Fast Searches for Effective Optimization Phase Sequences

Prasad Kulkarni[1], Stephen Hines[1], Jason Hiser[2]
David Whalley[1], Jack Davidson[2], Douglas Jones[3]

[1] *Computer Science Department, Florida State University, Tallahassee, Florida*

[2] *Computer Science Department, University of Virginia, Charlottesville, Virginia*

[3] *Electrical and Computer Eng. Department, University of Illinois, Urbana, Illinois*

Phase Ordering Problem

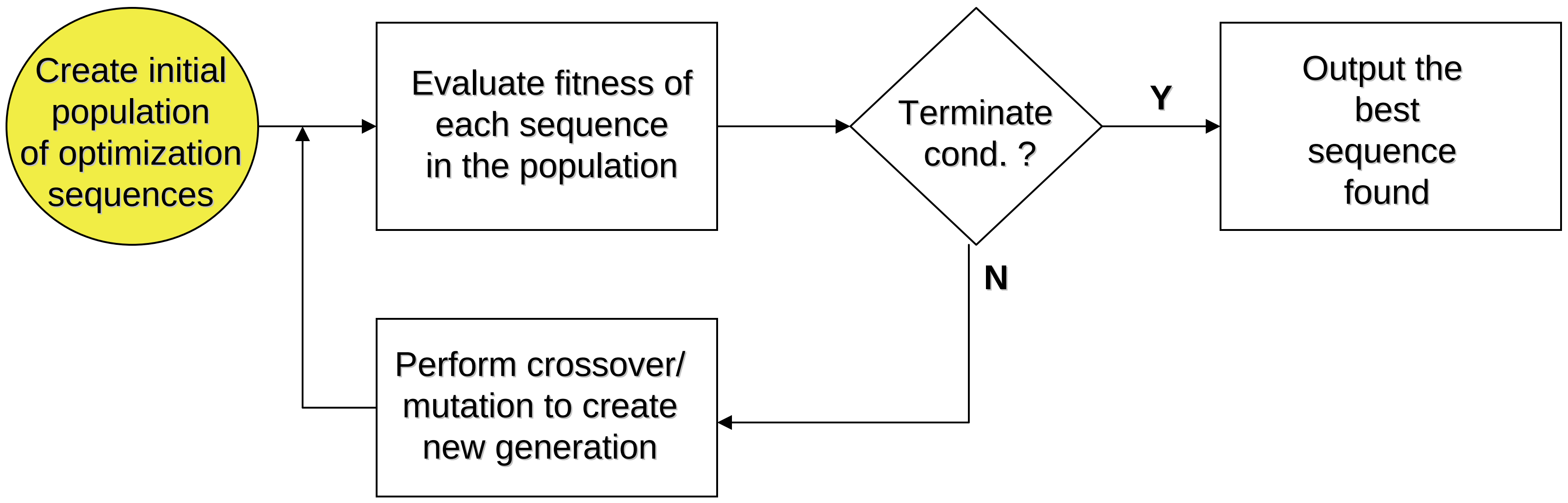
- A single ordering of optimization phases will not always produce the best code
  - – different applications
  - – different compilers
  - – different target machines
- Example
  - – *register allocation* and *instruction selection*

Approaches to Addressing the Phase Ordering Problem

- Framework for formally specifying compiler optimizations.
- Single intermediate language representation
  - repeated applications of optimization phases
- Exhaustive search?
- Our approach
  - intelligent search of the optimization space using genetic algorithm

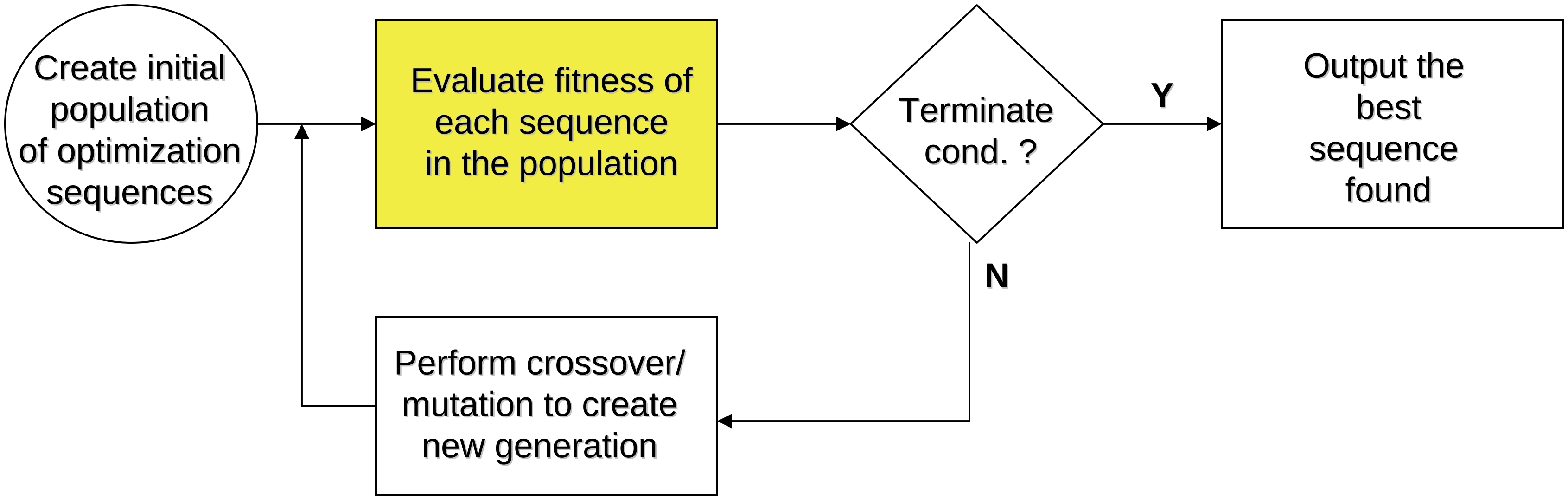
Genetic Algorithm

- A biased sampling search method
  - evolves solutions by merging parts of different solutions


Create initial population of optimization sequences
Evaluate fitness of each sequence in the population
Terminate cond. ?
Y
Output the best sequence found
N
Perform crossover/ mutation to create new generation

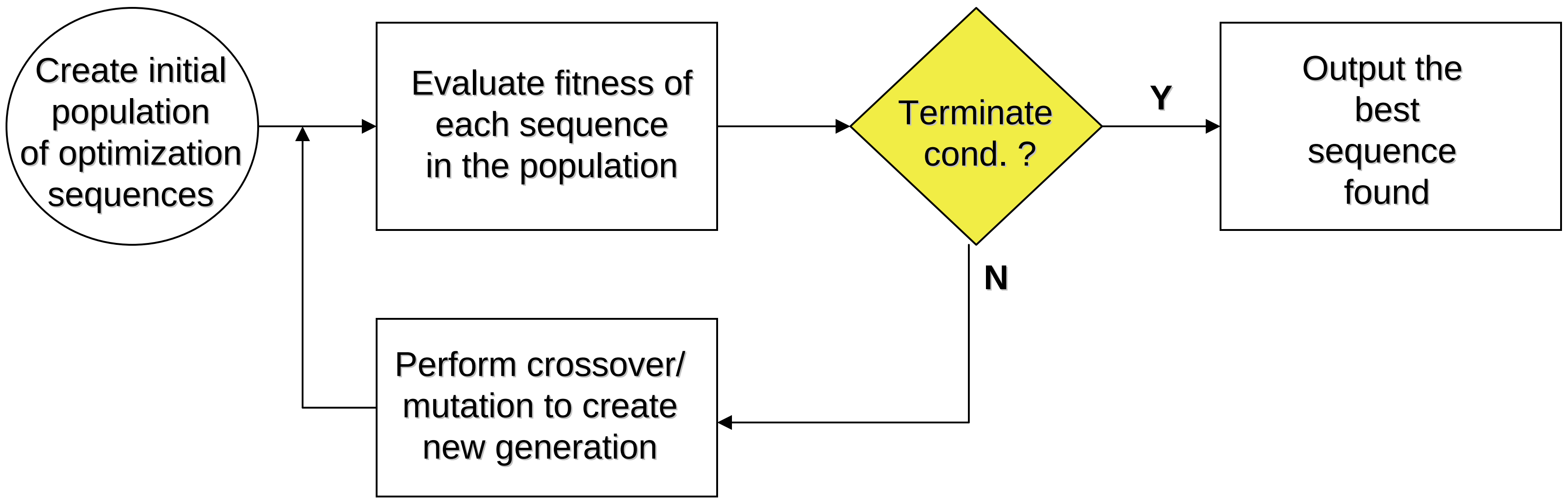
Genetic Algorithm

- A biased sampling search method
  - evolves solutions by merging parts of different solutions


Create initial population of optimization sequences
Evaluate fitness of each sequence in the population
Terminate cond. ?
Y
Output the best sequence found
N
Perform crossover/ mutation to create new generation

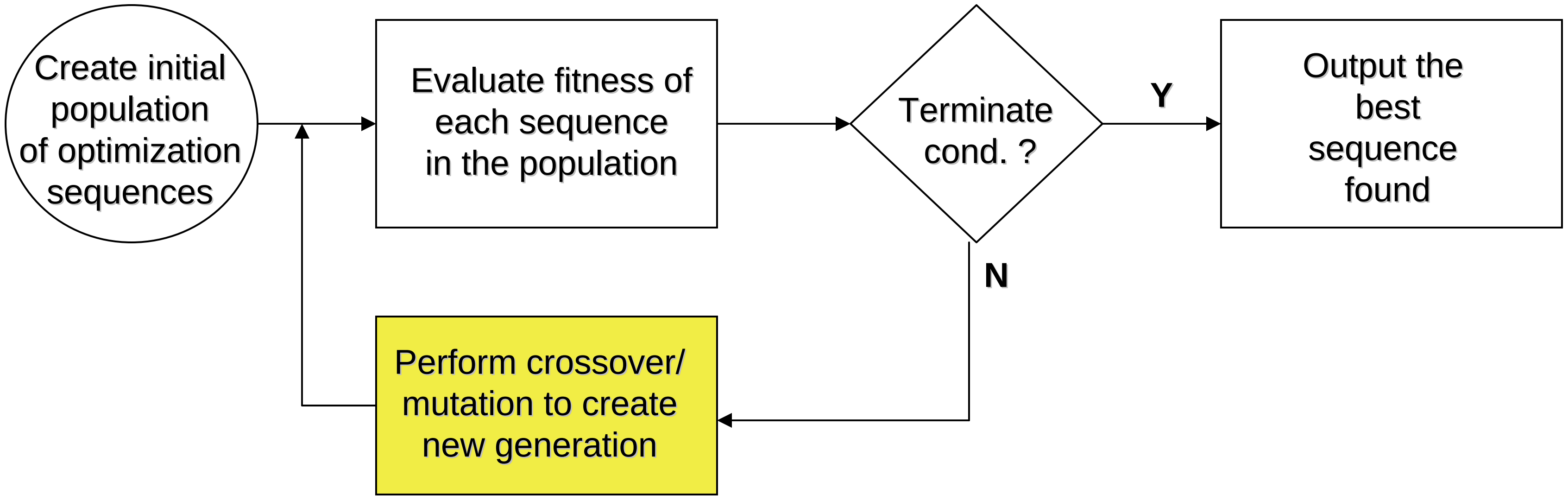
Genetic Algorithm

- A biased sampling search method
  - evolves solutions by merging parts of different solutions


Create initial population of optimization sequences
Evaluate fitness of each sequence in the population
Terminate cond. ?
Y
Output the best sequence found
N
Perform crossover/ mutation to create new generation

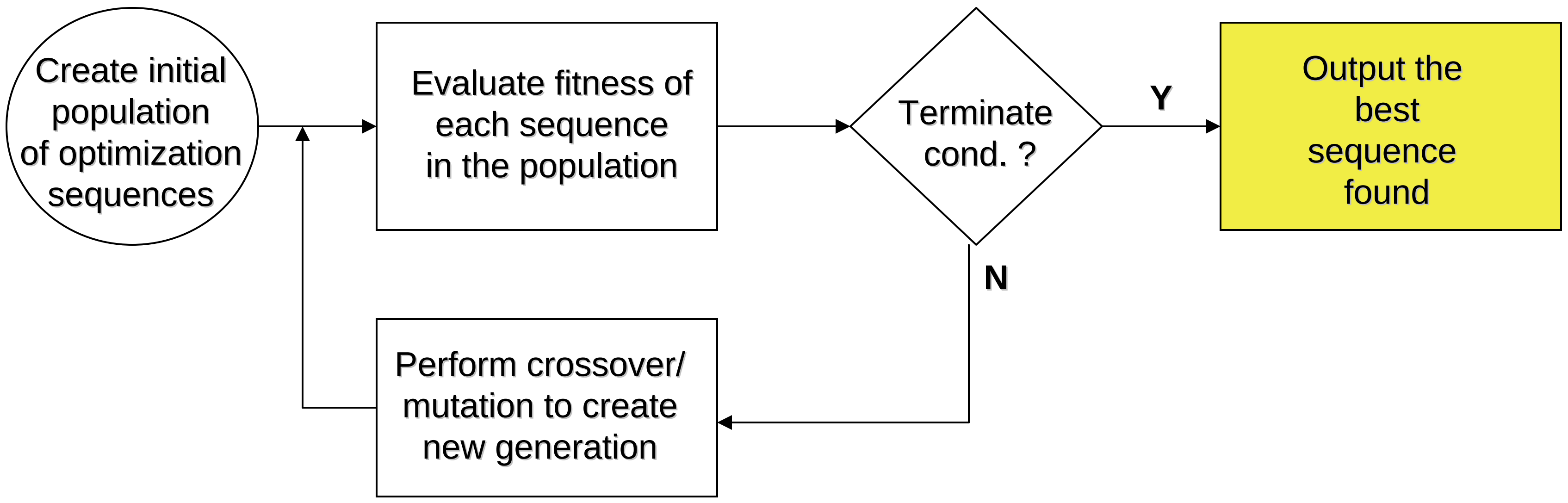
Genetic Algorithm

- A biased sampling search method
  - evolves solutions by merging parts of different solutions


Create initial population of optimization sequences
Evaluate fitness of each sequence in the population
Terminate cond. ?
Y
Output the best sequence found
N
Perform crossover/ mutation to create new generation

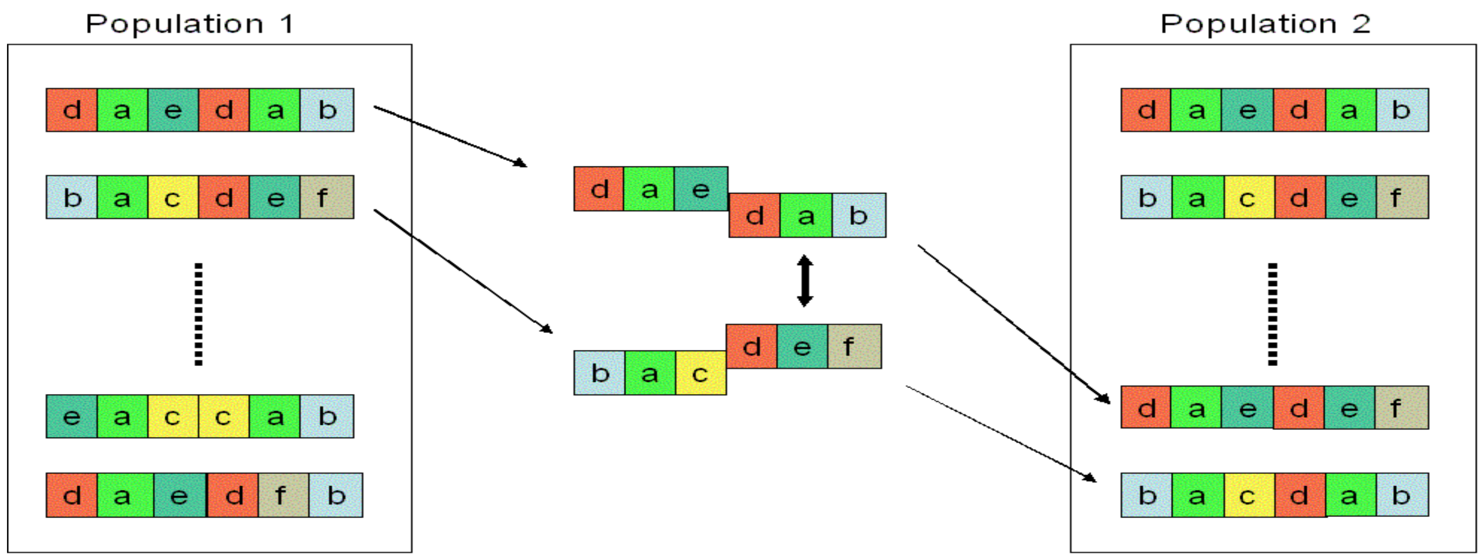
Genetic Algorithm

- A biased sampling search method
  - evolves solutions by merging parts of different solutions


Create initial population of optimization sequences
Evaluate fitness of each sequence in the population
Terminate cond. ?
Y
Output the best sequence found
N
Perform crossover/ mutation to create new generation

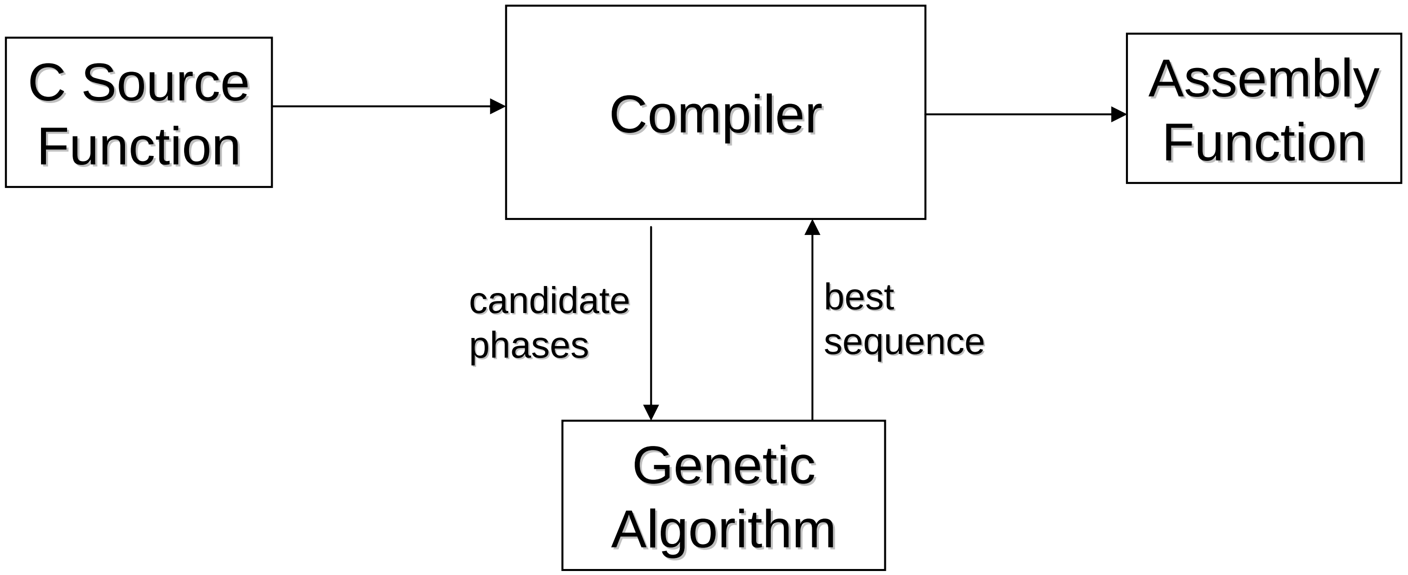
Genetic Algorithm (cont…)

- Crossover
  - 20% sequences in each generation replaced



- Mutation
  - phases in each sequence replaced with a low probability

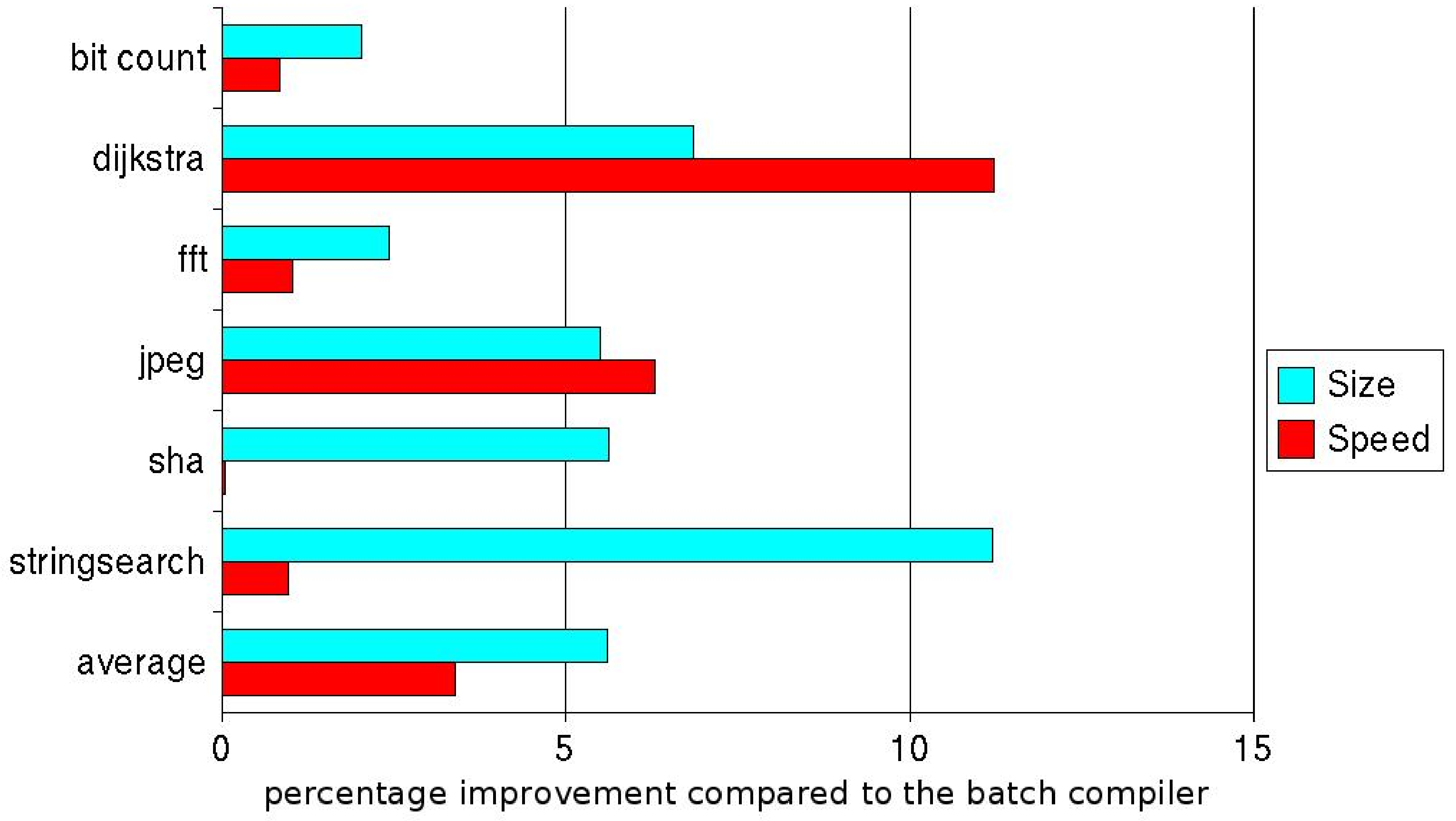
Genetic Algorithm (cont…)



Experiments

- Performed on six mibench benchmarks, which contained a total of 106 functions.
- Used 15 candidate optimization phases.
- Sequence length set to 1.25 times the number of successful batch phases.
- Population size set to 20.
- Performed 100 generations.
- Fitness value was 50% speed and 50% size.

Genetic Algorithm – Results



Our Earlier Work

- Published in LCTES ’03
  - – complete compiler framework
  - – detailed description of the genetic algorithm
  - – improvements given by the genetic algorithm for code-size, speed, and 50% of both factors
  - – optimization sequences found by the genetic algorithm for each function
  - – *Finding Effective Optimization Phase Sequences – http://www.cs.fsu.edu/~whalley/papers/lctes03.ps*

Genetic Algorithm – Issues

- Very long search times
  - – evaluating each sequence involves compiling, assembling, linking, execution and verification
  - – simulation / execution on embedded processors is generally slower than general-purpose processors
- Reducing the search overhead
  - – avoiding redundant executions of the application.
  - – modifying the search to obtain comparable results in fewer generations.

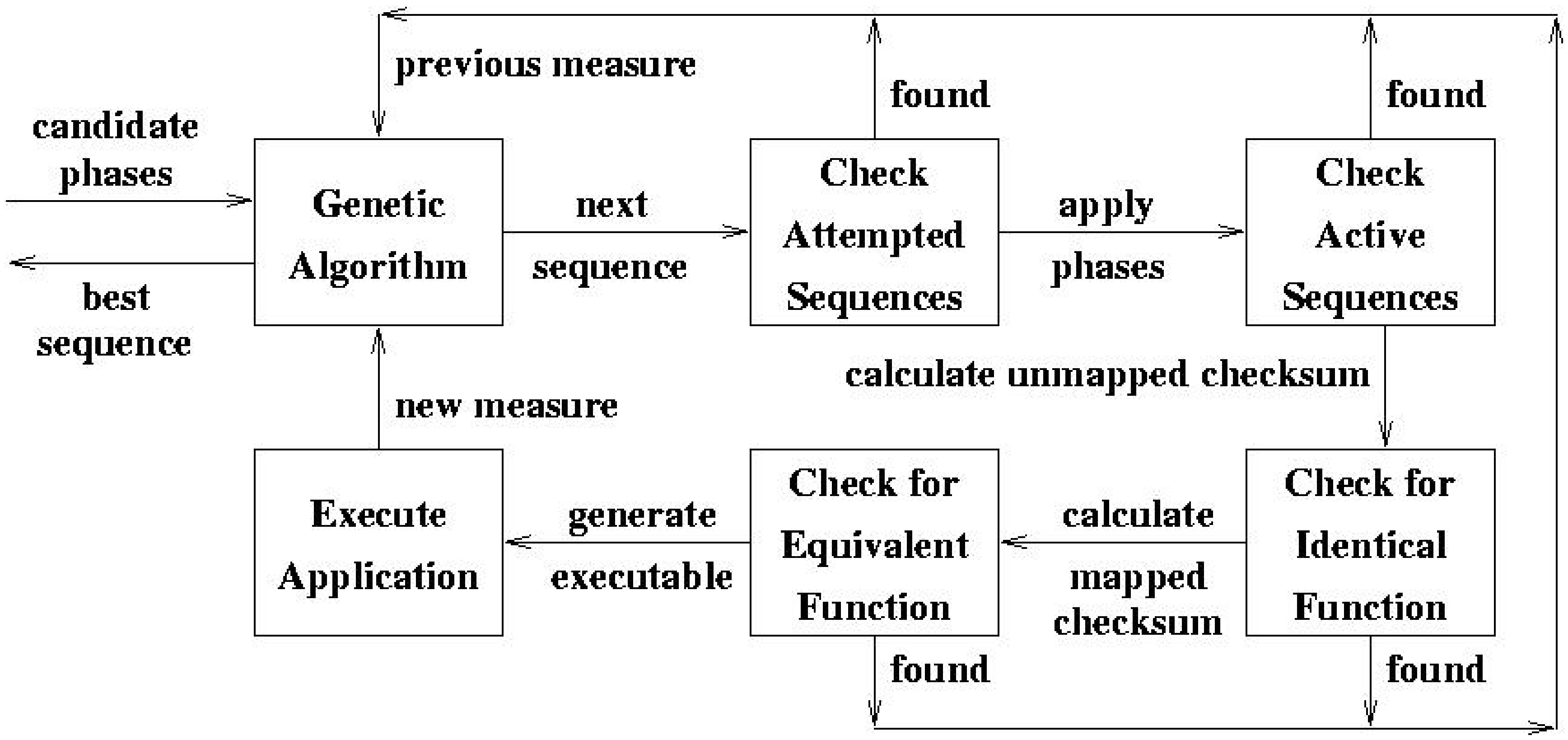
Methods for Avoiding Redundant Executions

- Detect sequences that have already been attempted.
- Detect sequences of phases that have been successfully applied.
- Check if an instance of this function has already been generated.
- Check if an equivalent function has already been generated.

Reducing the Search Overhead

- Avoiding redundant executions.
- Obtaining similar results in fewer generations.

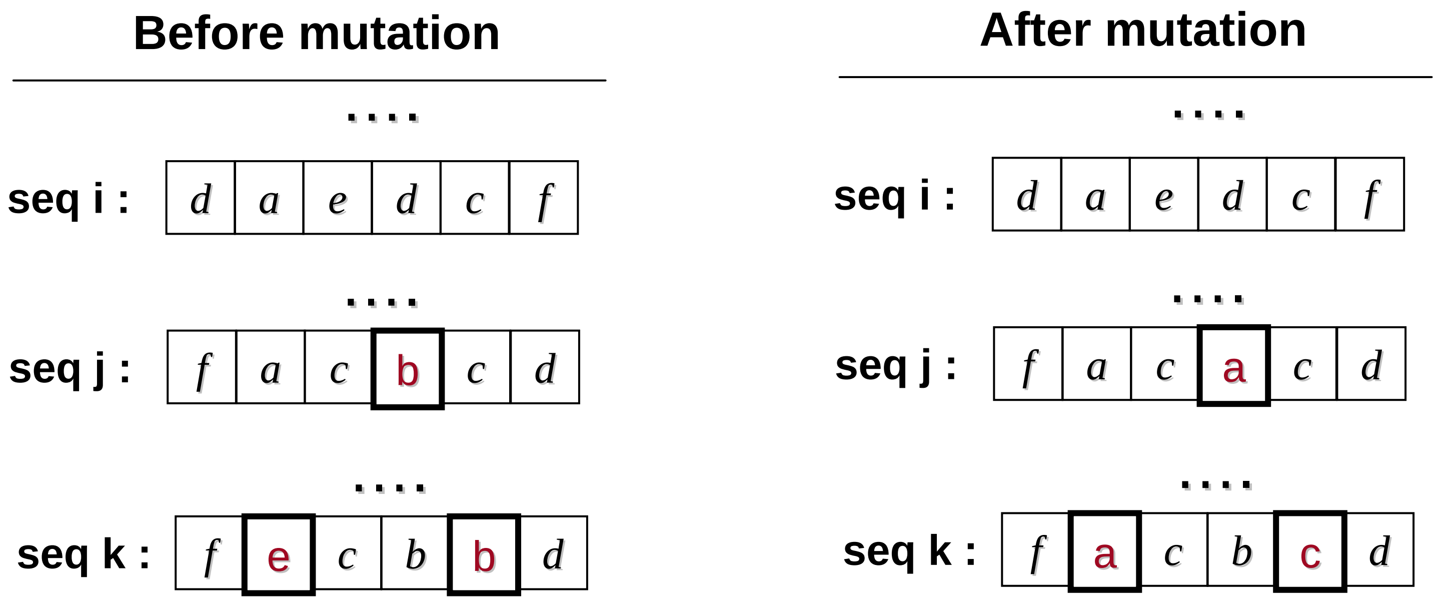
Overview of Avoiding Redundant Executions



Finding Redundant Attempted Sequences

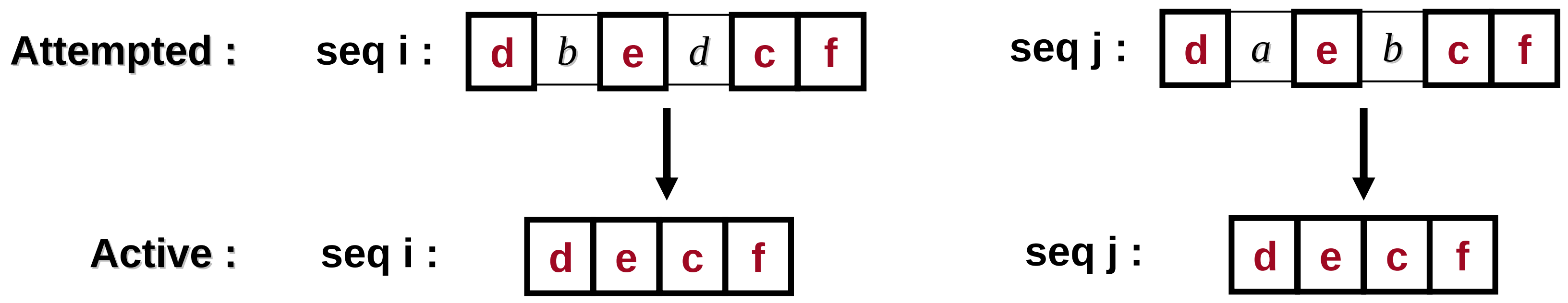
- Same optimization phase sequence may be reattempted
  - Crossover operation producing a previously attempted sequence
  - Mutation not occurring on any of the phases in the sequence
  - Mutation changing phases, but producing a previously attempted sequence

Finding Redundant Attempted Sequences (cont…)



Finding Redundant Active Sequences

- An active optimization phase is one that is able to complete one or more transformations.
- Dormant phases do not affect the compilation.
- Compiler must indicate if phase was active.



Detecting Identical Code

- Sometimes identical code for a function can be generated from different active sequences.
- Some phases are essentially independent
  - – branch chaining and register allocation
- Sometimes more than one way to produce the same code.

Detecting Identical Code (cont…)

- Example:

```
r[2] = 1;
r[3] = r[4] + r[2];

⇒instruction selection
r[3] = r[4] + 1;
```

```
r[2] = 1;
r[3] = r[4] + r[2];

⇒constant propagation
r[2] = 1;
r[3] = r[4] + 1;

⇒dead assignment elimination
r[3] = r[4] + 1;
```

- Used CRC checksums to compare function instances.

Detecting Equivalent Code

- Code generated by different optimization sequences may be equivalent, but not identical.
- Some optimization phases consume registers.
- Different ordering of such phases may result in equivalent instructions, but different registers being used.

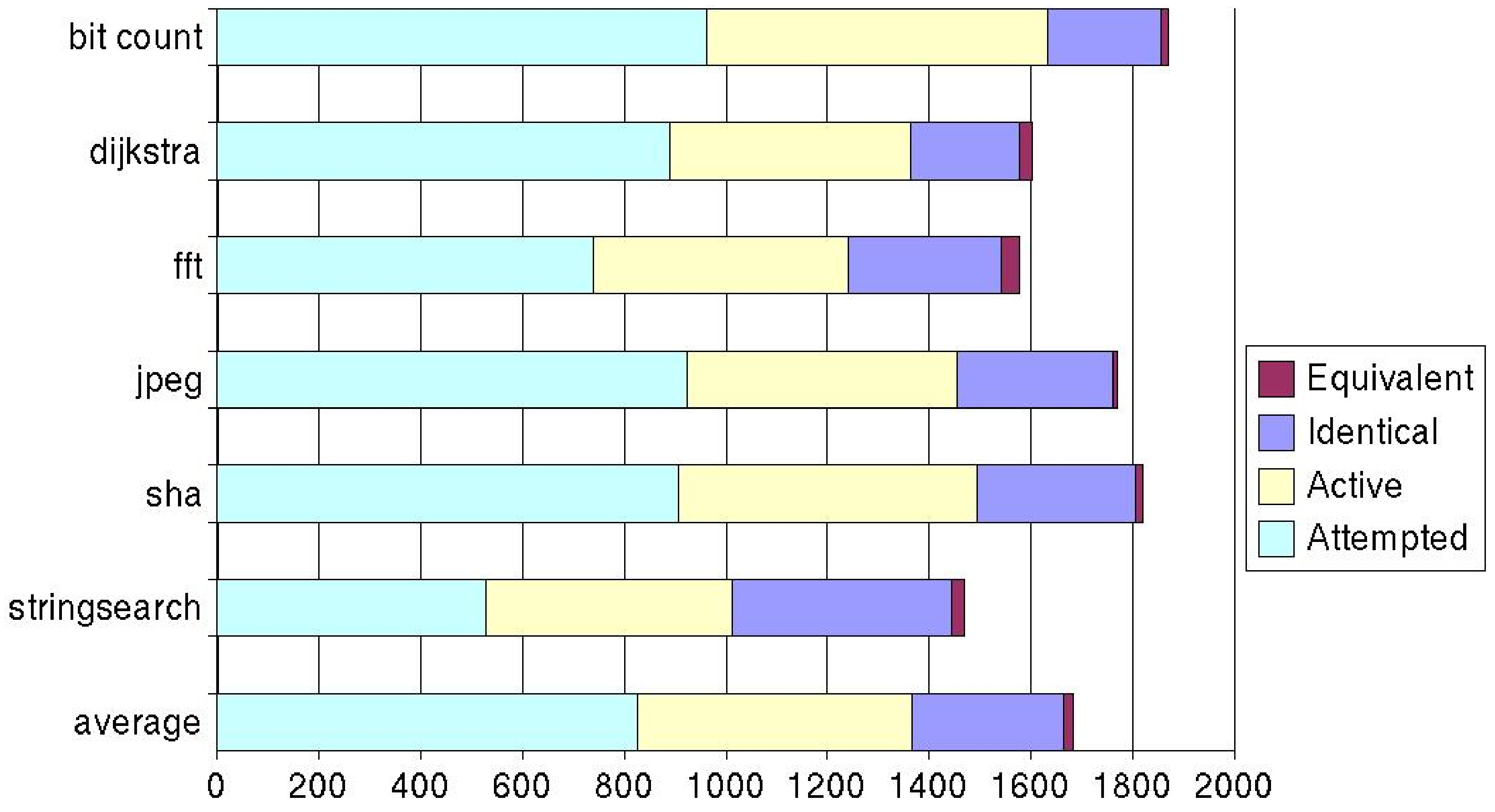
Detecting Equivalent Code (cont…)

```
sum = 0;
for (i = 0; i < 1000; i++ )
     sum += a [ i ];
```

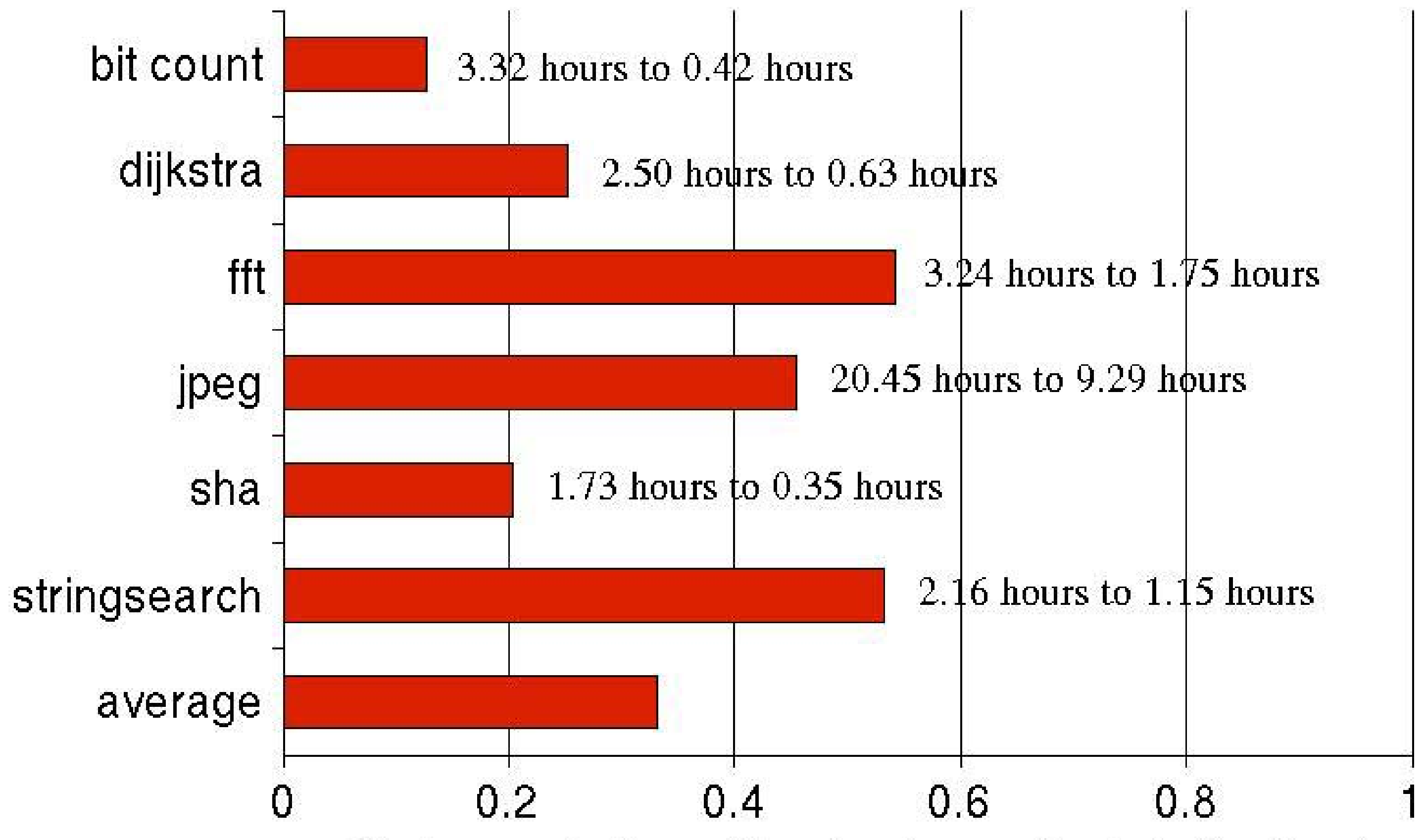
Source Code

r[10]=0;
r[12]=HI[a];
r[12]=**r[12]**+LO[a];
r[1]=**r[12]**;
r[9]=4000+**r[12]**;
L3
r[8]=M[r[1]];
r[10]=**r[10]**+r[8];
r[1]=r[1]+4;
IC=r[1]?r[9];
PC=IC<0,L3;
 	
Register Allocation before Code Motion
	r[11]=0;
	r[10]=HI[a];
	r[10]=**r[10]**+LO[a];
	r[1]=**r[10]**;
	r[9]=4000+**r[10]**;
	L3
	r[8]=M[r[1]];
	r[11]=**r[11]**+r[8];
	r[1]=r[1]+4;
	IC=r[1]?r[9];
	PC=IC<0,L3;
	
	Code Motion before Register Allocation
		r[32]=0;
		r[33]=HI[a];
		r[33]=**r[33]**+LO[a];
		r[34]=**r[33]**;
		r[35]=4000+**r[33]**;
		L3
		r[36]=M[r[34]];
		r[32]=**r[32]**+r[36];
		r[34]=r[34]+4;
		IC=r[34]?r[35];
		PC=IC<0,L3;
		
		After Mapping Registers

Number of Avoided Executions



Relative Total Search Time



Reducing the Search Overhead

- Avoiding redundant executions.
- Obtaining similar results in fewer generations.

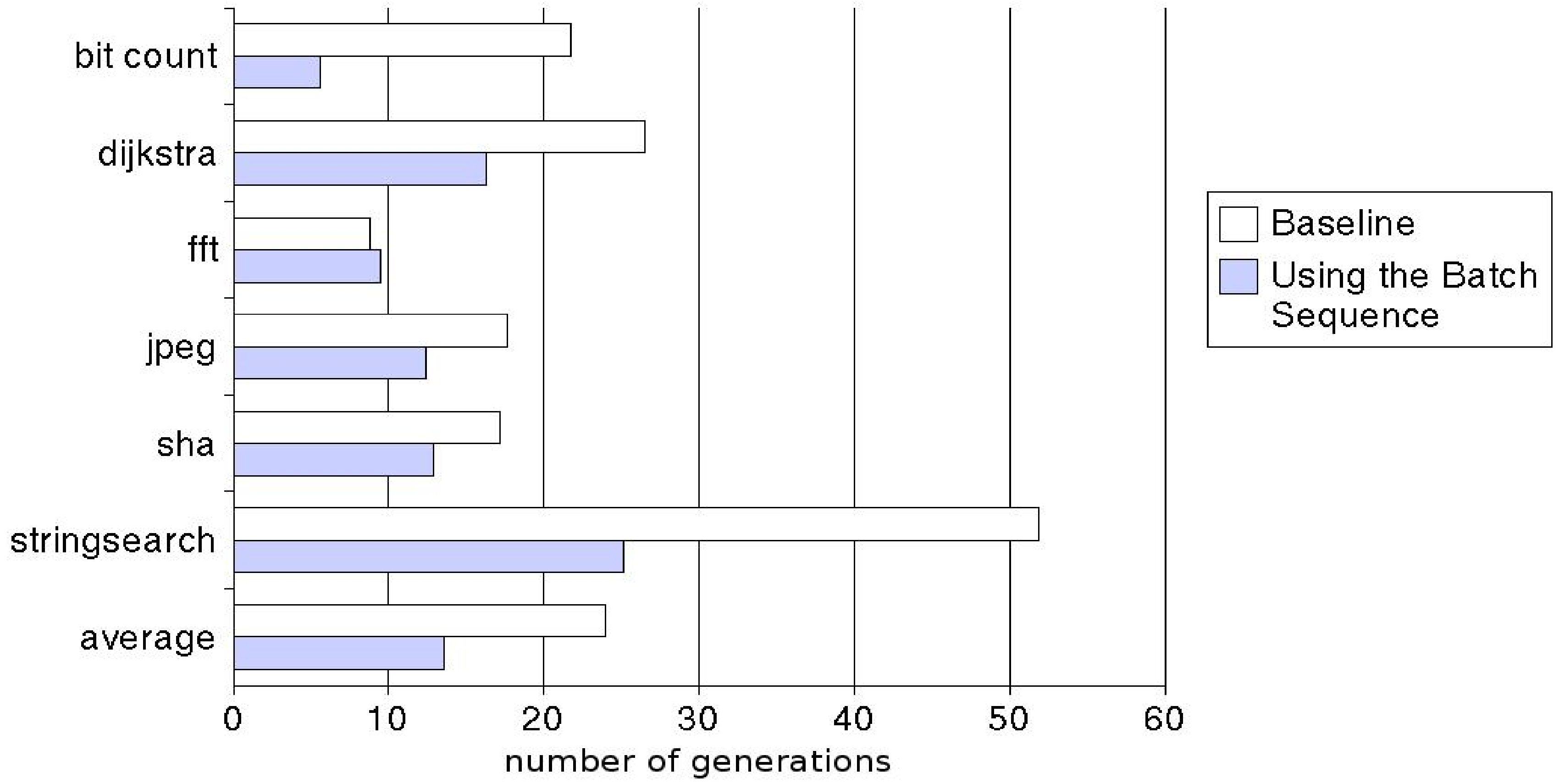
Producing Similar Results in Fewer Generations

- Can reduce search time by running the genetic algorithm for fewer generations.
- Can obtain better results in the same number of generations.
- We evaluate four methods for reducing the number of required generations to find the best sequence in the search.

Using the Batch Sequence

- Capture the active sequence of phases applied by the batch compiler.
- Place this sequence in the initial population.
- May allow the genetic algorithm to converge faster to the best sequence it can find.

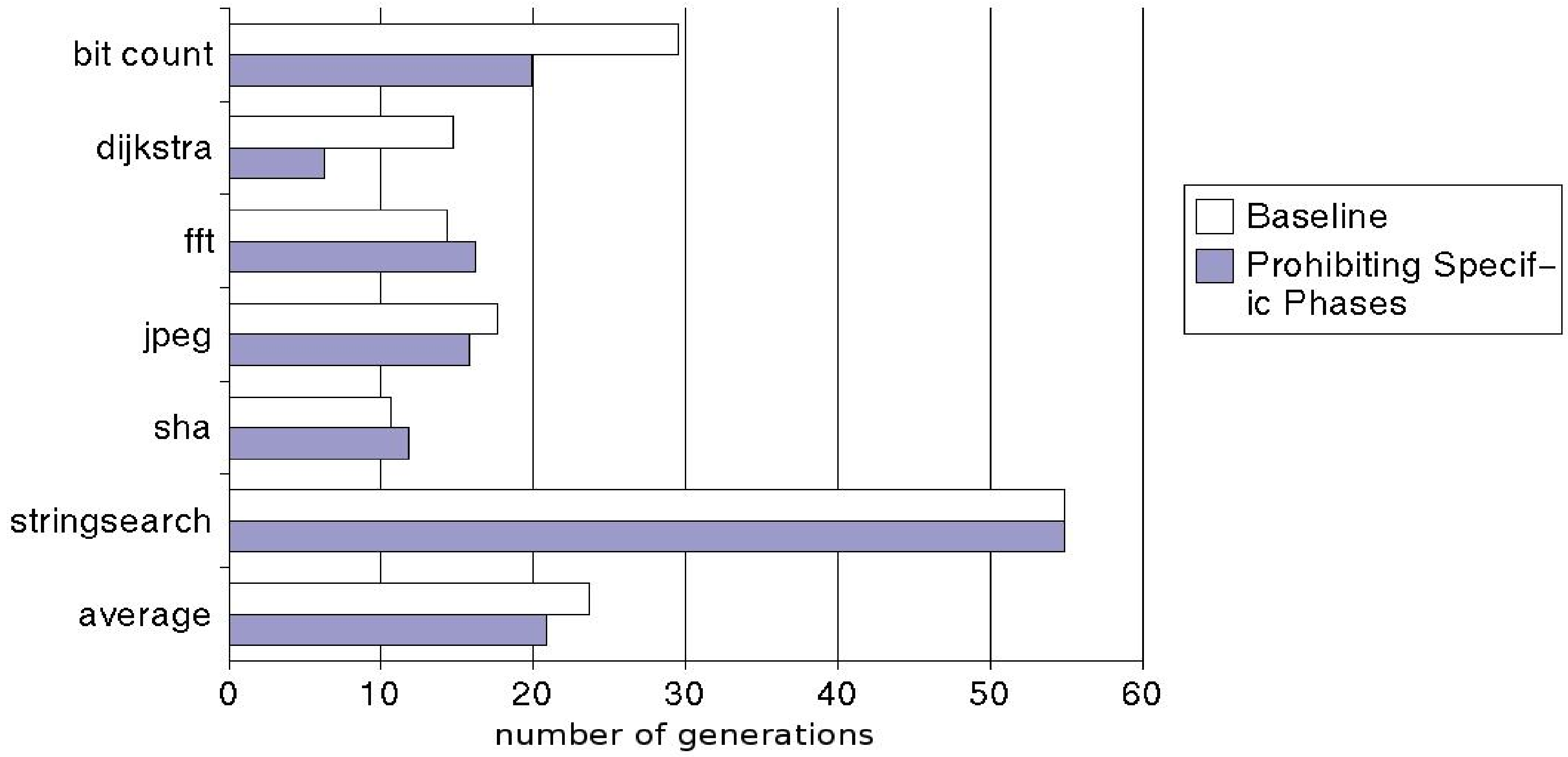
Number of Generations When Using the Batch Sequence



Prohibiting Specific Phases

- Perform static analysis on the function.
  - No loops, then no loop optimizations.
  - No scalar variables, then no register allocation.
  - Only one basic block, then no unreachable code elimination and no branch optimizations.
  - Etc.
- Such phases are prohibited from being attempted for the entire search for that function.

Number of Generations When Prohibiting Specific Phases

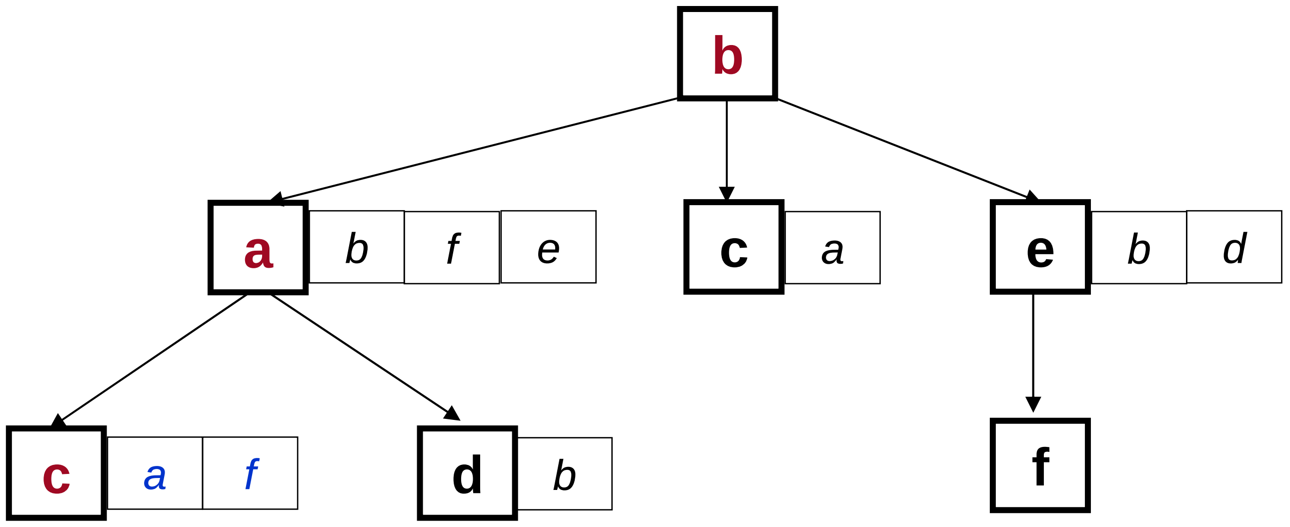


Prohibiting Prior Dormant Phases

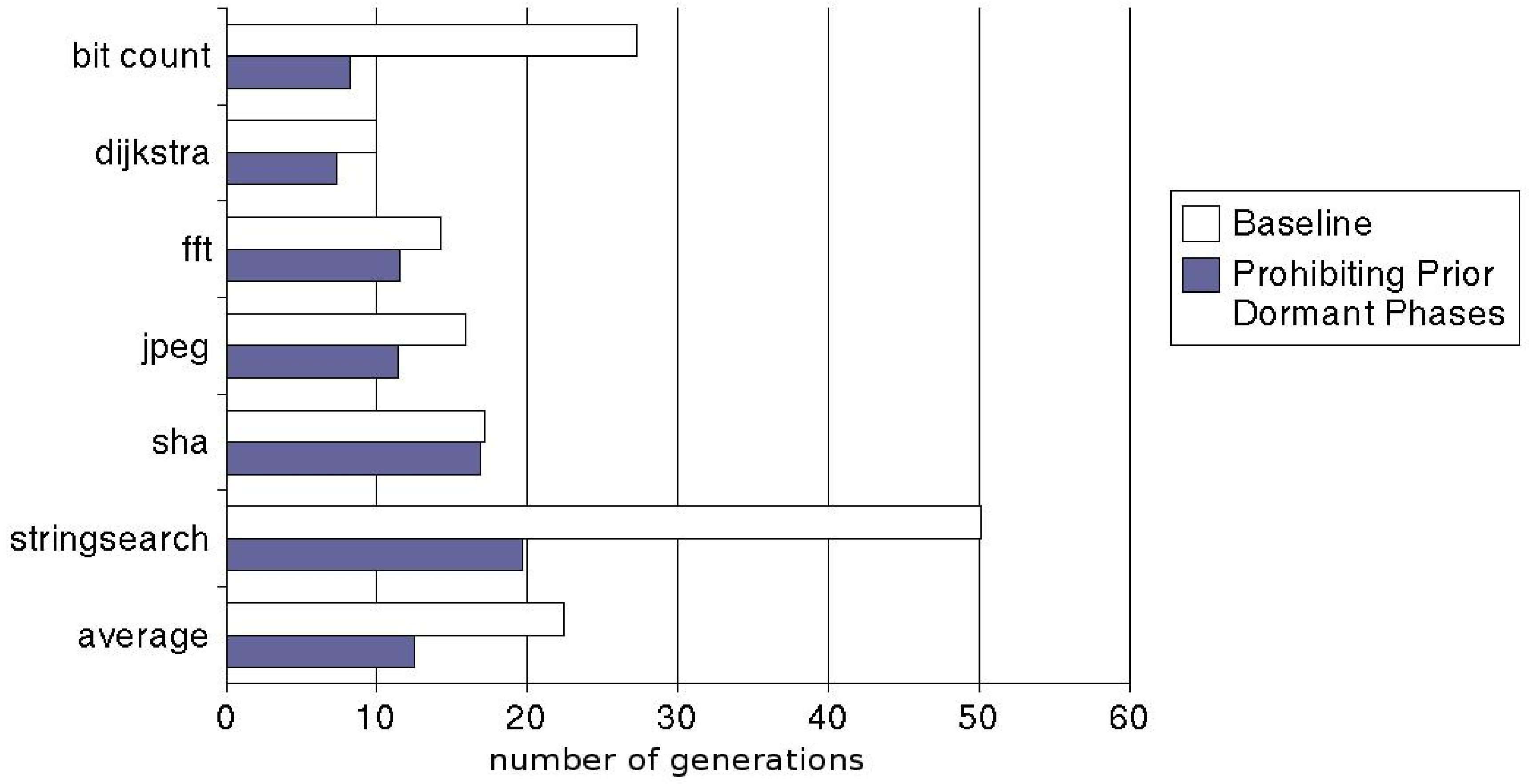
- Some phases will be found to be dormant given a specific prefix of active phases.
- If encounter the same prefix, then do not allow these prior dormant phases to be reattempted.
- Keep a tree of active prefixes and store the dormant phases with each node in the tree.
- Changed the genetic algorithm by forcing a prior dormant phase to mutate until finding a phase that has been active or not yet attempted with the prefix.

Prohibiting Prior Dormant Phases (cont…)

- ***a*** and ***f*** are dormant phases given the active prefix of ***bac*** in the tree.

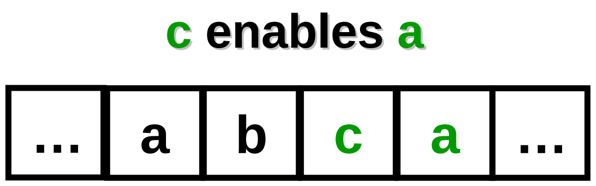


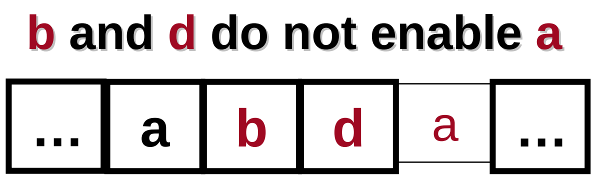
Number of Generations When Prohibiting Prior Dormant Phases



Prohibiting Un-enabled Phases

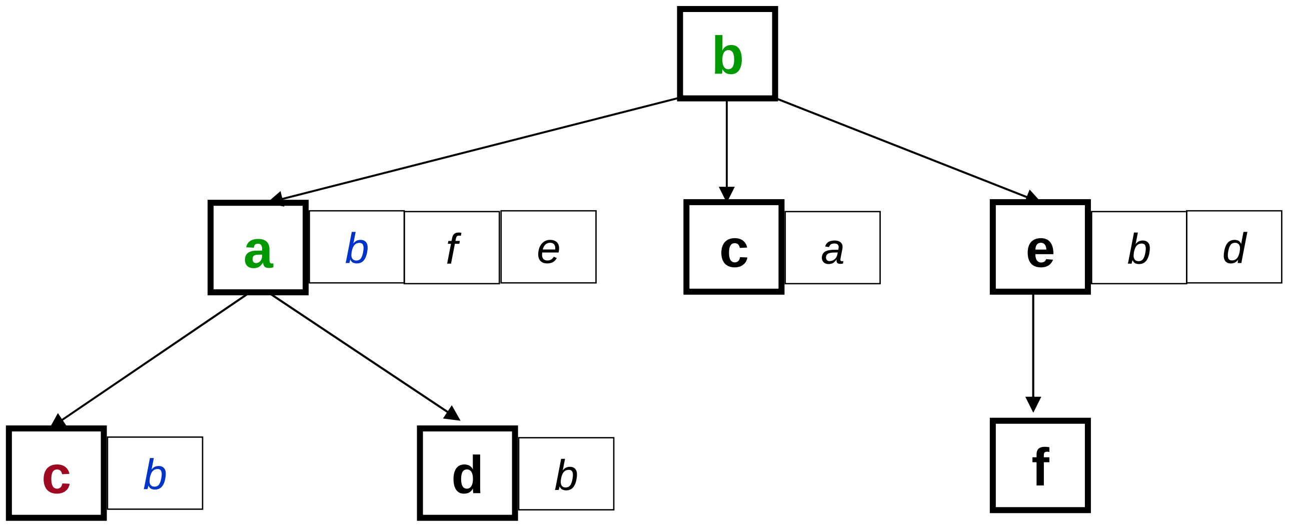
- Most optimization phases when performed cannot be applied again until enabled.
  - ex: Register allocation will not be enabled by most branch optimizations



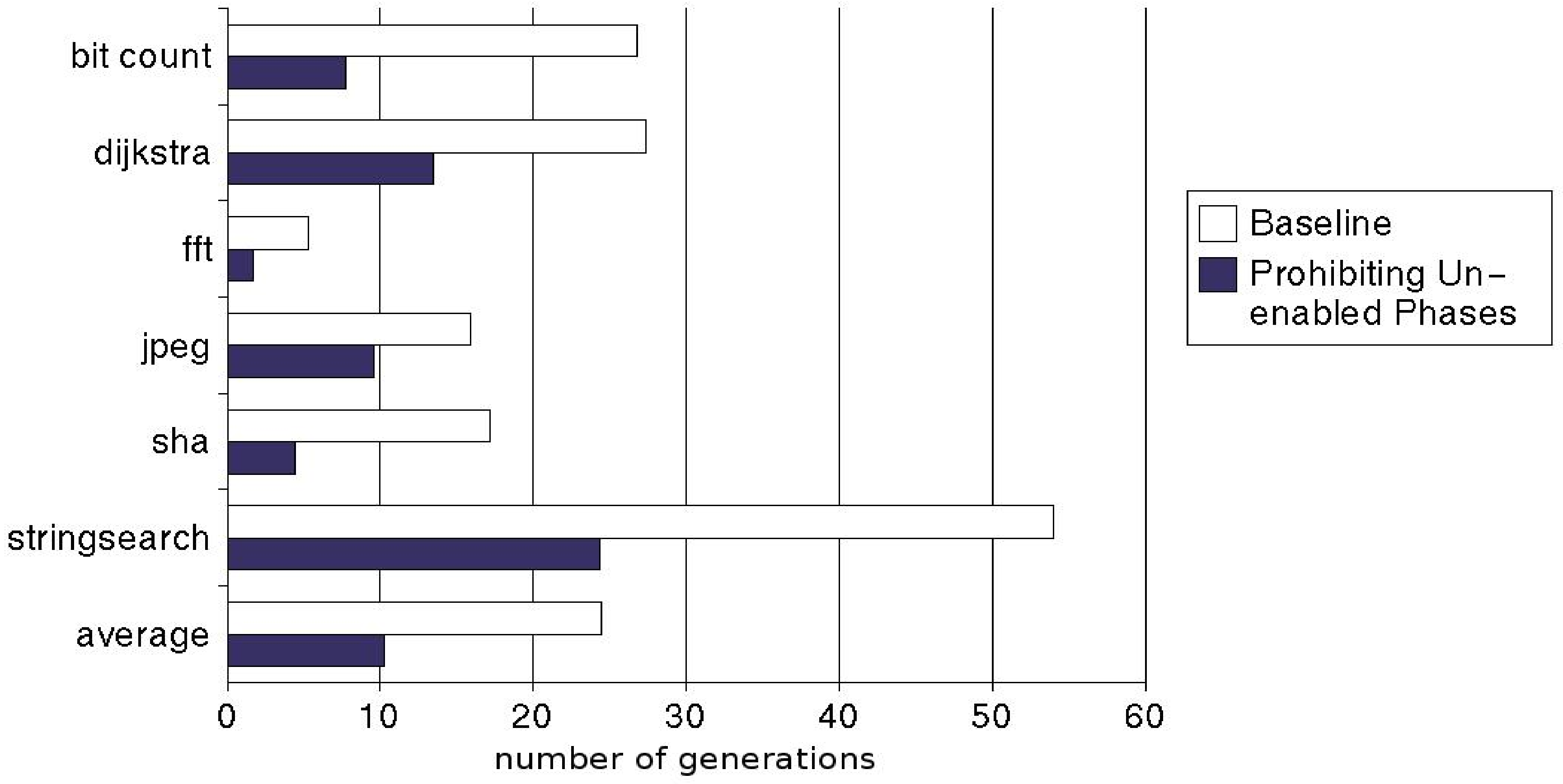


Prohibiting Unenabled Phases (cont.)

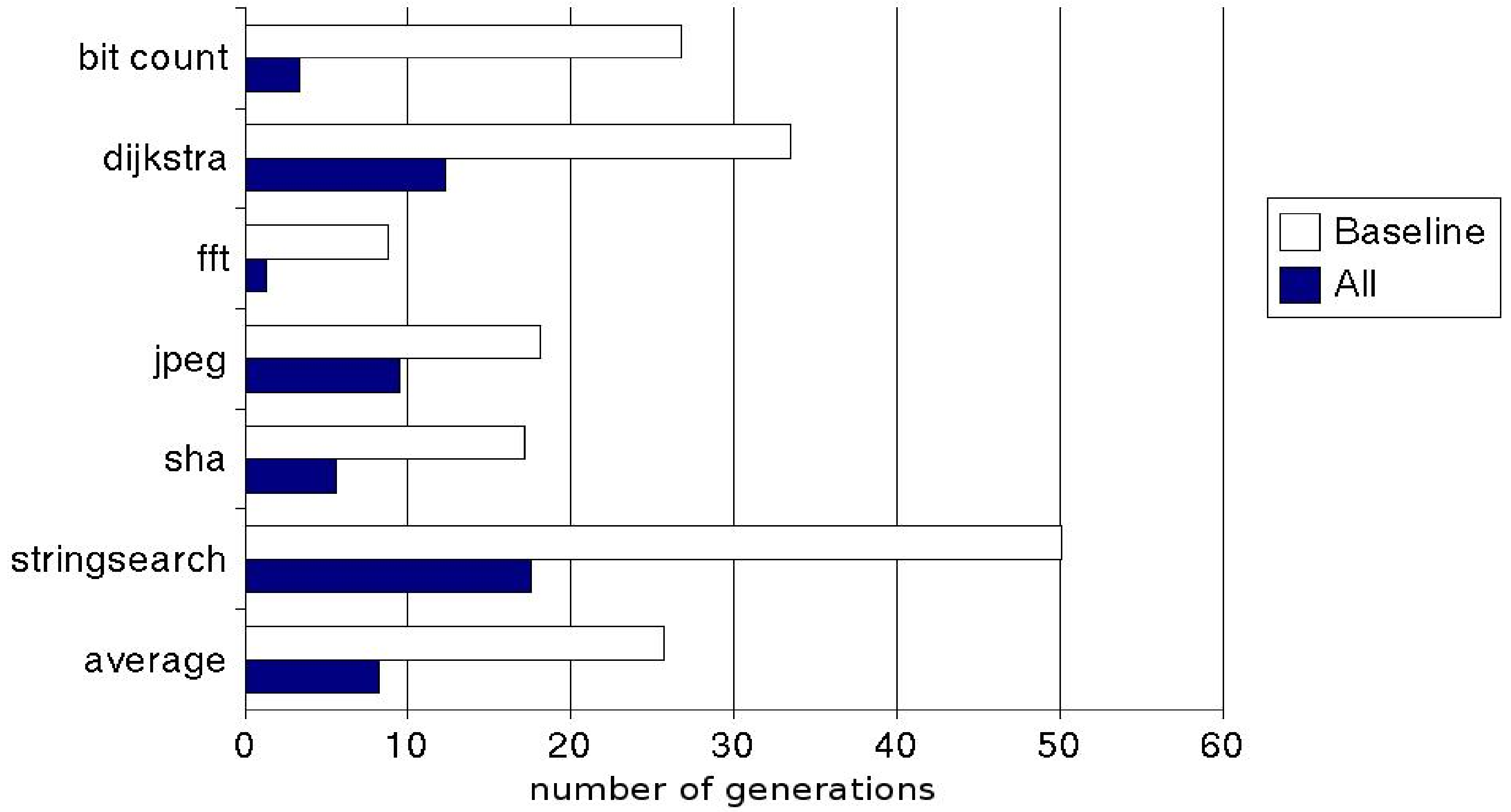
- *Assume **b** can be enabled by **a**, but cannot be enabled by **c**. Given the prefix **bac**, then **b** cannot be active at this point.*



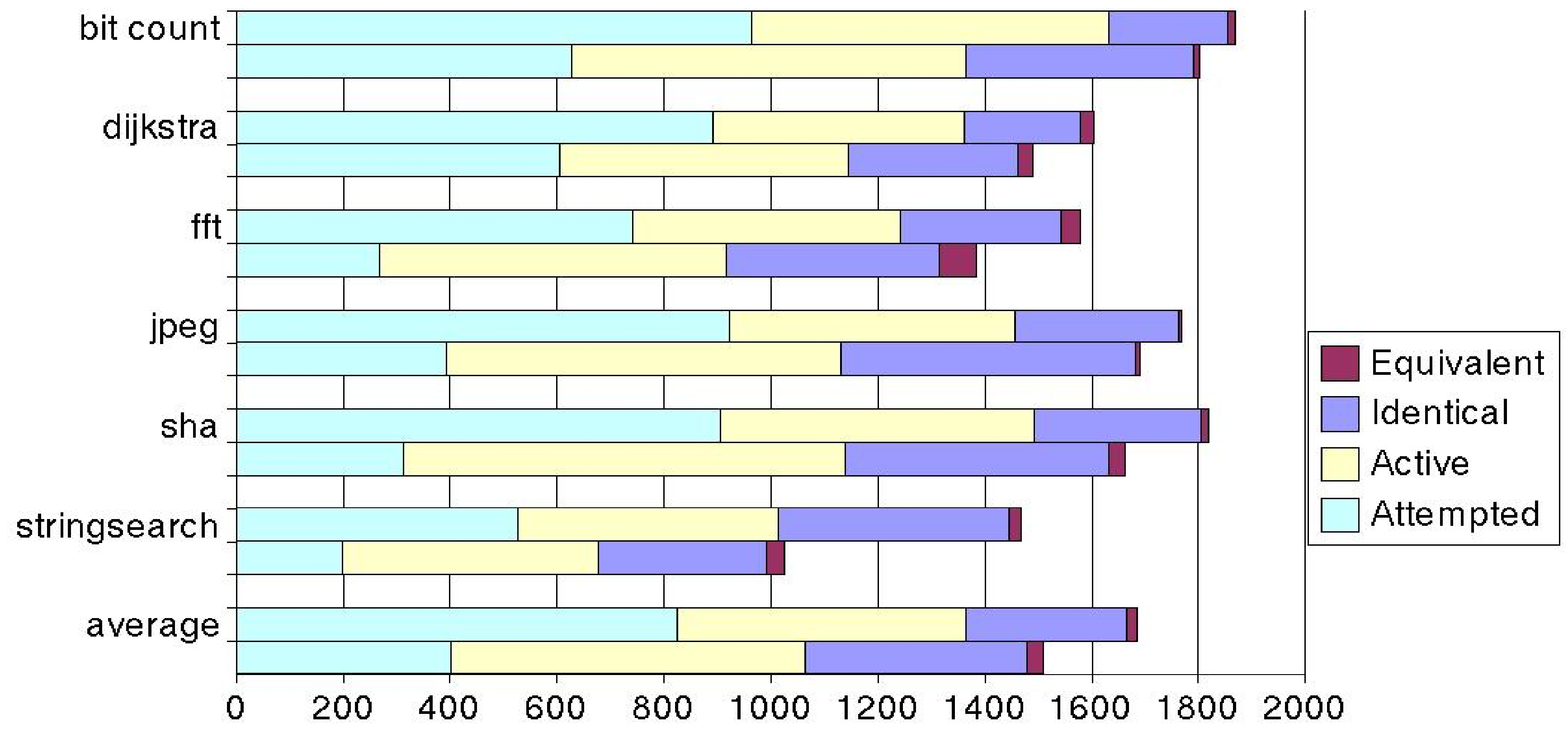
Number of Generations When Prohibiting Unenabled Phases



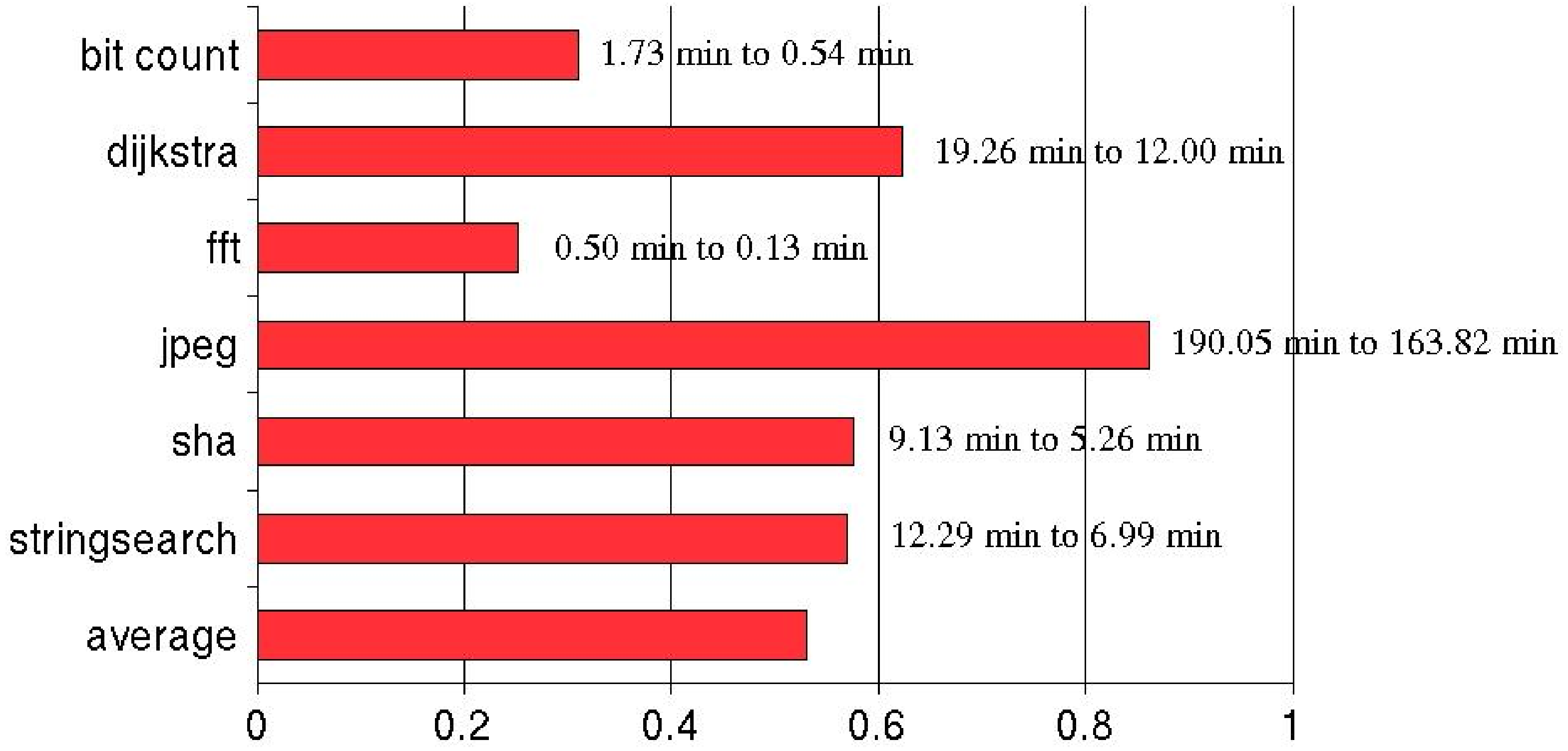
Number of Generations When Applying All Techniques



Number of Avoided Executions When Reducing the Number of Generations



Relative Search Time before Finding the Best Sequence



Related Work

- Superoptimizers
  - instruction selection: Massalin
  - branch elimination: Granlund, Kenner
- Iterative compilation techniques using performance feedback information.
  - loop unrolling, software pipelining, blocking
- Using genetic algorithms to improve compiler optimizations
  - Parallelizing loop nests: Nisbet
  - Improving compiler heuristics: Stephenson et al.
  - Optimization sequences: Cooper et al.

Future Work

- Detecting likely active phases given active phases that precede it.
- Varying the characteristics of the search.
- Parallelize the genetic algorithm.

Conclusions

- Avoiding executions:
  - Important for genetic algorithm to know if attempted phases were active or dormant to avoid redundant active sequences.
  - Same code is often generated by different active sequences.
- Reducing the number of generations required to find the best sequence in the search:
  - Inserting the batch compilation active sequence is simple and effective.
  - Can use static analysis and empirical data to often detect when phases cannot be active.