Reconciling Responsiveness with Performance in Pure Object-Oriented Languages

Urs Hölzle

David Ungar

Outline

- Self-93 System Overview
- Novel Optimization approaches
  - Type feedback
    - Use of profile information
  - Adaptive recompilation
    - Responsiveness
    - Performance

Self-93 System

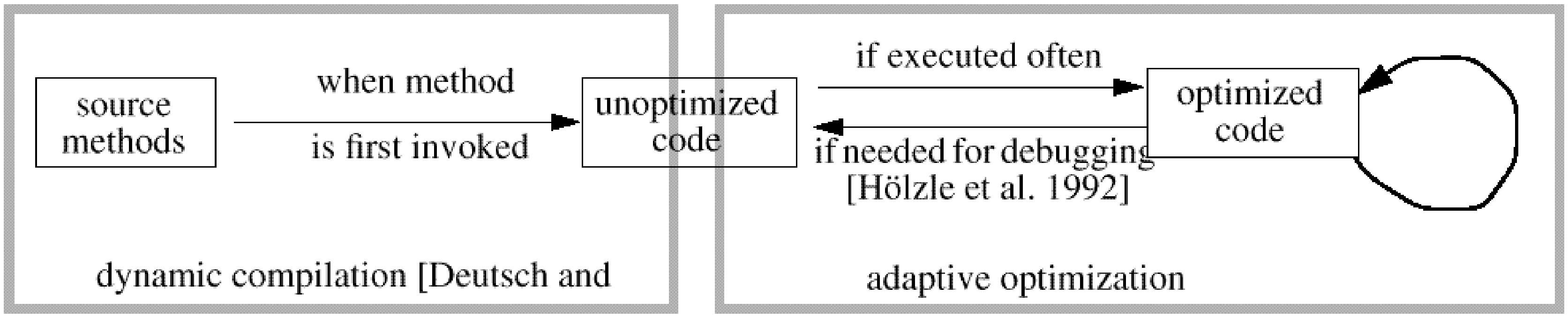


Figure 1. Compilation in the SELF-93 system

- Terminology
  - Dynamic Compilation
    - “Jit”
  - Adaptive Compilation

Type Feedback

- Profile Program
  - Receiver types
  - Frequency
- Profile Guided Optimization
  - Predict and inline dynamically dispatched calls
  - Splitting
  - Uncommon branch elimination

Inlining Strategies

- Not all calls should be inlined
  - Inlining A may require B to be inlined to reduce closure costs
  - May increase register pressure too much.
  - ???
- Self-93 currently inlines when
  - Callee is small
  - Caller not too big

Type Feedback: Benchmarks

	Benchmark	Size[a]	Description
small benchmarks	DeltaBlue	500	two-way constraint solver [Wilson and Moher 1989] developed at the University of Washington
	PrimMaker	1100	program generating "glue" stubs for external primitives callable from SELF
	Richards	400	simple operating system simulator originally written in BCPL by Martin Richards
large benchmarks	CecilComp	11,500	Cecil-to-C compiler compiling the Fibonacci function (the compiler shares about 80% of its code with the interpreter, CecilInt)
	CecilInt	9,000	interpreter for the Cecil language [Chambers 1993] running a short Cecil test program
	Mango	7,000	automatically generated lexer/parser for ANSI C, parsing a 700-line C file
	Typeinf	8,600	type inferencer for SELF [Agesen et al. 1993]
	UI1	15,200	prototype user interface using animation techniques [Chang and Ungar 1993][b]
	UI3	4,000	experimental 3D user interface[b]

Table 2: Benchmark programs

Type Feedback: Performance

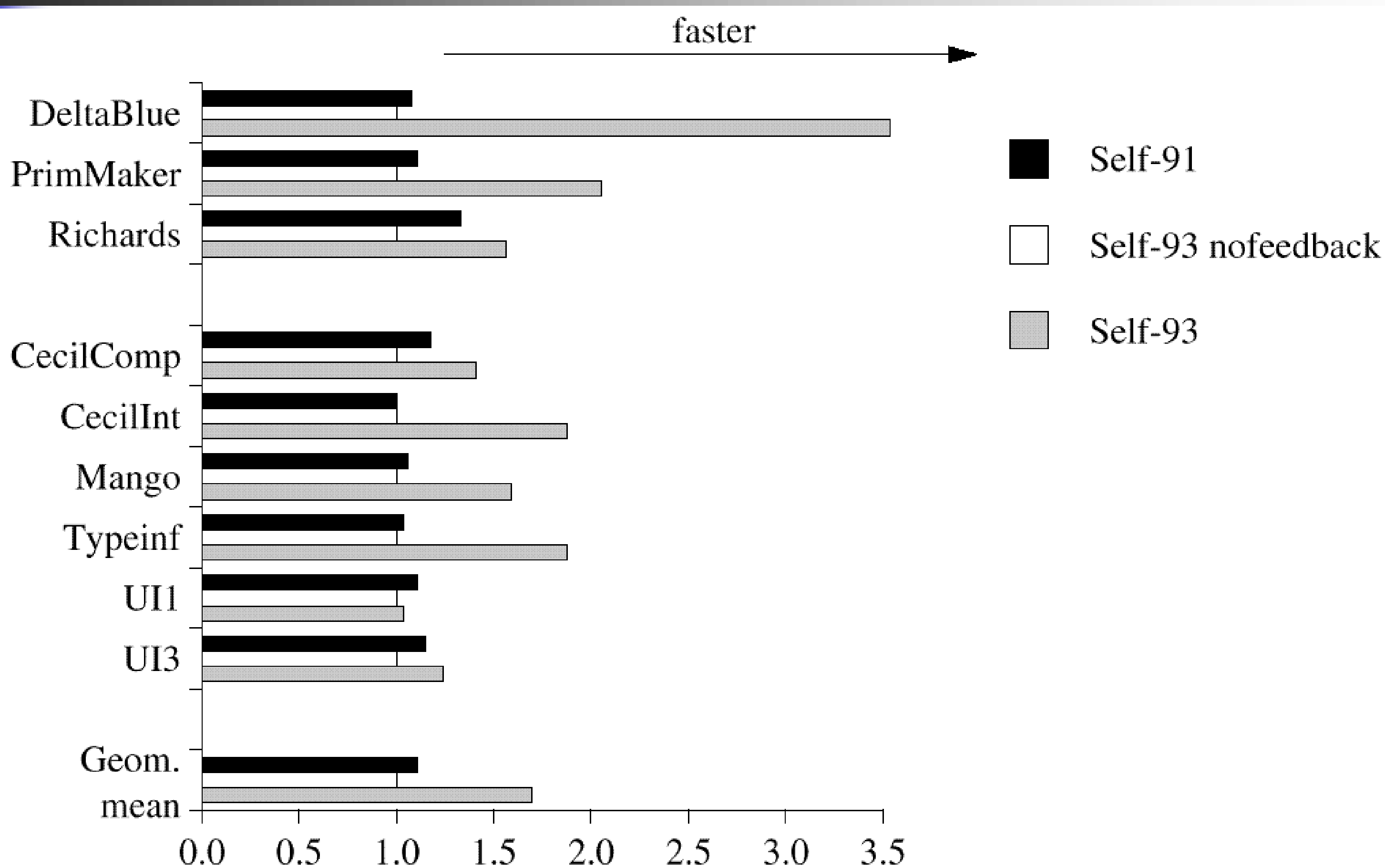


Figure 3. Performance impact of type feedback (all speeds relative to SELF-93-nofeedback)

Self-93 Performance Relative to Other Systems

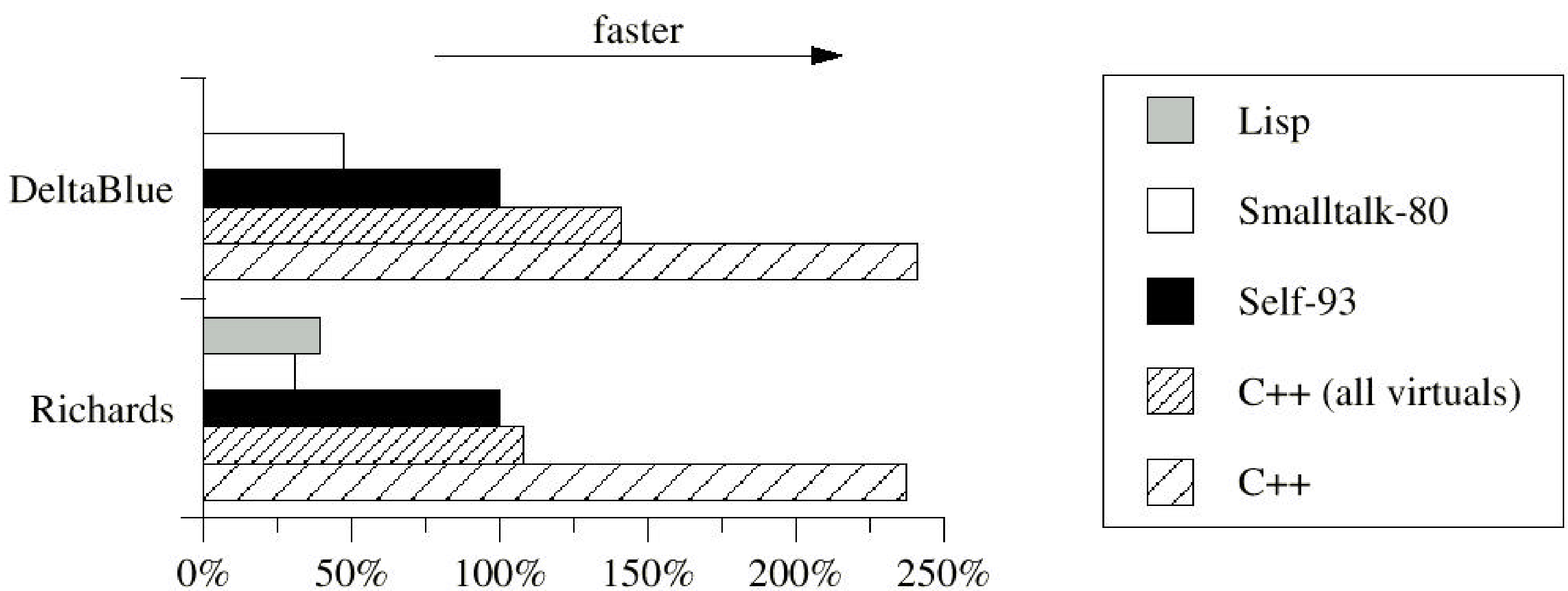


Figure 9. Execution speed (SELF-93 = 100%)

Self-93 Size

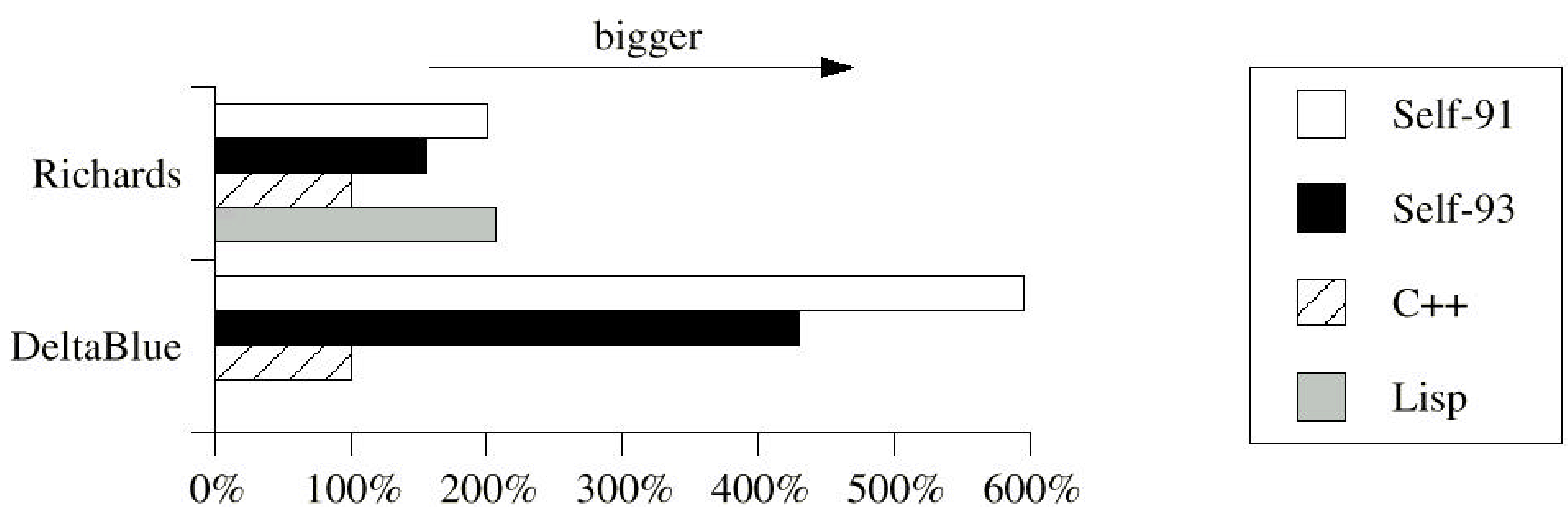


Figure 10. Code size relative to GNU C++

Type Feedback: Applicability for Other Systems

- Static compilation model
  - Actually advantageous
    - Has complete information
    - Compile time not an issue
  - One disadvantage
    - Cannot adapt to unforeseen circumstances
- Other Languages
  - Expect similar results
  - Not quite as extreme

Adaptive Compilation

- Goal
  - Achieve reasonable performance without introducing pauses
  - Emphasis: pause free execution
- Approach
  - Optimize only hot spots

When to Recompile?

- Ideal policy
  - Recompile a method only if it reduces total execution time.
  - Do so as early as possible
- Impossible to implement
  - Cannot predict future
  - Needs time to accumulate profile
  - Ignores interaction effects

When to Recompile: Self-93 Strategy

- Approximates ideal policy
  - Assume past predicts future
  - Method invocation counts
    - Counter > threshold triggers recompile
  - Eventually all methods trip counter
- Exponential Decay Mechanism
  - Decay counters every n seconds
    - Decay rate: half life time
  - Result
    - Frequency more important than total calls

Exponential Decay

- Is exponential decay the correct model
  - Misses infrequently executed methods
  - Invocation limit should vary per method
  - Half life time relative to machine speed?
  - Real time/cpu time
- Bottom line
  - Looks depressing
  - Simple counter strategy works great.
    - “What” more important than “when”

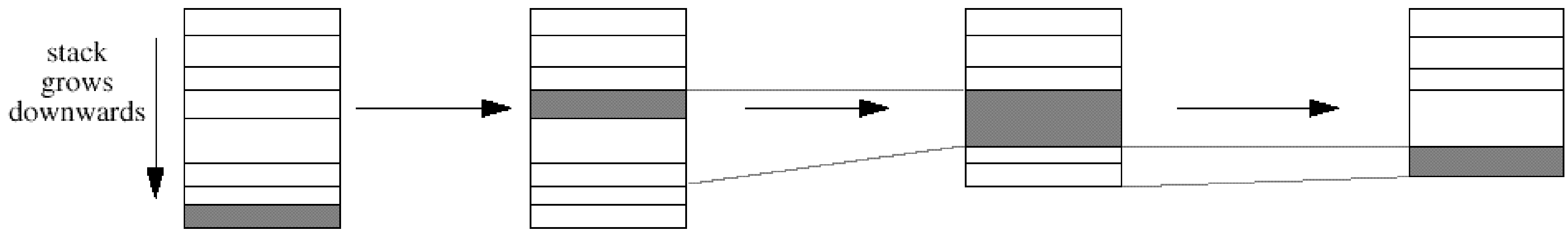
Other possibilities for “When”

- Edge counters
  - Not practical in Self
    - Everything is a call
    - Too much space
- PC sampling
  - Discover time-consuming methods
  - Not practical in self
    - Too many small methods
- *May be good for other languages*

“What” to recompile?

- Method overflows counter
  - Recompile just that method?
    - Bad plan
    - Example: set/get method
  - Idea
    - Walk up current stack
    - Look for “good” candidate to recompile
    - Use dynamic info to make inlining decisions

Finding Method to Recompile



A method overflows its invocation counter and triggers a recompilation

The system inspects the stack to determine which method to recompile. Then, it calls the compiler to generate new code.

The system replaces the old (unoptimized) stack frames with the frame of the newly compiled method. In the example, it replaces three unoptimized frames with one optimized frame.

The system continues until all of the remaining stack is optimized. Here, it performs one more optimization which replaces the bottom two frames.

Figure 12. Optimization process

Finding Method

- Characteristics considered
  - m.size
    - Size of method m
  - m.count
    - # invocations of m
  - m.sends
    - # calls made from m (approx)
  - m.versions
    - # times m has been recompiled

After Recompilation

- Replace Method
  - If possible, even currently running version
- Note Benefit
  - If no inlining occurred
    - Avoid recompiling method again
- Hopefully
  - No pauses,
  - Good performance
  - No training runs necessary

Measuring Pauses in Interactive Systems

- What constitutes compile pause?
  - Back to back compilation
    - Appears as one pause
- Define: "pause cluster"
  - Any period of time which
    - Starts or ends with a compilation
    - Compilation consumes > 50% cluster's time
    - No compilation-free interval > .5 seconds

Pause Clustering Example

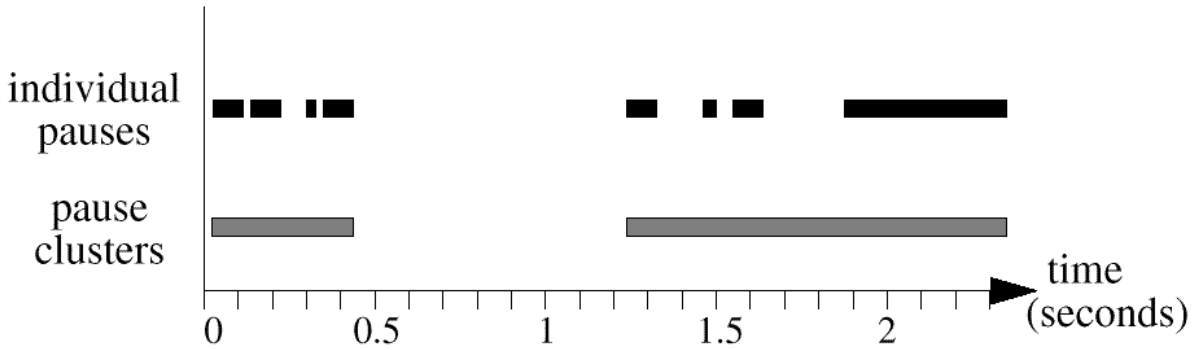
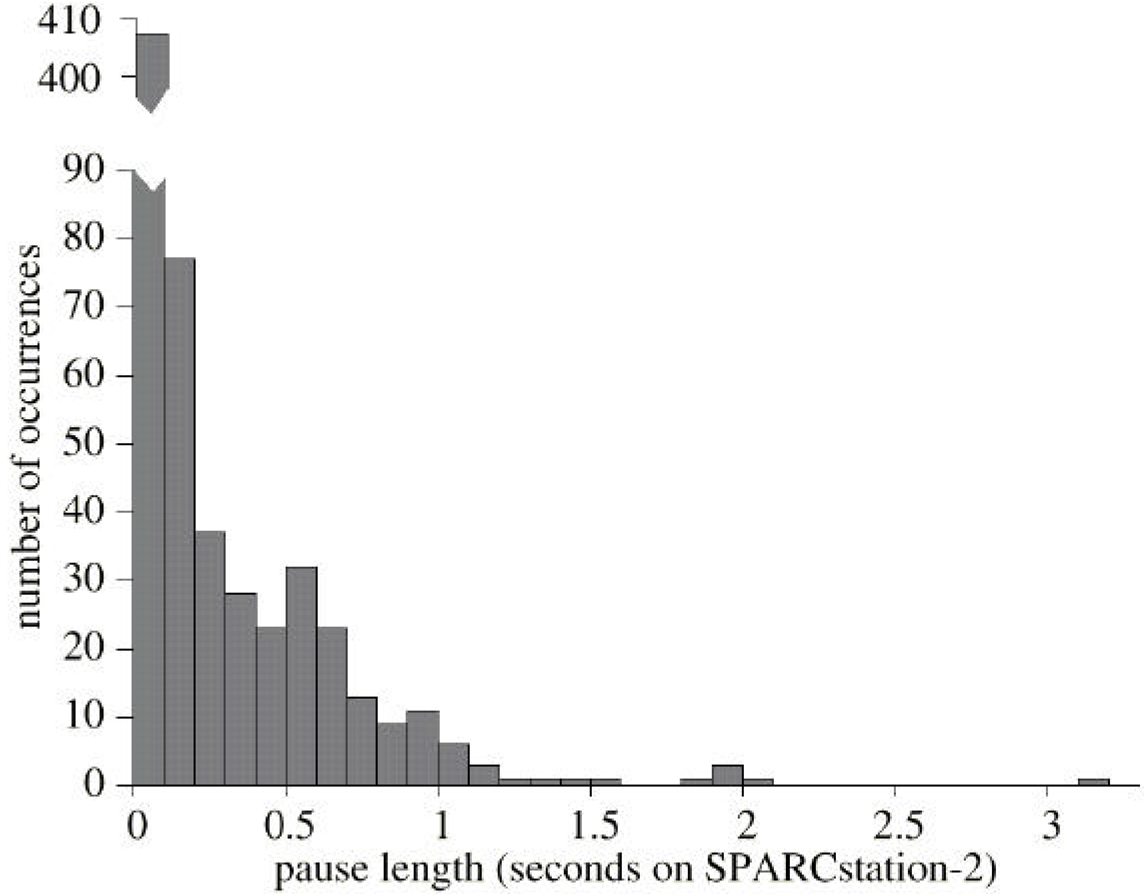


Figure 13. Individual pauses and the resulting pause clusters

- Makes big difference
  - W/out: < 2% exceed 0.1 sec
  - With: 37%

Evaluation of Interactive Behavior



15. Compile pauses during a 50-minute interaction

Evaluation of Interactive Behavior *(continued)*

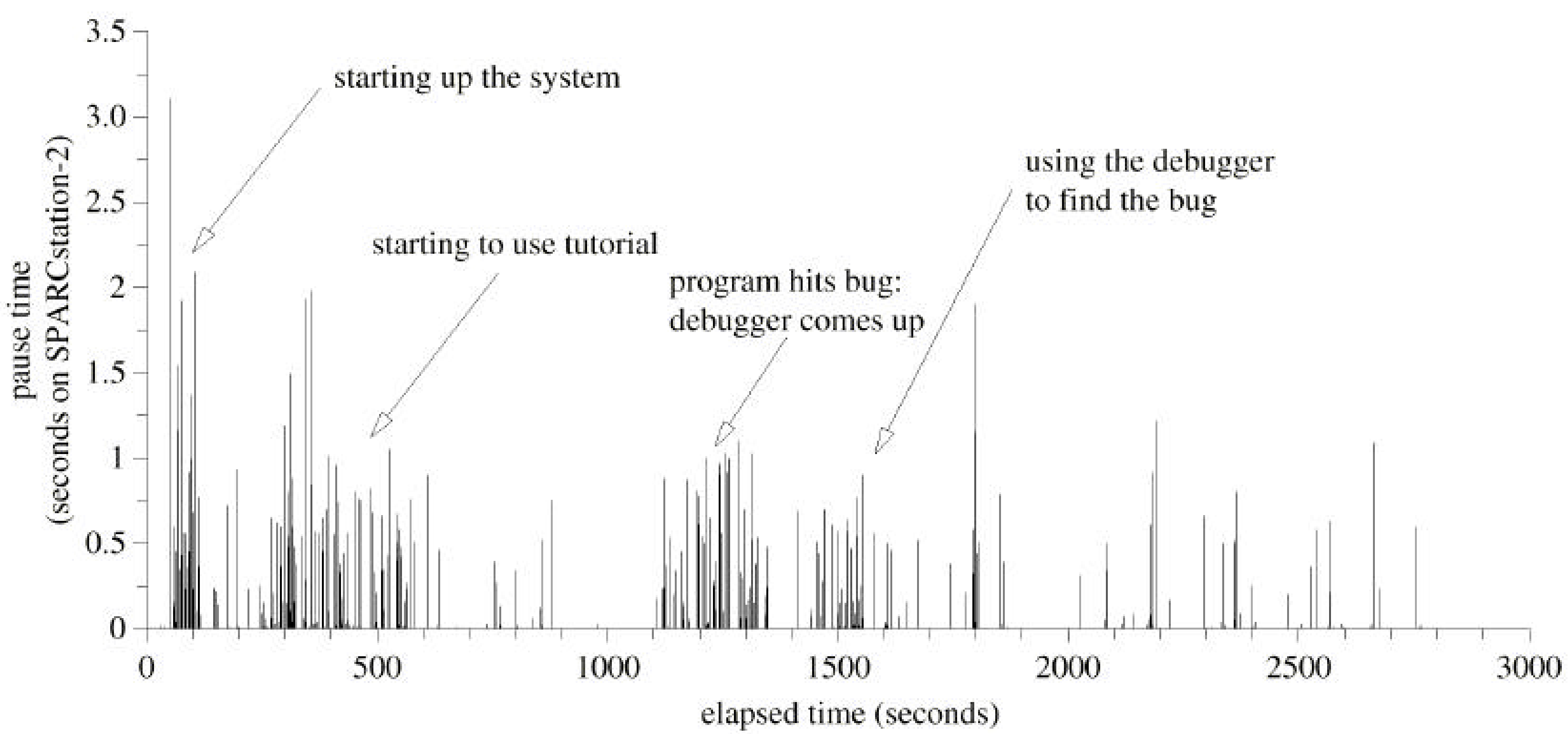


Figure 16. Distribution of compilation pauses over time

Startup Time

		program size	
		small	big
execution time	short	good	not good
	long	good	good

Table 6: Start-up behavior of dynamic compilation

Startup time *(continued)*

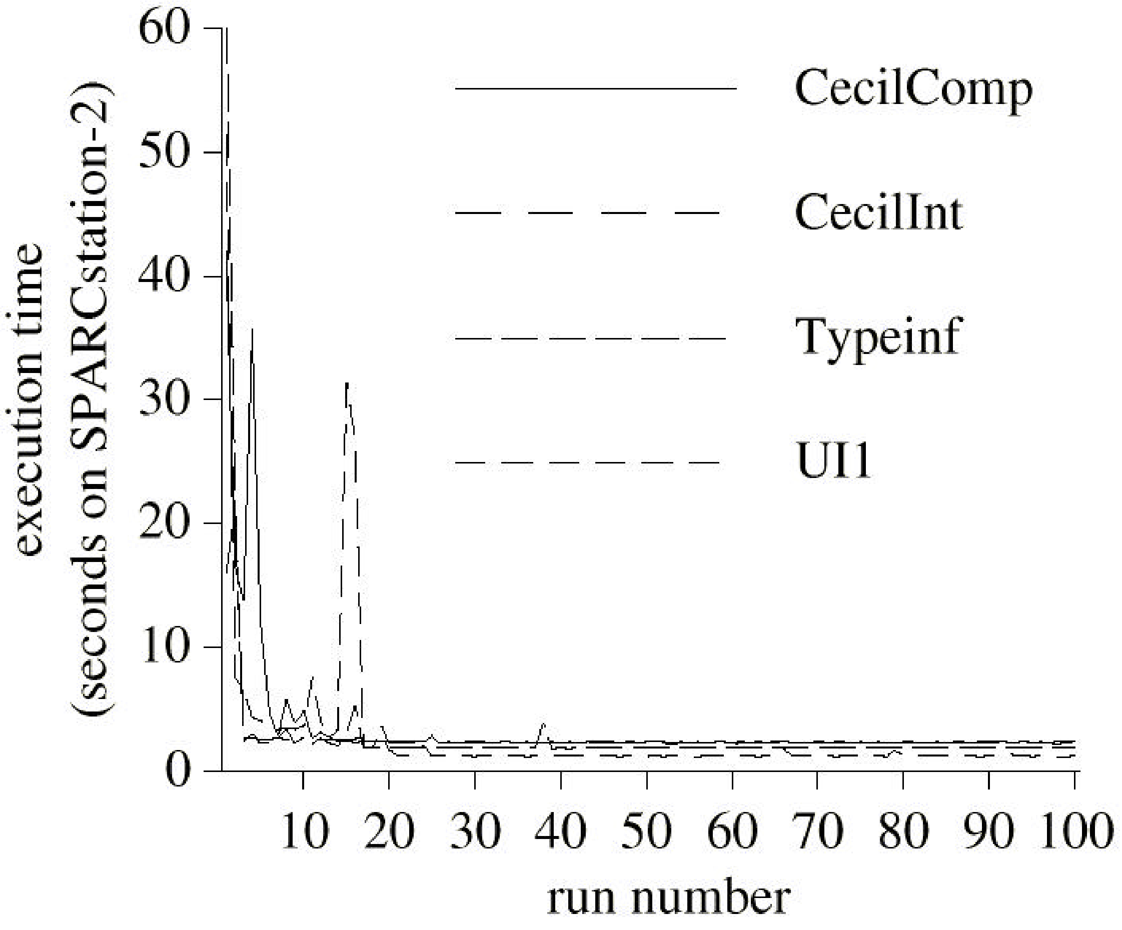


Figure 17. Start-up behavior of large applications

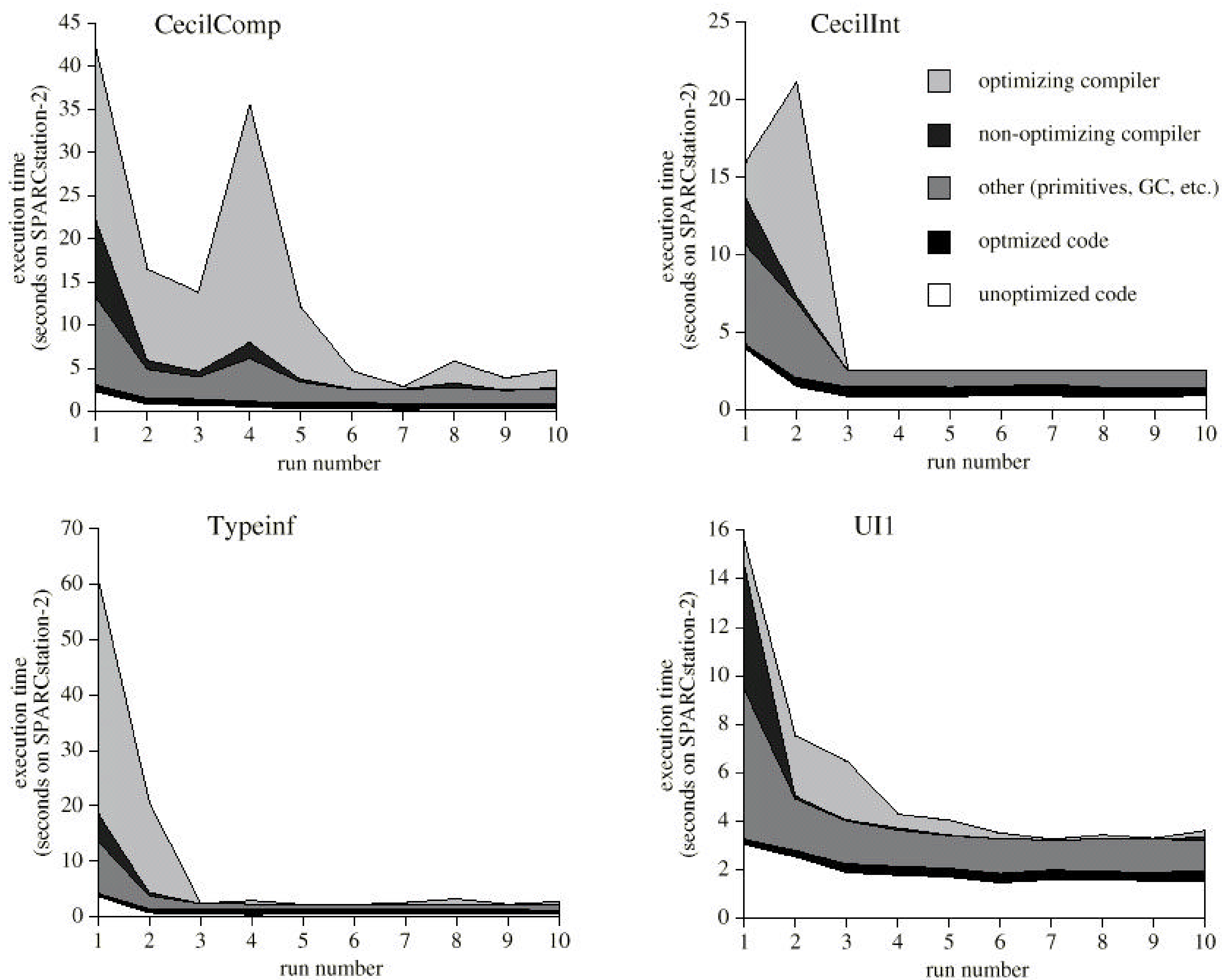
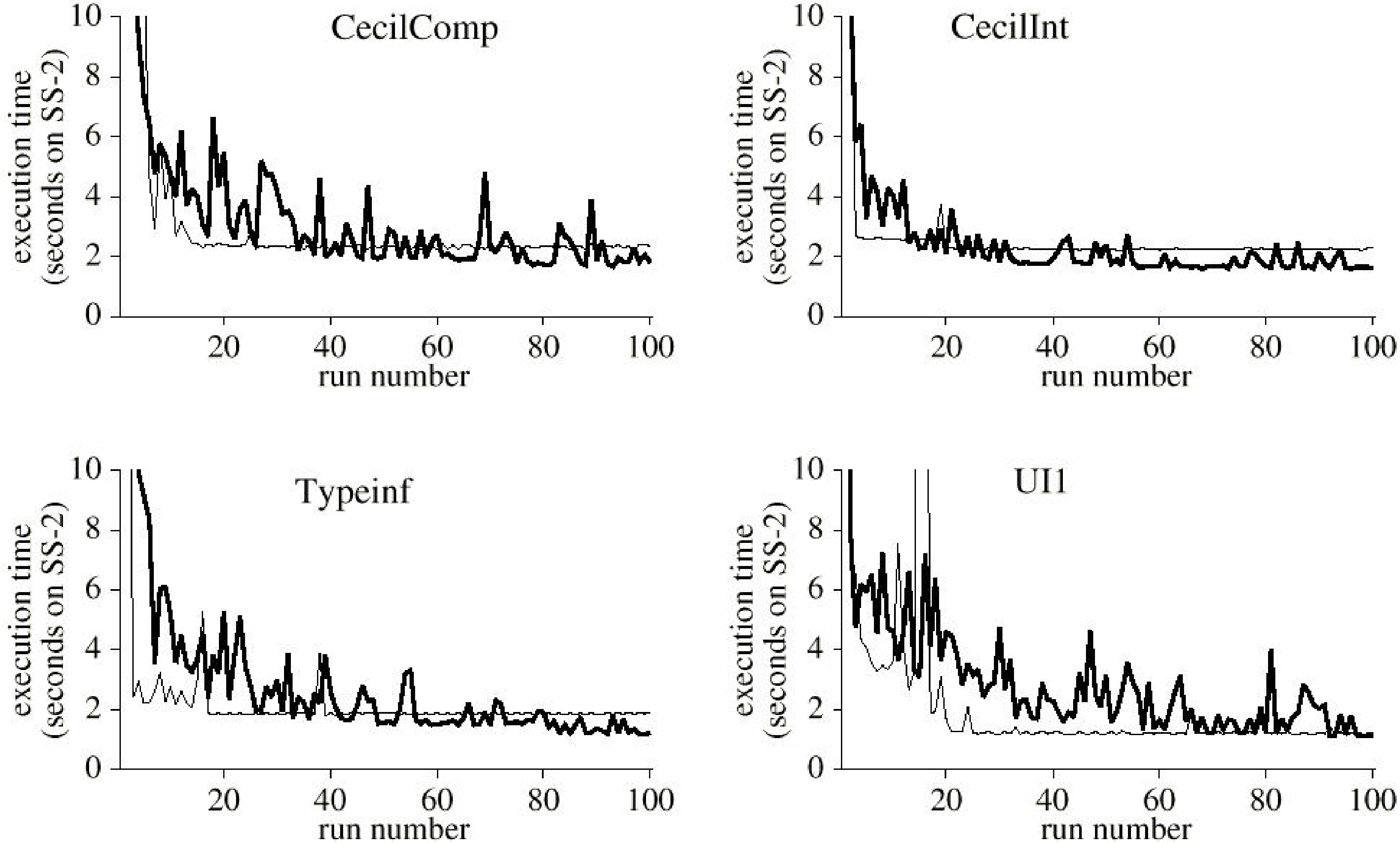


Figure 20. Start-up phase of selected benchmarks

Recompilation Parameters Decay vs No Decay



Conclusions

- Adaptive Recompilation
  - Good runtime performance
  - Good interactive performance
- Pause Clustering
  - Measures pauses as seen by user
- Type Feedback
  - 1.7 times faster than without

Future Work

- Compile during free cycles
- Java Java Java...
  - Same techniques
  - New techniques