SOFTWARE CACHE IN GLOBAL HEURISTIC SEARCHES

THESIS

Presented to the Graduate Council of
Texas State University-San Marcos
in Partial Fulfillment of the Requirements
for the Degree
Master of SCIENCE
by
Daniel I. Lowell, B.A.

San Marcos, Texas
May 2010

ACKNOWLEDGEMENTS

Thanks, to my Mother and Father, Eiko and Daniel Lowell. Also thanks to my thesis committee members: Dr. Dan Tamir, Dr. Clara Novoa, Dr. Jim Holt, and Dr. Khosrow Kaikhah. This thesis was submitted on May 4th, 2010.

TABLE OF CONTENTS

Page

LIST OF FIGURES

Figure Page

ABSTRACT

SOFTWARE CACHE IN GLOBAL HEURISTIC SEARCHES

by

Daniel I Lowell, B.A.

Texas State University-San Marcos

May 2010

SUPERVISING PROFESSORS: DAN TAMIR AND CLARA NOVOA

This thesis investigates the time-space tradeoff of cache used with heuristic searches applied to a combinatorial optimization problem known as feature selection. A model of Genetic Algorithm is implemented for selecting feature subsets and ranking data clustered with the ISODATA algorithm. Using a set associative cache, the speedup of Genetic Algorithm and the quality of solutions found is compared to Genetic Algorithm without cache. Together with replacement policies, such as LRU, LFU, and random , several cache set associative configurations are studied, and their relative performance characterized.

Keywords: Genetic Algorithm, Feature Selection, ISODATA,LRU, LFU, set associative, clustering

CHAPTER I

INTRODUCTION

When searching for an optimal solution in a large search space, the efficiency of the search depends on the intelligence applied to the problem. By trading space for time, recordkeeping may be employed to increase the efficiency of a search.

One record keeping model is a dedicated memory. Dedicated memory stores solutions already seen in a search as entries, eliminating the redundancy of processing previously visited solutions. Dedicated memory becomes fixed when full, and its size determines the level of redundancy in the search. Unbound dedicated memory which contains all solutions in the search space when filled, has no redundancy, but may be infeasible depending on the size of the search space. However, given an effective configuration and replacement policy, a cache can be an alternative record keeping method that can create a favorable time versus space complexity tradeoff over dedicated memory (Tanenbaum, 2006).

This thesis explores record keeping strategies in the context of processor *workload characterization* for power consumption tests. Workload characterization records operating temperatures, maximum power distribution, and power consumption, by performing multiple runs of computer code with the final purpose of evaluating the performance of the architecture (Joshi, Eeckhout, John, & Isen, 2008). For major semiconductor companies such as Freescale Semiconductors the performance of

processors under design is of special concern, and an emphasis is placed on the power performance of the designed processors.

One way to determine computer workload characterization involves clustering of fixed size slices of a trace of an application code obtained from a high level instruction set simulator, and identifying prototype slices (Luo, Joshi, Phansalkar, John, & Ghosh, 2008; Brock, 2010). Each trace-slice is represented by a set of features such as the number of arithmetic instructions in the slice, the number of memory references, and the number of register's bits altered. Prototype slices are analyzed through a low level software simulator and the results are used to estimate the performance of the entire code (Luo, Joshi, Phansalkar, John, & Ghosh, 2008; Brock, 2010). Numerous architecture and micro-architecture features can be extracted from the trace. Nevertheless, it is desired to identify an optimal feature sub-set in order to enable cost effective characterization.

There is a tradeoff between accuracy and speed with different architecture testing techniques. The Verilog (or VHDL) code utilized to design the processor along with a Verilog simulator can be used for an accurate evaluation of the processor performance. The Verilog simulator, however, is extremely slow due to the resolution of the computations. On the other hand, an instruction set architecture (ISA) simulator which is fast but inaccurate, can be used to supply several characteristics of the design. Combining the two simulators in an efficient way can be used to construct a relatively accurate and fast tool for performance evaluation. It is a challenging problem addressed in this thesis.

Feature selection and clustering can be used to reduce the amount of code executed on the slow, accurate, low level Verilog simulator while retaining close fidelity to complete test code. This can be accomplished by: 1) dividing the code into blocks, 2) identifying representative blocks, 3) executing the accurate, low level simulator only on the representative blocks, and 4) inferring the performance of the entire code based on the performance of the representative blocks. The two first stages are performed using the code and an ISA simulator and are relatively fast. One can categorize those blocks using clustering techniques with a selected set of performance features and construct an alphabet made up of representative blocks; the number of clusters equaling the number of characters in the alphabet (Friedman & Kandel, 1999).

Alphabet verification is done using the fast Verilog simulator, which introduces another tradeoff between time and accuracy; a small alphabet speeds up the Verilog simulations, but reduces the accuracy of the results. The quality of the clustering technique which is used to identify the alphabet is an important factor in the accuracy of the workload characterization.

The features used for clustering are probably the most important parameter. Therefore, the quality of code block clustering should be based on a good feature selection strategy (Shi, Shu, & Liu, 1998). This feature selection problem (FSP) is a technique which chooses a subset of features while attempting to minimize the effect on the recognition accuracy (Jain & Dubes, 1988). This is done in order to reduce the dimensionality of the feature space; thereby reducing the computational complexity of the pattern recognition task.

FSP is known to be an NP-hard problem and can be stated as a combinatorial optimization problem relevant to the areas of pattern recognition, statistics, and machine learning (Cover & Van Campenhout, 1997; Shi, Shu, & Liu, 1998; Guyon, Weston, & Barnhill, 2002). In many cases, computational complexity considerations and implementation constraints dictate the desired number of features in the selected subset. Under these constraints the FSP boils down to finding the optimal subset of features from a superset of features. This statement of the FSP entails selecting one of the combinations of features according to an optimality criterion (Cover, & Van Campenhout, 1997).

Computational complexity considerations and implementation constraints, can in some cases, dictate the desired number of features k in the selected subset. Under these constraints the FSP boils down to finding the optimal subset of n features from a superset of n features. This statement of the FSP entails selecting one of the $\binom{n}{k}$ combinations of features according to an optimality criteria. Given that FSP is a NP-Hard problem, searching a feature subspace can be a time consuming problem that cannot be tackled efficiently with exact methods, and therefore making a heuristic intelligent search is essential. This thesis implements an anytime heuristic for the FSP. Specifically, Genetic Algorithm (GA) that explores inherent time-space tradeoffs, and investigates reductions in time complexity by the use of record keeping methods such as cache (Ciesielski & Scerri, 1997).

Cache schemes are widely used in hardware and software, and exploit the principle of *locality of reference*. The spatial version of the principle assumes that there is a high probability that instructions or data with addresses that are close to the currently

addressed word are needed next (Tanenbaum, 2006). Hence, when a word is retrieved from main memory some portion of memory data adjacent to the word is loaded into the cache. *Temporal locality* is also important, it assumes that there is a relatively high probability that instructions, or data which have been currently accessed, will be used again in the near future (Tanenbaum, 2006).

Three basic cache replacement policies, and several dynamic/adaptive combinations of the three, are considered in this thesis (Tanenbaum, 2006). The first policy is random replacement. It randomly chooses the block to be evicted. Next, a recency-based method, evicts the least recently used block. This method mainly exploits temporal locality. Finally, a frequency-based method evicts the least frequently used block, thereby exploiting spatial locality. This study implements software data structures and tests these three methods.

The motivation behind combining a cache with GA search technique applied to FSP is to improve the efficiency of the heuristic by taking advantage of spatial and temporal locality. Efficiency in this thesis's context is defined as the number of heuristic evaluations of unique solutions for a given time interval. Efficiency can be considered a speedup if, with respect to a non-cached heuristic, it increases. Since heuristic evaluation has a high time computational cost, improving the efficiency of a search, with the use of a cache, may justify the tradeoff between space and time (Zhang, Fu, Goh, Kwoh, & Lee, 2009).

This thesis also implements a parallelization of GA. GA in particular is an algorithm which lends itself to parallelization due to the independence of the individual chromosome fitness evaluation (Cantu-Paz & Goldberg, 1999; Talbi, 2009).

Investigating PGA has increased relevance due to the trend in designing processors with multiple cores instead of pursuing pure clock speedup (Pacheco, 1997; Chandra, Dagum, Kohr, Maydan, McDonald, & Menon, 2001; Quinn, 2003). This change in approach is mainly due to the power requirements of CPU. Authors have found that in using multiple processing cores and computer code developed for this framework, a high degree speedup can be achieved (Pacheco, 1997; Chandra, Dagum, Kohr, Maydan, McDonald, & Menon, 2001; Quinn, 2003).

In this thesis, parallelization of GA consists of dividing the new chromosome population per generation among processing nodes, such that each processor receives exactly one chromosome. The purpose of this division is to accelerate the process of computing the fitness value associated with each chromosome. The level of parallelization implemented is considered program-level parallelization. GA is divided into sections for single program multiple data (SPMD) execution can be considered coarse-grained (Chandra, Dagum, Kohr, Maydan, McDonald, & Menon, 2001; Quinn, 2003).

Hypothesis

1.) Genetic Algorithm with set associative cache implementation will provide a speedup in performance to that of Genetic Algorithm without cache.

2.) For a given number of generations, Genetic Algorithm with set associative cache implementation will consistently find a feature subset with a higher ISODATA clustering quality than is found in an implementation without cache.

CHAPTER II

LITERATURE SURVEY

The literature on the FSP is extensive since the research on the FSP dates to the early 1960's (Yusta, 2009). This section presents a literature review on heuristic methods to solve the FSP with emphasis on works applying GA's. The section also reviews research on software caching in heuristic searches.

1. Heuristic Methods to Solve the FSP

Sequential forward selection (SFS) and sequential backward selection (SBS) are heuristic methods that have been extensively used in the past to solve the FSP (Nakariyakul, 2008). SFS starts with an empty set and evaluates the improvement in the criterion function from adding one feature at a time. The feature added to the subset is the one that maximizes the criteria function. SFS is repeated until the subset contains the desired number of features. SBS is also an iterative procedure that looks to maximize the criterion function by starting from a set with all the features and removing one feature at a time. SFS and SBS methods are greedy, ignore the interactions among features and suffer from a *nesting effect* since discarded features cannot be re-selected and selected features cannot be removed later. Consequently, SFS and SBS are sub-optimal methods that can be improved by the plus-l-take-away-r method where l steps of SFS are followed by r steps of SBS.

Genetic algorithms (GA) (Siedlecki & Sklansky, 1988)[30], mimetic algorithms (MA) (Yusta, 2009), tabu search (TS) (Wang, Yang, Teng, Xia, & Jensen, 2007; Zhang & Sun, 2007), ant-colonies (AC) (Bello, Puris, Nowe, Martinez, & Garcia, 2006), particle swarm (PS) (Wang, Yang, Teng, Xia, & Jensen, 2007), and GRASP (Yusta, 2009), are heuristic methods researched in recent years for solving the FSP. GA's are based on the principle of natural selection; i.e., survival of the fittest. GA evaluates and improves a finite population of solutions instead of improving a single-solution as in classical hill-climbing methods. Based on this characteristic, some authors describe GA as a parallel algorithm (Kudo & Sklansky, 2000). The optimization process is carried out in cycles or generations. Solutions are represented by chromosomes encoding a particular solution in the solution space, without ambiguity, and each is ranked through a fitness function (Kudo & Sklansky, 2000; Zhang, Fu, Goh, Kwoh, & Lee, 2009; Talbi, 2009). Successful solution of a particular practical problem through GA requires adequate manipulation of components such as: population size, number of generations, and crossover and mutation mechanisms to achieve an adequate balance between exploration and exploitation of the search space (Gen & Cheng, 2000).

Siedlecki and Sklansky (1989), used GA for solving the version of the FSP that searches for the smallest or least costly subset of features for which the classifier's quality does not drop below a pre-defined threshold. The authors compared sequential search, BB, and GA. To speed up the classification process in the range of 10^3 - 10^4, they built a model to simulate the classifier's error rate instead of using the true error rate function of the classifier. Experiments are done with simulated data and with limited tests on real data having 150-300 features. GA resulted more efficient than BB and outperformed

sequential search since it visited the feasible region in a more complete way (Siedlecki & Sklansky, 1989). They did not study the performance of recordkeeping on GA performance.

The work by Siedlecki and Sklansky (1989), evidenced that GA is a powerful tool compared to classical sequential search, especially when there are more than 20 features. A computational study by Kudo, Somol, Pudil, Shimbo, and Sklansky (2000), confirmed the superiority of a well-trained GA by comparing it to adaptive versions of sequential forward/backward feature selection, achieving similar performance in all problem sizes, but a speedup of two or three times in large-scale problems of more than 50 features (Kudo, Somol, Pudil, Shimbo, & Sklansky, 2000). Furthermore, the study compared GA to an extensive set of sequential search algorithms and BB variants, concluding: (1) among the algorithms studied, GA is the only practical choice for large-scale problems (more than 100 features) and (2) GA usually gives better answers than the other algorithms for small and medium scale problems at expense of an increase in computational time (Kudo & Sklansky, 2000).

After a thorough revision to seminal and successful literature on using GA for solving FSP, and the results from a set of preliminary experiments with other heuristics, GA is selected as the platform to solve the FSP and explore performance gains from cache implementation (Siedlecki & Sklansky, 1989; Kudo & Sklansky, 2000; Kudo, Somol, Pudil, Shimbo, & Sklansky, 2000). The evidence in Guan and Zhu's study also gives motivation for the use of GA in solving FSP's (Guan, Zhu, & Li, 2004).

2. Software Caching

Allen and Darwiche (2003) investigated the tradeoff between time and space of an optimal cache population strategy for traversal of decision trees (d-tree) for Bayesian networks. They exploited the hierarchal topology of the trees to accurately predict cache behavior and optimize the cache sizes based on depth-first branch and bound techniques and pruning (Allen & Darwiche, 2003). This technique is focused on reducing cache size while maintaining runtime efficiency. The memory structures studied, however, are tables and do not have the structure of a cache. Furthermore they do not consider replacement policies (Allen & Darwiche, 2003). This thesis is concerned with developing cache for combinatorial optimization problems (COP) which are space efficient. The concern however is improving runtime without degrading quality performance of GA.

Grant and Horsch (2007) investigated efficient cache designs and eviction policies to reduce the runtime of traversing decision trees. The authors' approached the problem of efficient cache design by the use of sub-caching where nodes of a d-tree share cache location entries. Those cache entries were exploited during traversal of the d-tree preventing collision of data from nodes in the cache (Grant & Horsch, 2007). The exploitation of hierarchal sequencing of trees to make cache more space-efficient showed that efficient cache sizes can be maintained while preserving the runtime reduction that the cache provides. This investigation gives a good idea on an approach to software cache design. However, their research does not provide a case where the topology is unknown, or gives an implementation of cache for COPs.

Interesting work on software implemented caching has been done by Aggarwal (2002). In his research he studied how a small software-based cache can significantly improve the performance of data intensive and computationally complex problems (Aggarwal, 2002). Aggarwal found improvements of up to 30% in computational time reduction using caching versus non-caching. Experiments focused on six programs to investigate how caching affects problems with a diverse set of data structures. Though the program problem set used by Aggarwal is diverse, there is no implementation for heuristic optimization problems of the type considered in this thesis (Aggarwal, 2002).

Hertel and Pitassi (2007) also studied time/space trade-offs in the context of heuristic search. They found that time requirements could be significantly reduced through record keeping, referring to their method as caching. Nevertheless, their cache is static and does not consider cache replacement policies, or cache organization issues. This configuration is a better match to the definition of dedicated memory provided in the introduction of this thesis.

Chang and Huang (2009) studied the behavior of hardware cache with GA. Using the principles of temporal and spatial locality, they reordered the sequence of GA's instructions to make the algorithm cache aware, attempting to minimize the miss rate of the simulated cache. In their experiments the cache size used is a 32KB 8-way set associative using a 64B cache line (Chang & Huang, 2009). The authors found that locality of reference became irrelevant when population sizes were small enough to be mostly contained in the cache. The study's main focus is in improving the generic, cache oblivious GA algorithm; simulating computer system hardware cache and driving down

the miss rate. They did not perform a detailed study on different cache replacement strategies or on how a dedicated cache itself can modify the behavior of the generic GA.

Santos et al. (2000), investigated cache diversity in GA. The cache is used to store partial results of the chromosome evaluation function. Nevertheless, they assume that the chromosome evaluation function can be decomposed into small units that represent the evaluation of parts of the chromosome and then recomposed based on mutations and crossovers of these parts. Moreover, despite referring to their record keeping as cache, the record keeping mechanism does not include provisions for replacement policies and can actually be considered as infinite dedicated memory. This limitations makes their approach only suitable for small problems, and for problems where the fitness function computation can be decomposed and recomposed.

3. Parallel Genetic Algorithm with Cache Implementation

There is extensive literature on parallel genetic algorithms (PGA's) but relatively few papers on PGA's with cache implementations. The study by Zhang, Fu, Goh, Kwoh, and Lee (2009) combined PGA with caching for a FSP using support vector machine (SVM) as classification method. After identifying that SVM feature selection has a very large computational cost, the authors implemented parallelization by dividing the population into subpopulations and distributing the workload across multiple processors. Intercommunication of chromosomes across different processor population domains is considered migration, and is implemented using MPI (Zhang Fu, Goh, Kwoh, & Lee, 2009). Caching is implemented in software to eliminate the re-computation of a particular chromosome. Each time, before a chromosome is evaluated, the GA-SVM

checks the cache to see if the chromosome has been run previously. If this is the case, the chromosome is not reevaluated. If it is not the case, the chromosome is evaluated, and the cache is updated with the new chromosome.

The study is similar to this thesis in that the purpose of cache is to limit reevaluations of chromosomes, given that the fitness function evaluation is computationally expensive. The authors do not say whether or not their cache implemented a replacement policy, the configuration, or the size of the cache. These are important details which may determine specific interpretations of behaviors of a software cache. They did find that the cache hits decreased dramatically when features increase for 14 to 123, and that their parallelization had on average a 5.76 times speedup (Zhang Fu, Goh, Kwoh, & Lee, 2009).

Given the gaps in the literature regarding implementation of cache mechanisms for GA's, this thesis explores the time/space tradeoffs associated with record keeping in the form of cache for a PGA that solves the FSP. To the best of our knowledge, the research reported in this paper is the first comprehensive and scalable study of time/space-offs due to caching within the context of PGA.

CHAPTER III

BACKGROUND

This section describes the k-means and the ISODATA clustering algorithm, along with error tolerance and quality measures. GA is presented in detail as it related to combinatorial optimization. Cache theory is divided into cache design and replacement policies. Finally, a brief overview of the motivation for parallelization and the limitations is included.

1. ISODATA and k-means Clustering Algorithms

k-means is an iterative clustering algorithm. A typical way to start the algorithm is by seeding the feature space with randomly selected cluster centers. Using the nearest neighbor method, the k-means algorithm associates patterns with clusters centroids (Jain & Dubes, 1988; Friedman & Kandel, 1999; Aggarwal, 2002). Next, clusters' centroids are recalculated. The process of assigning patterns to clusters and recalculating cluster centers continues until maximum number of iterations has been reached, or the centroids have not been modified within some predefined tolerance over two iterations (Jain & Dubes, 1988; Koza, 1992; Friedman & Kandel, 1999; Aggarwal, 2002). One such tolerance measure is the the Linde-Buzo-Gray (LBG) classification algorithm's minimum distortion error tolerance calculated using the *minimum average distortion* D_a (Linde, Buzo, & Gray, 1980).

The LBG tolerance uses a *vector quantization* mapping function to calculate the distortion; i.e., given a vector set S, a mapping to a representative centroid y using a mapping function:

$$\mathrm{y} = q(\mathrm{S}).$$

The distortion is then the sum of the distances between the representative vector y and S. This process is explained using vector quantization notation.

Let X be a set of n vectors in space R with dimensionality $\{1, \ldots, \mathrm{p}\}$, and the k^{th} cluster have δ_k members, such that:

$$\mathrm{X}^k = \{\mathrm{x}_1^k, \ldots, \mathrm{x}_{\delta_k}^k\},$$

with the j^{th} vector:

$$\mathrm{x}_j^k = [\mathrm{x}_{j1}^k, \ldots, \mathrm{x}_{jp}^k]^{\mathrm{T}}$$

The mapping function produces the set of representative vectors,

$$\mathrm{Y} = q(\mathrm{X}),$$

where $\mathrm{Y} = \{\mathrm{y}_1, \ldots, \mathrm{y}_\mathrm{m}\}$ is the set of centroids for all clusters (Ball & Hall, 1966; Linde, Buzo, & Gray, 1980; Theodoris & Koutroumbas, 1999). The squared error distortion is,

$$d(\mathrm{x}^k, \mathrm{y}^k) = \sum_{\iota=1}^{\delta_k} \left|\mathrm{x}_\iota^k - \mathrm{y}_\iota^k\right|^2.$$

For an iteration 'a' of a clustering algorithm, the minimum average distortion over all vectors in R becomes (Linde, Buzo, & Gray, 1980),

$$\mathrm{D}_a = n^{-1} \sum_{l=1}^{n} min\ d(\mathrm{x}_l, \mathrm{y});\ \ \mathrm{y} \in \mathrm{Y}_a,$$

where y exists in the set Y as a vector at iteration 'a'. The error tolerance is then calculated as,

$$\frac{D_{a-1} - D_a}{D_a} \leq \varepsilon.$$

One drawback to k-means is that the number of clusters (k) is predefined and fixed. This can generate empty clusters, clusters with few members, clusters that are too close to each other, or clusters with large dispersion. The ISODATA algorithm is a clustering technique that tries to overcome this limitation. The goal is to achieve high quality clustering where the number of clusters is between a low bound (say m) and a high bound (say n). The initial clustering of ISODATA is obtained in the same way as in the k-means algorithm (where $k = \frac{m+n}{2}$), using random seeding, and nearest neighbor assignment of patterns to cluster centers (centroids). Once this is done, ISODATA uses predefined or adaptive thresholds to split clusters with high dispersion, merge close clusters, and eliminate clusters with a small number of patterns.

ISODATA performs splitting of clusters which decreases their dispersion, as well as merging of clusters, which increases dispersion between clusters (Koza, 1992; Friedman & Kandel, 1999). The centroids and members of clusters are recalculated in the same way as in k-means. Once the centroids are recalculated, the merge, split, and eliminate steps are repeated. The ISODATA algorithm continues with these operations until one of three conditions exist (Friedman & Kandel, 1999): 1.) an iteration limit is reached, 2.) no splitting, merging, or elimination, has taken place over the last iteration, or 3.) a minimum error has been reached between one iteration and the next. For this thesis, the last criterion is based on the LBG minimum distortion error.

2. Clustering Quality

For an unsupervised clustering algorithm, there must be a means to gauge the "goodness", or quality, of the end result. The criteria for determining the quality of clustering typically involves a measure of clusters' compactness, and a measure of cluster separation. These can be represented by a *within scatter matrix* S_w, and a *between scatter matrix* S_b respectively (Fukunaga, 1990; Dy & Brodley, 2004).

The k^{th} cluster's mean vector of the set X^k is $m^k = [m_1^k, \ldots, m_{\delta_k}^k]^T$. For the i^{th} dimension, δ_k is the number of vectors in the k^{th} cluster, of which the component mean is (Dy & Brodley, 2004):

$$m_i^k = \delta_k^{-1} \sum_{j=1}^{\delta_k} x_{ij}^k$$

A scatter matrix S^k for the k^{th} cluster is composed of the sum of the squared distances between the cluster mean an its member vectors (Fukunaga, 1990; Dy & Brodley, 2004):

$$S^k = \sum_{j=1}^{\delta_k} (x_j^k - m^k)(x_j^k - m^k)^T$$

The *within* scatter matrix is made up of the sum of all internal scatter matrices of clusters in R (Fukunaga, 1990). The number of clusters denoted by *N*:

$$S_w = \sum_{k=1}^{N} S^k.$$

The *between* scatter matrix represents the sum of the distances between the mean of cluster *k* and the overall mean of means of the clusters (Fukunaga, 1990; Dy & Brodley, 2004).

The mean of means:

$$\mathrm{m} = \delta^{-1} \sum_{k=1}^{N} \delta_k \mathrm{m}^k$$

The between scatter matrix (Fukunaga, 1990; Dy & Brodley, 2004;):

$$\mathrm{S}_b = \sum_{k=1}^{N} (\mathrm{m}^k - \mathrm{m})(\mathrm{m}^k - \mathrm{m})^{\mathrm{T}}$$

As stated above, it is often desirable to have clustering which contains compact groupings with good separation between their representative centers. If a cluster is compact, the centroid vector y^kwill more closely represent its member vectors. Large overall mean separation of centroids indicates a well defined representation of data grouping as compared to a close mean separation. In terms of feature selection, compact clusters along with a large mean separation indicates the feature subset is tightly represented by Y; the representative set of centroids (Dy & Brodley, 2004). It follows, that those feature subsets having less compact clusters and a smaller mean separation are poorly represented by Y and of lower quality. This quality measure can be modeled with the trace of the ratio of the *between* scatter matrix over the *within* scatter matrix (Fukunaga, 1990; Dy & Brodley, 2004):

$$\mathrm{Q} = trace(\mathrm{S}_w^{-1}\mathrm{S}_b)$$

The greater the mean separation of the clusters and the more compact the clusters, the higher the quality. The trace of the ratio is applied because it is invariant under any nonsingular transformation (Fukunaga, 1990; Dy & Brodley, 2004). In this thesis, this quality is calculated for each feature subset represented by chromosomes in GA through ISODATA. ISODATA is a relatively complex algorithm which, in some instances,

requires long execution time, therefore, it is desirable that the feature selection utility performs the minimal number of chromosome evaluations (or ISODATA function calls).

3. Genetic Algorithm (GA)

In GA, for combinatorial optimization problems (COP), chromosomes are represented by bit-strings, where each bit represents a feature (Koza, 1992). Features chosen for as a subset are encoded as a 1-bit, whereas those not chosen are represented by a 0-bit. This is shown in figure 1 where A through M are features either chosen or not depending on the bit.

A	B	C	D	E	F	G	H	I	J	K	L	M
0	1	0	1	1	0	0	0	1	1	0	0	0

Figure 1. Bit string representation of a chromosome

Reproduction is done with *crossovers* where a single-point is chosen at random for each parent chromosome pair. Additionally, a double crossover method can be implemented where both parent chromosome is fragmented into three sections such that the child chromosome inherits a middle segments from one parent and an outer segment from the other (Koza, 1992; Talbi, 2009; Obitko, 2009). Figure 2 contains an example of the single point crossover operation where the crossover point is at the third gene of the chromosome. Therefore, the child chromosome inherits the first two genes from the father chromosome B and the reminder ones from the father chromosome A.

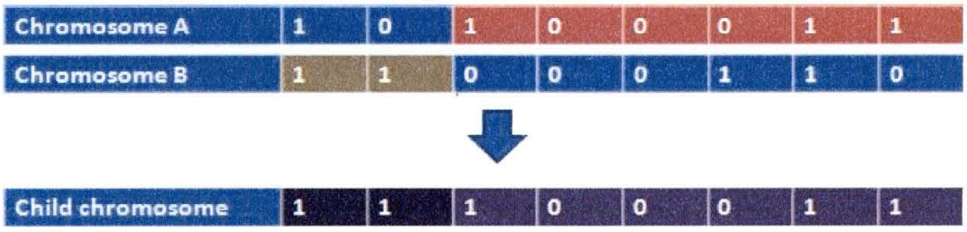

Figure 2. Single point crossover for bit-string chromosome generation.

Often a heuristic may converge toward a local optimum, or optima prematurely, leaving a large portion of the search space unexplored (Koza, 1992; Talbi, 2009; Obitko, 2009). To jump out of this convergence to an unexplored portion of the search space, a mutation is usually implemented. This mutation alters a chromosome to a controlled degree which may or may not improve its fitness (Koza, 1992; Talbi, 2009; Obitko, 2009).

GA begins with the entire population list filled with random chromosomes. The chromosomes are then processed through a fitness function. At the beginning of every subsequent generation, those chromosomes which are below a predefined fitness threshold are culled and are replaced with new chromosomes (Koza, 1992; Talbi, 2009; Obitko, 2009).

The generation of the new chromosomes starts with each child chromosome generated from a pair of parent chromosomes. A child chromosome is produced by a crossover method, and is mutated if the random criterion is met. Once the child list is populated, chromosome's quality is evaluated through a fitness function. The child generation process is shown in figure 3.

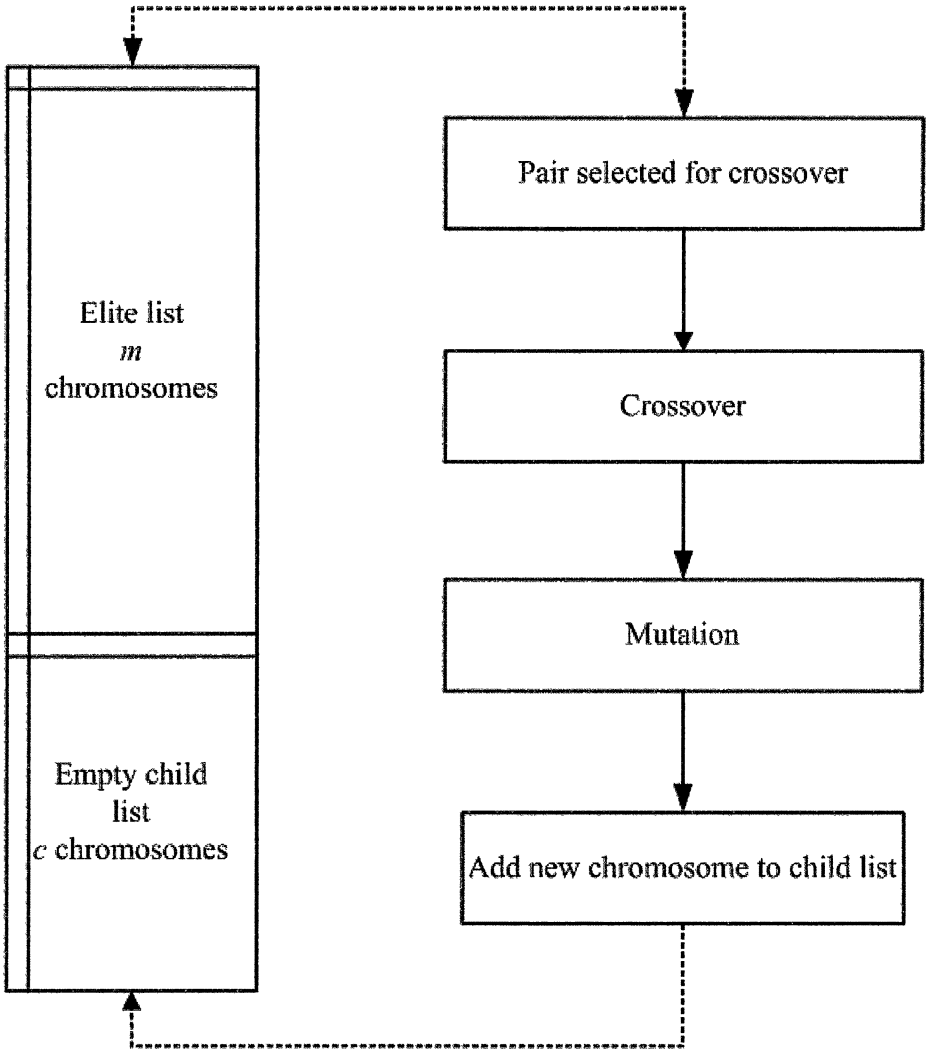

Figure 3. Crossover for bit-string chromosome generation

4. Cache Design

The purpose of cache in computer system is to reduce the time to fetch instructions, or data (words) from memory. Typically this involves, small-scale memory located very close to the CPU (Tanenbaum, 2006). Cache takes advantage of the principle of locality of reference. Locality of reference can be spatial or temporal. Spatial locality predicts that words located close by will be needed soon. Temporal locality says that those words which have been recently accessed may be needed again in a short period of time (Tanenbaum, 2006; Hennesy & Patterson, 2007).

Direct-mapped cache is a simple organization of cache where a block of words is stored in exactly one *block address*. When the cache is accessed, the *reference word* is used to find a word, or perhaps just a byte within a word. An example of a 32-bit reference word (Tanenbaum, 2006; Hennesy & Patterson, 2007):

Block Address bit numbers		Block Offset bit numbers	
31 16	15 5	4 2	1 0
TAG	LINE	WORD	BYTE

Figure 4. 32-bit reference word

Cache entries in direct-mapped cache are laid out in entries which are accessed by the LINE bits from the virtual address. The TAG bits are compared to the tag located at the cache line to see whether the requested block resides in cache and the appropriate word and, or byte offset is then retrieved (O'Hallaron & Bryant, 2002; Tanenbaum, 2006; Hennesy & Patterson, 2007). The choice of the low-order bits for the LINE bits allows consecutive memory locations to be mapped to different cache lines (O'Hallaron & Bryant, 2002). If the LINE bits used the high-order address bits, adjacent memory lines

would be mapped to the same cache line. This would mean that memory locations adjacent (0 to $2^{number\ of\ tag\ bits} - 1$) to each other would be mapped to the same address (O'Hallaron & Bryant, 2002).

Addressing direct-mapped cache with m number of entries is address as (Tanenbaum, 2006; Hennesy & Patterson, 2007):

$$block\ number = R\ (mod\ m).$$

Figure 5 show an example of direct-mapped block addressing. If there are 10 blocks, and the reference address is 14, the block number is 3.

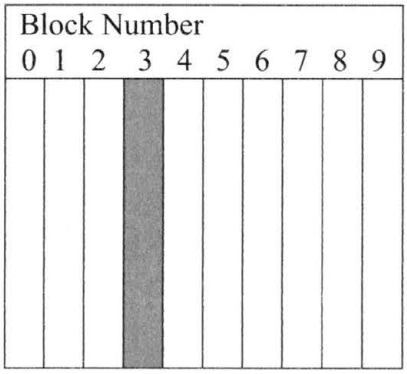

Figure 5. Direct-Mapped Cache

One issue which can occur with direct-mapped cache is if a needed memory line is located at an integral multiple distance away from the size of the line address space, a block which is potentially needed next may be evicted (Hennesy & Patterson, 2007). Set associative cache mitigates this problem by having more than one line for each entry (Tanenbaum, 2006; Hennesy & Patterson, 2007). In this case TAG field of the virtual address is compared, in parallel, with the TAG in each set of the cache line. For an n-set associative cache, and a reference address R, the block address is calculated as:

$$block\ number = R\ (mod\ n),$$

Figure 6. shows an example of set-associative addressing. If the set associativity is 2, and the reference address is 14, the set is 0 and the block can reside either in address 0 or 1.

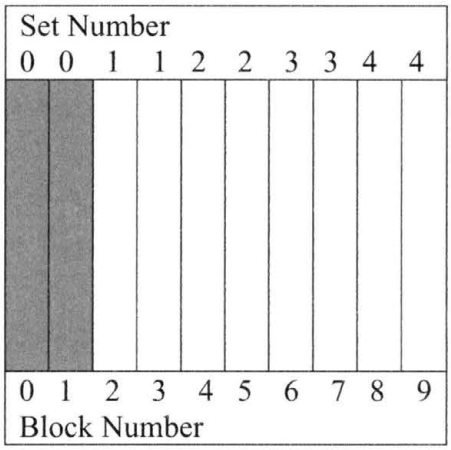

Figure 6. 2 Set-Associative Cache

Fully associative cache is cache where a block may be place in any location. In this model the TAG is not needed, as the entry is directly accessed. This model is very efficient, however, it is expensive to implement in hardware (O'Hallaron & Bryant, 2002; Tanenbaum, 2006; Hennesy & Patterson, 2007). This thesis implements 4, 8, and 16-way set associative, and fully associative cache to study how cache associativity affects the speedup of GA performance.

5. Cache Replacement Policies

There are three cache replacement policies implemented in this thesis; LFU, LRU, and random. *LFU* replacement policy implementation is straightforward. When a cache block is accessed a frequency counter associated with that entry is incremented. If a

write-to cache is performed and a block has to be evicted, the entry with the lowest frequency counter value is evicted, replaced and its counter reset to zero.

LRU is similar to a queue where the order of the cache is determined by the recency of access. In *LRU*, an address where all entries are filled may have an block accessed in a read-only operation. In this case, the counters of the cache entries not accessed at the address which have values less than the value of the accessed entry's counter are incremented. Counters with values greater than the accessed entry's counter remain unchanged and the counter of the accessed entry is set to zero.

The random replacement policy has a simple implementation in which the once the sets of an address are filled, they are evicted by random selection a new data value is to be entered. This replacement policy is useful if there is a situation where words are evicting each other at the same block address repeatedly causing cache misses.

6. Parallelism Performance

Parallelization seeks to speed up the performance of a program by using multiple processors at computationally expensive section of code. Ignoring the computational setup costs, the upper limit ψ for speedup obtained from parallelizing computer code is describe by Amadahl's Law (Quinn, 2003):

$$\psi \leq \frac{1}{f + (1 - f)/p}$$

where f is the fraction of the program instructions which must be run in serial and p is the number of processors available for computation (Amadahl, 1967; Quinn, 2003). Even if there could exist an infinite number of processors for computation, the speed up

is limited to the inverse of the fraction of the program instruction which are run in serial (Quinn, 2003).

The ideal parallel program is one where all sections are parallelized to take full advantage of processing nodes are hand. However, sections of code may be inherently serial. In that case careful software profiling should be done to identify and parallelize, if possible, the most computationally expensive sections, minimizing the impact of serial sections. While the potential for speedup from parallelization is great, care must be taken to maximize its effectiveness. This thesis takes advantage of this methodology.

CHAPTER IV

METHODOLOGY

This section describes the specific methodologies implemented in this thesis. This includes the design on GA and CGA, and dedicated memory. Additionally, the motivation and implementation of the parallelization of GA and CGA are covered.

1. Genetic Algorithm (GA) with ISODATA

In GA implementation, chromosomes represent features selected and they are encoded using a bit string. Given a number of features, those features chosen as a subset are encoded as a 1-bit, whereas those not chosen are represented by a 0-bit. In this thesis chromosomes are limited to a 32 bit bit-string; i.e., a 32-bit unsigned integer encoding a maximum of 32 features. Crossovers are done using a single-point crossover method where the crossover point is chosen at random for each parent chromosome pair. The mutation probability is fixed at 2%.

Since the number of desired features is given, and represented by the number of 1-bits in the chromosome, a valid chromosome must include a number of 1-bits equal to the size of the desired subset. The population list is stored in a data structure is of size 384 where the upper 256 indices are populated with the elite chromosomes (elite list); i.e., those chromosomes which have the highest ISODATA quality and comprise the parent stock. The lower 128 indices contain the lowest quality chromosomes.

Initially, the entire population list is filled with chromosomes and ISODATA is performed in parallel on all individuals, the list is then sorted by quality. At the beginning of every subsequent generation the child list is purged and refilled with new chromosomes, as seen in figure 7. This process follows as:

1.) Each child chromosome is generated from a random pair of elite chromosomes.

2.) A random crossover point is chosen for the parents, and the crossover performed.

3.) The resultant child chromosome is mutated or not depending on whether or not $rand()\%100 < 2$; where $rand()$ the random number generator as implemented in the C standard general utilities library. The number of 1-bits in the child chromosome must be equal to the number of features to be selected. If this condition is not satisfied, then the process of generating a child is repeated.

4.) The child is checked against the elite list for duplication as well as the (filling) child list. It is undesirable to have duplicates in the population for the sake of keeping the elite list from being filled with duplicate high quality solutions.

5.) If the generated child is not a duplicate, it is accepted into the filling child list.

6.) Once the entire child list is populated with 128 chromosomes, their fitness is evaluated through ISODATA clustering. In this thesis, ISODATA is carried out in a parallel batch mode. That is, at each generation, after the list of 128 children is populated, 128 processors are simultaneously assigned to perform ISODATA on exactly 1 chromosome.

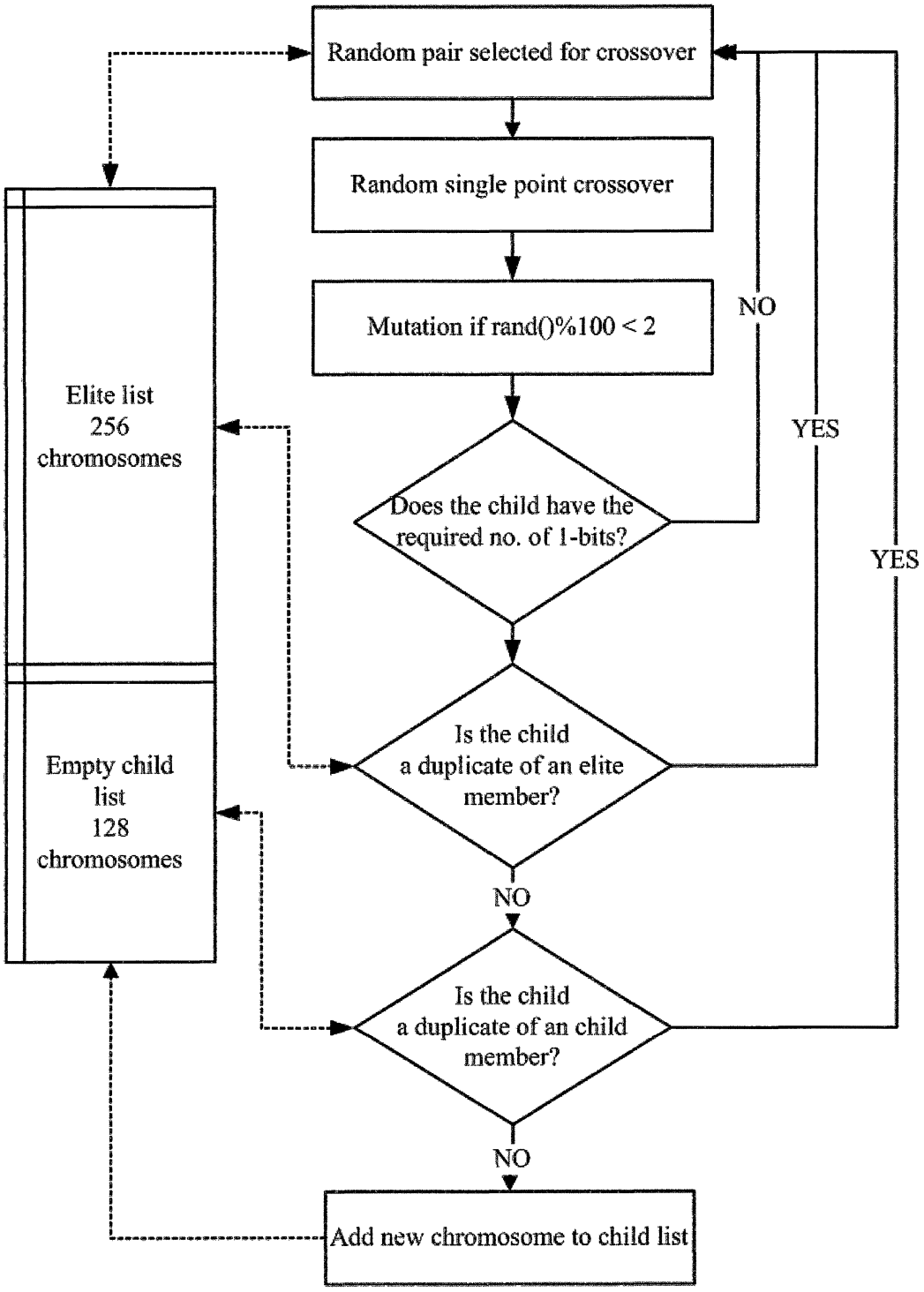

Figure 7. CGA child generation flowchart

7.) The entire 384 member population list is then sorted by quality, using the value of the 256th index of the population list as the cut-off selection rule. Chromosomes that are below the cut-off are considered inferior and can be purged. Due to sorting, it is possible that a child is promoted to the elite list; conversely, an elite member might be demoted.

The termination condition for the GA implementation is designed to be flexible. The algorithm can be configured to exit after a specific number of generations, a maximum number of unique chromosomes generated, or a number of total chromosomes generated.

2. Cache Addressing

Cache addresses for chromosomes are calculated by using the high-order bits of the bit string representation of a chromosome. The tag bits are the low-order bits and is stored in cache in place of a data word. For example, if the chromosome is 25-bits and the number of cache lines are 64k, then the 16 high-order bits is the address and the remaining 9 bits is the tag. This design choice is made because this thesis is primarily interested in minimizing the revisiting of chromosomes in GA.

3. GA Design with Cache (CGA) using ISODATA

With CGA the initial population list of 384 chromosomes is generated and processed in parallel as is done in GA. The entire initial population is then placed into the cache. Like GA, the population list is then sorted, using the 256th index as the cutoff (exactly1/3 of the population list) between the elite list and the child list. The subsequent generations behave in a more complex fashion.

As seen in figure 8, steps 1 and 2 of the CGA progress the same way as GA, however, next New steps 4-6 are as follows:

4.) The child chromosome is compared against the cache in a read-only operation. If the data exists in the cache (a hit), the chromosome is rejected because this means the chromosome has been seen before and does not require reprocessing through ISODATA. Once a chromosome has been rejected, the process of generating a child is begins again, otherwise the chromosome is accepted into the child list. If applicable, the cache counters for LFU and LRU are updated to reflect a hit on an entry.

5.) Once the child list contains 128 new chromosomes, their selected features are processed through ISODATA in parallel. Upon return of the ISODATA function, for all children, the entire population list is sorted by quality. Those chromosomes below the 256th index are considered inferior and are to be placed into the cache. This is done by taking chromosomes in order from below the 256th index and checking if they are in the cache (read/write operation). If there is an entry that contains the chromosome, then this is considered a cache hit. The counters of the cache are updated and the chromosome is purged from the population list. If instead, there is no match for the chromosome value, but all cache entries are populated for that address, it is considered a cache miss and a replacement policy is implemented. If there is no match at the chromosome's cache address, but an empty entry exists,

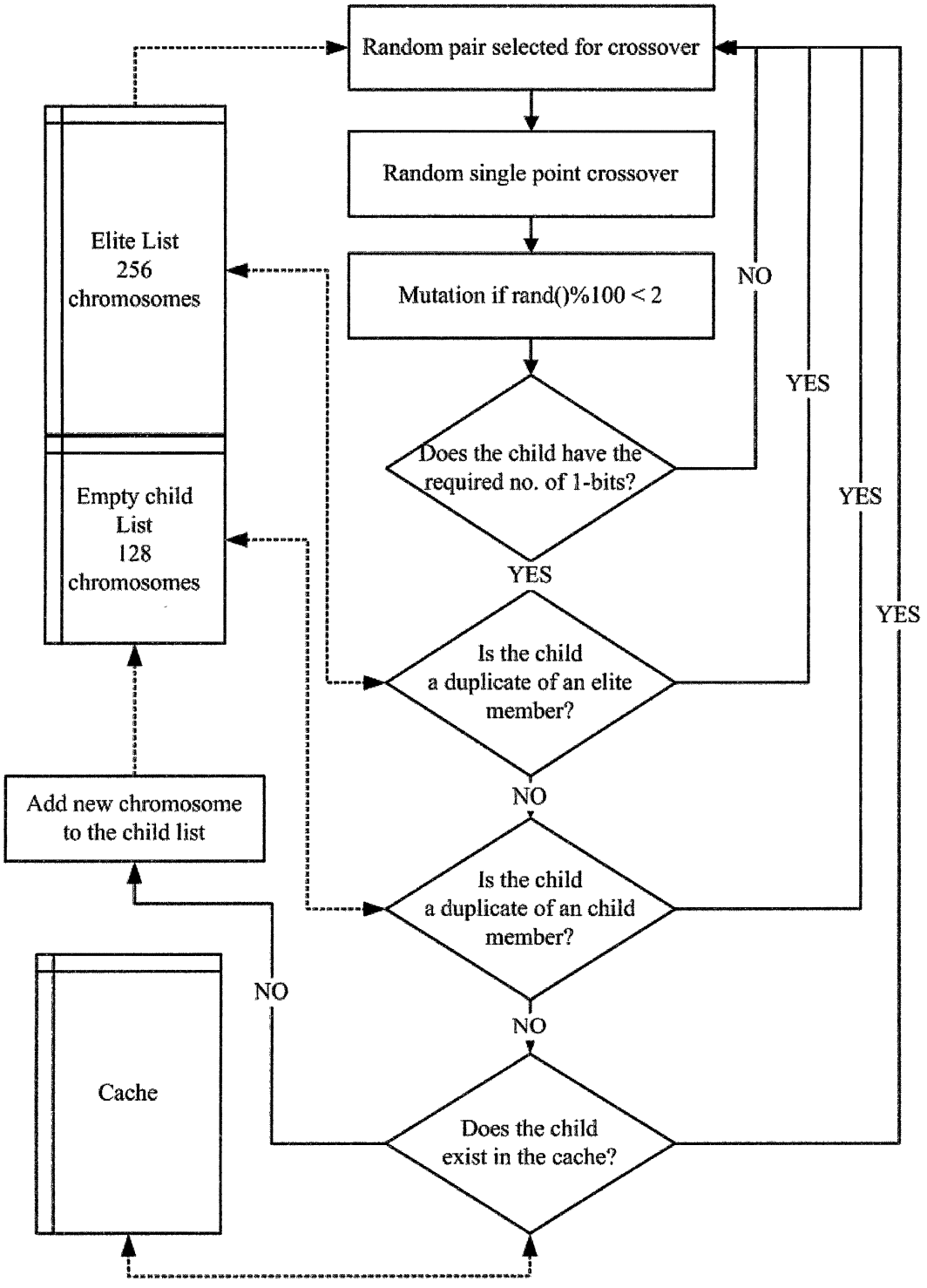

Figure 8. CGA child generation flowchart

this too is a cache miss (compulsory miss), and the empty entry is filled with the value of the chromosome's data value.

6.) After the lower 128 indices have updated the cache and are purged of values, the generation has completed and the process of filling the child list begins again. As with GA, the termination condition for CGA can be configured to exit after a specific number of generations, a maximum number of unique chromosomes generated, or a number of total chromosomes generated.

The cache is accessed in two places in CGA. One way in a read-only mode, and the other in a read-write mode. The read-only mode occurs when the child generating function (CG) checks the cache to see if the child it has produced is a revisited chromosome. Regardless if it is in cache, the function will only update the counters and not update the entries. For LFU the counter update consists of incrementing the counter for the look up entry. LRU's counter update consists of reassigning the order of the most recently seen block.

After the new child population list is processed though ISODATA and assigned a quality value, the cache update function (CU) is called and the read-write cache access takes place. The first task CU performs is to sort the entire population list, which includes the elite list plus the newly processed children. The list is sorted by descending value of quality with the best quality chromosome at the top of the population array. This action places the worse 128 chromosomes below the 1/3 cut off as undesirable feature subsets.

The CU now checks each bad chromosome against the cache. If there is a miss CU either replaces a cache entry, or if the cache line is unfilled, places a cache entry and updated the counters. If there is a hit, the cache only updates the counters.

Two experimental variables which are important indicators of cache performance are the hit percentage, and speed up. Hit percentage is defined as:

$$hit\ percentage = 100 \times \frac{H}{(H + M - M_c)}$$

where H is the number of cache hits, M is the number of cache misses, and M_c is the number of *compulsory misses*. Compulsory misses are cache misses due to an empty cache line.

Speedup is defined as the ratio of the number of times ISODATA is called for a particular cache, or dedicated memory, size to the number of times ISODATA is called in a baseline GA.

Choosing a fixed number of features selected for CGA experiments creates the problem of block addresses which cannot addressed. For block addresses requiring a number of addressing bits less than, or equal to the number of subset features, or dimensions, selected out of a bit string, the entire cache will be addressable. However, if the number of cache addressing bits is larger than the subset number there will be addresses which are unreachable.

The percentage of the cache addresses which cannot be accessed for a given subset number and a size of cache can be calculated. If $X = \log_2(cache\ addresses)$ is number of bits required to address a cache block, S is the number of subset features selected, and Z is the number of inaccessible addresses:

then the unaddressed percentage is:

—

Figure 9 presents the results for the percentage of cache addresses which are not addressable versus sizes of cache. Displayed are cache sizes ranging from 16k to 2M for 25 features choosing 13. The ascending trend in the figure indicates that larger caches have a larger number of non-addressable blocks.

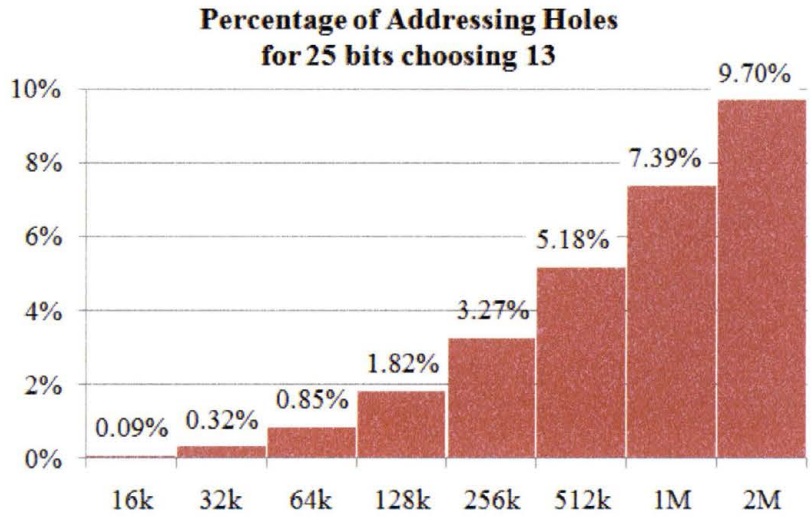

Figure 9. Non-addressable cache block percentages for 25 bits choosing 13

4. Dedicated and Unbounded Memory

In dedicated memory, space is allocated to hold a specific number of chromosomes visited and there is no replacement policy. Once the memory fills with solutions it becomes a fixed lookup table. Unbounded memory is the dedicated memory

model which is of a size sufficient to hold all the solutions encountered during a search in the solution space. The upper limit on the size of the unbounded memory is $\binom{M}{K}$, where M is the number of features and K is the subset number of features.

5. Parallel Implementation

All experiments, parallel and serial, in this thesis are run on the *Texas Advanced Computing Center's* (TACC) *Ranger* Linux Cluster available at the J.J. Pickle Research Center at the University of Texas-Austin. Ranger compute nodes configurations are four AMD 2.3Ghz quad-core Opterons per system board interconnected via Infiniband network ("Ranger User Guide", 2009). The programming language used to design all experiments is C which implements the openMPI application program interface (API).

This thesis has a core premise that the computational cost of the heuristic function, in this case ISODATA, is large when compared to remainder of the search algorithm. This premise justifies the time-space tradeoff of implementing cache. As a consequence of this premise, and given Amadahl's Law, parallelization of ISODATA would seem to be the logical choice. Profiling of GA using the profiling tool *gprof* shows that indeed ISODATA function calls per generation consume approximately 90% of the computational time. As stated in the introduction to this thesis, the parallelization in CGA consists of dividing the accumulated child list chromosomes across processors in every generation. This model does not reduce the runtime of an individual ISODATA function call, instead processes the entire child list in parallel in the time interval required for the processing of a single ISODATA function, plus MPI communications overhead.

CHAPTER V

EXPERIMENTAL DESIGN

This section presents the GA implementation in the FSP domain with and without caching. Cache implementations include set associative and fully associative. One trace program is implemented to test temporal locality. Also, one sequential search program is implemented to exhaustively check the quality of chromosomes. Two synthetic datasets are also developed.

1. Datasets

Raw data provided by Freescale Semiconductor is used to evaluate the performance of the proposed GA with caching (CGA). The data contains a trace of four computer benchmark programs, including fast Fourier transform (FFT), the Dijkstra's shortest path algorithm (DJK), quick-sort (QS), and basic mathematics suite (BMS). FFT and BMS both have 23 features, while DJK and QS both have 25 features. To maximize the size of the feature space a subset number of 12 features are chosen for FFT and BMS, giving a search space of 1,352,078 combinations. For DJK and QS the feature subset size is chosen as 13, resulting in a search space of 5,200,300 combinations.

Each of the above traces is divided into fixed length sequences of instructions referred to as slices. The sizes of the slices examined are 1000, 2000, 5000, and 10000 instructions. Following a feature extraction stage applied to slices, each slice is

represented by a set of architecture and micro-architecture features such as the number of integer operations per slice, the number of register transfers, and the number of memory accesses. These sets of features go through the feature selection stage described below, where an optimal subset of the features is sought.

Synthetic data are also developed for cache testing. The number of features and the number of data points for the synthetic data are tailored to match the DJK dataset; i.e., 25 dimensions, with 1000 data points. Further details about the 2 created datasets and their properties are extensively detailed in the next chapter.

2. ISODATA Parameters

The ISODATA algorithm has several tunable parameters, as listed:

LBG Error Tolerance: the minimum error of the distortion between two ISODATA iterations for sufficient convergence. It signals exit an condition.

Lump threshold: the maximum distance between cluster centers where two clusters are merged into one.

Split threshold: the minimum mean distance of all data points from the center of a cluster which causes two clusters it is divided into two across the dimension with the maximum mean distance.

Minimum Cluster Size: number of data points a cluster must have, otherwise it is removed as a cluster and the data points are distributed to nearby clusters using the nearest neighbor rule.

Maximum Clusters Lumped per Iteration: moderates the influence of the lumping phase of ISODATA.

Split Fraction: a fractional multiplier which moderates the distance of new cluster centers for any split.

Maximum Iterations: input parameter that allows for an early exit from ISODATA before convergence.

Initial Number of Clusters: Number of seed clusters for k-means initialization.

Through experimentation the parameter values are found and set which allow a mix of smooth convergence and clustering execution speed:

LBG Error Tolerance:	10^{-4}
Lump threshold:	10^{-5}
Split threshold:	2.5×10^{-4}
Minimum Cluster Size:	10
Max Clusters Lumped:	4
Split Fraction:	0.75
Maximum Iterations:	15
Initial Number of Clusters:	32

3. GA Data Collection

The series of experiments describes in this section establish a control model to this study. The data collected here are the basis of all experiments in this thesis. It provides an upper bound on the speedup of GA with cache and with dedicated memory.

Variables listed in this section are also collected in all other experimental sections. The following trace variables are collected to establish the baseline performance of GA without enhancements:

Number of unique chromosomes: The total number of unique chromosomes produced and processed through ISODATA.

Number of unique chromosomes per generation: Per generation number of unique chromosomes in the child population processed through ISODATA.

Number of ISODATA calls: The total number of times ISODATA is called on chromosomes; unique and revisited chromosomes.

Number of chromosomes generated: The total number of chromosomes generated regardless if they are run through ISODATA or not.

Time elapsed: Total wall time for execution of program.

Time for child calls per generation: Time elapsed between call of child population generator function and its return per generation.

Time per generation loop: Time required to execute a generation.

Elite list quality: The qualities of all elite list members.

Elite list chromosomes: Elite/parent list chromosomes. These are the features selected with the highest quality.

Top quality per generation: Trace of the best quality as it exists per generation.

4. CGA Data Collection

Using CGA, data are collected for each type of cache replacement policy; LFU, LRU, and random. The cache sizes are set between 32k ~ 2M with 4, 8, and 16 set associativity. In addition to all of GA data variables recorded, CGA records many cache variables.

These include:

Number of hits: The total number of cache hits inside the child function.

Number of misses: The total number of cache misses inside the child function.

Number of hits per generation: The number of read-only cache lookup hits per generation inside the child function.

Number of misses per generation: The number of read-only cache lookup misses per generation inside the child function.

Number of placements into cache: The total number of compulsory read-write cache misses encountered in the cache update function resulting in a placement of a chromosome segment into cache. Compulsory miss is a miss due to an unfilled cache entry set.

Number of replacements to cache: The total number of read-write cache misses encountered in the cache update function resulting in a replacement of a chromosome segment using the current replacement policy.

Final cache state: The complete representation of the final state of the cache data structure.

5. List of Experiments

This section describes the different categories of experiments which are performed.

i.) Temporal Locality of Reference

The degree of data temporal or spatial locality influences the cache performance in terms of hits and misses. Spatial locality does not exist in bit-string chromosome addressing of cache, therefore it is not considered. However, these series of experiments investigates the degree of temporal locality.

To test the locality, GA is run for 500 generations on all 4 Freescale Semiconductor datasets. The trace program running GA records the time interval of reappearances of each unique chromosome as well as the number of times a unique chromosome is revisited.

ii.) Baseline GA

These sets of experiments establish a lower bound on CGA performance since there is actually no cache. In the baseline experimental set, GA is run on all Freescale Semiconductor datasets. Each dataset is run once and GA's termination condition is set at 20,000 generations.

iii.) Dedicated Memory

Dedicated memory experiments follow a different design to CGA experiments. Memory is allocated to hold a specific number of chromosomes visited. This model is intended to show a theoretical baseline for time-space tradeoff.

There is no replacement policy for dedicated memory, once the memory fills with solutions it becomes a fixed lookup table. Data obtained in this section includes those found in the GA experimental section. The purpose of these experiments is to establish another control model. This control separates the effects of memory without replacement polices from cache with a replacement policy.

The experiments allocate memory at doubling intervals; 4k, 8k, 16k, 32k, 64k, and 128k. Each memory size is run 4 times, with different random number generator seeds, for 100k generations, and is run on all 4 Freescale Semiconductor datasets. The number of experiments total of 96.

iv.) Indicative Cache Sizes

The first set of CGA experiments tests various cache sizes and set associativities with all Freescale Semiconductor datasets. The experiments use the LFU replacement policy, and do not have multi-seed replicates per dataset, and are therefore indicative. The cache sizes studied here are 64k, 128k, 256k, 512k, and 1M, with set associativity: 4, 8, and 16. These parameters are tested in all combinations totaling 60 experiments.

For this set of experiments CGA termination condition is reached when the number of unique chromosomes produced equals the final number of unique chromosomes produced in experimental set 1(i.e., i). For example, if the Dijkstra dataset, run for 20k generations in experiment 1 and produces a final total of 50k unique chromosomes, then 50k becomes the termination condition for the current experiment set.

v.) Cache Replacement Policies

This set of experiments tests the effect of three different replacement policies on CGA; LRU, LFU, and random. This set uses all four Freescale Semiconductor datasets. The experiments have 4 replicates using different random number generator seeds. In this set of experiments, CGA terminates when 100k unique chromosomes are produced. The experiments are broken into two main groups depending on the number of features in each dataset.

The first group uses the datasets FFT and BMS. These two datasets have 23 features each. Cache sizes are 256k and 512k with set associativities of 8 and 16, using all three replacement policies; LRU, LFU, and random. The total number of experiments for this group is 48.

The second group includes the datasets DJK and QS. Their cache sizes include 512k and 1M with 8 and 16 set associativity, and all replacement policies. The total number of experiments for this groups is also, 48. The total number of experiments related to cache replacement policies is 96.

vi.) Exhaustive Quality

The purpose of this set of experiments is to do an exhaustive quality check on all datasets. This includes the 4 Freescale Semiconductor datasets as well as two synthetic datasets. For DJK, QS, and both synthetic datasets, 25 choosing 13 is used as the combination number for the chromosome bit string. For FFT and BMS the bit string chromosome combinations are 23 choosing 12. The choice of these combinations is to maximize the search space for each dataset's.

To perform this set of experiments, a program is developed to sequentially step through all combinations detailed above. The total number of experiments is 6.

vii.) Generation of Synthetic Data

This set of experiments investigates different synthetic datasets, which differ in quality profile significantly from the Freescale Semiconductor datasets. Multiple datasets are generated and processed exhaustively though ISODATA to determine the quality distributions. One or two of these datasets are selected for further testing.

viii.) Floating Number of Combinatorial Subsets

The floating set of experiments lifts the restriction of the number features a subset can have. This allows CGA to process feature subsets through ISODATA which have any number of features, including 1 and M, where M is the feature set size.

The DJK dataset and two synthetic datasets are processed though this floating CGA experiment. Three cache sizes: 128k, 512k, 128k are tested with 8 set associativity, and run with the LRU replacement policy. There are no replicates, so the total number of experiments here is 9.

ix.) Synthetic Data Testing

The purpose is to compare the behavior of CGA, seen in experiments 4 and 5, on a synthetic dataset having a different quality profile. The cache sizes tested are 512k and 2M with 8 set associative cache. The replacement policies LRU, LFU, and random are

tested. All experiments are done with 4 replicates, using a termination condition of 50k unique chromosomes. The total number of experiments in this section is 60.

x.) Fully Associative Cache

This last set of experiments tests the behavior of fully associative cache to compare the results with set associative cache and with dedicated memory. Two datasets are chosen, DJK and one synthetic data set. The cache sizes double in size from 2k to 128k, with LRU as the replacement policy. The experiments are done with 4 replicates for a total of 56.

CHAPTER VI

EXPERIMENTS AND RESULTS

This section reports the experiments performed and their results. A sequence of experiments and their conclusions are arranged by type and cache configurations. The memory configurations include: no memory, set associative cache, and fully associative cache. Each section contains relevant figures and conclusions.

1. Temporal Locality

An initial investigation of this thesis is to test the temporal locality of GA. The data is collected on all Freescale Semiconductor datasets and records the interval, in generations, between an appearance of a particular chromosome and its next appearance. The number of times individual interval lengths occur over the entire experiment are totaled and plotted as a distribution of frequencies in figures 10 and 11. Freescale Semiconductor datasets which have the same search space size are plotted together. That is, figure 10 plots FFT and BMS datasets with 23 features, run in GA with the combinatorial scale of $\binom{23}{12} = 1.35 \times 10^6$. Figure 11 plots the Dijkstra and quicksort datasets, both of which are run in GA with the combinatorial scale of $\binom{25}{13} = 5.2 \times 10^6$.

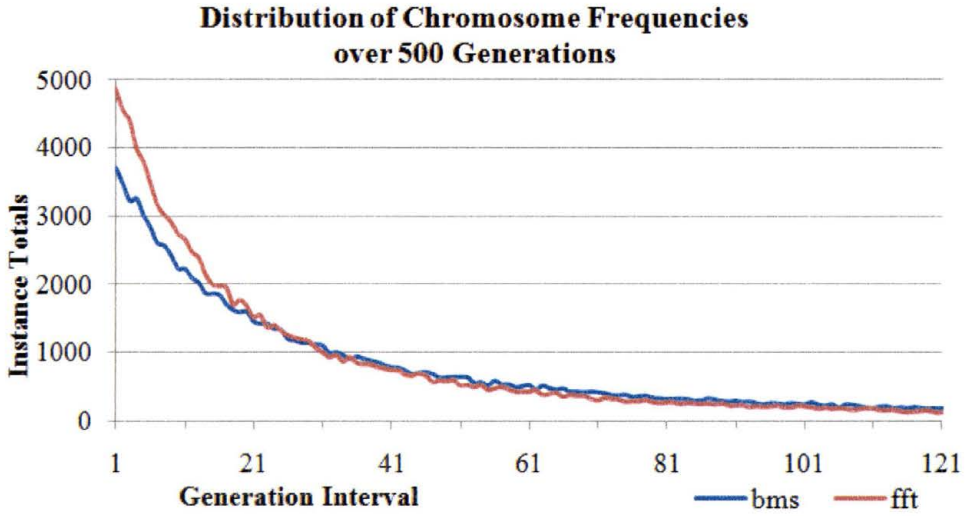

Figure 10. FFT and BMS frequency distribution

Figure 10 demonstrates that GA exhibits a high degree of temporal locality using FFT and BMS. The frequency distribution for this experiment indicates that for BMS the chance of a particular chromosome seen again within 50 generations is approximately 64%, while the chance that chromosome will be seen within 10 generations is approximately 21%. For the dataset FFT the distribution shows slightly greater temporal locality, with the chance of a chromosome being seen again within 50 generations approximately 72%, and in 10 generations, approximately 27%.

The Dijkstra and quicksort dataset frequency distribution, figure 11 shows a peak frequency (frequency of 1 generation) an order of magnitude smaller than those of figure 10. A reduction of the number of individual chromosome repetitions is expected due to the larger feature space.

The degree of locality, is similar to the BMS and FFT datasets. In the case of Dijsktra dataset, the chances of seeing a chromosome again within 50 generations is approximately 66%, while for quicksort the chances are closer to 75%. The chances a

chromosome will be seen again in 10 generations is similar to that in figure 10, with approximately a 20% chance for Dijkstra, and approximately 30% for quicksort.

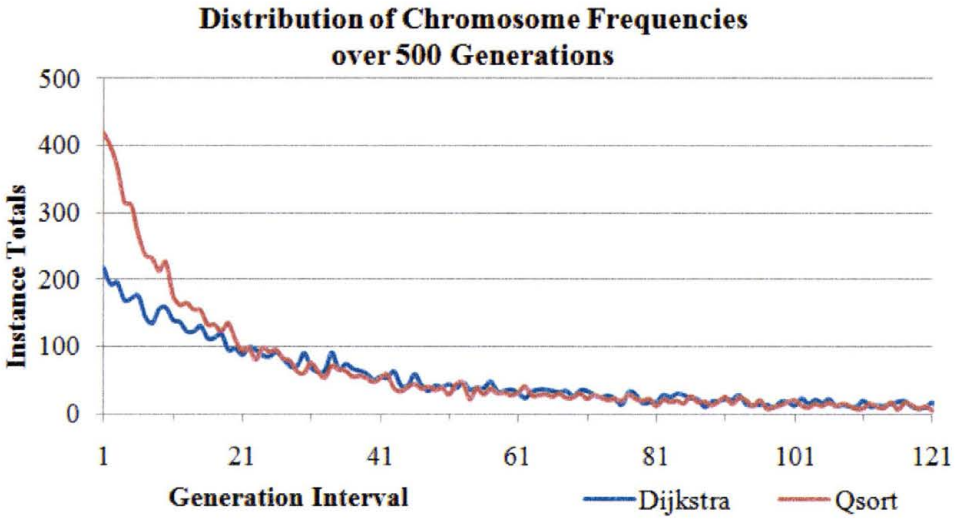

Figure 11. DJK and QS frequency distributions

These set of experiments, with GA and Freescale Semiconductor data, shows evidence that that temporal locality exists and can be exploited with CGA. There is little variation between datasets which suggests either a similarity between datasets, or an inherent locality of reference which exists in GA. Both, could also be the case.

2. Baseline Experiments for GA

This series of experiments gauges the redundancy ratio of GA with Freescale Semiconductor data. (as shown in figure 12). In this thesis, the redundancy ratio is defined as the number ISODATA function calls divided by the number of unique chromosomes generated. For example, if the GA produces 100k unique chromosomes, and the number of chromosomes processed through ISODATA to find those unique

chromosomes is 2M, then the redundancy is ——— . The number of ISODATA function calls is the number of chromosomes processed whether or not they have been seen before. The redundancy ratio describes the baseline inefficiency of GA, or the upper bound of speedup. Figure 12 shows the redundancy associated with the four Freescale Semiconductor datasets run through GA for 20,000 generations.

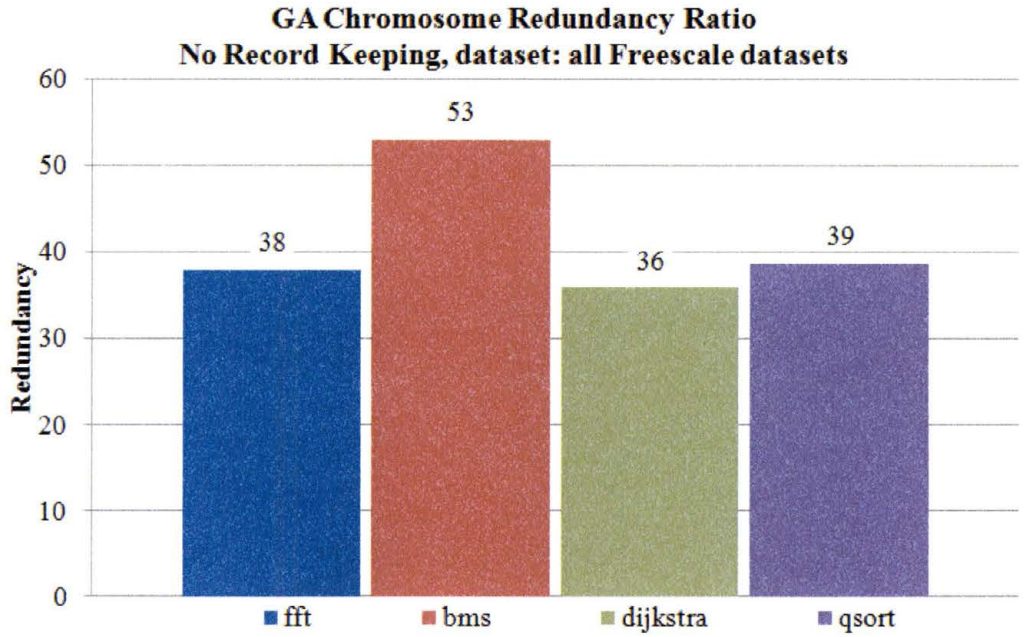

Figure 12. Redundancy for Freescale Semiconductor run through GA

The dataset FFT has 38x redundancy, BMS has 53x redundancy, while the Dijkstra and quicksort datasets have 36x and 39x redundancy respectively. The redundancy across the datasets is consistent with the previous sections results. Also, the larger redundancy of the dataset BMS indicates there is a degree of data dependency related to redundancy.

3. Dedicated Memory

The next series of experiments studies of the time-space tradeoff resulting from various of dedicated memory. The results presented in this section are obtained from experiments with 4 replicates using different random seeds and averaged.

Of particular interest is the speedup over GA, and the total hit percentage associated with different memory sizes. GA is run until 100k unique chromosomes are generated. Figure 13 shows the speedup related to doubling of memory size beginning with 2k entries and ending in 128k entries for the BMS dataset.

The experiments are run until 100k unique chromosomes have been processed by ISODATA, therefore, the memory size of 128k can hold every unique chromosome generated by GA; i.e., there are no misses at this scale. While not unbounded memory, the 128k size memory has the identical effect on GA performance and can be considered as the upper bound on speedup and hit percentage.

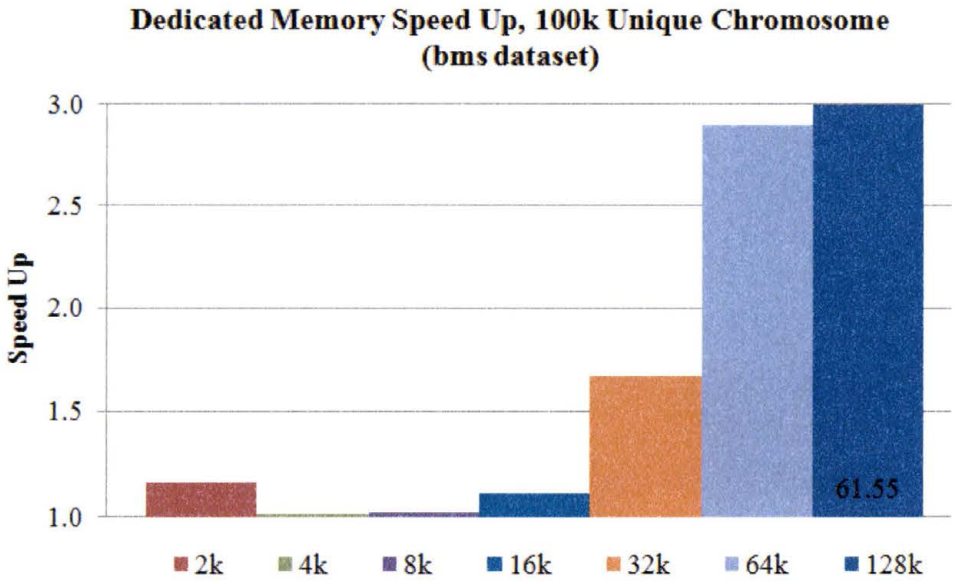

Figure 13. BMS dedicated memory speedup

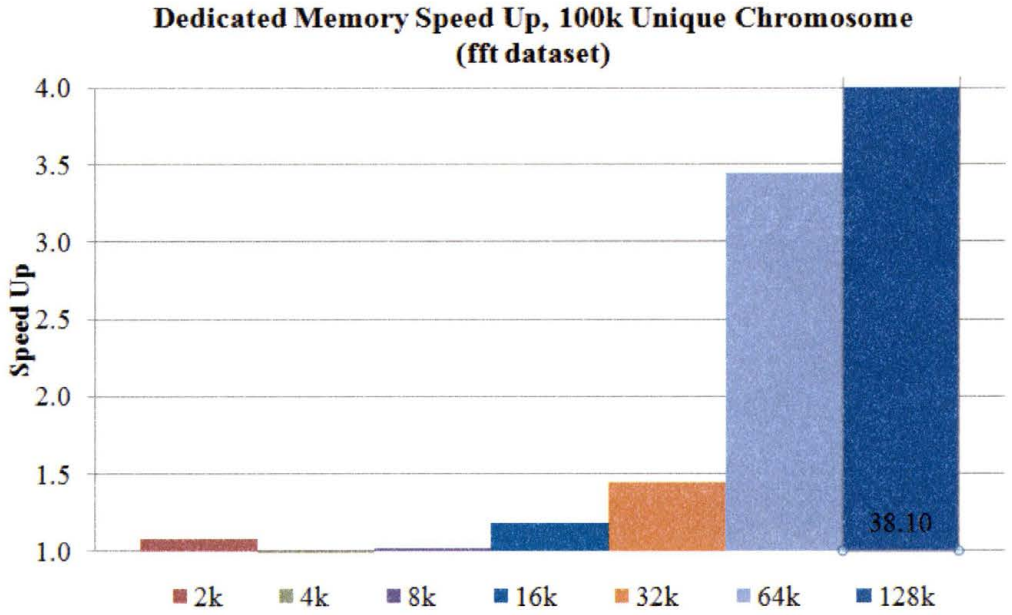

Figure 14. FFT dedicated memory speedup

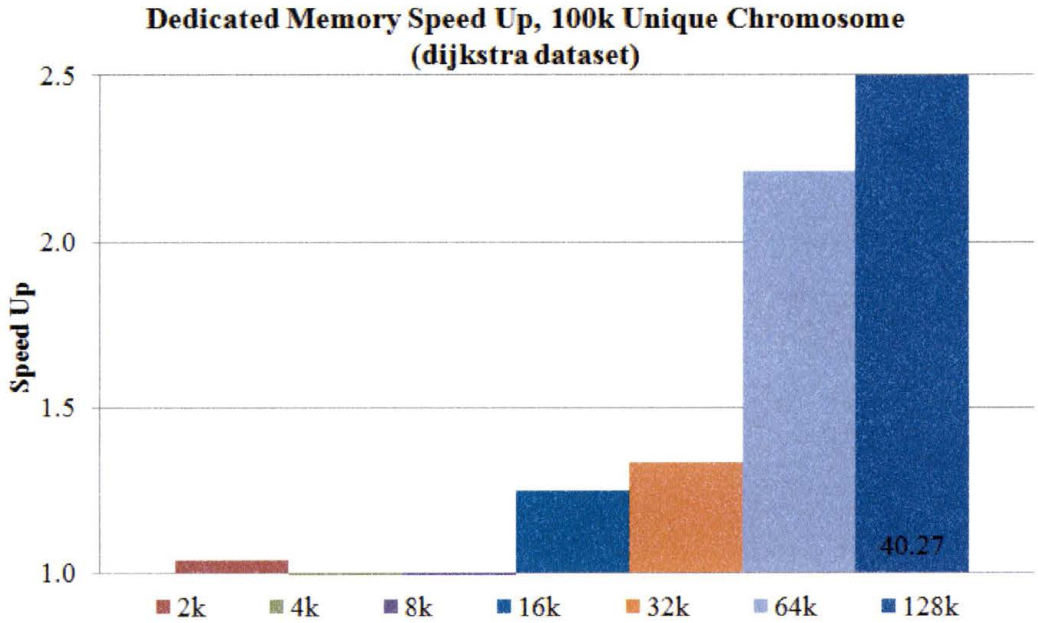

Figure 15. DJK dedicated memory speedup

For all Freescale Semiconductor datasets there is less than 50% speedup using memory sizes up to 32k (with the exception of the BMS dataset at 32k having a speedup

closer to 60%). At 64k, there is a jump in speedup. For datasets FFT and quicksort there is nearly a quadrupling of the speedup, while BMS and Dijkstra the jump is closer to triple the speedup.

At the upper bound of the dedicated memory, the datasets exhibit similar speedups of approximately 40x, a notable exception is BMS. BMS shows not only a higher degree of redundancy at baseline GA, but also, a greater maximum speedup. How significant this variation is, is unclear.

4. Dedicated Memory Hit Percentage

Figure 16 presents the hit percentage for dedicated memory using the BMS dataset. The upper bound memory size 128k, with a hit percentage of 100%, is omitted from the figure. The data here are from the same experimental runs as those from the previous section.

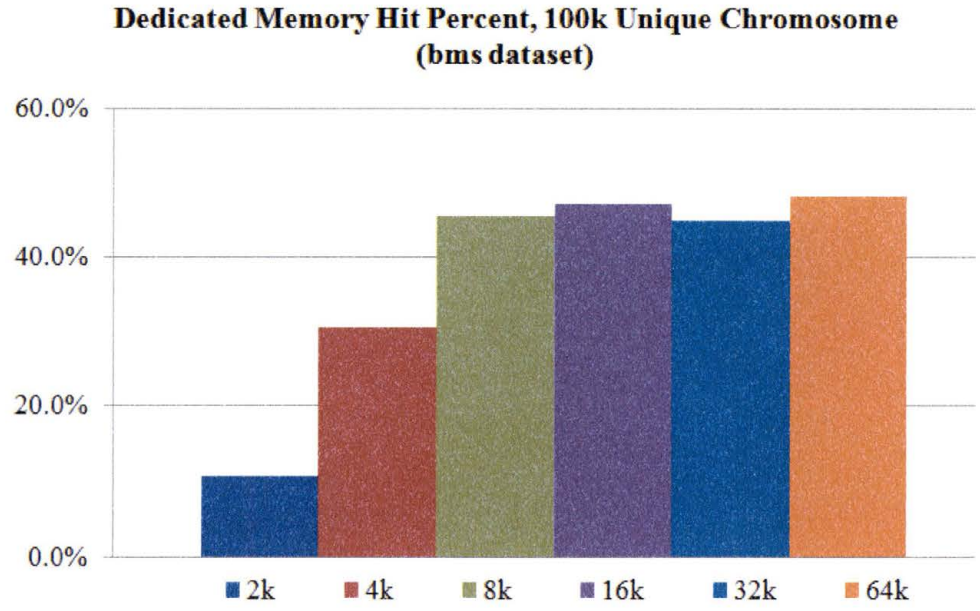

Figure 16. BMS dedicated memory hit percentage

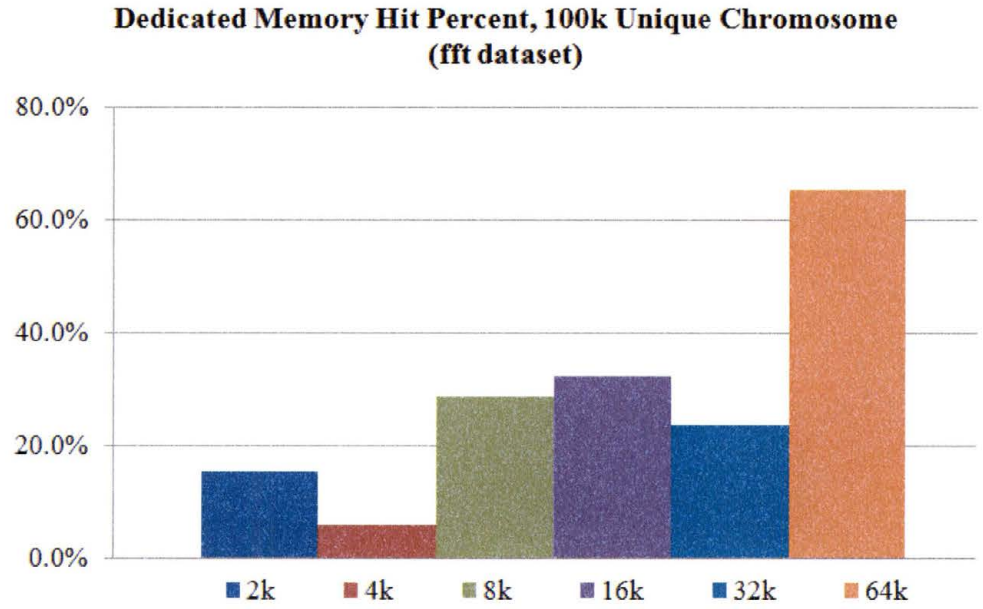

Figure 17. FFT dedicated memory hit percentages

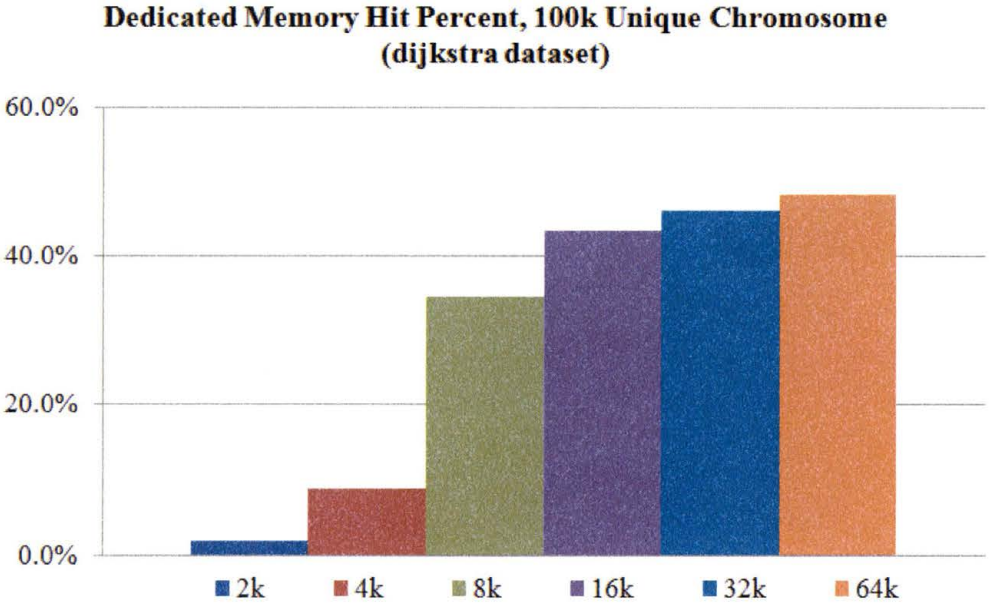

Figure 18. DJK dedicated memory hit percentages

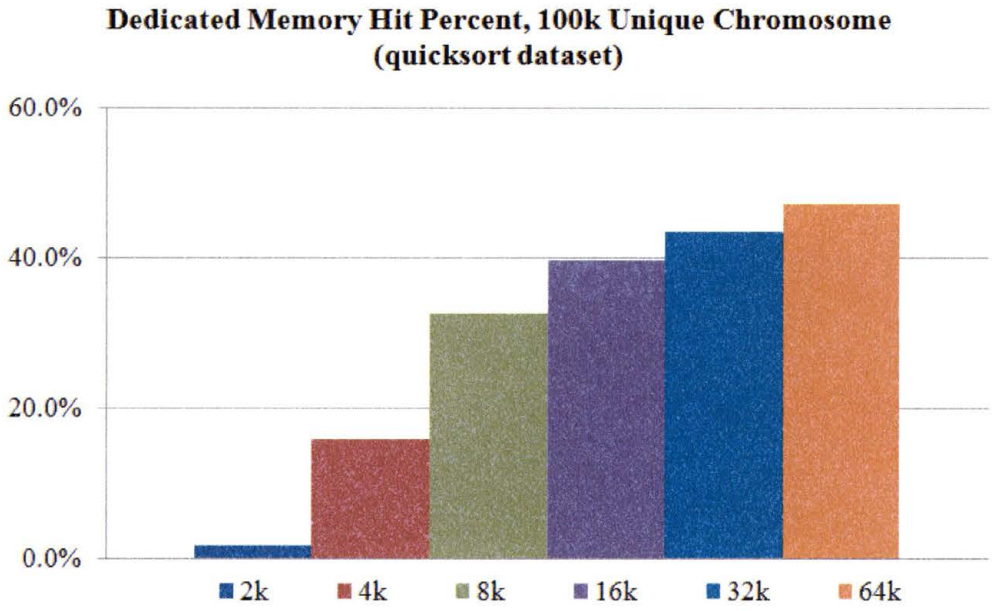

Figure 19 QS dedicated memory hit percentages

The 2k memory size has a very low hit percentage, ranging from an abysmal 1.7% for quicksort, to 15.3% for FFT. The maximum hit percentage at a memory size of 64k is consistent at approximately 48%, with the exception of FFT which has a maximum hit percentage of 65%. This relates to another item; FFT displays erratic behavior over all sizes of memory. The 2k memory size outperforms that of 4k by double and 16k outperforms 32k by half. Shown in figures 18 and 19, other datasets have similar hit percentages and speedup to the results found using BMS.

Dedicated memory provides a theoretic underpinning for performance of recordkeeping sizes for GA. The next step is the implementation of replacement policies with set associative cache.

5. Various Cache Sizes with LFU

This experimental section studies the speedup and the per-generation performance of CGA using different cache sizes and associativities. The CGA here implements the LFU eviction policy. The random replacement policy has less practical relevance and is not used, however, software implementation of LFU is relatively simple when compared to that of LRU, and so is chosen for the first series of experiments with CGA.

The dataset FFT from the baseline GA experimental section generated 66k unique chromosomes, while BMS generated 47k unique chromosomes. These two results form a baseline for the number of unique chromosome generated. Therefore, the termination condition for the GA experiments is the number of unique chromosomes produced in the previous section: 66k for FFT and 47k for BMS. Figure 20 shows the 4 set associative cache, the dataset FFT has limited speed up across different cache sizes.

Interestingly, the smaller cache size of 64k slightly outperforms cache sizes 128k, 256k, and 512k. However, once the cache size increases to 1 million entries the speedup increases dramatically by nearly 70%. The speedup for different cache sizes for 8 and 16 set associativity displays similar speedup performance (figures 21 & 22). The doubling of the cache sizes shows a much smoother transition from one size to the next. This is in contrast to the 4 set associative cache which has erratic performance when doubling of cache size. Interestingly, for 8 and 16 set associativity, when the cache size doubles from 256k to 512k the speedup doubles as well. Whereas, for smaller cache sizes the speedup remains below 50%.

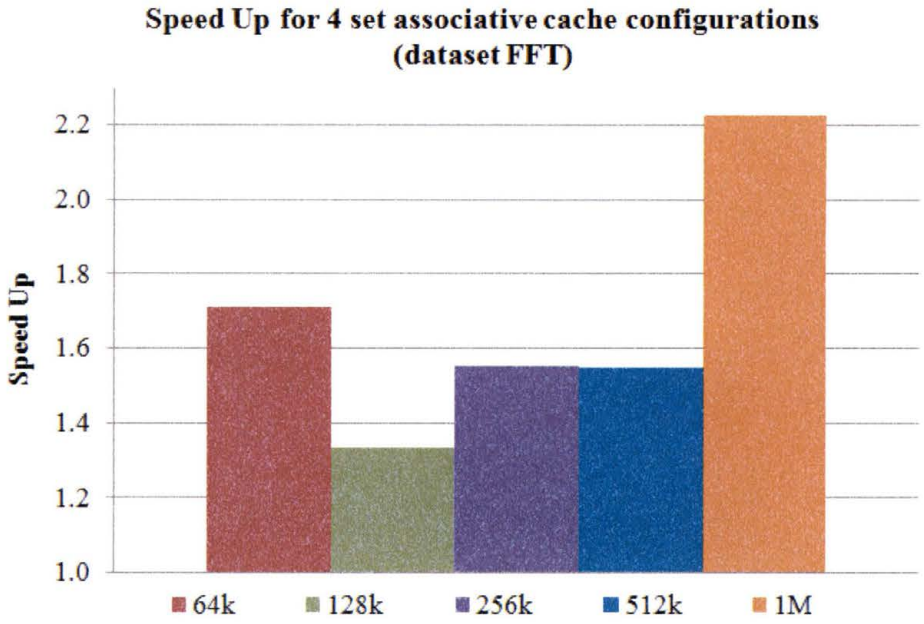

Figure 20. Speedup for FFT 4 set associative cache

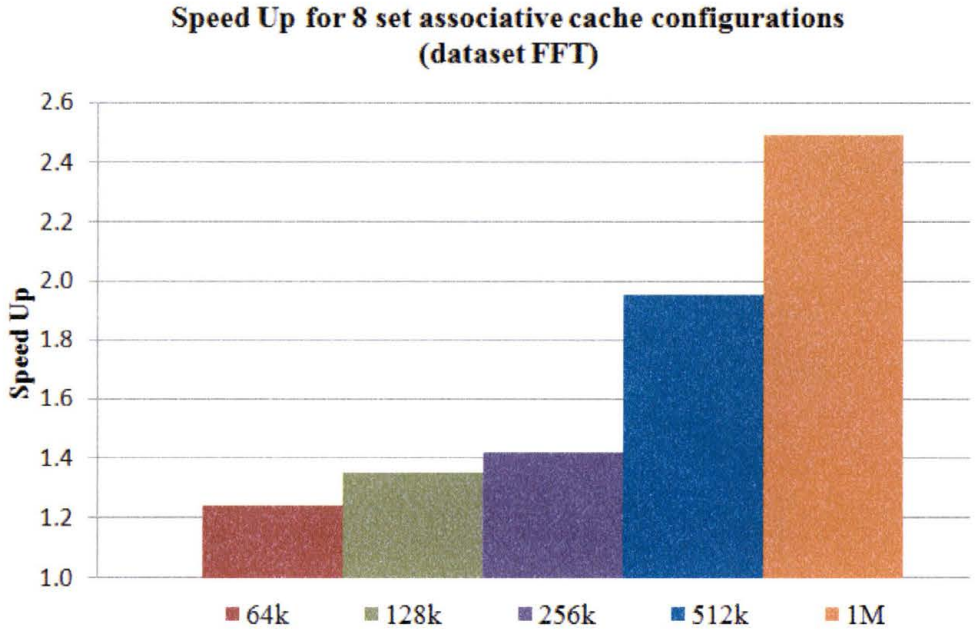

Figure 21. Speedup for FFT 8 set associative cache

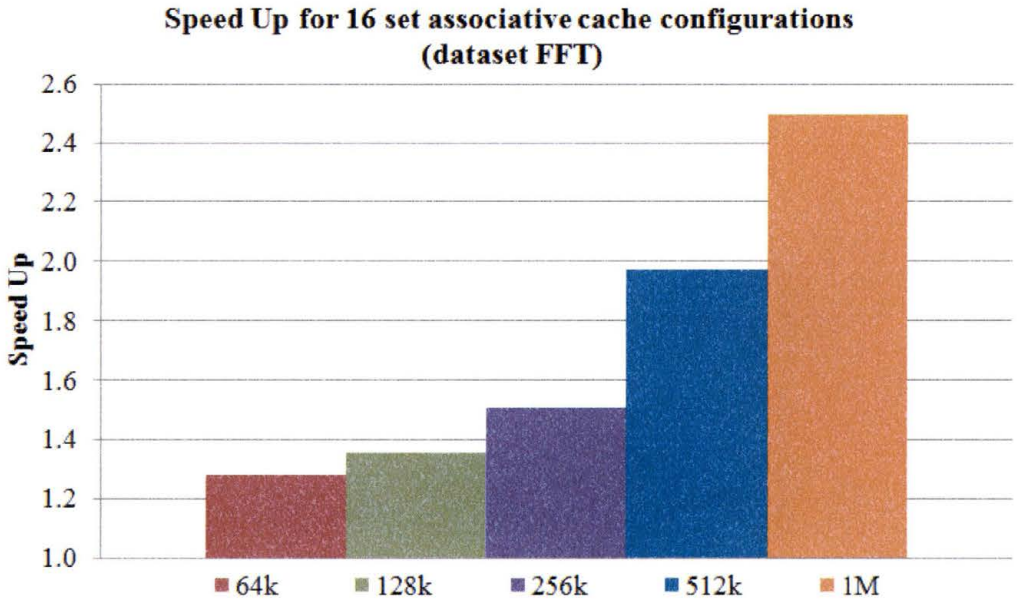

Figure 22. Speedup for FFT 16 set associative cache

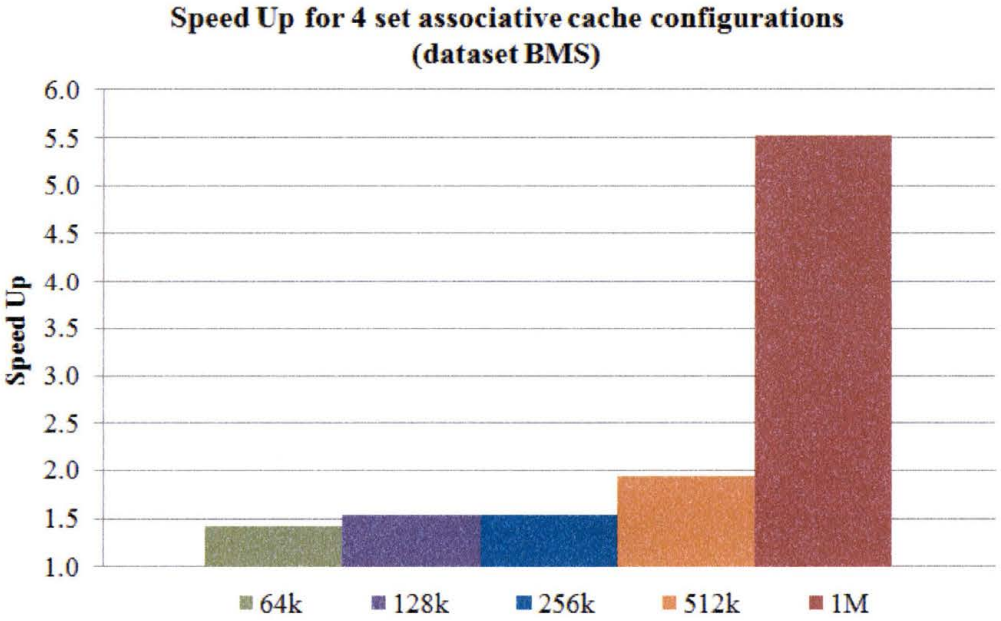

Figure 23. BMS speedup for 4 set associative cache

In the experiments with BMS, a few results stand out. In 4 set associative the speedup is 5.52, in 8 set the speedup is 10.6, and in 16 set the speedup is 15.9. This behavior is not duplicated in the dataset FFT. The second interesting item is the jump in speedup when moving from a cache size of 512k to 1M. The maximum performance for FFT is reached by BMS at half the size of cache. Figure 23 shows the speedup for BMS using 4 set associative cache, 8 and 16 set associative caches show the same trend but are not shown.

The most important observation from the previous results is that dedicated memory of 64k has a higher speedup than CGA for dataset FFT; 3.4 for dedicated memory as opposed to 1.3 for CGA with a cache size of 1M for all associativities studied. For BMS dedicated memory of size 64k outperforms, or does as well as, CGA at size of 512k for all associative sizes.

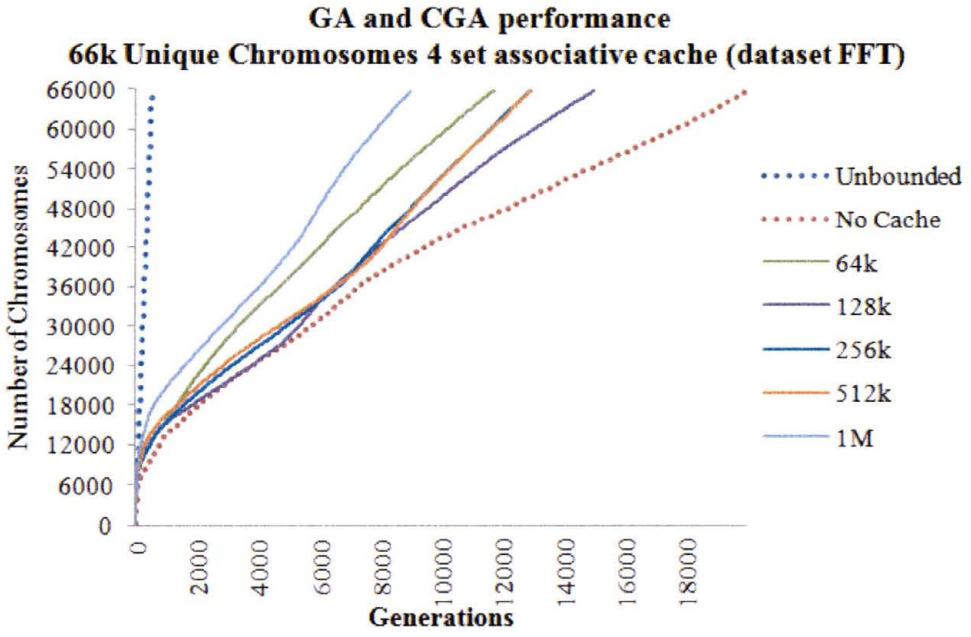

Figure 24. FFT per generation subtotal number of unique chromosomes for 4 set associative cache

Figure 24 shows the per generation performance of different cache sizes and 4 set associativity. The red dotted line in the three figures is GA with no cache (baseline) and is considered the lower bound for per generation performance. The blue dotted line is unbounded memory and is the upper bound on recordkeeping performance. The slope of the plot lines away from the slope of the lower bound indicates speedup.

This slope is shown to vary over generations. Indeed, an interesting observation is that there are jumps in performance. This behavior leads to another interesting observation, the performance curve from two or more cache sizes may cross multiple times. One can infer from this that, depending on the dataset, there are ranges of generations where a smaller cache size outperforms a larger cache size. This can impact the choice that a programmer may make regarding the value of a small cache over a large cache for an application of CGA using a particular generation limit.

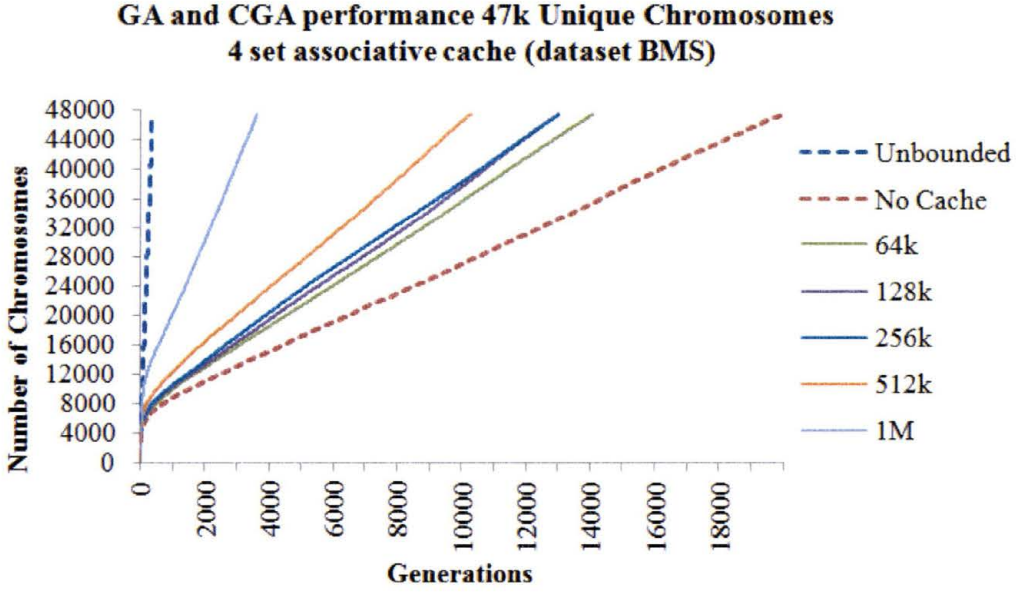

Figure 25. BMS per generation subtotal number of unique chromosomes with 4 set associative cache

Figures 25 shows the same type of graph for the dataset BMS. In this case the GA baseline produced 47k unique chromosomes over 20,000 generations. The results for BMS are similar to those of FFT over different cache sizes.

6. Cache Replacements Policies

The results from the experiments using the replacement policies LRU, LFU, and random, on dataset FFT, are shown in figures 26 and 27. The first item which is apparent is that the speedup of FFT is restricted to less than 2x for cache size of less than 512k regardless of replacement policy. The hit percentage for FFT is within 10% for both 256k and 512k cache sizes of 8 set associative cache. At 256k cache size figure 26 shows that LRU is the best performer, but only by 10% over LFU and random replacement policies. At cache size of 512k for 8 set associative, FFT shows even speedup between LFU and random replacement policies.

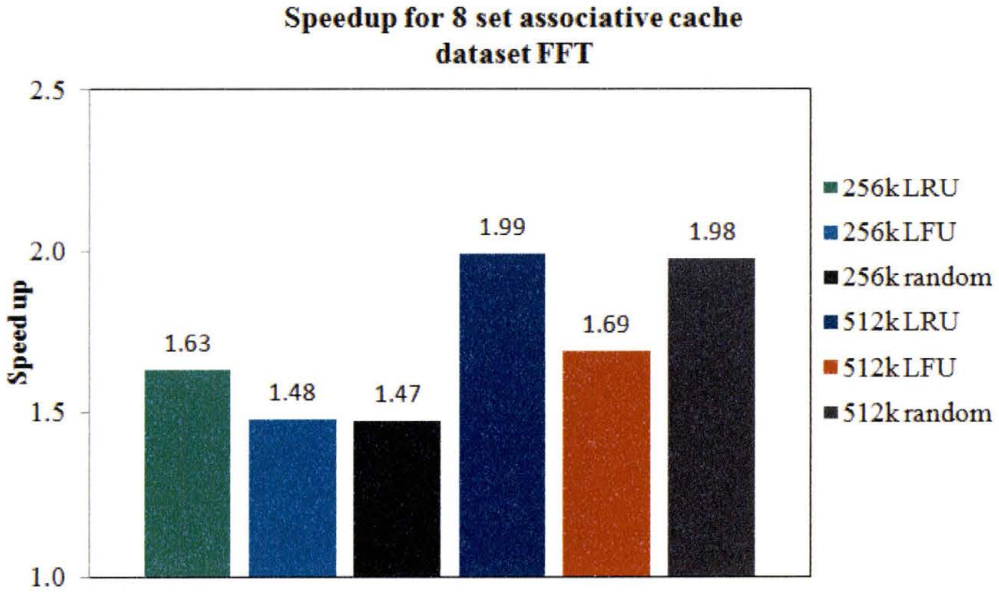

Figure 26. FFT speedup for different replacement policies using 8 set associative cache and 256k plus 512k cache sizes

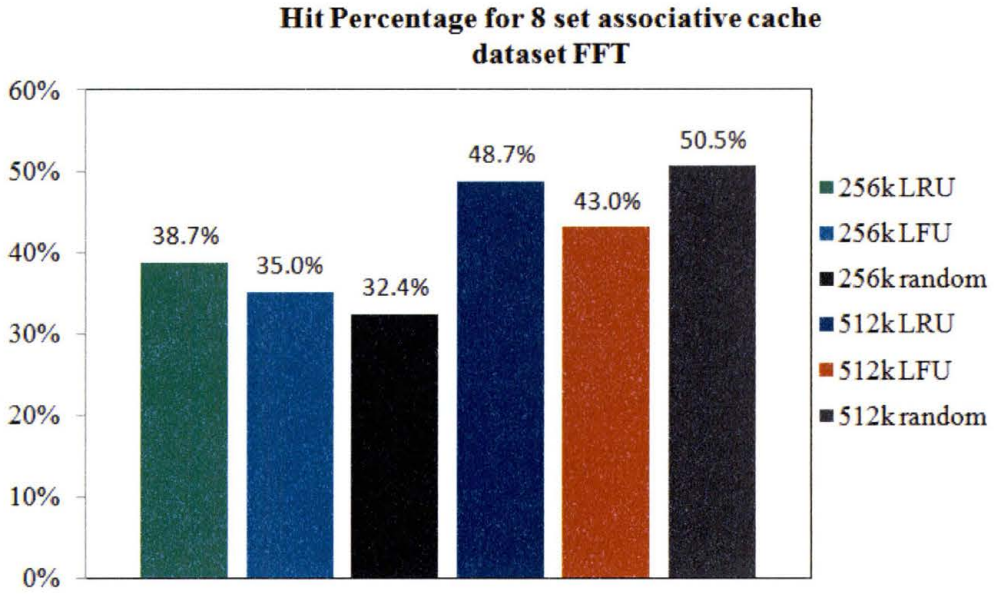

Figure 27. FFT hit percentages for different replacement policies using 8 set associative cache and 256k plus 512k cache sizes

Figure 28 shows LFU underperforms for the larger cache size of 1M when compared to LRU and random. This is consistent with the results of the smaller cache sizes. Doubling cache size, LRU and random more than double their speedup, while LFU does not. Despite the larger associatively in figure 23, the 512k cache performs almost identically to the 8 set associative cache in figure 21. This is perhaps an indication that associativity plays a smaller role in speedup of CGA than cache size or replacement policy.

Both figures 28 and 29 show hit percentages which are closely correlated to speedup. This may indicate that a good choice of replacement policy is essential to optimal CGA performance.

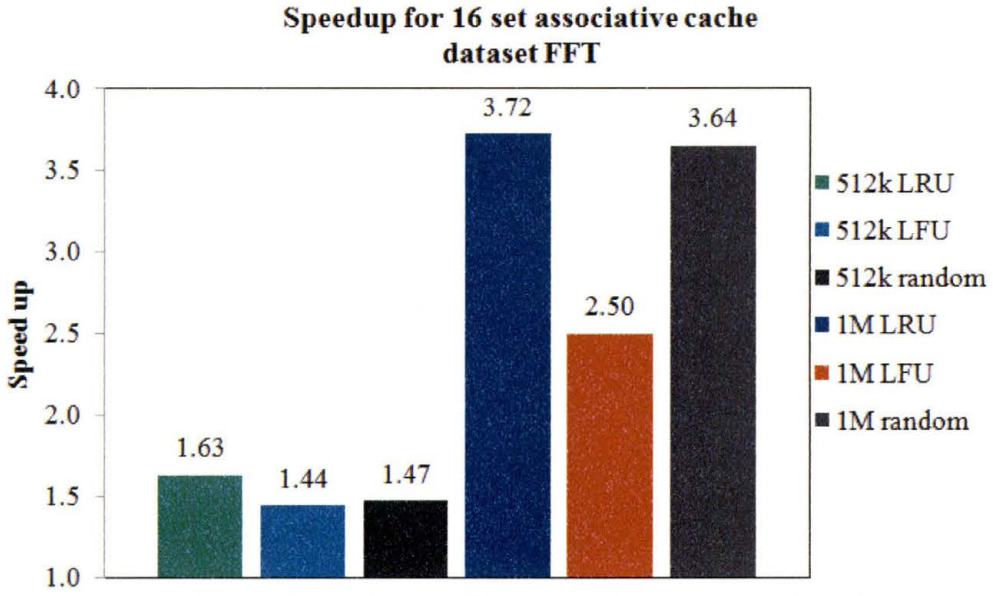

Figure 28. FFT speedup for different replacement policies using 16 set associative cache and 512k plus 1M cache sizes

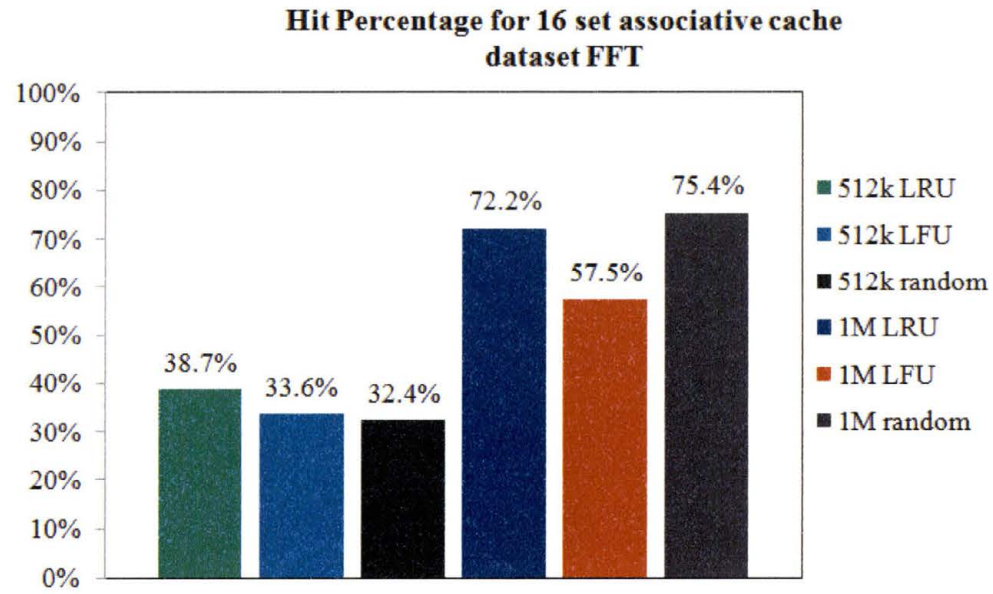

Figure 29. FFT hit percentages for different replacement policies using 16 set associative cache and 512k plus 1Mcache sizes

The next series of figures show experimental results from the same experiments run on FFT using the BMS dataset with 8 set associative cache. Figure 30 shows a speedup similar to FFT in that it remains below 2x. However, in this case the random replacement policy is superior. This is true not only for a cache size of 256k, but for the cache size on 512k as well. The increase is approximately 14% for 256k cache size, and 8.5% for 512k cache size.

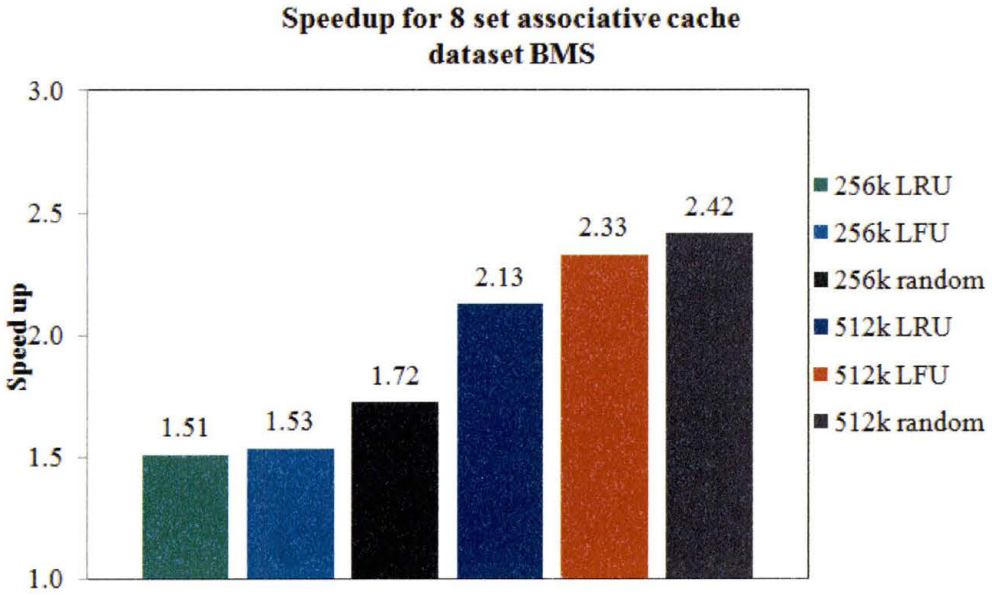

Figure 30. BMS speedup for different replacement policies using 8 set associative cache and cache sizes of 256k plus 512k

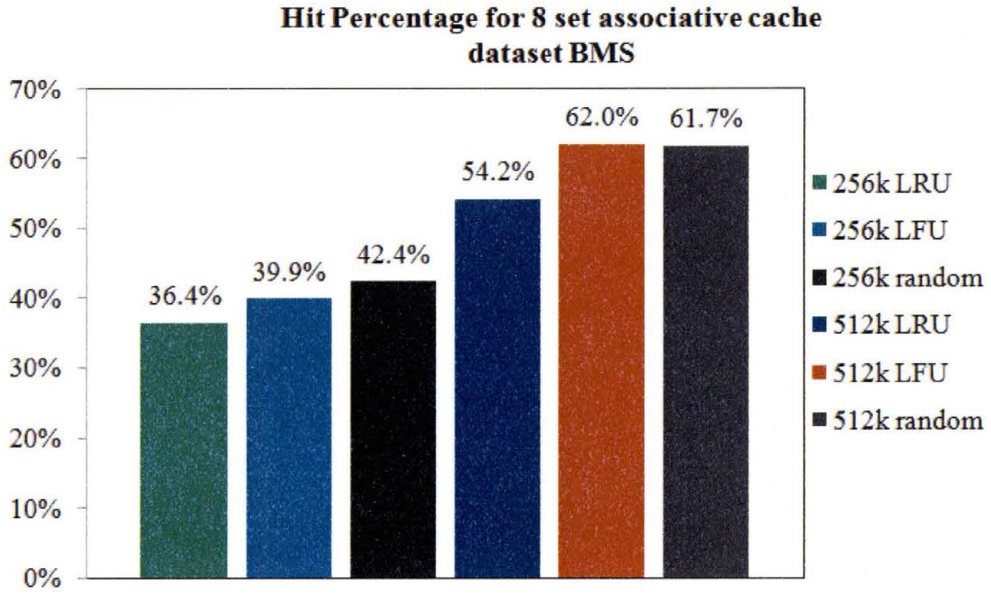

Figure 31. BMS hit percentages for different replacement policies using 8 set associative cache and cache sizes of 256k plus 512k

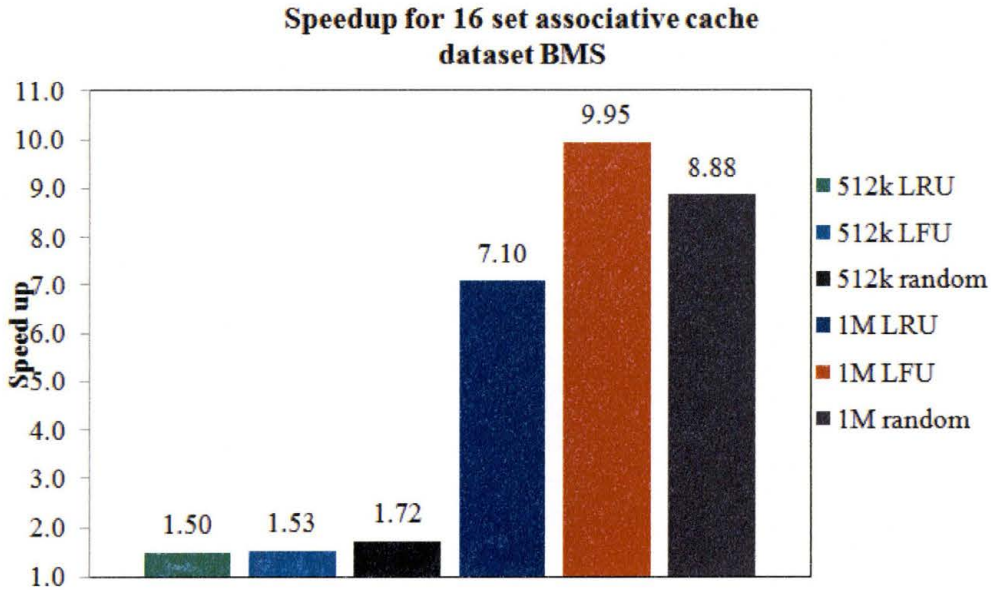

Figure 32. BMS speedup for different replacement policies using 16 set associative cache and cache sizes of 512k plus 1M

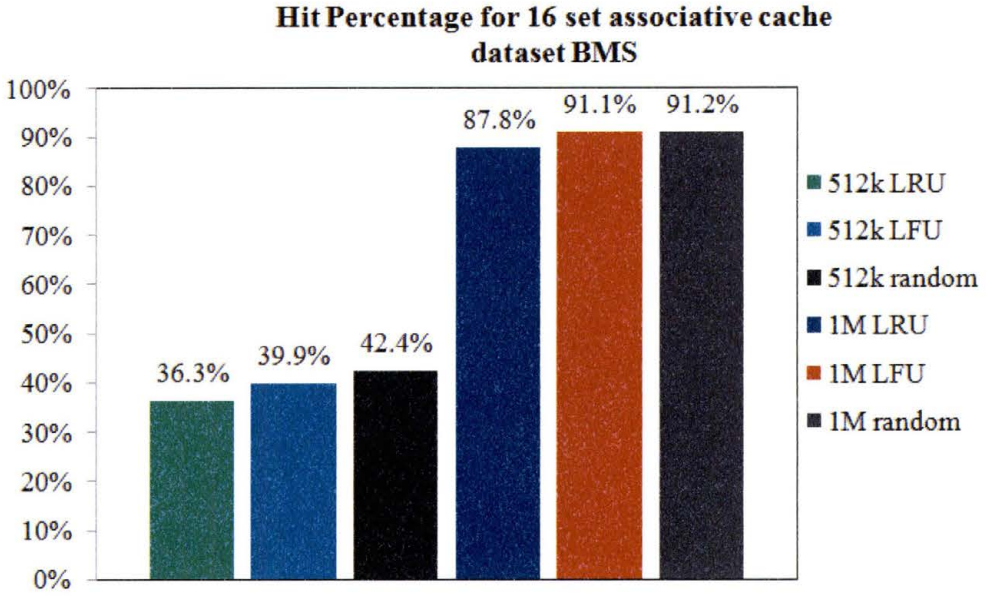

Figure 33. BMS hit percentages for different replacement policies using 16 set associative cache and cache sizes of 512k plus 1M

Figure 32 shows that at a cache size of 1M the results for 16 set associative cache using BMS are inconsistent with the results in the previous figures. In addition, the figure shows that by doubling the associativity, there is a drop in speedup. This observation is counter to that observed with FFT, which suggests that replacement policy performance is data dependent.

While the hit percentage for LFU in figure 33 and random replacement policies are closely matched, the speed up for LFU is approximately 12% higher than random, and 40% higher than LRU. Hit percentage in figure 28 shows for 1M cache size is approximately 90% across all replacement policies. This result is not surprising as the search space for these series of experiments using FFT and BMS datasets is and the cache size is 1M, a theoretical coverage of 73%.

Figure 34 shows the hit ratio for Dijkstra and quicksort datasets' experiments which use different replacement policies. All replacement policies have a hit percentage less than 50%. This can perhaps be explained by the larger feature search space.

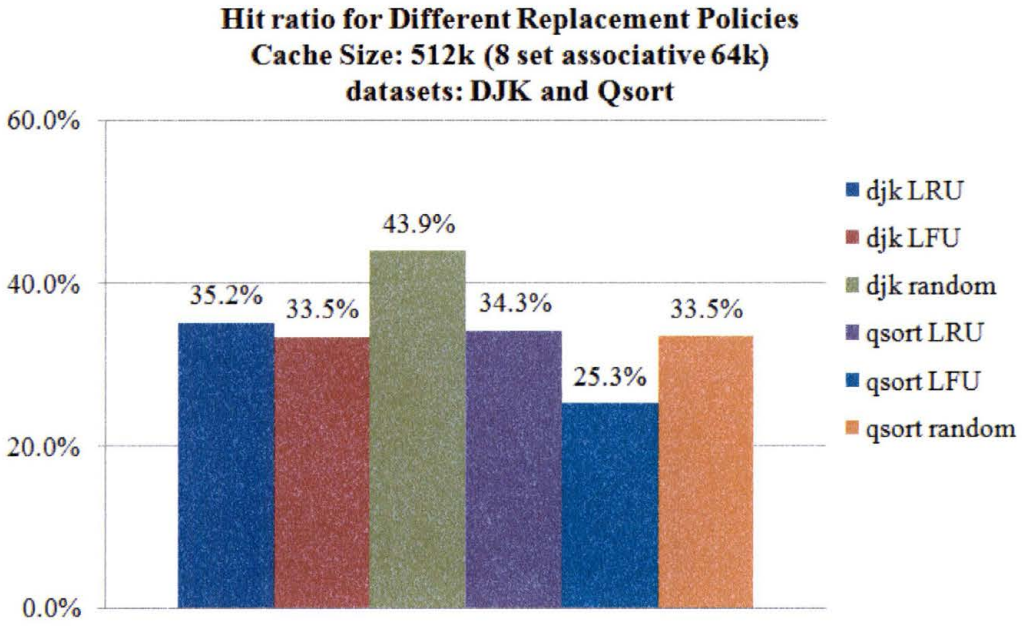

Figure 34. DJK and QS hit percentages for different replacement policies using 8 set associative cache and cache sizes of 512k plus 1M

Using the Dijkstra dataset, LFU performs better than LRU for cache size of 512k, but not better than the random replacement policy. The results in this section show that dedicated memory outperforms CGA in every regime. Dedicated memory, of only 64k, has a higher speedup than a cache size of 512k. This results is true for all datasets, associativities, and replacement policies. This raises a question to whether or not there is an issue with cache utilization.

7. Cache Utilization

Figures 35 and 36 show the cache utilization for different configurations with the dataset Dijkstra. Utilization here is defined as the ratio of the number of occupied cache blocks to the number of blocks in the cache. In figure 35, the cache size is fixed, while the set associativity varies from 4, to 8, to 16 running with an LRU replacement policy.

In all experiments from section 6 and 7 of this chapter CGA produces 100k unique chromosomes and then exits. The upper limit on utilization is the number of unique chromosomes generated divided by the size of the cache. For a 512k cache size, the upper limit is 20%, while for a 1M cache size it is 10%.

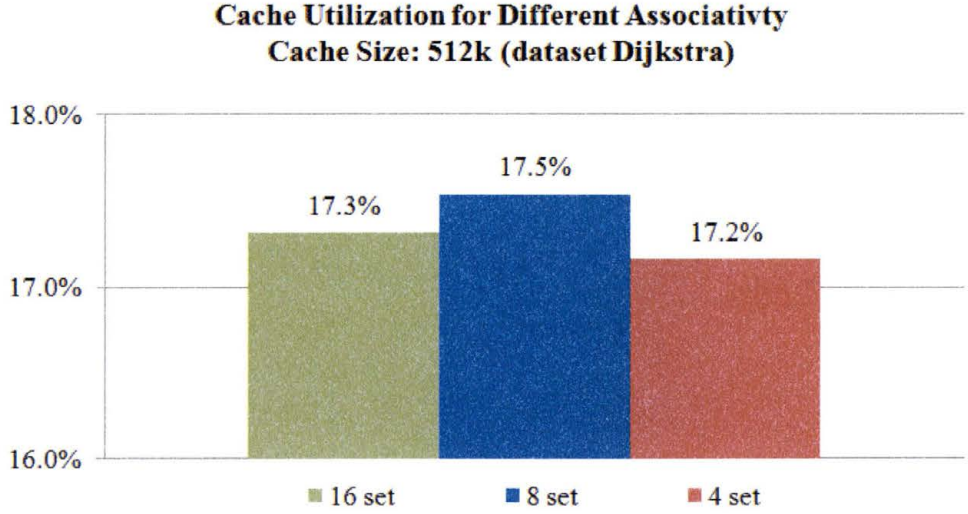

Figure 35. DJK cache utilization for 4, 8, and 16 set associativities with LRU using 512k cache size

The results from figure 35 show an average utilization across set associativity sizes of 17.4%, close to the upper limit. There is little affect on utilization of cache when varying set associativity.

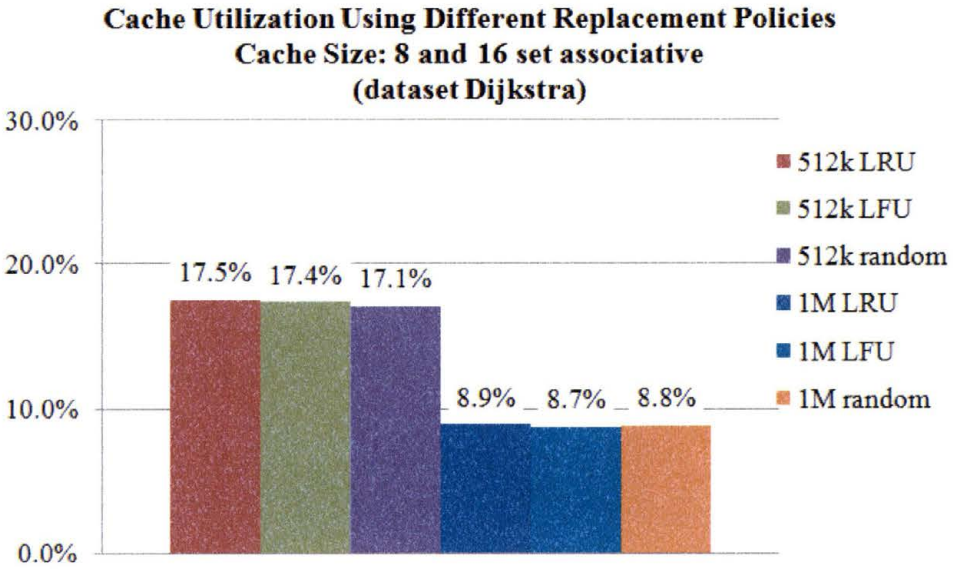

Figure 36. DJK cache utilization for different replacement polices using 512k cache size and 8 set plus 16 set associativities

Figure 36 shows cache utilization versus different replacement policies and two cache sizes. There is no difference in cache utilization when it comes to different replacement policies. For cache size of 1M the cache utilization across replacement policies is close to 9%.

As mentioned in the methodology chapter, to restricting the number of the features selected does have a small effect of creating address holes in certain cache sizes. These holes are portions of a cache which cannot be addressed, and thus remain vacant. However, the number of non-addressable blocks is small. In the above figures' block addresses 0.82%.

8. Quality Check

Inspecting the trend of ISODATA clustering quality at early generations shows that CGA and GA converge within 50 generations to near maximum quality. Figure 37 shows the top quality out of all chromosomes for a given generation for the dataset Dijkstra. The jumps in steps indicate that a new top quality set of sub-features have been found. The plateaus indicate that despite continuing search, no new best quality has been located in the search space.

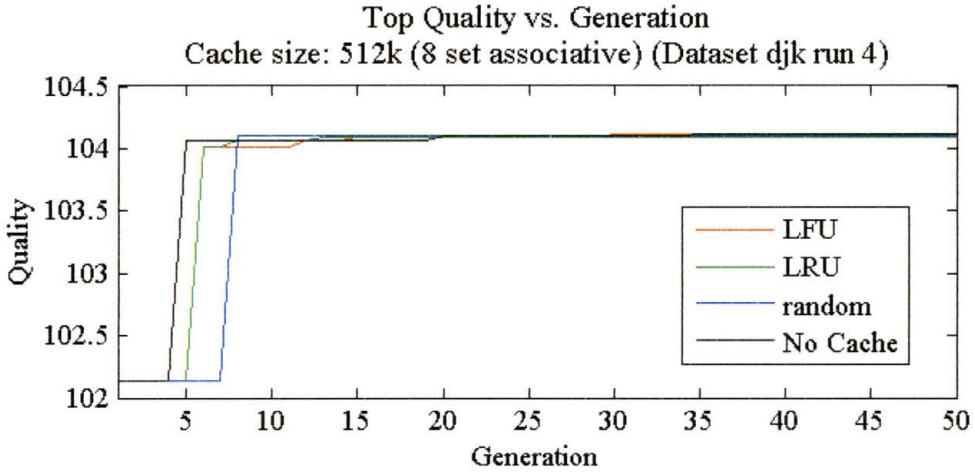

Figure 37. DJK best quality for different replacement policies

Figure 37 shows CGA replacement policies do not affect the convergence, and also, GA showed convergence in quality as quickly as CGA. The results for the other Freescale Semiconductor datasets reveal similar behaviors for all random number generation seeds. To check if the reason for the fast convergence is due to data dependence, quality histograms are produced for the Freescale Semiconductor datasets.

9. Quality Histogram

The exhaustive quality search is performed on all four Freescale Semiconductor datasets. The combinations chosen are those of the previous experiments; i.e., for BMS and FFT, $\binom{23}{12}$, for Dijkstra and quicksort $\binom{25}{13}$. The histograms in figures 38 through 39 have bin sizes of 0.25.

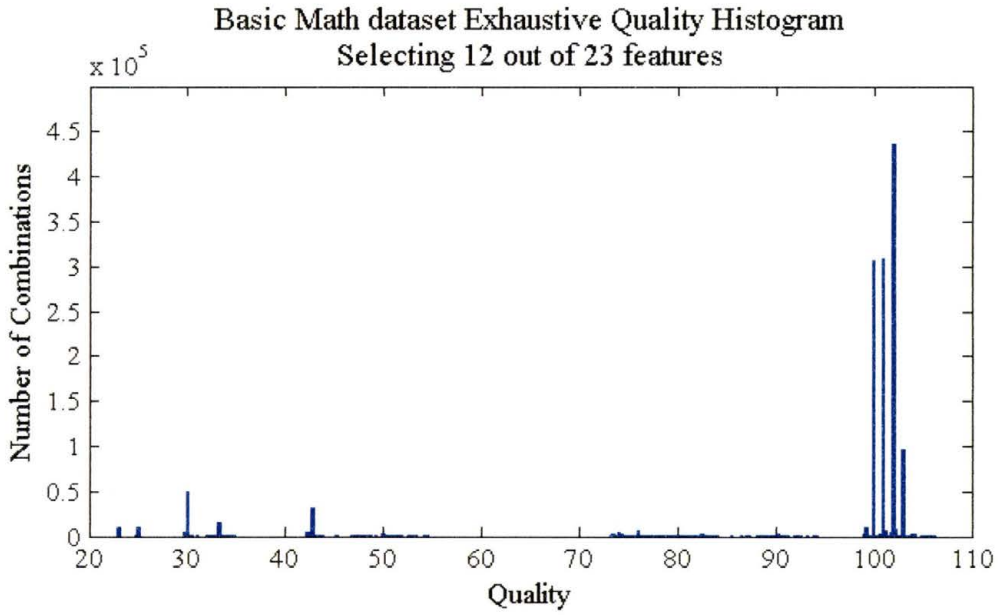

Figure 38. BMS quality histogram for 23 choosing 12 features

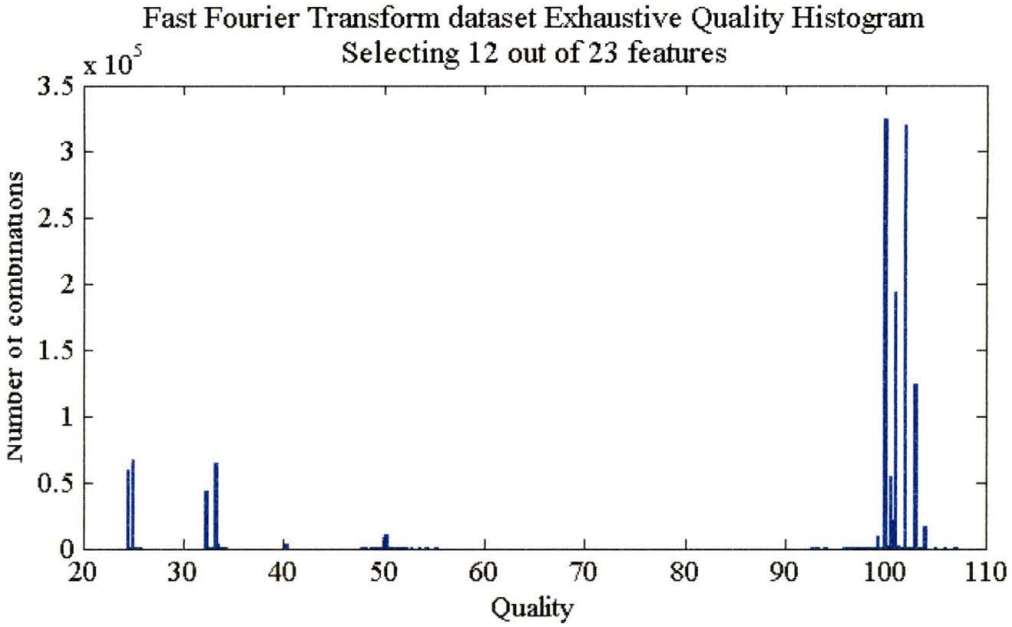

Figure 39. FFT quality histogram for 23 choosing 12 features

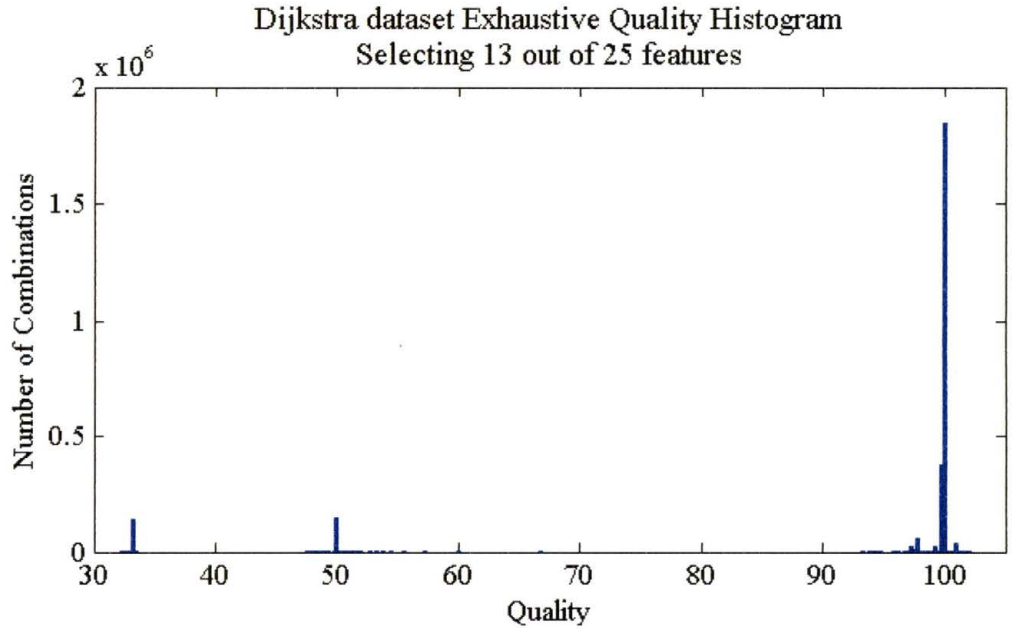

Figure 40. DJK quality histogram for 25 choosing 13 features

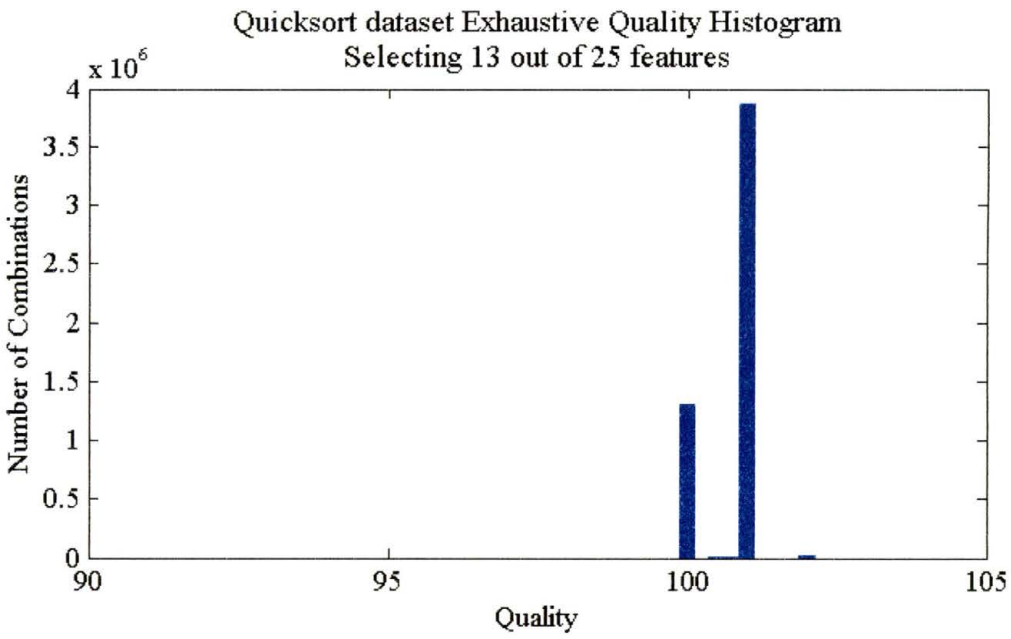

Figure 41. QS quality histogram for 25 choosing 13 features

From these figures it is clear that many of the combinations of features have similar qualities which are grouped tightly, and those groups are few. Dijkstra and quicksort dataset in particular have groupings of quality value which not only are narrow, but also exist within 10% of the maximum quality. For Dijkstra data set the largest bin contains 1.7×10^6 combinations, or 34% of the total combinations. For the quicksort dataset the situation is more extreme with approximately 98% of all combinations with a quality existing in two bins of size 0.25, all of which are within 10% of the maximum quality.

The results of these histograms gives rise to the speculation that the narrowness of the quality distribution for the Freescale Semiconductor datasets contributes to the quick convergence of the quality in CGA and GA. To test this hypothesis two synthetic datasets are generated with flatter quality histograms, and tested in the same fashion as the Freescale Semiconductor datasets.

Shown in figures 42 and 43 are the histograms for the two synthetic datasets, named synthetic 8 (S8), and synthetic 11 (S11). The histograms shows the quality for an exhaustive search of the combinatorial space for the selection of 13 dimensions out of 25, and have bin sizes of 0.25. Like the Dijkstra dataset, S8 and S11 have 1000 data points. The similarity in parameters makes the Dijkstra dataset a good control to S8 and S11 experiments. Therefore, the Dijkstra dataset is used to compare the performance of the synthetic datasets.

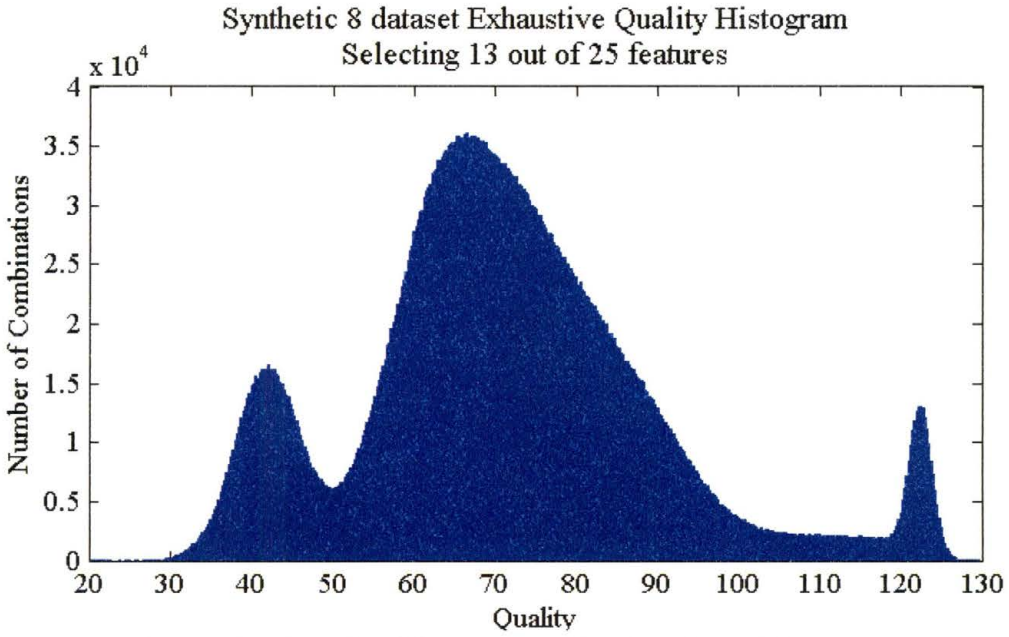

Figure 42. S8 quality histogram for 25 choosing 13 features

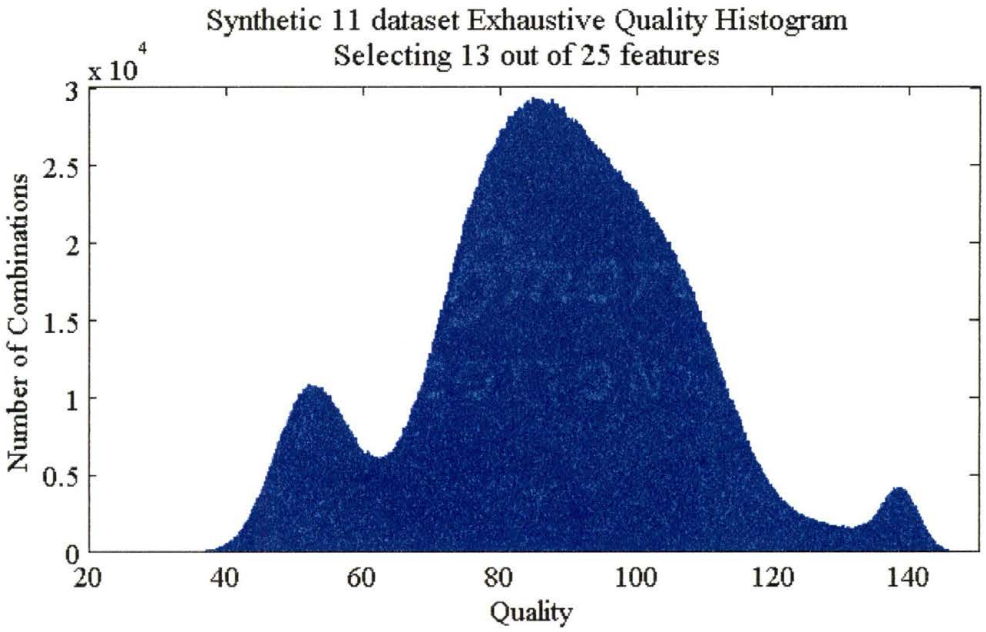

Figure 43. S11 quality histogram for 25 choosing 13 features

The criteria for selecting these sets is to have the same number of dimensions (features) as the larger of the Freescale Semiconductor datasets (25 dimensions). S8 and S11 show a large number of combinations located at the center of their histograms, and have additional smooth peaks located at each end of the histogram. At the low end of the quality scale, for both datasets, the secondary peak is wider and has a greater number of combinations than the secondary peak located at the higher end of the scale. This along with the primary peak distribution suggests that GA and CGA will spend more time away from the high quality combinations. Additionally, the maximum number of combinations in a bin is two orders of magnitude less than those in both the Dijkstra and quicksort histograms.

10. Floating Feature Subset Number

The first check of the synthetic data, using CGA, is to isolate the effects of the restriction that the GA must produce children with exactly a preset number of selected features. For S8 and S11, with 25 dimensions, CGA is allowed to choose any number of features. This means the search space is now 32M combinations in size. These sets of experiments are using the LRU replacement policy and CGA terminates once 200k unique chromosomes are produced.

Figures 44 and 45 show the hit percentages arranged from high to low cache sizes. In figure 44, the cache hit percentages are at, or below, 50%. This result is similar to the result found for the Dijkstra dataset experiment. Figure 45 shows the experiment for the Dijkstra dataset showing a pattern similar to S11, however the difference between cache sizes is more pronounced. The full features space exhibits hit percentages below 50% for three cache sizes. A similar result is found in the fixed subset number experiments.

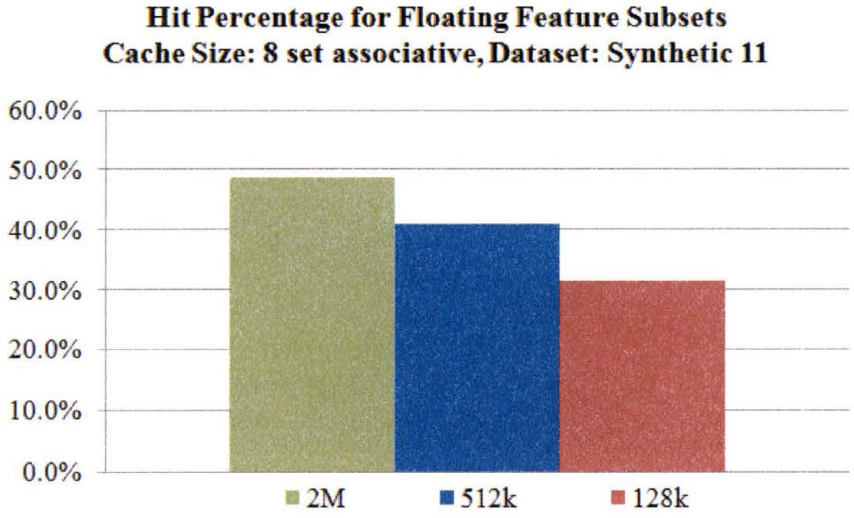

Figure 44. S11 hit percentage for 8 set associative cache and sizes 128k, 512k, and 2M

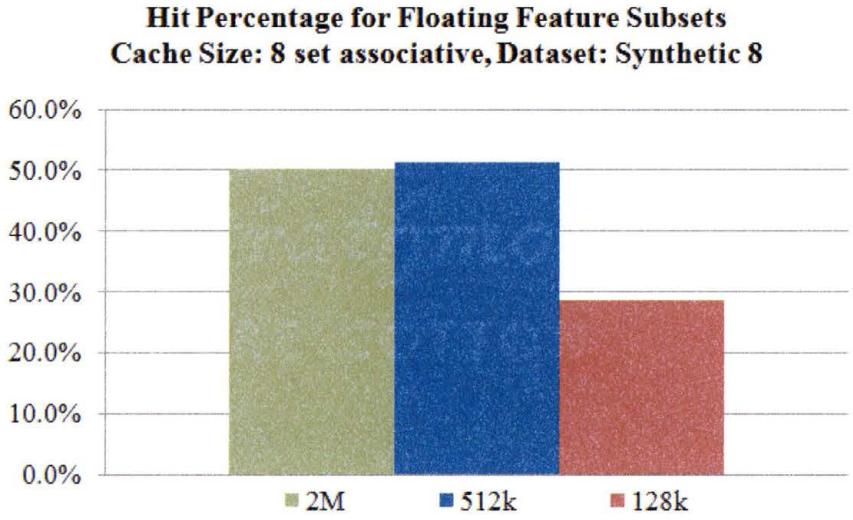

Figure 45. S8 hit percentage for 8 set associative cache and sizes 128k, 512k, and 2M

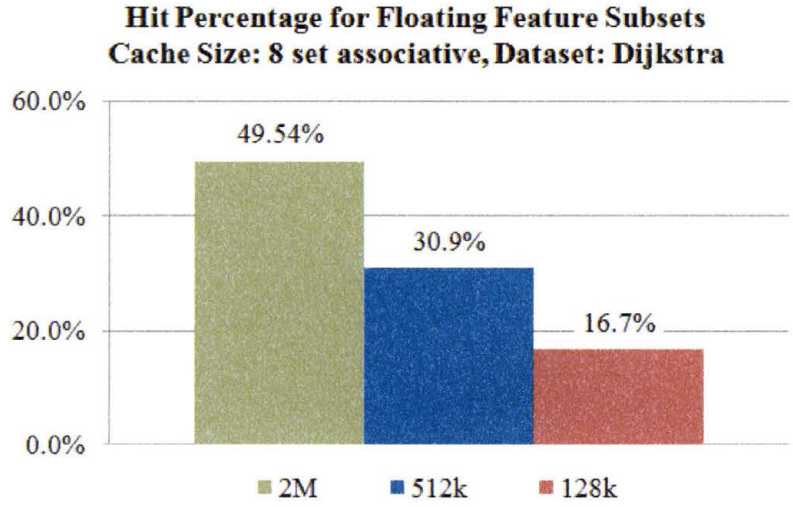

Figure 46. DJK hit percentage for 8 set associative cache and sizes 128k, 512k, and 2M

Figures 47 through 49 show the cache utilization for this series of experiments. Both synthetic datasets and the Dijkstra dataset, the cache utilization is high. For a 2M cache size, with 200k unique chromosomes generated, the upper bound on utilization is 10%, for 128k it is 100%.

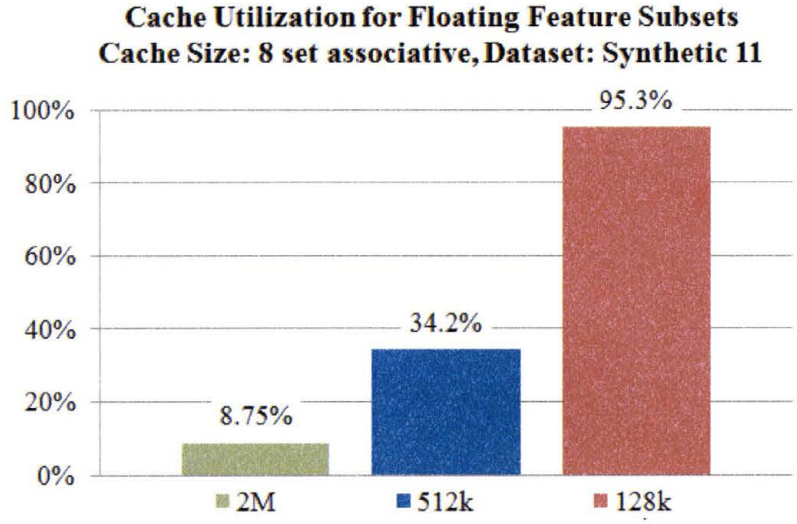

Figure 47. S11 cache utilization for 8 set associative cache and sizes 128k, 512k, and 2M

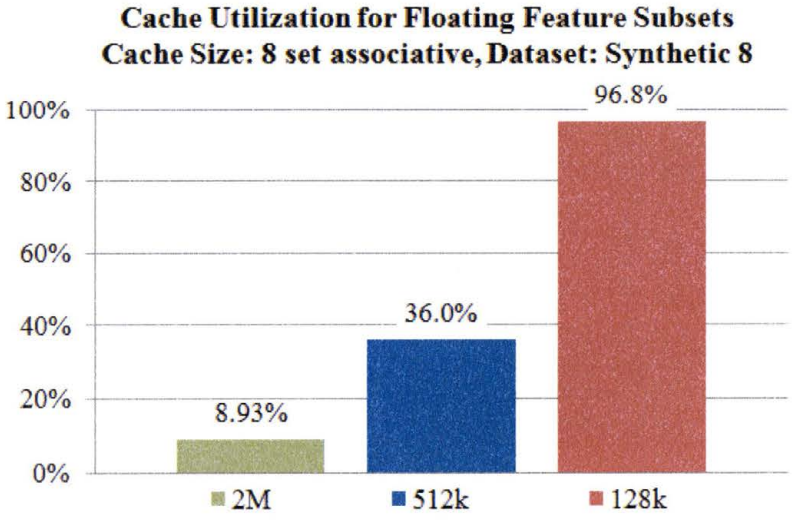

Figure 48. S8 cache utilization for 8 set associative cache and sizes 128k, 512k, and 2M

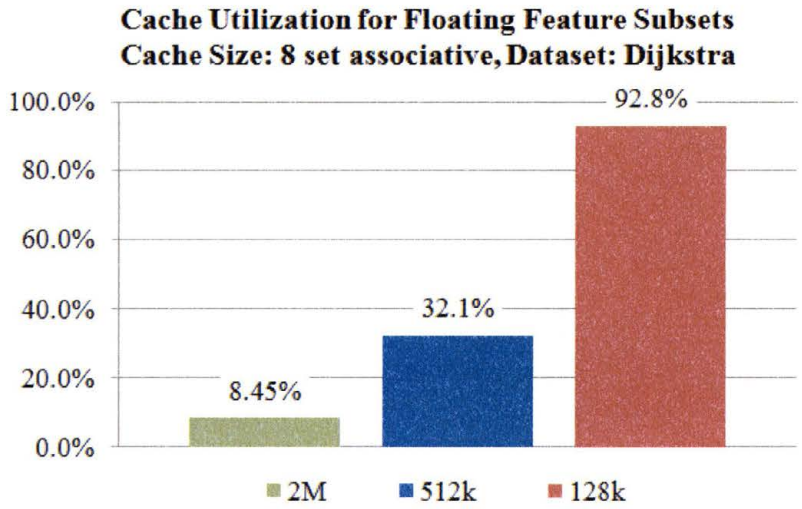

Figure 49. DJK cache utilization for 8 set associative cache and sizes 128k, 512k, and 2M

The same utilization percentage is found, in figures 47 through 49, using a fixed feature subset number. This result shows cache utilization is good while searching both the full subset feature space as well as the fixed subset number search space, for DJK, S11, and S8.

11. Replacement Policies for Synthetic Datasets

These next series of experiments looks at speedup and the hit percentage for S8 and S11 using different cache replacement policies. Cache size is fixed at 2M with 8 way set asssociativity. The results are averaged over 4 replicates and consists of data collected after running CGA with the termination condition of 50k unique chromosomes. Figure 50 shows the speedup for different replacement policies for the dataset S11.

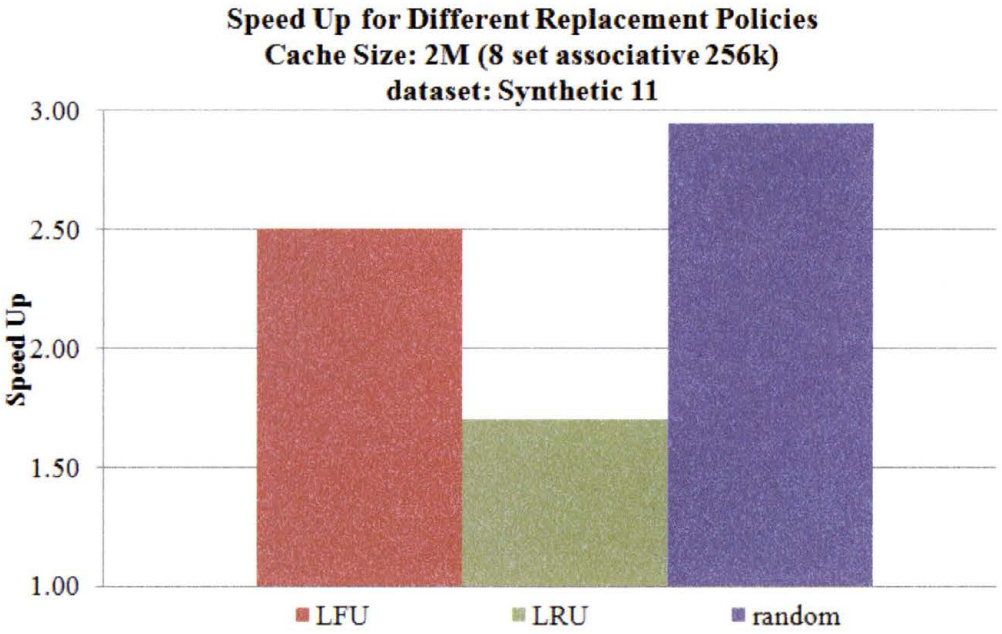

Figure 50. S11 speedup for different replacement policies with 256k cache size and 8 set associativity

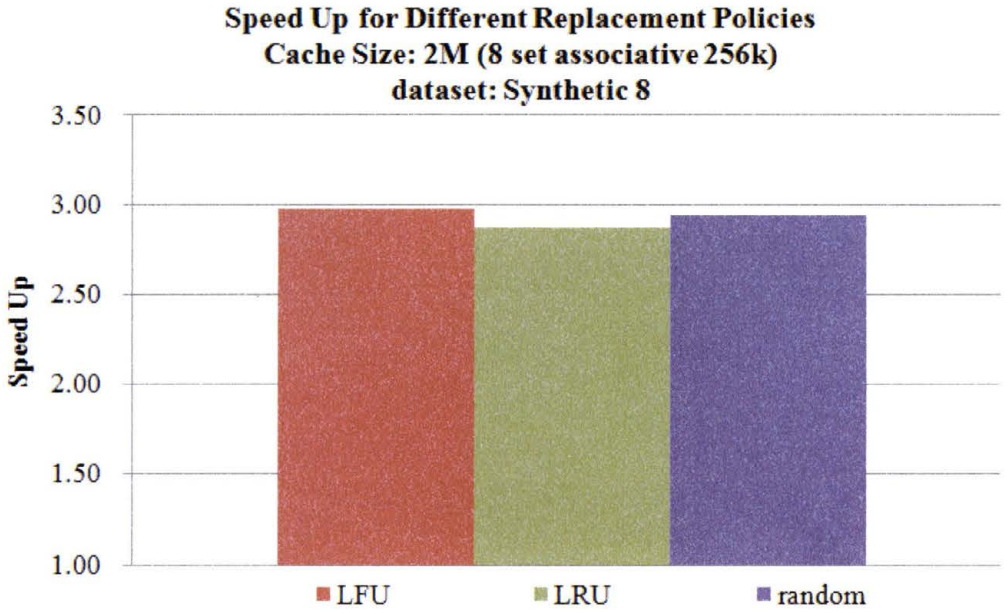

Figure 51. S8 speedup for different replacement policies with 256k cache size and 8 set associativity

The figure shows that random replacement policy outperforms LFU, and outperforms LRU by nearly 2x. This result contrasts with figure 51 using S8, where all three replacement policies performed equally. This suggest a strong correlation between the dataset and CGA speedup using different replacement policies.

12. Quality Convergence for Synthetic Data

The initial purpose for developing synthetic data is to determine if fast convergence to the global optimum seen by the studies with Freescale Semiconductor dataset is an artifact of their quality value distributions. In figures 52 and 53 the single data runs of cache sizes 512k for dataset S11 and 2M for dataset S8. As in figure 37, each step indicates a new best overall quality has been located, while the plateau indicated the best quality remains unchallenged.

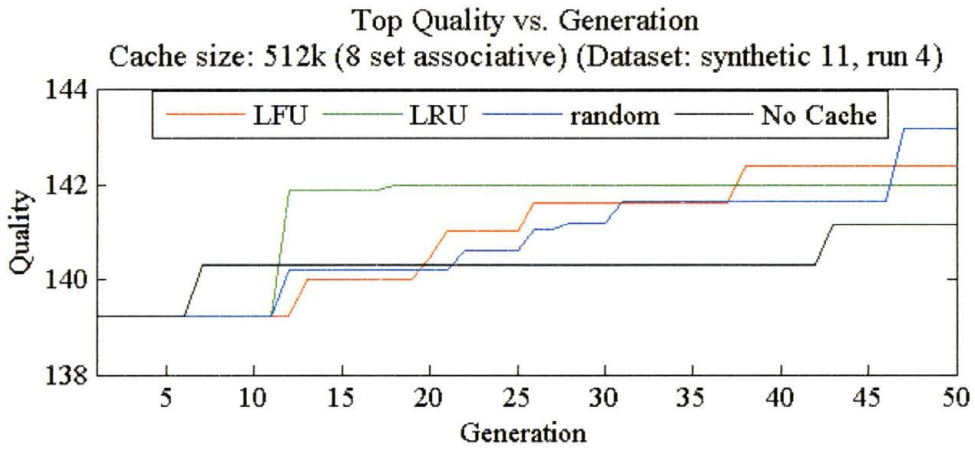

Figure 52. S11 best chromosome quality per generation using different replacement policies and 512k cache size

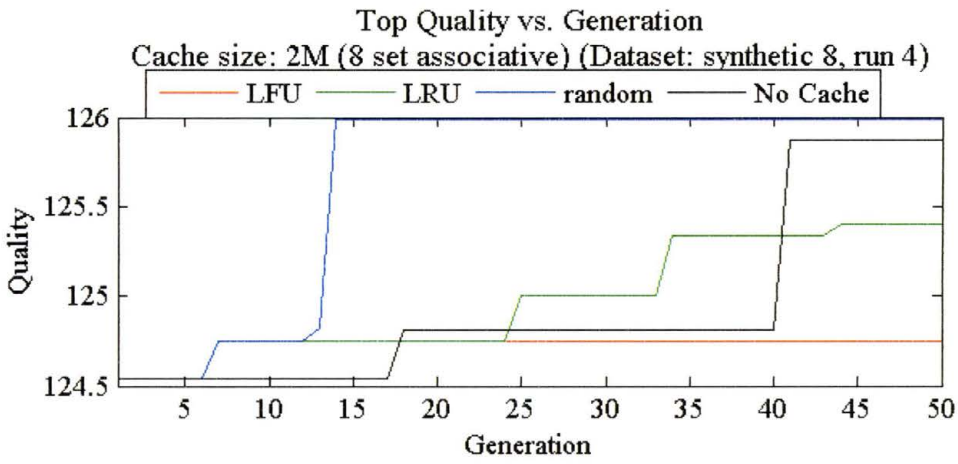

Figure 53. S8 best chromosome quality per generation using different replacement policies and 2M cache size

Like the results from the Freescale Semiconductor datasets, the synthetic dataset quickly converges to within 10% of optimum. Like the quality figure 37, the initial starting points for the top quality are relatively high. This result is not exclusive to the random seed of experiment run 4, but a result that exists in all experimental runs. Runs 1-3 are not shown in this section.

13. Study of Fully Associative Cache

An additional cache model is introduced in this section to address possible problems found with the set associative cache implemented with CGA. The fully associative cache is implement at various sizes, using only the LRU replacement policy. One drawback of fully associative cache is that it is expensive to implement in hardware and so it is rarely used (Hennesy & Patterson, 2007). In software, there is a cost to searching the list of entries, however this cost is minimal when compared to the run time of ISODATA. The results in this section are averaged over 4 runs. Figures 54, 56, 58

and 59 show fully associative cache using sizes doubling from 2k to 128k. In figures 55 and 57 show the dedicated memory experimental results for the Dijkstra dataset as a comparison.

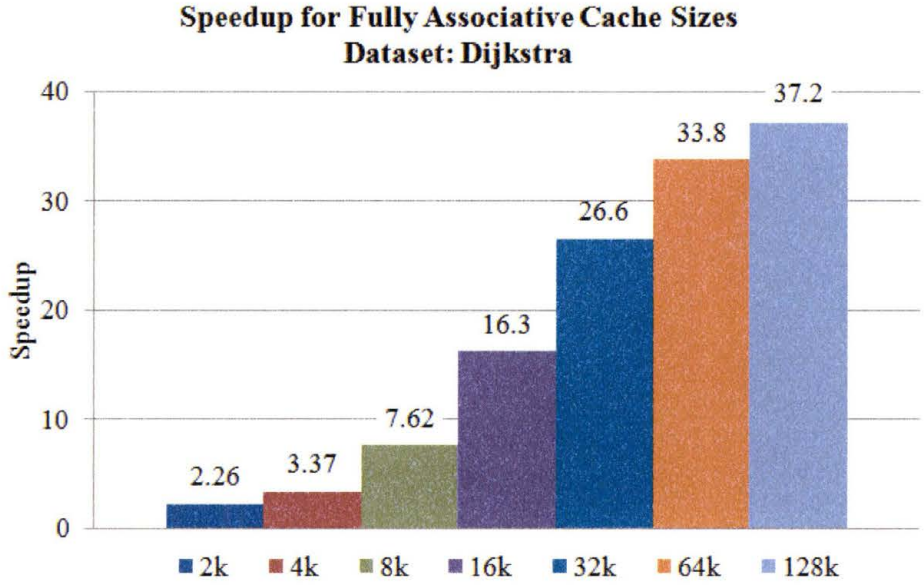

Figure 54. DJK speedup for fully associative cache of various sizes

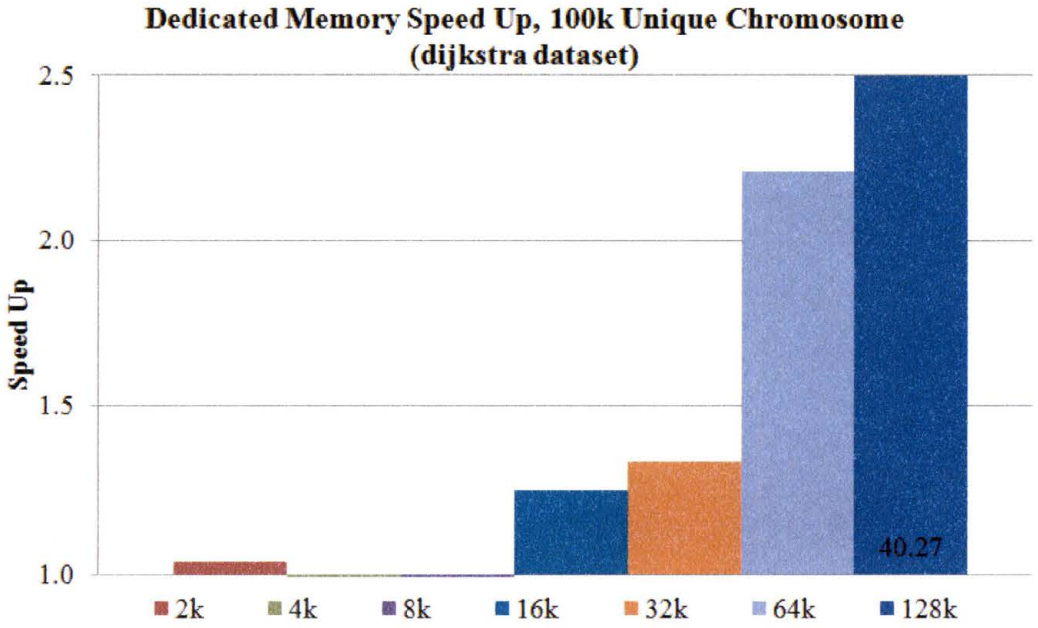

Figure 55. DJK dedicated memory speedup for various sizes

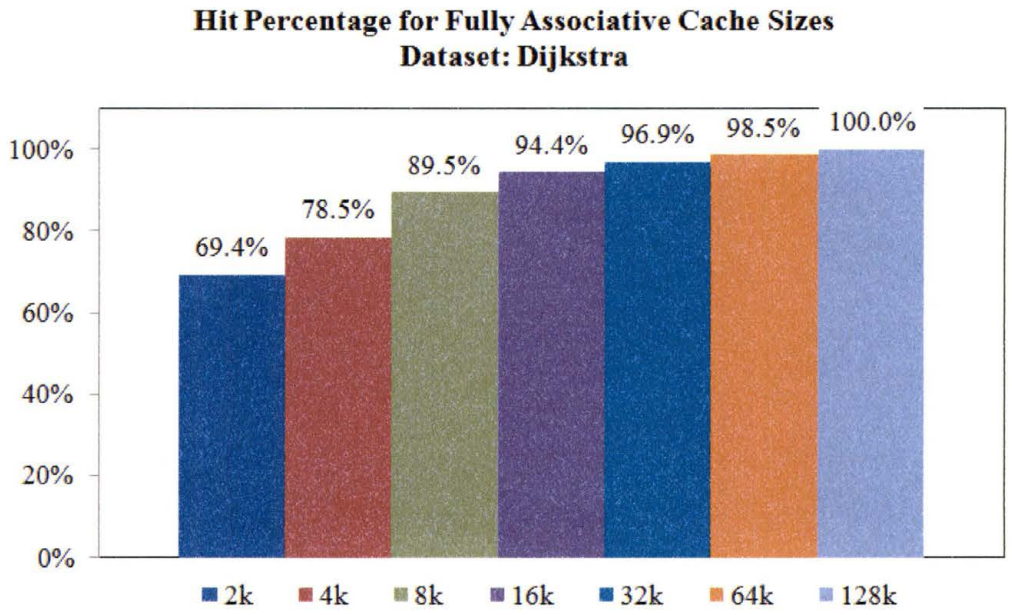

Figure 56. DJK hit percentage for fully associative cache of various sizes

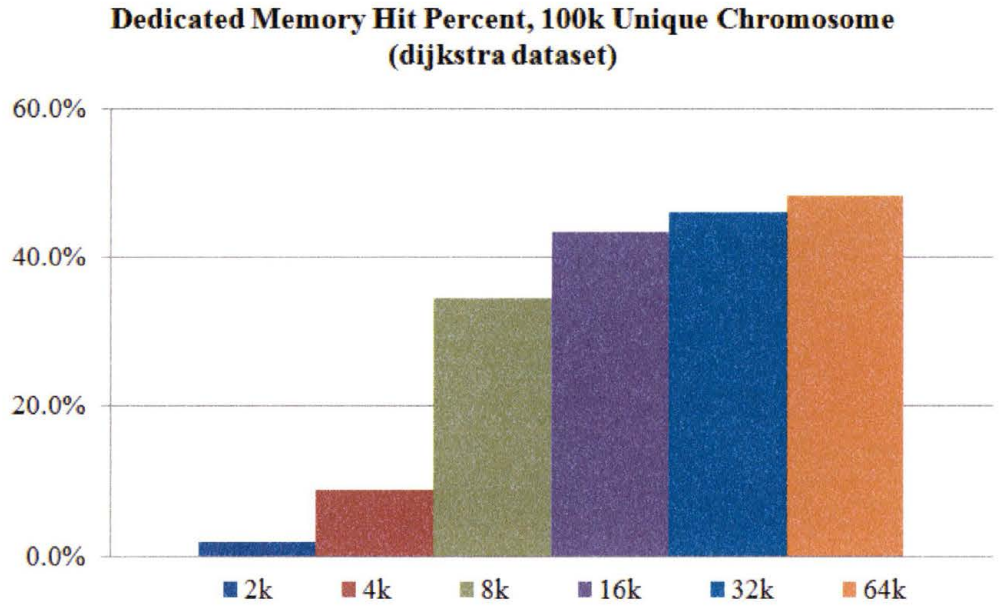

Figure 57. DJK dedicated memory hit percentage

Fully associative cache shows much higher performance in speedup and hit percentages over set associative cache. Figure 54 show that at 2k the Dijkstra dataset attains a speedup of over 2x, which is greater than what is achieved with 1M set associative cache.

The speedup for fully associative cache shows better performance versus dedicated memory (figure 54 & 55). At the smallest cache size, 2k, fully associative cache has a speedup close to a dedicated memory size of 64k. The hit percentages in figures 56 and 57 show the effectiveness of a replacement policy on speedup when comparing fully associative cache with dedicated memory.

Figures 58 and 59 show the experiments for the dataset S11. The results are similar to those of the Dijkstra dataset.

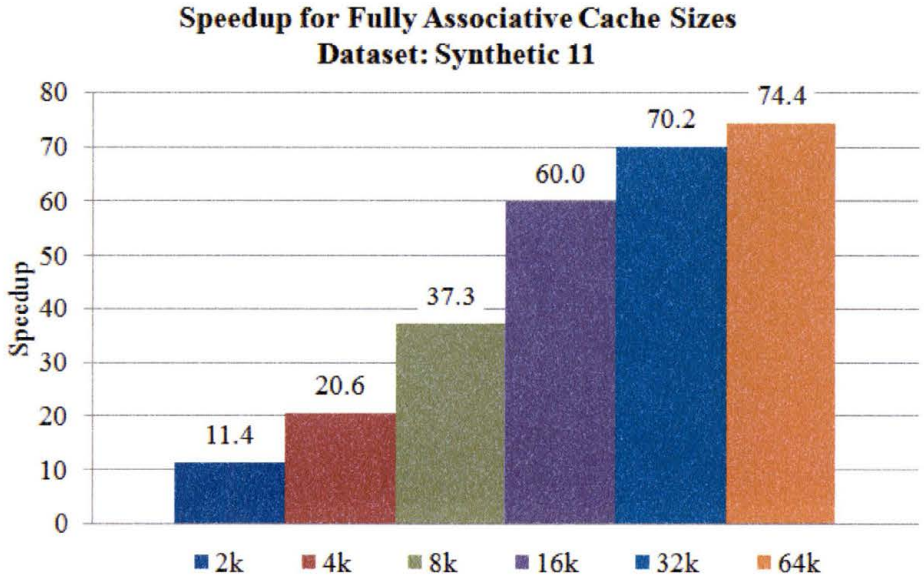

Figure 58. S11 speedup for fully associative cache of various sizes

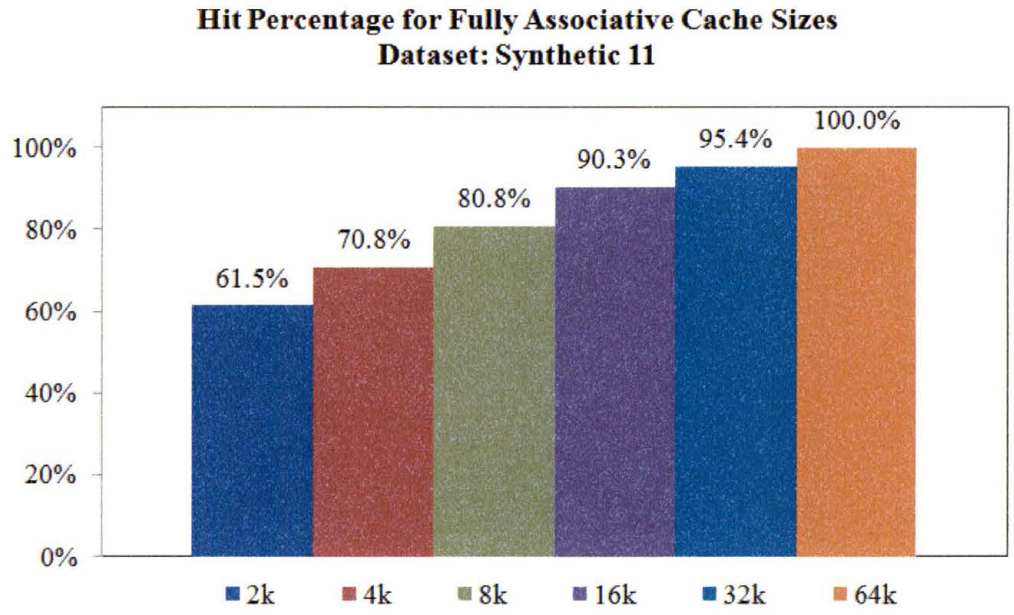

Figure 59. S11 hit percentage for fully associative cache of various sizes

A check of the quality convergence for fully associative cache is shown in figure 60. What is apparent is that the different model of cache does not impact the rate of convergence to the global optimum.

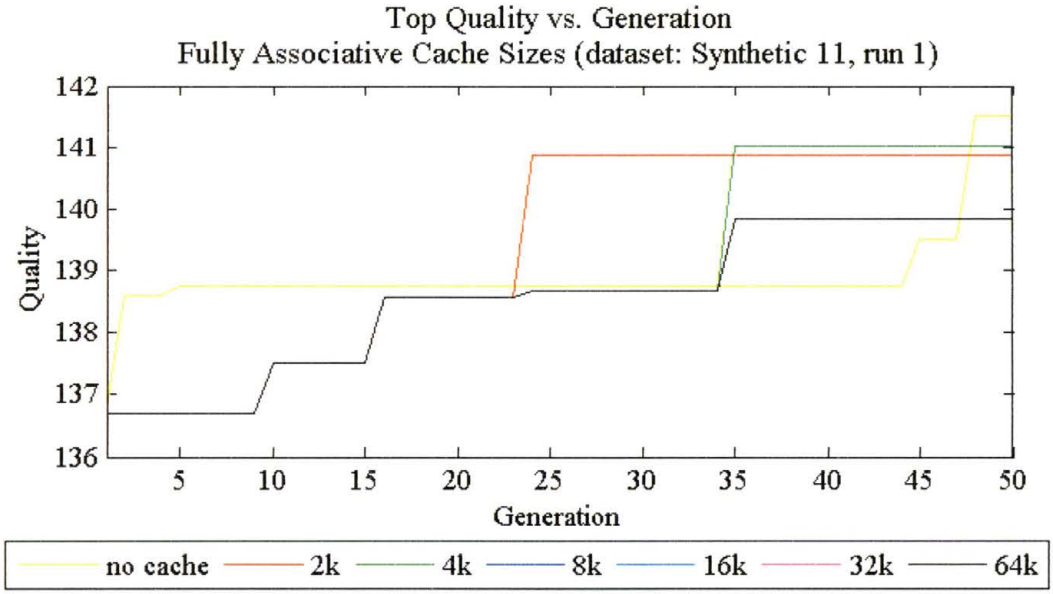

Figure 60. S11 best quality per generation for fully associative cache sizes

CHAPTER VII

ANALYSIS OF RESULTS

The similarity of the results between the Freescale Semiconductor datasets and the synthetic datasets suggests that the quick convergence of to the global optimum is not a data dependent issue. Regardless of the cache size or replacement policy, CGA invariably decreases in performance well before 1000 generations have elapsed.

Figure 61 shows the decay of the number of unique chromosomes per generation, with the upper bound being the size of the child list: 128 chromosomes.

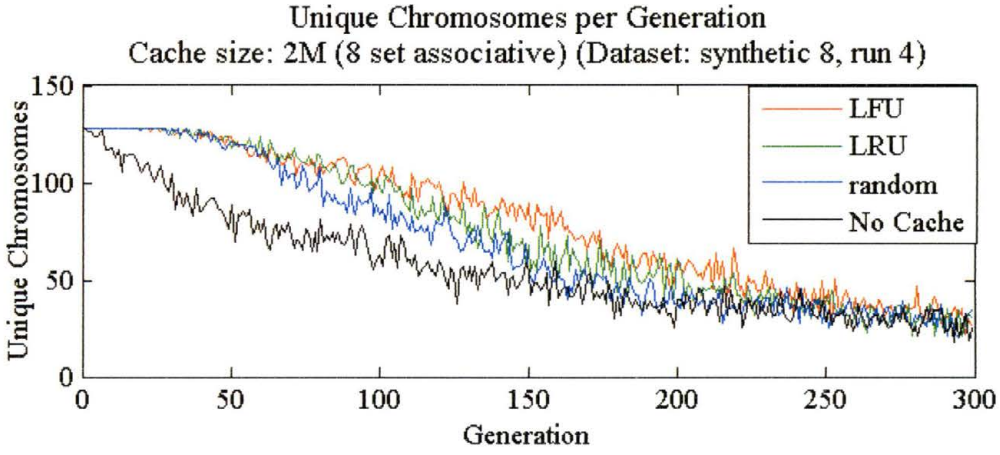

Figure 61. S8 unique chromosome production rate decay using 8 set associative cache with a cache size of 512k

This figure shows that the decay is rapid, such that by generation 300 the difference in the number of unique chromosomes produced per generations between CGA and GA disappears.

The results of this thesis shows that set associative cache can provide speedup applied to GA in the context of COP. However, the results also show that this speedup comes at a high space tradeoff when compared to dedicated memory or fully associative cache. Replacement policies do not affect this behavior, nor does the use of different datasets.

The poor performance of set associative cache lead this thesis to investigate the convergence of CGA to the global optimum of the feature space. It is found is that CGA does not outperform GA in quality convergence within 50 generations regardless of the replacement policy implemented. This is a surprising result which does counter this thesis's hypothesis that CGA will find a feature subset of higher ISODATA clustering quality.

To check the CGA data dependency, a quality histogram is produced to profile the four Freescale Semiconductor. The histograms showed tight grouping of feature subset solutions with quality measures close to the global optimum. To provide an experimental control to this property, synthetic data is generated with a flatter quality profile.

Using these synthetic datasets, experiments are redone to check the effectiveness of cache sizes, set associativity, and replacement policies. The results confirmed the work done on the Freescale Semiconductor datasets. First, cache sizes larger than that of dedicated memory are required for equal speedup. Second, set associativity did not affect the behavior of cache to any large degree. Third, while the relative performance of individual replacement policies is data dependent, the quality of the solutions found for a

given number of generations is no better than that found with GA. The hit percentages for CGA are below 50% for all datasets when looking at cache sizes of 512k or less.

In all experiments cache utilization is shown to be high. The results of the fixed subset number cache experiments, figure 36 in chapter 6, show that the cache utilization is not dependent on dataset, or replacement policy. The utilization is the same in both the experiments using the full feature subset search space as the experiments using the fixed feature subset search, showing that the utilization is independent of the size of the search space.

CHAPTER VIII

CONCLUSIONS AND FUTURE RESEARCH

This thesis implemented Genetic Algorithm with the focus on the feature selection problem. Using ISODATA as the heuristic fitness function the idea is to produce a high fidelity feature subset by searching the combinatorial space. This work follows others in the study of heuristic algorithms to solve the FSP. In particular, work such as that done by Siedlecki and Sklansky (1989), suggested GA may outperform other heuristic methods in solving FSP.

A combinatorial search may often be computationally costly, such that some recordkeeping method may be needed, trading greater time performance for space allocation. Given an effective set associative configuration and replacement policy a cache can be designed to outperform a simple dedicated memory scheme not only in speedup, but in quality of solutions for a given number of generations.

Experiments done in this thesis with dedicated memory show that a simple recordkeeping scheme can provide a speedup of over 2x for memory sizes of as little as 64k. This is not a surprising result given the research done by Hertel and Pitassi (2007) with static recordkeeping in heuristic search. However, the relatively small dedicated memory size required for speedup is interesting.

The degree of temporal locality discovered in this research suggests set associative cache would outperform dedicated memory. Work done by Chang and Huang (2009) on GA with hardware cache, in part, motivated this line of research. The experimental evidence did not show this to be the case. While CGA did provide a speedup often much greater than that seen in dedicated memory, the increase in memory size required for cache to match the speedup found in dedicated memory, is four fold. An interesting result as well is the lack of difference between set associativity on CGA performance. This may be due to the lack of spatial locality inherent in the chromosome bit field representation of the address space.

Testing different replacement policies such as LRU, LFU, and random, revealed a high level of data dependence. The peak performance of CGA with a particular replacement policy is shown to vary between dataset. The fact that the random replacement policy outperformed LRU and LFU on many instances can be related to GA's stochastic behavior originating from crossover point selection, parent selection, and the random mutation rate. This randomness may blur the effects of temporal locality, reducing the effectiveness of any "best" replacement policy.

It is apparent from this research that set associative cache does not improve upon dedicated memory in terms of time-space tradeoff. Instead, cursory work done on fully associative cache hold a much great performance gains in terms of speedup, but still not in terms of quality solutions. Further testing is needed to find the reason why CGA cannot find a higher quality over GA. One avenue of investigation is presetting the population list with known low quality solutions to test the rapidness of convergence to optima. A study also of interest, is the testing of the fidelity of the best quality feature set

found in this thesis, for the workload characterization datasets from Freescale Semiconductor, on the ISA simulator and comparing the results from the Verilog simulator.

One area of expanded research is reordering of GA instructions, to optimize for typical hardware cache. This process of making GA cache-aware may yield greater speedup than seen in this thesis for set associative cache. Another area of investigation, is a study on the effects of parallelism on the performance of cache hit percentages and speedup. A third future study direction can be experiments with larger number of features, perhaps an order of magnitude and greater, showing the performance of CGA on a more complex feature space.

LITERATURE CITED

Aggarwal, A. (2002). Software Caching vs. Prefetching, *Proceedings of the 3rd International Symposium on Memory Management*, 157-162.

Allen, D. and Darwiche, A. (2003). Optimal Time-Space Tradeoff in Probabilistic Inference, *Proceedings of the International Joint Conferences on Artificial Intelligence*, 969-975.

Amadahl, G. M. (1967). Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. *American Federation of Information Processing Societies Conference Proceedings*, 30 483-485.

Ball J. H. and Hall. D. J. (1966). A Clustering Technique for Summarizing Multivariate Data. *Behavioral Science*, 12(2), 153-155.

Bello, R., Puris, A., Nowe, A., Martinez, Y., and Garcia, M. (2006). Two Step Ant Colony System to Solve the Feature Selection Problem, *Progress in Pattern Recognition, Image Analysis and Applications, vol. 4225*. Springer Berlin-Heidelberg, Germany, 588-596.

Brock, M. (2010) *Feature Selection for Slice Based Workload Characterization and Power Estimation*, Texas State University, Master Thesis.

Cantú-Paz, E. and GoldBerg, D. (1999). On the Scalability of Parallel Genetic Algorithms, *Evolutionary Computation*, 7(4).

Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., and Menon, R. (2001). *Parallel Programming in OpenMP*, London, UK: Academic Press.

Chang, F. and Huang, H. (2009). A Study on the Cache Miss Rate in a Genetic Algorithm Implementation, *Fifth International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, 795-797.

Ciesielski, V. and Scerri, P. (1997). An anytime algorithm for scheduling of aircraft landing times using genetic algorithms, *Australian Journal of Intelligent Information Processing Systems*, 4, 206-213.

Cover, T. M., and Van Campenhout J. M. (1997). On the possible orderings in the measurement selection problem. *IEEE Transactions on Systems, Man and Cybernetics* 7(9) 657–661.

Dy., J. G. and Brodley, C. E. (2004). Feature Selection for Unsupervised Learning. *Journal of Machine Learning Research* 5 845-889.

Friedman, M. and Kandel, A. (1999). *Introduction to Pattern Recognition*, London, UK: Imperial College Press.

Fukunaga, K. (1990). *Statistical Pattern Recognition (second edition)*. California: Academic Press.

Gen, M. and Cheng, R. (2000) *Genetic Algorithms and Engineering Optimization.* New York: Wiley-Interscience.

Grant, K. and Horsch, M. C. (2007). Efficient Caching in Elimination Trees, *Florida Artificial Intelligence Research Society Conference,* 98-103.

Guan, S. U., Zhu, F., and Li, P. (2004). Modular Feature Selection using Relative Importance Factors. *International Journal of Computational Intelligence and Applications* 4(1) 57-75.

Guyon, I. J., Weston, S., and Barnhill, V. (2002). Gene selection for cancer classification using support vector machines. *Machine Learning* 46(1).

Hennesy, J. L. and Patterson, D. A. (2007). *Computer Architecture: A Quantitative Approach 4th Ed. San Francisco*, CA: Morgan Kaufmann.

Hertel, P. and Pitassi, T., (2007). An Exponential Time/Space Speedup for Resolution, *Electronic Colloquium on Computational Complexity*, 46 1-25.

Jain, A. K. and Dubes, R. C. (1988). *Algorithm for Clustering Data*, New Jersey: Prentice-Hall, Inc.

Joshi, A., Eeckhout, L., John, L. K., and Isen, C., (2008). Automated Microprocessor Stressmark Generation, *The 14th International Symposium on High Performance Computer Architecture*, pp. 229-239.

Koza, J. R. (1992). *Genetic Programming*, Cambridge, Massachusetts: MIT Press.

Kudo, M. and Sklansky, J. (2000). Comparison of Algorithms that Select Features for Pattern Classifiers. *Pattern Recognition* 33 25-41.

Kudo, M., Somol, P., Pudil, P., Shimbo, M., and Sklansky, J. (2000). Comparison of Classifier Specific Feature Selection Algorithms, *Advances in Pattern Recognition, vol. 1876*. Springer Berlin-Heidelberg, Germany, 677-686.

Linde, Y., Buzo, A., and Gray, R. M. (1980). An Algorithm for Vector Quantization Design. *IEEE Transactions on Communications*, 28(1) 84-95.

Luo, Y., Joshi, A., Phansalkar, A., and John, L. K., Ghosh, J., (2008). Analyzing and Improving Clustering Based Sampling for Microprocessors, *Journal of High Performance Computing and Networking*, 5(4), 352-366.

Nakariyakul, S. (2008). On the Suboptimal Solutions using the Adaptive Branch and Bound Algorithm for Feature Selection. *Proceedings of the 2008 International Conference on Wavelet Analysis and Pattern Recognition*, 384–389.

Obitko, M., Hochschule für Technik und Wirtschaft Dresden, Czech Technical University. *Introduction to Genetic Algorithms.* Retrieved January 10, 2009, from the website: http://cs.felk.cvut.cz/~xobitko/ga/

O'Hallaron, D. R. and Bryant, R. E. Carnegie Mellon University, (2002). *Cache Memories*. Retrieved April 20, 2010 from website: http://www.cs.cmu.edu

Pacheco, P. S. (1997). *Parallel Programming with MPI*, Morgan Kaufmann Publishers, Inc.

Quinn, M. J. (2003). *Parallel Programming in C with MPI and OpenMP*, McGraw Hill Higher Education.

Santos, E. E. and Santos E., Jr. (2000). Cache Diversity in Genetic algorithm Design. *Florida Artificial Intelligence Research Society Conference* 107-111.

Siedlecki, W. and Sklansky, J. (1988). On Automatic Feature Selection. *International Journal of Pattern Recognition Artificial Intelligence* 2(2) 197-220.

Siedlecki, W. and Sklansky, J. (1989). A note on genetic algorithms for large-scale feature selection. *Pattern Recognition Letters,* 10 335-347.

Shi, D., Shu, W., and Liu, H. (1998). Feature selection for handwritten Chinese character recognition based on genetic algorithms, *IEEE International Conference on Systems, Man, and Cybernetics Vol. 5*, 4201-4206.

Talbi, E. (2009). *Metaheuristics, From Design to Implementation*, New Jersey: John Wiley & Sons, Inc.

Tanenbaum, A. S. (2006). *Structured Computer Organization, 5th Ed.*, New Jersey: Pearson Prentice Hall.

Texas Advance Computing Center, University of Texas. (2009). Ranger User Guide. Retrieved 2009 from: http://services.tacc.utexas.edu/index.php/ranger-user-guide

Theodoris, S. and Koutroumbas, K. (1999). *Pattern Recognition,* London, UK: Academic Press.

Wang, X., Yang, J., Teng, X., Xia, W., and Jensen, R. (2007). Feature Selection Based on Rough Sets and Particle Swarm Optimization. *Pattern Recognition Letters* 28 459-471.

Yusta, S. C. (2009). Different Meta-Heuristic Strategies to Solve the Feature Selection Problem. *Pattern Recognition Letters*, 30 525–534.

Zhang, T., Fu, X., Goh, R. S. M., Kwoh, C. K., and Lee, G. K. K. (2009). A GA-SVM Feature selection Model Based on High Performance Computing Techniques, *Proceedings of the 2009 IEEE International Conference on Systems, Man, and Cybernetics*, 2653-2658.

Zhang, H. and Sun, G. (2002). Feature Selection using Tabu Search Method. *Pattern Recognition* 35 701-711.

VITA

Daniel Isamu Lowell, was born in Annapolis, Maryland on September 24th, 1975, the son of Eiko Nakamura Lowell and Daniel Anthony Lowell. He received his B.A. in physics from the University of Colorado, Boulder in 1999. In 2007 he enrolled in Texas State University-San Marcos Computer Science graduate program.

Permanent Address:

1317 Kenwood Ave.

Austin, TX 78704

This thesis was typed by Daniel I. Lowell.