On Minimizing the Number of Page Faults with Complete Information (preliminary version)

Teofilo F. Gonzalez
Programs in Mathematical Sciences
The University of Texas at Dallas
Richardson Tx 75080
USA

ABSTRACT

Let P represent a computer program specified by a set of m chains, each chain consists of an ordered list of page references. All the chains and the page references are known in advance. Program P is to be executed by a system with k page frames. Our problem is to find an order in which these chains are to be executed and to find a page replacement strategy which guarantees that the number of page faults is minimized. When m is 1, it is well known that our problem can be solved efficiently. When m is arbitrary (input), we show that minimizing the number of page faults is NP_hard.

I. INTRODUCTION

Let P denote a computer program. P is specified by a set of m chains $\{p_1,p_2,\ldots,p_m\}$. Chain p_i consists of ℓ_i instructions denoted by $x_{i,1},x_{i,2},\ldots,x_{i,\ell_i}$. Instruction $x_{i,j}$ cannot be executed until $x_{i,j-1}$ has been completed ($1 < j \leq \ell_i$). Program P is to be executed in a computer system with one central processing unit (CPU). Let $Q = y_1,y_2,\ldots,y_z$ where $z = \Sigma\ell_i$, be a chain. Chain p_i is said to be <u>included</u> in Q if for all $1 \leq j \leq \ell_i$, $y_{k_j} = x_{i,j}$ and $k_1 < k_2 < \ldots < k_{\ell_i}$. Chain Q is called a <u>superchain</u> for $P = \{p_1,p_2,\ldots,p_m\}$ if all p_i's are included in Q and if we fix the inclusion of each p_i then each element in Q is covered by exactly one chain. Superchain Q specifies the order in which the different instructions in chains $p_1,p_2,\ldots,p_m$ are to be executed by the CPU. The maximum number of different superchains for program P is roughly $m ** \Sigma\ell_i$.

The main memory of our computer system is partitioned into equal-sized blocks of adjacent memory locations called <u>page frames</u>. Also, the program space (address space) is partitioned into fix sized blocks called <u>pages</u>. A page can be assigned to any of the page frames. Initially all the pages are in auxiliary storage. Whenever a page is referenced and it is not in main storage, we say that there is a <u>page fault</u>. In this case the page has to be brought from auxiliary storage to main storage. Let k be the number of page frames and let n be the number of pages in P. If $k \geq n$, then page i is stored in page frame i and there will be exactly k page faults. On the other hand, if $k < n$ then each page might have to be brought from auxiliary storage to main storage more than once while executing P. If there are k pages in main storage and there is page fault, then one of the k pages in main storage will be replaced by the page now being referenced.

Let us assume that the execution of instruction $x_{i,j}$ requires the use of exactly one page ($r_{i,j}$). Now we say that p_i consists of ℓ_i page references denoted by $r_{i,1}, r_{i,2}, \cdots, r_{i,\ell_i}$. A superchain Q is defined the same way it was defined for instruction chains. Our problem, which we shall denote PF, is defined as follows:

<u>PF</u>: Given k, m, ℓ_i ($1 \leq i \leq m$) and $r_{i,j}$ ($1 \leq i \leq m$ and $1 \leq j \leq \ell_i$), find a way to execute program P and find a page replacement strategy which produces the least number of page faults.

When $m = 1$, problem PF reduces to the problem of finding an optimal page replacement strategy. A simple strategy was presented in [B] and [ADU] to solve this problem. The replacement scheme is as follows: "the page to be replaced is the one with the largest forward distance, where the forward distance of page w is the number of pages that need be referenced before page w is referenced again". This simple strategy can be computed in polynomial time, i.e., there is an efficient algorithm to solve the PF problem when m is 1.

When $m > 1$, we also need to specify the order in which each of the chain's elements will be executed. Our problem, PF, can be restated as follows:

PF: Given k, m, ℓ_i ($1 \leq i \leq m$) and $r_{i,j}$ ($1 \leq i \leq m$ and $1 \leq j \leq \ell_i$), find a superchain such that if we apply the optimal page replacement algorithm to it, the least possible number of page faults for P will be generated.

Since PF, for $m = 1$, can be easily solved, it is not at all clear why we are making the problem harder. The reason for this is that the more chains we use to represent program P, the more likely we are to produce a smaller number of page faults when executing P. One of the questions that arises at this time is: Who finds the superchain Q? ,i.e.,Who specifies the order in which the chains or parts of chains are to be executed?. This task could be performed by the programmer or by the system. Since k is usually known only at execution time, then it seems appropriate to obtain the superchain Q at execution time, unless there exists a superchain with the property that for all k it can be used to solve optimally our problem. Unfortunately this is not the case in general. The following example proves otherwise:

<u>EXAMPLE 1:</u>

p_1 = a,b,c,a,b,c,1,2,3

p_2 = 1,2,3,a,b,c,a,b,c

For the case of $k = 1$, one can show that the superchain 1,2,3,a,a,b,b,c,c,1,2,3 is the only chain producing the least number of page faults which is 12. This superchains generates 9 page faults when k is 3.

However the superchain

a,b,c,a,b,c,1,1,2,2,3,3,a,b,c,a,b,c

produces only 7 page faults when k is 3. It can be easily shown that all superchains producing 7 page faults (which is the minimum) when k is 3, will generate more than 12 page faults when k is one.

end of example 1

It should be clear the advantage of specifying a program as a set of chains rather than by using only a single chain. Clearly, the more chains we use, the smaller number of page faults would result for some values of k.

Since it is required to know all the page reference strings in advance, problem PF is not likely to arise in practice. In practice one could find systems in which programs are specified in the format used to describe PF but without the knowledge of the reference strings. However, it is still worthwhile studying problem PF because of the following potential applications in the type of systems just described. If there is some simple strategy to solve PF, as in the case of $m = 1$, it might be possible to estimate the information used by the algorithm at each step and use it in the case when the reference strings are not known in advance. Another application of our results is in the performance evaluation of systems. Once we fix some policy in a system which executes programs specified in the format of PF but without knowing the reference strings, we could execute these programs and then after their termination use our optimal strategy to check how far from optimal is our replacement policy. We could also use our algorithms while executing programs under some policies different than the optimal , to detect when the policy deviates far from the optimal policy.

In the following section we study the complexity of the PF problem. It will be shown that this problem is NP_hard even when k is 1. The case when k and m are equal to one can be solved in polynomial time. We conjecture that even when m is some fixed constant, PF remains NP_hard. The should not stop research in this area, but motivate the search for efficient approximation algorithms to solve the PF problem.

II The Complexity of the PF problem

In this section we study the complexity of the PF problem. First of all it is shown that the PF problem is NP_hard when k is 1. We prove this result by reducing the Shortest Common Superstring (SCS) problem to it. Then we show that the PF problem is NP_hard for all $k > 1$. This will be shown by reducing the PF ($k = 1$) problem to it. Before proving our result we define the SCS (Shortest Common Supersequence) problem, which was shown to be NP_hard in [M].

DEFINITION:
SCS: Given a set $R = \{S_1, S_2, \ldots, S_m\}$ of sequences, find a supersequence S' for R with the least number of elements. If $S = s_1, s_2, \ldots, s_n$ then S" is said to be a subsequence of S when S" is S after deleting x symbols ($0 \leq x \leq n$) from S. S is said to be a supersequence of S". Now, S' is a supersequence of R if S' is a supersequence of S_i for $1 \leq i \leq m$.

THEOREM 1: PF for $k = 1$ is NP_hard.
PROOF: The proof is omitted.

We now show that the PF problem is NP_hard for all values of k. We prove our result by reducing the PF ($k = 1$) problem to it.

THEOREM 2: PF is NP_hard.
PROOF: Given an instance $P = \{p_1, p_2, \ldots, p_m\}$ of the PF ($k = 1$) problem, we construct an instance, (P', ℓ), for the PF problem with $k = \ell$. Assume without loss of generality that the set of distinct pages in P is $\{a_1, a_2, \ldots, a_n\}$. Let $b_{i,j,1}$, $b_{i,j,2}$, ... , $b_{i,j,\ell-1}$ for $1 \leq i \leq m$ and $1 \leq j \leq \ell_i$ be other pages not included in $\{a_1, a_2, \ldots, a_n\}$. P' is defined as follows:
For $i = 1, 2, \ldots, m$
let $p'_i = B_{i,1}, r_{i,1}, B_{i,1}, r_{i,1}, B_{i,2}, r_{i,2}, B_{i,2}, r_{i,2}, \ldots, B_{i,\ell_i}, r_{i,\ell_i}, B_{i,\ell_i}, r_{i,\ell_i}$

where $B_{i,j}$ is $b_{i,j,1}, b_{i,j,2}, \ldots, b_{i,j,\ell-1}$.
Since the construction of P' can be carried out in polynomial time it is only required to show that (P', ℓ) has an optimal solution with $h + (\ell - 1)*(\Sigma \ell_i)$ if and only if P has an optimal solution which generates h page faults.

Since the proof of this part is straight forward, it will be omitted. We will just present an example to illustrate the reduction.

EXAMPLE 2:
Let $P = \{p_1, p_2, p_3\}$, where

$p_1 = 1,4,3,2$
$p_2 = 2,4,3,2$
$p_3 = 1,2,4,3,2.$

(P', ℓ) constructed by our rules is as follows:

$p'_1 = B_{1,1}, 1, B_{1,1}, 1, B_{1,2}, 4, B_{1,2}, 4, B_{1,3}, 3, B_{1,3}, 3, B_{1,4}, 2, B_{1,4}, 2$
$p'_2 = B_{2,1}, 2, B_{2,1}, 2, B_{2,2}, 4, B_{2,2}, 4, B_{2,3}, 3, B_{2,3}, 3, B_{2,4}, 2, B_{2,4}, 2$
$p'_3 = B_{3,1}, 1, B_{3,1}, 1, B_{3,2}, 2, B_{3,2}, 2, B_{3,3}, 4, B_{3,3}, 4, B_{3,4}, 3, B_{3,4}, 3, B_{3,5}, 2, B_{3,5}, 2$

where $B_{i,j} = b_{i,j,1}, b_{i,j,2}$.
It is simple to show that h is 5 in this case.

III. DISCUSSION

We should point out that when m is 2 the SCS problem can be solved in polynomial time [M]. The results in [M] indicates that the SCS problem is NP_hard when m is a variable (input). The complexity of the SCS problem for the case when m is some fixed constant greater than 2 is not known. The same holds true for the PF problem when k is 1. We conjecture that the PF problem is NP_hard even when m is some fixed constant.

It is worthwhile studying the complexity of finding approximate solutions to the PF problem. For the case when k is one, it is simple to find an approximation algorithm which guarantees solutions whose number of page faults is within $\lceil m/2 \rceil$ of the optimal number of page faults. For other values of k one can easily obtain efficient algorithms which guarantee solutions within a bound of m from the true optimal solution value.

IV. REFERENCES

[ADU] Aho,A., Denning, P. and Ullman, J., "Principles of Optimal Page Replacement", Journal of the ACM, Vol. 18, Jan 1971, pp 80-93.

[B] Belady, L., "A Study of Replacement Algorithms for a virtual-storage computer", IBM SYSTEMS JOURNAL, Vol 5, No. 2, 1966, pp 78-89.

[CD] Coffman, E and Denning, P. "Operating Systems Theory", Prentice Hall , 1973.

[M] Maier, D. "The Complexity of Some Problems on subsequences and supersequences", Journal of the ACM, Vol. 25, No. 2, 1978, pp322-336.

EVALUATING ARITHMETIC EXPRESSIONS
(preliminary
Teofilo Gonzalez* a
Department of Co
The Pennsylvania
University Par

1. INTRODUCTION

Several interesting results concerning the problem of code generation for arithmetic expressions have been established by several authors. Extending the work of Anderson [A], Nakata [N], and Redziejowski [R], Sethi and Ullman [SU] have presented an efficient algorithm to generate minimal length codes for a special type of arithmetic expressions, namely those expressions with no common subexpressions. Aho and Johnson [AJ] have found a more general algorithm which allows general addressing features such as indirect addressing, but again restricting themselves to the same type of expressions. The case of arbitrary expressions has been proven to be difficult in a precise sense, i.e., it is NP-complete ([K]), even for the class of one-register machines with no algebraic identities allowed (Bruno and Sethi [BSe]). Aho et. al. [AJU] have shown that the problem remains NP-complete for dags whose shared nodes are leaves or nodes at level one and have developed heuristic algorithms to generate good codes.

The effect of algebraic laws on code generation has received little attention in the literature. Sethi and Ullman [SU] have discussed the case where some of the operators of an expression tree are associative and commutative, and Breuer [B] used the distributive law to factor polynomials in a manner similar to that of Horner's algorithm. When certain algebraic transformations apply for an arithmetic expression A, we are not required to generate codes for A, but we may generate codes for any equivalent expression A' obtained by successive applications of the algebraic laws. Since the number of arithmetic operations may then vary, the optimality criterion of generated codes should depend on the number of arithmetic operations as well as on the code length. In this paper, we assume that the distributive law holds and consider the problem of minimizing the number of arithmetic operations for a single arithmetic expression which involve only addition and multiplication. We also assume that addition is commutative and associative and that multiplication is associative. In section 2, we show that minimizing the number of multiplication nodes is NP-hard even when the given dag is a leaf dag and the expression is of

* Supported in part by the National Science Foundation under Grant MCS 77-21092.

** Supported in part by the National Science Foundation under Grant MCS 78-06118.