Hierarchical Pointer Analysis for Distributed Programs

Amir Kamil and Katherine Yelick

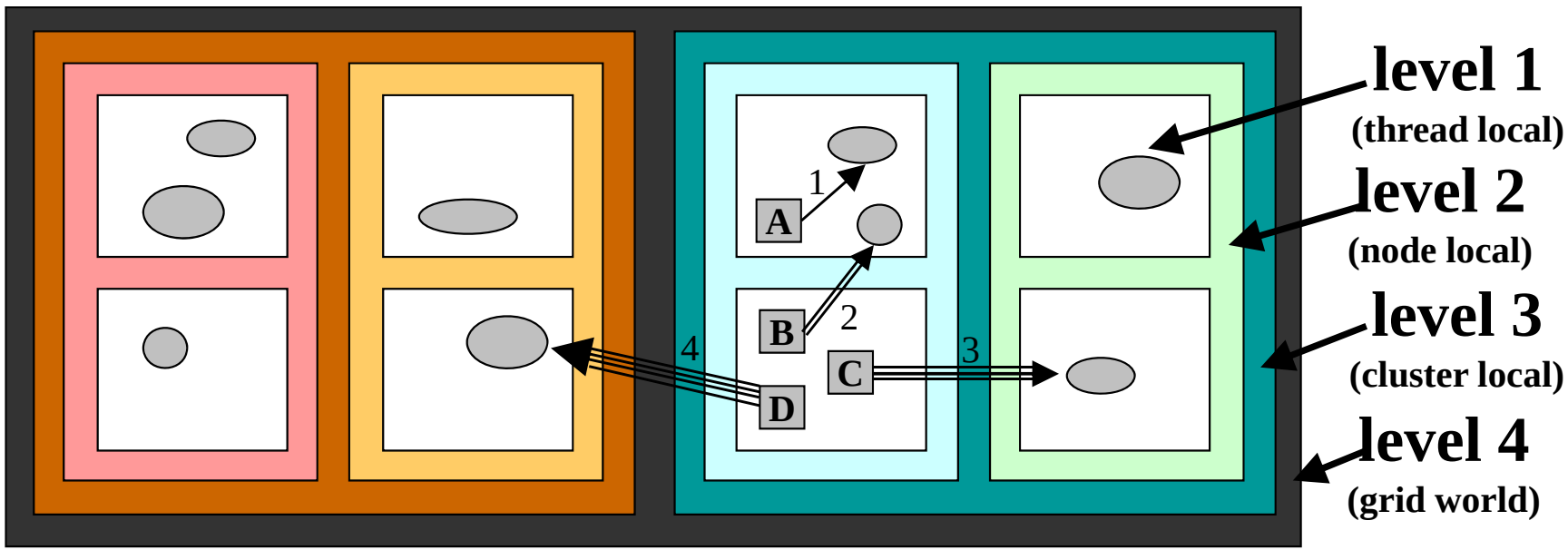
U.C. Berkeley

August 23, 2007

Background

Hierarchical Machines

- **Parallel machines often have hierarchical structure**



Partitioned Global Address Space

- **Partitioned global address space (PGAS) languages provide the illusion of shared memory across the machine**
- ***Wide* pointers used to represent global addresses**
  - Contain identifying information plus the physical address

- ***Narrow* pointers can still be used for addresses in the local physical address space**

The Problems

Three Problems

- **What data is private to a thread?**
- **What data is local to the physical address space?**
- **What possible race conditions can occur?**

Data Privacy

- **Data is *private* if it cannot leak beyond its source thread**
- **Useful to know which data is private for global garbage collection, monitor optimization, and other applications**

Data Locality

- **Recall: global pointers composed identifying information and an address**

Process ID: 1	Address: 0xf9a0cb48

- **When dereferenced, runtime system must perform a check to determine if the data is actually in the local physical address space**
  - If local, then access directly
  - If not local, then perform communication
- **Thus, global pointers are more costly in both space and time, even if the actual data is local**

Race Detection

- **Shared memory introduces the possibility of race conditions**
  - Two threads access the same memory location
  - The accesses can be simultaneous (no intermediate synchronization)
  - At least one access is a write

The Solution

Hierarchical Pointer Analysis

- **A pointer analysis that takes into account the machine hierarchy can answer the preceding questions**
- **For each variable, we want to know not only from which allocation sites the data could have originated, but also from which threads**

Related Work

- **Thread-aware pointer analysis has been done by others**
  - Rugina and Rinard , Zhu and Hendren, Hicks, and others
  - None of them did it for hierarchical, distributed machines
- **Data privacy and locality detection previously done by Liblit, Aiken, and Yelick**
  - Uses constraint propagation
  - Does not distinguish allocation sites

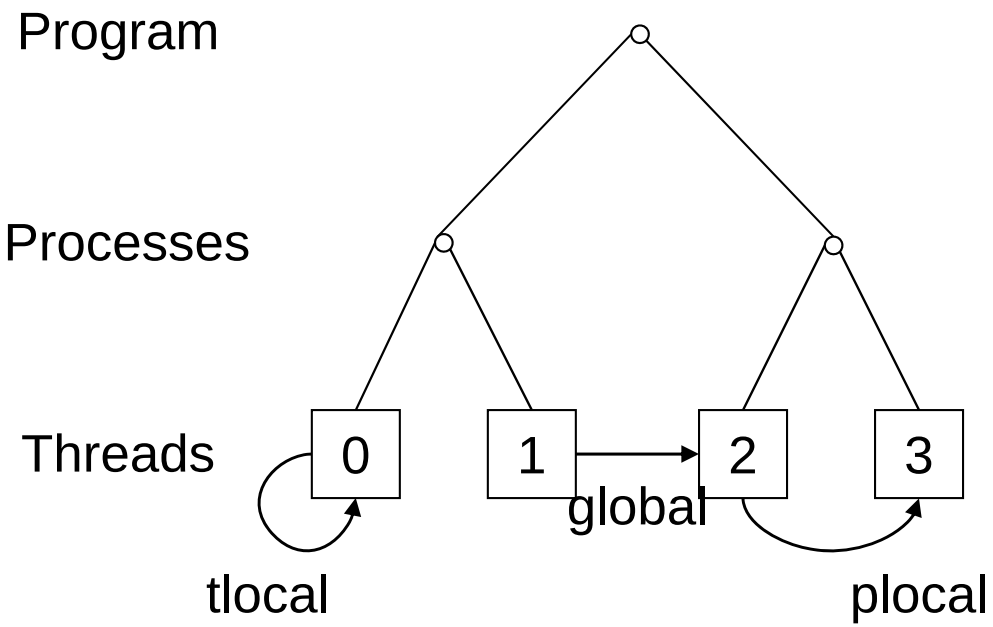
The Implementation

Titanium

- **Titanium is a single program, multiple data (SPMD) dialect of Java**
  - All threads execute the same program text
- **Designed for distributed machines**
- **Global address space – all threads can access all memory**
- **At runtime, threads are grouped into processes**
  - A thread shares a physical address space with some other, but not all threads

Titanium Memory Hierarchy

- **Global memory is composed of a hierarchy**



- **Locations can be thread-local (tlocal), process-local (plocal), or potentially in another process (global)**

The Analysis

Approach

- **We define a small SPMD language based on Titanium**
- **We produce a type system that accounts for the memory hierarchy**
  - The analysis can handle an arbitrary number of levels, but we use three levels in this talk
- **We give an overview of the pointer analysis inference rules**

Language Syntax

- **Types**

  τ **::= int | ref$_q$** τ

- **Qualifiers**

  q ::= tlocal | plocal | global

  (tlocal ⊂ plocal ⊂ global)

- **Expressions**

  e ::= new$_l$ τ **(allocation)**

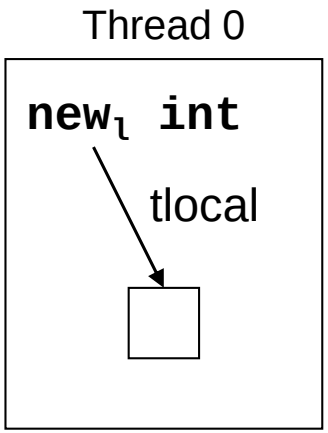
  | transmit e_1 from e_2 (communication)

  | e_1 ″ e_2 (dereferencing assignment)

  | convert(e, n) (type conversion)

Type Rules – Allocation

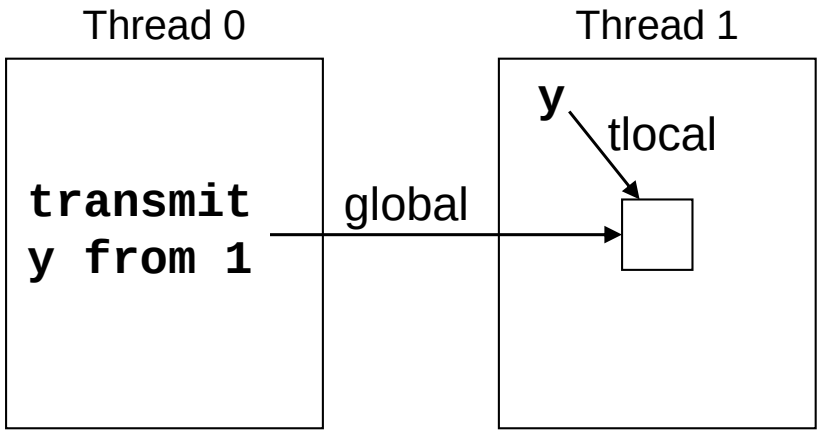
- **The expression $\text{new}_l\ \tau$ allocates space of type τ in local memory and returns a reference to the location**
  - The label l is unique for each allocation site and will be used by the pointer analysis
  - The resulting reference is qualified with `tlocal`, since it references thread-local memory



$$\frac{}{\Gamma \subset \textbf{new}_l\ \tau : \textbf{ref}_{\textbf{tlocal}}\ \tau}$$

Type Rules – Communication

- **The expression `transmit` e_1 `from` e_2 evaluates e_1 on the thread given by e_2 and retrieves the result**
- **If e_1 has reference type, the result type must be widened to global**
  - Statically do not know source thread, so must assume it can be any thread



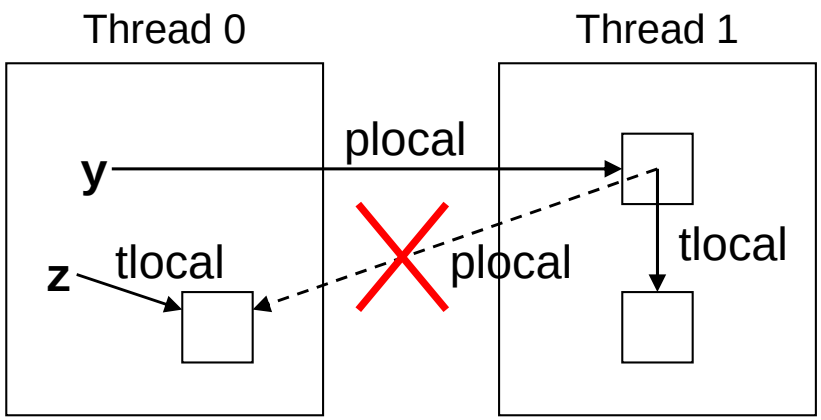
$$\frac{\Gamma \subset \mathbf{e_1} : \tau \qquad \Gamma \subset \mathbf{e_2} : \mathbf{int}}{\Gamma \subset \mathbf{transmit}\ \mathbf{e_1}\ \mathbf{from}\ \mathbf{e_2} : \mathit{expand}(\tau, \mathit{global})}$$

$$\mathit{expand}(\mathit{ref}_q\ \tau, q') = \mathbf{ref}_{w(q, q')}\ \tau$$
$$\mathit{expand}(\tau, q') = \tau \ \text{otherwise}$$

Type Rules – Dereferencing Assignment

- The expression $e_1 := e_2$ puts the value of e_2 into the location referenced by e_1 (like $*e_1 = e_2$ in C)
- Some assignments are unsound

$$\frac{\Gamma \subset e_1 : \text{ref}_q\ \tau \qquad \Gamma \subset e_2 : \tau \qquad \textit{robust}(\tau, q)}{\Gamma \subset e_1 := e_2 : \text{ref}_q\ \tau}$$

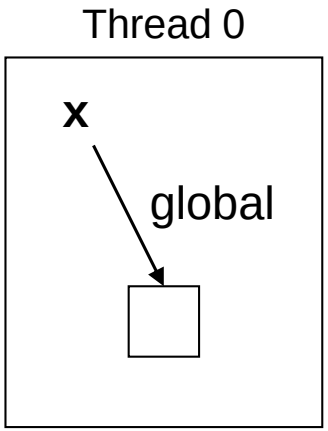


robust($\text{ref}_q\ \tau$, q') ≔ ***false*** if q ⊂ q'

robust(τ, q') ≔ ***true*** otherwise

Type Rules – Type Conversion

- **The expression `convert(e, q)` is an assertion that e refers to data that is no further than q**
  - Titanium code often checks if data is `plocal` and then casts to it before operating on it for efficiency



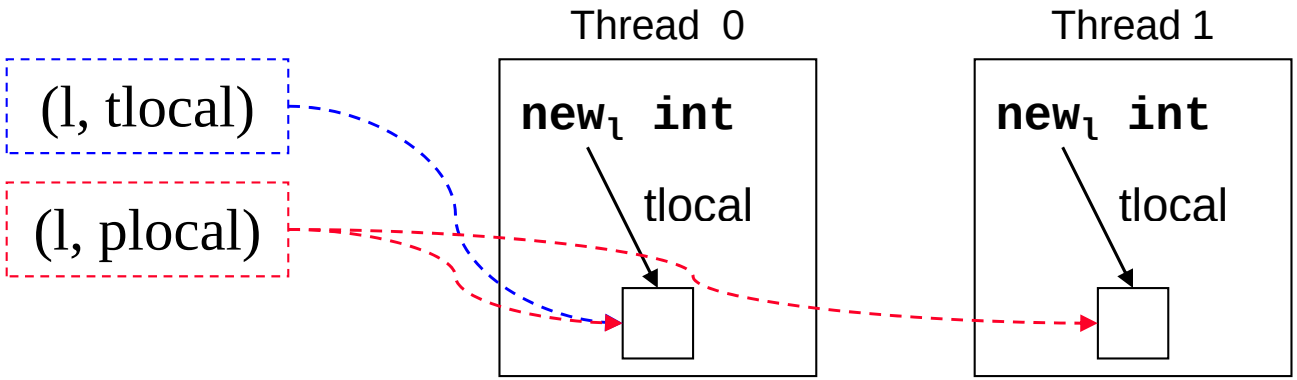
$$\frac{\Gamma \subset \mathbf{e} : \mathbf{ref}_q, \tau}{\Gamma \subset \mathbf{convert(e, q)} : \mathbf{ref}_q\ \tau}$$

Pointer Analysis

- **Since language is SPMD, analysis is only done for a single thread**
  - We use thread 0 in our examples
- **Each expression has a points-to set of *abstract locations* that it can reference**
- **Abstract locations also have points-to sets**

Abstract Locations

- **Abstract locations consist of label and qualifier**
  - A-loc `(l, q)` can refer to any concrete location allocated at label `l` that is at most distance q from thread 0



Pointer Analysis – Allocation and Communication

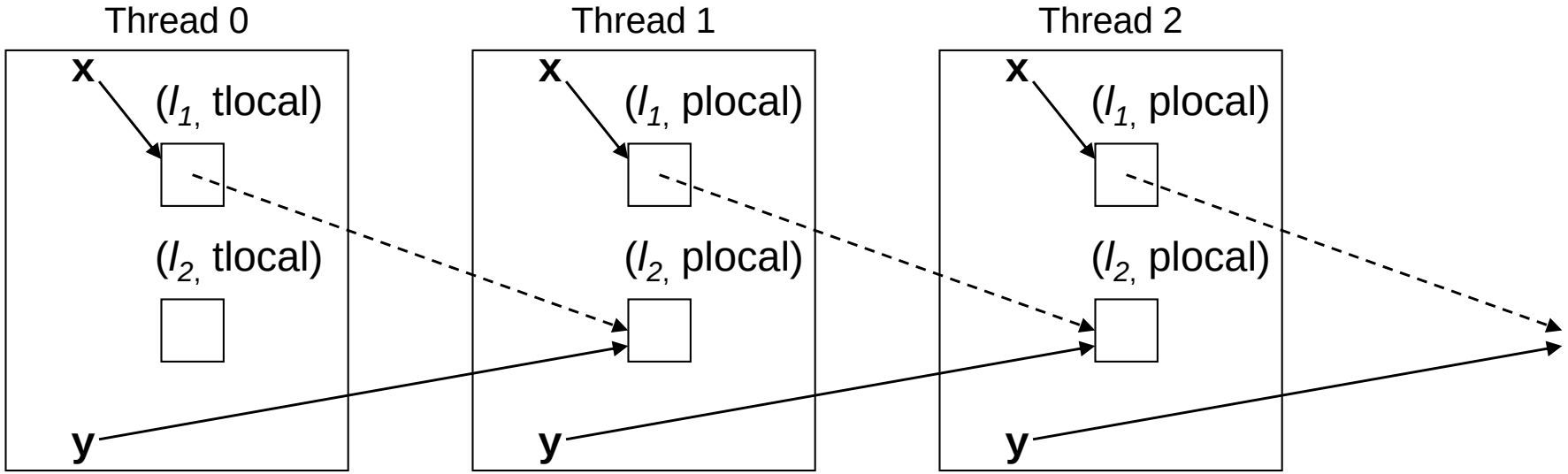
- **The inference rules for allocation and communication are similar to the type rules**
- **An allocation $\text{new}_l\ \tau$ produces a new abstract location `(l, tlocal)`**
- **The result of the expression `transmit` e_1 `from` e_2 is the set of a-locs resulting from e_1 but with global qualifiers**

$$e_1\ \$ \quad \{(l_1, \text{tlocal}), (l_2, \text{plocal}), (l_3, \text{global})\}$$

$$\text{transmit } e_1 \text{ from } e_2\ \$ \quad \{(l_1, \text{global}), (l_2, \text{global}), (l_3, \text{global})\}$$

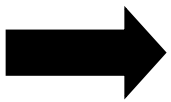
Pointer Analysis – Dereferencing Assignment

- **For assignment, must take into account actions of other threads**



$x \$ \{(l_1, \text{tlocal})\}$,

$y \$ \{(l_2, \text{plocal})\}$



$\mathbf{x} \parallel \mathbf{y}$: $(l_1, \text{tlocal}) \$ (l_2, \text{plocal})$,

$(l_1, \text{plocal}) \$ (l_2, \text{plocal})$,

$(l_1, \text{global}) \$ (l_2, \text{global})$

Pointer Analysis – Type Conversion

- **In the type conversion `convert(e, q)`, the program is illegal if e evaluates to a location further than q**
- **Thus, the result of the expression `convert(e, q)` is the set of a-locs resulting from e with the qualifiers reduced to at most q**

$$e \; \$ \quad \{(l_1, \text{tlocal}), (l_2, \text{plocal}), (l_3, \text{global})\}$$

$$\text{convert}(e, \text{plocal}) \; \$ \quad \{(l_1, \text{tlocal}), (l_2, \text{plocal}), (l_3, \text{plocal})\}$$

Evaluation

Benchmarks

- **Five application benchmarks used to evaluate the pointer analysis**

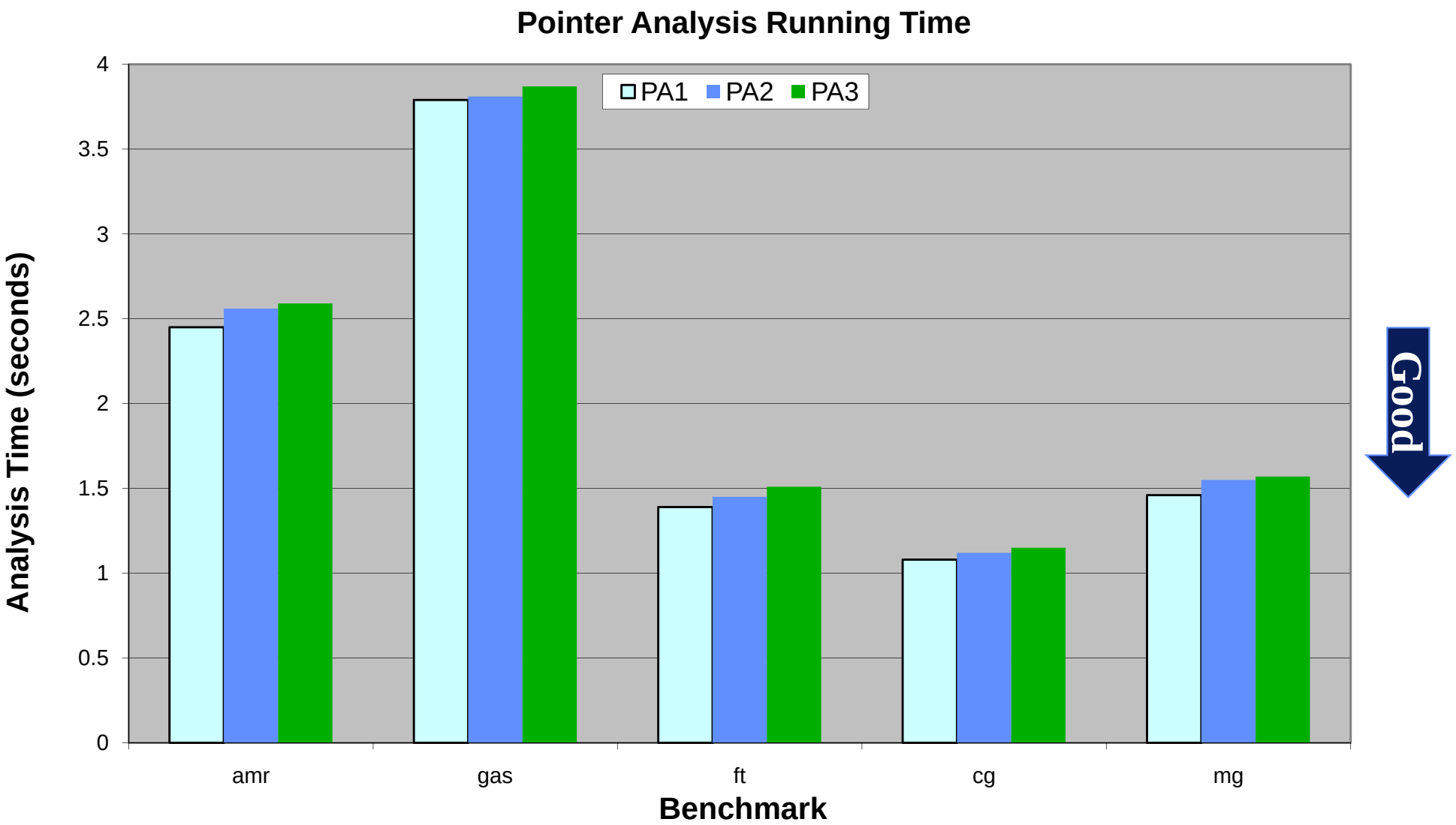
Benchmark	Line Count	Description
amr	7581	Adaptive mesh refinement suite
gas	8841	Hyperbolic solver for a gas dynamics problem
ft	1192	NAS Fourier transform benchmark
cg	1595	NAS conjugate gradient benchmark
mg	1952	NAS multigrid benchmark

Running Time

- **Determine actual cost of introducing multiple levels into the pointer analysis**
- **Tests run on 2.4GHz Pentium 4 with 512MB RAM**
- **Three analysis variants compared**

Name	Description
PA1	Single-level pointer analysis
PA2	Two-level pointer analysis (thread-local and global)
PA3	Three-level pointer analysis

Running Time Results

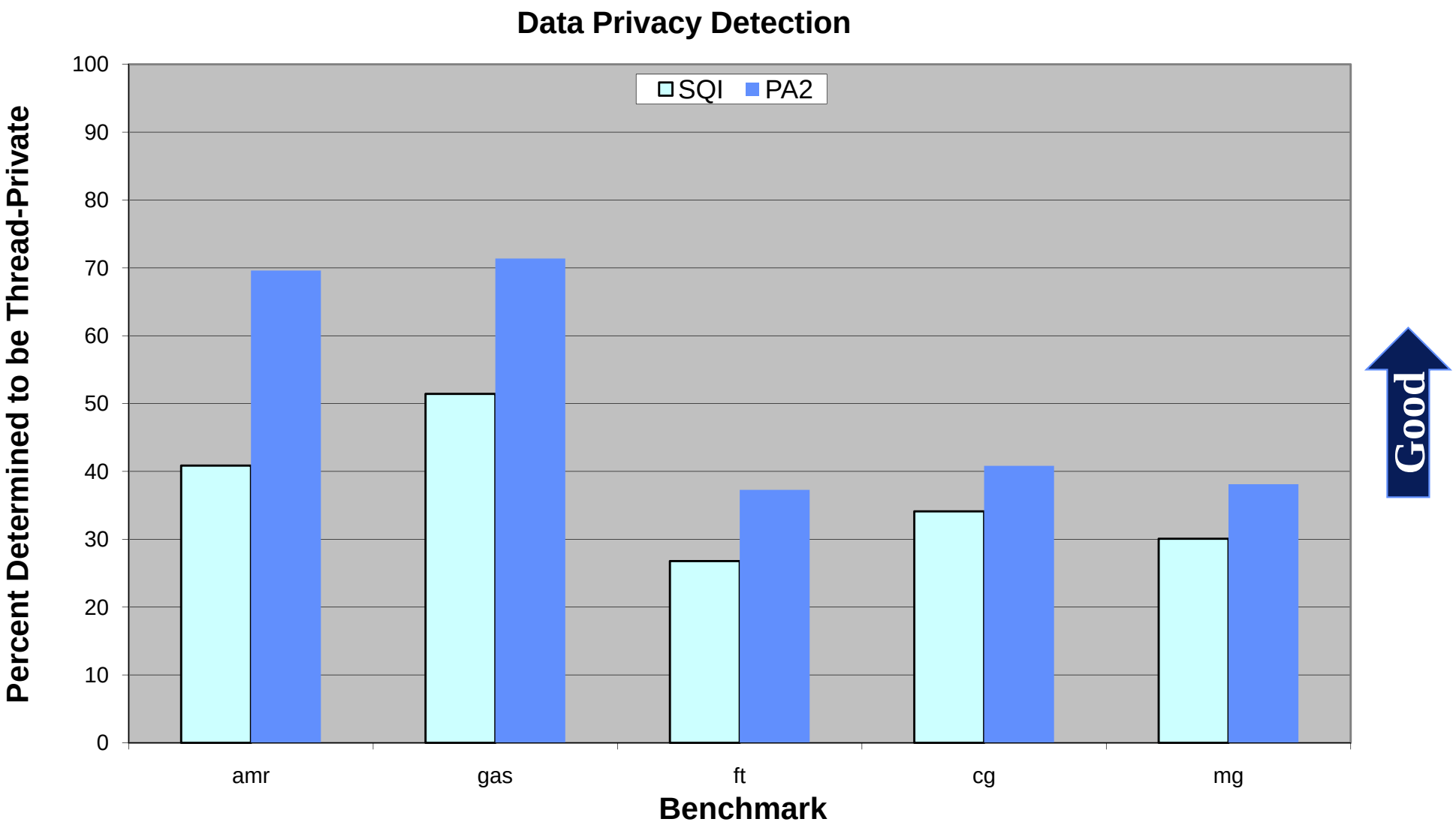


Data Privacy Detection

- **In pointer analysis, an allocation site is private if only thread-local references to it are used**
  - Thus, only two levels, thread-local and global, needed in the pointer analysis
- **Two types of analysis compared**

Name	Description
SQI	Constraint-based analysis by Liblit, Aiken, and Yelick; does not distinguish allocation sites
PA2	Two-level pointer analysis (thread-local and global)

Data Privacy Detection Results

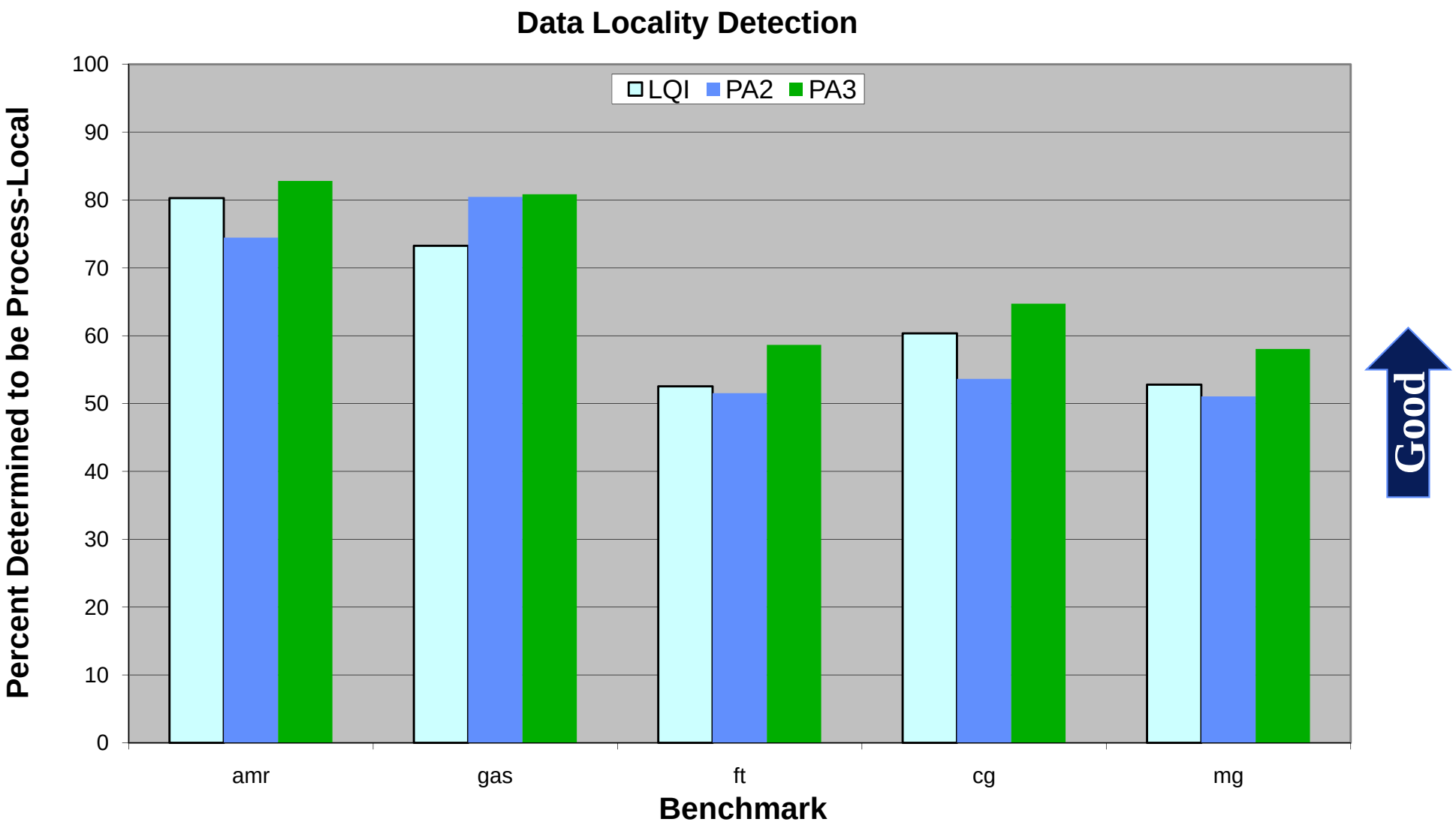


Data Locality Detection

- **Goal: statically determine which pointers must be process-local**
- **Three analyses compared**

Name	Description
LQI	Constraint-based analysis by Liblit and Aiken; does not distinguish allocation sites
PA2	Two-level pointer analysis (thread-local and global)
PA3	Three-level pointer analysis

Data Locality Detection Results

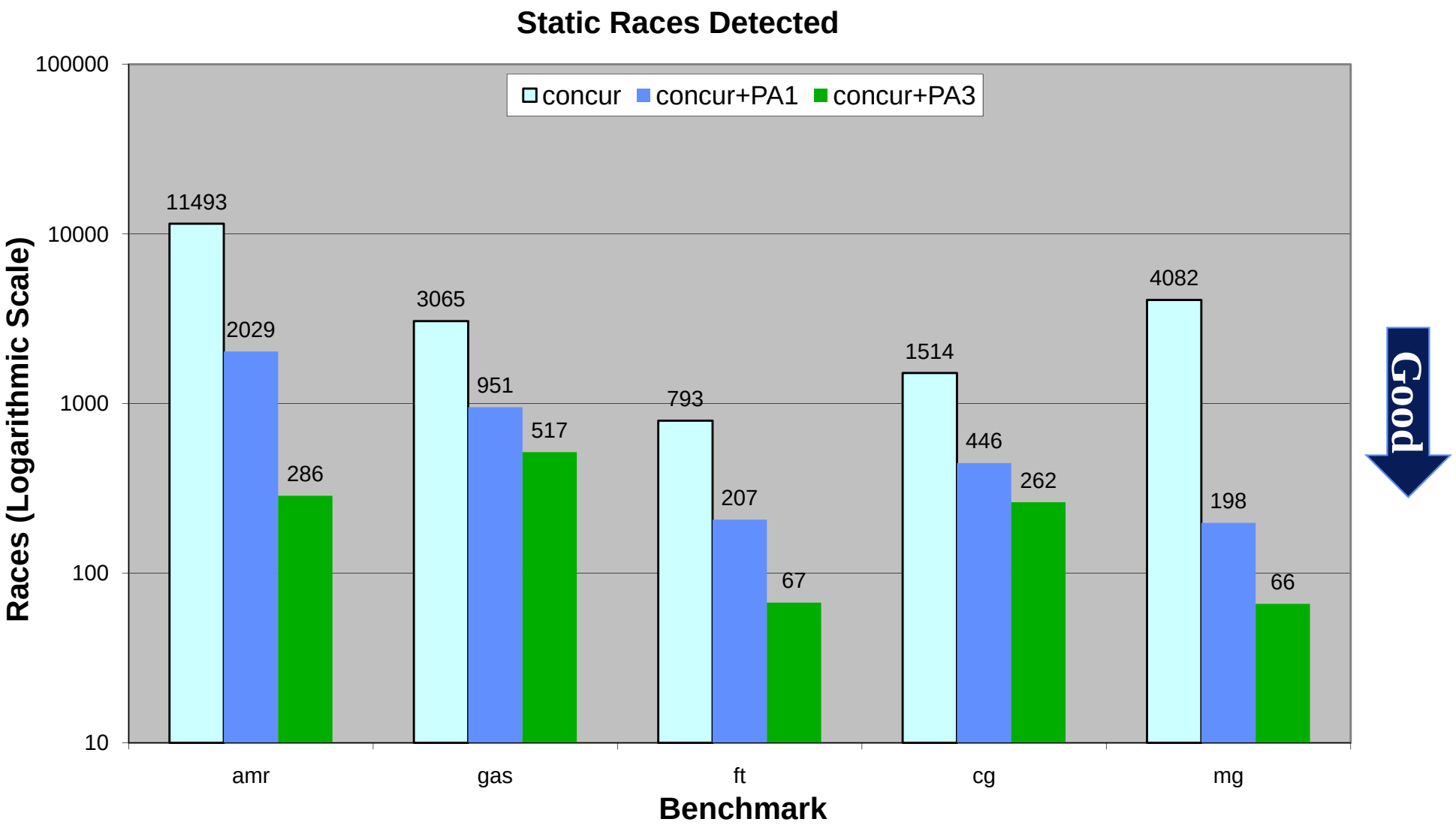


Race Detection

- **Pointer analysis used with an existing concurrency analysis to detect potential races at compile-time**
- **Three analyses compared**

Name	Description
concur	Concurrency analysis plus constraint-based data sharing analysis and type-based alias analysis
concur+PA1	Concurrency analysis plus single-level pointer analysis
concur+PA3	Concurrency analysis plus three-level pointer analysis

Race Detection Results



Conclusion

Conclusion

- **We developed a pointer analysis for hierarchical, distributed machines**
- **The cost of introducing the memory hierarchy into the analysis is small**
- **On the other hand, the payoff is large**

Future Work

- **Scientific programs tend to use a lot of array-based data structures**
  - Need array index analysis to properly analyze them
- **Implement a dynamic race detector**
  - Use static results to minimize the program locations that need to be tracked

Questions