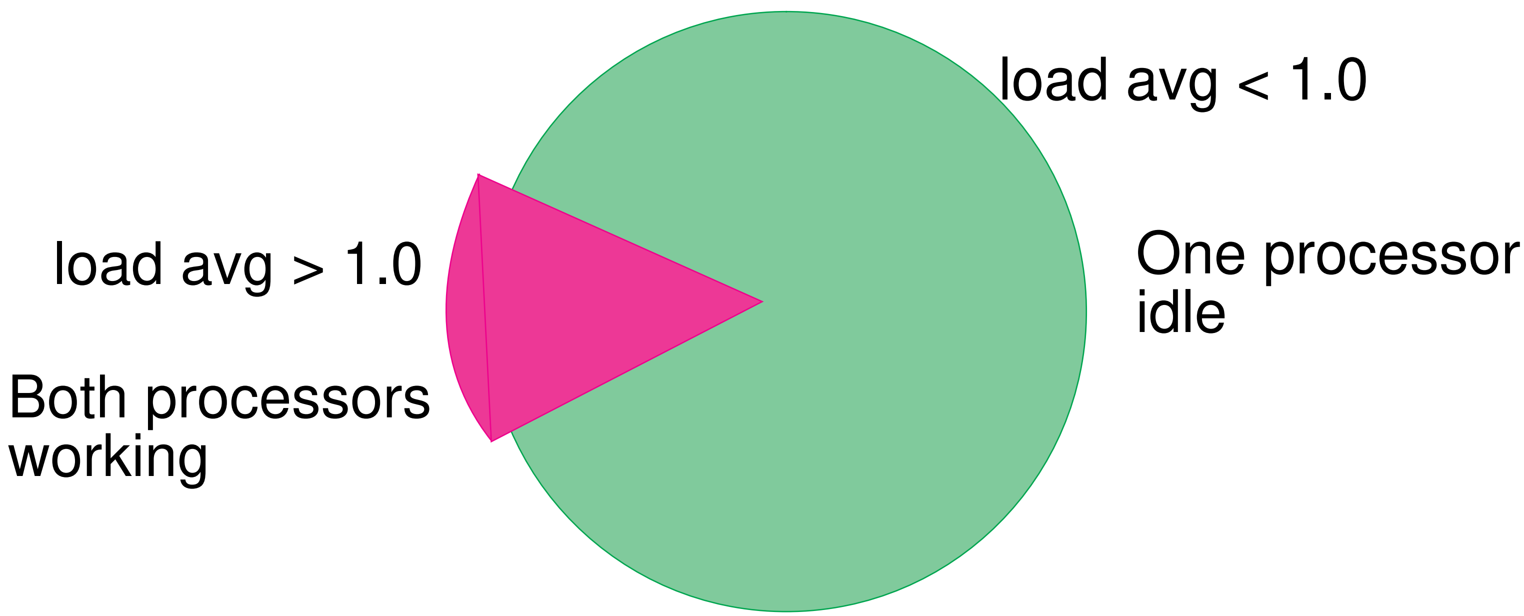
Efficient Run-time Monitoring Using Shadow Processing

Harish Patil and Charles Fischer

Computer Sciences Department
University of Wisconsin — Madison

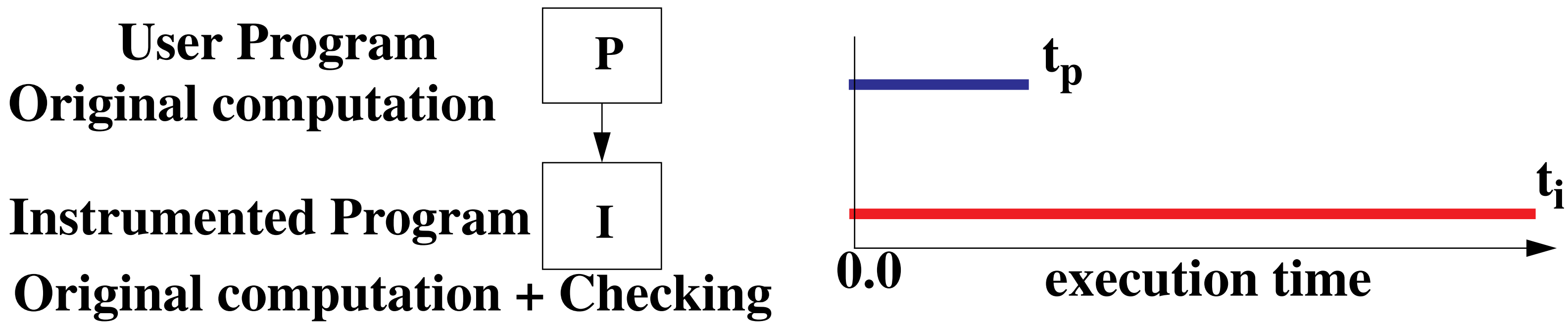
Efficient run-time monitoring using general purpose multiprocessors

1. Target architecture:
   Multiprocessor workstation
2. Load average on busy dual processor workstations



3. Objective: Use idle processors for run-time analyses — memory access checking, profiling, ...

Current memory access checking: sequential

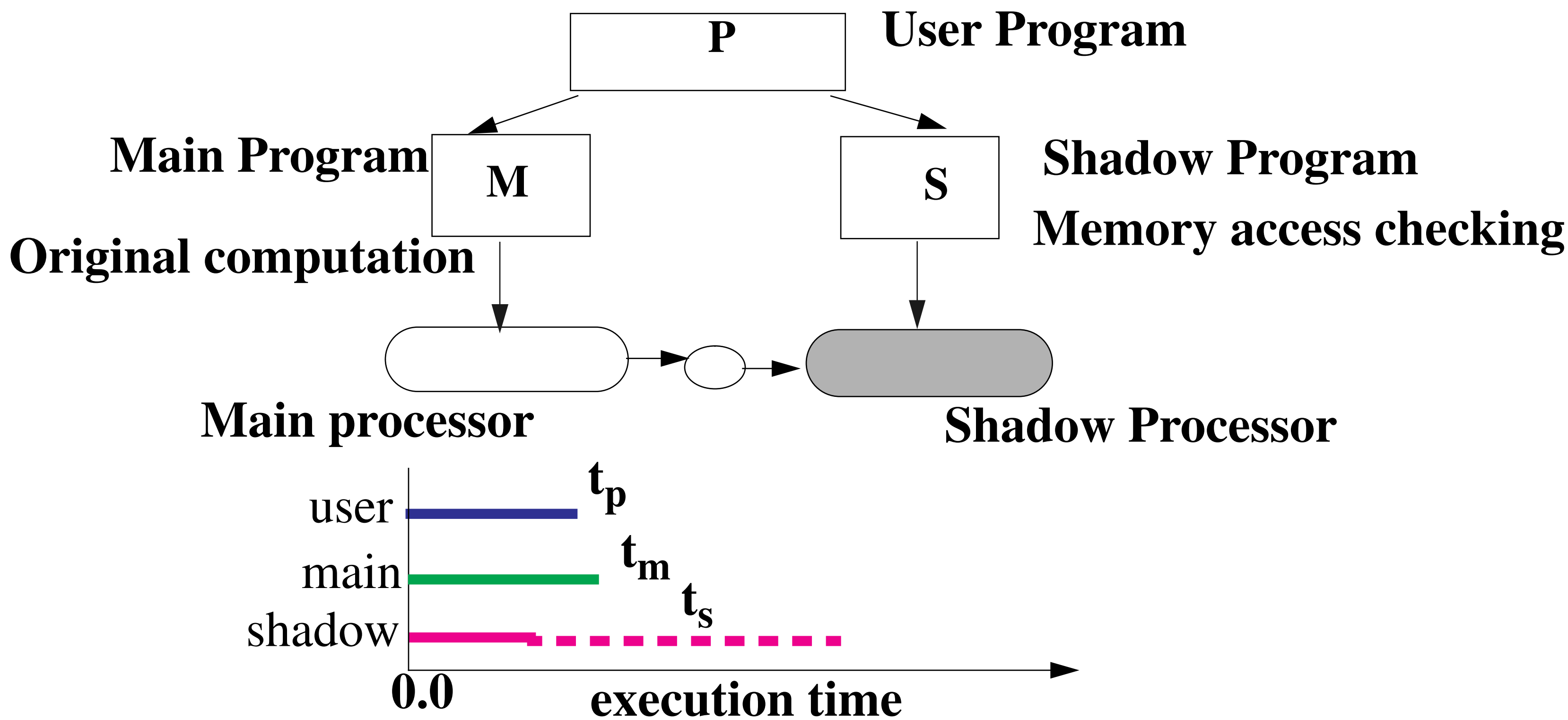


1. Large slowdown: delayed results
   5-6 times in Purify; 200 times in Codecenter

2. Not suitable for heavily used programs
   Most of the runs are error free: checking seems a waste

3. Turned off after program development and testing
   a. Errors may crash programs
      Empirical study: 26-33% of utilities crashed on random input
   b. Errors may produce subtly or intermittently wrong answers

Shadow Guarding
Memory access checking that is always enabled



1. Perform memory access checking on a second processor

2. Recompute control flow: Communicate values *only if* needed
   interactive input, return values of system calls

Outline

1. Introduction
2. Shadow guarding: Overview
3. Examples
4. Key results
5. Generating shadow programs: techniques and performance
6. Conclusions

Shadow Guarding: Overview

1. Check validity of pointer and array accesses

- Spatial access error:
  Out of bounds array access
- Temporal access error:
  Access of a deallocated object

2. Maintain pointer attributes in shadow using GUARDS

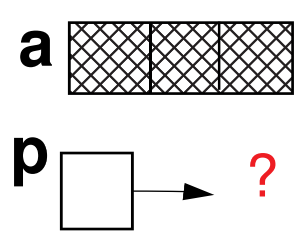
Main	Shadow
`char *p;`	`char_guard G_p;`
`..p = ..`	`..Modify G_p..`
`..*p..`	`..CHECK(G_p)..`

Guards: Attributes for spatial checking

Main

```
1.  char a[3];
2.  char *p;
3.  ..a[i]..

4.  ..*p..
```



Shadow

```
1.  char_guard G_a={3,0};
2.  char_guard G_p={0,0};
3.CHECK(G_a);
(0<= i+G_a.index<G_a.count)
4.CHECK(G_p);
(0<= G_p.index <G_p.count)
```

	count	index
G_a	3	0
G_p	0	0

Fields of a **`char_guard`** for catching out of bounds accesses

1. count: Number of elements
2. index: Current element being pointed to

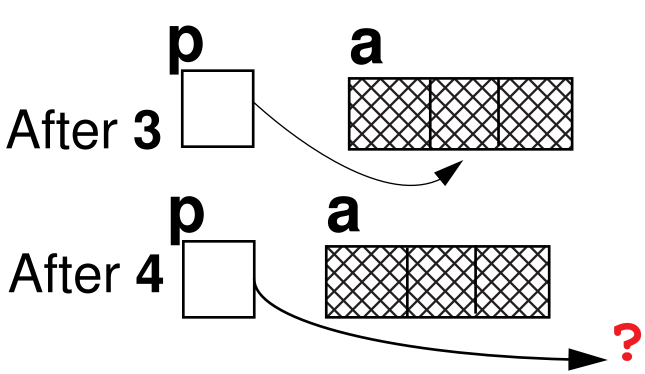
Example: Out of bounds access

Main

```
1 char *p;
2 char a[3];

3 p = &a[1];

4 p = p + 4;
5..*p..
```



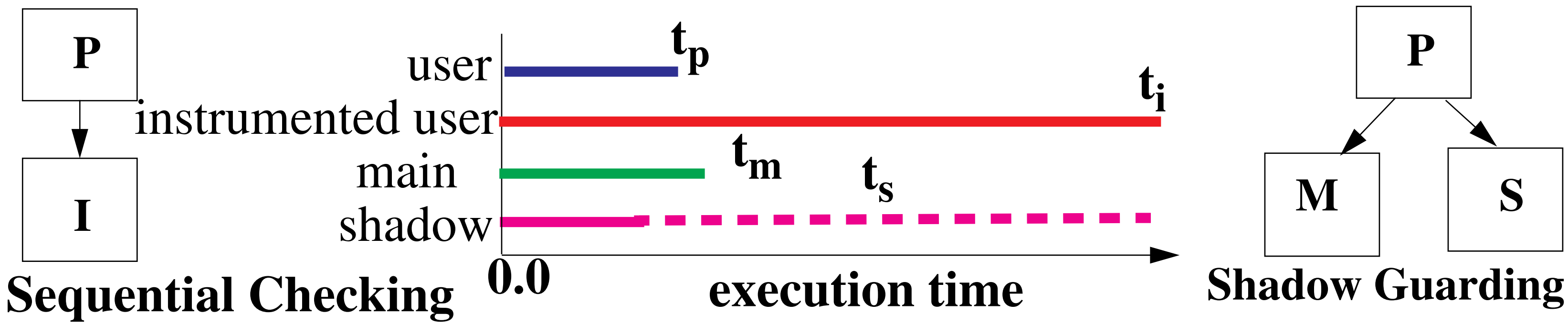
Shadow

```
1 char_guard G_p={0,0};
2 char_guard G_a={3,0};

3 G_p.count=G_a.count;
   G_p.index=G_a.index+1;
4 G_p.index += 4;
5 CHECK(G_p);-- Fails
```

		count	index
After 3	G_p	3	1
After 4	G_p	3	5

Shadow Guarding: Key results



1. Execution time: main program = user program
   + Overhead < 10% in most test cases
   + No need to turn off checking

2. Shadow guarding process: faster than sequential checking
   + Up to 2 times faster

3. Found errors in heavily used C programs
   + UNIX (SunOs) utilities: 6 earlier reported, 6 new
   + SPEC92: decompress(1), sc(2): No program crash

Slowdown of the user program

Increase in real execution time

alvinn	eqntott	xlisp	sc	compress
1.13%	2.98%	8.3%	0%	5.88%

Costs:

1. Communication
   interactive input, return values of system calls

2. Memory contention
   Shadow process needs resources

Outline

1. Introduction
2. Shadow guarding: Overview
3. Examples
4. Key results
5. Generating shadow programs: techniques and performance
6. Conclusions

Techniques for Generating Shadow Programs

1. Delete output calls

   Calls affecting external environment

   fprintf(), **rmdir()**

   \+ Shadow never affects outcome of the user program

2. Use slicing technology

   Retain only statements affecting pointer values

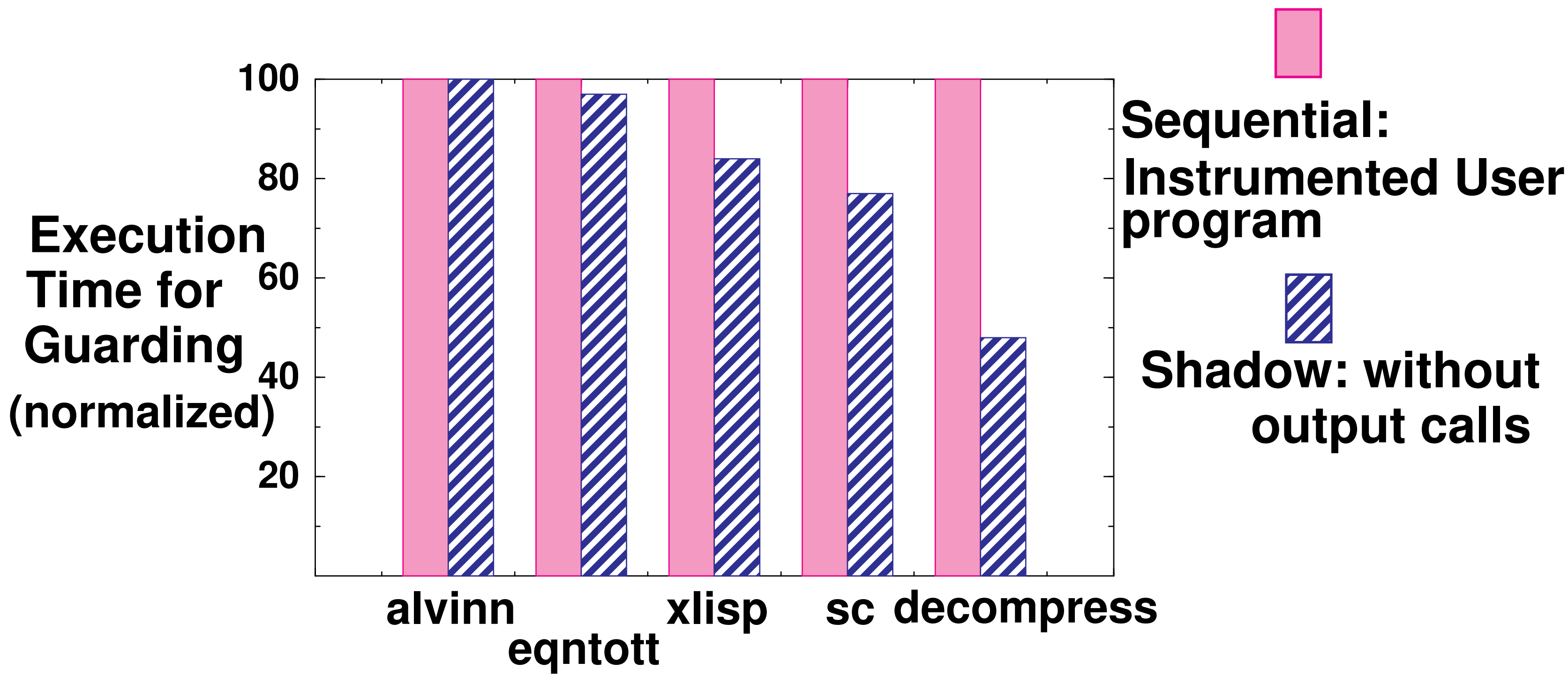
   \+ Shadow performs minimal computation

Performance Evaluation:

Comparison with sequential checking

(—Instrumented user program)

Deleting output calls



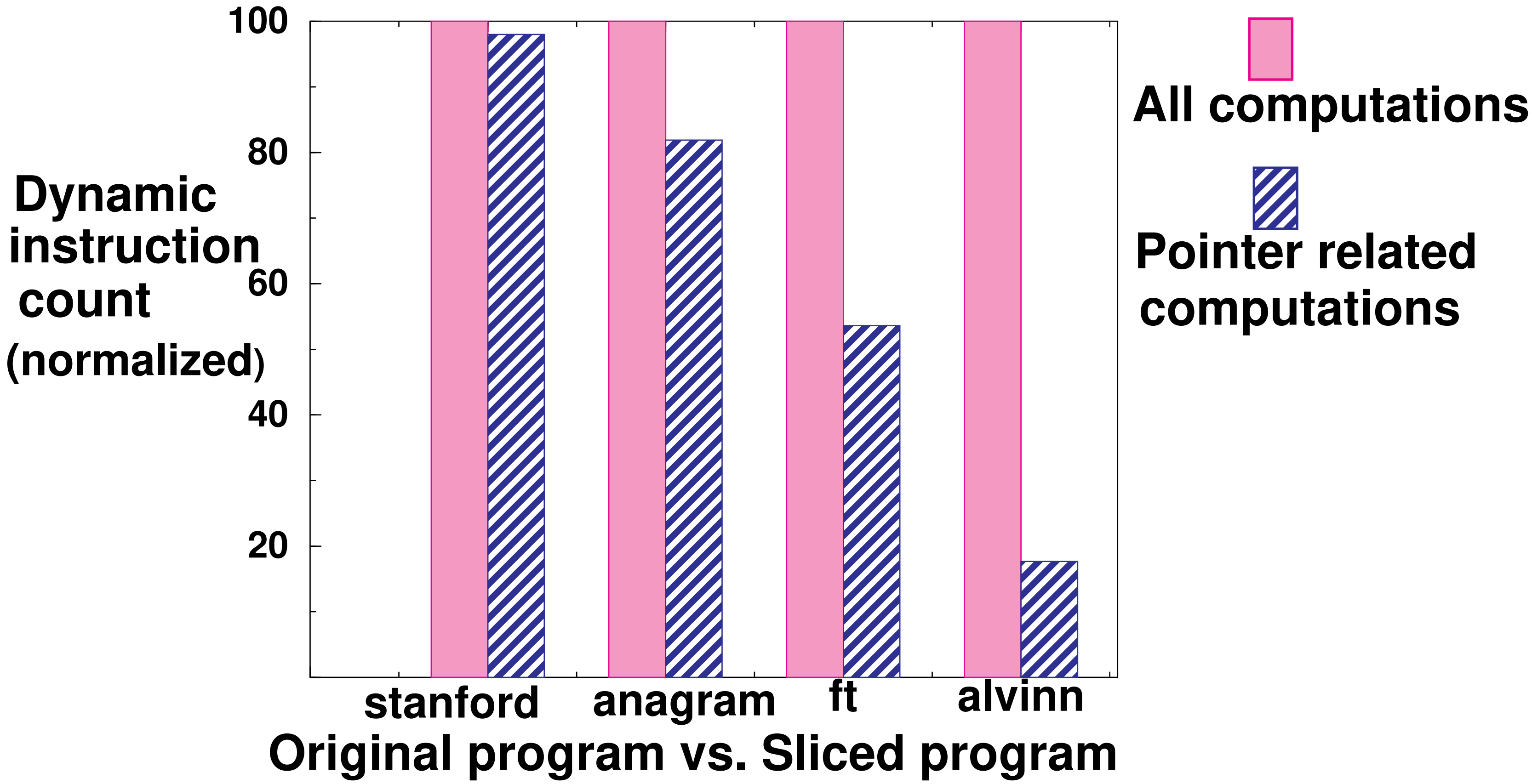
Sequential checking vs. Shadow guarding

Output calls: Calls that affect the external environment

Shadow program = (input program — output call) + checking

Instrumented user program = input program + checking

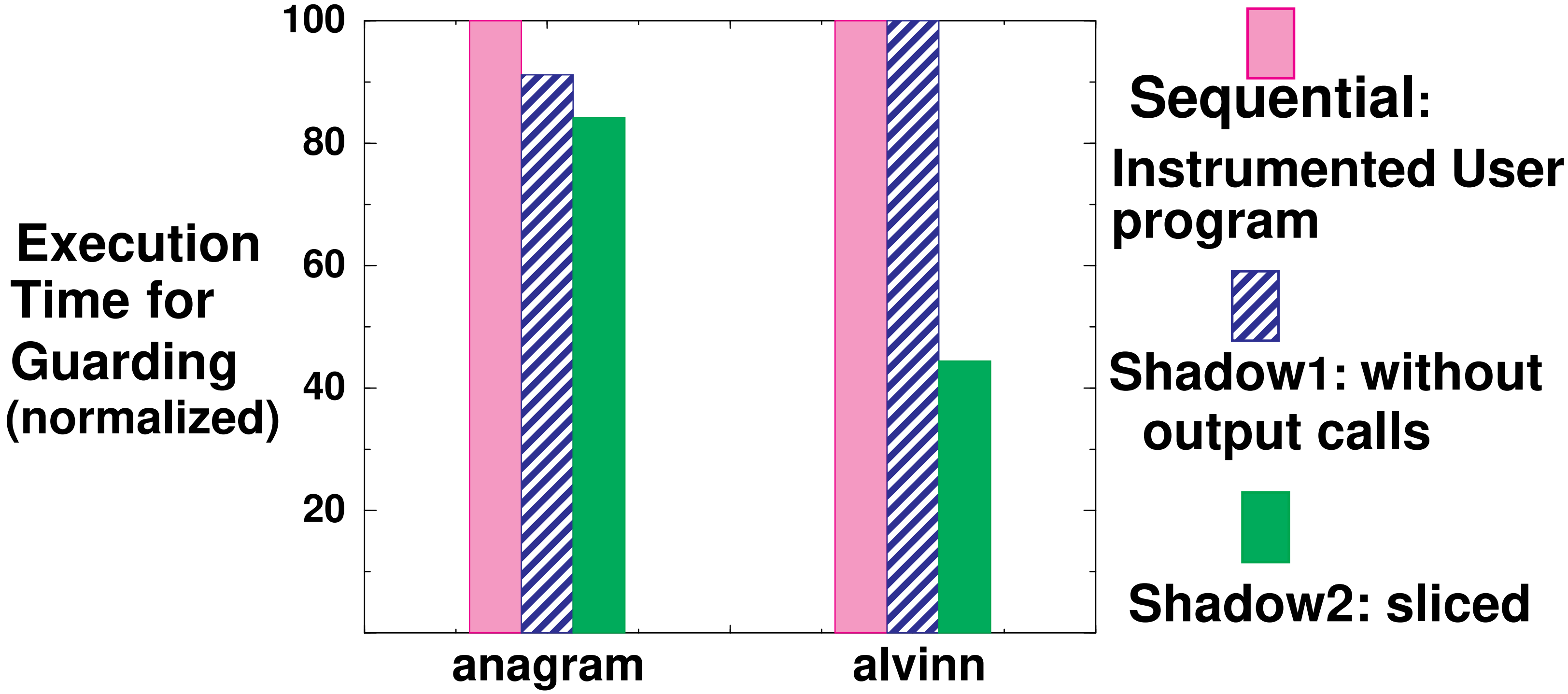
Potential benefit of using slicing



1. Slicing tool from Wisconsin Program Integration System
2. Pointer operations + related control flow in C programs

Executing program slices for Shadow Guarding

Slicing criteria: Dereferenced pointers, array indices



Sequential checking vs. Shadow guarding

Shadow1 = (input program — output call) + checking

Shadow2 = *slice*(input program) + checking

Conclusions

1. Shadow Guarding: Use idle processors for memory access checking

   Extended for memory leak detection

2. Advantages:

   + Low perceived overhead
   + Fast error reporting
   + Effective in catching errors in heavily used C programs

3. Future work

   Handle parallel programs

   Spot checking of programs as processors become available

   Novel analyses now considered too expensive