Code Compaction Using Genetic Algorithms

Keith E. Mathias, Larry J. Eshelman, J. David Schaffer
Philips Research 345 Scarborough Road, Briarcliff Manor, NY 10510

Lex Augusteijn, Paul Hoogendijk, Rik van de Wiel
Philips Research Prof. Holstlaan 4, Eindhoven, The Netherlands

Abstract

One method for compacting executable computer code is to replace commonly repeated sequences of instructions with macro instructions from a decoding dictionary. The size of the decoding dictionary is often small in comparison to the number of all possible macros. Choosing the macros that yield the best compaction is a difficult subset selection problem because multiple, but colliding, macros may be applicable to many code segments. We show that a genetic algorithm using a new crossover operator, MSX, gives better compaction than heuristics designed specifically for this problem. We also compare MSX with other crossover operators on a surrogate problem that models the essential properties of the code compaction problem.

1 INTRODUCTION

For large volume consumer electronics, the main cost factor of the embedded software is its memory usage, both ROM and RAM. For many products, like cellular handsets, memory costs are an increasingly greater portion of the total product cost. A reduction of the code memory size has several benefits. It may reduce the direct cost of the product, by reducing the bill of materials, usually silicon area or chip count. A reduction of the code size can also be used to fit more features into the same ROM, enhancing the value of the product. One method for compacting executable computer code is to replace commonly repeated sequences of instructions with *macro instructions* from a decoding dictionary.

Choosing macro instructions to replace commonly occurring code segments to compress the executable code footprint is a subset selection (SSS) problem. The goal is to select the subset of macros that will yield the best compaction of the code. These macros can then be stored in a dictionary and used for decoding during execution.

The problem of determining the best subset dictionary is known to be NP-complete (Storer and Szymanski, 1982). Collisions between macros are expected to be prevalent. Collisions occur at the points in the code where multiple macros are applicable. The number of macros that can fit into a decoding dictionary is small compared with the total number of macros possible in a given program. To evaluate the effectiveness of a macro dictionary, the macros must be "tiled" on the code and the actual compact code size measured. This process is computationally expensive, scaling with the size of the macro dictionary, the size of the original code and the number of collisions encountered. Current approaches to finding the best subset of macros have been limited to greedy search techniques (Kozuch and Wolfe, 1994) and heuristics (Fraser and Proebsting, 1995; Hoogerbrugge, et al., 1999).

Another special feature of our code compaction problem is that, while the size of the dictionary is given, one may not have complete freedom to choose all macros that will fit; if some macros are chosen, one or more others may be needed to de-reference longer macros and will be automatically included to insure that the dictionary is complete. This means that if a chromosome specifies a list of macros equal in size to the dictionary size, some of them may not be used. This has significant ramifications for crossover operators.

SSS problems can be classified according to whether the subset size is known or unknown. Genetic algorithms have been shown to be useful in SSS problems and a number of operators have been designed specifically for this purpose. Simple bit selection schemes are often used for feature selection problems (Bala,

et.al, 1996; Whitley, et.al, 1997; Raymer, et.al, 1997) where the subset size is unknown and often there is a secondary goal of minimizing the size of the subset. Other representations and operators have been devised for situations where the subset size is known (Radcliffe, 1993; Crawford, et al., 1997).

To facilitate the exploration of operators and algorithms we have devised a SSS problem generator that models the essential properties of our code compaction problem but requires significantly less evaluation time, and the degree of epistasis (i.e., collisions) for any problem can be tuned using our generator.

We present performance results for various SSS recombination operators for randomly generated problems, varying the degree of epistasis and introduce a new crossover operator, MSX. MSX performs better than other operators when only an upper bound on the subset size is known. We then compare the performance of MSX to a domain-specific search heuristic on several benchmark programs, as well as some "in-house" programs.

2 GENETIC RECOMBINATION OPERATORS FOR SUBSET SELECTION

Because the upper bound on the subset size is small relative to the overall set size in our code compaction problems, we chose an integer representation. That is, our chromosome is a list of integers that are the indexes of the items being selected. While other representations are often used (e.g., binary selection) we found their performance to be uncompetitive on our problems.

Previously, we have characterized crossover as a form of convergence controlled variation (CCV) (Eshelman, Mathias, and Schaffer, 1996): crossover uses the convergence of the population to constrain variation. This follows from the property of crossover that Radcliffe calls respect: if two parents have a feature in common, then the children will inherit that feature (Radcliffe, 1993). Thus, for operators with the property of respect, variations are constrained to features that have not converged.

In order to guarantee respect when using an integer representation, the crossover operator needs to copy all indexes which are common to the two parents into each child. How the remaining elements are filled in is an open question from a CCV point of view. One method, suggested by Radcliffe, is to fill the "remaining places in the child with a random selection of the unused elements from the two parents." Radcliffe refers to such an operator as Random Respectful Recombination (RRR). We have implemented two variations of RRR. The first operator we refer to as match and mix crossover (MMX). Its only difference from RRR is that the common elements keep the same positions in the child as they had in the parent, whereas RRR doesn't require that the common elements retain the same position. Our second operator pairs the indexes that are unmatched, and then randomly swaps them between individuals. We shall refer to this operator as match and swap crossover (MSX).

Both MMX and MSX, as described so far, constrain the set of values for the unmatched indexes to values represented in the parents. Not only are children guaranteed to inherit all the indexes that are in both parents, but children are guaranteed not to acquire indexes absent from both parents. They "inherit" the common absence of certain indexes. Such an operator can be said to have both positive respect (the common presence of properties) and negative respect (the common absence of properties). MMX and MSX include a parameter which allows us to relax the requirement for negative respect, thus allowing new values to replace indexes for which there is no match. This is a form of mutation, but it is convergence controlled mutation since mutation is restricted to those indexes that are not matched in the two parents. Converged indexes are protected from mutation, and as the population converges, the effective rate of mutation decreases.

Both MMX and MSX (differing at the second step) produce two offspring from two parents as follows:

1. Match: The common elements in the two parents are copied to the two children, preserving their original positions. Thus, one of the children will have the common elements in the positions of the first parent, and the second will have the common elements in the positions of the second parent.
2. a) Swap (MSX): The remaining (unmatched) elements are paired up in order and swapped with a 50% probability. For example, if the first unmatched element in the first parent is in the fifth position, and the first unmatched element in the second parent is in the fourth position, then these two elements will be swapped with a 50% probability (see Figure 1-a).
   b) Mix (MMX): The remaining (unmatched) elements from the two children are placed in a bag, and then the former positions of the unmatched elements of the two children are filled by randomly selecting elements from the bag without replacement (see Figure 1-b).

3. Mutation: Positions with unmatched elements are mutated with a specified probability. The new value is chosen randomly from the set of all elements, with a uniform distribution, taking care not to duplicate indexes currently in the child.

Besides RRR, Radcliffe introduced another subset selection recombination operator, the random assorting recombination (RAR) operator (Radcliffe, 1993). To have the attribute of proper assortment, a recombination operator must be able to produce, in a single operation, any potential child whose total genetic material is contained in one or the other parent. In the context of subset selection this means that any subset θ of size $|\theta|$, composed of elements from the two parents, can be generated by the recombination operator. For SSS problems proper assortment comes at the expense of respect. RAR has a parameter, w, for controlling the degree to which the RAR is likely to produce a respectful recombination. As w approaches infinity, RAR approaches RRR.

As in RRR, the child is created by drawing elements from a bag. The bag can contain both positive and negative copies of elements. If a positive element is drawn, then the element is included in the child. However, if a negated element is drawn, then the element is marked as no longer available for inclusion in the child. The RAR operator produces a child (i.e., subset, θ of size $|\theta|$) from a total set Θ of size $|\Theta|$ as follows:

1. Convert w into a fraction, w_1/w_2. (If w is 0, w_1 is set to 1, and w_2 to infinity.)
2. Place w_1 copies of each element common to both parents in a bag. Place w_1 negated copies of each absent element (i.e., in neither parent) in the bag. Place outside the bag a positive copy of each absent element.
3. Place w_2 copies of each unique element (i.e., element found in only one parent) in the bag. Place w_2 negated copies of each unique element in the bag.
4. Repeatedly draw elements from the bag. Whenever an element is drawn, remove any duplicates or complements from the bag. If the element is positive, place it in the child. Whenever a negative common element is drawn, remove its positive complement from the elements outside the bag so that they are no longer available.
5. Continue step 4 until either: a) $|\theta|$ positive elements from the bag have been chosen, b) $|\Theta| - |\theta|$ negative elements have been chosen, in which case complete the child with the remaining positive elements inside and outside the bag (all remaining elements are necessary to complete the child), or c) no more positive elements exist in the bag, in which case, select positive elements from outside the bag until the child is complete.

By including negated absent elements in the bag and weighting these elements by the same w_1 as the positive common elements, RAR gives equal weight to positive and negative respect. But this also means that RAR introduces mutation into the operator since the child can have elements that are not in either parent. It should be noted, however, that mutation can only occur when the operation is respectful (i.e., all common elements are passed to the child), so mutation is always convergence controlled.

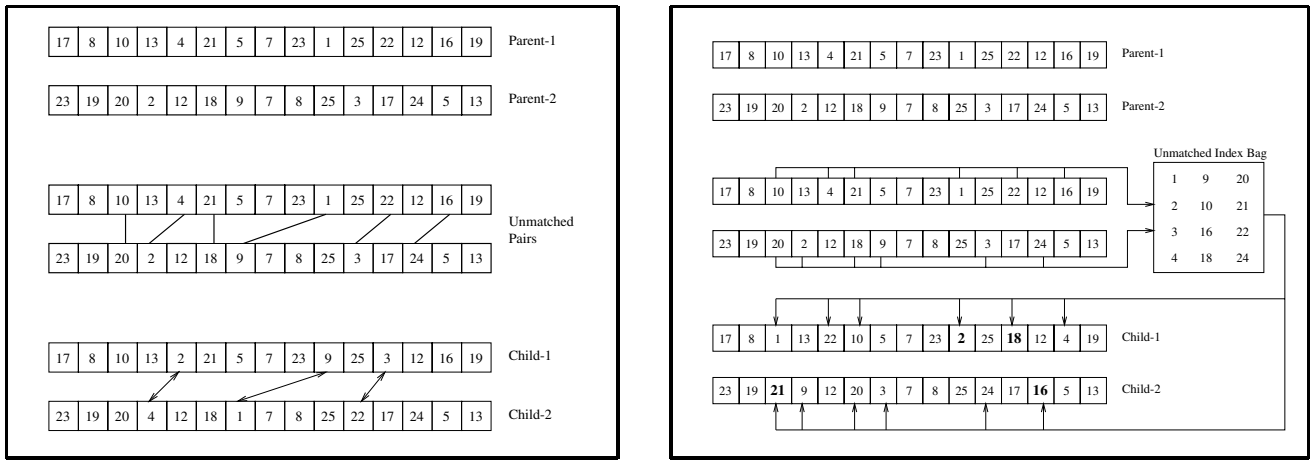
3 CODE COMPACTION USING A DICTIONARY OF MACROS

A well known code compaction technique is to compile application programs to an application-specific instruction set (Fraser and Proebsting, 1995), which includes basic instructions and macro instructions. Given an application program we have to come up with an instruction set that minimizes the code size of the application. We construct such an instruction set in two steps. We start with a small fixed set of basic instructions that are sufficient to implement every C/C++ program. This guarantees that each program can be compiled to the instruction set independent of the application-specific instructions. Then we add *macro instructions* that correspond to sequences of basic instructions, making the instruction set application-specific. To be able to execute the application-specific instructions on a standard processor, the code has to be translated to native processor instructions. This translation can be done by a software interpreter, as shown by Hoogerbrugge (Hoogerbrugge, 1999), or by a hardware decode unit (Benes, Wolfe and Nowick, 1998; Game and Booker, 1998).

3.1 MACRO INSTRUCTIONS

If we have a sequence of basic instructions that occurs frequently in a compiled program, we can introduce a *macro instruction* to encode this instruction sequence. Replacing all occurrences of this sequence by the new macro instruction reduces the number of instructions in the code. In a way, this technique creates CISC-like instructions to optimize for code compactness.

When we use a hardware decode unit to translate the compact instructions to native processor instructions, this unit must be programmed with the translation of



(a - MSX) (b - MMX)

Figure 1: Example of MSX and MMX crossover operators.

each compact instruction. Because the decode unit has limited memory, we can only add a limited number of macros to the instruction set. In our case, the total number of instructions, basic and macro instructions, is limited to 511. Furthermore, the length of a macro (the number of basic instructions represented) is limited to 4. To minimize the code size, we have to select the best set of macros, with maximal length 4, such that the total number of instructions is at most 511. We try to find the best set of macro instructions by constructing a large pool of candidate macro instructions, based on the program, and select the best subset from this pool.

Consider the following code sequence where A, B, C, and D are instructions with parameters in parentheses:

$$A(1)\ B(6)\ C\ D(1)\ A(2)\ B(6)\ C\ D(2)$$

Assume we have the following set of candidate macros $M1$, $M2$, $M3$, $M4$, and $M5$:

$$\begin{array}{lcl} M1 & : & A(1)\ B(6)\ C\ D(1) \\ M2 & : & A(2)\ B(6)\ C\ D(2) \\ M3(x) & : & A(x)\ B(6)\ C\ D(x) \\ M4 & : & B(6)\ C\ D(1) \\ M5 & : & C\ D(1)\ A(2) \end{array}$$

Macro $M3$, is a parameterized macro. The value of the parameter of $M3$ will be used as a parameter for sub instructions A and D. So macro $M3(1)$ corresponds to the sequence $A(1)\ B(6)\ C\ D(1)$, which is the same as the sequence of macro $M1$.

If we look at the program fragment above, we can replace it by the sequence of macros $M1\ M2$ but also with sequence $M3(1)\ M3(2)$. In such a case, we say that macro $M3$ *collides* with macros $M1$ and $M2$. In general, macro $M3$ will be more expensive than $M1$ and $M2$ because its parameter has to be encoded in the instruction stream. Implementing the program fragment by sequence $M1\ M2$ will give denser code, but we use two macros instead of one. Since the number of macros is limited, it may be that for a complete program macro $M3$ may ultimately be a better choice, but given only the local program fragment, using macros $M1$ and $M2$ would be the better solution.

In addition to collisions between macros that can cover the same sequence of basic instructions, there can also be a collision when we have partial overlap. This occurs for macros $M4$ and $M5$. Due to the limitation on the number of macros we may select, the collisions between macros, and the number of possible candidate macros in large programs, selection of a good set of macros is extremely difficult.

4 METHODS

Crossover operator performance in this work was tested in the context of the CHC adaptive search algorithm (Eshelman, 1991). CHC has been shown to be very robust, yielding effective optimization results without tuning the algorithm parameters. CHC with standard parameter settings generally performs better than the simple genetic algorithm (SGA) (Eshelman, 1991; Mathias and Whitley, 1994; Whitley, et.al, 1995). CHC is a generational style GA with three distinguishing features. First, selection in CHC is monotonic: only the best M individuals, where M is the population size, survive from the pool of both the offspring and parents. Second, in order to maintain ge-

netic diversity and slow population convergence, CHC prevents parents from mating if their genetic material is too similar (i.e., incest prevention). When using the index representation, incest prevention was implemented by permitting matings for only those pairs of individuals whose number of unmatched indexes was greater than the incest threshold. For these experiments we have used an initial incest threshold of $|\theta|-2$. As usual in CHC, the incest threshold is decremented whenever no offspring are accepted into the parent population, causing the incest threshold to drop as the population converges. Third, CHC uses a "soft-restart" mechanism. When convergence has been detected, or the search stops making progress, the population is *diverged*. For these experiments this is done by preserving the best individual found so far and generating the rest of the population randomly. The initial population is also randomly generated. In both cases, no index is used more than once in a chromosome.

5 A SURROGATE TEST PROBLEM GENERATOR

The search for good macros in code compaction is a SSS problem and evaluating the compaction obtained in real programs for a given subset of macros is computationally expensive. Designing a surrogate problem that models the characteristics of our real-world problem without the computational cost allows more latitude for exploration. We developed a SSS problem generator, modeled after our code compaction problem, that allowed us some control over the interaction between subset members (analogous to macro collisions) and which required significantly less computation for evaluation. The objective in these problems was to find the best subset, θ, of a specified size, $|\theta|$, from a candidate pool, Θ, where $\Theta = \{x_i : i = 1, |\Theta|\}$ and $|\Theta| >> |\theta|$. The fitness of θ is determined by:

$$f(\theta) = \sum_{i=1}^{|\Theta|} \begin{cases} C(x_i) & \text{if } x_i \in \theta \\ 0 & \text{otherwise} \end{cases}$$

where $C(x_i)$ is the fitness contribution of x_i as a member of θ. Elements that are not a member of θ do not contribute to the subset fitness for these problems.

The simplest form of this problem models the case where there are no-collisions (i.e., no epistasis). Let κ represent the **minimum** number of elements in Θ that interact with each element x_i. Then this is the $\kappa = 0$ case. Problems in this class were generated with the values of $C(x_i)$ randomly selected in the range $[1, 10]$ except for $|\theta|$ of them (randomly chosen) which were set to zero. The fitness of the single optimum is zero.

Another class of problems were generated with inter-element interactions ($\kappa > 0$) to model collisions between macros. The fitness contribution $C(x_i)$ of an element x_i is computed according to:

$$C(x_i) = \beta(x_i) + I(x_i)$$

$$I(x_i) = \sum_{j=1}^{|\Theta|} \begin{cases} \psi(x_i, x_j) & \text{if } x_j \in \theta \text{ and } i \neq j \\ 0 & \text{otherwise} \end{cases}$$

where $\beta(x_i)$ is the base value for the element x_i, $I(x_i)$ is the interaction function for element x_i, and $\psi(x_i, x_j)$ is the interaction value for the link from x_i to x_j. Thus, $f(\theta)$ depends only on the elements contained in θ. The object in the search for the problems where $\kappa > 0$ was to find the subset, θ, whose cumulative base values and intra-subset interaction penalties were minimal. Problem instances in this class were constructed by randomly assigning "base" values, β, in the range [-10.0, 0]. For each element, x_i, in Θ, κ other elements were randomly chosen, without replacement, as the interactors of element x_i. Each of these interactions, $\psi(x_i, x_j)$ is randomly assigned a positive real value in the range $(0.0, 10.0]$ so as to interfere with the negative β values. *Reciprocal* interaction values, $\psi(x_j, x_i)$, are also randomly chosen from the same range for each of the κ elements. Thus, if macro x_i interferes with macro x_j there will always be a reciprocal, but not necessarily equal, interference. The optimal fitness for problems where $\kappa > 0$ is not known.

5.1 SUBSET SELECTION WHEN ONLY AN UPPER BOUND ON θ IS KNOWN

Substantial testing has shown that when the subset size is known, MMX is as good as, and often better than, the RAR and MSX operators. However, in our real-world code compaction problem only an upper bound on the size of the subset is known. This is because some of the macros specified by the chromosome require other macros for support (i.e., recursion of depth one). These *support macros* are automatically added to θ when required, consuming valuable slots in the dictionary. Thus, the number of macros that may be specified by the chromosome is uncertain and context sensitive (i.e., depends on the preceding macros in the chromosome). Knowing only an upper bound on $|\theta|$, we are forced to code for the maximum number of elements that can be chosen. To simulate this added difficulty in our SSS problem generator and test the effectiveness of these subset operators, we set our chromosome length to 125, but used only the first 100 elements when determining the fitness.

The results reported here are based on 30 random problems for each of five interactivity levels (i.e., $\kappa = 0$,

Alg	κ = 0 Solved	Trials	Fitness	κ = 5 Fitness	κ = 10 Fitness	κ = 20 Fitness	κ = 50 Fitness
RAR w=0.25	0	( )	16.83 (0.28)	-881.53 (2.52)	-782.59 (3.76)	-472.56 (8.95)	1148.53 (20.43)
RAR w=7	0	( )	12.4 (0.26)	-891.67 (2.18)	-811.54 (2.83)	-544.50 (5.50)	890.91 (12.67)
MMX	11	178,002 (6,745)	0.67 (0.10)	-922.39 (1.72)	-874.80 (1.87)	-697.50 (4.21)	432.44 (11.14)
MMX-S	30	31,905 (2,245)	0.00 (0.00)	-928.01 (1.73)	-892.81 (1.65)	-751.25 (3.34)	325.33 (11.25)
MSX	30	37,597 (4,258)	0.00 (0.00)	-927.99 (1.62)	-892.51 (1.80)	-753.13 (3.94)	309.14 (11.09)

Table 1: Performance for algorithms on SSS for κ=0, 5, 10, 20, and 50 interactors.

5, 10, 20, and 50), keeping the problems constant for all algorithms. All of the problems were of the size $|\Theta| = 1024$ and $|\theta| = 125$, using only the first 100 elements in θ. For the non-interactive problems ($\kappa = 0$), the optimal elements are distributed randomly throughout Θ and the optimal solution is 0. The optimal solution for the interactive problems (i.e., $\kappa > 0$) is not known. A maximum of 200,000 trials was allowed for each of the algorithms on all problems. We used a mutation rate of 5% for MMX and MSX and values of 0.25 and 7 for w for RAR. Radcliffe (1993) recommended a value of $w = 0.25$, but after extensive testing we found $w = 7$ was best for our problems.

Table 1 shows the average best results, with the standard error of the mean in parentheses. RAR performs the worst on these problems, unable to solve the non-interactive problem a single time in the 200,000 trials allowed. MMX also performs quite poorly but is able to solve 11 of the 30 non-interactive problems. MSX performs much better than MMX or RAR. The reason that MMX and RAR perform poorly is that the values on the string not common to the parents are mixed in a bag and then randomly chosen for insertion into the offspring. The order that the elements are inserted into the offspring is completely random. This causes indexes to be moved back and forth across the boundary on the chromosome between the 25 elements that do not contribute to the fitness evaluation and the 100 positions that do. This makes it more difficult for the GA to converge on the meaningful elements (i.e., the first 100 indexes in the string). MSX, on the other hand, maintains the relative position that the indexes occupy in the parents, swapping the indexes directly. This prohibits indexes from relocating randomly in the chromosome. The reason that MMX performs better than RAR on these problems is that MMX has *some* positional stability, i.e., those elements common to both parents remain in their parental positions after recombination while in RAR this information is lost.

The positional instability exhibited by the MMX operator can be remedied by sorting the elements chosen from the bag according to their parental positions. Then the elements can be inserted into the available slots (MMX-S) in the offspring using the parental positions to produce a relative order.[1] This method adds a great deal of stability, improving the performance of the normal MMX operator in this selection environment as indicated by the performance results of MMX-S in the table. However, the added computational complexity makes this a less desirable solution than MSX. Similar positional information could be included for the RAR operator but the best way to do this is still an open question.

We ran an ANOVA using problem as a random factor (30) and with κ (5, 10, 20, 50) and operator (MSX, MMX, MMX-S, RARw=0.25, RARw=7) as fixed factors. After pulling out the obvious κ main effect, we still saw a significant operator main effect: there was no significant difference between MSX and MMX-S but MSX and MMX-S were significantly better than RAR, regardless of the value of w. A significant operator-κ interaction revealed that the inferiority of MMX first appears at $\kappa = 20$ and is sustained for $\kappa = 50$, while the inferiority of the RAR operator was significant from $\kappa = 5$ and sustained for all higher κ's.

6 CODE COMPACTION BENCHMARK RESULTS

The surrogate SSS problem generator proved to be a good model of our code compaction problem and resulted in the development of the MSX crossover operator. The performance rankings of the various crossover operators was the same in both the surrogate and real-world problems. However, the performance advantage of MSX over the other operators was more dramatic in our code compaction problem than in the surrogate SSS problems and MMX-S was not as competitive as was observed on the surrogate SSS problems. Thus, we compared the performance of MSX with that of a code-compaction-specific search heuristic (CCSSH).

[1] All elements in the offspring could be sorted according to the positions occupied in the parents; however, this is computationally expensive and the effort remains constant throughout the search. The computational effort for sorting only the indexes inserted from the bag decreases as the population converges.

	Pool = 12,000 Macros				Pool = 20,000 Macros				Best of Two Pools	
	CCSSH	MSX	Δ Bytes	Δ %	CCSSH	MSX	Δ Bytes	Δ %	Δ Bytes	Δ %
compress92	1472	1436	36	2.4%	N/A				36	2.4%
compress95	1864	1780	84	4.5%	N/A				84	4.5%
eqntott	7012	6764	248	3.5%	7148	6668	480	6.7%	344	4.9%
li	15428	15304	124	0.8%	15740	15136	604	3.8%	292	1.9%
Philips-1	17124	16520	604	3.5%	17332	16428	904	5.2%	696	4.1%
Philips-2	22280	21464	816	3.7%	22480	21492	988	4.4%	788	3.5%
sc	25560	24936	624	2.4%	25868	24980	888	3.4%	580	2.3%
m88ksim	44996	43824	1172	2.6%	45416	43668	1748	3.8%	1328	3.0%
espresso	48124	47420	704	1.5%	48296	47068	1228	2.5%	1056	2.2%
ijpeg	52944	51588	1356	2.6%	53376	51896	1480	2.8%	1348	2.5%
Philips-3	69028	67660	1368	2.0%	69864	67952	1912	2.7%	1076	1.6%
go	72632	71996	636	0.9%	73008	71328	1680	2.3%	1304	1.8%
perl	86600	85568	1032	1.2%	87356	84928	2428	2.8%	1672	1.9%

Table 2: Code sizes for benchmarks using CCSSH and MSX and improvements of MSX over CCSSH (Bytes/%).

Our CCSSH begins by generating all possible macros having four basic instructions or fewer. The frequency of occurrence for each macro in the program is determined at generation time. After all macros have been generated, an initial subset of macros is selected. For each macro its "gain" is calculated. The gain of a macro is its length minus 1, multiplied by its occurrence count. This gain represents the savings of a macro if no other macros were used (i.e., no collisions). The $|\Theta|$ macros with the highest gain are selected to form the macro pool, Θ (i.e., set of macros to choose from), where $|\Theta|$ is determined by the user. After the generation of the macro pool, an initial tiling is made, assigning a cost of one byte for each macro. Then the following steps are iterated until the decoding dictionary contains 511 macros:

1. From the occurrence count of each macro a new encoding cost is calculated using Huffman's algorithm: the 255 most frequently occurring macros get cost 1, the next 256, cost 2, and the remaining macros, cost 3.
2. Using the occurrence counts and the costs of the macros, the gains of the macros are recalculated. The gain of a macro is its relative savings times its occurrence count. The relative savings of a macro is the sum of the costs of its sub-instructions minus the cost of the macro itself. Thus, the savings of a macro is the increase in code size if each occurrence of a macro in the program were replaced by its sub-instructions.
3. After recalculating the gain of the macros, a fixed percentage of the macros with the lowest gain are removed. Using the new set of macros, the program is re-tiled.

There is no known method for determining a lower bound on the compression that is possible given the constraints of our code compaction problem. It is also difficult to determine how best to limit the pool of macros, Θ, considered in the search, since searching the entire set seems pointless since many macros will save only one byte.

Table 2 shows the search results for both algorithms using macro pool sizes of 12,000 and 20,000, except for the compress92 and compress95 benchmarks, whose total macro pool sizes are less then 10,000. The benchmark set consists of the standard benchmarks from SPEC CINT92 and CINT95 (SPEC CPU Benchmarks, 1992 and 1995), as well as three Philips applications. CCSSH is deterministic and of fixed complexity, scaling with the size of the benchmark and the size of the macro pool and number of collisions encountered. The results given for the GA are after 200,000 evaluations.

Using CHC and the MSX operator yields the smallest code footprint on all 14 benchmarks. The improvements range between 36 and 1368 bytes (0.8% - 4.5%) when using a macro pool of 12,000, and 480 - 2428 bytes when using a macro pool of 20,000 (2.3% - 6.7%). However, the GA takes significantly longer to run than CCSSH. In the case where millions of instances of an application will be fielded in silicon and every byte is critical the extra time taken to compress the code using the GA may be worth the savings; however, it is impractical to optimize each time a small change is made during development. Furthermore, the performance of CCSSH declines as macro pool sizes become larger, while the GA usually improves on most benchmarks when a larger macro pool is used.

The last column in Table 2 represents the advantage of the GA over CCSSH when the best results for each algorithm are chosen independent of pool size (i.e., usually a pool size of 12,000 for CCSSH and usually 20,000 for the GA). Comparisons based on these values show

that the the GA performs 1.6% - 4.9% better than CCSSH.

The results in Table 2 reflect a single run of the GA. Since the GA is a stochastic algorithm, it is important to measure the repeatability of the search. Given the evaluation time it takes to run multiple experiments, repeatability ranges have only been established for a few of the benchmarks. The total range of solutions observed for the Philips-1 benchmark is 49 bytes (i.e., solutions range from 16428 to 16477), or 3 parts per thousand.

7 CONCLUSIONS

Developing a surrogate problem that accurately models a real-world problem and saves computation time allows greater exploration of algorithms/operators than is possible when testing only on the real-world problem. The process of constructing a surrogate also contributes to better understanding of the real problem and may lead to better algorithms/operators. In this case, exploration lead to the development of MSX.

MSX is able to discover significantly better subsets than the MMX and RAR operators on problems where only an upper bound on the the size of the subset is known. MSX decreases the positional instability created by recombination when there isn't truly an ordering subproblem but rather a boundary between elements that will be evaluated in the subset and those that will not.

A strong specialized method is almost always better at solving the target problem than a GA. However, in our case CHC using MSX is able to find decoding dictionaries that yield better code compaction than CCSSH. And while the computational cost for this added performance is not insignificant, it may well be worth the time when memory savings are valuable.

References

J. Bala, K. De Jong, J. Huang, H. Vafaie, and H. Wechsler. Using Learning to Facilitate the Evolution of Features for Recognizing Visual Concepts. *Journal of Evolutionary Computation*, 4(3):297–311, 1996.

Martin Benes, Andrew Wolfe, and Steven M. Nowick. A high-speed asynchronous decompression circuit for embedded processors. In *Conf. on Advanced Research in VLSI*, September 1997.

Kelly D. Crawford, Cory J. Hoelting, Roger L. Wainwright, and Dale A. Schoenefeld. A Study of Fixed-Length Subset Recombination. In R. Belew and M. Vose, editors, *Foundations of Genetic Algorithms - 4*, pages 365–378. Morgan Kaufmann, 1997.

Larry Eshelman. The CHC Adaptive Search Algorithm. How to Have Safe Search When Engaging in Nontraditional Genetic Recombination. In G. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 265–283. Morgan Kaufmann, 1991.

Larry Eshelman, Keith Mathias, and J. David Schaffer. Convergence Controlled Variation. In R. Belew and M. Vose, editors, *Foundations of Genetic Algorithms - 4*. Morgan Kaufmann, 1997.

Christopher W. Fraser and Todd A. Proebsting. Custom instruction sets for code compression. *http://research.microsoft.com/~toddpro/papers/pldi2.ps*, 1995.

Mark Game and Alan Booker. *CodePack: Code Compression for PowerPC Processors*. International Business Machines (IBM) Corporation, 1998.

Jan Hoogerbrugge, Lex Augusteijn, Jeroen Trum, and Rik van de Wiel. A code compression system based on pipelined interpreters. *Software - Practice and Experience*, 29(11), 1999.

M. Kozuch and A. Wolfe. Compression of Embedded System Programs. *Int'l Conf. on Computer Design*, 1994.

Keith E. Mathias and L. Darrell Whitley. Changing Representations During Search: A Comparative Study of Delta Coding. *Journal of Evolutionary Computation*, 2(3):249–278, 1994.

Nicholas Radcliffe. Genetic Set Recombination. In L. Darrell Whitley, editor, *Foundations of Genetic Algorithms - 2*, pages 203–219. Morgan Kaufmann, 1993.

Michael L. Raymer, William F. Punch, Erik D. Goodman, Paul C. Sanschagrin, and Leslie A. Kuhn. Simultaneous Feature Scaling and Selection Using a Genetic Algorithm. In Thomas Bäck, editor, 7^{th} *International Conference on Genetic Algorithms*, pages 561–567. Morgan Kaufmann, 1997.

Standard Performance Evaluation Corporation. SPEC CPU92 Benchmarks. *http://www.specbench.org/osg/cpu92*, 1992.

Standard Performance Evaluation Corporation. SPEC CPU95 Benchmarks. *http://www.specbench.org/osg/cpu95*, 1995.

James A. Storer and Thomas G. Szymanski. Data compression via textural substitution. *Journal of the ACM*, 29(4):928–951, October 1982.

D. Whitley, J. R. Beveridge, C. Guerra-Salcedo, and C. Graves. Messy Genetic Algorithms for Subset Feature Selection. In Thomas Bäck, editor, 7^{th} *International Conference on Genetic Algorithms*, pages 568–575. Morgan Kaufmann, 1997.

Darrell Whitley, Keith Mathias, Soraya Rana, and John Dzubera. Evaluating Evolutionary Algorithms. *Artificial Intelligence*, 85, 1996.