Low-Level Haskell Code: Measurements and Optimization Techniques

PhD Proposal @ Rice University

David M Peixotto
0xd September 2011
Problem Statement

Modern science demands extensible, correct, and efficient software.

Haskell provides many flexible and safe abstractions for the programmer, but these abstractions can be detrimental to performance.

Runtime trace-based optimization can be used to overcome the overheads of Haskell's abstractions and improve the performance of Haskell programs.
Unique Research Contribution

• Comparison of low-level Haskell and traditional imperative codes
  submitted: Haskell `10

• Measurements of program traces in Haskell
  submitted: IFL `11

• Optimization of Haskell using low-level trace-based optimizations
  in progress: PLDI `12
Broad Motivation

• 84.3% state that developing software is important for their own research

• 30% of their time developing software

• 48.5% - strictly desktop development [Hannay et al., 2009]

• Software bugs have led to retracted publications [Merali, 2010]
Why Haskell is Great

Key Abstractions

• **First-class functions**: extensibility

• **Laziness**: correctness and extensibility

• **Type Classes**: extensibility

Bonus Features

• **Purity**: correctness and extensibility

• **Strong Static Typing**: correctness and efficiency

• **Parallelism**: efficiency
Why Haskell is (not so) Great

Key Abstractions

• **First-class functions:** (inefficient) extensibility

• **Laziness:** (inefficient) correctness and extensibility

• **Type Classes:** (inefficient) extensibility

• Also space usage, debugging, education, libraries
Context of My Research

• **High-level** optimizations
  • Very effective (584% improvement)

• **Low-level** optimizations
  • Little effect on performance (1.5% improvement)

• **New technique** needed for Low-level optimizations
Why Trace-Based Optimization

- Abstraction mechanisms obscure the control flow
- Static techniques cannot fully recover the control flow
- Runtime traces expose the actual control flow
Demonstrating Feasibility

• Select a simple Haskell program
• Build traces by hand
• Perform a feasible optimization
• Evaluate the results
Example Program (simplified)

\[
\text{root} = \text{sum} \ (\text{enumFromTo} \ 1 \ 300000000)
\]

\[
\text{enumFromTo} :: \text{Int} \rightarrow \text{Int} \rightarrow [\text{Int}]
\text{enumFromTo} \text{ from to} =
\begin{cases} 
\text{[]} & \text{if from} > \text{to} \\
\text{from : } \text{enumFromTo} \ (\text{from} + 1) \ \text{to} & \text{else}
\end{cases}
\]

\[
\text{sum} :: (\text{Num a}) \Rightarrow [\text{a}] \rightarrow \text{a}
\text{sum} \ l = \text{sum'} \ l \ 0
\]
\[
\text{where}
\begin{align*}
\text{sum'} \ [] & = \text{acc} \\
\text{sum'} \ (x:xs) & = \text{sum'} \ xs \ (\text{acc} + x)
\end{align*}
\]
Control Flow (no laziness)

enumFromTo

root

sum
Control Flow (simplified)

- **root**
- **enumFromTo**
- **Runtime**
- **sum**
- **Num**

- **thunk update**
- **thunk eval**
- **method lookup**
- **unknown function application**

- **direct control flow**
- **lazy control flow**
- **type-class control flow**
Control Flow (gory details)

enumFromTo

sMX CC: from

s0
CC: to

stg_upd_frame

sat_s00
thunk eval
add from + 1

sum

Num. fromInteger

stg_ap_p

sNj
CC: fromInt

sMI
eval: list

sNi
return point

ap_0_fast
eval accum

sNh
cons check

sMG
eval accum + x

stg_ap_pp
apply plusInt

enumFromTo

sum

class

runtime

enumFromTo trace

first time execution

optimized trace

direct jump (solid arrow head)

indirect jump (open arrow head)

Tuesday, September 13, 11
## Hand-Coded Trace Results

<table>
<thead>
<tr>
<th>Trace Version</th>
<th>Median Time</th>
<th>% Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>None (baseline)</td>
<td>6.83 seconds</td>
<td>--</td>
</tr>
<tr>
<td>EnumFromTo</td>
<td>6.50 seconds</td>
<td>5%</td>
</tr>
<tr>
<td>Sum</td>
<td>6.34 seconds</td>
<td>8%</td>
</tr>
<tr>
<td>Linked</td>
<td>5.89 seconds</td>
<td>16%</td>
</tr>
<tr>
<td>Optimized</td>
<td>4.92 seconds</td>
<td>39%</td>
</tr>
</tbody>
</table>
Beyond Hand-Coded Traces

• **DynamoRIO** is a binary trace-based optimization framework

• Arbitrary code modification

• Custom trace building

How well does it work on Haskell codes?
Overview of DynamoRIO

Figure 2.1: Flow chart of DynamoRIO. A context switch separates the code cache from DynamoRIO (though it all executes in the same process and address space). Application code is copied into the two caches, with control transfers (shown by arrows in the figure) modified in order to retain control.

System Components

<table>
<thead>
<tr>
<th>Component</th>
<th>Performance Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic block cache</td>
<td>∼300x</td>
</tr>
<tr>
<td>SPECFP</td>
<td>∼300</td>
</tr>
<tr>
<td>SPECINT</td>
<td>17.16x</td>
</tr>
<tr>
<td>Emulation</td>
<td>3.04x</td>
</tr>
<tr>
<td>Link direct branches</td>
<td>1.32x</td>
</tr>
<tr>
<td>Link indirect branches</td>
<td>1.05x</td>
</tr>
<tr>
<td>Traces</td>
<td>1.02x</td>
</tr>
<tr>
<td>Optimizations</td>
<td>0.88x</td>
</tr>
</tbody>
</table>

Table 2.2: Performance summary of the fundamental components of DynamoRIO described in this chapter: a basic block cache, linking of direct and indirect branches, and building traces. Average numbers for both the floating-point (SPECFP) and integer (SPECINT) benchmarks from the SPEC CPU2000 suite are given (our benchmarks are described in Section 7.1). We overcame numerous architectural challenges (Chapter 4) to bring each component to the performance level listed here. The final entry in the table shows the best performance we have achieved with DynamoRIO, using aggressive optimizations to surpass native performance for some benchmarks (see Section 9.2).
Running Haskell With DynamoRIO is Expensive

Fibon S/F Ratio (−O2 vs. −O2 + DynamoRIO)

-14% SPEC

70% slowdown
Indirect Branch Lookup is a Large Source of Overhead

Fibon – DynamoRIO PC Profile Data

- Xsact
- TernaryTrees
- SpectralNorm
- Simgi
- Regex
- Qc
- Pidigits
- Pappy
- Palindromes
- Nbody
- Mandelbrot
- Hgalib
- Happy
- HaLex
- Gf
- Funsat
- Fst
- Fgl
- Fannkuch
- Crypto
- Bzlib
- BinaryTrees
- Agum

- Interp
- Dispatch
- Monitor
- Syscall_Handler
- IBL
- Off_Trace
- On_Trace
- Unknown

Tuesday, September 13, 11
Future Work

Building Traces

• How do we select good traces for Haskell codes?

• How do we reduce the overheads of building traces?

Optimizing Traces

• What optimizations can be done?

• What optimizations are profitable?

• What optimizations are specifically important for Haskell codes?
Selecting Haskell Traces

• What is the best way to mark trace heads for Haskell?
  • Backward branches mark trace heads [DynamoRIO, HotpathVM, SPUR]
  • Functions mark trace heads [Lambdachine, SPUR]

• What about trace specialization points?
Trace Specialization Points

• Type-class dictionaries (e.g. Num.+)
• Thunk info table (e.g. enumFromTo)
• Internal Control Flow (e.g. cons check)
• Tail recursion (e.g. sum loop)

Trace trees to combat over-specialization [Gal]
Optimizing Haskell Traces

• Code motion
• Value numbering
• Load elimination
• Register allocation
• Combining Heap Checks
Combining Heap Checks

• Every allocation must include a heap overflow check

• Multiple allocations on the same trace can share a single heap check

• Reduces the number of comparisons and branches on the trace
Related Work

• **Haskell Optimizers**
  
  Peyton Jones and Santos [1998] (Transformations)
  Boquist [1999] (GRIN)
  Wakeling [1999] (X-Machine)
  Schilling [2011] (Lambdachine)

• **Trace-Based Optimizers**
  
  Bala et al. [2000] (Dynamo)
  Bruening [2004] (DynamoRIO)
  Gal [2006] (Java HotpathVM)
  Bebenita et al. [2010b] (SPUR JS CIL)
<table>
<thead>
<tr>
<th>Timeline Event</th>
<th>Date/Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Propose</td>
<td>September 2011</td>
</tr>
<tr>
<td>Building Haskell Traces</td>
<td></td>
</tr>
<tr>
<td>Low-Level Runtime Optimizations</td>
<td></td>
</tr>
<tr>
<td>Thesis Writing</td>
<td></td>
</tr>
<tr>
<td>Defend</td>
<td>March 2012</td>
</tr>
<tr>
<td>Thesis Submission Deadline</td>
<td>April 20, 2012</td>
</tr>
</tbody>
</table>
References


In Defense of Laziness

• A new kind of glue (John Huges)
• A "hair shirt" (Simon Peyton Jones)
• Convenience (Simon Marlow)
  \[ f \ x \ y = \text{if } x > y \text{ then } a \text{ else } b \]
  where
  \[ a = \text{expensive} \]
  \[ b = \text{expensive} \]
• Function Reuse (Lennart Augustss)
  \[ \text{any :: (} a \rightarrow \text{Bool)} \rightarrow [a] \rightarrow \text{Bool} \]
  \[ \text{any } p = \text{or \ . \ map } p \]
## Binary Optimization Past Results

<table>
<thead>
<tr>
<th>System</th>
<th>Arch</th>
<th>Suite</th>
<th>Baseline</th>
<th>Worst</th>
<th>Best</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamo</td>
<td>PA-RISC</td>
<td>SPECInt 95</td>
<td>HP-O2</td>
<td>-3%</td>
<td>22%</td>
<td>9%</td>
</tr>
<tr>
<td>Dynamo</td>
<td>x86</td>
<td>SPECInt 2000</td>
<td>GCC -O3</td>
<td>-50%</td>
<td>2%</td>
<td>-12%</td>
</tr>
<tr>
<td>Dynamo</td>
<td>x86</td>
<td>SPECFp 2000</td>
<td>GCC -O3</td>
<td>-1%</td>
<td>45%</td>
<td>12%</td>
</tr>
<tr>
<td>Adore</td>
<td>Itanium</td>
<td>SPEC 2000</td>
<td>ORC -O2</td>
<td>-2%</td>
<td>57%</td>
<td>8%</td>
</tr>
<tr>
<td>Adore</td>
<td>Itanium</td>
<td>BLAST</td>
<td>ORC -O2</td>
<td>-3.5%</td>
<td>58%</td>
<td>20%</td>
</tr>
</tbody>
</table>
High-level optimizations work well

584% average improvement
Low-level optimizations have little effect

1.5% average improvement
```haskell
sum :: (Num a) => [a] -> a
sum l = sum' l 0

where
  sum' [] acc = acc
  sum' (x:xs) acc = sum' xs (acc+x)

-- call sites
sum (enumFromTo 1 1000)    -- original
sum (buildList 1 1000)     -- different thunk
sum (buildList 1.0 1000.0) -- different dict
```
Trace Example

Haskell Source Code

\[ \text{sum}' \ [\] \Rightarrow \text{acc} = \text{acc} \]
\[ \text{sum}' \ (x:xs) \Rightarrow \text{acc} = \text{sum}' \ xs \ (\text{acc}+x) \]

STG Code

\[
\text{sum'} \ l \ a = \\
\text{case } l \text{ of} \\
[\] \Rightarrow a \\
x \ xs \Rightarrow \text{case } a \text{ of} \\
al \Rightarrow \text{let } t = \text{Num.}+ \ d\text{Num} \ a \ x \\
in \ \text{sum'} \ xs \ t
\]
// r0 = l, r1 = acc
sum':
  r2 = eval r0
  case r2[0]
      Nil:
        r3 = eval r1
        ret r3
    Cons:
        r4 = eval r1
        r5 = THUNK (Num.+, $dNum, r4, r2[1])
        call sum' r2[2] r5
// r0 = l, r1 = acc

sum':

  guard(r0[0] == \texttt{enumFromTo})
  \ldots\text{ code for }\texttt{enumFromTo} \ldots
  \sum' iterations

  // r2 = eval r0
  guard(r2[0] == \texttt{Cons})
  guard(r1[0] == \texttt{Num.\,+})
  \ldots\text{ code for }\texttt{Num.\,+} \ldots

  // r4 = eval r1
  r0 = r2[2]
  r1 = \texttt{THUNK}(..., r4, r2[1])
  goto \text{sum}'

\textbf{Redundant on subsequent sum' iterations}
Unrolling a Trace

sum':
  guard(r0[0] == enumFromTo)
  ... code for enumFromTo ...
  guard(r2[0] == Cons)
  guard(r1[0] == Num.+)
  ... code for Num.+ ...
  ... code for sum' ...
LOOP:
  ... code for enumFromTo ...
  guard(from <= to)
  ... code for Num.+ ...
  ... code for sum' ...
goto LOOP
Example Program (detailed view)

\[
\begin{align*}
\text{root} & = \text{sum} \ (\text{enumFromTo} \ 1 \ 300000000) \\
\text{enumFromTo} & :: \text{Int} \rightarrow \text{Int} \rightarrow [\text{Int}] \\
\text{enumFromTo} \ from@(I# m) \ to@(I# n) & = \\
& \quad \text{if } m \geq n \ \text{then } [] \ \text{else} \\
& \quad \text{from} : \text{enumFromTo} \ (I# (m +# 1#)) \ \text{to} \\
\text{sum} & :: (\text{Num} \ a) \Rightarrow [a] \rightarrow a \\
\text{sum} \ l & = \text{sum'} \ l \ 0 \\
\text{where} \\
& \quad \text{sum'} \ [] \ \!acc = \ acc \\
& \quad \text{sum'} \ (x:xs) \ \!acc = \ \text{sum'} \ xs \ (acc+x)
\end{align*}
\]
My Proposed Approach

Traditional Path

Programmer → Haskell Program

Haskell Compiler

machine code

1011010
1010101
0110101
0101010

Static Optimizer

1011010
1010101
0110101
0101010

Bytecode

Compile Time

Runtime Optimizer

1011010
1010101
0110101
0101011
0111010

Runtime

CPU

Something New

1011010
1010101
0010101
0101101
0101101
0111010

Traditional Path

1011010
1010101
0110101
0101010

Something New

1011010
1010101
0010101
0101101
0101101
0111010

Tuesday, September 13, 11
Figure 1.3: The DynamoRIO runtime code manipulation layer. DynamoRIO intercepts itself between an application and the underlying operating system and hardware. It executes a copy of the application’s code out of a code cache to avoid emulation overhead. Key challenges include managing multiple threads, intercepting direct transfers of control from the kernel, monitoring code modification to maintain cache consistency, and bounding the size of the code cache. The cached code can then be executed natively, avoiding emulation overhead. However, shifting execution into a cache that occupies the application’s own address space complicates transparency. One of our most significant lessons is that DynamoRIO cannot run large, complex, modern applications unless it is fully transparent: it must take every precaution to avoid affecting the behavior of the program it is executing.

To reach the widest possible set of applications (to be universal and practical), DynamoRIO targets the most common architecture, IA-32 (a.k.a. x86), and the most popular operating systems on that architecture, Windows and Linux. The efficiency of a runtime code manipulation system depends on the characteristics of the underlying hardware, and the Complex Instruction Set Computer (CISC) design of IA-32 requires a significant effort to achieve efficiency. To be universal, DynamoRIO must handle dynamically-loaded, generated, and even modified code. Unfortunately, since any store to memory could legitimately modify code on IA-32, maintaining cache consistency becomes challenging.
Fibonacci Weighted Average Trace Length
SPEC Weighted Average Trace Length

Tuesday, September 13, 11
Fibonacci Number of Traces Needed to Contain $\geq 50\%$ of Samples

- Agum
- BinaryTrees
- Bzlib
- Crypto
- Fannkuch
- Fgl
- Fst
- Funsat
- Gf
- HaLex
- Happy
- Hgalib
- Mandelbrot
- Nbody
- Palindromes
- Pappy
- Pidigits
- Qc
- Regex
- Simgi
- SpectralNorm
- TernaryTrees
- Xsact
- Median

Number of Traces

- 0
- 10
- 20
- 30
- 40
- 50

12.00

Tuesday, September 13, 11
Fibon Most Frequently Executed Trace

Percent of Total Samples

Agum  BinaryTrees  Bzlib  Crypto  Fannkuch  Fgl  Fst  Funsat  Gf  HaLex  Happy  Hgalib  Mandelbrot  Nbody  Palindromes  Pappy  Pidigits  Qc  Regex  Simgi  SpectralNorm  TernaryTrees  Xsact

Mean – 95% CI

0.0  0.2  0.4  0.6  0.8  1.0  1.2

0.0  0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9  1.0

Tuesday, September 13, 11
SPEC Number of Traces Needed to Contain $\geq 50\%$ of Samples

Number of Traces

Tuesday, September 13, 11
SPEC Most Frequently Executed Trace
Research Applications

• Haskell codes
• Other managed languages
• Lazy control flow in other languages
  • Python generators
• Scala sequence comprehensions
Example Improvement

Current DynamoRIO
Trace A

A
B
C
Branch Check (D)
D

Trace E

E
F
G

Check Fails

Proposed Optimization
Trace A

A
B
C
Branch Check (D)
D

IBL

Check Fails

Branch Check (E)

Trace E

E
F
G
Related Work

• Static Haskell optimizations
  • Deforestation, Inlining, Pointer tagging

• Dynamic optimization in virtual machines
  • Smalltalk, Self, Java, JavaScript

• Dynamic binary optimization
  • Dynamo, DynamoRIO, Adore
Related Work

• Haskell Optimizers
  Peyton Jones and Santos [1998] (Transformations)
  Boquist [1999] (GRIN)
  Schilling [2011] (Lambdachine)

• Trace-Based Optimizers
  Bala et al. [2000] (Dynamo)
  Bruening [2004] (DynamoRIO)
  Gal [2006] (Java HotpathVM)
  Bebenita et al. [2010b] (SPUR JS CIL)
Related Work

• **Static Optimization**  
  Peyton Jones and Santos [1998]  
  Boquist [1999] (Transformations) (GRIN)

• **Trace-based VM Dynamic Optimization**  
  Gal [2006]  
  Bebenita et al. [2010b] (Java HotpathVM) (SPUR JS CIL)

• **Binary Dynamic Optimization**  
  Bala et al. [2000]  
  Bruening [2004] (Dynamo) (DynamoRIO)
Haskell Related Work

• **STG**
  Peyton Jones [1992]  (Execution model)

• **High-Level Optimization**
  Gill et al. [1993]
  Peyton Jones and Santos [1998]  (Deforestation)
  (Transformations)

• **Low-Level Optimization**
  Marlow and Jones [2007]  (Pointer tagging)
  Terei and Chakravarty [2010]  (LLVM backend)
  Boquist [1999]  (GRIN)
Virtual-Machine Related Work

• **Dynamic Languages**
  Deutsch and Schiffman [1984] (Smalltalk)
  Hölzle et al. [1991] (Self)

• **Method-Based VMs**
  Wakeling [1998] (Haskell)
  Burke et al. [1999] (Java)

• **Trace-Based VMs**
  Gal [2006] (Java)
  Gal et al. [2009] (JavaScript)
  Bebenita et al. [2010b] (CIL)
Binary Optimization Related Work

• **Dynamo**
  - Bala et al. [2000] (Overview)
  - Duesterwald and Bala [2000] (NET traces)

• **DynamoRIO**
  - Bruening et al. [2003] (Overview)
  - Bruening [2004] (Thesis)

• **Adore**
  - Lu et al. [2004] (Prefetching)
  - Das et al. [2005] (BLAST optimization)