Improving Both the Performance Benefits and Speed of Optimization Phase Sequence Searches

Prasad Kulkarni and Michael Jantz

EECS, University of Kansas

David Whalley

CS, Florida State University

Optimization Phase Ordering Problem

- Compilers provide many optimization phases
  - to improve binary program speed, size, power
- Different orders or application of phases may produce distinct codes
  - phases enable, disable, interact with each other
- No single phase sequence is known to produce *optimal* code for all programs.

How to find the phase sequence for each function/program that generates the best code?

Iterative Compilation

- Evaluate the performance of several phase sequences to find the best one.
- How to generate different phase sequences?
  - exhaustive algorithms are often not feasible
  - *intelligent* machine learning algorithms
- Search time still substantial.
- Issues with iterative compilation
  - what is best granularity (function, file, program) to conduct phase sequence searches?
  - how to reduce iterative compilation search time?

Outline

- Introduction
- Experimental framework
- Tradeoffs with search granularities
- Hybrid iterative search algorithm
- Future work
- Conclusions

Experimental Framework

- We used the VPO compilation framework
  - established compiler backend, started development in 1988
  - comparable performance to gcc -O2
- All VPO phases operate on a single intermediate representation
  - iteratively applies phases until no more improvements
  - possible to arbitrarily reorder most phases
- Experiments use all 15 reorderable VPO phases.
- Target architecture is StrongARM SA-100
  - SimpleScalar simulator to evaluate performance

VPO Optimization Phases

Optimization Phases	
branch chaining	loop transformations
common subexpression elim.	code abstraction
remove unreachable code	eval. order determination
loop unrolling (2, 4, 8)	strength reduction
dead assignment elimination	reverse branches
block reordering	instruction selection
minimize loop jumps	remove useless jumps
register allocation	

Benchmarks

- Two programs each from six MiBench categories.
- 12 programs, 51 files, 251 functions, 90 executed.

Category	Program	File	Func	Description
auto	bitcount	10	18	test processor bit manipulation abilities
	qsort	1	2	sort strings using the quicksort algorithm
network	dijkstra	1	6	Dijkstra's shortest path algorithm
	patricia	2	9	construct patricia tree for IP traffic
telecomm	fft	3	7	fast fourier transform
	adpcm	2	3	compress 16-bit linear PCM samples
consumer	jpeg	7	62	image compression / decompression
	tiff2bw	1	9	convert color tiff image to b/w
security	sha	2	8	secure hash algorithm
	blowfish	6	7	symmetric block cipher
office	stringsearch	4	10	searches for given words in phrases
	ispell	12	110	fast spelling checker

Iterative Search Algorithm

- Genetic algorithm based search algorithm
  - 20 *chromosomes* per *generation,* 200 generations
  - sequence length twice *active* batch sequence length
- Algorithm details
  - first population of sequences randomly initialized
  - apply each sequence, and sort by performance
  - replace 4 sequences in low performing half using *crossover* (gene mixing)
  - replace each phase with another randomly selected phase with 5-10% probability during *mutation*
  - fitness criteria is 50% speed and 50% size, over whole program performance
- Searches were performed at the program, file, and function levels.

Program-Level Searches

```
DO
   determine next compilation settings;
   compile entire program with these settings;
   IF any function is not redundant THEN
      get entire program performance results
        by simulating the program;
UNTIL number of iterations completed;
```

- single phase sequence for all functions in program
- sequence length is twice maximum active batch length over all program functions
- each simulation evaluates fitness of one sequence for *all* program functions

File-Level Searches

```
FOR each file in program DO
   DO
      determine next compilation settings;
      compile all functions in file with
        these settings;
      IF any function is not redundant THEN
         get performance of functions in file
           by simulating the program;
   UNTIL number of iterations completed;
END FOR
```

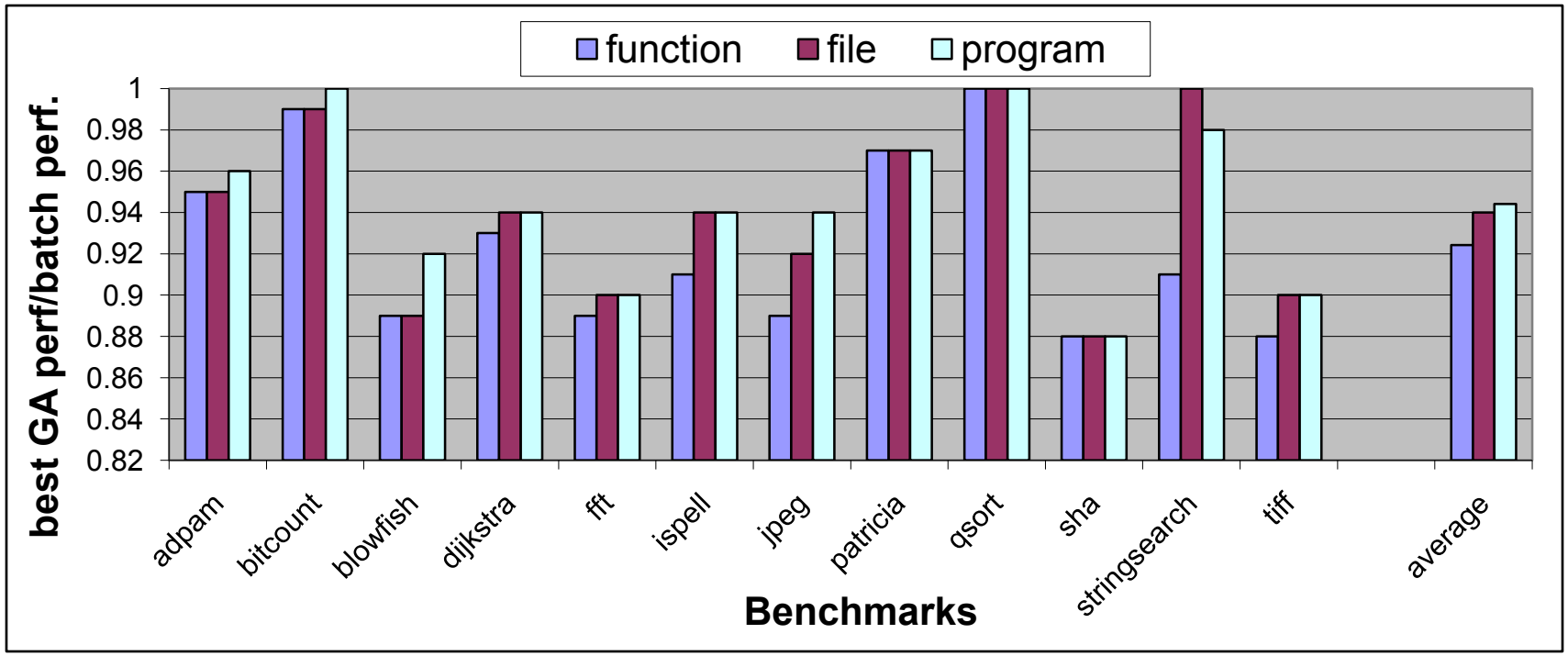
- single phase sequence for all functions in file
- finest granularity for compilers that do not allow different sequences for individual functions
- smaller average sequence length; more simulations

Function-Level Searches

```
FOR each function in program DO
   DO
      determine next compilation settings;
      compile function with these settings;
      IF function is not redundant THEN
         get function performance
           by simulating the program;
   UNTIL number of iterations completed;
END FOR
```

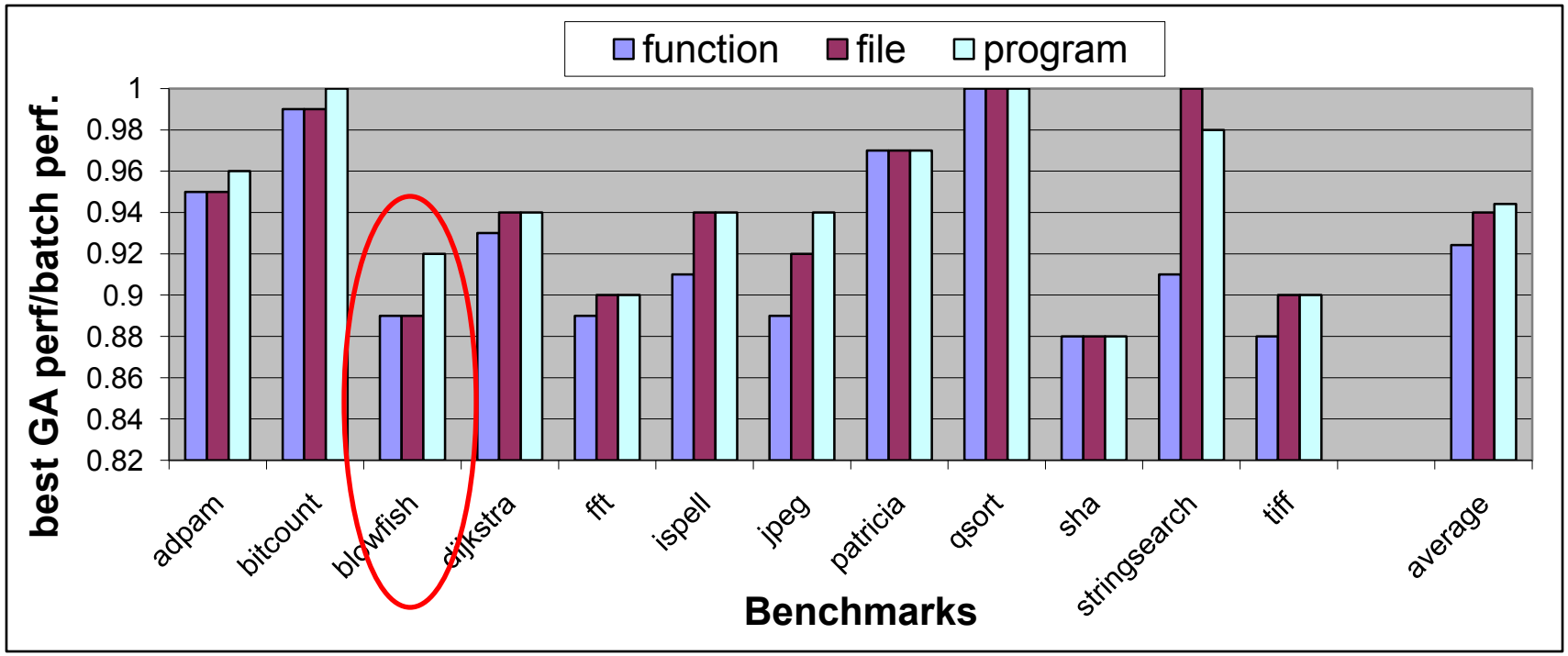
- different sequences and search for each function
- smallest average sequence length; most simulations
- greatest flexibility in customizing phase orderings over smaller code regions

Performance Tradeoffs



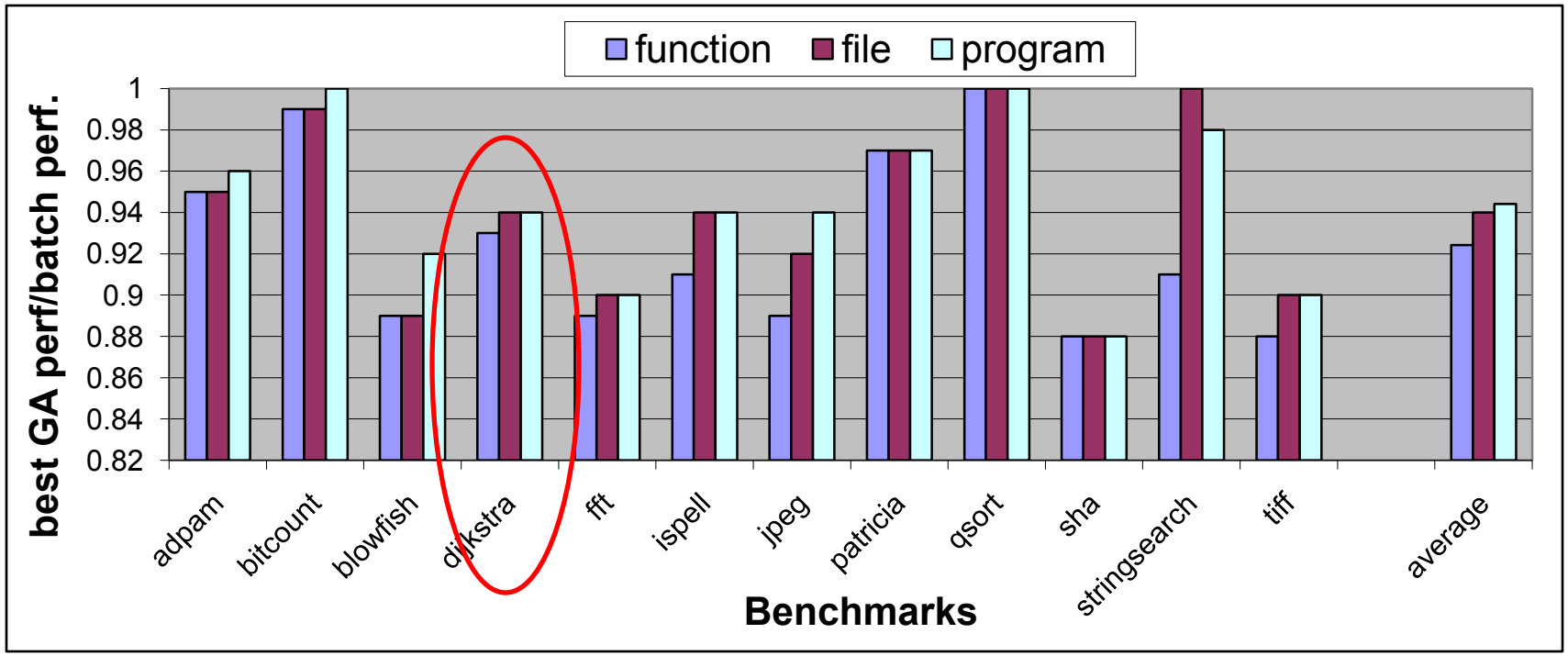
- Lower granularity function-based search achieves best performance.

Performance Tradeoffs



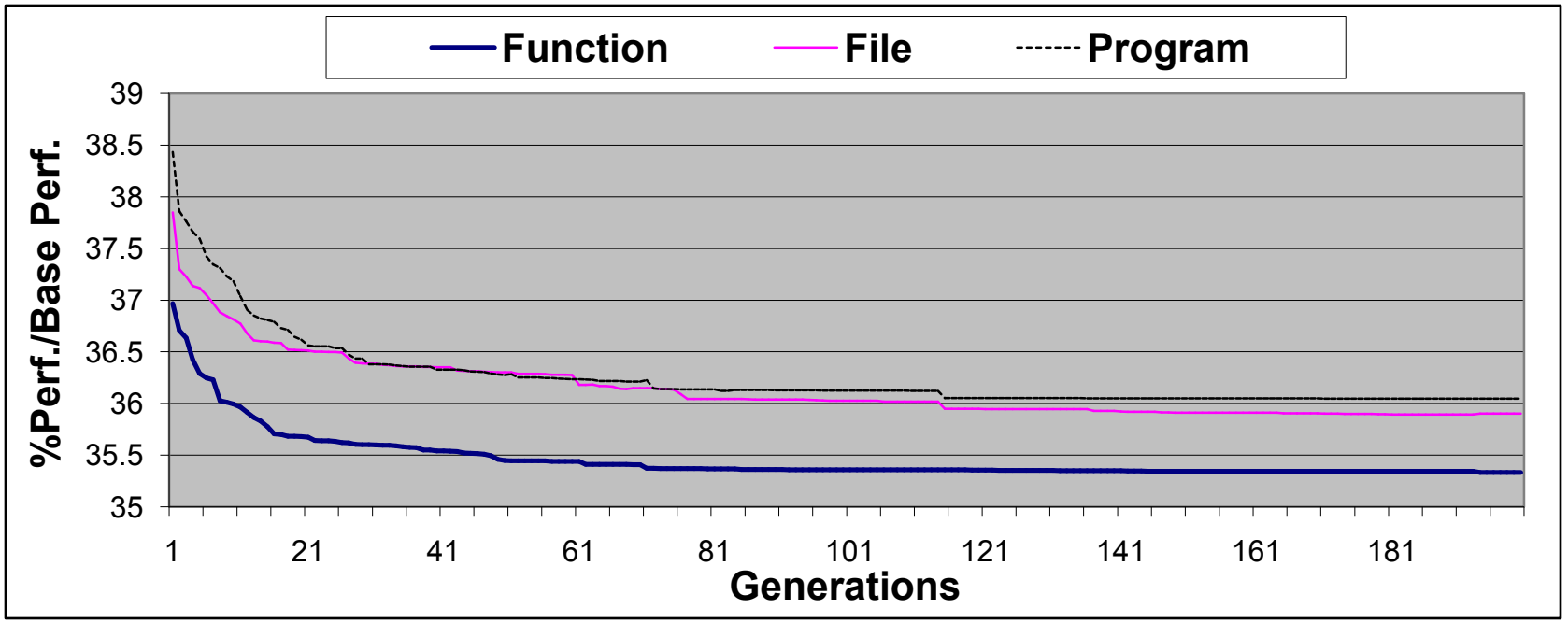
- Lower granularity function-based search achieves best performance.
- File-based = function-based, if each file has a single function

Performance Tradeoffs



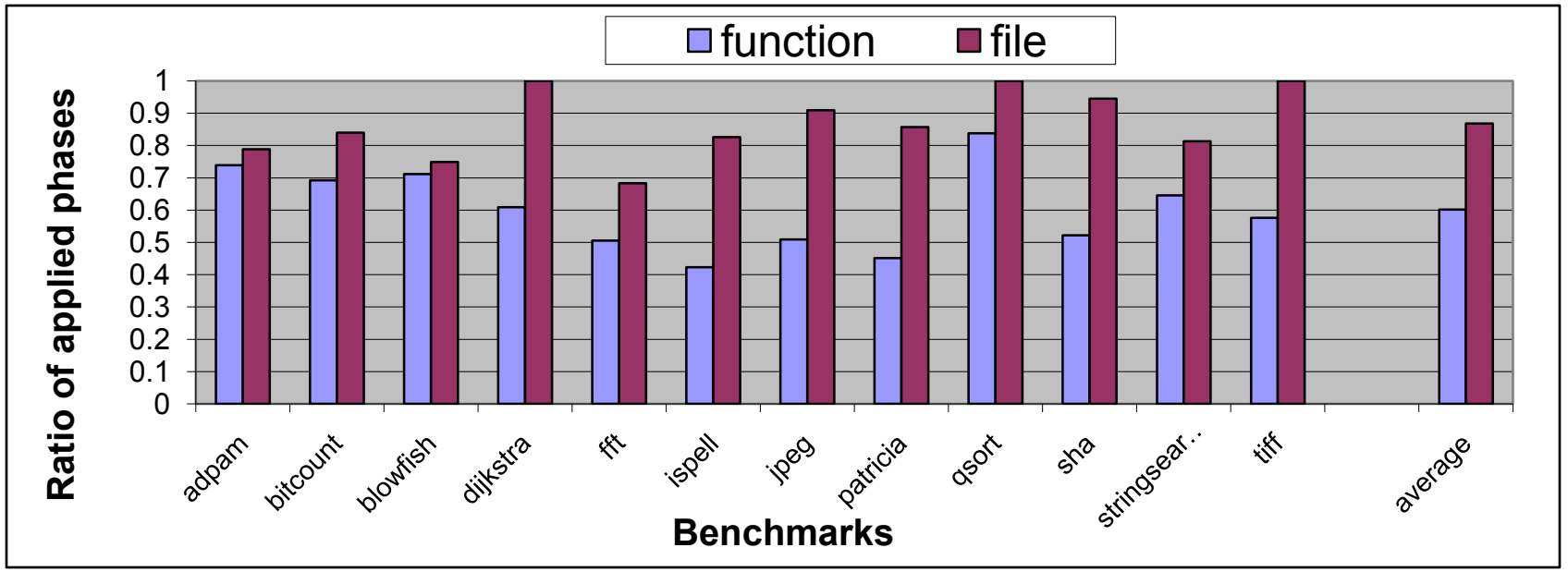
- Lower granularity function-based search achieves best performance.
- File-based = function-based, if each file has a single function.
- File-based = program based, for single file programs.

Search Progress



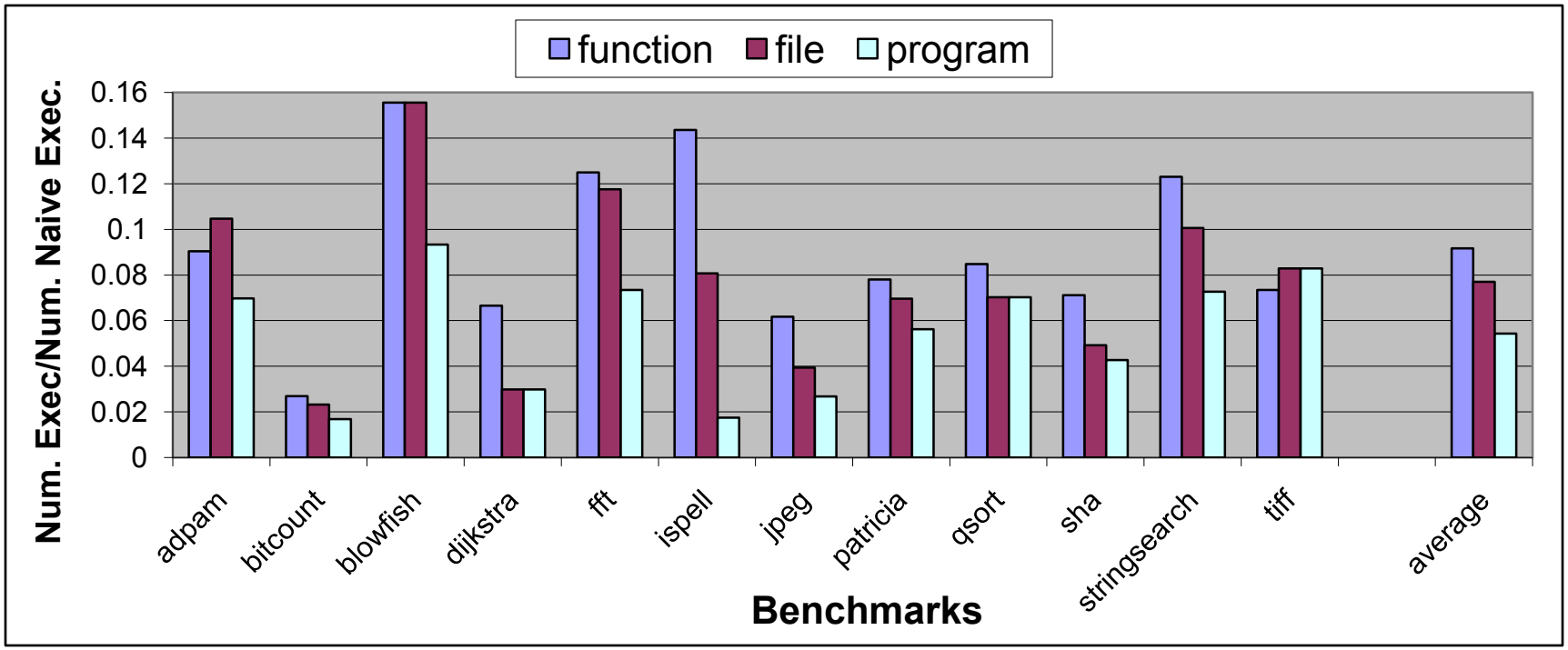
- All search types achieve respective best results at about the same time.
- Function-level searches perform well even with smaller number of generations.

Compilation Time



- Function-level search applies 60% of phases compared to program-level search.
- File-level search applies 87% of phases compared to program-level search.

Number of Simulations



- Program-level search requires 59% of the executions required for function-level search.
- File-level search requires 84% of the executions required for function-level search.

Need for Hybrid Search Strategy

- Function-level searches
  - + finer search granularity generates best code
  - + custom sequence lengths minimize compile time
  - – individual function evaluations require the greatest number of simulations
- Program-level searches
  - – generates less efficient code after search
  - – longer sequence lengths increase compile time
  - + smallest number of program simulations

Can we achieve the best of both worlds ?

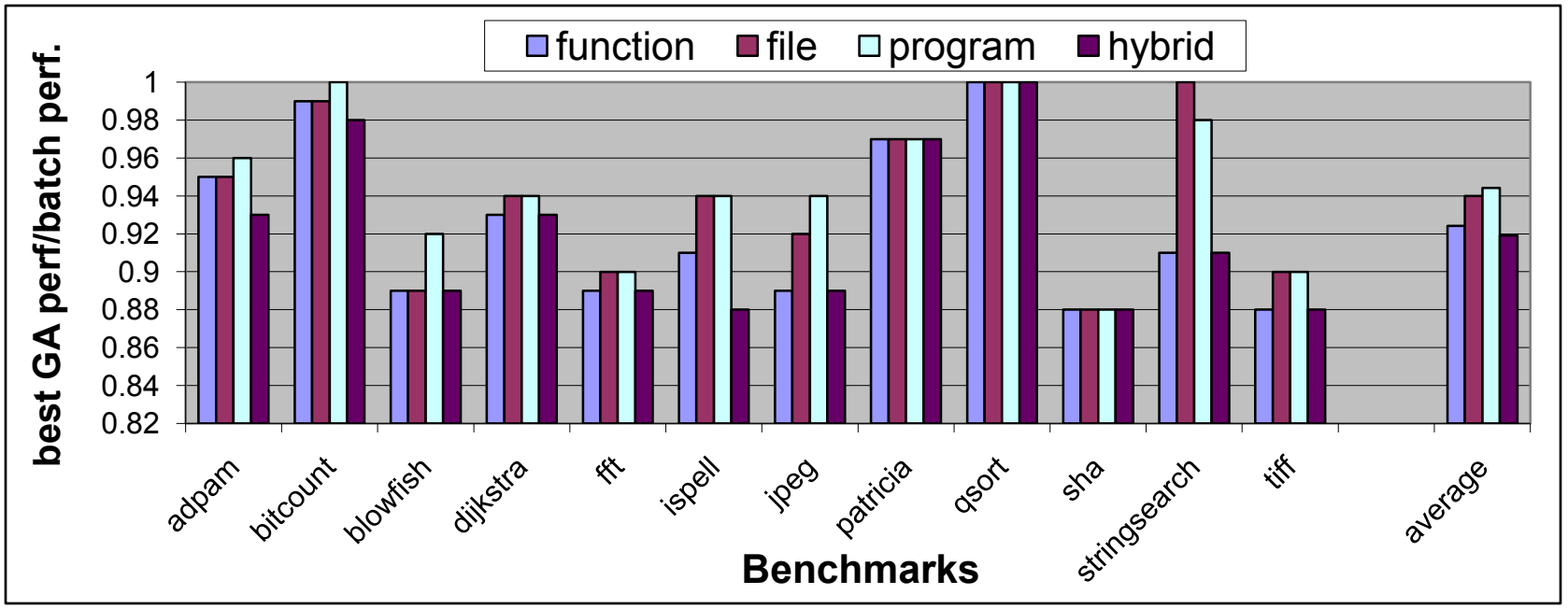
Hybrid Search Strategy

- Perform individual function searches in *parallel*.
- Delay program simulation until each function has an instance to evaluate.
  - each simulation evaluates one (*different*) sequence for each function in the program
- Each function search may be at different stages of completion.
- Typically requires fewer simulations than even a program-based approach

Hybrid Search

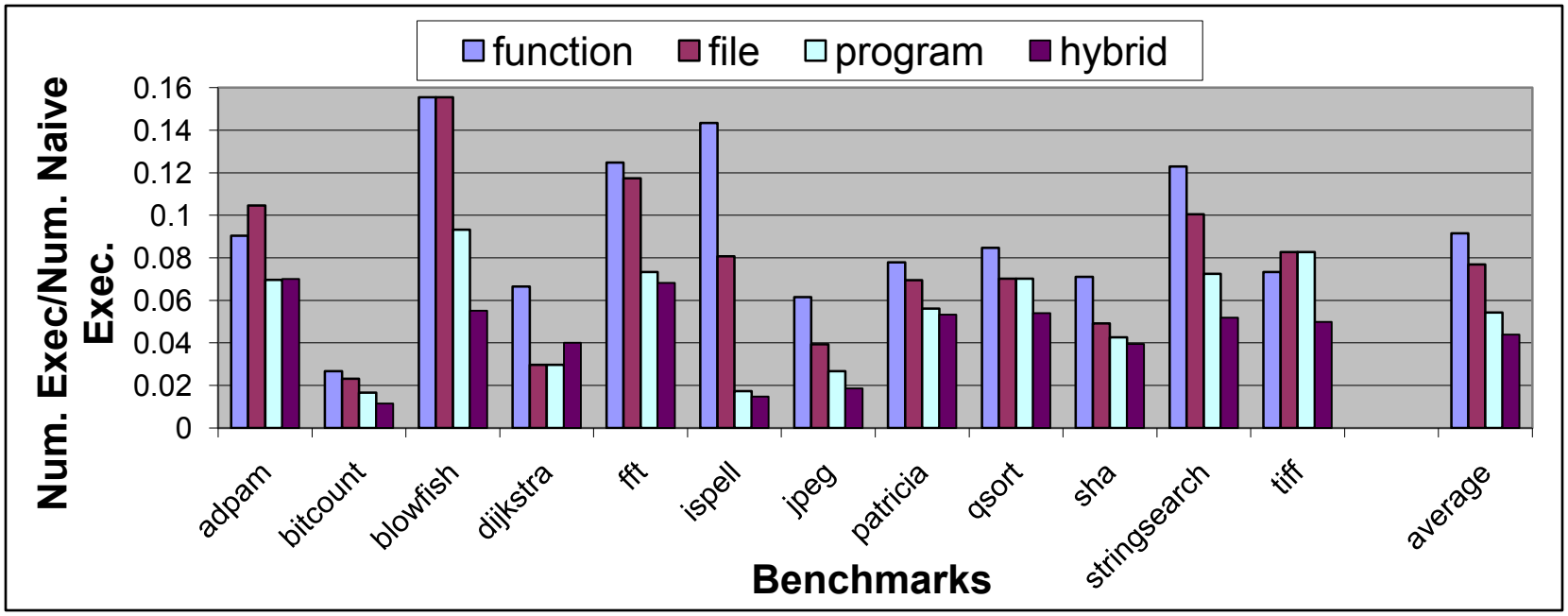
```
DO
   FOR each function in program DO
      IF function search still incomplete THEN
         DO
            determine next compilation settings
              for this function;
            compile function with these settings;
         UNTIL function is not redundant OR
           function search is complete
      ENDIF
   END FOR
   get results of each function by
     simulating program once;
UNTIL number of search generations completed
  for all functions in program;
```

Hybrid Search – Code Performance



- Achieves performance comparable to function-level search in most cases.
- 8% performance improvement over aggressive *batch* compilation.

Hybrid Search – Number of Simulations



- Fewest number of program simulations.
- Requires 48% of the simulations required for function-level search.

Future Work

- Finer granularity searches achieve better quality code
  - find effective phase sequences over individual loops
- Determine how well our hybrid approach works with other search algorithms.
- Searches on multi-processor machines
  - explore parallelism in various search algorithms
  - study benefit of hybrid strategy on multi-processor machines

Related Work

- Triantafyllis et al. [CGO 2003] and Cooper et al. [LCTES 2005] used static performance estimators to reduce search time.
- Agakov et al. [CGO 2006] used static features to focus their iterative search.
- Kulkarni et al. [PLDI 2004, LCTES 2006] used pruning techniques to avoid redundant executions.
- Fursin et al. [HiPEAC 2005] evaluated versions of the same function in a single execution.

Conclusions

- We discovered that current iterative search algorithms for effective optimization phase sequences operating at *function*, *file* and *program* levels have different tradeoffs
  - program-level searches reduce search times due to smallest number of program simulations
  - function-level searches achieve better code performance and reduce compilation time
- We introduced a *hybrid* search strategy
  - that can achieve code performance comparable to function-level search with faster search times than program-level searches