that exactly one variable is evicted in correspondence to each $\sigma^{(j)}$. Indeed, if this is not the case, we choose in an arbitrary way a program variable q_j among those that are evicted for $\sigma^{(j)}(j \in S_0)$. Then, we replace the original allocation with one that evicts only q_j for $\sigma^{(j)}$. The new allocation is feasible and it costs no more than the original one. But the new allocation immediately corresponds to a set cover of size z, and the result follows. ▮

The case $\alpha = N$ is the most general as it allows requests for any number of pages. However, $\alpha = 1$ in off-line write-back paging and the previous lemma does not hold. Instead, we prove the following

THEOREM 3.2. *The decision problem corresponding to local register allocation (i.e., given a basic block σ and two positive integers N and z, decide whether there is a register allocation that costs no more than z) is NP-complete even for the particular case of off-line write-back paging.*

Proof. The reduction uses the construction in the previous lemma. The only remaining problems are requests of the form $\sigma_j = (\texttt{read}, Y)$ for some set $Y = (y_1, y_2, \dots, y_{|Y|}\} \subseteq V$ that has $|Y| > 1$. Such a request is mapped to a subsequence $\sigma_j' = (\sigma_j'')^4$, where σ_j'' is the sequence

$$\sigma_j'' = \big((\texttt{read}, \{y_1\}), (\texttt{read}, \{y_2\}), \dots, (\texttt{read}, \{y_{|Y|}\})\big).$$

Call the resulting sequence σ'. We will how that an allocation $\mathscr{A}'$ for σ' corresponds to an allocation $\mathscr{A}$ for σ with $c(\mathscr{A}') = c(\mathscr{A})$. First, we will show how to transform $\mathscr{A}'$ into a feasible allocation for σ' with the property that at most one program variable $i \notin \sigma_j$ is evicted during σ_j' for all j's. Recall that at most $N + 1$ program variables are live during σ_j. If the number of live variables is at most N, then no eviction takes place in either allocation. Suppose now that $N + 1$ variables are live during σ_j. Notice that once a program variable $i \notin Y$ is evicted during σ_j', all program variables in Y can be accommodated in registers and no further eviction is necessary during σ_j'. Suppose that during σ_j' the allocation $\mathscr{A}'$ evicts a program variable $i \notin Y$ and also a program variable $i' \in Y$. Then, the eviction of a variable $i' \in Y$ during σ_j' can be avoided without losing feasibility and without increasing the cost. Assume now that only program variables in Y are evicted during σ_j'. If $i \in Y$ is the last program variable evicted during the hth repetition of σ_j'' ($h = 1, 2, 3$), then it will be reloaded during the $(h + 1)$st repetition of σ_j''. At that step, some other variable $i' \in Y$ is not in any register because there are $N + 1$ live variables and only variables in Y are evicted. We conclude that at least one eviction takes place during each of the first three repetitions of σ_j'' and at least one load during each of the last three. Therefore, during the $(h + 1)$st repetition of σ_j'', $\mathscr{A}'$ executes at least one load. So, the total cost on σ_j' is at least 3. But then it would have been cheaper to evict a program variable not in Y for a cost of at most 2 for loading and storing it. It follows that we can obtain an allocation that during σ_j' evicts program variables not in Y and that does not cost more than the original one. Then, we obtain a feasible allocation $\mathscr{A}$ for σ by evicting the same page q_j that $\mathscr{A}'$ evicts during σ_j', and the theorem follows. ▮

4. ALGORITHMS

In this section, we give algorithms for LRA. First, we establish general properties on the placement of loads and stores. Then, we give a new algorithm for the case when $W = \emptyset$ that is based on the solution of a minimum cost network flow problem. Our algorithm is instrumental for getting our 2-approximation algorithm for LRA. We will also describe a tie-breaking rule for FF which yields a $(2C)$-approximation algorithm that runs in almost linear time.

4.1. *Spill Code Placement*

In this section, we will establish general properties on the placement of loads and stores. Specifically, we will show that if an algorithm violates such properties, no cost reduction arises. Thus, we can assume that such placement properties hold without loss of generality for LRA algorithms. Throughout the section, we will continuously refer to the example in Fig. 1 for clarity of presentation.

We begin with some observations concerning the laziness of weighted multipaging algorithms. Consider variable 2 in Fig. 1. Since $Q_0 \not\ni 2$ and variable 2 is not referenced before step 3, we can assume without loss of generality that variable 2 is loaded for the first time at step 3. Indeed, variable 2 is not needed at steps 1 and 2, and one more register is available during those steps if 2 is not loaded. In general, if $i \notin Q_0$, then we can assume without loss of generality that i is not loaded before it is requested for the first time. We now turn to comment on the feasibility of evicting a variable. A dead variable can always be evicted because its value is not needed any longer. Moreover, we can assume without loss of generality that a variable i is evicted as soon as it ceases to be live, because no cost is paid to store i or to reload i at some subsequent step, and one more register becomes immediately free. The situation is more complicated for live variables. We define a *value range* as a maximal sequence of steps where a live variable can be evicted.

EXAMPLE. Variable 2 in Fig. 1 cannot be evicted at steps 3 and 7 because it is requested at those steps. Moreover, variable 2 cannot be evicted at steps 1 and 2 because without loss of generality it is not in a register during those two steps. Finally, variable 2 cannot be evicted after step 8 because we have assumed without loss of generality that variable 2 is evicted immediately after step 7. On the whole, variable 2 can be evicted in steps 4, 5, and 6, and it is evicted immediately after step 7 without loss of generality. Hence, variable 2 has one value range $\{4, 5, 6\}$. Steps 8 to 10 do not count as a value range because variable 2 is dead.

In general, a live variable can be evicted after its first reference at all steps when it is not requested. Hence, a value range of i is a maximal sequence of steps strictly between two references to i. In addition, if $i \in Q_0$, variable i has a value range that starts at step 1 and ends before the first reference to i. Finally, if $i \in L$, variable i has a value range that begins after the last reference to i and ends at step r. We define $K(i)$ to be the number of value ranges of variable i and $K = \max_{1 \leq i \leq M} K(i)$. We will number value ranges by increasing the starting point.

We now turn to estimate the costs for loading program variables. Suppose that a variable i is evicted at some step of one of its value ranges. Then, i must be reloaded by the end of the value range because i is used at that point. Conversely, evictions outside a value range are either infeasible or are for free and occur without loss of generality.

EXAMPLE. Variable 2 is evicted before step 5, and so it must be reloaded at step 7 at a cost S_2. The eviction of variable 2 at step 7 does not cause any cost because the eviction is for free, variable 2 will never be reloaded, and one more register becomes free.

Therefore, an LRA algorithm has to make eviction decisions only during value ranges. We will now argue that at most one eviction and one load of a variable i take place in one value range of i. Specifically, we claim that if a variable i is evicted along one of its value ranges, then without loss of generality variable i is evicted at the beginning of its value range and reloaded at the end. Indeed, if a variable is reloaded at a step j preceding the end of a value range, capacity costs do not decrease, while there is one less available register at step j. Similarly, if a variable is evicted at a step j after the beginning of a value range, capacity costs do not decrease, while there is one available register at step j. As a result, at most one eviction and one load of one variable i takes place in one value range of i. Therefore, the eviction of a program variable i along one of its value range can be associated with a binary decision variable $y_{ik} \in \{0, 1\}$ because the position of the resulting load and store is determined without loss of generality. We have $y_{ik} = 1$ if and only if variable i is evicted during its kth value range.

We now turn to examine the position of store operations. If $i \in W$, a cost S_i is charged for storing i independent of the step j where the store occurs as long as i is live and dirty at step j. It follows that if a live program variable $i \in W$ is ever stored, then we can assume without loss of generality that it is stored at the very inception of its live range. Consequently, we can assume that the decision of storing i is a binary decision variable $x_i \in \{0, 1\}$, where $x_i = 1$ if and only if variable i is stored. Moreover, $x_i \geq y_{ik}$ for $i \in W$ and $k = 1, 2, \dots, K(i)$.

4.2. *Weighted Multipaging*

We turn to the special case of LRA when $W = \emptyset$. Since the weighted paging problems is LRA for $W = L = \emptyset$ and $\alpha = 1$ [35], and we now allow $\alpha > 1$, we term this problem *weighted multipaging*. We will show a polynomial algorithm for weighted multipaging. Since no polynomial algorithm exists for LRA (unless P = NP), we conclude that the hardness of LRA stems from cases when $W \neq \emptyset$.

In weighted multipaging, the capacity cost is due only to loads. Value ranges imply a reload cost if the variable is evicted during that range. The only exception is the last value range of variables in L. Indeed, if a variable live on exit is evicted after its last use in the basic block, then it does not need to be reloaded. We define $K_L(i)$ to be the number of live ranges where an evicted variable has to be reloaded. Clearly, $K_L(i) = K(i)$ for all $i \notin L$ and $K_L(i) = K(i) - 1$ for all $i \in L$. The capacity cost is $\sum_{i=1}^{M} \sum_{k=1}^{K_L(i)} S_i y_{ik}$. We also need to impose constraints that enforce a number N of registers. We continue with the example from Fig. 1. At step 5, there are four live variables, namely 1–4. Since we have only $N = 3$ registers, one of those live variables cannot be in a register. We use ρ_j to denote the number of live variables that must be out of registers at step j. In other words, ρ_j is the maximum of zero and of the difference between the total number of live variables and the number N of registers. In the example, $\rho_5 = 1$. We will say that ρ_j is the *register pressure* at step j. Obviously, variables 3 and 4 cannot be out of registers because they are read at exactly step 5. Hence, the choice is between variables 1 and 2. Recall that a value range is a maximal sequence of steps where a variable is live and it is legal to evict it. Clearly, no value range of 3 and 4 contains step 5. However, there are value ranges of 1 and 2 that contain step 5. Define L_j to be the set of variables that have a value range containing step j. In the example, $L_5 = \{1, 2\}$. Hence, the general principle is that if at step j there are more live variable than registers, then we choose the ρ_j unallocated live variables from L_j. Therefore, $\sum_{i \in L_j} y_{ik_{ij}} \geq \rho_j$, where k_{ij} is the index of the value range of i that contains step j. For example, step 5 forces $y_{11} + y_{21} \geq 1$. On the whole, an integer programming formulation of weighted multipaging is

$$\min \sum_{i=1}^{M} \sum_{k=1}^{K(i)} S_i y_{ik} \tag{1.1}$$

$$\text{s.t.} \sum_{i \in L_j} y_{ik_{ij}} \geq \rho_j \qquad j = 1, 2, \ldots, r \tag{1.2}$$

$$y_{ik} \in \{0, 1\} \qquad i = 1, 2, \ldots, M;\ k = 1, 2, \ldots, K(i). \tag{1.3}$$

Let A be the matrix of coefficients of this program. Notice that A is a zero−one matrix. Moreover, we now argue that A has the consecutive ones properties in the columns; that is, the ones in the columns of A appear in consecutive rows. Indeed, a column corresponds to a value range, and that column has unit coefficients for the constraints corresponding to the steps that belong to that value range. However, the steps in a value range are an interval of the basic blocks. We conclude that A has the consecutive ones property. It follows that (1) corresponds to a minimum cost network flow problem [34], and thus it is polynomially solvable [2]. Although the reduction of (1) to a network flow problem is known in the literature, we explicitly describe the resulting flow problem in order to derive a time bound and for use in the general LRA construction. Basically, the reduction transforms inequalities into equalities by introducing slackness variables, inserts a last row with the equation $0 = 0$, and then subtracts the $(j+1)$st row from the jth for $j = r, r-1, \ldots, 1$ [2]. Hence, each column has exactly one $+1$ and one -1 and thus (1) corresponds to a minimum cost network flow problem. The resulting minimum cost network flow problem is exemplified by the network $\mathcal{N}_1$ in Fig. 2 and is as follows. For each equation of the resulting problem we associate a node v_j $(j = 1, 2, \ldots, r+1)$. The right hand side (node supplies) will be equal to $b_j = \rho_j - \rho_{j-1}$ for $j = 2, \ldots, r$; that is, b_j is the change in register pressure at step $j+1$. Moreover, $b_1 = \rho_1$ and $b_{r+1} = -\rho_r$. The slackness variables give rise to backward arcs $t_{j+1} = (v_{j+1}, v_j)$ $(j = 1, 2, \ldots, r)$ with infinite capacity and no cost. The value range variable y_{ik} would originate forward arcs $(v_{j'}, v_{j''})$ of unit capacity and cost S_i, where j' and $j'' - 1$ are the starting and ending point of the kth value range of variable i. In the actual graph $\mathcal{N}_1$, we break the arc $(v_{j'}, v_{j''})$ into the sequence of three arcs $(v_{j'}, v_{ik})$, $l_{ik} = (v_{ik}, v'_{ik})$, and $l''_{ik} = (v'_{ik}, v_{j''})$ where v_{ik} and v'_{ik} are newly introduced nodes, the arcs $(v_{j'}, v_{ik})$ and l''_{ik} have infinite capacity and no cost, while l_{ik} has unit capacity and S_i cost. For example, the figure shows how an arc (v_5, v_9) is divided into the sequence of three arcs. Such transformation does not change the nature of the problem and will be useful for the general case of LRA. We observe that the amount of flow in l_{ik} is equal to y_{ik}. We also notice that the number of arcs and nodes is proportional to the number of value ranges and thus it is $O(n)$.

4.3. *Weighted Caching*

The problem of weighted caching is the weighted multipaging problem for $\alpha = 1$ and $W = L = \emptyset$. Weighted caching is similar to paging without write-backs except that the value of S_i is not necessarily equal for all pages. The on-line version of this problem was introduced in [24]. Intuitively, weighted off-line caching is much simpler than write-back paging because costs do not depend on the page state. The subnetwork $\mathcal{N}_1$ in Fig.

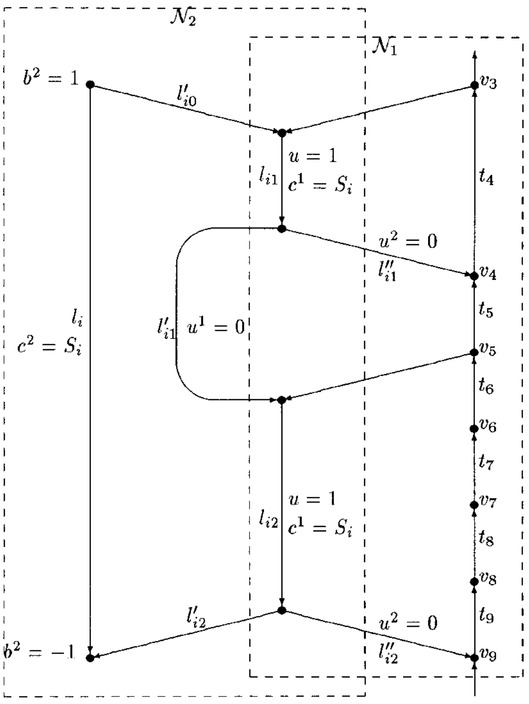

FIG. 2. Part of the multicommodity flow problem relative to one program variable. Arc cost and capacity for commodity $h = 1, 2$ are denoted by c^h and u^h, respectively. Node supply for the second commodity is denoted by b^2.

2 gives an example of minimum cost network flow problems arising from a weighted off-line caching problem.

Since $\alpha = 1$, the register pressure never increases by more than one. It follows that the maximum supply is one. Consequently, weighted off-line caching can be solved in $O(r^2 \log\log r)$ time with the successive shortest path algorithm [2, 32]. Previous algorithms solve the more general k-server problem [11], rely on algorithms for the assignment problem, and run in $O(r^2\sqrt{r}\log(rC))$ time and in strongly polynomial time in $O(r^3)$ [2].

4.4. *General LRA*

The program (1) can be modified to take into account the case $W \neq \emptyset$ by introducing constraints $x_i \geq y_{ik}$. The resulting formulation is

$$\min \sum_{i \in W} S_i x_i + \sum_{i=1}^{M} \sum_{k=1}^{K_L(i)} S_i y_{ik} \tag{2.1}$$

$$\text{s.t.} \sum_{i \in L_j} y_{ik_{ij}} \geq \rho_j \qquad j = 1, 2, \ldots, r \tag{2.2}$$

$$x_i \geq y_{ik} \qquad i \in W;\ k = 1, 2, \ldots, K(i) \tag{2.3}$$

$$y_{ik} \in \{0, 1\} \quad i = 1, 2, \ldots, M;\ k = 1, 2, \ldots, K(i) \tag{2.4}$$

$$x_i \in \{0, 1\} \quad i \in W. \tag{2.5}$$

Notice that the constraint $y_{ik} \in \{0, 1\}$ can be relaxed to $0 \leq y_{ik} \leq 1$. Indeed, suppose that the value of the x_i's has been fixed to an integer value. If $x_i = 0$, then we simply set $y_{ik} = 0$ ($k = 1, 2, \ldots, K(i)$). If $x_i = 1$, then the constant $x_i \geq y_{ik}$ is superfluous. Hence, the program with fixed integral x_i's is once again a minimum cost network flow problem, and the y_{ik}'s will then be integral in a basic solution, even if the integrality constraint is relaxed.

The x_i's can be incorporated into the network flow by introducing a second commodity as depicted by the network $\mathscr{N}_2$ in Fig. 2 and detailed below. For each $i \in W$, insert a new source and sink of unit supply for the second commodity and an arc l_i of unit capacity and cost S_i. We will also constrain the second commodity to take integer values. We will show later on that the flow on l_i will take a value equal to x_i. We connect all arcs l_{ik} into a path from the source to the sink of the second commodity through auxiliary arcs l'_{ik} of zero cost as depicted in Fig. 2. The arcs l'_{ik} have zero capacity for the first commodity and infinite capacity for the second commodity. Conversely, the arcs l''_{ik} have infinite capacity for the first commodity and zero capacity for the second commodity. Therefore, the first commodity is confined to the subnetwork $\mathscr{N}_1$ and the second commodity to the subnetwork $\mathscr{N}_2$. Finally, the arcs l_{ik} have unit bundle capacity. In the previous section, we had set the capacity of the l_{ik} arcs to be one for the first commodity. Such constraint is now redundant because it is subsumed by the bundle constraint on l_{ik}. However, we will keep those redundant constraints as they will play a technical part in the construction and analysis of our approximation algorithm. Recall that the flow (of the first commodity) through l_{ik} is equal to y_{ik}. Hence, if there is a k for which $y_{ik} > 0$, then, by the bundle capacity constraints and by integrality, no flow of the second commodity can flow through l_{ik}, and so the flow through l_i is one. The second commodity flow can avoid l_i and the associated cost S_i if and only if $y_{ik} = 0$ for all k's. Since the flow on l_i is greater or equal to all y_{ik}, such flow can be identified with x_i. We will use the following notation: $c^h(e)$ denotes the cost of arc e for the hth commodity ($h = 1, 2$).

4.5. *A 2-Approximation Algorithm*

This section is devoted to the proof of the following

THEOREM 4.1. *There is a polynomial-time approximation algorithm for local register allocation with a multiplicative performance guarantee of* $2 - 1/K$.

The gist of the algorithm $\mathscr{W}$ is to distribute the cost S_i of storing program variable i among all the value ranges of i and then solve the resulting weighted off-line caching problem. If store costs are distributed equally, no constant approximation factor is possible (compare with the analysis below), and so we will distribute store cost according to a different scheme.

A fundamental component of the argument is to compare the integer multicommodity flow formulation and its *linear relaxation*, which is the problem where arc flows can take noninteger values. Clearly, the optimum value of the linear relaxation is a lower bound on the integer optimum. Finally, we exploit the theory of Lagrangian relaxation as follows. We notice that the difficulty of the problem stems from the interaction between the subnetworks $\mathscr{N}_1$ and $\mathscr{N}_2$. We can decouple those networks by considering the bundle constraints on the arcs l_{ik} and moving them into the objective function with some *prices* $w(l_{ik})$, which are Lagrangian multipliers for the bundle constraints. The resulting flow problem is as follows. The flows of the first and second commodity are confined to $\mathscr{N}_1$ and $\mathscr{N}_2$, respectively, and there is no bundle constraint. However, we maintain the constraint that the l_{ik} arcs have unit capacity for the first commodity. Although such capacity is redundant in the multicommodity flow formulation, it is critical after the bundle constraints have been relaxed. Recall that $c^h(l_{ik})$ is the cost across arc l_{ik} for the hth commodity. The cost $c^h(l_{ik})$ is changed into a new cost $c^h(l_{ik}) + w(l_{ik})$, where $w(l_{ik})$ is a price or Lagrangian multiplier. On the whole, we have decoupled a two-commodity flow problem into two single commodity network flow problems on the networks $\mathscr{N}_1$ and $\mathscr{N}_2$. The arcs l_{ik} have now a cost $c^h(l_{ik}) + w(l_{ik})$ for the hth commodity and unit capacity for the first commodity. We need the technique of Lagrangian relaxation to correlate the optimum value of the resulting "priced" problem with the optimum value of the original problem.

Once we have assigned certain prices $w(l_{ik})$ to the l_{ik} arcs, we solve the problem on $\mathscr{N}_1$ to obtain an optimum flow. The optimum flow is an integer without loss of generality and we interpret it as a register allocation as we described above (that is, store a variable $i \in W$ if and only if i is evicted along one of its ranges). A simple way to assign the $w(l_{ik})$ prices is to charge S_i to the last value range and nothing to the other ones: $w(l_{iK(i)}) = S_i$ and $w(l_{ik}) = 0$ for all $k < K(i)$. Such an algorithm is a 2-approximation algorithm, by a proof similar to the one we give below. A more uniform distribution of prices leads to a better $(2 - 1/K)$-approximation algorithm.

We let

$$w(l_{ik}) = \begin{cases} \dfrac{1}{2K(i) - 1} S_i & \text{if } k < K(i) \\ \dfrac{K(i)}{2K(i) - 1} S_i & \text{otherwise.} \end{cases} \tag{3}$$

Then, solve the minimum cost network flow problem on $\mathscr{N}_1$ with costs $c^1(l_{ik})$ replaced by $c^1(l_{ik}) + w(l_{ik})$. Let ϕ^1 be the resulting flow of the first commodity. We will assume without loss of generality that ϕ^1 is integral. We denote by $E \subseteq W$ the set of program variables that are stored in the resulting allocation, which is to say $E = \{i : i \in W,\ \exists k \in \{1, 2, \dots, K(i)\}.\ \phi^1(l_{ik}) = 1\}$. The set E implicitly gives a flow ϕ^2 for the second commodity; that is, $\phi^2(l_i) = 1$ if and only if $i \in E$. An example of algorithm $\mathscr{W}$ is reported in the Appendix.

We now conduct the analysis of $\mathscr{W}$ and show that it establishes Theorem 4.1. The proof is summarized as follows. The optimum cost of a local register allocation is at least equal to the value of the linear relaxation of the multicommodity integer flow. In turn, the optimum cost of the linear relaxation is at least equal to the value of the Lagrangian relaxation when the bundle constraints are brought into the objective function with prices w. But the value of the Lagrangian relaxation can be expressed in terms of the optimum flow ϕ that is determined by the algorithm $\mathscr{W}$, and the performance guarantee will follow. Throughout the proof, it is important to appreciate the difference between the optimum value of the integer multicommodity problem, which is equal to the allocation capacity cost, the flow cost on $\mathscr{N}_1$ and $\mathscr{N}_2$, the optimum value of the relaxation, and the algorithm cost.

Proof of Theorem 4.1. Since original costs are integers and prices are fractional numbers, the precision required to perform a minimum cost flow computation increases. However, notice that the ratio of the largest over the smallest cost is no more than $O(CK) = O(rC)$, and the precision needed to carry out the computation is increased only by an $O(\log r)$ additive term when prices are added to costs.

The flow ϕ^1 of the first commodity is optimum with respect to the new costs $c^1 + w$ by definition of the algorithm. The construction above implies a flow ϕ^2 of the second commodity that is optimum with respect to the prices $c^2 + w$ because $c^2(l_{ik}) = 0$ and $\sum_{k=1}^{K(i)} w(l_{ik}) = S_i$ is the cost of l_i. Let $z_1^*(w) = \sum_{e \in A} c^1(e)\phi^1(e)$. Observe that only nonpriced costs appear in the expression for $z_1^*(w)$. Hence, $z_1^*(w)$ is not the flow cost on the priced network $\mathscr{N}_1$, but is the actual register allocation capacity cost that is due to loads only. Let $z_2^*(w) = \sum_{e \in A} c^2(e)\phi^2(e) = \sum_{i \in E} S_i$ be the cost due to stores only. The total cost of the register allocation is $c(\mathscr{W}) = z_1^*(w) + z_2^*(w)$. The costs $z_1^*(w)$ and $z_2^*(w)$ are at this point unrelated, and this fact constitutes a difficulty in the proof. The next argument is motivated by the need of bounding the cost $z_2^*(w)$ in terms of the cost $z_1^*(w)$.

Partition the set E of stored variables into two subsets E^0 and E^1 depending on whether those variables are evicted during their last value range or not, that is, $E^0 = \{i \in E : \phi^1(l_{iK(i)}) = 0)$ and $E^1 = \{i \in E : \phi^1(l_{iK(i)}) = 1\}$. Correspondingly, let for $h = 0, 1$,

$$z_{h1}^*(w) = \sum_{i,k\,:\,i \in E^h} c^1(l_{ik})\phi^1(l_{ik})$$

and

$$z_{h2}^*(w) = \sum_{i \in E^h} S_i. \tag{4}$$

Notice that $z_{01}^*(w) \geq z_{02}^*(w)$ because for a program variable $i \in E^0$ there is a value range $k < K(i)$ with $\phi^1(l_{ik}) = 1$ to cause $i \in E$. We have now succeeded in bounding $z_2^*(w)$ in terms of $z_1^*(w)$ with respect to E^0 only, and this will be enough for the proof.

Let $z_{21}^*(w) = \sum_{i \notin W} \sum_{k=1}^{K(i)} c^1(l_{ik})\phi^1(l_{ik})$ be the register allocation cost due to the program variables that do not have to be stored. On the whole, $z_1^*(w) = z_{01}^*(w) + z_{11}^*(w) + z_{21}^*(w)$, $z_2^*(w) = z_{02}^*(w) + z_{12}^*(w)$, and $c(\mathscr{W}) = z_1^*(w) + z_2^*(w) = z_{01}^*(w) + z_{11}^*(w) + z_{21}^*(w) + z_{02}^*(w) + z_{12}^*(w)$.

The cost of the optimum allocation OPT can be estimated as follows. We relax the integrality constraints. We also take the Lagrangian relaxation of the 2-commodity flow problem by bringing the bundle constraints into the objective function with prices w. The value of the Lagrangian relaxation can be obtained with a standard formula [2] and yields

$$c(\mathrm{OPT}) \geq z_1^*(w) + z_2^*(w) - \sum_{i,k\,:\,\phi^1(l_{ik}) + \phi^2(l_{ik}) = 0} w(l_{ik}).$$

If $i \in E^0$, then

$$\sum_{k\,:\,\phi^1(l_{ik}) + \phi^2(l_{ik}) = 0} w(l_{ik}) \leq \left(1 - \frac{1}{2K(i) - 1}\right) S_i \leq \left(1 - \frac{1}{2K - 1}\right) S_i,$$

because at least one value range $k < K(i)$ has $\phi^1(l_{ik}) = 1$.

If $i \in E^1$, then

$$\sum_{k:\phi^1(l_{ik})+\phi^2(l_{ik})=0} w(l_{ik}) \le \frac{K(i)-1}{2K(i)-1} S_i \le \frac{K-1}{2K-1} S_i$$

because at least $\phi^1(l_{iK(i)}) = 1$ and $K(i) \le K$.

Finally, if $i \in W - E$, then

$$\sum_{k:\phi^1(l_{ik})+\phi^2(l_{ik})=0} w(l_{ik}) = 0$$

because $\phi^2(l_{ik}) = 1$ for all k's.

Then, by (4),

$$\begin{aligned} c(\mathrm{OPT}) &\ge z_1^*(w) + z_2^*(w) - \sum_{i\in E^0}\left(1 - \frac{1}{2K-1}\right)S_i - \sum_{i\in E^1}\frac{K-1}{2K-1}S_i \\ &\ge z_{01}^*(w) + \frac{1}{2K-1}z_{02}^*(w) + z_{11}^*(w) + \frac{K}{2K-1}z_{12}^*(w) + z_{21}^*(w). \end{aligned}$$

Define $\beta = 1/(2K-1)$ and notice that $1 + \beta = 2\beta K$. The cost ratio is at most

$$\frac{c(\mathscr{W})}{c(\mathrm{OPT})} \le \frac{z_{01}^*(w) + z_{02}^*(w) + z_{11}^*(w) + z_{12}^*(w) + z_{21}^*(w)}{z_{01}^*(w) + \beta z_{02}^*(w) + z_{11}^*(w) + \beta K z_{12}^*(w) + z_{21}^*(w)}.$$

Elementary calculus shows that the ratio is maximized for $z_{01}^*(w) = z_{02}^*(w)$ and $z_{11}^*(w) = z_{21}^*(w) = 0$, and

$$\begin{aligned} \frac{c(\mathscr{W})}{c(\mathrm{OPT})} &\le \frac{2z_{02}^*(w) + z_{12}^*(w)}{(1+\beta)z_{02}^*(w) + \beta K z_{12}^*(w)} = \frac{2z_{02}^*(w) + z_{12}^*(w)}{2\beta K z_{02}^*(w) + \beta K z_{12}^*(w)} \\ &\le \frac{1}{\beta K} = 2 - \frac{1}{K}, \end{aligned}$$

which completes the proof. ∎

The running time of the algorithm is slightly worse than before because the register pressure may change by more than one. Consequently, the total supply is now bounded by n. The running time of the algorithm is $\tilde{O}(n^2)$ [2].

4.6. *Fast Approximation*

In this section, we will establish that

THEOREM 4.2. *There is a $(2C)$-approximation algorithm for local register allocation that runs in $\tilde{O}(n)$ time.*

The algorithm is similar to furthest-first. It has an additional tie-breaking rule for the case when there is more than one page requested furthest in the future. The algorithm chooses to evict a program variable that in some sense is the cheapest to remove from the register configuration.

Conservative Furthest-First (*CFF*). In response to a fault, determine the set F of program variables requested furthest in the future. Evict a program variable in F that is not live. If all program variables in F are live, evict a variable in F that is clean. If all program variables in F are live and dirty, evict one variable in F arbitrarily.

We claim that the algorithm conservative furthest-first establishes Theorem 4.2. We will assume familiarity with the proof of optimality of FF [25]. The proof is complicated by the fact that some program variables can be evicted without being subsequently reloaded.

Proof of Theorem 4.2. Let $\mathscr{Q}$ be an allocation. Define the eviction points of $\mathscr{Q}$ to be the step indexes where $\mathscr{Q}$ evicts a program variable from the register configuration. By optimality of furthest-first, CFF has the least possible number of eviction points. Let $f_1 < f_2 < \cdots < f_h$ be the eviction points of CFF and t the number of CFF's eviction points where CFF evicts clean variables that do not have to be reloaded during the basic block. The cost of CFF is then at most $2C(h-t)$. Let $\mathscr{Q}$ be the optimum allocation, $g_1 < g_2 < \cdots < g_h$ be the first h eviction points of $\mathscr{Q}$, and $t_{\mathscr{Q}}$ be the number of $\mathscr{Q}$'s eviction points where $\mathscr{Q}$ evicts clean variables that do not have to be reloaded during the basic block. The optimum cost is then at least $h - t_{\mathscr{Q}}$. By the proof in [25], $f_j \geq g_j$ for $j = 1, 2, \ldots, h$. It follows that if $\mathscr{Q}$ evicts during $g_1, g_2, \ldots, g_h$ a clean variable that is not reloaded during the basic block, so does CFF. Therefore, $t \geq t_{\mathscr{Q}}$, and the result follows. ∎

We can also show that our analysis is tight. Let

$$\sigma = ((\texttt{write}, \{1\}), (\texttt{read}, \{3\}), (\texttt{write}, \{2\}),$$
$$(\texttt{read}, \{2\}), (\texttt{read}, \{3\}), (\texttt{read}, \{1\})),$$

and $S_1 = S_2 = C$, $S_3 = 1$, $L = \emptyset$. For $N = 2$, CFF evicts 1 at a cost of $2C$, whereas if we evict 3 we pay only $S_3 = 1$.

APPENDIX: AN EXAMPLE

In the Appendix, we give an example of LRA and illustrate the behavior of $\mathscr{W}$ and CFF.

Let $\sigma = ((\texttt{write}, 1), (\texttt{write}, 2), (\texttt{write}, 3), (\texttt{write}, 4), (\texttt{read}, 2), (\texttt{write}, 5), (\texttt{read}, 1))$ be the sequence of requests, $Q_0 = \varnothing$ the initial register configuration, $L = \{2, 3, 5\}$ the set of variables live on exit, and $C = 1$.

The following table reports the behavior of CFF on σ. It also gives the optimum register allocation.

		CFF		OPT	
Step	σ	Reg. config.	Cost	Reg. config.	Cost
1	(write, 1)	{(1, dirty)}	0	{(1, dirty)}	0
2	(write, 2)	{(1, dirty), (2, dirty)}	0	{(1, dirty), (2, dirty)}	0
3	(write, 3)	{(2, dirty), (3, dirty)}	1	{(1, dirty), (3, dirty)}	1
4	(write, 4)	{(2, dirty), (4, dirty)}	1	{(1, dirty), (4, dirty)}	1
5	(read, 2)	{(2, dirty), (4, dirty)}	0	{(1, dirty), (2, clean)}	1
6	(write, 5)	{(2, dirty), (5, dirty)}	0	{(1, dirty), (5, dirty)}	0
7	(read, 1)	{(1, clean), (2, dirty)}	2	{(1, dirty), (5, dirty)}	0
		Total:	4		3

Note that the eviction of 4 can be performed for free because 4 is not live on exit. The live ranges are given in the following table:

Variable	Decision variable	Range
1	l_{11}	{2, 3, 4, 5, 6}
2	l_{21}	{3, 4}
2	l_{22}	{6, 7}
3	l_{31}	{4, 5, 6, 7}
4	l_{41}	{4}
5	l_{51}	{6, 7}

Correspondingly, the algorithm $\mathscr{W}$ defines and solves the minimum cost network flow in Fig. 3. All live range arcs l_{ik} have unit capacity and all t_i arcs have infinite capacity. Moreover, the cost of the t_i's is zero. The flow is positive only on the value range arcs l_{21}, l_{22}, and l_{31}. The eviction of the corresponding variables during those three value ranges yields the optimum allocation.

ACKNOWLEDGMENTS

Uli Kremer has been an invaluable source of information on the issues arising in existing optimizing compilers. Two anonymous referees greatly helped to improve the presentation of this paper. Rakesh Barve pointed out to us Ref. [35]. Waleed Meleis Ref. [9], and anonymous referees Refs. [3, 11]. We thank Sri Divakaran, Mike Grigoriadis, Sampath Kannan, Leonid Khachyian, Sanjeev Khanna, Lorant Porkolab, Barbara Ryder, Jorge Villavicencio, and Shiyu Zhou for helpful discussions.

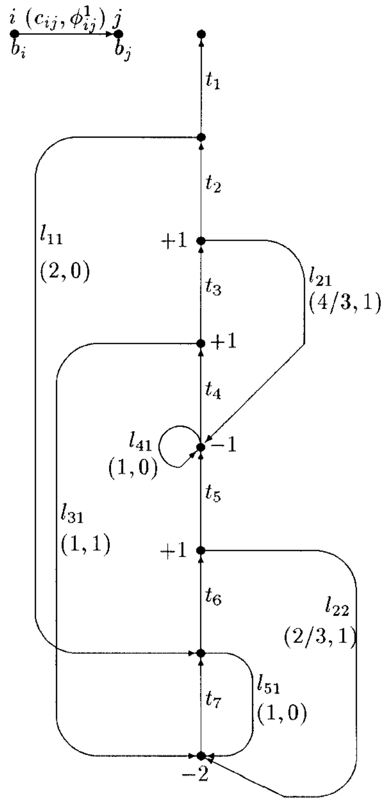

FIG. 3. Sample flow problem solved by $\mathscr{W}$.

REFERENCES

1. A. V. Aho, R. Sethi, and J. D. Ullman, "Compilers: Principles, Techniques, and Tools," Addison-Wesley, Reading, MA, 1986.
2. R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, "Network Flows," Prentice-Hall, Englewood Cliffs, NJ, 1993.
3. J. Backus, The history of FORTRAN I, II, and III, *in* "History of Programming Language" (R. Wexelblat, Ed.) ACM Monographs, pp. 25–45, Assoc. Comput. Mach., New York, 1981.
4. J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Heerick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt, The Fortran automatic coding system, *in* "Western Joint Computer Conference," pp. 188–198, 1957.
5. J. C. Beatty, Register assignment algorithm for generation of highly optimized code, *IBM J. Res. Develop.* **18** (1974), 20–39.
6. L. A. Belady, A study of replacement algorithms for a virtual storage computer, *IBM Systems J.* **5** (1966), 78–101.

7. P. Briggs, K. D. Cooper, and L. Torczon, Improvements to graph coloring register allocation, *ACM Trans. Progr. Languages Systems* **16** (1994), 428–455.
8. D. Callahan and B. Koblenz, Register allocation via hierarchical graph coloring, *in* "Proceedings of the ACM SIGPLAN '91 Conference on Programming Languages Design and Implementation," pp. 192–203, June 1991.
9. M. C. Carlisle, "On Local Register Allocation," Honors Thesis, University of Delaware, 1991.
10. R. W. Carr, "Virtual Memory Management," Computer Science: Systems Programming, Vol. 20, UMI Research Press, Ann Arbor, MI, 1984.
11. M. Chrobak, H. Karloff, T. Payne, and S. Vishwanathan, New results on server problems, *SIAM J. Discrete Math.* **4** (1991), 172–181.
12. A. Fiat and Z. Rosen, Experimental studies of access graph based heuristics: Beating the LRU standard? *in* "Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms," pp. 63–73, 1997.
13. C. N. Fischer and R. J. LeBlanc, Jr., "Crafting a Compiler," Benjamin–Cummings, Menlo Park, CA, 1988.
14. J. A. Fisher, Very long instruction word architectures and ELI-512, *in* "Proc. Tenth Symposium on Computer Architecture," pp. 140–150, June 1983.
15. C. W. Fraser and D. R. Hanson, Simple register spilling in a retargetable compiler, *Software* **22** (1992), 85–99.
16. L. J. Hendren, G. R. Gao, E. R. Altman, and C. Mukerji, A Register Allocation Framework Based on Hierarchical Cyclic Interval Graphs, ACAPS Technical Memo 33, McGill University, February 1993.
17. J. L. Hennessy and D. A. Patterson, "Computer Architecture: A Quantitative Approach," second ed., Morgan Kaufmann, San Mateo, CA, 1996.
18. L. P. Horwitz, R. M. Karp, R. E. Miller, and S. Winograd, Index register allocation, *J. Assoc. Comput. Mach.* **13** (1966), 43–61.
19. W.-C. Hsu, C. N. Fischer, and J. R. Goodman, On the minimization of load/stores in local register allocation, *IEEE Trans. Software Eng.* **15** (1989), 1252–1260.
20. S. Irani, A. R. Karlin, and S. Phillips, Strongly competitive algorithms for paging with locality of reference, *SIAM J. Comput.* **25** (1996), 477–497.
21. K. Kennedy, Index register allocation in straight line code and simple loops, *in* "Design and Optimization of Compilers" (R. Rustin, Ed.), pp. 51–63, Prentice-Hall, Englewood Cliffs, NJ, 1972.
22. D. Kranz et al. ORBIT: An optimizing compiler for scheme, *in* "Proceedings of the ACM SIGPLAN '86 Conference on Programming Languages Design and Implementation," 1986.
23. V. Liberatore, M. Farach-Colton, and U. Kremer, Evaluation of algorithms for local register allocation, *in* "Compiler Construction (CC'99)" (S. Jähnichen, Ed.), Lecture Notes in Computer Science, Vol. 1575, pp. 137–152, 1999.
24. M. S. Manasse, L. A. McGeoch, and D. D. Sleator, Competitive algorithms for server problems, *J. Algorithms* **11** (1990), 208–230.
25. L. A. McGeoch and D. D. Sleator, A strongly competitive randomized paging algorithm, *Algorithmica* **6** (1991), 816–825.
26. W. G. Morris, CCG: A prototype coagulating code generator, *in* "Proceedings of the ACM SIGPLAN '91 Conference on Programming Languages Design and Implementation," pp. 45–58, June 1991.
27. I. Nakata, On compiling algorithms for arithmetic expressions, *Commun. ACM* **10** (1967), 492–494.
28. T. A. Proebsting and C. N. Fischer, Demand-driven register allocation, *ACM Trans. Progr. Languages System* **18** (1996), 683–710.

29. A. Silberschatz and J. L. Peterson, "Operating Systems Concepts," Addison-Wesley, Reading, MA, 1983.
30. D. D. Sleator and R. E. Tarjan, Amortized efficiency of list update and paging rules, *Comm. Assoc. Comput. Mach.* **28** (1985), 202–208.
31. A. S. Tanenbaum, "Modern Operating Systems," Prentice-Hall, Englewood Cliffs, NJ, 1992.
32. M. Thorup, On RAM priority queues, *in* "Proc. Seventh ACM-SIAM Symposium on Discrete Algorithms," pp. 59–67, 1996.
33. J.-P. Tremblay and P. G. Sorenson, "The Theory and Practice of Compiler Writing," McGraw-Hill, New York, 1985.
34. A. F. Veinott, Jr. and H. M. Wagner, Optimal capacity scheduling—I. *Oper. Res.* **10** (1962), 518–532.
35. N. Young, The *K*-server dual and loose competitiveness for paging, *Algorithmica* **11** (1994), 525–541.