ABSTRACT

The design and implementation of compilers for high performance compiler systems require a thorough understanding of the target architecture to deliver the highest level of performance. It is important to know the I/O behavior on memory hierarchies. Compile time data transfer analysis computes an accurate approximation on the number of data transfers between main and secondary memory when the Least Recently Used replacement policy is applied. This analysis is performed at compile time; since some details are unknown, simplifying assumptions have to be accepted from the outset. The analyzer estimates reasonably precisely the number of page transfers that a program needs for execution, based on the parameters of the system available at compile time. The results reflect the real behavior of the system when executing the program and can therefore be used to compare situations derived for the same program after code restructuring.

Key Words: Compiler, I/O profile, page transfer, out-of-core

1. Introduction

The speed gap between processors and disks continues to increase as VLSI technology advances at a tremendous rate while disk technology is relatively stagnant. As a result, disk I/O has become a serious bottleneck for many high performance computer systems. Hence, it is critically important to be able to construct I/O minimal programs [9].

A number of run-time libraries for out-of-core computations and a few file interfaces have been proposed, among them SIO [10], MPI-IO [2], and an extension of the traditional Unix file I/O interface for handling the parallel accesses to parallel disk subsystems [5]. The parallel file systems and run-time libraries for out-of-core computations provide considerable I/O performance, but they require much effort from the user; also they are not portable across a wide variety of parallel machines with different disk subsystems.

The difficulty of handling out-of-core data and writing an efficient out-of-core program limits the performance of high performance computers. Execution of some out-of-core programs does not perform well when they rely on the virtual memory management (VMM) system. There is a clear need for compiler directed explicit I/O for out-of-core computations [1][3][4][12][13][14][15][16].

A global description of the data paths that occur during a program execution (I/O profile) can be very important in deciding whether or not the program is efficient from the I/O point of view [6]. An I/O profile describes how much I/O has been performed and where; additionally it will represent information related to I/O wait time. The I/O profile may pinpoint where “too many” data transfers have occurred; therefore it provides guidance in improving the I/O behavior. Besides, the I/O profile can predict the amount of I/O and the run time required for a program similar to the one on which the I/O profile is based. [7][8][11] design an algorithm that, at compile time, computes an approximation on the number of data transfers between main and secondary memory when the Least Recently Used replacement policy is applied. When a program does require too much I/O work, program restructuring to minimize the data transfers becomes necessary. This approximation can be used as guidance to transform I/O-intensive programs, to achieve better performance. More details about the algorithm will be given in Section 2.

In this paper, we concentrate on creating an analyzer to implement the algorithm. This analysis is performed at compile time; since some details are unknown, simplifying assumptions have to be accepted from the outset. Both at the start and during the execution of the analysis, several hypotheses are assumed. We will list system and program assumptions and required input in
Section 3. The analyzer estimates reasonably accurately the number of page transfers that a program needs for execution, based on the parameters of system currently available. The results reflect the real behavior of the system when executing the program and can therefore be used to compare situations derived for the same program after restructuring. We will discuss the implementation in Section 4. In Section 5, we summarize our work and goal.

2. Algorithm

The entire analysis is based on the PRS (Page Reference Sequence), which keeps track of the pages containing array elements referenced in the present iteration. A page reference sequence $PRS = (R_1[P], R_2[P], \ldots, R_n[P])$ is an extension of the ARS (Array Reference Sequence) associated with a loop body where $R_i[P]$ denotes the $i^{th}$ page which contains the array $R_i$. An array reference sequence $ARS = (R_1, R_2, \ldots, R_n)$, referring to a particular loop, represents the ordered list of the array references in the loop body. From each loop part, we obtain $ARS$ and for each array reference its index expression and the number of unique arrays in $ARS$. For example, assume that the arrays $A[n][n]$ and $B[n][n]$ are declared in the C language and stored in row major order in pages of size $p$ [34]. $p$ is fixed and $p < n$.

The code is the following:

```c
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        A[i][j] = B[2*i][j];
```

**Figure 2.1 Example 1**

For the first iteration: $ARS = (B, A)$ and $PRS = (B[1], A[1])$.

The $ARS$ of a loop remains a constant; its $PRS$ is continually updated on the basis of the values of the array indices. The $PRS$ can be defined as follows:

$PRS = (R_1[P], R_2[P], \ldots, R_n[P])$, where $P = [SI/P]+1$, $P$ is the page size.

$SI_i$ (Starting Index) represents the index of the first array element referenced in the $i^{th}$ array. The general formula to calculate $SI$ for a two-dimensional array $R$ in the C language is as follows:

$SI = I_1 \cdot D_2 + I_2$

where we assume that the following information is extracted from the program:

- the total number of dimensions of the array is 2;
- $D_1$ and $D_2$ are the dimension specifications of the array;
- the initial index value declared in dimension $j$ of the array is always 0 and $I_1$ is the value of the same index at the start of the loop.

For our example on Figure 2.1, we get $D_1 = n$, $D_2 = n$, and $P = p$. For the first iteration, $I_{11} = 0$, $I_{21} = 0$, $I_{12} = 0$, $I_{22} = 0$.

\[
SI_1 = I_{11} \cdot D_2 + I_{21} = 0 \cdot 10 + 0 = 0; \\
SI_2 = I_{12} \cdot D_2 + I_{22} = 0 \cdot 10 + 0 = 0; \\
P_1 = (SI_1 / P) + 1 = (0 / p) + 1 = 1 \\
P_2 = (SI_2 / P) + 1 = (0 / p) + 1 = 1
\]

We get $PRS = (B[1], A[1])$.

The goal of the algorithm is to calculate the amount of data transferred between the main memory and the disk. The data transfers in a loop cycle can be calculated by studying the evolution of the $PRS$ with respect to the modification of the pages it contains. This value can be decomposed into three terms corresponding to phases in the life of a $PRS$. 

1) Calculate $b$, the total number of data transfers due to the establishment of the $PRS$:

? if $U \neq WS$, $b = U$; where $U$ is the number of arrays in $ARS$ and $WS$ is the size of the working storage set (in terms of the pages it contains);

? otherwise call Algorithm A (see Section 4.1) [11].

In our example on Figure 2.1, assume that $WS = 3$. There are two arrays $A$ and $B$, so $U = 2$. Since $2 < 3$, $b = 2$.

2) Calculate $d$, the total number of transfers due to changes of the same pages referenced in the $PRS$. The number of transfers due to page moves is defined as:

$$ d = ?_i (N - F_i) / M_i $$

For all $R_i (i = 1, 2, \ldots, m)$ in the list $PM$ (Page Moves which can be obtained from Algorithm B, see Section 4.2): if $F_i < N$ and $F_i > 0$ then $dd_i = (N - F_i) / M_i$ else $dd_i = 0$;

?? $N$ is the total number of iterations of the innermost loop.

?? $F_i$ is the iteration in which the $i^{th}$ array in $PM$ changes for the first time.

$F_i$ can be calculated as: if $II_i = 0$ then $F_i = 0$,

else $F_i = (P \cdot SI_i) / II_i$;

where $P$ represents the page size;

$P \cdot SI_i$ gives the amount of data between the $i^{th}$ array element in position $SI_i$ of working storage and the end of the page in which it is stored. It is defined as:

$$ P \cdot SI_i = P - (SI_i \bmod P) $$

?? $II_i$ is the index increment which counts the number of elements between two references of the same loop of the same array (the $i^{th}$ array in $PM$) in successive iterations.

$II_i$ can be calculated as follows (under our assumptions):

If the $i^{th}$ array is one-dimensional then $II_i = ST \cdot C_1$

else $II_i = ST \cdot C_1 \cdot D_2 + ST \cdot C_2$;

where $ST$ represents the step increment of the innermost loop variable.

If the $i^{th}$ index contains the innermost loop variable ($j = 1, 2$), ($j = 1$ means the first index, or row index, and $j = 2$ means the second, or column index)

then $C_j$ = the coefficient of the innermost loop variable, else $C_j = 0$. 

?? M_i is the frequency at which the i-th array in PM changes page.
?? M_i can be obtained by: if \( \Pi_i = 0 \) then \( M_i = 0 \),
else \( M_i = \frac{P}{\Pi_i} \)

where \( P \) is the page size and \( \Pi_i \) is the index increment.

In our example of Figure 2.1, \( PM = \{B, A\} \) (Obtained from Algorithm B (see Section 4.2), such that array B is \( R_1 \) and array A is \( R_2 \) in the PM list; we have \( ST = 1 \), \( N = n \), and \( P = p \)).

?? For \( R_1 \) (B array):

Only the second index contains the innermost loop variable, such that \( C_1 = 0 \) and \( C_2 = 1 \):
\[
\Pi_1 = ST * C_1 * D_2 + ST * C_2 = 0 + 1 = 1;
M_1 = \frac{P}{\Pi_1} = \frac{p}{1} = p.
\]

It is the first iteration, such that SI_i = 0; \( F_1 = (P - (SI_i \text{ MOD } P)) / \Pi_1 = (P - (0 \text{ MOD } p)) / 1 = p \).
Since \( p < n \) and \( F_1 = p \) then \( dd_1 = (N - F_1) / M_1 = (n - p) / p = n \) / \( p - 1 \).

?? For \( R_2 \) (A array):

In the same way we get \( M_2 = p, F_2 = p, and dd_2 = n / p - 1 \)

?? \( d = \Pi_i dd_i \), for all \( i \)
\[
d = dd_1 + dd_2 = 2 * n / p - 2.
\]

3) Calculate \( c \), the number of data transfers to execute one loop iteration:

If \( U < WS \) then \( c = 0 \) otherwise call Algorithm A (see Section 4.1)

In our example of Figure 2.1:
\[
U = 2 \text{ and } WS = 3, \text{ such that } 2 < 3, \text{ then } c = 0.
\]

4) The total number of data transfers in the innermost loop is \( t = b - c + d + N * c \). For our example, \( t = 2 - 0 + 2 * n / p - 2 + n * 0 = 2 * n / p \).

The total number of data transfers for the whole program is therefore: \( n * 2 * n / p = 2 * n^2 / p \) as indicated in [9].

3. Assumptions and Required Input

Based on the algorithm in [7], we have designed an analysis tool working with simplified C programs. The tool is written in the C language and uses the parser utility function of Linux; it works under the RedHat 5.0 Linux operating system on a PC with a PentiumPro (160MHz or faster) processor and 64MB or more RAM. The system has a single process accessing its own memory without sharing those addresses with other processes.

We work on simplified versions of C programs which handle a large amount of data typically stored in matrices (two-dimensional arrays) in row major. This simplified C program usually contains a single function and there is one main loop control statement inside the function. We assume some simplifications, such as that only the for loops present in the target program are analyzed and all the other parts are simply skipped (while loops are assumed to be converted to for loops), and thus only the array references are counted when transferring data between main memory and disk. We will consider no other statements except other for loops contained in a loop body.

There are three groups of input required for the data transfer analysis. The first group of parameters is related to the given computer system; it consists of the working storage set (which is defined as the set of addresses the program is going to reference over a short period of time) in the memory, the page size, the replacement policy (currently fixed to Least Recently Used (LRU)), and the memory storage pattern (row major or column major; currently fixed to row major). The second group has parameters related to the data structures such as array dimension specifications and the total number of arrays declared. The third group has the information about the program to be analyzed. We assume that we can obtain this information about each loop such as start, end, and step value in loop expressions from a static analysis of the syntax of the program.

4. Implementation

In this section we present the implementation of our algorithm whose main ideas were described in Section 2. Several additional assumptions are made: the input program does not contain any syntax errors; the input program does not contain any function declarations or calls; and only the assignment statements inside a loop are analyzed. More specifically, the simplified version target program is structured with declarations for the arrays and the for loops.

Our implementation can be separated into successive phases: initially the target program is scanned and tokens and symbol tables are generated; then the information needed to perform our calculations is extracted and ARS (Array Reference Sequence) is generated. Based on these data and ARS, PRS (Page Reference Sequence) is generated and the data are analyzed in steps (see Steps 1, 2, 3, and 4 in Section 2). Our analysis refers to the data transferred during the execution of the for loops of the target program.

4.1. Transfers During the Execution of a Single Iteration

Algorithm_A is used to calculate the number of transfers occurring between the main memory and the disk during the execution of a single iteration of the innermost loop according to the LRU policy. In this algorithm, the memory is considered a queue ordered from the last to the first page transferred to the main memory.
We have modified and adjusted Algorithm_A to reflect the data transferred from memory to disk. In order to tell which array references are written and which array references are read only, we attach a flag to the array references in ARS. This can be derived by inspecting the program code (currently we assume the arrays whose values are changed will be transferred from memory to disk).

4.2. Page Reference Sequence Evolution

Since the array indices are continually updated and different portions of the arrays are addressed, several PRS changes take place during the loop lifetime. The PRS is updated with respect to the previous one during iteration. Additional transfers may be needed in order to bring the new pages into the memory.

Algorithm_B is used to recognize whether an array’s change of pages adds new transfers to the ones already determined. The purpose of this algorithm is to determine exactly whether an array element is still in memory from the previous cycle when it must be accessed. Its output, PM, is the list of the arrays used in the calculation of d.

5. Summary

This compile time data transfer analysis computes an accurate approximation on the number of pages transferred between main memory and disk. The result obtained represents the behavior of the system and offers a valid parameter to an I/O minimizing compiling environment to reconstruct an out-of-core program.

The goal of our research is to provide a fine-grained measurement of I/O complexity. An I/O complexity analysis can compute the number of block transfers on the basis of syntactic properties of the program and certain parameters of the computing system and the data set only. It does not require the execution of the program with some data set and can be carried out at compile time.

The ultimate purpose of this work is not only to create a tool for compile time data transfer analysis, but also to use this tool as part of the Comanche runtime system [16]. Although some approximations and restrictions apply, our final result reflects adequately the real behavior of the system when executing the program. We intend to use our tool to test the performance derived for the same program to guide the restructuring of the code for I/O minimization.

References