DIMACS Technical Report 97-33
July 1997

On Local Register Allocation

by

Martin Farach[1,2]
Rutgers University
and
Bell Labs

Vincenzo Liberatore[3,4]
Rutgers University

[1]Permanent Member
[2]Department of Computer Science, Hill Center, Rutgers University, New Brunswick, NJ 08903 and Information Sciences Research Center, Bell Labs, 700 Mountain Avenue, Murray Hill, NJ 07974. E-mail: `farach@cs.rutgers.edu`. URL: `http://www.cs.rutgers.edu/~farach/`. Research 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.
[3]Affiliated Graduate Student Memeber
[4]Department of Computer Science, Hill Center, Rutgers University, New Brunswick, NJ 08903. E-mail: `liberato@cs.rutgers.edu`. URL: `http://www.cs.rutgers.edu/~liberato/`. Research 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.

DIMACS is a partnership of Rutgers University, Princeton University, AT&T Labs, Bellcore, and Bell Labs.

DIMACS is an NSF Science and Technology Center, funded under contract STC–91–19999; and also receives support from the New Jersey Commission on Science and Technology.

ABSTRACT

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 thirty years in the theory and compiler communities.

It was not known if LRA is NP-hard, but no subexponential time algorithm was known. Furthermore, the most popular heuristics in use in compilers can perform arbitrarily poorly in the worst case. In this paper, we present the following results:

- We show that the Local Register Allocation problem is NP-hard.
- We show that a variant of the furthest-first heuristic achieves a good approximation ratio.
- We give a 2-approximation algorithm for LRA.

We report the experimental performance of a branch-and-bound algorithm and both approximation algorithms on standard benchmarks.

1 Introduction

Register allocation is a crucial task of optimizing compilers. It adds the largest single performance improvement to compiled programs [HP96]. Register allocation has received widespread attention in academic and industrial research in the past few decades [ASU86, FL88, TS85].

Register allocation is the problem of deciding which variables occupy the register file at each step of a program. The objective of the register allocation phase is to minimize the total memory traffic between the CPU and the memory system. Register allocation is tremendously effective because of the huge speed gap between processors and memories.

In addition to the attention register allocation in its own right, it has both Off-Line Paging with Write-Backs [Bel66, Car84] and Off-Line Weighted Caching [MMS90, You94] as particular cases.

Local Register Allocation In this paper, we will focus on local register allocation and we will provide strong 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 and weighted caching. An optimum local allocation schedules the loading of values from memory into registers and the storing from registers into memory. The main difficulty of local register allocation stems from the trade-off between the cost of loads and the cost of stores.

In the compiler context, local register allocation should take priority over global register allocation [ASU86, PF96]; for example, inner loops should have priority over outer loops [ASU86]. Once a local allocation has been determined, it can be extended to a global register allocation [Bea74, CK91, HGAM93, Ken72, K^+86, Mor91, PF96]. 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 global 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 [HFG89].

The problem of local register allocation is left as an open question in the "Dragon Book" ([ASU86], section 9.6, function `getreg`, step 3) and in other standard textbooks on compilers [FL88, TS85]. Local register allocation has been performed since the first FORTRAN compiler [BBB^+57], and formally studied since the mid-sixties [HKMW66]. Currently, the best algorithm for local register allocation takes exponential time and space in the worst case, but no lower bound is known.

New Results In this paper, we present the first NP-hardness proof for the local register allocation problem. Our proof settles the complexity issue of this fundamental outstanding question. Moreover, our proof holds also for the special case of off-line paging with write-backs.

We also provide a 2-approximation algorithm $\mathcal{W}$ for the local register allocation problem. Our algorithm is the first polynomial-time approximation algorithm for this problem with a performance guarantee.

We consider the furthest-first algorithm algorithm that is used in existing compilers [FH92, HGAM93]. We propose a simple modification 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 [You94]. We present the first algorithm that runs in (almost) quadratic time.

We conduct experiments with our algorithms on real benchmarks. We compare a branch-and-bound algorithm based on linear programming relaxation, the approximation algorithm $\mathcal{W}$, and CFF. The approximation algorithms $\mathcal{W}$ and CFF returns solutions of comparable cost on average, but $\mathcal{W}$ is much slower. Thus, $\mathcal{W}$'s interest lies mainly in the fact that it has a better worst-case performance guarantee. The branch-and-bound procedure takes polynomial space and never ran for more than nine minutes even on long basic blocks and with 256 registers. The previous optimum algorithm failed to terminate on much shorter basic blocks and with only 4 registers due to insufficient memory space [HFG89]. So, the branch-and-bound procedure is an optimum algorithm that is more general and practical than the previous dynamic programming algorithm [HKMW66, HFG89].

Previous Work To the best of our knowledge, local register allocation was first considered formally in a 1966 paper by Horwitz, Karp, Miller, and Winograd [HKMW66]. 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. Unfortunately, after more than thirty years, the index register model does not reflect the costs of a modern architecture with general purpose registers. Nonetheless, the algorithm was implemented as recently as 1989 [HFG89], followed by heuristics to fix the points were 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 2 registers and short basic block were considered [Ken72]. However, the algorithm fails to terminate in more recent implementations due to excessive memory space demands [HFG89].

Some heuristics have been proposed to produce quickly a local register allocation [ASU86, FL88, HFG89, TS85]. For example, the *furthest-first* algorithm evicts the program variable that will be referenced furthest in the future [Bel66, FH92, HGAM93, Nak67]. Related heuristics have been proposed in the context of *spilling*, for whose definition and algorithms we refer the reader to the textbooks [ASU86, FL88, TS85].

Previous to this paper, no optimum algorithm was known to take less than exponential time and space in the worst case, nor to work "reasonably" well on actual benchmarks. On the other hand, no lower bound was known.

Contents In section 2, we will define the local register allocation problem, compare it with paging, and state our results. In section 3, we will discuss the NP-hardness of local register allocation. In section 4, we will present a fast branch-and-bound algorithm, the approximation algorithms $\mathcal{W}$ and CFF, and discuss experimental results.

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 have 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 written back in 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 [Car84] because it captures the total traffic between fast and slow memory. 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 [IKP96, MS91, ST85] and experimental [FR97, Tan92] point of comparison for on-line algorithms, and stimulate researchers to find on-line approximations [SP83]. 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 [BCT94] 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 [Fis83, HP96].

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. A load immediate is considered to cost half of a memory access by some authors [BCT94].

2.2 Local Register Allocation

We now formulate the problem of local register allocation. Let $V = \{1, 2, \ldots, M\}$ be a set of *program variables*. At each step in the basic block, one or more read and write accesses to the program variables in V can be generated. Let $T = \{\mathsf{read}, \mathsf{write}\}$ and define a read or write request to a program variable as an element of $V \times T$. For the sake of simplicity and consistently with the previous literature, (i, write) will be denoted by i^* and (i, read) simply as i. At each step, a set of at most α requests to the program variables in V can be issued. We will say that a *reference* is a subset of $V \times T$ of size at most α. In typical applications $\alpha = 2$, but we do not fix an *a priori* bound on α. In write-back paging, $\alpha = 1$. The set of all possible references is denoted by R. A first input to the local register allocation problem is a *trace*, which is defined as a sequence of references of some positive length r. We will use the following notation: if $\sigma \in R^r$ is a trace then the jth element of σ is denoted by σ_j and σ can be written as $\sigma = \langle \sigma_1\ \sigma_2\ \ldots \sigma_r \rangle$. Define $s = \sum_{j=1}^{r} |\sigma_j|$. We will restrict our attention to a proper subset of R^r defined by the following conditions:

(Non-simultaneity) A program variable cannot be read and written at the same time, and,

(Write-Once) If a program variable i is written at step j, then

- i is not written at any other step and
- i is read only after step j.

The non-simultaneity condition is always enforced in applications. The write-once condition will be shown to hold without loss of generality when we introduce the cost model.

Program variables can be partioned into those that are written at some step of the basic block and variables that are never written and only read. For notational simplicity, we will assume that the program variables $1, 2, \ldots, m \leq M$ are written at some point in σ, and the program variables $m+1, m+2, \ldots, M$ are only read. Moreover, we will assume that all program variables are referenced, that is, for all $i \in V$, there is a j with $1 \leq j \leq r$ and an $s \in T$ such that $(i, s) \in \sigma_j$.

Notice that while a program variable $i \leq m$ is written at some trace step, we do not impose that i be read anywhere else. Indeed, program variables might hold values that are computed in the current basic block and used only in subsequent basic blocks. Although those variables are not currently needed, their value has to be maintained for future reference. A program variable is said to be *alive on exit* if its value has to be maintained across basic block boundaries. The set of variables alive on exit is denoted by L. The variables alive on exit are determined by a live range analysis procedure [ASU86, FL88, TS85].

We will denote with $\lambda_0(i)$ the first step where program variable i is referenced: $\lambda_0(i) = \min\{j : \exists s \in T.(i, s) \in \sigma_j\}$. We will denote with $\lambda(i)$ the last step where program variable i is referenced: $\lambda(i) = \max\{j : \exists s \in T.(i, s) \in \sigma_j\}$. We will say that a program variable i is alive at step j if either $i \in L$ or $\lambda(i) \geq j$.

We turn now to model the memory device that is used to satisfy the trace. Let $S = \{\mathsf{clean}, \mathsf{dirty}\}$ be the set of statuses of the program variables in the registers. If a program variable is clean, then its contents in register and in memory are identical, otherwise, they are different. Let N be the number of register and define a *register configuration* Q as a subset of $V \times S$ of size at most N with the property that if $(i, s) \in Q$ and $t \neq s$ then $(i, t) \notin Q$. A memory configuration represents the program variables that are currently in a register and their status. For the sake of simplicity and consistently with the previous literature, (i, dirty) will be denoted by i^+ and (i, clean) simply as i.

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

- If $i \in \sigma_j$, then either $i \in Q_j$ or $i^+ \in Q_j$;
- If $i^* \in \sigma_j$, then $i^+ \in Q_j$ and Q_h contains neither i nor i^+ for all $h = 0, 1, \ldots, j-1$.

We define an allocation strategy as a rule that associates a register allocation to a trace σ.

We now describe our cost model. If a program variable i is requested and it is not in the register configuration, 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. If the value of i' is inconsistent with the value of i' in secondary storage, then i' has to be stored, at a cost of $S_{i'}$. 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.

Suppose that a program variable i 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 [HP96]. Compulsory misses cause a fixed cost that any allocation has to pay. Therefore, compulsory misses can be disregarded while seeking an optimum allocation. Moreover, if compulsory misses are disregarded, the cost of an allocation does not depend on the initial register configuration Q_0. Henceforth, compulsory misses will be ignored, and we will consider only non-compulsory or, in the terminology of [HP96], *capacity misses*.

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, $C \simeq 2$ typically [BCT94]. The cost w^* of changing the contents of one register from $(i, s) \in X \times S$ to $(h, t) \in X \times S$ at step j is

$$w^*[j, (i, s), (h, t)] = \begin{cases} S(i, s) & \text{if } i = h \text{ and } t = \mathsf{clean} \text{ or if } i \neq h \text{ and } j = \lambda_0(h) \\ S(i, s) + S_i & \text{if } i \neq h \text{ and } j > \lambda_0(h) \\ 0 & \text{otherwise} \end{cases}$$

where $S(i, s)$ is the costs of storing (i, s):

$$S(i, s) = \begin{cases} S_i & \text{if } i \text{ is alive and } s = \mathsf{dirty} \\ 0 & \text{otherwise} \end{cases}$$

We will say that the allocation $\mathcal{Q}$ *stores* i^+ at step j if $i^+ \in Q_{j-1} - Q_j$. We will say that the allocation $\mathcal{Q}$ *evicts* $(i, s) \in X \times S$ at step j if $(i, s) \in Q_{j-1} - Q_j$.

Let $Q_1 = \{q_1^1, q_2^1, \ldots, q_N^1\}$ and $Q_2 = \{q_1^2, q_2^2, \ldots, q_N^2\}$ be two register configurations. We will assume that the elements of Q_1 and Q_2 are numbered so that the same subscript is given to an element of Q_1 and an element of Q_2 that contain the same program variable, that is, if $q_h^1 = (i, s)$, then either $q_h^2 = (i, t)$ or, for all $t \in S$, $(i, t) \notin Q_2$. The cost of the transition from Q_1 to Q_2 at step j is given by

$$w(j, Q_1, Q_2) = \sum_{h=1}^{N} w^* \left[j, q_h^1, q_h^2 \right] .$$

We define the cost of $\mathcal{Q}$ as

$$c(\mathcal{Q}) = \sum_{j=1}^{r} w(j, Q_{j-1}, Q_j) .$$

The *optimum register allocation* $\mathcal{Q}$ from Q_0 is the one whose first configuration is Q_0 and that minimizes $c(\mathcal{Q})$.

We will now return to the write-once condition. First, we need to recall the notion of live ranges. A *live range* for the program variable i is an interval $[j_0, j_1]$ such that $i^* \in \sigma_{j_0}$ and j_1 is the last step where i is alive and has not been rewritten at any step j' with $j_0 < j' \leq j_1$. Intuitively, we will enforce the condition that every program variable is written only once by replacing a program variable with the collection of its live ranges. More precisely, replace a program variable i with a collection of newly introduced program variables each of which is alive during exactly one live range of i. Clearly, no program variable is written more than once. A simple induction shows that the new problem is equivalent to the original one.

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 $\alpha = C = 1$.

2.3 New Results

In this paper, we consider the following local register allocation problem:

Instance: A trace σ and a positive integer N

Solution: A minimum cost register allocation for σ with N registers

Our main results are as follows.

Theorem 2.1 *The decision problem corresponding to local register allocation (i.e. given a trace σ and two positive integers N and z, decide whether there is a register allocation for σ with N registers that costs no more than z) is NP-complete even for the particular case of off-line write-back paging.*

Theorem 2.2 *There is a polynomial-time approximation algorithm for local register allocation with performance guarantee of* 2.

Theorem 2.3 *There is a $(2C)$-approximation algorithm for local register allocation that runs in $\widetilde{O}(s)$ time.*

No hardness result and no performance guarantee were previously known. In the regular paper, we will give only sketches or intuitive explanations of these results and defer the complete proofs to the appendix.

3 Hardness of Off-Line Paging

In this section, we will discuss theorem 2.1 and give some insight as to why write-back paging is a hard problem. At the same time, we will give some definitions and make some observation to introduce the topics of the next section.

Write-back paging is much harder than paging without write-backs. Indeed, the latter can be optimally solved by the *furthest-first rule*: in response to a page fault, evict the page that will be referenced furthest in the future [Bel66, HKMW66] (see also [MS91] for a shorter proof). Consequently, the minimum number of page faults can be found in $O(r \log\log N)$ by an implementation of furthest-first with Thorup's priority queues [Tho96]. By contrast, write-back paging is proved to be NP-hard. Since the optimum value of write-back paging is a non-negative integer value less than or equal to $2r$, write-back paging does not admit a fully polynomial time approximation scheme unless P = NP.

The difference between paging with and without write-backs can be intuitively explained in terms of the following observations and definitions.

If $i \leq m$, a cost $S(i, \mathsf{dirty})$ is charged for storing i independently of the step j where the store occurs as long as i is alive and dirty at step j. It follows that if a program variable $i \leq m$ 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.

Notice that it is legal to evict a program variable i at step j if either i is not alive at step j, or i is alive, but is not requested at step j. Define a *value range* of a program variable i to be a maximal sequence of steps where i is alive and it is legal to evict it. We claim that if a program variable i is ever evicted along its kth value range, then we can assume without loss of generality that i is evicted at the beginning of the kth value range. Indeed, if we pay a cost S_i for reloading i at the end of the value range, the same cost S_i is charged independently of the step where i had been evicted. So, the decision of evicting program variable i along its kth value range is a binary decision variable.

We are now able to give some insight as to why write-backs make the off-line paging problem hard. The cost of an allocation breaks down into the cost for loading and the cost for storing. The cost for loading is proportional to the number of times a page is evicted while alive. However, the cost for storing is a fixed cost that is paid once or not at all. If a cost is charged for storing, that cost does not depend on the number of times a page is actually evicted. An optimum allocation is hard to find because of the fixed costs due to stores.

4 Algorithms

In this section, we will present our algorithms for finding optimum and approximate local register allocations. We first formulate local register allocation as a 2-commodity minimum cost flow problem with integrality constraints. The formulation is instrumental in obtaining our algorithm. As an intermediate step, we will describe a new algorithm for weighted off-line caching. Then, we will describe a 2-approximation algorithm for the local register allocation problem. Finally, we will then describe a tie-breaking rule for furthest-first which yields a $(2C)$-approximation algorithm that runs in almost linear time.

4.1 A 2-Approximation Algorithm

An Integer 2-commodity Formulation In this section, we will formulate the problem of local register allocation as a minimum cost 2-commodity flow problem with integrality side constraints. Our formulation leads to an approximation algorithm for the local register allocation problem. Moreover, we feel that the formulation is of independent practical interest. Indeed, we used the formulation in a branch-and-bound algorithm to solve exactly the local register allocation problem. The resulting procedure is very fast on real programs and, in contrast with the dynamic programming algorithm [HFG89], it takes a polynomial amount of space[1]. As such, the formulation and the related branch-and-bound algorithm constitutes a substantial practical improvement over previous implementations of the dynamic programming algorithm.

Let A_j be the set of pairs (i,k) with the property that the kth value range of i contains step j. Let

$$[x]^+ = \frac{|x|+x}{2} = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

and $\rho_j = [|A_j| + |\sigma_j| - N]^+$. The quantity ρ_j gives the number of live program variable that have to be out of the register configuration at step j. The quantity ρ_j will be said to be the *register pressure* at step j.

We can now describe the integer 2-commodity flow problem. We will denote with $\mathcal{N} = (N, A)$ the network that defines the problem, and with $\mathcal{N}_1$ and $\mathcal{N}_2$ the subnetworks of $\mathcal{N}$ where the first and the second commodity are respectively confined.

A partial example of this formulation is shown in figure 1 and will be explained later. We will use the following notation: if $e = (i,j) \in A$ is an arc, then $\mathsf{head}(e) = j$ and $\mathsf{tail}(e) = i$. If $e \in A$, then $x^i(e)$ $(i = 1,2)$ is the flow of the ith commodity through arc e.

First, we will model the cost due only to loads. The construction determines a subnetwork of the type of $\mathcal{N}_1$ in the example in figure 1. Introduce a path $t_r \to t_{r-1} \to \cdots \to t_1$ whose arcs correspond to the steps of the trace, and let $\mathsf{tail}(t_j) = v_j$ $(1 \le j \le r)$ and $\mathsf{head}(t_1) = v_0$. We also insert a supply at node j equal to $b_j = \rho_j - \rho_{j-1}$. We now insert arcs corresponding to value ranges. The number of distinct value ranges of program variable i will be denoted by $K(i)$. We will also consider the collection of value ranges of a program

[1] The branch-and-bound algorithm and experimental results are reported in the appendix.

variable i as ordered by starting point. Then, if the kth value range of i ($m < i \leq M$) starts at step v_{j_0} and ends at step v_{j_1} we have the three arcs $(v_{j_0}, v_{ik}), l_{ik} = (v_{ik}, v'_{ik})$, and $l''_{ik} = (v'_{ik}, v_{j_1})$, where v_{ik} and v'_{ik} are newly introduced nodes. The arcs l_{ik} (for $i = 1, 2, \ldots, M$, $k = 1, 2, \ldots, K(i)$) will be said to be value range arcs. Value range arcs have unit capacity for the first commodity and all others arcs have infinite capacity. We will later impose capacities that confine the first commodity flow to $\mathcal{N}_1$.

We will interpret the flow $x^1(l_{ik})$ as follows: $x^1(l_{ik}) = 1$ if and only if program variable i will be evicted along its kth value range. The construction is justified as follows. By definition of register pressure, there are at least ρ_j program variables that are alive at step j, but that do not reside in the register configuration at the same step j. In other words, there are at least ρ_j pairs (i, k) such that i is evicted along the kth value range arc and the kth value range of i contains step j. If (i, k) is one of these pairs, then we have $x^1(l_{ik}) = 1$. It is now routine to check that the values of the supplies on the nodes $v_0, v_1, \ldots, v_r$ force the right number of value range arcs l_{ik} to carry a unit of flow. Consequently, the right number of program variables are out of the register configuration at step j. We can now define the cost $c^1(l_{ik})$ of the arc l_{ik} for the first commodity. If i is evicted during its kth value range and it has to be reloaded at the end of it, then we charge a cost $c^1(l_{ik}) = S_i$. However, if $k = K(i)$ and $i \in L$, program variable i will not be reloaded, and $c^1(l_{ik}) = 0$.

The second subnetwork $\mathcal{N}_2$ accounts for the cost of stores. We introduce a second commodity to model the cost of storing i. The value range arc l_{ik} has unit bundle capacity. The arc l''_{ik} has zero capacity for the second commodity, which has the effect of confining the second commodity to $\mathcal{N}_2$. Then, for $i = 1, 2, \ldots, m$, we join the arcs l_{ik} corresponding to program variable i into a path $l'_{i1} \to l_{i1} \to l'_{i2} \to \ldots \to l_{iK(i)} \to l'_{iK(i)}$, where the l'_{ik}'s are newly introduced arcs. The arc l'_{ik} has zero capacity for the first commodity, which has the effect of confining the first commodity to $\mathcal{N}_1$. Moreover, we introduce an arc $l_i = (\mathsf{tail}(l'_{i1}), \mathsf{head}(l'_{iK(i)}))$ with cost S_i. We place a unit supply of the second commodity at $\mathsf{tail}(l'_{i1})$ and a unit demand of the second capacity at $\mathsf{head}(l'_{iK(i)})$. All other costs are taken to be zero and all other capacities are taken to be infinite. The resulting construction corresponds to the network $\mathcal{N}_2$ in figure 1 and can be justified as follows. Given a program variable $i \leq m$, if there is some k with $x^1(l_{ik}) = 1$, then $x^2(l_i) = 1$, and the cost S_i is charged. But if $x^1(l_{ik}) = 1$, then i is evicted, and so it is stored for a cost S_i.

Finally, we impose unit individual capacity on the value range arcs. The individual capacities are redundant, but we introduce them because later we will decouple the problem and solve two single-commodity network flow problems.

In figure 1, we give the part of the graph corresponding to a program variable i that has two value ranges extending over step 4 only and from step 6 to 9. In the picture, u^h is the capacity for commodity h and u is the bundle capacity; omitted values denote an infinite capacity. Finally, we omit supply and demands of the first commodity and individual capacities, and we let b^2 be the value of demand of the second commodity.

We also notice that $|A| = O(s)$ and $|N| = O(s)$.

Weighted Caching The problem of weighted caching is the local register allocation problem for $\alpha = 1$ and $m = 0$. 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 [MMS90]. Intuitively, weighted off-line caching is much simpler than write-back paging because the costs do not depend on the page status. The subnetwork $\mathcal{N}_1$ in figure 1 gives an example of minimum cost network flow problems arising from weighted off-line caching.

Since $\alpha = 1$, the register pressure never increases by more than one. It follows that the maximum supply is one. Consequently, weighted off-line weighted caching can be solved in $O(r^2 \log\log r)$ time with the successive shortest path algorithm [AMO93, Tho96]. No explicit algorithm had been previously given. However, the construction in [You94] relies on algorithms for the assignment problem and implicitly yields algorithms that run in $O(r^2\sqrt{r}\log(rC))$ time and in strongly polynomial time in $O(r^3)$ [AMO93].

The 2-Approximation Algorithm We will now extend the algorithm for weighted off-line caching to a polynomial time 2-approximation algorithm for the general local register allocation problem.

The gist of the algorithm is to distribute the cost S_i of storing program variable i among all the value

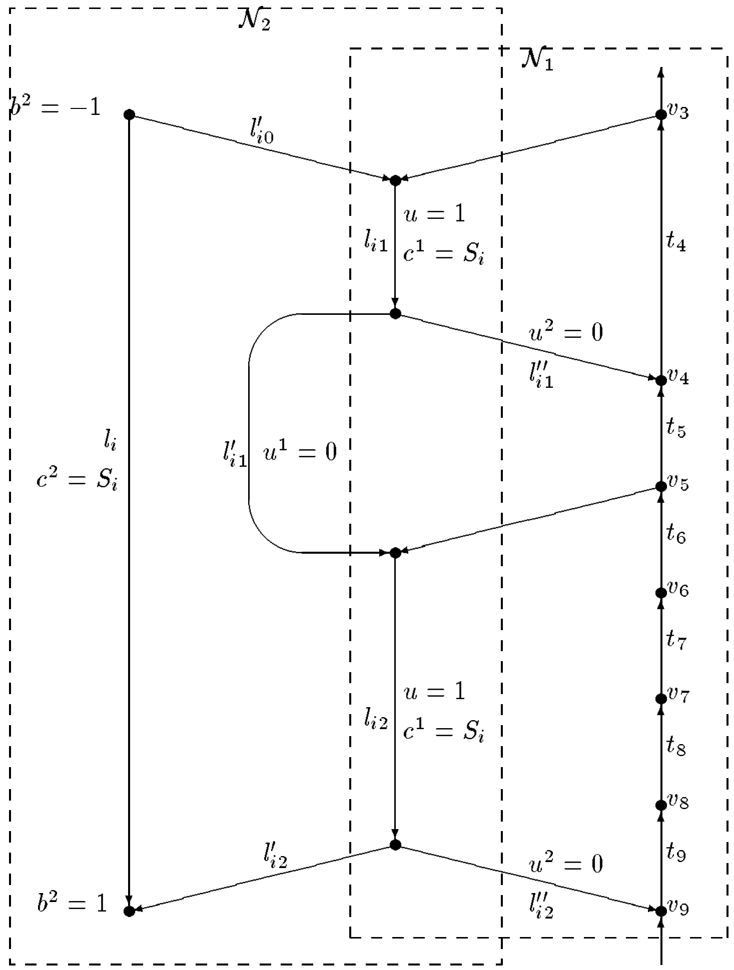

Figure 1: Part of the multicommodity flow problem relative to one program variable

ranges of i, and then solve the resulting weighted off-line caching problem. However, the algorithm gives a greater price to value ranges whose load cost is zero in order to balance the cost of loads and stores. Define $K = \max\{K(i) : 1 \leq i \leq m\}$ and

$$w(l_{ik}) = \begin{cases} \frac{1}{2K} S_i & \text{if } k > 1 \\ \frac{2K - K(i) + 1}{2K} S_i & \text{otherwise} \end{cases} \tag{1}$$

Then, solve the minimum cost network flow problem on $\mathcal{N}_1$ with costs $c^1(l_{ik})$ replaced by $c^1(l_{ik}) + w(l_{ik})$. Let $\hat{x}^1$ be the resulting flow of the first commodity. We now turn to define the corresponding flow for the second commodity. We will store a program variable i if there is a value range of i along which i is evicted. Let $S = \{i : 0 \leq i \leq m, \exists k \in \{1, 2, \ldots, K(i)\}.\hat{x}^1(l_{ik}) = 1\}$ be the set of stored program variables and define

$$\hat{x}^2(l_i) = \begin{cases} 1 & \text{if } i \in S \\ 0 & \text{otherwise} \end{cases}$$

We claim that this algorithm establish theorem 2.2. The proof is based on a Lagrangian relaxation of the 2-commodity flow problem. The rest of the proof is deferred to the appendix.

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 s. The running time of the algorithm is $\widetilde{O}(s^2)$ [AMO93].

4.2 Fast Approximation

In the case of off-line paging, if we consider both compulsory and capacity costs, then furthest-first is a 2-approximation algorithm: for any load, we can at most charge an extra store. We will propose below a simple modification of furthest-first that achieves a performance guarantee of 2 even if we account only for capacity misses. In fact, our algorithm has a performance guarantee of $2C$, and thus it establishes[2] theorem 2.3.

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 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 alive. If all program variables in F are alive, evict a variable in F that is clean. If all program variables in F are alive and dirty, evict one variable in F arbitrarily.

We compared the approximation algorithms $\mathcal{W}$ and CFF on real benchmarks[3]. The algorithms produce on average comparable allocations and are within 1% of the optimum for most choices of N. The worst-case allocation of $\mathcal{W}$ is typically better than the worst-case allocation of CFF. On the other hand, $\mathcal{W}$ is much slower.

Acknowledgements

Uli Kremer has been an invaluable sources of information on the issues arising in existing optimizing compilers. Rakesh Barve pointed out to us reference [You94]. We thank Sri Divakaran, Mike Grigoriadis, Sampath Kannan, Leonid Khachyian, Sanjeev Khanna, Lorant Porkolab, Barbara Ryder, Jorge Villavicencio, and Shiyu Zhou for helpful discussions.

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.

[BBB+57] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, 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*, pages 188–198, 1957.

[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.

[2] The proof is found in the appendix.
[3] Detailed experimental results are reported in the appendix.

[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.

[Car84] Richard William Carr. *Virtual Memory Management*, volume 20 of *Computer Science: Systems Programming*. UMI Research Press, Ann Arbor, MI, 1984.

[CK91] David Callahan and Brian Koblenz. Register allocation via hierarchical graph coloring. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Languages Design and Implementation*, pages 192–203, June 1991.

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

[Fis83] J. A. Fisher. Very long instruction word architectures and ELI-512. In *Proc. Tenth Symposium on Computer Architecture*, pages 140–150, June 1983.

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

[FR97] Amos Fiat and Ziv Rosen. Experimental studies of access graph based heuristics: Beating the LRU standard? In *Proceedings of the Eight Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 63–73, 1997.

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

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

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

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

[IKP96] Sandy Irani, Anna R. Karlin, and Steven Phillips. Strongly competitive algorithms for paging with locality of reference. *SIAM J. Comput.*, 25(3):477–497, June 1996.

[K$^+$86] 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.

[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.

[MMS90] M. S. Manasse, L. A. McGeoch, and D. D. Sleator. Competitive algorithms for server problems. *Journal of Algorithms*, 11:208–230, 1990.

[Mor91] W. G. Morris. CCG: A prototype coagulating code generator. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Languages Design and Implementation*, pages 45–58, June 1991.

[MS91] Lyle A. McGeoch and Daniel D. Sleator. A strongly competitive randomized paging algorithm. *Algorithmica*, 6:816–825, 1991.

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

[NW88] George L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization.* Wiley, New York, 1988.

[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.

[SP83] Abraham Silberschatz and James L. Peterson. *Operating Systems Concepts.* Addison-Wesley, Reading, MA, 1983.

[ST85] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, 1985.

[Tan92] Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, Englewood Cliffs, NJ, 1992.

[Tho96] Mikkel Thorup. On RAM priority queues. In *Proc. Seventh ACM-SIAM Symposium on Discrete Algorithms*, pages 59–67, 1996.

[TS85] Jean-Paul Tremblay and Paul G. Sorenson. *The Theory and Practice of Compiler Writing.* McGraw Hill, New York, 1985.

[You94] Neal Young. The K-server dual and loose competitiveness for paging. *Algorithmica*, 11(6):525–541, 1994.

A Hardness of Off-Line Paging

In this section we prove

Theorem 2.1 *The decision problem corresponding to local register allocation (i.e. given a trace 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. First of all, we will show that the problem is NP-hard for $\alpha = N$, then we will show the result for $\alpha = 1$.

The reduction is from set cover: given a collection $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ of subsets of $S_0 = \{1, 2, \ldots, n\}$, 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 $j \notin S$.

The set cover notation is consistent with the reduction. Indeed, we will have program variables numbered from 1 to m, and for those pages the first access is a write. We introduce a program variable $p_j = m + j$ for each $j \in S_0$, so that $M = m + n$. Let $r_j = \{p : j \notin S_p\} \cup \{p_j\}$ for $1 \leq j \leq n$, $\sigma_1 = \langle 1^* \ 2^* \ldots m^* \ r_j \rangle$, and $\sigma_j = \langle 1 \ 2 \ldots m \ r_j \rangle$ for $2 \leq j \leq n$. The trace is $\sigma = \sigma_1 \cdot \sigma_2 \cdot \ldots \cdot \sigma_n \cdot \langle 1 \ 2 \ldots m \rangle$. Finally, $L = P$ and $N = m$. Clearly, the size of the problem is still polynomial in the size of the input.

We will show that every set cover of size C corresponds to an allocation of cost $C + n$, and that conversely an allocation of cost $C + n$ corresponds to a cover of size at most C.

First, notice that, after the request r_j, the program variable p_j can be evicted for free. It follows that σ can be satisfied by evicting one program variable at every request r_j. Given a set cover $\mathcal{S}'$ of size C, we construct an allocation as follows. We store all the program variables p with $S_p \in \mathcal{S}'$, and for every r_j, we choose to evict a page p with $j \in S_p \in \mathcal{S}'$ in any arbitrary way. The cost due to stores is exactly C. The evicted variable has to be reloaded at the end of σ_j, and the cost due to loads is exactly n. Hence, the total cost of the allocation is $C + n$. Conversely, suppose that there is an allocation $\mathcal{Q}$ with $c(\mathcal{Q}) \leq C + n$. Notice that any allocation evicts at least one program variable during each σ_j. We choose in an arbitrary way a program variable q_j among those that are evicted during σ_j, and we repeat the process for all j's. Finally, we replace the original allocation with one that evicts only q_j during σ_j. The new allocation is feasible and its cost is no more than that of the original one. But the new allocation immediately corresponds to a set cover of the given size, and the result follows.

We now modify the previous construction for the case $\alpha = 1$. A request $r_j = \{i_1, i_2, \ldots, i_{|r_j|}\}$ is mapped to a subsequence $r'_j = (r''_j)^4$, where r''_j is the sequence $r''_j = \langle i_1 \ i_2 \ldots i_{|r_j|} \rangle$. Call the resulting sequence σ'. The intuition behind this substitution is the following. Our objective is to impose the condition that all program variables in r_j reside in the register configuration during the subsequence r'_j. To this end, we make it very expensive for an allocation to evict any program variable in r_j during r'_j. Indeed, we request the program variables in r_j many times in a round-robin fashion. Then, it becomes very expensive for an allocation to repeatedly swap out and back in the program variables in r_j. Consequently, a cheap allocation will always choose a program variable not in r_j to be evicted during r'_j, and all pages in r_j will reside in the register configuration during r'_j.

First, we show that the previous discussion still yields a feasible allocation once the set cover is given. Observe that σ' can be satisfied by evicting one program variable $p \notin r_j$ during each r'_j. The set cover determines a set of program variables that are stored, and we choose among these program variables the one to evict during r'_j.

It remains to show that an allocation $\mathcal{Q}'$ for σ' corresponds to an allocation $\mathcal{Q}$ for σ with $c(\mathcal{Q}) \leq c(\mathcal{Q}')$. Once a program variable $p \in r_j$ is evicted, all program variables in r_j can be accommodated in the register configuration and no further eviction is necessary during r'_j. Suppose that during r'_j the allocation $\mathcal{Q}'$ evicts a program variable $p \notin r_j$ and also a program variable $p' \in r_j$. Then, the eviction of a program variable $p' \in r_j$ during r'_j can be avoided without losing feasibility and without increasing the cost. Suppose now that the allocation $\mathcal{Q}'$ does the opposite and evicts a program variable $p' \in r_j$ during r'_j. The previous discussion implies that no program variable $p \notin r_j$ is evicted during r'_j, or else we could obtain a cheaper allocation by evicting a program variable $p \notin r_j$ rather than p'. Hence, during the three last repetitions of r''_j, there is a program variable that has to be reloaded. So, the total cost on r'_j is at least 3. But then it would have been cheaper to evict a program variable not in r_j for a cost of at most 2. It follows that we can obtain an allocation that during r'_j evicts a program variable not in r_j and that does not cost more than the original one.

Suppose that there is an allocation $\mathcal{Q}'$ for σ' that costs no more than $C+n$. Hence, $\mathcal{Q}'$ induces a schedule $\mathcal{Q}$ for σ with $c(\mathcal{Q}) \le c(\mathcal{Q}') = C+n$. In turn, we would obtain a set cover of size less than C. We have thus reached a contradiction and the proof is complete. □

B The Approximation Algorithm

We now conduct the analysis of the approximation algorithm in section 4.1.

First, notice that the costs are now no more than $CK = O(rC)$, and the precision needed to carry out the computation is increased only by a $O(\log r)$ additive term.

Note that the flow $\hat{x}$ retuned by the algorithm is integer and feasible for the multicommodity flow problem. Hence, the flow $\hat{x}$ determines an allocation $\mathcal{W}$. The flow $\hat{x}^1$ of the first commodity is optimum with respect to the prices w by definition of the algorithm. The flow $\hat{x}^2$ of the second commodity is optimum with respect to the prices w because $\sum_{k=1}^{K(i)} w(l_{ik}) = S_i$. Let $z_1^*(w) = \sum_{e \in A} c^1(e)\hat{x}^1(e)$ be the cost of the algorithm for the first commodity, that is, the cost due to loads only. Let $z_2^*(w) = \sum_{i \in S} S_i$ be the cost for the second commodity, that is, the cost due to stores only. The total cost paid by the algorithm is $c(\mathcal{W}) = z_1^*(w)+z_2^*(w)$. 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 program variables not in S into two subsets S^0 and S^1 with $S^0 = \{i \in S : \hat{x}^1(l_{iK(i)}) = 0\}$ and $S^1 = \{i \in S : \hat{x}^1(l_{iK(i)}) = 1\}$. Correspondingly, let $z_{h1}^*(w) = \sum_{i,k: i \in S^h} c^1(l_{ik})\hat{x}^1(l_{ik})$ and $z_{h2}^*(w) = \sum_{i \in S^h} S_i$ for $h = 0, 1$. Notice that $z_{01}^*(w) \ge z_{02}^*(w)$ because for a program variable $i \in S^0$ there is a value range $k < K(i)$ with $x^1(l_{ik}) = 1$ to cause $i \in S$. We have now succeeded in bounding $z_2^*(w)$ in terms of $z_1^*(w)$ with respect to S^0 only, and this will be enough for the proof.

Let $z_{21}^*(w) = \sum_{i=m+1}^{M} \sum_{k=1}^{K(i)} c^1(l_{ik})\hat{x}^1(l_{ik})$ be the cost due to the program variables that do not have to be stored. On the whole, $z_1^*(w) = z_{01}^*(w) + z_{11}^*(w) + z_{21}^*(w)$.

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. The resulting price on arc e is denoted by $w(e)$. Therefore, the cost of the optimum allocation OPT is at least equal to

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

If $i \in S^0$, then

$$\sum_{k: \hat{x}^1(l_{ik}) + \hat{x}^2(l_{ik}) = 0} w(l_{ik}) \le \left(1 - \frac{1}{2K}\right) S_i$$

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

If $i \in S^1$, then

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

because at least $\hat{x}^1(l_{iK(i)}) = 1$.

Finally, if $i \le m$ and $i \notin S$, then

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

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

Then,

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

Define $\beta = \frac{1}{2K} > 0$ and $\gamma = \frac{2K-K(i)+1}{2K} \geq \frac{1}{2}$. The cost ratio is at most

$$\frac{c(\mathcal{W})}{c(\mathrm{OPT})} \leq \frac{z_{01}^*(w) + z_{02}^*(w) + z_{11}^*(w) + z_{12}^*(w) + z_{21}^*(w)}{z_{01}^*(w) + \beta z_{02}^*(w) + z_{11}^*(w) + \gamma z_{12}^*(w) + z_{21}^*(w)} .$$

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

$$\frac{c(\mathcal{W})}{c(\mathrm{OPT})} \leq \frac{2z_{02}^*(w) + z_{12}^*(w)}{(1+\beta) z_{02}^*(w) + \gamma z_{12}^*(w)} \leq \frac{2z_{02}^*(w) + z_{12}^*(w)}{z_{02}^*(w) + \frac{1}{2} z_{12}^*(w)} = 2 ,$$

which completes the proof of theorem 2.2.

C Fast Approximation

We will now prove that CFF is a $(2C)$-approximation algorithm for local register allocation. We will assume familiarity with the proof of optimality of furthest-first [MS91].

If program variable i is evicted by CFF and $\bar{k}$ is defined as the index of the first value range that CFF evicts. If program variable i is not evicted by CFF, then let $\bar{k} = K(i)$. Define the prices as $w(l_{i\bar{k}}) = S_i$ and $w(l_{ik}) = 0$ for all $k \neq \bar{k}$. CFF induces a first commodity flow on $\mathcal{N}_1$. The cost of that flow with respect to the costs $c^1(e) + w(e)$ is exactly equal to $c(\mathrm{CFF})$. Let $\tilde{x}^1$ to be the optimum flow in $\mathcal{N}_1$ with respect to the costs $c^1(e) + w(e)$ and let $\tilde{c}$ to be the cost of that flow again with respect to the costs $c^1(e) + w(e)$. The flow $\tilde{x}^1$ is integer without loss of generality. Therefore, $\tilde{x}^1$ and $\tilde{x}^2(l_i) = \max\{\tilde{x}^1(l_{ik}) : 1 \leq k \leq K(i)\}$ define an allocation $\widetilde{\mathcal{Q}}$. The optimum cost is at least equal to the value of the Lagrangian relaxation with prices w, which in turn is exactly equal to $\tilde{c}$. It remains to show that $c(\mathrm{CFF}) \leq 2C\tilde{c}$.

Let $\mathcal{Q}$ be an allocation. Define the eviction points of $\mathcal{Q}$ to be the step indexes where $\mathcal{Q}$ evicts a program variable from the register configuration. Let $ep(\mathcal{Q})$ be the number of eviction points of $\mathcal{Q}$. Let $d(\mathcal{Q})$ be the number of eviction points where $\mathcal{Q}$ evicts a program variable that is not alive. The allocation $\mathcal{Q}$ induces a flow on $\mathcal{N}_1$. The quantity $ep(\mathcal{Q}) - d(\mathcal{Q})$ gives the number of value range arcs that carry positive flow. Let $a(\mathcal{Q})$ be the number of eviction points where $\mathcal{Q}$ evicts a program variable that is alive and not requested ever again in the future and clean at that point for CFF. The quantity $ep(\mathcal{Q}) - (d(\mathcal{Q}) + a(\mathcal{Q}))$ gives the number of value range arcs e that carry positive flow and that have positive cost $c^1(e) + w(e) > 0$.

By optimality of furthest-first, CFF has the least possible number of eviction points. Therefore $ep(\widetilde{\mathcal{Q}}) \geq ep(\mathrm{CFF})$. Let $f_1 < f_2 < \ldots < f_h$ ($h = ep(\mathrm{CFF})$) be the eviction points of CFF and $g_1 < g_2 < \ldots < g_h$ be the first h eviction points of $\widetilde{\mathcal{Q}}$. By the proof in [MS91], $f_j \geq g_j$ for $j = 1, 2, \ldots, h$. Therefore, $d(\widetilde{\mathcal{Q}}) + a(\widetilde{\mathcal{Q}}) \leq d(\mathrm{CFF}) + a(\mathrm{CFF})$. It follows that the number of paid arcs that carry flow for CFF is no more than the number of paid arcs that carry flow for $\widetilde{\mathcal{Q}}$. Since the cost of a paid arc is between 1 and $2C$, $c(\mathrm{CFF}) \leq 2C\tilde{c}$, which was to be proved in theorem 2.3.

We can also show that our analysis is tight. Let $\sigma = \langle \{1^*\}\{3\}\{2^*\}\{2\}\{3\}\{1\} \rangle$, 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 $S_3 = 1$.

D Experimental Results

D.1 Experimental Setup

We used the ILOC compiler [BCT94] (see also `http://softlib.rice.edu/MSCP/iloc.html`) to generate code for the 4 programs of the SPEC benchmarks that are available to us: doduc, fpppp, matrix300, and tomcatv. We applied the following optimizations from the ILOC library in the given order: operator strength reduction, constant propagation, partial redundancy elimination, and dead code elimination. We based our experiments on highly optimized object code because register allocation is usually the last pass of a optimizing compiler.

The resulting code was processed to extract integer and floating point traces. ILOC does not assign a type to program variables. We had to make our own assumptions on the type of variables. We decided that a variable was a floating point variable if it had not previously defined as integer.

During the extraction of traces, we had the additional problem of having to separate local allocation from global decisions. We assumed that all variables alive at the entry point of a basic block were dirty and we enforced our choice by means of a preamble of write references to variables live at the entry point.

We took $S_i = 1$ for load immediates and $S_i = 2$ otherwise. The resulting cost function is analogous to that in [BCT94]. We did experiments for $N = 2, 4, 8, 16, \ldots, 256$.

The following tables report the result of the experiments. The tables include data only for the basic blocks and the choices of N for which the number of registers N was insufficient to accommodate all variables. The first table report characteristics of the traces. Column 1 gives the benchmark name and column 2 gives the number N of registers. Column 3 and 4 report characteristics of the floating point traces. Column 3 gives the number of basic blocks and column 4 gives their average length. Columns 5 and 6 report the same data for integer traces.

D.2 Approximation Algorithms

In this section, we report experimental results on the approximation algorithms. We implemented CFF with binary heaps. We implemented $\mathcal{W}$ in two different ways. First, we used the successive shortest paths algorithm, where each shortest path is computed by an implementation of Dijkstra with binary heaps. In this case, we assigned prices as integer multiples of K. Our prices distribution differs from the one above. If a program variable i had a value range arc with no load cost, then we assigned a price $S_1/2 = 1$ to it. The remaining store cost is subdivided as equally as possible among all other value ranges. It is easy to see that the new price distribution does not worsen the performance guarantee. We also implemented $\mathcal{W}$ via a call to the CPLEX network optimizer. Here, we rounded the prices to their double precision representation. Again, we balanced the prices among all the value ranges of a program variable. The two methods produce nearly the same results.

The experimental results are reported in table 2 and 3. Column 1 gives the benchmark name and column 2 gives the number N of registers. Column 3 and 4 report the cost of the allocation produced by $\mathcal{W}$. The cost is given as percentage increase over optimum (i.e. $c(\mathcal{W})/c(\mathrm{OPT}) - 1$). Column 3 gives the average cost increase and column 4 gives the worst-case cost increase. Column 5 and 6 report the same quantity for Conservative Furthest-First. The last set of rows summarizes the experiment outcomes across the benchmarks. Columns 3 and 5 give the geometric average of the cost increase. Columns 4 and 6 give the worst case cost increase.

Finally, we report that CFF is about one or two orders of magnitude faster than the present CPLEX implementation of $\mathcal{W}$. However, careful algorithm enginnering is needed before ruling out $\mathcal{W}$ as a slow algorithm.

In conclusion, $\mathcal{W}$ and CFF have nearly the same performance on average. Both algorithms are within 1% of the optimum for most choices of the number N of registers. CFF is faster than $\mathcal{W}$, but $\mathcal{W}$ has better worst case performance than CFF.

benchmark	N	**Floating point** **blcks**	**Floating point** **avg len**	**Integer** **blcks**	**Integer** **avg len**
doduc	2	1320	43.757	2239	55.872
	4	1092	51.712	2017	59.174
	8	806	65.032	1185	85.766
	16	328	121.23	722	116.54
	32	90	313.19	476	152.94
	64	40	337.88	229	217.72
	128	4	926.25	73	355.44
	256	-	-	7	763.43
fpppp	2	434	110.5	486	207.06
	4	419	114.1	462	188.73
	8	368	128.6	398	216.72
	16	284	161.38	360	237.08
	32	149	278.64	349	243.22
	64	85	443.88	324	257.11
	128	2	12560	224	316.6
	256	1	23002	102	397.92
matrix300	2	1	8	74	32.986
	4	-	-	74	32.986
	8	-	-	73	33.233
	16	-	-	42	41.952
tomcatv	2	32	22.375	120	60.658
	4	11	53.636	118	61.534
	8	4	111.5	104	68.135
	16	1	376	54	113.69
	32	1	376	26	168.92
	64	-	-	4	449.75
	128	-	-	3	393.33

Table 1: Trace characteristics

benchmark	N	$\mathcal{W}$ avg	$\mathcal{W}$ worst	CFF avg	CFF worst
doduc	2	0.00293423%	6.25%	0.0792242%	33.3333%
	4	0.201631%	33.3333%	0.848622%	33.3333%
	8	0.506098%	28.5714%	1.81703%	100%
	16	0.770571%	12.5%	1.27312%	100%
	32	1.38673%	10%	1.36081%	100%
	64	2.34962%	25%	6.20301%	100%
	128	0%	0%	52.7132%	75.8621%
fpppp	2	0%	0%	0.0052063%	5%
	4	0.0579505%	100%	0.326776%	100%
	8	0.124945%	25%	0.324858%	25%
	16	0.0378911%	2.3622%	0.535888%	7.69231%
	32	0%	0%	1.13717%	2.06363%
	64	0%	0%	1.29743%	1.78093%
	128	0.860446%	0.865099%	2.52756%	2.54123%
	256	1.3507%	1.3507%	4.92041%	4.92041%
matrix300	2	0%	0%	0%	0%
tomcatv	2	0.275482%	16.6667%	0.275482%	16.6667%
	4	0%	0%	0.574713%	50%
	8	0.970874%	1.1236%	1.94175%	2.24719%
	16	1.85185%	1.85185%	0%	0%
	32	0%	0%	0%	0%
summary	2	0.0695335%	16.6667%	0.089916%	33.3333%
	4	0.0864913%	100%	0.583141%	100%
	8	0.533377%	28.5714%	1.35854%	100%
	16	0.884022%	12.5%	0.60165%	100%
	32	0.460123%	10%	0.830895%	100%
	64	1.167989%	25%	3.72122%	100%
	128	0.429302%	0.865099%	25.1292%	75.8621%
	256	1.3507%	1.3507%	4.92041%	4.92041%

Table 2: Floating point traces: approximation algorithms

benchmark	N	$\mathcal{W}$ avg	$\mathcal{W}$ worst	**CFF** avg	**CFF** worst
doduc	2	0.109395%	14.2857%	0.224586%	25%
	4	0.222363%	20%	0.235191%	42.1053%
	8	0.252076%	22.2222%	0.199221%	50%
	16	0.223326%	17.6471%	0.154321%	100%
	32	0.0897914%	18.1818%	0.151955%	100%
	64	0.0203057%	25%	0.147942%	100%
	128	0%	0%	0.37357%	100%
	256	0%	0%	0.527704%	4.87805%
fpppp	2	0.0344837%	20%	0.0663148%	25%
	4	0.0494505%	20%	0.109203%	33.3333%
	8	0.0328494%	6.12245%	0.0899788%	5.88235%
	16	0.0294941%	12.1212%	0.0408379%	4.28571%
	32	0.0210668%	5.35714%	0.043819%	5.55556%
	64	0.00104033%	0.214133%	0.0322503%	8.57143%
	128	0.0016452%	0.294985%	0.0345491%	2.32558%
	256	0.00831186%	1.20482%	0.141302%	3.61446%
matrix300	2	0.635122%	8.33333%	0.889171%	8.33333%
	4	0.249792%	5.26316%	0.582848%	4.16667%
	8	2.25847%	33.3333%	2.00753%	14.2857%
	16	0%	0%	0%	0%
tomcatv	2	0.236714%	5.55556%	0.272222%	11.1111%
	4	0.147124%	11.1111%	1.82433%	20%
	8	1.18447%	17.3913%	0.601942%	32.4324%
	16	2.18087%	17.3913%	0.581564%	100%
	32	1.67665%	11.4286%	0.299401%	3.7037%
	64	3.96301%	9.49367%	0.792602%	3.73832%
	128	1.44928%	3.33333%	1.44928%	3.33333%
summary	2	0.253661%	20%	0.362586%	25%
	4	0.167152%	20%	0.685615%	42.1053%
	8	0.928145%	33.3333%	0.721778%	50%
	16	0.604318%	17.6471%	0.193916%	100%
	32	0.592939%	18.1818%	0.165004%	100%
	64	1.013111%	25%	0.323708%	100%
	128	0.481329%	3.3333%	0.617329%	100%
	256	0.004156%	1.20482%	0.334317%	4.87805%

Table 3: Integer traces: approximation algorithms

D.3 Branch and Bound

D.3.1 A Simplified Formulation

We used the 2-commodity network flow formulation in a number of experiments. However, the formulation can be simplified as follows. First, notice that once the flow of the second is fixed to an integer value, the optimum value of the first commodity flow is integer without loss of generality. We can then declare only the the second commodity flow as integral, and substantially reduce the number of binary variables. In the final formulation, the number of binary variables is reduced even more because of the following two manipulations. If a program variable i has only one value range, then we can eliminate the variable z_i and assign to y_{i1} a cost of $2S_i$. If a program variable i is written at step j and read at step $j+1$, then i is not evicted at step j and reloaded immediately afterward. Therefore, the first value range of i can be eliminated. In turn, if i has zero or one remaining value ranges, the variable z_i can be discarded too. Since in practice many program variables fall in this case, the manipulations above results in a drastic reduction of binary variables. Finally, the number of constraints is reduced by replacing the portion of $\mathcal{N}_2$ relative to variable i with equations of the form $x^2(l_i) \geq x^2(l_{ik})$ for $k = 1, 2, \ldots, K(i)$.

D.3.2 Experimental Setup

We performed experiments on the same traces as in the previous section. We cast the resulting traces into the formulation described in the previous subsection. We used the CPLEX callable library to solve the resulting integer program. We turned off the CPLEX presolve aggregator, solved the relaxation with the network optimizer followed by the dual symplex algorithm, and fed the resulting optimum solution to the branch-and-bound solver for mixed integer programming. We selected to try always $x^2(l_i) = 0$ before $x^2(l_i) = 1$. The instance with the largest number of binary variables had 26595 constraints, 32039 variables, and 441 binary decision variables. In this case, an optimum integer solution was returned in 490 seconds on a SUN UltraSparc1 (143MHz/64Mb). Unlike the previous algorithm [HFG89], we never ran out of space. The optimum cost for loads and stores is better than CFF by 0% to 100%.

D.3.3 Experimental Results

The following tables report the result of the experiments. Column 3 gives the average time needed to solve the local register allocation problems to optimality in milliseconds. Column 4 reports the average number of nodes visited by the branch-and-bound algorithm. Column 5 gives the number of basic blocks where the relaxation yielded an optimum integer value. Column 6 gives the number of basic blocks whose relaxation is integral and that have a unique optimum solution, plus the number of basic blocks for which local register allocation reduces to a minimum cost network flow problem. Column 7 reports the average number of nodes visited by the branch-and-bound procedure for the basic block whose linear relaxation was not integral. Finally, column 8 gives the percentage of time spent in setting up and solving the relaxation as a fraction of the total time spent setting up and executing the CPLEX optimization routines. The times include user and system time accounted for those routines.

It is clear that the linear relaxation provides excellent results in practice. Indeed, in almost all cases the relaxation has a unique integral solution. Conversely, there is only one basic block where the number of nodes affects the percentage of time spent to solve the relaxation. Preliminary experiments suggest that the same results is true for different choices of the optimization passes and of objective function values.

Finally we note that the geometric average of the time spent solving the relaxation for $N = 16$ is 86.61% for floating point traces and 72.33% for integer traces.

D.3.4 Conclusions

We now discuss whether an optimum solution for the local register allocation problem is viable in practice.

An optimum local register allocation cannot be found on polynomial time unless P = NP. However, the allocation problem might be quickly found for real program if the problem is not too big. We took local register allocation problems from the SPEC benchmarks. We cast all those problems into an integer program obtained as the simplification of the 2-commodity flow problem in the section above. We found local allocation problems with up to 441 binary variables and 26595 constraints for an (unrolled) basic block. We conclude that local register allocation problem are not always small. On the other hand, our experiments provide a very surprising picture.

The branch-and-bound algorithm never took much more than nine minutes per basic block and, in contrast with the dynamic programming algorithm [HKMW66, HFG89], we never ran out of space. A crucial experimental result is that the relaxation provides excellent results[4]. In almost all experiments, the relaxation has a unique integer solution. In practice, the time spent solving the relaxation accounts for most of the time used by a branch-and-bound algorithm. For example, in the case $N = 16$, the relaxation takes on average 87% of the time for floating point traces and 72% for integer traces. The remaining 13% or 28% respectively of the execution time is spent initializing parameters and visiting the nodes of the branch-and-bound tree.

We conclude that the branch-and-bound procedure significantly outperforms in practice the previous dynamic programming algorithm, and might be a practical algorithm.

[4] At this time, it is not clear how good the relaxation is for fixed-cost problems. For example, we heard practitioners reporting that the relaxation provides very good results for fixed-cost integer programs arising in facility location problems, but some authors hold a quite different view [NW88].

benchmark	N	time	avg ndc	int rel	unique	nonint ndc	rel time
doduc	2	27.317	3.0303E-3	1317	1317	1.33333	94.5522%
	4	17.123	3.47985E-2	1069	1069	1.65217	89.9163%
	8	15.495	3.47395E-2	793	793	2.15385	86.7713%
	16	22.334	1.52439E-2	324	324	1.25	85.0403%
	32	77.79	3.33333E-2	88	88	1.5	90.5253%
	64	90.989	0.525	39	39	21	86.0075%
	128	217.61	0	4	4	-	91.1102%
fpppp	2	414.929	0	434	434	-	99.2359%
	4	313.902	0.0477327	409	409	2	98.9489%
	8	207.988	0.0298913	364	364	2.75	98.3165%
	16	247.167	0	284	284	-	98.2981%
	32	350.129	0	149	149	-	97.9404%
	64	506.212	0	85	85	-	97.7903%
	128	172119	6.5	1	1	13	38.4111%
	256	126942	47	0	0	47	83.094%
matrix300	2	1.25	0	1	1	-	72.64%
tomcatv	2	3.1675	0	32	32	-	67.0366%
	4	5.7894	0	11	11	-	67.2817%
	8	12.881	0	4	4	-	79.5439%
	16	38.422	0	1	1	-	77.1199%
	32	40.419	0	1	1	-	79.8857%

Table 4: Floating point traces: branch-and-bound

benchmark	N	time	avg ndc	int rel	unique	nonint ndc	rel time
doduc	2	6.1672	1.33988E-3	2236	2236	1	68.1251%
	4	6.7298	1.88399E-2	1994	1994	1.65217	69.4564%
	8	9.4023	1.94093E-2	1177	1177	2.875	71.7235%
	16	12.137	2.35457E-2	717	717	3.4	73.933%
	32	14.292	0	476	476	-	74.9149%
	64	19.166	0	229	229	-	77.2957%
	128	29.066	0	73	73	-	79.5225%
	256	56.189	0	7	7	-	79.2711%
fpppp	2	21.969	2.05761E-3	485	485	1	81.1689%
	4	24.249	2.1645E-2	456	456	1.66667	82.6363%
	8	28.759	5.02513E-3	397	397	2	83.7211%
	16	31.414	0	360	360	-	84.4011%
	32	30.333	0	349	349	-	83.9741%
	64	28.869	0	324	324	-	83.2283%
	128	33.281	0	224	224	-	84.1091%
	256	47.619	0	102	102	-	88.8555%
matrix300	2	4.6308	0	74	74	-	58.3395%
	4	4.7624	4.05405E-2	71	71	1	59.501%
	8	4.9152	8.21918E-2	71	71	3	60.1404%
	16	5.0804	0	42	42	-	59.2238%
tomcatv	2	7.3539	0	120	120	-	71.8792%
	4	7.2041	1.69492E-2	116	116	1	70.7627%
	8	14.518	0.144231	97	97	2.14286	79.0853%
	16	24.579	0.222222	53	53	12	74.077%
	32	48.144	1	25	25	26	67.3206%
	64	143.27	0	4	4	-	91.8277%
	128	159.6	0	3	3	-	94.8556%

Table 5: Integer traces: branch-and-bound