

Introduction

- Phase ordering problem
 - Traditional compilers have a fixed order in which optimization phases are applied.
 - This problem can be more severe when generating code for embedded applications.
 - VISTA allows the user to finely control both the order and scope of applying optimizations.

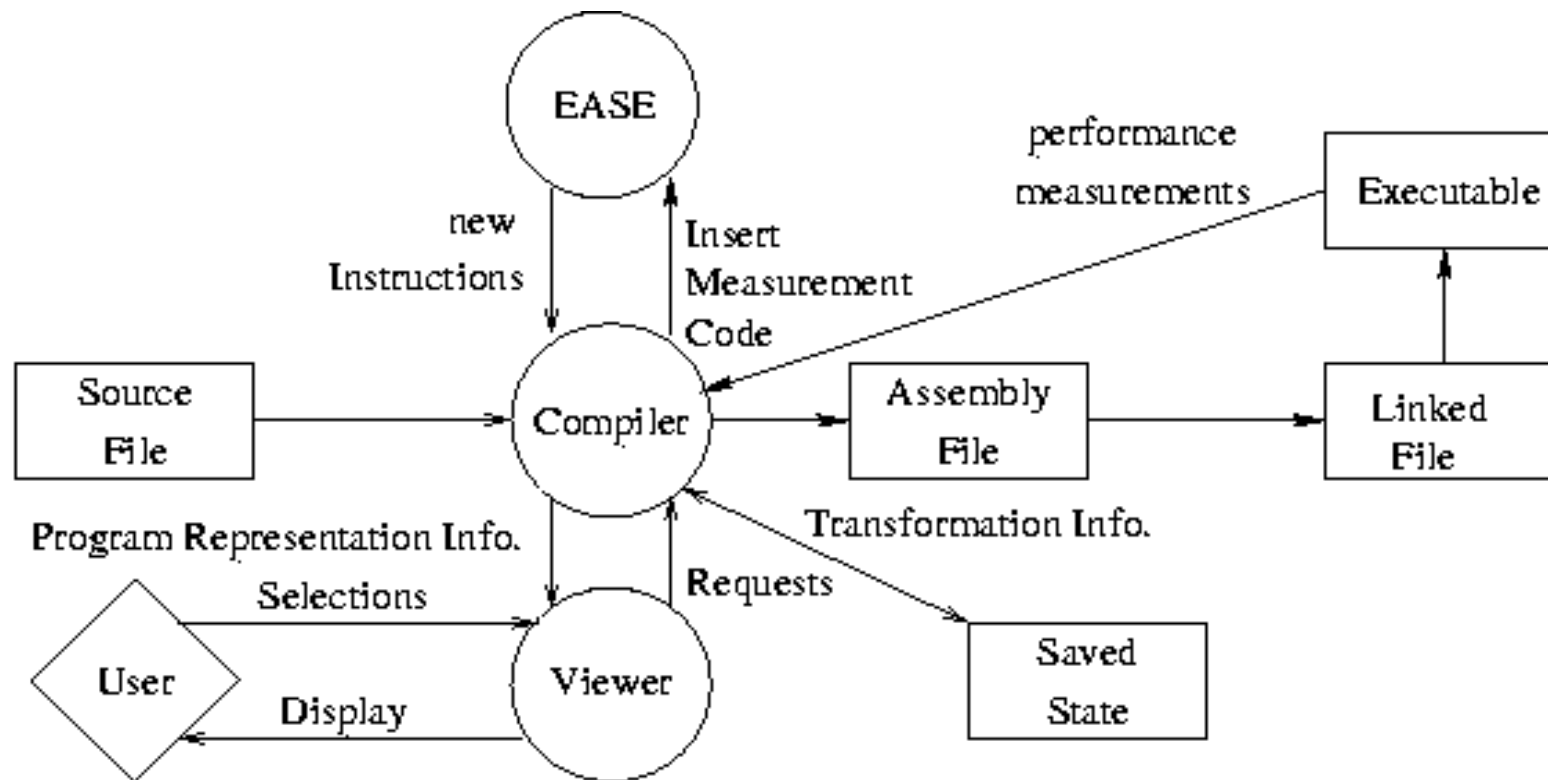
Introduction (cont...)

- Enhancing VISTA to make it more proficient at finding effective optimization sequences
 - Getting program performance measures anytime
 - Performance driven *interactive* code tuning
 - High level language like constructs to specify optimization phase orders
 - Performance driven *automatic* code tuning

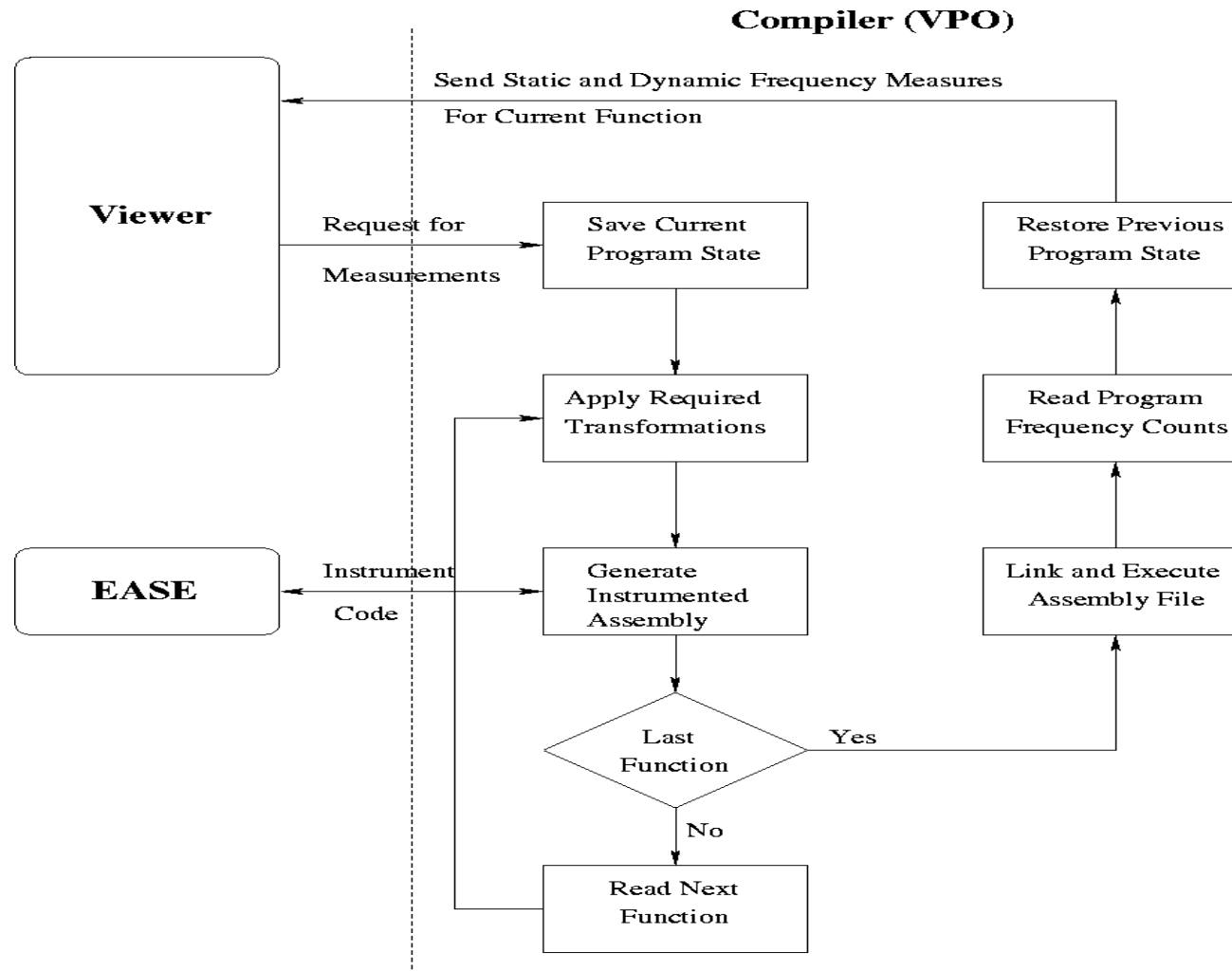
Outline of the Talk

- Overview of VISTA
- Getting performance measures in VISTA
- Support for interactive code tuning
- Support for automatic code tuning
- Experimental results
- Future work
- Conclusions

Overview of VISTA



Getting Performance Measures



Interactive Code Tuning

- VISTA provides the user with performance measures during interactive compilation.
- VISTA currently provides two types of performance counts:
 - *Static counts* – a count of the number of static instructions in that function
 - *Dynamic counts* – a count of the number of instructions executed during a particular run of the program
- VISTA also displays the execution frequency of each basic block.

Interactive Code Tuning (cont...)

- VISTA provides two options for getting measures interactively.
 - Get frequency measures
 - Start / Stop measurements

Get Frequency Measures

The screenshot shows a software interface titled "UserInterface" with a control flow graph (CFG) on the right. The CFG consists of four nodes, each containing a list of code statements and a frequency measure. The nodes are connected by arrows indicating the flow of execution.

Node 1 | L2 | freq: 48.455%

```
r[8]=HI[1en];
r[8]=r[8]+L0[1en];
r[8]=R[r[8]];
r[9]=r[30]+.10.0_i;
r[9]=R[r[9]];
r[10]=2;
r[9]=r[9]{r[10];
r[10]=HI[table];
r[10]=r[10]+L0[table];
r[9]=r[9]+r[10];
R[r[9]]=r[8];
```

Node 1 | L3 | freq: 26.43%

```
r[8]=r[30]+.10.0_i;
r[8]=R[r[8]];
r[9]=1;
r[8]=r[8]+r[9];
r[9]=r[30]+.10.0_i;
R[r[9]]=r[8];
```

Node 1 | L5 | freq: 22.111%

```
r[8]=r[30]+.10.0_i;
r[8]=R[r[8]];
r[9]=255;
IC=r[8]?r[9];
PC=ICs0,L2;
```

Node 1 | | freq: 0.068%

```
r[8]=0;
r[9]=r[30]+.10.0_i;
R[r[9]]=r[8];
PC=L9;
```

The interface also includes a table of transformations, a control panel with buttons for "start writing in", "execute from file", and "Option", and a message box at the bottom stating: "Warning !!! Discard the transformations will lose information done".

Function	Trans Number	State	Total
init_search	9	after	9

transformations	Number	Code Size	Inst. Exec.
Register Assignment	9	100.00	

Message: Warning !!! Discard the transformations will lose information done

Start / Stop Measurements

The screenshot displays a software interface with a table of compilation statistics on the left and a control flow graph on the right.

Compilation Statistics Table:

Function	State	Trans Number	Total	
init_search	after	256	256	
transformations				
	Number	Code Size	Inst. Exec.	
Register Assignment	9	100.00	100.00	
Inst Selection	66	59.52	59.18	
Merge Basic Blocks	3	59.52	59.18	
Register Allocation	57	59.52	59.18	
Dead Variable Elim	32	59.52	59.18	
Common Subexpr Elim	15	40.47	40.90	
Code Motion	3	45.23	40.97	
Inst Selection	24	40.47	32.00	
Register Allocation	24	40.47	32.00	
Strength Reduction	20	40.47	32.00	
Fix Entry Exit	(3)	3	42.85	32.04

Control Flow Graph:

- Block 1:** | freq: 0.483%
r[14]=SV[r[14]+-96];
r[8]=r[24];
ST=HI[strlen]+L0[strlen];
r[9]=HI[1en];
R[r[9]+L0[1en]]=r[8];
r[11]=0;
r[12]=HI[1en];
r[13]=HI[table];
PC=L5;
- Block 2:** | L2 | freq: 68.736%
r[8]=R[r[12]+L0[1en]];
r[9]=r[11]{2};
r[10]=r[13]+L0[table];
R[r[9]+r[10]]=r[8];
r[11]=r[11]+1;
- Block 3:** | L5 | freq: 27.601%
IC=r[11]?255;
PC=ICs0,L2;
- Block 4:** | freq: 0.214%
r[11]=0;
r[12]=HI[1en];
r[13]=HI[table];

Interface Controls:

- Buttons: |< << < > >> >|
- Buttons: Option Exit
- Buttons: start writing in seq1.txt execute from file
- Message: Warning !!! Discard the transformations will lose information done

Interactive Code Tuning (cont...)

- Control Statements in VISTA
 - High-level programming language like constructs are used in VISTA to conditionally invoke an optimization phase.
 - if-changes-else
 - if-changes-then-else
 - do-while-changes
 - while-changes-do

Automatic Code Tuning

- The previous approach requires user knowledge, intuition and effort to guide the code improvement process.
- We provided two new constructs in VISTA to support automatic code tuning
 - select best sequence
 - select best combination

Select Best Sequence

- The user selects two or more different optimization sequences.
- Each sequence is evaluated by the compiler for its performance.
- The user can specify weights for static and dynamic counts to determine the overall improvement.
- The best performing sequence is found and re-applied by the compiler.

Select Best Combination

- The user specifies a set of optimization phases.
- The compiler tries to determine the best ordering of this sequence of phases.
- The compiler forms different combinations of phases.
- Each is evaluated for performance, depending on weights specified by the user.
- Only the best performing sequence is re-applied.

Select Best Combination (cont...)

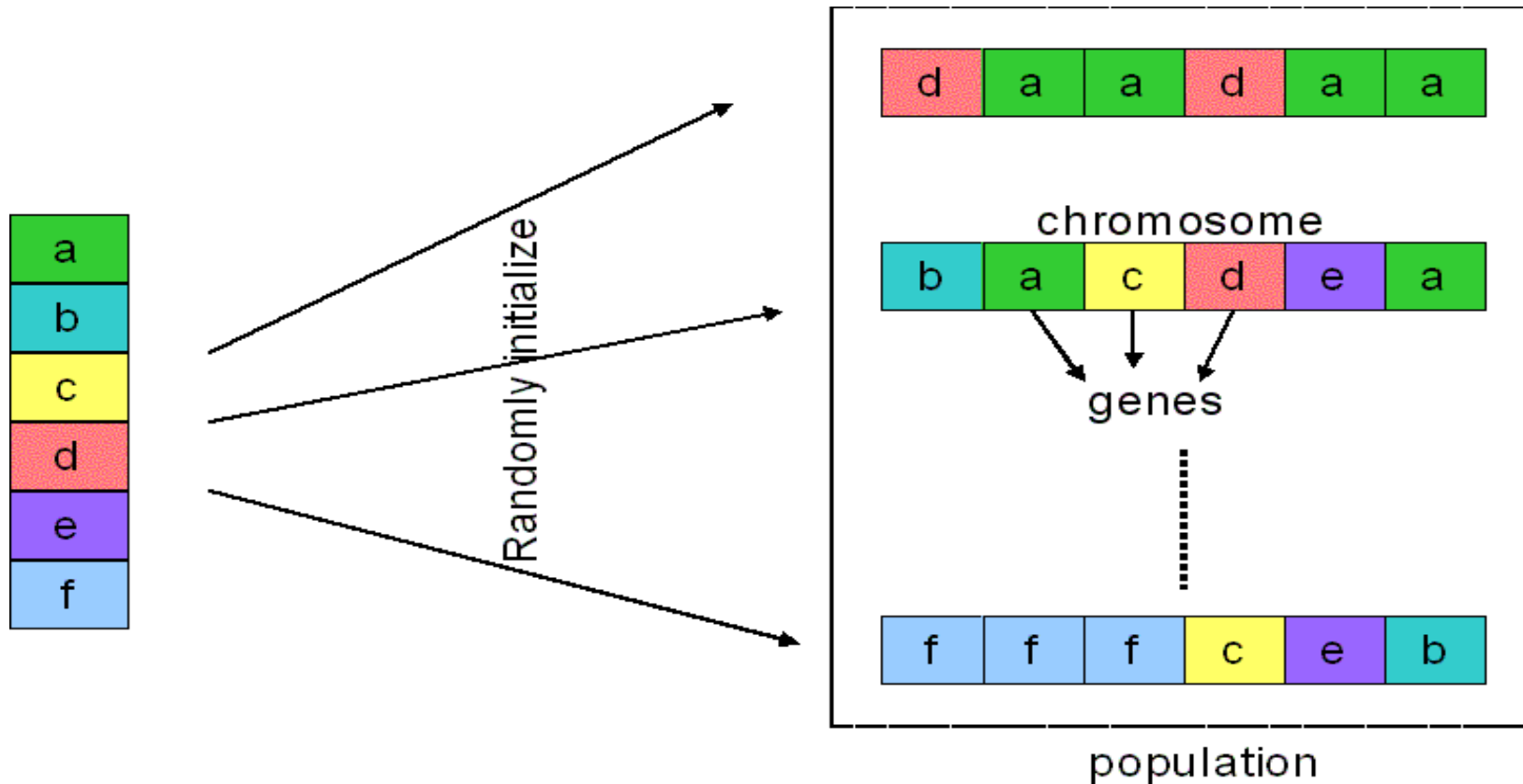
- The compiler finds the next combination to evaluate based on the search option specified by the user.
- Search options
 - Exhaustive search – All possible combinations are attempted by the compiler
 - Biased sampling search – Compiler uses a genetic algorithm to probe the search space for an effective sequence
 - Permutation Search – Compiler attempts to evaluate all permutations of the specified length

Genetic Algorithms

- These are search algorithms designed to mimic the process of natural selection and evolution in nature.
- Some genetic algorithm terms
 - Chromosome – optimization sequence
 - Gene – individual optimization phase in a sequence
 - Population – set of chromosomes
 - Fitness value – performance of that optimization sequence
 - Crossover – combination of sequences to form new sequences
 - Mutation – individual phases in a sequence are replaced
 - Generation – time step for evaluation of sequences in one population and formation of the next population

Genetic Algorithm Used

- Initialization of first population
 - The first population of optimization sequences is randomly generated

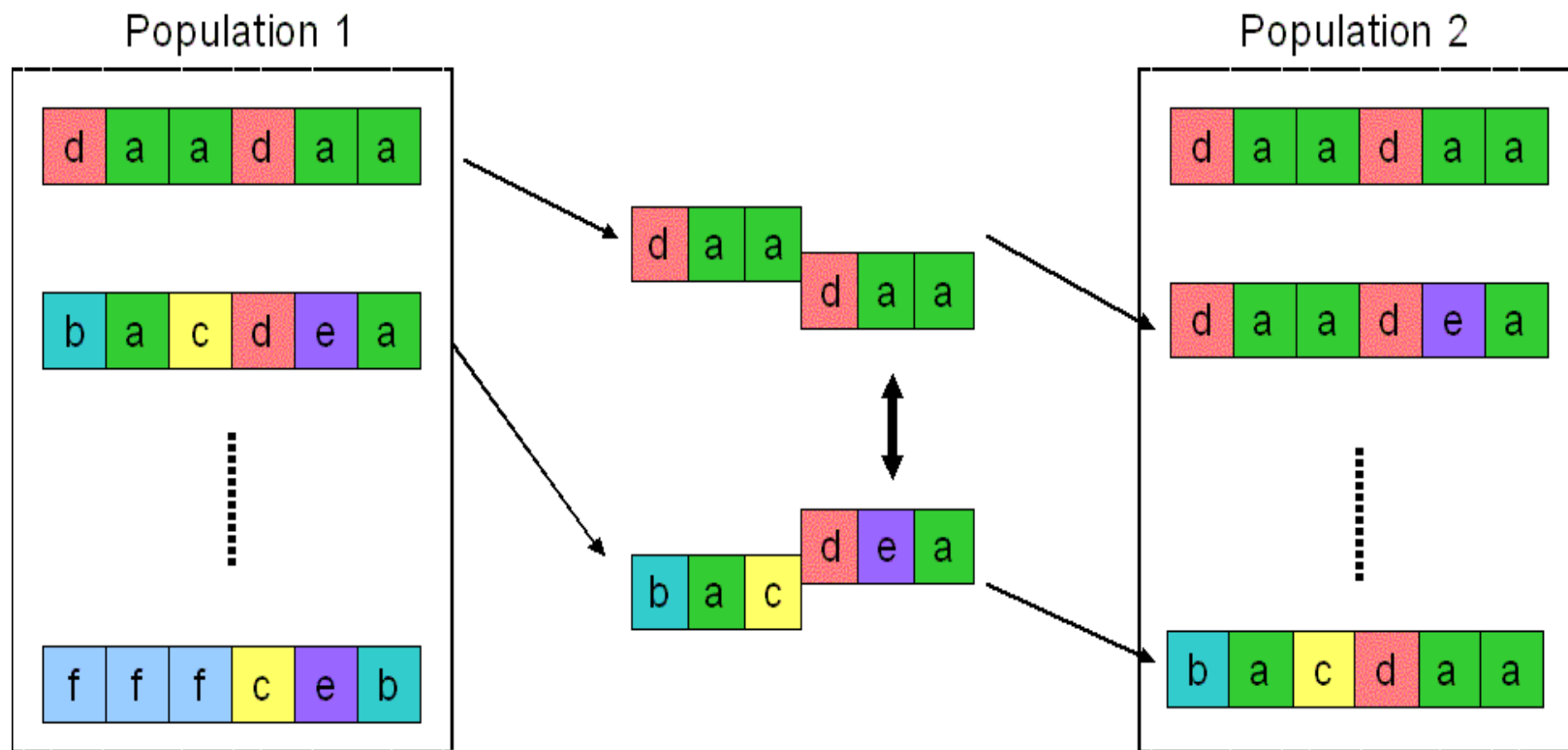


Genetic Algorithm Used (cont...)

- The performance of each sequence in the population is evaluated.
- The chromosomes are sorted based on performance.
- The population is divided into two halves.
- Some chromosomes from the poorly performing half are deleted.
- The vacancies are filled using the crossover and mutation operation.

Genetic Algorithm Used (cont...)

- Crossover operation
 - upper half of the first chromosome is combined with lower half of the second and vice-versa.



Genetic Algorithm Used (cont...)

- The chromosomes are subjected to mutation.
- The best performing chromosome over all the generations is maintained.

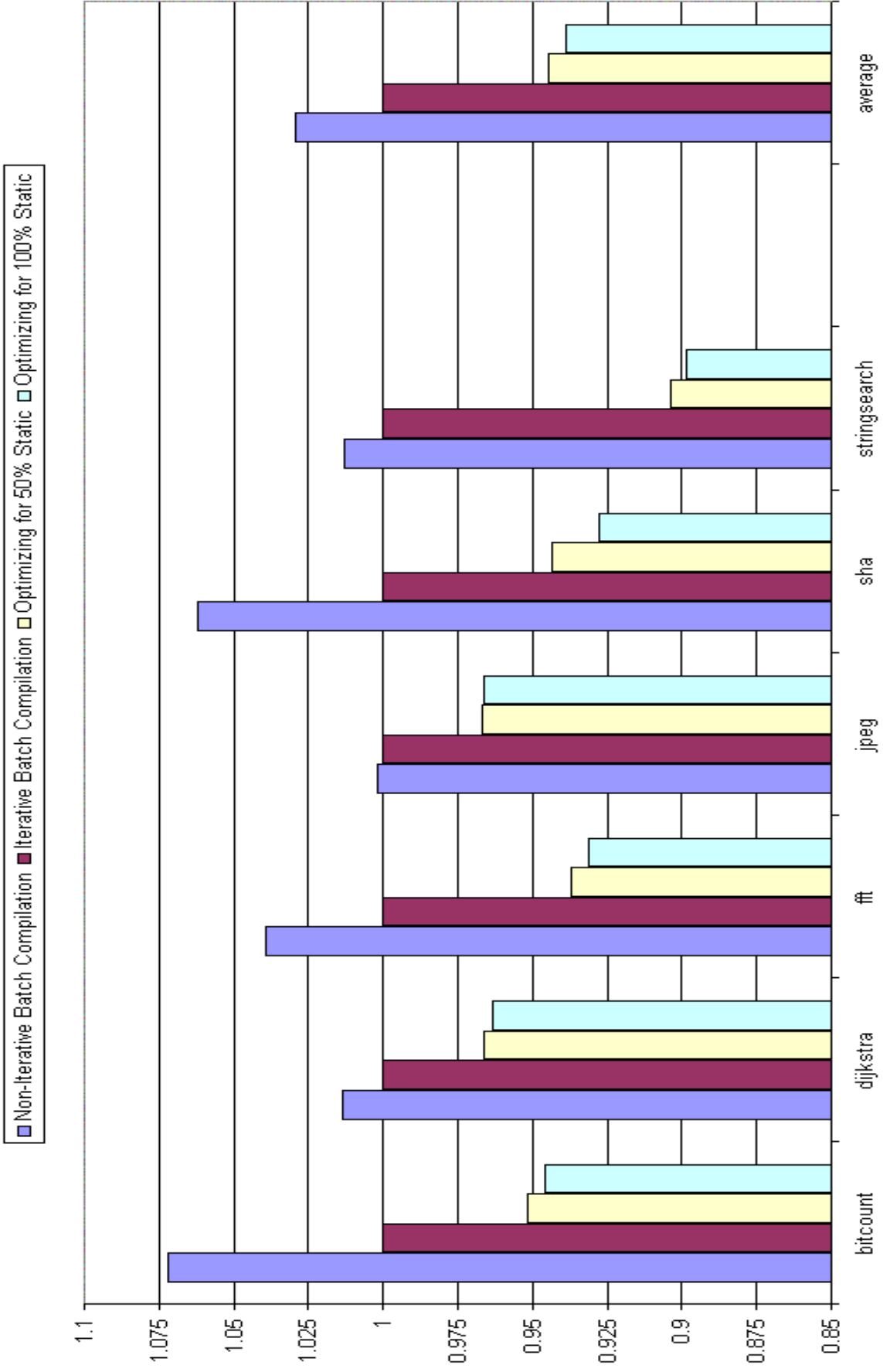
Experimental Results

- A set of experiments were conducted to illustrate the effectiveness of using VISTA's biased sampling search.
- The experiments were conducted on a set of *mibench* programs.
- The target architecture for the experiments was the SPARC.
- The genetic algorithm was used to find the best sequence among 14 phases between *register assignment* and *fix entry exit*.
- The *sequence length* was set to 1.25 times the length of sequence applied during batch compilation.

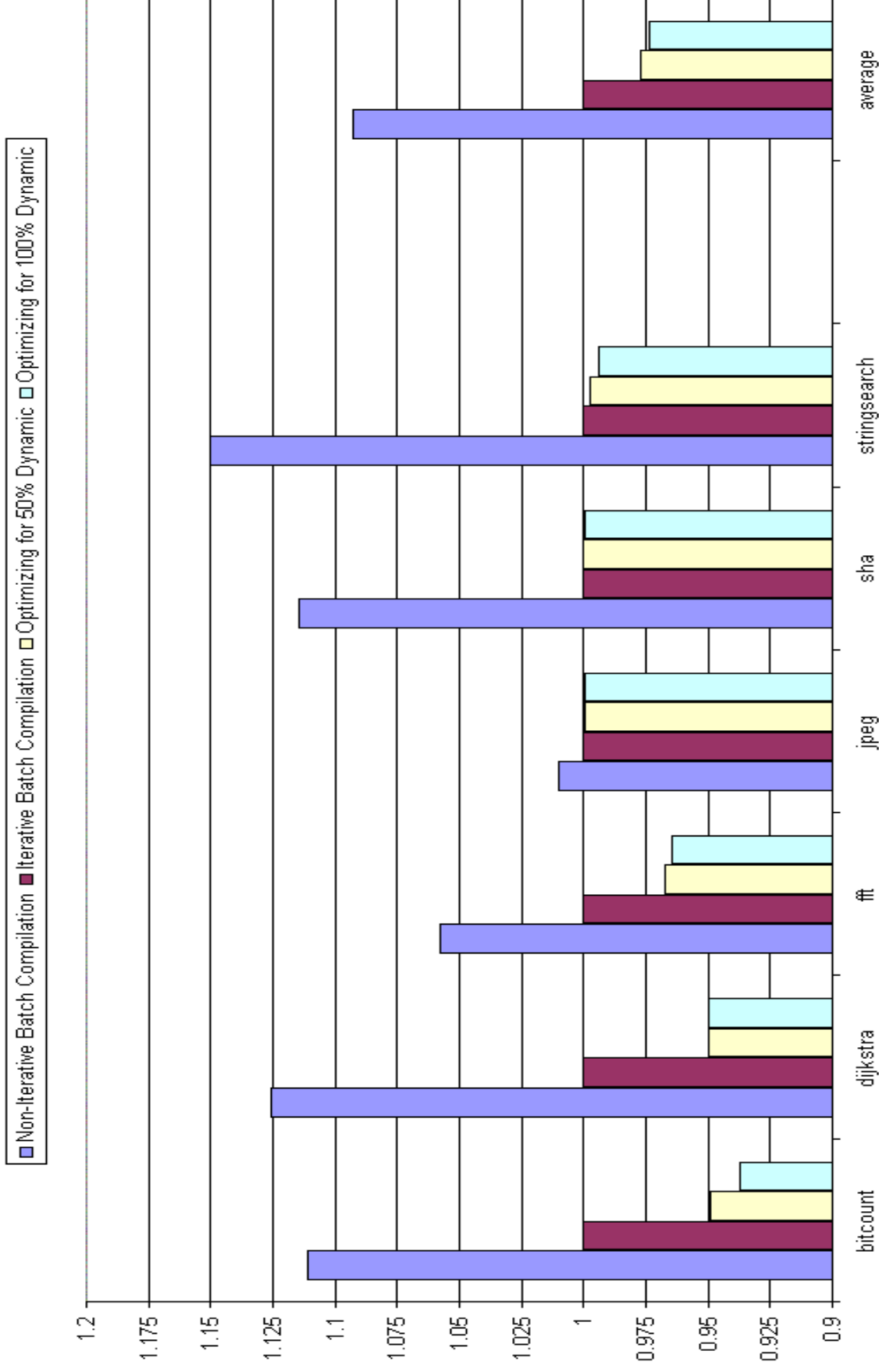
Experimental Results (cont...)

- Interactive compilation measures
 - An attempt was made to find an optimization sequence giving equal or better performance than that given by the batch compiler.
 - Genetic algorithm was used to probe the search space.
 - The population size was fixed at 20.
 - The algorithm was repeated for 100 generations.
 - Results were obtained for 3 different criteria, static count only, dynamic count only and 50% for each factor.

Overall Effect on Static Instruction Count



Overall Effect on Dyanmic Instruction Count



Future Work

- Obtaining measurements on a real embedded systems architecture
- Getting a more accurate measure of the dynamic performance
- Study the effect of varying the parameters in the genetic algorithm
- Study the result of performing genetic algorithm searches on sets of basic blocks in a function

Conclusion

- We have developed an interactive compilation system that automatically provides performance feedback information.
- Structured constructs are provided for specifying optimization sequences interactively.
- Constructs are provided to automatically select optimization phase sequences.
- Experiments were performed to illustrate the effectiveness of using a genetic algorithm to search for effective optimization sequences.

