Program Analysis for Page Size Selection

K. Gopinath & Aniruddha P. Bhutkar *
Department of Computer Science & Automation
Indian Institute of Science, Bangalore

Abstract

To support high performance architectures with multiple page sizes, it is necessary to assign proper page sizes for array memory in order to improve TLB performance as well as reduce memory contention during program execution. Typically, while a smaller page size causes higher TLB contention, a larger page size causes higher memory contention and fragmentation but also has the effect of prefetching pages required in future thereby reducing the number of cold page faults. Each array in a program contributes to these costs/benefits depending upon how it is referenced in the program. The page size assignment analysis determines a proper page size for every array by analyzing memory reference patterns (which is shown to be NP-hard). We discuss various policies that can be followed for page size assignment in order to maximize performance along with cost models and present algorithms for page size selection.

1. Introduction

As memories get larger and cycles per instructions (CPI) of processors get smaller, the time spent in TLB miss handling can become a performance bottleneck. A decade ago, machines had relatively smaller memories and hence the TLB could map a substantial fraction of main memory. The programs at that time also had smaller working sets; hence, the TLBs missed less frequently. On architectures with inverted page tables or hash tables, such as PowerPC, the cost of a TLB miss is even much greater.

There are many ways in which one can improve the performance by reducing the time spent on TLB miss handling: it can be in hardware, compilers or operating systems. One solution is to make the TLB larger to hold more number of entries. However the extent to which TLB size can be increased depends upon whether we use a physically or virtually tagged cache but both have negative effects. A second solution is to use a larger page size, so that TLBs can map a larger fraction of the main memory without having to hold more number of entries; however, this increases fragmentation and the size of the working sets of the program. [TKH 92] show that increasing the page size from 4 KB to 32 KB causes a significant increase in the working set size (namely 60%) but a reduction in TLB's contribution to the CPI by a factor of eight.

Another solution, the subject of this paper, is the use of multiple page sizes. To effectively use multiple page sizes available in an architecture for improving overall system performance, a proper page-size assignment policy is needed. We need to solve two problems in an OS or compiler: how is the the address space of the program to be divided into different regions and what criterion to apply to assign a particular page size to each region. We consider only the problem of page size selection by static program analysis and do not consider the OS-based approaches.

*Author for correspondence: gopi@csa.iisc.ernet.in

2. Page Size Assignment is NP-Hard

Let n be the number of arrays in the program and let m page sizes be available on the machine. With each assignment of a page size to an array, we associate some cost (memory contention) and some profit (TLB savings and prepaging). These costs cannot be computed from memory access patterns alone as the memory contention caused by other arrays in turn depends upon the page sizes assigned to them. Hence, in general, it might be necessary to consider all combinations of arrays and page sizes rather than treating each array independently and compute the minimum cost. Given thus a collection of n sets with each element of the set containing m tuples with each tuple having pagesize (p), cost (c) and memory used (w) as components, the page size assignment problem (PSA) is then to select one tuple from each set such that total

0-8186-7557-8/96 $5.00 © 1996 IEEE

weight is less than W where W is the main memory size and the total cost is minimized.

The decision problem for PSA is as follows: Given n sets, each of m elements, is it possible to choose one element from each set such that total weight is less than W_p and total value is greater than a given number V_p? This problem can be proved to be NP-hard by reduction to the Knapsack decision problem (KNAP): Given n elements, is it possible to select a set of elements whose weight is less than W_k and value greater than V_k? From an instance K of decision problem KNAP, we construct an instance P of the decision problem PSA such that K has a solution if P has a solution. For simplicity, we take $m = 2$ (only 2 page sizes) in the following reduction.

K: Given a set $S = \{a_1, a_2, .., a_n\}$ and W_k and V_k, does there exist a subset S' such that $\sum_{a_l \in S'} W_{a_l} \leq W_k$ and $\sum_{a_l \in S'} V_{a_l} \geq V_k$?

Construct P with n sets $\{a_1, b_1\}, \{a_2, b_2\}, .., \{a_n, b_n\}$ where $V_{b_i} = 1$ and $W_{b_i} = 1$ for each b_i, $W_p = W_k$, and $V_p = V_k + n$.

If there exists a solution set $S'' = \{c_1, c_2, ..., c_r\}$ for P (with $c_i = a_i$ or $c_i = b_i$), satisfying $\sum_{c_l \in S''} W_{a_l} \leq W_p$ and $\sum_{c_l \in S''} V_{c_l} \geq V_p$, remove those c_l's that are not a_l's to give a new set S'''.

As $\sum_{c_l \in S'''} W_{a_l} \leq W_p$, it follows that $\sum_{c_l \in S'''} W_{a_l} \leq W_k$, and as $\sum_{c_l \in S'''} V_{c_l} \geq V_p - n$, it also follows that $\sum_{c_l \in S'''} V_{c_l} \geq V_k$.

Hence S''' satisfies K. □

3. Page Size Assignment Policies

3.1. An Operating System Approach

Page clustering considers faulting of clusters of small pages combined with TLB miss tracking by the OS to determine benefit of allocating a larger page size. Talluri et al. [TKH 92] divide the address space into various regions and then assign proper page sizes to each region. During the execution of the program, the OS maintains a list of all blocks accessed in the last T references (around 10 million). The decision to promote a chunk to a large page depends upon the number of blocks accessed within the window of the last T references. If all the blocks in a chunk have been accessed, then the chunk should certainly be mapped as a large page. If only one block has been accessed, then the chunk remains mapped as a small page. The threshold used to promote the page size is whether half or more blocks in a chunk have been accessed, so that, at the worst, the working set size is only doubled (and of course the TLB performance is similar or better than if only small pages are used). This policy uses the inherent spatial locality in the programs without the compiler or OS taking care to align the data structures on the page boundaries. This policy increases the working set size from 1% to 20% but TLB performance improves by a factor of 3 to 8 [TKH 92] with 2 pages of 4 and 32 KB.

However, there are certain costs to the approach. Page promotion can be costly, especially as it is invariably associated with copying the smaller pages into a larger page. Another point to note is that many architectures do not support small increments in page sizes (say 4-32K). For example, the increment can be as large as 256K (HP-PA) to 4MB (SuperSparc) from one page size to the next. In this situation especially, the copy costs can make the promotion strategy impractical.

In addition, many programs can be divided into various segments with each segment having a characteristic behaviour; it is more logical to observe the behaviour in each segment and then determine which page size will be more suitable for a particular array considering all program segments than the behaviour in last T references during the program execution.

3.2. A Compiler Approach

A compiler can analyze the *loop structure* of program (given by the abstract syntax tree—AST) and the array references with their subscript expressions) to find the *memory reference pattern* of the arrays. Then it can determine the optimum page sizes for the arrays that will improve TLB performance without much increase in the working set size by minimizing the overall cost due to TLB contention and memory contention. System calls have to be added in the code to inform the OS about aligning the pages and assigning page sizes to the arrays. Let us consider various criteria that can be used for page-size assignment, starting from the simplest ones.

Stride: The **stride** of an array reference inside a loop is the distance between the two elements of the array accessed in successive iterations. Assume that there are only two page sizes. Then for an array reference inside a nested loop, if the stride is greater than the small page size and less than the larger page size (and the memory utilization is also acceptable), then the array should be given the large page size since every iteration of the loop will need a new TLB entry thereby causing very high TLB contention. If the total size of the array is less than the small page size, or if the total address space of the array needed to be accessed within the entire execution of the loop is less than the small page size, then the array should not be considered for assigning large page size. Further analysis is needed if

these simple conditions do not hold.

Array Reference Patterns: If an array is referred at two or more different places in the program and these reference patterns are very different, different page sizes may be necessary. Unlike the case where the OS decides the page sizes, a compiler can determine how many times page size promotion and demotion are needed. However, promoting the page size of an array has certain costs associated with it as given below [TKH 92]: these costs are somewhat complex to determine as even the TLB miss penalty varies depending on whether 1 or more page sizes exist[1].

1: Cost of updating the mapping data structures and invalidating TLB entries for the small pages. This is about 20-50 cycles for each entry for TLB misses handled in software.

2: Copying the small pages already in memory to one large page. This can be from 3-20 cycles per word depending on the architecture, caching, memory latencies and the care with which the routines are written (especially, software pipelining). This is by far the largest cost. Hence dynamically changing page sizes has to be carefully evaluated: this is one reason why compile-time analyses may be better in many situations. Page coloring (see Section 4.3) or similar techniques may mitigate the expense of this cost and may be necessary for it to be useful. For the same reason, page size promotion handled by the compiler (providing different page sizes at different points in the program depending on the reference patterns) also has to be carefully evaluated.

3: Paging in/zeroing the small pages not resident in the memory. This cost can be considerable as it involves, in addition, a disk access; this access, however, is typically masked in multiprogramming environments. The cost of a full context switch with disk access masked is around 20000 cycles for SPARC. In batch environments, the full cost of paging has to be absorbed unless it is masked by multithreading.

Because of the expense of the copy costs, a static analysis that determines the page size at compile time is preferable.

Programming Environment: In a batch environment (large scientific applications), then the TLB state and the memory state will be governed by the program alone. We need information about the different references that conflict (and hence affect page size decisions of one another), so that the conditions imposed by them on one another can be formulated in the form of constraints to minimize the cost. A different analysis is needed for the multiprogramming case (section 4.2.1.)

Structure of Program: The memory and TLB contention costs associated with an array reference depend also on the loop nesting level of the array reference in the program.

In this work, we take all of the above concerns into account but placing the highest importance to program structure. We introduce tile trees for analysing program structure in the next section and formulate the PSA problem in the context of tile trees.

[1]For Sparc, with 2 page sizes, the miss handling is 25% longer in code [TKH 92].

4. Tile Trees and PSA

The concept of tile tree was first introduced by Callahan [CK 91] for exposing a program's loop and conditional structure and for better placement of spill code in register allocation. Although the intuitive concept of tile tree used here is similar, it has been modified substantially for the page size assignment problem. In the register allocation problem, the tile tree is constructed using the control flow graph of the program with the elements of a tile being basic blocks. Here, the tile consists of various array references which are in the same loop nest. The tile tree is constructed from the AST of a program.

A tile tree groups references together and eases the compiler's task of formulating constraints for the conflicting references. It can also be used for page size promotion and demotion so that page size assignment is sensitive to the program structure. The page size promotion and demotion code can be places in infrequently executed portions of the program and at the same time satisfying the page size requirements of the various regions.

4.1. Preliminaries

Let R be the set of all array references in the program. For simplicity, we consider only structured loops. Let T be a collection of sets t_1, t_2, ...t_n of references which covers the set R. We call T a tile tree and each element t_i of T a tile if the following conditions hold:

1: Each pair of sets in T is either disjoint or one is proper subset of the other. If there exist two tiles t_i and t_j such that $t_i \subset t_j$, then t_j is an ancestor of t_i. If there is no tile t such that $t_i \subset t$ and $t \subset t_j$, then t_i is a child of t_j, and t_j is the parent of t_i. Define the $ref(t)$ to be the set of those references which belong to tile t but do not belong to any child tile of t.

2: There is a special tile (the root tile) which contains all the references in R.

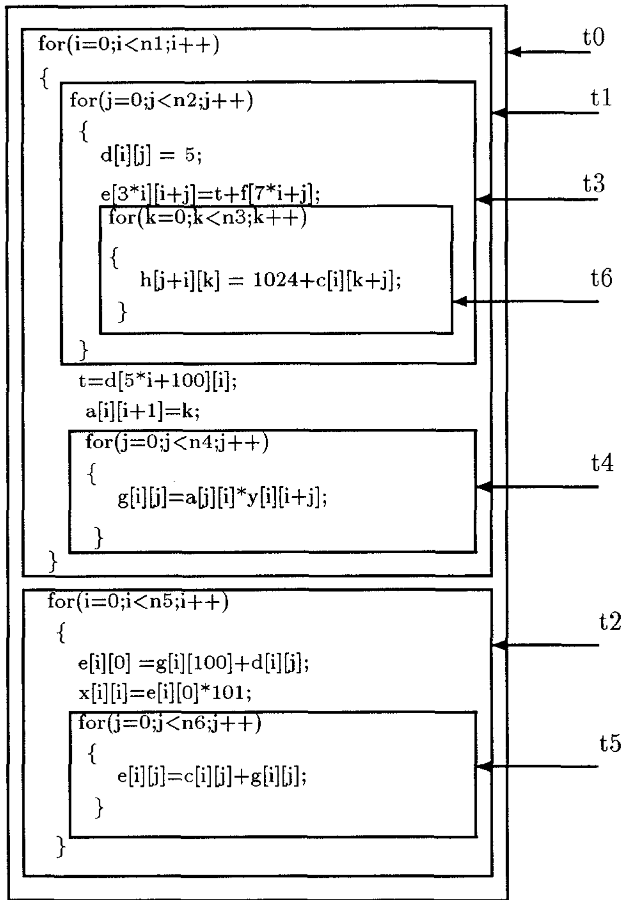

Figure 1. Division of a Program into Tiles

3: Tile t_i is a child of tile t_j iff there exist two iterative nodes[2] N_1 and N_2 in the AST such that N_2 is nested inside N_1, and $ref(t_j)$ contains all but no more than the references in the statements that belong to the body of N_1 but not to the body of N_2 and $ref(t_i)$ contains all but no more than the references in the statements that belong to the body of N_2.

In our definition, there is only one tile tree possible for a given program but it is not so for the approach described in [CK 91]. If a loop is tightly nested within another, $ref(t)$ of the tile corresponding to the outer of the two loops is empty. The depth of a tile in a tile tree gives the nesting level of the references in that tile. Consider the program in Figure 1. Root tile $t0$ has $t1$ (with arrays a,d) and $t2$ (g,e,d,x) subtiles. $t1$ has $t3$ (d,e,f) and $t4$(a,y,g). $t2$ has $t5$ (c,e,g). $t3$ has $t6$(c,h).

Every tile in the tile tree corresponds to a loop and, in an implementation, contains the nesting level of all the references in the tile, the bounds of the iterative variables for the current loop and the encompassing loops, a pointer to the child tile, a pointer to the sibling tile with the leftmost child followed by the right siblings and the linked list of references. Every reference in the tile tree contains a pointer to the symbol table entry for the array referred, memory reference expressions for the reference with the affine subscript expressions linearized, and a pointer to the next reference in the tile. We currently do not handle array references that are non-affine or with array bounds that are not known at compile time.

[2]in our AST implementation: DO_NODE, DO_WHILE or REPEAT_WHILE

4.2. Optimization Problem Formulation

To compute the cost of a page size being assigned to an array, we have to compute costs and profits for all references in the program. At different points in the program, the costs may be different and we have to give weightage to each reference depending upon its frequency of execution. We first consider the case without page promotion. We later relax this condition for the batch case.

4.2.1 Multiprogramming Environment

The CPU time spent in handling the TLB misses can be directly computed from the number of TLB misses, since it takes a fixed number of machine cycles to handle a TLB miss. However, the CPU time wasted due to page faults in a multiprogramming environment depends also on the process scheduling policy, the scheduling quantum, the size of physical memory, the number of applications currently active, etc. It is likely that these factors also have a non-linear behaviour. If we assume that all the above are kept constant and that the process scheduling policy schedules the processes so as to keep resources like main memory fully utilized, then under these simplifying assumptions, every extra page required in an application will cause a corresponding page fault in another and every extra TLB entry allotted to an application causes a TLB miss in another. Hence the TLB contention cost is proportional to the number of TLB entries required and the memory contention cost is proportional to the amount of memory consumed. Let

t_{ij} = TLB entries required by a particular reference when a page size i is assigned to array j,

m_{ij} = Memory required by a particular reference when a page size i is assigned to array j,

x_{ij} = 1 if page size i is assigned to array j, else 0

t = total TLB requirement in a tile,

m = total memory requirement in a tile,

c_t = cost of TLB contention in a tile,
c_m = cost of memory contention in a tile,
c = overall contention cost for a tile.
w = weightage assigned to a particular tile.
Then,

$$t = \sum_{j=1}^{n} \sum_{i=1}^{l} t_{ij} x_{ij}$$

and

$$m = \sum_{j=1}^{n} \sum_{i=1}^{l} m_{ij} x_{ij}$$

Since the TLB contention cost is proportional to the number of TLB entries required and memory contention cost is proportional to the amount of memory required, we have
$c_t = k_1 * t$, and
$c_m = k_2 * m$, and
$c = w * (c_t + c_m)$
where k_1 and k_2 are constants. These constants have to be measured in a real environment but we can make the following estimates: k_1 can be taken to be cost of handling a TLB miss and k_2 is the cost of a full context switch per page size plus the cost of DMA/zeroing per word (assuming that the disk I/O is fully masked). We assume k_1 to be about 20 cycles and k_2 to be about 5 cycles per extra word.

The x_{ij}'s that satisfy $\sum_{i=1}^{n} x_{ij} = 1$ for all j and minimize Σc over all the tiles gives us the optimal page size assignment. As this is an integer programming formulation, it can take exponential time in the worst case but heuristics can be used to solve it in reasonable time.

4.2.2 Batch Environment

In this case, the cost of TLB (memory) contention for an array reference is not proportional to number of TLB entries (amount of memory space) needed for that reference, but also depends upon the contention due to other references in that tile. The page sizes of conflicting references also determine the cost of TLB (memory) contention. The TLB (memory) cost can be related to the number of TLB entries (amount of address space) by a function which can assumed to be piecewise linear, as an increase in the TLB entries does not increase TLB contention much until a limit is reached. Past this limit, however, there is a cost for every additional TLB entry needed. Similar is the case with memory contention. To solve this non-linear integer programming problem, we introduce integers δ_1, δ_2, δ_3 and δ_4 which hold values 0 or 1 depending upon whether the above mentioned limit is exceeded or not.

The total TLB and memory requirement for a tile can be given as:

$$t = \sum_{j=1}^{n} \sum_{i=1}^{l} t_{ij} x_{ij}$$

and

$$m = \sum_{j=1}^{n} \sum_{i=1}^{l} m_{ij} x_{ij}$$

Here, the TLB and memory contention costs are not proportional to the number of entries required, but are given by:
$c_t = \delta_1 * t_1 * c_{t1} + \delta_2 * (t_2 * c_{t2} + \alpha_t)$
$c_m = \delta_3 * m_1 * c_{m1} + \delta_4 * (m_2 * c_{m2} + \alpha_m)$
$c = c_t + c_m$
subject to the condition that
$\delta_1 t \leq t_{tot}$, the needed number of TLB entries
$\delta_2 t > t_t$
$\delta_3 m \leq m_{tot}$, the needed amount of memory
$\delta_4 m > m_t$
where,
$t_1 = t, t_2 = t - t_{tot}$
$m_1 = m, m_2 = m - m_{tot}$
$\alpha_t = t_{tot} * c_{t1}$, $\alpha_m = m_{tot} * c_{m1}$, and c_{t1}, c_{t2}, c_{m1}, c_{m2} are the cost coefficients of the TLB and memory contention in the two regions of the piecewise approximation.

Then the x_{ij}'s that satisfy $\sum_{i=1}^{n} x_{ij} = 1$ for all j, and which minimize Σc over all the tiles, gives the optimal page size assignment.

4.3. Page Size Promotion

In this case, we have the choice of assigning different page sizes to the same array in different regions of the program, if references at different regions need different page sizes. We order the tiles according to their frequency of execution and assume that the cost of promoting a page size for an array is proportional to the factor by which page size is increased (as those many pages will have to be copied into a single page). As discussed earlier, the cost of page promotion can be considerable due to the copy costs and hence this optimization has to be carefully evaluated. Page promotion may be advantageous if the page size increment is small or techniques like page coloring [3] are used that reduce copy costs as contiguous virtual pages also are often contiguous in physical space. We assume that one of these techniques is in use when page promotion is attempted.

[3] Page coloring attempts to maintain a constant offset between VPN and PPN in the interests of reducing variance in execution times[TDF 90].

A similar analysis holds for page size demotion but the cost of demoting a page is less as there is no copying involved and only page table and TLB entries have to be appropriately changed [4].

4.3.1 Conflicting Tiles

The tiles that have the same parent and will be executed one after another conflict with one another. When selecting page sizes for a tile, we have to consider the page sizes in the conflicting tiles also.

Consider a tile tree with 8 as root. 8 has 6,7 as subtiles with 6 having 1,2,3 and 7 having 4,5 as subtiles. Tiles 1 and 6 do not conflict because tile 1 is clearly nested inside tile 6 with higher weightage and hence the page size assignment for tile 6 is after that for 1, 2 and 3. Also tiles 3 and 4 do not conflict: if tiles 3 and 4 require different page sizes for the arrays, then the page promotion code will be placed in tile 8, which is less frequently executed. Hence while considering tile 4, there is no need to consider tile 3. Tiles 3 and 4 are weakly linked. When page sizes in tiles 1 and 2 are different, the promotion code will be placed in tile 6. Hence tiles 1 and 2 are also not weakly linked. In summary, tiles (1,2), (2,3), (1,3), (4,5) and (6,7) conflict whereas tiles (1,6) and (3,4) do not.

4.3.2 A Greedy Algorithm

In the first phase (see Figure 2), we visit the most frequently executed tile and determine its optimal page size assignment. Then we visit its conflicting tiles and find their page sizes. In the second step, the algorithm traverses internal nodes in the tile tree bottomup to locate those arrays which are *not* referenced in their children but referenced in the node itself and allocate page sizes depending upon the reference patterns as in step 1.

for each level in the tile tree **do**
while there is any unvisited tile **do**
begin
  Select the unvisited tile t_i with the highest execution freq
  Find optimal page size assignment AS_i for t_i
  Mark t_i as visited
  Make a queue of tiles that conflict with t_i and order it by execution frequency
  for each tile t_j from the queue **do**
  begin
    1.Find optimal page size assg AS_{opt} for t_j without considering AS_i. Add to it the cost of promoting AS_i to AS_{opt}. Also if there is any other tile t_k that conflicts with t_j and is marked 'visited', add the cost of promoting AS_{opt} to AS_k.
    2.Find the cost of assigning AS_i to t_j.
    3.Find the cost of assigning AS_k to t_j.
    Select the AS from above three with minimum cost and assign it to AS_j.
    Mark t_j as visited.
    Add to the queue all the tiles that conflict with t_j and are not visited and sort the queue by frequency of execution.
  end
end
end

Figure 2. Algorithm for page size selection using page size promotion and demotion

5. Conclusions and Further Work

In this paper, we have shown that PSA is an NP-hard problem and presented integer programming formulations. A greedy algorithm in the context of page promotion and demotion has been also presented.

Compiler directed preloading of the TLB can also be attempted by this analysis. For example, if an architecture supports only a single page size and some array is found to require a page size larger than the only page size available, then the the pages corresponding to the rest of the larger page can be preloaded the first time the first part is brought in. Similarly, for an architecture supporting multiple page sizes whose sizes are wide apart, a similar preloading may be attempted by assuming that other page sizes are available and proceeding with the analysis. This can be useful as RISCs are beginning to provide hardware support for cache-line prefetch and some OSs are starting to experiment with more sophisticated pagers that accept preload hints.

[4] This suggests an overall policy of providing a slightly larger page size than needed and an one-time demotion if appropriate.

References

[CK 91] Callahan D., Koblenz B., "Register Allocation via Hierarchical Graph Coloring", PLDI'91.

[TKH 92] Talluri M., *et al* "Tradeoffs in Supporting Two Page Sizes", ISCA'92.

[TDF 90] George Taylor et al,"The TLB slice," ISCA'90.