Leveraging Managed Runtime Systems to Build, Analyze, & Optimize Memory Graphs

Rebecca Smith and Scott Rixner
Motivation

Optimizing memory management is a major challenge of embedded systems programming

- **Scarcity**
  - 32 - 256KB of SRAM
  - 128KB - 1MB of on-chip flash

- **Heterogeneity**
  - Varied proportions of SRAM and flash
Approach

Build, analyze, and optimize memory graphs

- **Prerequisites:** structured memory
  - All objects must be instances of well-defined types
  - All objects must include type and size identifiers

- Managed runtime systems impose the necessary structure
GEM: Graphs of Embedded Memory

A versatile framework for building and transforming graphs of embedded memory

- **Contributions:**
  1. Graph-based approach to memory transformation
  2. Versatile framework implementing this approach
  3. Four novel low-level graph transformations
  4. Four representative high-level use cases
Context

- **Owl**: An embedded Python run-time system and development toolchain
  - Satisfies the necessary prerequisites
  - Standard CPython types + additional “packed” types

<table>
<thead>
<tr>
<th>type: PTP</th>
<th>size: 80</th>
<th>length: 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>type: INT</td>
<td>size: 12</td>
<td>value: 1</td>
</tr>
<tr>
<td>type: INT</td>
<td>size: 12</td>
<td>value: 2</td>
</tr>
<tr>
<td>type: INT</td>
<td>size: 12</td>
<td>value: 3</td>
</tr>
</tbody>
</table>

- Available at [embeddedpython.org](http://embeddedpython.org)
Related Work

- **Memory graphs**
  - Program visualization [e.g. Zimmerman02]

- **Transformative use cases**
  - De-duplication
    - File / block de-duplication [e.g. Kulkarni04]
    - Runtime interning [e.g. Kawachiya08]
  - Compilation with size constraints [e.g. Naik02]
GEM: Graphs of Embedded Memory
Tool Organization

Owl Runtime System

GEM Toolchain

- Dump
- Memory Transfer
- Memset
- Parse
- Memory Conversion
- Unparse
- Graph Transformation

Toolchain Structure
Graph Transforms
Use Cases
Results
Memory Transfer

● **Goal:** Transfer contents of memory to and from the microcontroller in raw byte format

● **dump from microcontroller:**
  1. Dump heap metadata
  2. Dump heap contents
  3. Dump flash contents (reachable from heap)

● **memset to microcontroller:**
  1. Dump memory of destination microcontroller
  2. Splice source graph into destination graph
  3. Overwrite heap on microcontroller
Memory Conversion

- **Goal**: Convert between raw byte format and graph format

- Auto-generated during RTS compilation
  - **Input**: Header files containing type definitions
  - **Output**: Python file containing parse and unparse functions for each type
Graph Transformation

- **Goal**: Transform memory graph

- Low-level transformation passes can be combined to implement high-level use cases

- Seven passes are currently available
  - De-duplication
  - Unpacking
  - Splicing
  - Splitting
  - ...

De-duplication

Before:
- type: TUP
  - size: 44
  - length: 3
  - items: ( , , , )
  - type: INT
    - size: 12
    - value: 3

- type: TUP
  - size: 44
  - length: 3
  - items: ( , , , )
  - type: INT
    - size: 12
    - value: 1

After:
- type: TUP
  - size: 44
  - length: 3
  - items: ( , , , )
  - type: INT
    - size: 12
    - value: 3

- type: TUP
  - size: 44
  - length: 3
  - items: ( , , , )
  - type: INT
    - size: 12
    - value: 1
Unpacking

**Before:**
- type: PTP
  - size: 80
  - length: 3
  - value: 1
- type: INT
  - size: 12
  - value: 2
- type: INT
  - size: 12
  - value: 3

**After:**
- type: TUP
  - size: 44
  - length: 3
  - items: (•, •, •)
- type: INT
  - size: 12
  - value: 1
- type: INT
  - size: 12
  - value: 2
- type: INT
  - size: 12
  - value: 3
Splicing

Before:

Source
SRAM1
a
b
c

Dest
SRAM2
i
h
j

Flash1
d
e
f

Flash2
d
k
b
Splicing

Before:

Source
SRAM1
a
b
c
Flash1
d
e
f

Dest
SRAM2
i
h
j
Flash2
d
k
b
Splicing

Before:

Source

SRAM1

a

b

c

Flash1
d
e
f

Dest

SRAM2

i

h

j

Flash2
d

k

b
Splicing

Before:

Source
SRAM1
a
b
c
Flash1
d
e
f

Dest
SRAM2
i
h
j
Flash2
d
k
b
Splicing

Before:

Source
SRAM1
- a
  - b
  - c
Flash1
  - d
  - e
  - f

Dest
SRAM2
- i
  - h
  - j
Flash2
  - d
  - k
  - b
Splicing

Before:

Source
SRAM1
- a
  - b
  - c

Flash1
- d
- e
- f

Dest
SRAM2
- i
  - h
  - j

Flash2
- d
- k
- b

After:

(SRAM1 U Flash1) \ Flash2
- e
- a
- f
- c

Flash2
- d
- k
- b
Splitting

Before:

After:
Use Cases

● GEM’s transformations are building blocks
  ○ Can be combined to address a variety of use cases

● Four use cases have been implemented:
  1. Interactive visualization
  2. De-duplication of Python library code
  3. Heterogeneous compilation
  4. Transparent migration
Library De-duplication

● **Motivation:** Memory is scarce
  ○ The Python compiler generates many duplicate objects; eliminating them saves precious space

● **Approach:** Consolidate immutable duplicate objects at compile-time
Workflow

Compile-time:
1. Begin compilation
2. Construct graph
3. Unpack
4. De-duplicate
5. Compact
6. Unparse
7. Finish compilation
8. Shift references
9. Unparse
Workflow

Compile-time:
1. Begin compilation
2. Construct graph
3. Unpack
4. De-duplicate
5. Compact
6. Unparse
7. Finish compilation
8. Shift references
9. Unparse

Runtime:
N/A
Heterogeneous Compilation

- **Motivation:** Memory is scarce; memory architectures are heterogeneous
  - Catering to the lowest common denominator of each region of memory is wasteful

- **Approach:** Distribute code across different regions of memory, based on availability
  - Specify upper bounds on each memory region at compile-time
Workflow

Compile-time:
1. Begin compilation
2. Construct graph
3. Unpack
4. De-duplicate \((optional)\)
5. Split
6. Unparse
7. Finish compilation
8. Shift references
9. Unparse
Workflow

Compile-time:
1. Begin compilation
2. Construct graph
3. Unpack
4. De-duplicate (optional)
5. Split
6. Unparse
7. Finish compilation
8. Shift references
9. Unparse

Runtime:
1. Boot
2. Dump
3. Construct graph
4. Splice
5. Unparse
6. Memset
Partitioning Algorithm

● **Constraints:**
   1. Adhere to upper bounds on each memory region
   2. Place modules needed to boot in flash
   3. Ensure that nothing in flash references SRAM

● **Algorithm:**
   ○ Begin with all modules needed to boot in flash; all else in SRAM
   ○ Greedily promote components $C$ from SRAM to flash

   $C = \{o\} \cup \{o' \mid ((o \text{ references } o') \land (o' \in \text{SRAM}))\}
Partitioning Example

Inputs:

SRAM (All Objects)

- Flash cap = 20 KB
- SRAM cap = 64 KB

- a = 8 KB
- b = 2 KB
- c = 6 KB
- d = 6 KB
- e = 6 KB
- f = 8 KB
- g = 2 KB
Partitioning Example

Inputs:
SRAM (All Objects)

Flash cap = 20 KB
SRAM cap = 64 KB

Candidate Components:

= 22 KB
= 22 KB
= 18 KB
Partitioning Example

Inputs:

SRAM (All Objects)

- a → d → f
- b → c → e → g

Flash cap = 20 KB
SRAM cap = 64 KB
- a = 8 KB
- b = 2 KB
- c = 6 KB
- d = 6 KB
- e = 6 KB
- f = 8 KB
- g = 2 KB

Output:

SRAM

- a → f
- b

Flash

- d
- c → e → g
Evaluation

Evaluated each use case on three platforms

1. Desktop machine
   ○ Not resource-constrained

2. Stellaris LM3S9B92
   ○ 96 KB SRAM
   ○ 256KB flash

3. STM32F4-Discovery
   ○ 192KB SRAM
   ○ 1 MB flash
De-duplication

De-duplication of Python Libraries

![Graph showing de-duplication results for different platforms.](image)

- **Desktop**
  - Unpacked
  - Packed
  - Unpacked, de-duplicated
  - Hybrid

- **Stellaris**
  - Unpacked
  - Packed
  - Unpacked, de-duplicated
  - Hybrid

- **STM32**
  - Unpacked
  - Packed
  - Unpacked, de-duplicated
  - Hybrid
Heterogeneous Compilation
Conclusions

● Optimizing memory is hard!
  ○ Scarcity, heterogeneity

● Managed runtime systems provide new opportunities to tackle memory challenges
  ○ Facilitate construction of memory graphs

● GEM leverages memory graphs to provide a modular means of solving varied use cases