TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute for Computer Engineering

Outline of Scope

Efficient Method Invocation on an Embedded Bytecode Processor by Autonomous Functional Hardware Submodules and Bytecode Pre-Processing

Thomas B. Preußer

Dresden, 02. Dezember 2009

Itinerary

- Embedded Java
- SHAP
  - Stack Module
  - Method Cache
  - Interface Dispatch
- Summary

Embedded Java
Motivation

- Large pool of experienced programmers,
- Increased programmer productivity (Hardin, aJile Systems: 25-40%) through:
  - the API: extensive standard libraries,
  - the bytecode: portable (and small application code),
  - the memory model: no pointers, checked accessing, garbage collection; and
  - structured exception handling: debugability.

→ Reduced time to market.

- Widespread availability of platform:

→ Even projects for translators of native applications to Java bytecode (e.g. NestedVM, Cibyl).

Embedded Java
Implementation

- Software JVMs as regular applications (OS-based systems):
  Interpretation (KVM, JamVM) and AOT compilation (Excelsior) quite common next to JIT / DAC (CACAO, Jamaica).
- JVMs as OS replacements (JavaOS, JX).
- Architecture support for Java execution:
  - Interpretation acceleration: ARM's Jazelle DBX, Atmel's JEM.
  - Simplification of bytecode compilation: ARM's Jazelle RCT.
  - On-the fly bytecode-to-native translation by co-processor: JStar.
  - Native bytecode execution: Cjip, FemtoJava, picoJava, MAJC, Komodo, JOP, SHAP.

Embedded Java
Playground for Java Processors

+ Small memory footprint, no runtime compilation overhead, dense bytecode.
- Stack-oriented bytecode hiding ILP and causing significant amount of data movement.
- Bytecode folding (picoJava) expensive and not very effective.

→ JIT and DAC approaches on well-evolved RISC processors offer better performance and are the more flexible choice *unless* there are tight system constraints.

As research platforms:

- Full control over architectural features: exploration and evaluation.

Commercial Processors:

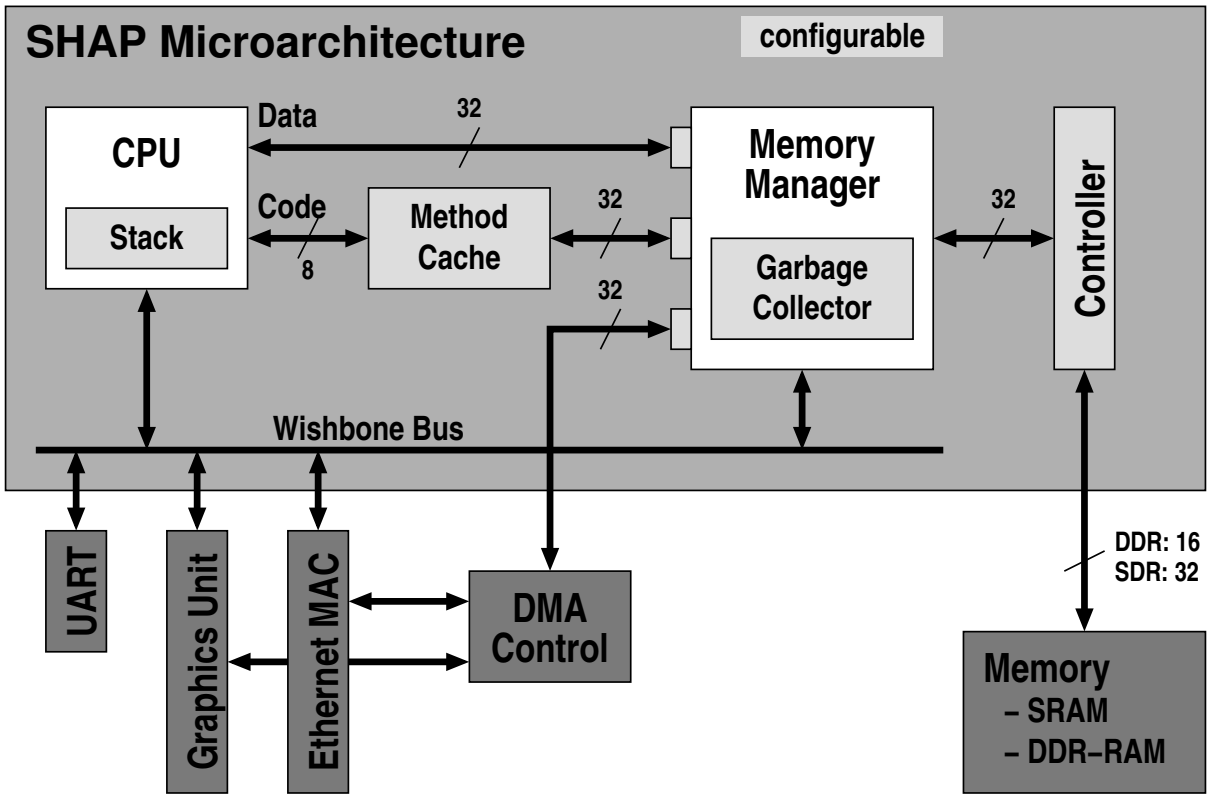
- SUN: picoJava unsuccessful, UltraJava built as graphics processor MAJC.
- Imsys: Cjip still around but no real Java.
- aJile: Next generation aJ-200 announced for this year.

Embedded Java

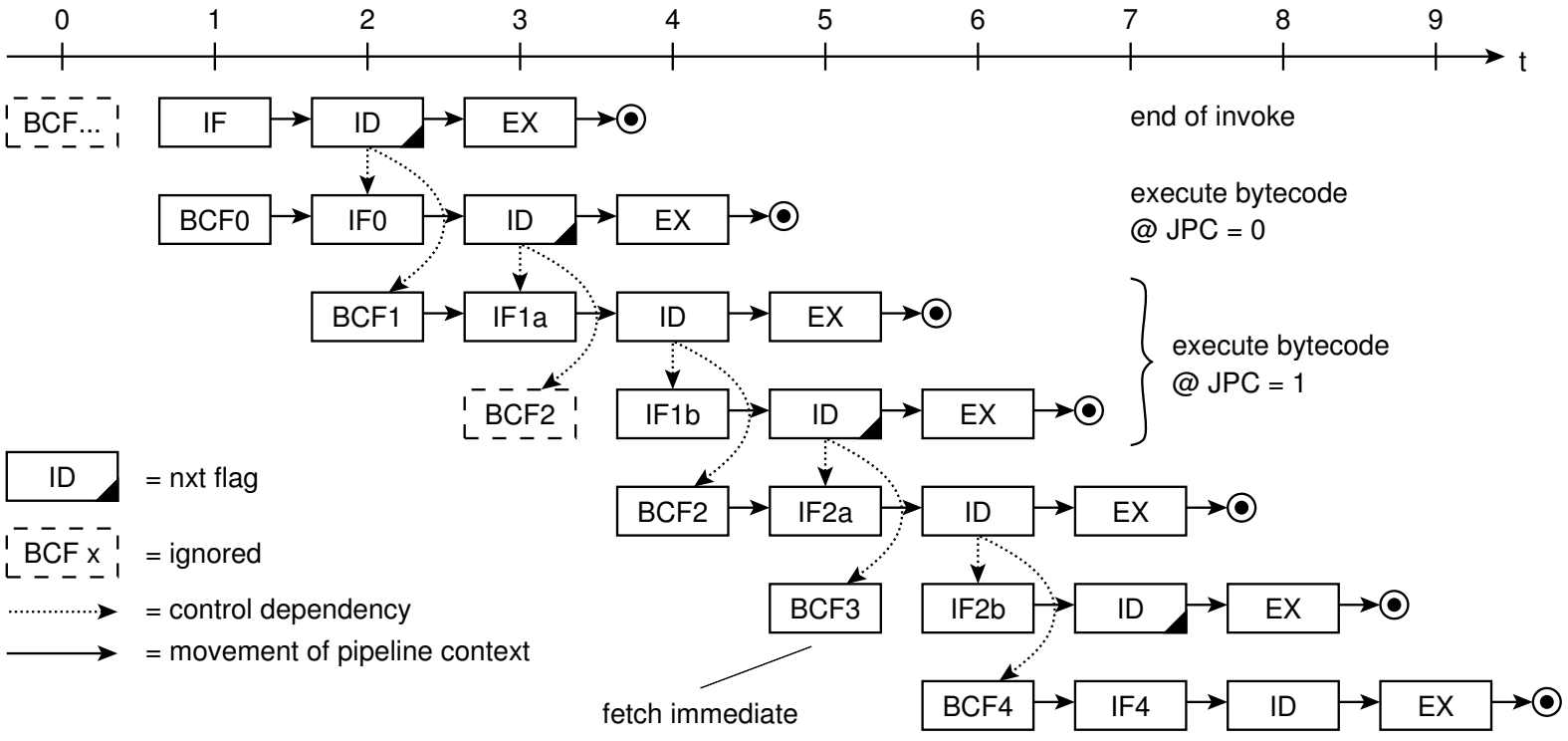
Java Processor Features

	FemtoJava	picoJava-II	Komodo	JOP	aJ-100	SHAP
Architectural Features						
Pipeline Depth	5	6	5	4	?	4
Instruction Folding	-	+	-	-	-	-
Microcode	-	+	+	+	+	+
Software Traps	-	+	+	+	+	+
Stack Realization	e	r	?	i	r	ii
Java Language Features						
Objects	-	+	+	+	+	+
Multiple Threads	-	+	+	(+)[a]	+	+
Scheduling	-	S	H	S	M	M
Synchronization	-	H/S	?	(M)[b]	M	M
Garbage Collector	-	S	S	S[c]	S	H
Runtime API						
CLDC	-	+	+	-	+	+

SHAP
Microarchitecture



SHAP
Execution Pipeline



SHAP
Features

Covered here:

- Multi-threaded stack with autonomous frame management.
- Predictable method caching with address translation.
- Constant-time interface method dispatch.

Others:

- Generic design with adjustable design parameters.
- Optimizing application pre-linking, and
- Java bootstrapping and online dynamic linking with relocation patching.
- Concurrent GC module with support for weak references and reference queues.

SHAP
Stack Module

Theses:

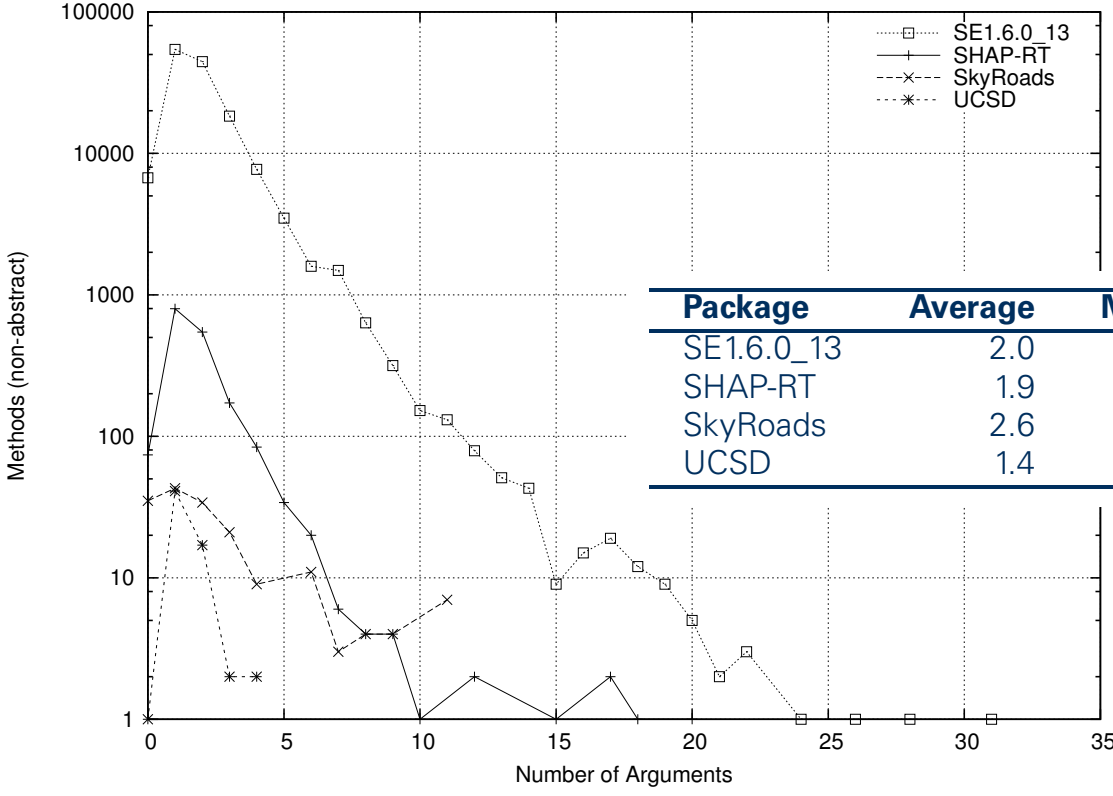
- Stack frames are small:
  - Stack only contains primitive and reference values.
  - All arrays and record structures on heap but not on stack.
- Random data access restricted to topmost frame:
  - Access to method arguments and local variables.
  - No address pointers to buried stack data.

Features:

- Registered two topmost stack entries `TOS` and `NOS`.
- Fast method frame construction and destruction in hardware.
- Autonomous management of frame, variable and stack pointers.
- Backed solely by on-chip memory.
- Dynamic allocation and switching of stacks of concurrent threads.

SHAP: Stack Module

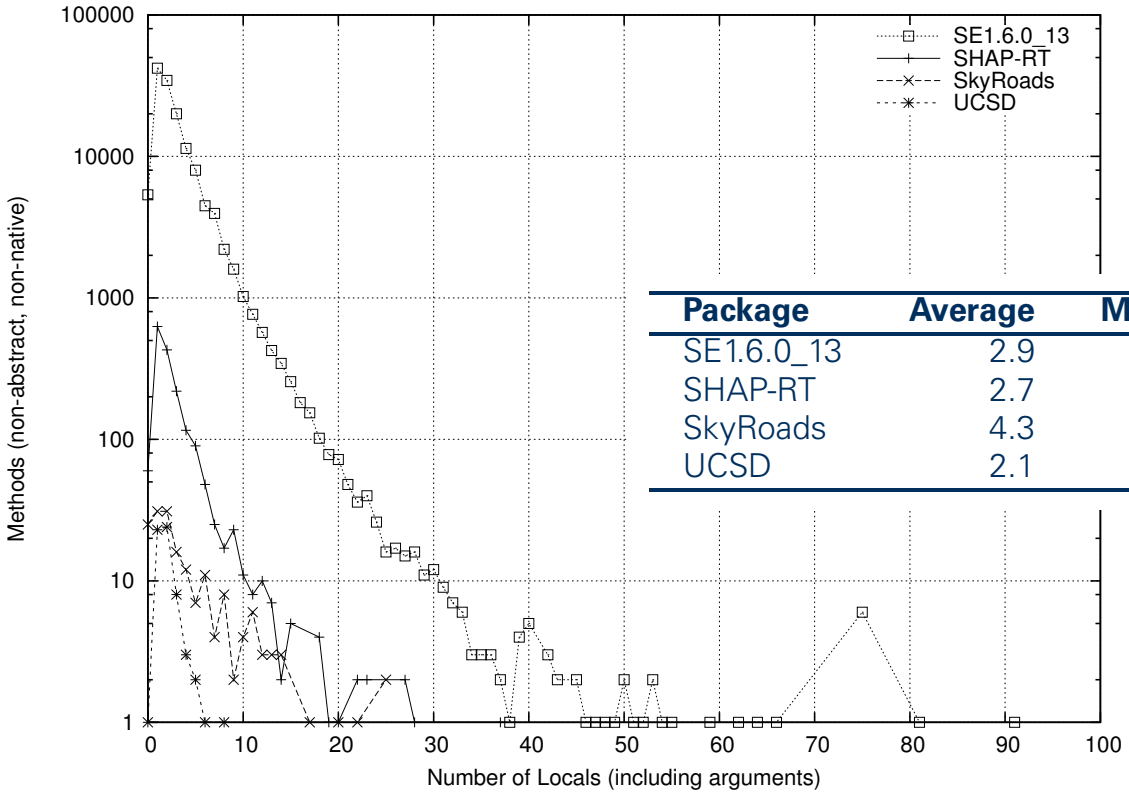
Distribution of Argument Counts



Package	Average	Median	Skew	Maximum
SE1.6.0_13	2.0	2	2.7	31
SHAP-RT	1.9	2	4.3	20
SkyRoads	2.6	2	1.6	11
UCSD	1.4	1	1.7	4

SHAP: Stack Module

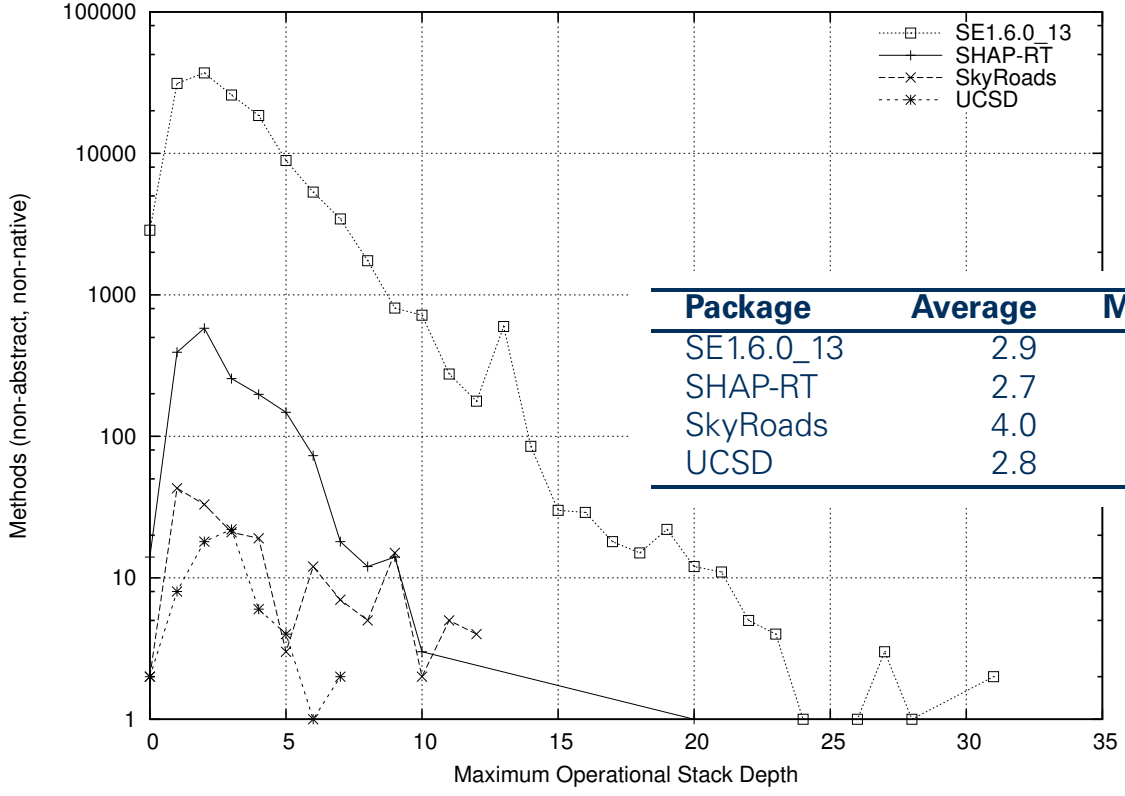
Distribution of Local Variable Counts



Package	Average	Median	Skew	Maximum
SE1.6.0_13	2.9	2	4.5	91
SHAP-RT	2.7	2	4.0	37
SkyRoads	4.3	2	1.9	25
UCSD	2.1	2	1.9	8

SHAP: Stack Module

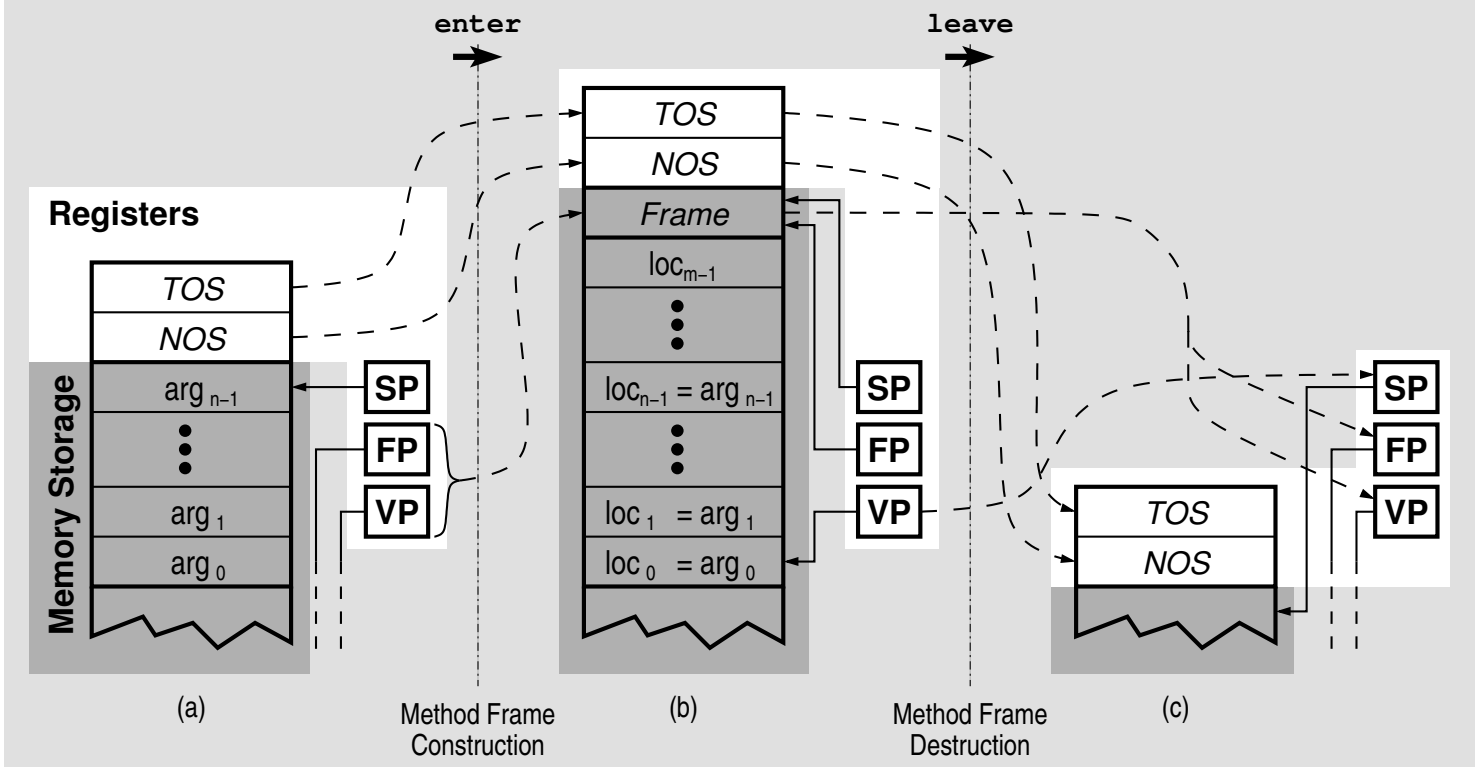
Distribution of Maximum Operand Stack Width



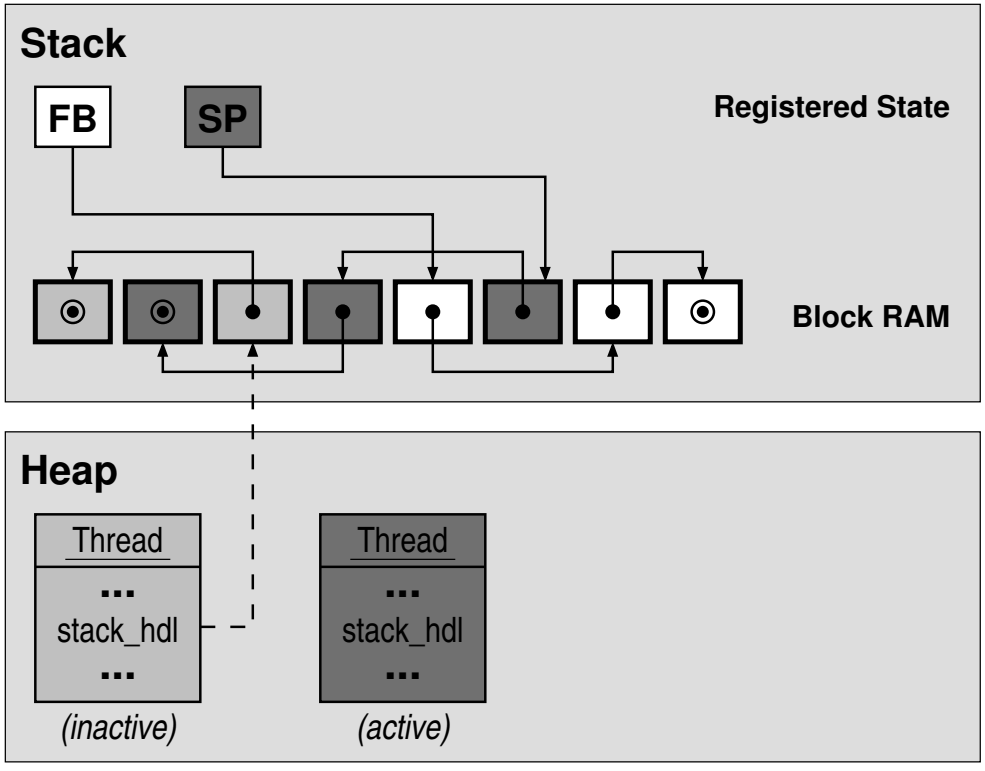
Package	Average	Median	Skew	Maximum
SE1.6.0_13	2.9	2	2.0	31
SHAP-RT	2.7	2	1.7	20
SkyRoads	4.0	3	1.0	12
UCSD	2.8	3	0.8	7

SHAP Stack Module

Frame Management



SHAP Stack Module
Thread Management



SHAP

Method Cache

Theses:

- Methods are typically small:
  - Delegation to other objects and their methods is widespread.
- Indirect recursions are rare:
  - If a method appears twice in the call hierarchy, then mostly adjacently.
  - Practical relevant exception: Composite Pattern.

Features:

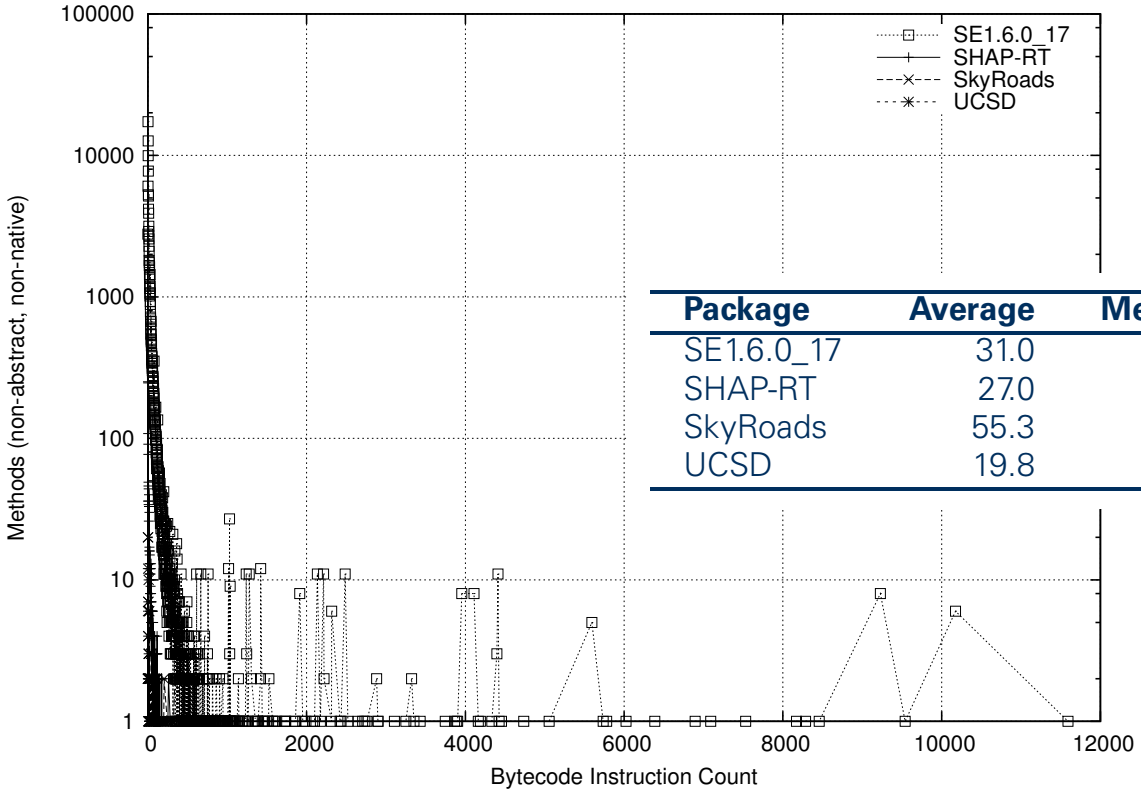
- Translation of bytecode offsets into memory addresses.
- Treats methods as whole entities.

Bump-pointer implementation:

- Deterministic miss / hit behavior without caller / callee aliasing.
- Reduced but powerful associativity with only three comparators.
- Compact storage of cached data.

SHAP: Method Cache

Java Method Sizes

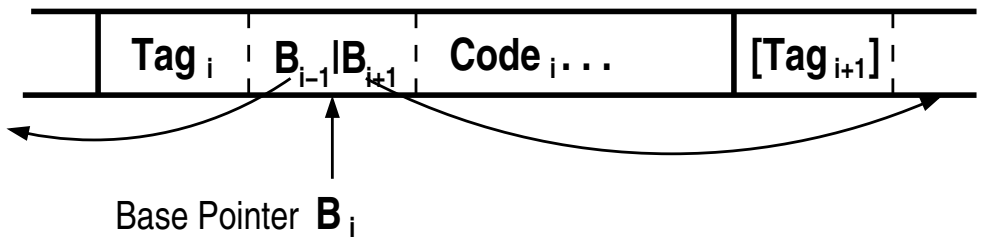


Package	Average	Median	Skew	Maximum
SE1.6.0_17	31.0	9	36.7	11591
SHAP-RT	27.0	8	5.9	720
SkyRoads	55.3	17	2.9	580
UCSD	19.8	10	2.4	118

SHAP: Method Cache

Cache Data Layout

Ring Buffer with Method Entries:



Additional State:

- `PB, CB, NB` – registered copies of previous, current and next method base pointer
- `PT, CT, NT` – corresponding registered tags
- `VP` – valid pointer: start of memory area not overridden by farthest extend of call hierarchy

Hits can be distinguished into:

- returning – one stack frame is unwound
- recursive – re-invocation of calling method
- invoking – re-invocation of previously invoked method

SHAP: Method Cache
Discussion

- Hit rates of about 90% comparable to a blocked cache with 4 blocks.
- Sliding window view on cache space is acceptable.
- Larger cache memory is only utilized by multi-slice implementation:
  - Slice size according to average call hierarchy.
  - Slices reduce thrashing in loops.
  - Number of comparators grows with slice number.
  - Slicing and coloring of slices may benefit multithreaded analysis.

SHAP: Method Cache

Anomalies with Multithreading

StringTest regularly ran faster when forked into extra Thread.

SHAP: Method Cache

Anomalies with Multithreading

StringTest regularly ran faster when forked into extra Thread.
Normal layout of method cache content (looped):

```
StringAtom.execute() ->
    StringBuffer.append() ->
        String.length()|
        String.getChars() -> System.arraycopy()
```

SHAP: Method Cache

Anomalies with Multithreading

StringTest regularly ran faster when forked into extra Thread.
Normal layout of method cache content (looped):

```
StringAtom.execute() ->
    StringBuffer.append() ->
        String.length() |
        String.getChars() -> System.arraycopy()
```

Cache layout after destructive interruption of forking Thread yielding:

```
System.arraycopy() ->
    String.getChars() ->
        StringBuffer.append() ->
            StringAtom.execute() |
            String.length()
```

Mutual replacement now limited to small methods.

SHAP
Method Dispatch – Multiple Inheritance

Approaches:

- Reflective Search
- Sparse Method Tables
- Hashed Lookup with Conflict Resolution

On statically-typed platforms:

- class-specific data structures
- problem size reduction by indirection through `itable`
  set of interface methods → set of interfaces

JVM Implementations:

Jalapeño (Jikes RVM)	hashed lookup in fixed-size IMTs
CACAO	sparse arrays of `itable` references
JOP/SableVM	sparse interface method tables
SHAP	`itable` attachment through type coercion

SHAP: Interface Method Dispatch

Interface Type Coercion

Coloring of *reference values* to identify class-specific `itable` for desired interface.
Goals:

- Automatic coercion by compiler or classfile loader.
- Allow use of standard / legacy classfiles.
- Coloring must be transparent to all but interface-related operations.

SHAP: Interface Method Dispatch

Interface Type Coercion

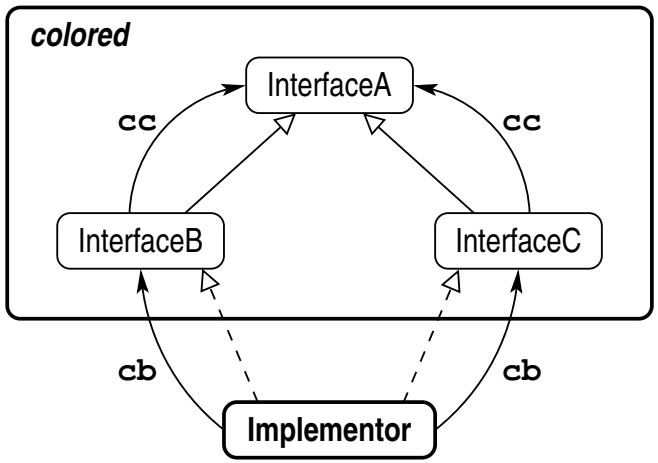
Coloring of *reference values* to identify class-specific `itable` for desired interface.

Goals:

- Automatic coercion by compiler or classfile loader.
- Allow use of standard / legacy classfiles.
- Coloring must be transparent to all but interface-related operations.

Enabled by:

- DFA based on type meta-data found in classfiles using ASM.
- Injection of constant-time coercion bytecodes.
- Reference value representation as (Object, Color) tuple.



SHAP: Interface Method Dispatch

Data Flow Analysis

Construction of data flow graph by abstract interpretation.

Source Verteces

- method arguments
- values returned by a method call
- successful `checkcasts`
- caught exceptions

Inner Verteces

- stack values
- values in local variables (including arguments)

Sink Verteces

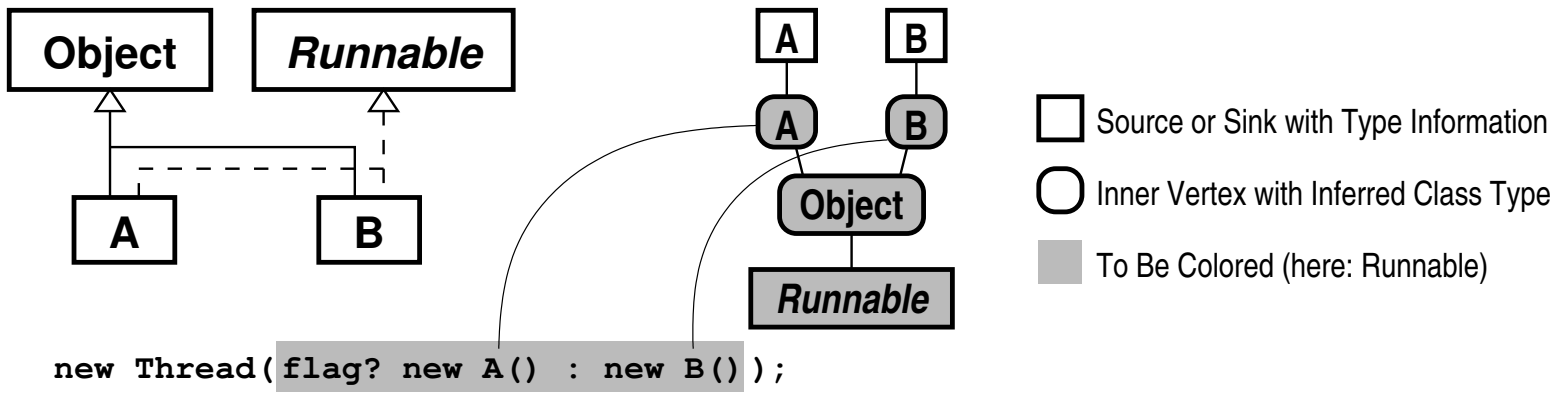
- arguments passed to a method
- values stored into an array

Edges reflect data movement.

Interest limited to connected components with interface type sinks!

SHAP: Interface Method Dispatch

Data Flow Graph – Example Component



SHAP: Interface Method Dispatch

Legacy Issue: Multitype

```
    ifxxx @0                          – Branch
    new A                             – Create Instance of A
    dup
    invokespecial A.<init>()V
    goto @1
@0:
    new B                             – Create Instance of B
    dup
    invokespecial B.<init>()V
@1:                                   – Merging Control Flows
    dup
    invokeinterface I1.f()V           – Invoke Interface Method of I1
    invokeinterface I2.g()V           – Invoke Interface Method of I2
```

SHAP: Interface Method Dispatch

Legacy Issue: Late Runtime Verification

```
    ifxxx @0                                  – Branch
    new A                                     – Create Instance of A
    dup
    invokespecial A.<init>()V
    goto @1
@0:
    new Object                                – Create Instance of Object
    dup
    invokespecial Object.<init>()V
@1:                                           – Merge Control Flows
    invokeinterface I1.f()V                   – Invoke Interface Method
```

SHAP: Interface Method Dispatch

Disarming Runtime Casts

Runtime casts to interface types require expensive `itable` search.
Fast interface method dispatch is rendered absurd if preceded by a cast.

Measures

- Never uncolor references.
- Simply check the color before performing a searching cast (fast!).
- Defensive coloring for pre-Tiger legacy code (after SableVM evaluation).

Result

References stored in generic data structures will already have the correct color if:

- having been passed as typed method argument such as through `addActionListener(ActionListener)`, or
- having been stored in an appropriately parametrized generic collection such as `ArrayList<ActionListener>`.

SHAP: Interface Method Dispatch

SHAP Integration

Embedded JVM with restricted runtime type information:

- CLDC 1.1 runtime implementation, and
- no support of reflection (yet?).

`itable`s integrated into VMT:

- `cb` uses statically resolved `itable` offset.
- Invocation through VMT with additional displacement provided by color.
- `itable` maintains fixed VMT position within all subclasses.
- `cc` corrects displacement by constant offset into current `itable`.

SHAP: Interface Method Dispatch

Discussion

- Coercion impact on code size highly dependable on application.
- Typically well below 0.1%.
- Optimizing compilers supplying tight `StackMapTable`s would be favorable for:
  - virtualization of interface method calls,
  - devirtualization of virtual method calls, and
  - abandoning defensive coloring.

Summary

SHAP platform implements a J2ME/CLDC platform with

- HW support for multiple concurrent threads,
- predictable code caching, and
- non-blocking concurrent garbage collection.

Method dispatch is assisted throughout the architecture and performed in constant-time for all call types.

Thank you!