Improving Spatial Locality of Programs via Data Mining *

Karlton Sequeira, Mohammed Zaki, Boleslaw Szymanski, Christopher Carothers
Computer Science Department, Rensselaer Polytechnic Institute, Troy, NY, USA
{sequek,zaki,szymansk,chrisc}@cs.rpi.edu

ABSTRACT

In most computer systems, page fault rate is currently minimized by generic page replacement algorithms which try to model the temporal locality inherent in programs. In this paper, we propose two algorithms, one greedy and the other stochastic, designed for program specific code restructuring as a means of increasing spatial locality within a program. Both algorithms effectively decrease average working set size and hence the page fault rate. Our methods are more effective than traditional approaches due to use of domain information. We illustrate the efficacy of our algorithms on actual data mining algorithms.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications-Data Mining

Keywords

Program Locality, Code Restructuring, Page Clustering

1. INTRODUCTION

A computer has fast, relatively small (expensive) main memory (RAM), and slow, relatively large secondary memory (hard disk). A large program may thus not completely fit into RAM. To remedy this, a program is considered to be made up of fixed-size blocks of memory called pages. To execute an instruction the page containing that instruction must reside in RAM. If it does not, a page fault is said to occur. If RAM is currently completely occupied, a page may be selected from those resident in RAM for replacement and sent to disk, and the desired page will occupy its position in RAM. The page is then referenced and the instruction is executed. The fraction of times a page fault accompanies a page reference is called the page fault rate (*pfr*). Program execution time is governed by a number of factors, among which *pfr* plays a prominent role. While *pfr* can be reduced by a number of operating system and hardware techniques, specialized software-based techniques, which exploit locality specific to a program are yet to be widely used.

Although the cost of main memory has been continually decreasing, the resulting gains have been mitigated by the continual rise in program memory requirements (by as large a factor as 2 per year [14]). Also, due to fields like sensor networks [2], there remains a need for programs to run in small amounts of memory (for energy efficiency). A well-known rule of thumb is that a program spends approximately 90% of its execution time in 10% of the code [14]. This is a manifestation of the principle of temporal locality. Since this 10% of code is not necessarily contiguous in memory, a program spends that 90% of its execution time requesting considerably more than 10% of its total number of pages. One of the key program-specific ideas is to utilize the temporal locality to optimize program spatial locality, i.e., the portions of a program's code segment are reordered in the pages of the executable to increase spatial locality. In doing so, a fewer number of pages are requested more often, thereby decreasing the average working set size corresponding to the program and hence decreasing the *pfr*.

Program spatial locality optimization is the specific problem we seek to solve. It is related to the constrained clustering problem. Several challenges to cope with include: determining what data to collect, selecting the data model to represent it, dealing with an unknown distribution of input parameters and making the solution applicable to the widest possible range of programs (which means that heuristics used should not be program-specific).

1.1 Related Work

The early work done on program restructuring concentrated on sector-level traces. Hatfield and Gerald [8] achieved paging performance improvements in the range of 2 to 10 times by relocating sectors (subroutine and data modules) of compilers, editors and assemblers. Johnson [11], provided an array of clustering algorithms and inter-sector reference models. He also formally analyzed the impact of the structural ordering of a program's relocatable sectors on the program's *pfr*.

Procedural-level program restructuring targets reduction of cache conflict misses rather than page faults. The former are easier to avoid if the architecture uses a separate instruction cache. It is also known that compiler optimization-invoked code reordering sometimes unintentionally increases cache misses [7]. Pettis and Hansen [15] used greedy clustering algorithms similar to those described by Johnson, but with the additional goal of arranging procedures within code blocks relative to each other. Gloy and Smith [7] proposed

*This work was supported under NSF NGSP grant EIA-0103708. Zaki was also supported by NSF CAREER Award IIS-0092978 and DOE Career Award DE-FG02-02ER25538.

SIGKDD '03, August 24-27, 2003, Washington, DC, USA

using temporal information, in a manner similar to our approach, for improving paging and demonstrated its advantage over the Pettis Hansen procedure placement algorithm.

Recently, Batchu and Levy [3], showed that an average working set containing 20% of the program's functions is sufficient to ensure a *pfr* of less than 2%. Our approach uses function-level program restructuring as well.

To model the data, Hatfield and Gerald [8] (HG model) suggested using a transformation of the sector trace into a "nearness matrix" to represent inter-sector reference behavior. However, they only capture associations between adjacent sector references. To counter this deficiency, Johnson [11], proposed an LRU (least recently used) stack based inter-sector model (LRU_STACK_MODEL). Gloy and Smith [7], used a temporal relationship graph (TRG) to represent relationships between procedures. Each edge of TRG has a weight equal to the number of times that two successive references to any procedure are interleaved with at least one reference to the other.

Constrained clustering is a semi-supervised learning technique, in which data is partitioned into meaningful subgroups called clusters, based on some similarity measure and subject to some a priori known constraints. Considerable work has been done in this field [4, 16]. Wagstaff *et al.* [16] have shown that domain-based heuristics contribute significantly to improving the resulting clustering. We utilize this idea in our approach.

1.2 Contributions

Our work introduces a new domain to which data mining can be applied. There is currently very little effort to solve such system-related problems using data mining.

We experimentally show that paging performance varies with the code structure. We demonstrate the effect of using domain information while determining an assignment of program functions to pages. We develop two algorithms to the program restructuring problem, by formulating it first as a constrained clustering problem and then as a stochastic optimization problem. We compare the resulting assignments to select the best assignment of code to pages. We apply our techniques on data mining problems.

We have profiled data mining programs for three reasons. Firstly, there is a strong trend towards reducing disk i/o requirements of data mining systems. Secondly, the predominantly combinatorial nature of data mining makes these programs highly compute-intensive. Finally, data mining programs often have code segments spanning a number of pages. Such programs benefit considerably from lower *pfr*s.

2. FORMULATION

We now formally describe the problem. Let the program be made up of a set $F = \{f_1, f_2, ..., f_{|F|}\}$ of functions and let there exist a mapping $Size : F \rightarrow Z$ which gives the size of each function (note: Z denotes the set of positive integers). Consider the virtual address trace $VT = a^1, \ldots, a^i, \ldots a^T$ of virtual addresses invoked during a program execution. There are two mappings defined by the memory system for each program: $f : A \rightarrow F$ and $p : A \rightarrow P$, where A is the set of all virtual addresses of the program and P is the set of all program pages. These two mappings define a function trace $FT = f(VT)$ (so $FT = f(a^1), f(a^2), \ldots f(a^T)$) and a page trace $PT = p(VT)$ (which is a sequence of pages, each of size $PageSize$, referenced during execution).

We assume that functions f_i are constrained to not cross page boundaries, so there is a many-to-one mapping from functions to the pages of a program. Most functions are smaller than the default page size and large functions can be split up to ensure that they do not cross page boundaries, so this assumption does not restrict the generality of our approach. Let $\mathcal{A}$ denote the set of all mappings $Assign : F \rightarrow P$. Our goal is to find an assignment $Assign_o \in \mathcal{A}$, such that

- the sum of sizes of functions assigned to a single page does not exceed $PageSize$ (An assignment satisfying this condition is called a *legitimate assignment*), and
- the *pfr* invoked by $Assign_o$ is the smallest among all legitimate assignments of functions onto pages.

For each page $p_i \in P$, there is an inverse mapping $\pi_i = Assign^{-1}(p_i)$ of all functions assigned to it. Hence, each legitimate assignment induces a partitioning Π of the function set F, into $|\Pi|$ disjoint clusters. Each cluster (page) π_i is constrained in that the sum of sizes of all its functions does not exceed the page size. Thus, the problem is formulated as one of constrained clustering.

We illustrate our solutions using the following program as an example (showing function definitions):

```
1.   f2() {   f1()   } //means that f2 invokes f1
2.   f3() {}
3.   f4() {}
4.   f5() {   f1()   f2()   }
5.   f1() {}
6.   main (int i) {
7.     if (i ≥ 20) f3() else f4()
9.     while (i − −) f5()
10.  }
```

where $main, f_1, f_2, f_3, f_4, f_5$ occupy say 3KB, 1KB, 2KB, 2KB, 2KB, 2KB respectively and the page size is 4KB. The default assignment places the functions as per their order of definition in the program. So, page 1 contains f_2, f_3, page 2 contains f_4, f_5, and page 3 contains $f_1, main$. This legitimate assignment is written as a clustering of functions as follows: $\{(f_2, f_3), (f_4, f_5), (f_1, main)\}$.

Our method involves three stages,

1. Instrumentation: Using the symbol table of the program executable to be optimized, we determine the set of functions and the size of each function. Using Instrumentation Database IDB [13], we insert instrumentation code at the start of each function to produce an instrumented executable. This executable is run with a variety of input parameters to produce a set of function traces, which are partitioned into a training set and a test set of function traces. For the sample program above, the trace would be:
   $< main \wedge (f_3 \vee f_4) \wedge (f_5 \wedge f_1 \wedge f_2 \wedge f_1)^i >$.
2. Mining: We then use our algorithms **GreedyClustering** and **MCA** to mine the $Assign$ function assigning functions to pages of the code segment using the training set. We test the model on the test set.
3. Reordering: Based on the $Assign$ function mined, we remove instrumentation code and reorder the functions in the executable via an object code editor.

By dealing with the executable at the time of instrumentation, one can account for compiler optimizations like inlining, template instantiations, etc. This paper concentrates on the mining stage. As an example for the trace above, one intuitively better assignment is $\{(f_2, f_5), (f_4, f_3), (f_1, main)\}$. If the sample program above, is allocated only 2 pages, a

simple analysis shows that LRU replacement [14], incurs at least $(1 + 2i)$ page faults i.e. $pfr \approx 0.5$, for the default assignment and no more than 4 page faults for the better one. We aim to find good assignments automatically via mining.

3. ALGORITHM

We devised two algorithms. The first is a greedy clustering algorithm which uses domain-specific heuristics to give an ordering of the functions in memory. The second algorithm recasts the problem as one of stochastic optimization.

To convert temporal locality into spatial locality, it is necessary to model the temporal locality first. We use a weighted graph $G = (F, E)$, to represent the temporal associations between the different functions to be ordered. Each vertex of G corresponds to a function and the weight $w(f_i, f_j)$ on each edge (f_i, f_j) in $F \times F$, corresponds to the strength of the temporal association between the functions connected by the edge. The edge (f_i, f_j) is akin to an association rule $f_i \rightarrow f_j$ with confidence proportional to $w(f_i, f_j)$. The higher the weight, the stronger the temporal association, and the higher the probability of page faults if f_i and f_j are assigned to different pages.

The graph is created by streaming through the trace, creating a node for every new function in the trace and updating the weights on edges as described below.

3.1 Weight function

We now define the weight function $w : F \times F \rightarrow Z$ for the trace graph. Johnson [11], suggested the use of a sector replacement stack to simulate the overlapping page stack. Analogously, we use a function replacement stack to simulate the overlapping page stack. If K_p pages are allocated to the program code segment, then it is highly probable that at least K_p functions will fit into memory, as most functions are smaller than a page. Given a function trace FT of length T and a function stack of size $K_f = K_p$,

$$w(f_i, f_j) = \sum_{t=1}^{T-1} I_w(f_i, f_j, t)$$

Accordingly, I_w is defined as:

$$I_w(f_i, f_j, t) = \begin{cases} p(f_i, f_j, t) \text{ if } & (f^{t+1} = f_j) \wedge \\ & (\Delta_f^t(f_i) \leq L_f) \wedge \\ & (\Delta_f^t(f_j) > K_f) \\ 0 \qquad & otherwise \end{cases} \quad (1)$$

where $\Delta_f^t(f_i)$ gives the height of function f_i in the function replacement stack at time $t, p(f_i, f_j, t)$ is the penalty function determining the increment to the weight on edge (f_i, f_j) due to a function fault at time t and L_f is the number of functions at the top of the stack whose edges with f_j are incremented.

The higher the function f_i in the replacement stack, the more closely temporally associated, it is with the faulting function f_j. Eq. 1 makes it explicit that the replacement algorithm and parameters must be incorporated into the function w, unlike the HG-model [8]. For $p(f_i, f_j, t)$=1 and L_f=1, I_w denotes the number of times f_i calls f_j. We have tested three penalty functions applied to all function stack members, i.e. $L_f = K_f$, to approximate the temporal associations between the functions:

LRU_STACK_MODEL(LSM) [11]: Johnson suggested equal penalties for all stack members, i.e.,

$$p_{LSM}(f_i, f_j, t) = 1 \quad (2)$$

The disadvantage of using this method of weighting the graph is that all functions in the function stack are incremented uniformly for a function fault. Thus each function in the stack is equally responsible for assigning the function for which the fault occurred, to its page. This model can promote pages in which functions having larger inter-reference times are equally probably adjacent as those having smaller inter-reference times. Thus LSM converts temporal locality into spatial locality in a manner that we expect would perform poorly. In Fig. 1, we show the graph produced for sample program in Sec. 2 using the LSM weighting function, $K_f = 2$ and $i \geq 20$. $w(main, f_3) = 1$ due to the cold start. $w(f_5, f_2) = w(f_1, f_2) = i$ because f_2 is referenced i times and each reference results in a fault since f_5 and f_1 are on the stack. Similarly, $w(f_1, f_5) = w(f_2, f_5) = i - 1$ because f_2 is referenced $i - 1$ times after stack contents are (f_2, f_1).

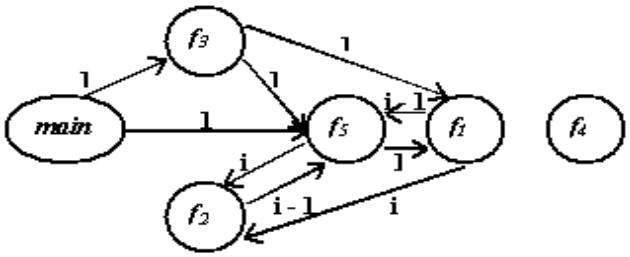

Figure 1: Graph for sample program and p_{LSM}

PENALTY_BASED_MODEL(PBM): To remedy the problem above, we penalize functions in the function stack for function faults, with increments proportional to their distance in the trace from the function for which the function fault occurred. Alternatively, from a stack-based viewpoint, we penalize them with increments inversely proportional to their probability of replacement from the function stack, which depends upon the replacement algorithm. For LRU,

$$p_{PBM}(f_i, f_j, t) = K_f - \Delta_f^t(f_i) \quad (3)$$

GDS_BASED_MODEL(GBM): GDS is a cache replacement algorithm used for replacing cached files at proxy servers [5]. To optimize cache hit rate, GDS uses time of last reference and the size of the file as input while replacing files. Analogously, to better model the associations between the functions, we incorporate the size of the function into the increments.

$$p_{GBM}(f_i, f_j, t) = \frac{K_f - \Delta_f^t(f_i)}{Size(f_j)} \quad (4)$$

Thus a function suffers a lesser increment if the function temporally closely associated with it is large in size, because proximity is normalized by function size.

3.2 GreedyClustering

This algorithm is similar to the greedy fractional knapsack algorithm [10]. In the fractional knapsack problem, we greedily choose items based on the expected gain in optimization per unit quantity of the item. Similarly, we choose functions to fill the pages based on the expected contribution towards minimization of page faults per unit byte occupied by the function in the page. We present the algorithm below:

GreedyClustering(G, P):
// G is the graph produced by function traces
// P is the set of pages to be filled
// P_l is the current page to be filled

//$FuncList$ is the list of functions to be assigned
1. $FuncList = V(G), Assign_{Current} = \phi$
2. while($FuncList$ not empty)
3. f_t=**pickFunction**$(G, Assign_{Current}, FuncList, P, P_l)$
4. update $Assign_{Current}, FuncList, P$
5. compact $Assign_{Current}$

Initially none of the functions are assigned to pages (line 1). Until all functions are assigned, we continue to select functions from the unassigned list using **pickFunction** and update the assignment. If there exist partially filled pages which may be packed together, we use a bin packing algorithm (line 5), since doing so implies no higher a *pfr*.

The **pickFunction** routine is shown below. It evaluates all candidates in $FuncList$ for a fit into the space available in the current page P_l. It picks candidates based on their C_{Score} (lines 1-3). If no function, which fits in the space on the current page, is temporally associated with the functions already on this page (line 4), the algorithm starts filling a new page (lines 5-8). Thus, it can assign f_i to P_l, even if f_i has a stronger temporal association with f_j, not assigned to P_l. Thus, this method differs from agglomerative hierarchical clustering, since it does not always assign the two most temporally associated groups of functions to the same page [15].

pickFunction$(G, FuncList, P, P_l)$:
// $Space : P \rightarrow Z$ gives the space available in a page
1. $\forall f_i \in FuncList$,
2. compute $C_{Score}(f_i, P_l)$
3. choose $f_t \in FuncList$ based on $C_{Score}(f_t, P_l)$
4. if $\nexists f_i \in FuncList$ such that
4. $Size(f_i) \leq Space(P_l)$ and $C_{Score}(f_i, P_l) > 0$
5. increment P_l
6. $\forall f_i \in FuncList$,
7. compute $C_{Score}(f_i, FuncList)$
8. choose $f_t \in FuncList$ based on $C_{Score}(f_t, FuncList)$
9. return f_t

The algorithm selects the function to place next in the current page based on the score computed by Eq. (6).

$$C_{Score}(f_i, P_l) \propto \sum_{f_j \in P_l} \frac{w(f_i, f_j) + w(f_j, f_i)}{Size(f_i) + \sum_{f_k \in P_l} Size(f_k)} \quad (5)$$

$$C_{Score}(f_i, FuncList) \propto \max_{f_j \in FuncList} \frac{w(f_i, f_j) + w(f_j, f_i)}{Size(f_i) + Size(f_j)} \quad (6)$$

Since, the number of functions assignable to P_l is bounded above by the page size which is constant, it is easy to see that **pickFunction** has complexity $O(|E|)$. Thus, **GreedyClustering** has complexity $O(|F||E|)$.

3.3 Stochastic Optimization

For a number of NP-hard problems, stochastic optimization has provided solutions which considerably outperform non-heuristic classical algorithms like gradient descent, etc. It is our goal to determine if stochastic optimization can use the domain-specific knowledge incorporated into the graph to outperform **GreedyClustering**.

Objective Function: In stochastic minimization, we produce a number of solutions, whose goodness is based on how much they minimize the *pfr*. Estimating the *pfr* by simulation on test traces is not practical, as the function traces are generally very long. There is a need for an objective function that is computationally inexpensive to evaluate, yet accurate in measuring the *pfr* due to an assignment $Assign \in \mathcal{A}$. We define the objective function (also known as the energy function) to be the sum of weights over all edges, such that the vertices of each such edge are assigned to different partitions, i.e.,

$$Energy(Assign, G) = \sum_{Assign(f_i) \neq Assign(f_j)} w(f_i, f_j) \quad (7)$$

where w is described in Sec. 3.1. This heuristic approximation permits calculation of change in energy caused by a system perturbation in time $O(F)$.

MCA $(ConvergeCriterion, G, Assign_{Current}, BunchSize)$:
// $ConvergeCriterion$: determines if system has converged
// G: the weighted directed graph corresponding to $\{FT\}$
// $Assign_{Current}$: current assignment of functions to pages
// $BunchSize$: the number of functions reassigned at a time
1. $Assign_{Best} = Assign_{Current}$
2. $eDemon = 0$
3. while (notConverged($ConvergeCriterion$))
4. Shuffle order in which functions are reassigned
5. for each bunch of functions
6. $Assign_{New} = Assign_{Current}$
7. for all functions in the bunch
8. remove them from the current page
9. for all functions in the bunch
10. for each page visited in random order
11. if function fits in page
12. update $Assign_{New}$
13. $eDiff = Energy(Assign_{New}, G)$
$- \ Energy(Assign_{Current}, G)$
14. if $eDemon > eDiff$
15. $eDemon = eDemon - eDiff$
16. $Assign_{Current} = Assign_{New}$
17. if $Energy(Assign_{Current}, G) < Energy(Assign_{Best}, G)$
18. $Assign_{Best} = Assign_{Current}$
19. $eDemon = eDemon * DecayRate$

Microcanonical Annealing (MCA): Microcanonical annealing [6], shown above, is a particle physics-based algorithm for the simulation of statistical systems. The system consists of a large number of particles which are in different configurations, i.e., have different position and momentum coordinates. The system as a whole is believed to be isolated and hence it has a constant total energy. Statistical physics states that in such an equilibrium, all states are equally likely [9]. A particle which loses kinetic energy by changing its momentum coordinates, contributes to the potential energy in the system, allowing another particle to use it to increase its kinetic energy (line 12). Creutz [6], assumes the existence of a demon that is in charge of transferring energy from particles giving up energy, to those consuming it. The procedure produces a random walk through configurations. At the end of each such random walk, the demon decays the energy that it holds (line 18), cooling the system and hence the system converges (line 3). Decay rate governs the rate of convergence (we use a decay rate of 0.99).

MCA is similar to simulated annealing in that, unlike other Markov Chain Monte Carlo simulation algorithms [9], transition probabilities change as demon energy decays. It differs in that, unlike simulated annealing, we do not have to compute the probability of a configuration change being accepted or not. We simply see if the demon can accommodate the loss in energy (lines 13-14), if any. Hence, it is compu-

tationally cheaper. Also, we can use a number of demons in parallel, each having their own demon energy, to speed up convergence. In our formulation, the functions correspond to particles, and the position coordinates to their assigned pages. Initially, each function is assigned to a unique page, i.e., the assignment for the program in Sec. 2 would be $\{(f_1),(f_2),(f_3),(f_4),(f_5),(main)\}$. This is a very high energy state, resulting in the early transitions contributing to the demon energy, thus avoiding local minima. For example, consider the transition of f_1 from its current page to that of f_3 to produce legitimate assignment $\{(\phi),(f_2),(f_3,f_1),(f_4),(f_5),(main)\}$. A subsequent random transition is that of f_4 to the earlier page of f_1 resulting in $\{(f_4),(f_2),(f_3,f_1),(\phi),(f_5),(main)\}$. Then, f_5 may transition to the page of f_2 resulting in $\{(f_4),(f_2,f_5),(f_3,f_1),(\phi),(\phi),(main)\}$ raising the demon energy significantly, allowing further transitions. We assume convergence when demon energy is less than 100 and after 500 iterations during which no functions make transitions to lower energy states.

4. EXPERIMENTS

We evaluate our algorithms using two metrics:

1. Page Fault Ratio (PFR): If y is the fraction of code segment in main memory,

$$PFR(y) = \frac{pfr_{Assign_{default}}(y)}{pfr_{optimized}(y)}, y \in [0,1] \quad (8)$$

2. Approximate Working Set Size Ratio (AWSSR): Let $\gamma(x)$ be a function, which returns the minimum number of pages, whose cumulative references account for $(1-x)$ of the total references. As $x \to 0$, $\gamma(x)$ is an approximation to the working set size.

$$AWSSR(x) = \frac{\gamma_{Assign_{default}}(x)}{\gamma_{optimized}(x)}, x \in [0,1] \quad (9)$$

A high PFR implies that the optimized assignment has a considerably lower *pfr* and a high AWSSR implies that the optimized assignment has a smaller approximate working set than the default assignment. The "amortized" PFR is the ratio of the sum of the page faults over all test traces, prior to and after optimization.

In our experiments, we used function traces for two data mining algorithms viz. SPADE [17] and K-metis [12]. Both programs were compiled using g++ with O3 optimization. Table 1 shows the characteristics of the two programs. Out of the set of traces from each program, we use 5 for testing and the remaining for training.

Table 1: Experimental Program Characteristics

Characteristic	SPADE[17]	K-metis[12]
Functionality	Freq Seq Miner	Graph Partitioning
Parameters	data-file min_{sup}	graph-file #(parts)
#(functions)	98	230
#(4KB Pages)	11	43
#(Traces)	20	20
μ(Trace length)	3,227,499	1,781,648
σ(Trace length)	3,322,530	2,679,482

Instrumented SPADE was run on data produced by the IBM data generator [1]. A number of input files for SPADE were generated by varying the parameters determining the number of items in each transaction, the length of the patterns interspersed and the correlation between patterns. These files were fed with different min_{sup} parameters to SPADE and the corresponding function traces were used as input to our program. For K-metis, we created a number of random graphs spanning a range of connectivities and types, i.e., weighted, unweighted, with vertex weights, etc., and varied the number of partitions to split them into and fed these as input to K-metis.

The output of the instrumented code is a sequence of program-specific function identification names/numbers, i.e., FT. The page size is assumed to be 4,096 bytes. By default, we use **GreedyClustering** to produce the assignment and weight the edges of the graph using LSM and use three demons, i.e. $BunchSize = 3$, for **MCA**.

4.1 Effect of domain knowledge on PFR

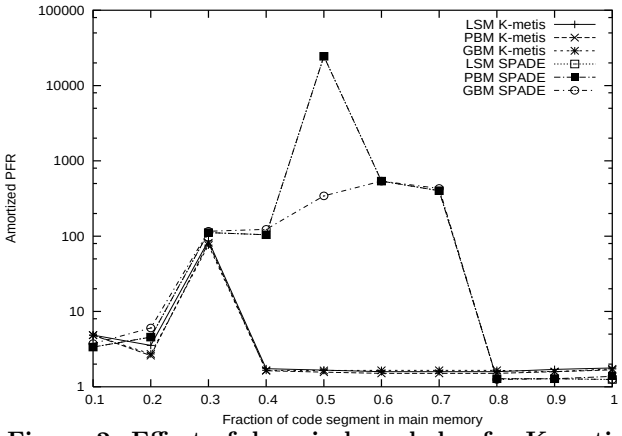

Figure 2: Effect of domain knowledge for K-metis

In Fig. 2, we examine the variation of amortized PFR (Y-axis) as the amount of main memory available to the code segment (X-axis) increases. We examine the effect of the weight function for each of the studied programs. We observe that the three weight functions are not that different. For SPADE, GBM does not have the same peak as the others. Nonetheless, it marginally outperforms LSM and PBM for other fractions of the code segment in main memory.

4.2 GreedyClustering versus MCA

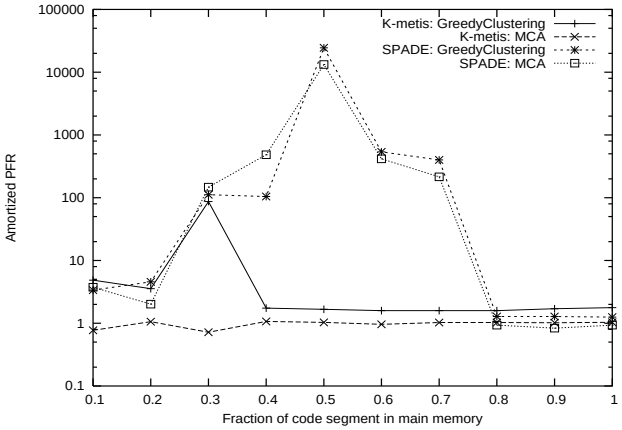

Figure 3: GreedyClustering v/s MCA on PFR

Effect on PFR: In Fig. 3, the amortized PFR obtained as a result of using the two algorithms are compared. It is evident that **GreedyClustering** outperforms **MCA** for

the K-metis program. In fact, the performance of **MCA** is worse than the default assignment for some fractions of the code segment in main memory, as it fails to converge for the parameters provided. For the SPADE program, **MCA** and **GreedyClustering** outperform each other for different fractions of the code segment in main memory. Also note that the PFR peaks and dips for significantly different fractions of the code segment in memory for the two programs.

Effect on AWSSR: In Fig. 4, we have plotted the variation of AWSSR as a function of the fraction of the total references to pages outside the approximate working set, for the different algorithms and programs. For example the bottom-most right-most point has coordinates $(5 \times 10^{-4}, 1.2)$. This means that the SPADE program, when provided an assignment by **MCA**, yields an approximate working set of size 1.2 times smaller than that of the default assignment, where an approximate working set is defined as the minimum number of pages which account for $(1 - 5 \times 10^{-4})$, i.e. 99.95% of the total references. It is evident that **MCA** is outperformed by **GreedyClustering**. AWSSR ≥ 1 implies that code restructuring does decrease the approximate working set size. Experiments were carried out with the code segments being allocated half the number of main memory pages required by them to completely fit in main memory.

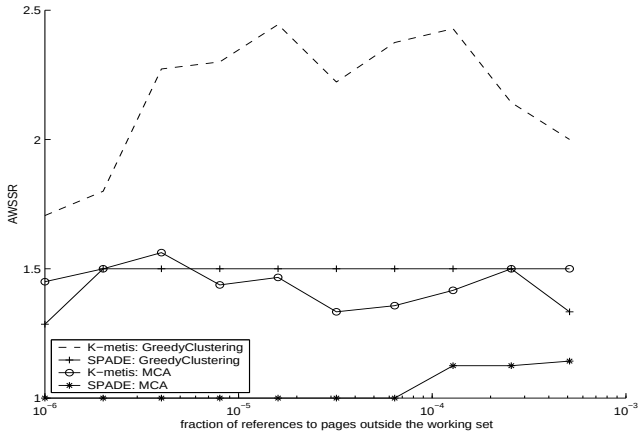

Figure 4: GreedyClustering v/s MCA on AWSSR

5. DISCUSSION

The work presented in this paper is conducted under the PerfMiner project at RPI which aims at developing the PerfMiner engine for performance mining of large-scale data-intensive next generation distributed object applications. The problem of improving the spatial locality of programs via data mining (and other problems in computer systems) opens a new domain for application of data mining techniques. It has a number of interesting aspects to it. The probability distribution of input parameters can evolve, e.g., high compute programs may receive inputs with increasing size over time. Also, unlike a number of other domains, the optimal objective function for assessing the quality of a clustering is known viz., the *pfr*, and hence efficacy of clustering algorithms for this problem can be compared. Solutions to many optimization problems lie in using a greedy algorithm to locate a subspace of interest and then using stochastic refinement. The difficulty in accurately and efficiently modelling *pfr* (Sec. 3.3) makes application of such an approach to this domain hard. Finally, most of the constrained clustering results that we have seen pertain to inter-instance level constraints. However, in our problem, constraints are imposed on the cluster characteristics, i.e., $PageSize$ constrains the sum of the sizes of the functions assigned to a cluster.

Fig. 3 and Fig. 4 imply that the **MCA** and **GreedyClustering** do not have the same relative effect on program locality optimization on both SPADE and K-metis data. **GreedyClustering** generally outperforms **MCA**; the performance is also program specific.

In future work, we intend using domain knowledge influenced techniques to determine better ways to organize secondary memory to minimize page fault cost, and new ways of prefetching to minimize latency and network traffic using proxy servers.

6. REFERENCES

[1] R. Agrawal, R. Srikant. Mining sequential patterns. 11th Intl. Conf. on Data Engingering, 1995.

[2] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, E. Cyirci. Wireless Sensor Networks: A Survey. Computer Networks, 38(4):393-422, 2002.

[3] R. Batchu, S. Levy. Working Sets at Function Level. Rutgers University, Computer Science Dept., Technical Report (DCS TR-480), 1998.

[4] P. S. Bradley, K. P. Bennett, A. Demiriz. Constrained K-means clustering. Technical Report MSR-TR-2000-65, Microsoft Research, 2000.

[5] P. Cao, S. Irani. Cost-Aware WWW Proxy Caching Algs. USENIX Symp. on Internet Tech. and Sys., 1997.

[6] M. Creutz. Microcanonical Monte Carlo Simulation. Physical Review Letters, 50(19):1411-1414, 1983.

[7] N. Gloy, M. D. Smith. Procedure Placement using Temporal-Ordering information. ACM Trans. on Prog. Languages and Systems, 21(5):977-1027, 1999.

[8] D. J. Hatfield, J. Gerald. Program Restructuring for Virtual Memory. IBM Systems Journal, 10(3):168-192, 1971.

[9] L. Herault, R. Horaud. Figure-ground discrimination: Combinatorial Optimization Approach. IEEE Trans. on Pattern Anal. & Machine Intelligence, 15(9):899-914, 1993.

[10] E. Horowitz, S. Sahni, S. Rajasekeran. Computer Algorithms. W. H. Freeman. NY 1998.

[11] J. W. Johnson. Program Restructuring for Virtual Memory Systems. Project MAC TR-148, MIT, 1975.

[12] G. Karypis, V. Kumar. Multilevel k-way Hypergraph Partitioning. Design Automation Conference, 1999.

[13] J. Nesheiwat, B.K. Szymanski. Instrumentation Database System for Performance Analysis of Parallel Scientific Applications. Parallel Computing, 28(10):1409-1449, 2002.

[14] D. Patterson, J. Hennessy. Computer Architecture: A Quantitative Approach. Morgan Kaufmann, 1990.

[15] K. Pettis, R. Hensen. Profile-guided code positioning. In Conf. on Programming Language Design and Implementation, 1990.

[16] K. Wagstaff, C. Cardie, S. Rogers, S. Schroedl. Constrained K-means clustering with Background Knowledge. Int'l Conf. on Machine Learning, 2001.

[17] M. J. Zaki. SPADE: An Efficient Algorithm for Mining Frequent Sequences. Machine Learning Journal, 42(1/2):31-60, 2001.