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

Rebecca Smith
Rice University
rjs@rice.edu

Scott Rixner
Rice University
rixner@rice.edu

Abstract

Optimizing memory management is a major challenge of embedded systems programming, as memory is scarce. Further, embedded systems often have heterogeneous memory architectures, complicating the task of memory allocation during both compilation and migration. However, new opportunities for addressing these challenges have been created by the recent emergence of managed runtimes for embedded systems. By imposing structure on memory, these systems have opened the doors for new techniques for analyzing and optimizing memory usage within embedded systems. This paper presents GEM (Graphs of Embedded Memory), a tool which capitalizes on the structure that managed runtime systems provide in order to build memory graphs which facilitate memory analysis and optimization. At GEM's core are a set of fundamental graph transformations which can be layered to support a wide range of use cases, including interactive memory visualization, de-duplication of objects and code, compilation for heterogeneous memory architectures, and transparent migration. Moreover, since the same underlying infrastructure supports all of these orthogonal functionalities, they can easily be applied together to complement each other.

1. Introduction

Managed runtime systems, already common in traditional computing systems, are increasingly being adopted by embedded systems programmers to increase productivity. Modern embedded runtime systems such as eLua [2], p14p [5], Micro Python [7], and Owl [4] raise the abstraction level of embedded systems programming by providing interpreters for high-level languages and automating tasks such as thread scheduling, garbage collection, and inter-process communication. While the adoption of runtime systems has primarily been motivated by aspirations of productivity, these systems also provide new opportunities for improving memory analysis and optimization by imposing structure on memory.

Embedded systems exhibit a combination of resource constraints and heterogeneity which makes it difficult, yet essential, to carefully analyze and optimize memory. In terms of resource constraints, a typical mid-range microcontroller may have only 32–256 KB of SRAM and 128 KB–1 MB of flash, necessitating thoughtful management of memory at both compile-time and runtime.

The task of memory management is further complicated by heterogeneity; a single embedded system may consist of multiple microcontrollers with varied proportions of SRAM and flash. As an example, one member of STM32's Cortex-M series of microcontrollers has 256 KB of flash and 32 KB of SRAM [39], while another has only 128 KB of flash, but 64 KB SRAM [38]. In a heterogeneous system, allocation and data layout schemes could simply cater to the lowest common denominator of each type of memory. However, this is impractical due to the scarcity of memory; a more flexible solution is needed.

Outside the domain of embedded systems, tools have been developed to leverage the structure imposed by managed runtime systems to model memory as a graph [8, 16, 26, 30, 32, 43]. By providing a clear and intuitive representation of the relationships between objects, graphs facilitate visualization, analysis, and transformations across multiple nodes. This is hugely valuable to memory transformation, as inter-object references make it nearly impossible to operate on a single object in isolation. Yet, existing tools use memory graphs solely for visualization and analysis, transforming only insofar as is necessary to display the graph.

The challenges faced by embedded systems motivate the development of tools which use memory graphs not only for analysis, but also for optimization. However, embedded applications have traditionally been written in C, where the unstructured memory layout and presence of unidentifiable pointers inhibits the construction of memory graphs. In contrast, runtime systems for high-level languages organize

VEE '16, April 2–3, 2016, Atlanta, GA, USA

DOI: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2892242.2892253

memory in a way that is amenable to graph-based analysis and transformation.

This paper introduces GEM, a memory configuration tool for embedded runtime systems which builds, analyzes, and transforms graphs of the entire memory space of a program. At its core is a set of fundamental transformation passes that can be combined into high-level use cases. Additionally, GEM includes a mechanism for installing a transformed graph in memory, allowing these abstract graph transformations to impact the actual memory layout of the system. Due to its flexible and extensible infrastructure, GEM has proven valuable in a variety of scenarios. In particular, this paper presents four key contributions:

- A versatile framework for combining low-level graph transformations to achieve high-level use cases.
- A mechanism for installing graphs in memory.
- Four novel low-level transformations: splicing, splitting, unpacking, and generic fine-grained de-duplication.
- Four high-level use cases: interactive visualization, object and code de-duplication, compilation for heterogeneous memory architectures, and transparent migration.

Evaluation of the four representative use cases showed that GEM is instrumental in improving memory analysis and optimization. For instance, its visualizer uncovered systemic inefficiencies, and its de-duplication capability reclaimed up to 24% of the flash consumed by language-level libraries.

The next section describes the context in which GEM operates. Section 3 presents GEM's infrastructure and transformations. Section 4 explains the workflow for each high-level use case, and Section 5 evaluates these use cases. Section 6 discusses related work, and Section 7 concludes the paper.

2. Context

Memory optimization tools are particularly valuable at the intersection of embedded systems and managed runtime systems. Embedded systems have much to gain from memory optimization; runtime systems establish the order within the memory space needed to achieve these gains.

GEM requires the memory organization to satisfy two properties. First, all data must be stored in objects of well-defined types known to the runtime system. Second, all objects must have explicit or implicit type and size identifiers. The fundamental concepts behind GEM are generalizable, and could be implemented for any runtime system which meets these requirements, such as the Oracle JVM or CPython. Because embedded systems face particularly stringent memory constraints, a prototype was built for Owl [4, 9], an embedded Python runtime system representative of managed runtime systems for embedded microcontrollers.

Like CPython, Owl includes both a compiler and an interpreter. Python source code is first compiled into code objects, which contain bytecodes. These bytecodes are then executed by the interpreter at runtime. Owl stores the entire Python program, including the code objects, as Python objects. Each object includes a 4 B object descriptor specifying its type and size, and has fields containing data and references to other objects.

These Python objects are distributed across SRAM and flash. All runtime state, including threads and stack frames, is allocated on the Python heap which resides in SRAM. At compile-time, modules of Python library code are stored in flash. Owl first compiles the modules using the standard CPython compiler, generating one Python code object for each module. Each code object includes the code itself — a series of bytecodes, in the form of a string of bytes — as well as metadata such as the filename and variable names. Next, the Owl toolchain converts the compiled modules into a "packed" format, using two special types: packed code objects and packed tuples. Packed objects do not reference other objects, instead storing constituent objects internally as a contiguous array. This saves space by eliminating references; further, this eliminates the need for a dynamic linker, as each module is completely self-contained.

Finally, at boot-time, Owl allocates a global object on the heap to hold the runtime state, including "module paths" consisting of the addresses of the code objects for each module. The module paths serve two purposes: they are used to import modules at runtime, and they ensure that all of the Python objects in flash are reachable from the heap.

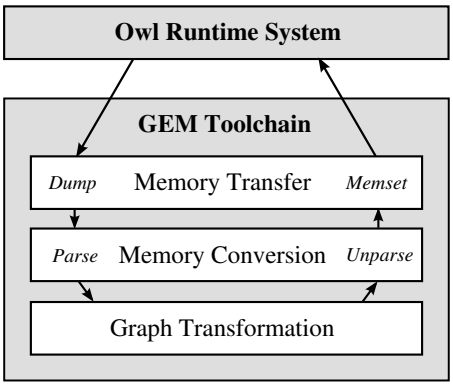
Because Owl stores Python objects in both SRAM and flash, GEM's memory graphs have two components, one for each memory region. These graphs could easily be expanded to include components for additional memory regions, to support systems with more complex memory hierarchies.

3. Tool Organization

Due to resource constraints, GEM operates offline, so that it can build and manipulate graphs without using any of the microcontroller's precious memory space. This requires a "host" machine with sufficient resources to run GEM. A single host can serve a system of multiple microcontrollers, and could be connected via either a serial port or a wireless network. Further, this host need not be dedicated to GEM; for instance, the host could also be used to perform other deployment and management tasks required by the system.

GEM is composed of three layers. First is the memory transfer layer, which transports memory contents to and from the microcontroller in a raw byte format using two primitives, `dump` and `memset`. Next is the memory conversion layer, which uses an auto-generated parser and unparser to convert between two representations of memory: the raw byte format from the previous layer and a graph format used by the final layer. Last is the graph transformation layer, which consists of transformation passes that operate on a

Figure 1. GEM architecture



memory graph. Figure 1 depicts the GEM toolchain architecture.

Across these three layers, GEM uses two representations of memory. First, a `PyMem`, which is an intermediate representation of the microcontroller's entire memory space. This simple container is used by the parser to aggregate objects prior to building the graph. It includes a mapping for each region of memory — in Owl, SRAM (heap) and flash (library code objects) — which associate memory addresses with `PyObjects`. Each `PyObject` o has a type, a size, and two name-to-value mappings of its fields: `data`(o): `data_names`(o)$\mapsto$`data_vals`(o), for primitive types, and `pyptrs`(o): `pyptrs_names`(o)$\mapsto$`pyptrs_vals`(o), for pointers to other Python objects.

Frame objects include a third mapping: `cptrs`, which includes pointers that are not base addresses of Python objects. These are not arbitrary pointers, but well-defined pointers into other Python objects. In particular, there are three: the instruction pointer, the stack pointer, and a pointer to the last stack slot. The first of these points into a separate bytecode object; the others point within this same frame object.

Ultimately, GEM's graph transformation layer operates on a `MemGraph`. This is a directed graph of memory which, like the `PyMem`, encompasses multiple regions of memory. Each node corresponds to a `PyObject`, and edges run from a given `PyObject` to the nodes of each of the objects in its `pyptrs_vals` set. Formally, a `MemGraph` is a graph $G = (V, E)$ such that $V(G) = \{u \mid u \text{ is a } \texttt{PyObject}\}$ and $E(G) = \{(u, v \mid address(v) \in \texttt{pyptrs\_vals}(u)\}$.

3.1 Memory Transfer Layer

GEM provides two commands, `dump` and `memset`, which transfer a byte string representation of the memory space between the microcontroller and the host machine. These operations allow the memory transformations made by GEM to affect the actual layout of memory on the microcontroller.

The first of these, `dump`, consists of three stages. In each stage, GEM compiles and sends over the connection a function call which, upon execution by the interpreter on the microcontroller, causes the microcontroller to send heap data back to GEM. GEM first queries for metadata including the base address of the heap. It then requests the heap contents, beginning at this base address. Last, GEM fetches all of the Python code objects stored in flash.

The reverse of `dump` is `memset`, which overwrites the heap on the microcontroller. The heap being `memset` need not have originated from the destination device. However, if the source and destination are different microcontrollers, the base addresses of the heaps and the contents of flash may also be different. Therefore, `memset` does not blindly replace the heap. Instead, GEM first dumps the memory of the destination device, and then splices the SRAM (heap) component from the source into the flash component from the destination using a technique that will be described in greater detail in Section 3.3.5. Finally, GEM prepares to place the heap on the destination device by shifting all intra-heap references by the difference between the source and destination base addresses, using a reference-updating technique that will be described in Section 3.3.1.

After constructing the spliced graph, GEM unparses this graph and initiates an overwrite by compiling and sending a custom MEMSET bytecode to the destination device. Unlike `dump`, `memset` cannot be implemented as a function call, since overwriting the heap obliterates the call stack. GEM then sends the heap contents and metadata over the connection in a byte format. In executing the MEMSET bytecode, the microcontroller reads the heap contents and metadata, overwriting both. Once MEMSET completes, the program automatically resumes execution from the exact point at which its memory was dumped.

3.2 Memory Conversion Layer

While graphs are highly amenable to analysis and transformation, they are not compact enough to serve as the memory format on the microcontroller. Instead, objects in memory are laid out as a contiguous array of bytes. GEM provides a parser and unparser to convert between the byte format and the `MemGraph` format. These include top-level `parse` and `unparse` functions which take in a byte string or graph, respectively, as well as `parse_<type>` and `unparse_<type>` functions for each Python object type and for free blocks.

The top-level `parse` function first builds an intermediate `PyMem` representation by reading the byte string and processing one object at a time. It interprets the first four bytes of each object as an object descriptor, extracts the size and type, and dispatches a call to the appropriate type-specific function, which creates a `PyObject` or free block. It then maps the object's address to the newly-created `PyObject`. Once the entire string has been processed, the parser builds the `MemGraph` by iterating over the `PyObjects`, inserting them as nodes, and adding edges for each of their `pyptrs`.

Conversely, `unparse` receives a `MemGraph` and outputs a byte string. It first constructs a sorted list of the addresses of all heap nodes in the graph. None of GEM's transformations introduce gaps in the address space, so these addresses will

be contiguous assuming that GEM was given a valid image when it initially built the graph. It then unparses each object in order by invoking the appropriate `unparse_<type>` function to obtain a byte string representation of that object. Last, it concatenates the byte strings for the individual objects to form a complete representation of the heap.

Moreover, GEM auto-generates all of the type-specific functions at compile-time. It passes the header files containing the type definitions through `pycparser` [6], and uses a custom visitor to process each type definition. For each type, a list of tuples of (field name, field type, field size) is output to an intermediate file. This metadata is used by the parser generator to auto-generate the `parse_<type>` and `unparse_<type>` functions. The parse functions classify each field into `data`, `pyptrs`, or `cptrs` according to type, and advance the index into the byte string based on the field's size and the amount of padding between fields. The unparse functions fetch and unparse each field in order, concatenating the results together and adding padding as needed.

This auto-generation adds robustness to the parser and unparser. As new types are introduced to the runtime system, or fields are added or removed from existing types, no maintenance is required. Likewise, Owl accepts configuration options, specified as C defines, which may affect the type definitions; GEM's parser generator ensures that the parser and unparser are always consistent with the current configuration. Further, this auto-generation makes GEM extensible to other systems. With minor modifications to the parser generator, GEM could generate a parser and unparser for any system whose memory layout adheres to the principles outlined in Section 2.

3.3 Graph Transformation Layer

Converting the memory space into a graph unlocks a multitude of possible transformations; this section presents seven that are currently supported by GEM. Strategic composition of these transformations enables use cases including, but by no means limited to, the four that will be presented in Section 4.

3.3.1 Updating References

Moving even a single object in memory requires updating all references to it to reflect its new address. GEM provides two means of updating references. First, GEM can be given an offset by which to shift the entire heap; this involves shifting all intra-heap `pyptrs` and `cptrs` by this offset.

Second, given a mapping of old to new addresses, GEM can relocate all objects to their new addresses. To accomplish this, GEM iterates over the `pyptrs` and `cptrs` of each object. If it encounters a `pyptr` that is a key in the old-to-new mapping, it updates the reference to point to the corresponding new address. Updating the `cptrs` is slightly more complicated; while these references may not match any of the keys in the old-to-new mapping, they may point into an object whose base address is a key in the mapping. Thus, GEM checks if the base address of the object into which each `cptr` points has been shifted, and if so shifts the `cptr` by the difference between that object's old and new addresses.

3.3.2 Allocation and Garbage Collection

GEM requires a memory allocator and garbage collector. These need not use the same algorithms as the runtime system, as long as they maintain the proper structure of the free list. However, both the memory allocator and garbage collector within GEM mimic those used by Owl, for simplicity.

The memory allocator is first fit, splitting the free block if it significantly exceeds the requested size. Owl's garbage collector performs mark-and-sweep collection. GEM supplements this with an optional compacting pass, effectively providing mark-compact collection. Compaction moves all live objects into contiguous addresses, aided by the reference updating technique from Section 3.3.1, and then coalesces the free list.

3.3.3 Eliminating Duplicates

To identify duplicates, GEM builds a set of object "pools" using a deep equality checker. This equality checker takes a pair of `PyObjects`, directly compares their `data` and `cptrs`, and recursively compares their `pyptrs`. Objects deemed equal are placed in the same pool. Finally, pools of size one — representing unique objects — are filtered out.

Using the remaining pools of duplicates, GEM can free large chunks of memory by eliminating redundancies. For each pool, GEM chooses one representative object, and uses the technique from Section 3.3.1 to update all references to other objects in the pool to point to the representative. It then uses the garbage collector described in Section 3.3.2 to free the non-representative duplicates.

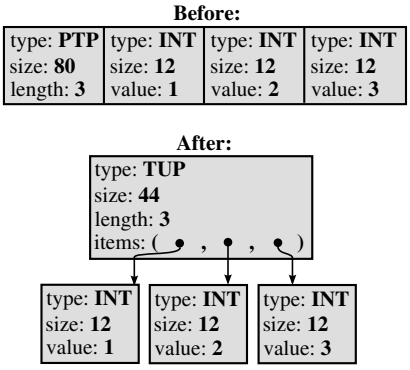
While GEM can consolidate all types of objects, in practice it only does so for immutable objects, as combining distinct mutable objects would violate the semantics of Python. Despite this, GEM finds substantial opportunities for deduplication, as Python has many immutable types including booleans, integers, floating points, strings, and tuples.

3.3.4 Unpacking Objects

As described in Section 2, Owl contains packed types which embed their constituents within themselves. During graph construction (Section 3.2), GEM generates individual nodes for the top-level object and each constituent, drawing edges from the top-level object to each constituent.

Unpacking a packed object consists of allocating an equivalent unpacked object and freeing the original object. First, GEM allocates the top-level unpacked object using the memory allocator from Section 3.3.2. Second, GEM creates references from the unpacked top-level object to the existing nodes for the constituents. Constituents which are themselves packed are unpacked recursively. As an example, figure 2 shows the same tuple before and after being unpacked.

Figure 2. Unpacking a packed tuple



Taken alone, unpacking consumes additional space, due to the extra layer of indirection. However, unpacked objects present opportunities for fine-grained de-duplication. Thus, unpacking can be a valuable asset when taken in conjunction with de-duplication, as will be shown in Section 5.2.

3.3.5 Splicing and Splitting

In addition to the intra-graph transformations discussed in Sections 3.3.1–3.3.4, GEM offers two transformations which operate on multiple graphs: splicing and splitting.

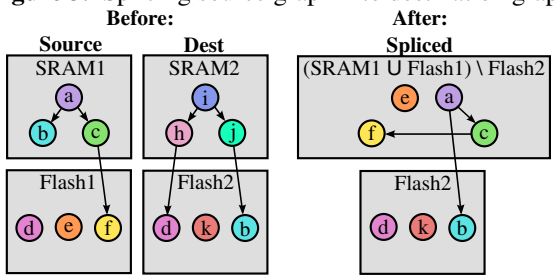
Splicing takes a source and destination graph and fuses one or more components of the source into one or more components of the destination. Splicing can be done with or without de-duplication; this section describes GEM's default behavior, which includes de-duplication. As an example, Figure 3 depicts splicing the entire source graph with the destination flash.

To accomplish this, GEM classifies the objects in the source component(s) to be spliced into two categories: those for which equivalent objects are present in the destination component(s), and those that have no equivalent in the destination component(s). This is achieved via an inter-graph duplicate search, using the procedure from Section 3.3.3. Once again, duplicates are only sought amongst immutable objects, to preserve program semantics.

For objects in the former category, no allocation is needed; GEM simply constructs a mapping from source addresses to destination addresses. For each unique object in the latter category, GEM allocates an equivalent object within the SRAM portion of the spliced graph, using the memory allocator from Section 3.3.2. During allocation, GEM updates the aforementioned mapping to map the original addresses of these objects to the addresses of the newly-allocated equivalents. Finally, GEM updates all references within the spliced graph according to this mapping, using the second technique from Section 3.3.1.

Splitting takes a single-component graph and partitions its objects into multiple components. GEM's splitting framework allows the user to input a custom algorithm for partitioning the objects. However, several different partitioning

Figure 3. Splicing source graph into destination graph



algorithms are already built into GEM, one of which will be presented in Section 4.3.1. GEM uses the input algorithm to split the objects into two disjoint subsets, and then constructs new graphs representing each subset. To eliminate the gaps introduced by partitioning the objects, GEM applies the compaction pass from Section 3.3.2 to each new graph.

4. Use Cases

The transformations and mechanisms described in Section 3 serve as building blocks which can be combined to support a multitude of use cases. To illustrate the versatility of GEM, this section presents four sample use cases. First, GEM can be augmented with a GUI to visualize the memory space. Second, it can be used to de-duplicate at compile-time, substantively decreasing code size. Also at compile-time, GEM can customize the layout of code across SRAM and flash, allowing the same runtime system to be deployed on a wide spectrum of memory architectures. Fourth, GEM can be used at runtime to transparently migrate a running program.

4.1 Memory Visualization

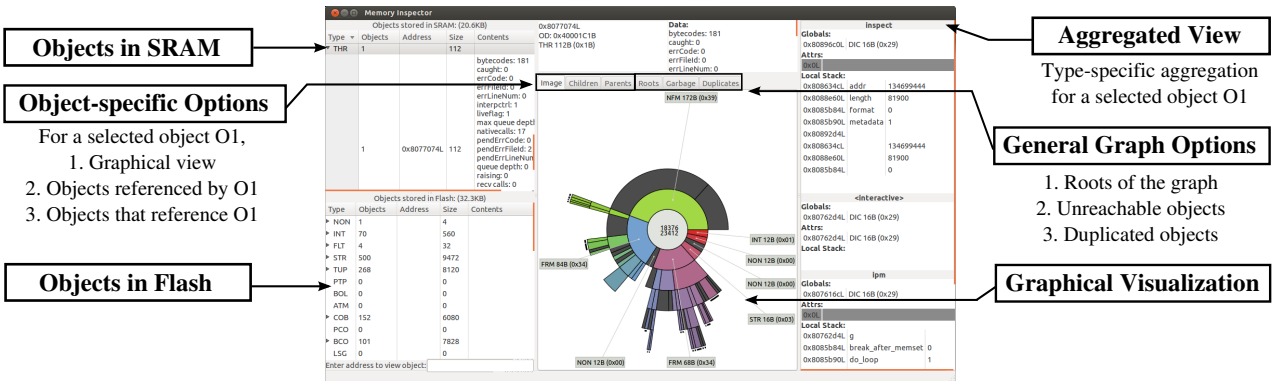
Extracting meaningful information from a raw heap dump is quite challenging. Thus, GEM supports a graphical visualizer which provides an organized view of the entire memory space of the microcontroller. This visualizer allows the user to easily view how objects are laid out in memory, see which objects are live at a given point in execution, and identify the most costly objects. The techniques from Section 3.3 are used to find and display duplicate objects and unreachable objects. GEM's visualizer is highly interactive, allowing the user to sort, search, navigate, and inspect objects.

4.1.1 User Interface

The GUI for GEM's visualizer is divided into three vertical panes, as shown in Figure 4. The left pane is a glossary of all objects in memory. It contains two sub-panes, one for SRAM and one for flash. Objects can be sorted by address, size, type, or value. A search bar enables lookup by address, and menu options allow for moving sequentially to the previous or next address. Selecting an object in the left pane updates several object-specific facets of the middle and right panes.

The middle pane offers six different display options. Three of these are specific to the selected object: textual

Figure 4. Graphical display for GEM's interactive memory visualizer



"parent" and "child" displays, plus a graphical "ringschart" display. An object's parent set consists of objects that reference it; its child set contains those objects that it references.

The graphical display is based on the open-source Linux Graphical Disk Usage Analyzer [1]. It displays a subset of the graph as a ringschart centered around the selected item. The center of the ringschart shows the total size of the weakly-connected component rooted in the selected object, broken down into SRAM and flash consumption. Each sector of the ringschart represents a reference to an object, and has an arc length proportional to the size of that object relative to sizes its "sibling" objects.

Hovering over a sector of the ringschart highlights that sector and spawns tooltips from its children, annotating them with type and size. If there are aliases, only one reference to the aliased object (sector in the ringschart) is colorized, with the rest displayed in grey. Hovering over a sector highlights all of its aliases. Rather than displaying the entire component rooted in the selected object, the GUI limits the depth of the rings to a number that can be configured via "zoom in" and "zoom out" options in the drop-down menu. It then annotates branches of the ringschart that are deeper than can be displayed. Clicking on a sector in the ringschart is equivalent to selecting the corresponding object from the left pane: the ringschart is re-centered around the selection, and all other object-specific panes are re-rendered.

The middle pane also includes three views of general properties of the graph: "roots", garbage, and duplicates. The roots view lists the addresses and types of the roots of the graph: global objects in memory which are not referenced by any other objects, and from which jointly all live objects are reachable. The garbage view lists objects that are not reachable from these roots. Last, the duplicates view displays each object pool — found using the technique from Section 3.3.3 — by listing the addresses of all objects in that pool.

The contents of the right pane are also specific to the selected object. This pane serves as an aggregated display for compound objects. It has special display modes for lists, tuples, sets, dictionaries, frames, threads, and code objects. For instance, in Figure 4, the selected object is a thread, and the right pane displays the stack of frames running in that thread. Each frame is tagged with the name, attributes, global variables, and local variables of the function to which it corresponds; each variable is tagged with its name, address, and current value.

4.2 De-duplication

The standard CPython compiler generates code objects that are fraught with duplicates. These code objects are then placed in read-only memory, from which they cannot be consolidated at runtime. The duplicates are relatively harmless on a desktop machine with vast resources. However, in an embedded system where memory is precious, eliminating duplicate objects within the Python code is critically important. GEM uniquely applies de-duplication to code objects at compile-time. Thus, the savings that GEM achieves are completely orthogonal to those of runtime de-duplication techniques such as interning.

For correctness, GEM limits de-duplication to immutable objects. In reality, this is no limitation at all, as all of the constants referenced by the Python code objects belong to immutable types. Due to the prevalence of duplicates, GEM's de-duplication yields substantial memory gains, as will be shown in Section 5.2.

4.2.1 Workflow

At a high-level, de-duplication proceeds as follows:

1. Build a graph of the library code, in Owl's default packed format, using the parser from Section 3.2.
2. Unpack the library modules, as described in Section 3.3.4.
3. Eliminate duplicates, as described in Section 3.3.3, using the compacting garbage collector to eliminate any gaps.
4. Unparse the graph using the unparser from Section 3.2.
5. Output the unparsed image to a C file, which will then be compiled into the runtime system by the Owl toolchain.

6. Post-compilation, find the base runtime address of the image and shift all references by this base address, again using the technique from Section 3.3.1.

In greater detail, de-duplication begins after Owl's image creator builds a binary representation of the Python library code. GEM parses this binary representation into a graph, using a base address of 0 since the runtime memory address of the library has not yet been determined.

The packed objects that Owl's image creator produces by default cannot be de-duplicated, since they directly contain their constituent objects. Therefore, the next step is to unpack these objects. GEM provides two levels of unpacking. It can indiscriminately unpack all objects; however, while the conversion to unpacked objects enables de-duplication, it introduces an overhead by reinstating references. Thus, GEM also provides a more sophisticated hybrid approach: selectively unpacking only those objects for which the projected savings due to de-duplication exceed the projected overhead due to reference reinstatement. Section 5.2 will show that either level of unpacking, in conjunction with de-duplication, yields significant savings, but the latter approach consistently surpasses the former.

After unpacking, GEM finds and consolidates duplicates, using the compacting garbage collector to regain a contiguous sequence of objects. Next, GEM determines the module paths — the addresses of the code objects representing the library modules. The graph is now in the correct structure, but its base address is still 0. Therefore, GEM stores the graph and its module paths, to be adjusted once the C linker has placed the image. It then passes unparsed versions of the graph and its module paths back to the image creator.

For now, GEM is done, and normal compilation resumes. The Owl toolchain auto-generates a C file containing the unparsed library image and module paths and compiles this file into the runtime system. At this point, the base runtime address of the library has been assigned, so GEM can adjust the intra-library references. It uses `readelf` to find this base address, reloads the de-duplicated graph that was stored previously, and shifts all references by an offset equal to the base runtime address. It likewise increments each module path by the base address. Finally, GEM unparses the library and the module paths back into binary representations and overwrites the ELF file with the adjusted versions.

4.3 Heterogeneous Compilation

Enabling a runtime system to fit within a broad spectrum of microcontroller memory architectures eases the construction of heterogeneous embedded systems. GEM uniquely supports customization of the runtime system to a particular memory architecture by partitioning the Python library code amongst SRAM and flash. The amount of SRAM and flash available for these libraries can be specified at compile-time; given these constraints, GEM will compile and partition the runtime system, failing only if the capacity is too low.

4.3.1 Workflow

To increase the likelihood of meeting the given memory constraints, heterogeneous compilation is bracketed by the de-duplication procedure outlined in Section 4.2.1. Thus, its complete compile-time workflow is as follows:

1. Build a graph of the library code in Owl's default packed format, using the parser from Section 3.2.
2. Unpack the library modules, as described in Section 3.3.4.
3. Eliminate duplicates, as described in Section 3.3.3.
4. Partition the library code between SRAM and flash, using the technique from Section 3.3.5.
5. Unparse both graphs using the unparser from Section 3.2.
6. Output the unparsed image to a C file, which will then be compiled into the runtime system by the Owl toolchain.
7. Post-compilation, find the base runtime address of the image and shift all references by this base address, using the technique from Section 3.3.1.

In addition, the objects designated for SRAM are installed in the heap at runtime using a six-step process:

1. Boot the runtime system.
2. Dump the memory using the technique from Section 3.1.
3. Parse the memory into a graph, as in Section 3.2.
4. Splice the SRAM modules into the current memory graph using the technique from Section 3.3.5.
5. Unparse the augmented graph, as in Section 3.2.
6. Memset the augmented graph using the technique from Section 3.1.

As with de-duplication, GEM enters the compilation process just after Owl's image creator builds the binary representation of the packed Python library code. GEM then performs the same parsing, graph construction, unpacking, and duplicate elimination as it did for de-duplication. However, rather than compacting the objects into a single contiguous chunk of memory, it partitions these objects into two components, one for flash and one for SRAM.

For this use case, the partitioning algorithm must adhere to three constraints. First, it must respect the upper bounds on each memory region's size specified at compile-time. Second, since the SRAM modules will not be loaded until the end of boot-time, it must ensure that all modules necessary to boot the runtime system are relegated to flash. Third, it must guarantee that nothing placed in flash references anything placed in SRAM, since the runtime SRAM addresses are not known prior to boot time. Some microcontrollers cannot overwrite flash memory during execution, and it's expensive for those that can; additionally, references from flash

into SRAM would become dangling references upon reboot, potentially causing the runtime system to crash.

GEM currently uses a greedy algorithm to satisfy these constraints, which the evaluation in Section 5.3 proves effective. This algorithm begins by placing only those objects that are required to boot the runtime system in flash, and assuming that all other objects are in SRAM. To satisfy the third constraint, the algorithm promotes objects to flash on the granularity of components, where each candidate component consists of a given object plus all other objects still in SRAM that are reachable from that object. It greedily chooses the largest component in SRAM that will fit in flash without exceeding capacity, terminating when even the smallest component left in SRAM is larger than the remaining capacity. If the size of the remaining SRAM component exceeds the specified bounds, the algorithm reports failure; otherwise, GEM proceeds with compilation.

After executing this algorithm GEM builds and compacts two new graphs, one for each memory region, and assigns a temporary base address to each. GEM then stores both graphs and their module paths for later reference adjustment and resumes the normal compilation process. As in deduplication, it adjusts references to objects in flash once the runtime address of the flash library has been established.

Finally, the SRAM modules must be allocated on the heap. The runtime system is first booted into a special SRAM installation mode which imports only the modules needed to perform `dump` and `memset`. The mode minimizes the set of modules that are required to be placed in flash at compile-time, widening the range of flash sizes that GEM can accommodate. The memory space is then dumped and parsed into a graph format, using the techniques from Sections 3.1 and 3.2, and the SRAM graph built during compile-time is spliced in. During this splicing process, GEM keeps track of the addresses assigned to any top-level modules. Last, GEM unparses and memsets the graph, overwriting the current heap and augmenting the module paths with the addresses of the SRAM modules.

4.4 Transparent Migration

Transparent migration, which preserves the runtime state, is valuable in many scenarios. For instance, if a device fails, migrating a pre-crash checkpoint to another device prevents complete loss of work. Likewise, a system with substantial startup delays may benefit from migration of a pre-booted image [28]. While many existing systems support transparent migration, the value of GEM's approach lies in the ease with which it harnesses graph transformations — in particular, its novel splicing technique — to enable migration between devices with disparate hardware and software.

4.4.1 Workflow

In theory, a program running upon a runtime system can be migrated by transplanting the heap. However, as described in Section 3.1, the base address of the heap is not guaranteed to be identical across all instances of the runtime system. Further, the contents of flash may not be the same at the source and destination, which is problematic as objects on the heap may reference flash. GEM's `memset` command automatically corrects for these differences by adjusting all intra-heap references and allocating all missing flash objects in SRAM — since, as mentioned in Section 4.3.1, many microcontrollers do not support mutating flash at runtime.

Because `memset` automatically addresses these challenges, GEM's migration process is quite simple:

1. Dump the source memory, as described in Section 3.1.
2. Parse the memory into a graph, as in Section 3.2.
3. Optionally perform any desired transformations such as de-duplication or free list compaction.
4. Save the snapshot of the transformed graph.
5. Memset the snapshot onto the destination using the technique from Section 3.1.

In the prototype implementation, the `dump` and `memset` are initiated by the user, though they could be automated. To allow the user to initiate a `dump` or `memset`, GEM includes two mechanisms by which the user can pause the program and access the interactive prompt. First, if the programmer knows the point in the program at which he or she wishes to migrate in advance, he or she can insert a call to a built-in function which pauses execution.

However, the programmer may not always anticipate wishing to migrate. Thus, GEM extends Owl's interpreter to support pausing at any time by pressing a button on the microcontroller. Once the program is paused, the user can access the interactive prompt to perform migration. While manual activation serves as an effective proof-of-concept, the same pause/resume logic could be combined with Owl's message-passing capabilities to enable remote triggering.

5. Evaluation

Each of the use cases from Section 4 was evaluated on a series of benchmarks, where each benchmark is a snapshot of a specific workload running on a specific platform, named in the form <platform>_<workload>. GEM was evaluated on three platforms: a Stellaris LM3S9B92 microcontroller ("Stellaris"), an STM32F4-Discovery microcontroller ("STM32"), and a desktop machine ("Desktop"). The LM3S9B92 has 96 KB of SRAM and 256 KB of flash; the F4-Discovery board has 192 KB of SRAM and 1 MB of flash. The workloads include two that are not application-specific — compile time ("compile") and boot time ("boot") — as well as an application that uses an accelerometer and TFT display to present an artificial horizon display ("ahd").

5.1 Memory Visualization

GEM's memory visualizer has exposed multiple opportunities to improve the system design of Owl. For instance,

Table 1. Duplicate Objects in Unpacked Library Code

	All Objects		Intra-Module Duplicates		Inter-Module Duplicates		All Duplicates	
Type	Count	Total Size (B)	Count	Total Size (B)	Count	Total Size (B)	Count	Total Size (B)
None	132	528	108	432	23	92	131	524
Integer	111	888	17	136	27	216	44	352
Float	5	40	1	8	0	0	1	8
String	1418	23156	711	11212	213	2776	924	13988
Tuple	604	11340	284	2708	58	692	342	3400
Bytecode	111	7876	37	852	11	244	48	1096
Code Object	148	5920	0	0	0	0	0	0
Native Object	121	968	0	0	0	0	0	0
Total	2687	51568	1158	15348	332	4020	1490	19368

Table 2. Size of Python Libraries (KB)

Platform	Unpacked	Packed	Unpacked, De-duplicated	Hybrid
Desktop	50.4	41.1	31.5	31.3
Stellaris	86.4	71.4	62.7	60.3
STM32	97.1	80.9	70.3	68.9

GEM's memory visualizer inspired new memory formats for storing the Python library code in flash. Originally, Owl used the same unpacked, duplicated object format as CPython. However, using GEM to analyze the contents of flash revealed considerable wasted space. Therefore, two new library formats were proposed: the packed format described in Section 2, and the de-duplicated format from Section 4.2.

A comparison of the original and packed formats, which will be presented in Section 5.2, validates the intuition that the references in the unpacked version consume an exorbitant amount of space. Yet, further analysis using GEM's memory visualizer exposed many duplicates within these packed code objects, as shown in Table 1. GEM indicated that the gains to be had by eliminating duplicates would outweigh the losses associated with adding references. Both proposed formats were incorporated into Owl, and the user may now choose which format is used at compile-time.

5.2 De-duplication

Since de-duplication occurs at compile-time, it was evaluated on the *_compile benchmarks. For each benchmark, the Python library code was stored in four formats: the unpacked format output by the CPython compiler, the packed format inspired by GEM's visualizer, and two de-duplicated formats which utilize the procedure described in Section 4.2, first with naive unpacking and second with selective unpacking based on a heuristic that approximates the potential savings. Table 2 shows the space consumed by the library code in each of these four formats.

The modules included in the Python library vary across platforms. Many modules are platform-independent, such as `math`, `time`, and `types`. The desktop platform is smallest, as it consists solely of these platform-independent modules. Stellaris and STM32 include additional platform-specific modules to support hardware peripheral access.

Across all three platforms, GEM's de-duplication consistently saved memory. Compared to the unpacked format used by CPython, the savings amounted to 19.1 KB, 26.1 KB, and 28.6 KB for desktop, Stellaris, and STM32, respectively; in each case this was a decrease of more than 29%. Table 1 breaks down the de-duplication performed by GEM by type. Strings accounted for over 72% of the de-duplication savings; the next largest sources of redundancies were tuples and bytecodes. Note that significant duplication occurs even amongst nested objects such as tuples.

Even as compared to the compact packed format, de-duplication yielded savings of 9.8 KB, 11.1 KB, and 12.0 KB for the three platforms: improvements of 14.8–23.8%. The percent improvement was greatest for the desktop version. However, Stellaris and STM32 yielded better absolute decreases. This is due to the fact that the extra platform-specific modules within Stellaris and STM32 primarily contain constants such as register addresses, and thus have an unusually small proportion of duplicate objects.

The presence of modules with a low proportion of duplicate objects motivated the creation of the hybrid format described in Section 4.2. The difference in size between the third and fourth formats in Table 2 indicates that GEM's hybrid approach elected to leave some low-redundancy modules packed, as it found insufficient duplicates to offset the overhead of unpacking. This selectivity proved profitable for each platform, saving up to an additional 2.4 KB.

5.3 Heterogeneous Compilation

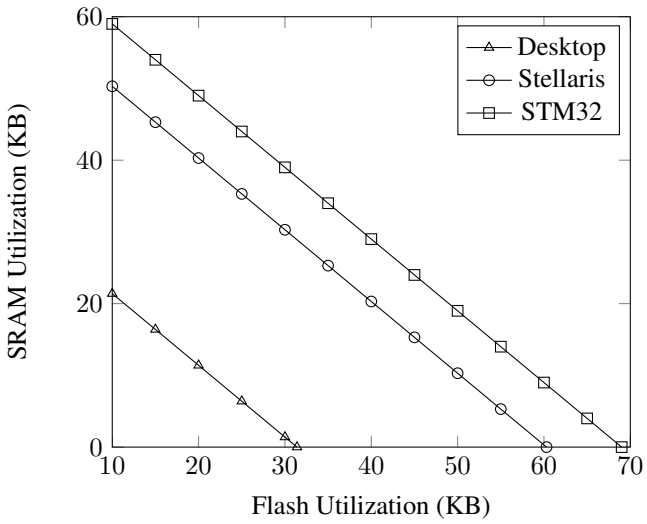
By default, the Owl toolchain places the Python library code in a separate region of memory from the heap. On the microcontroller platforms, this region is flash; on the desktop platform, it is a read-only region of RAM. For the purposes of brevity, both types of read-only regions will be referred to as "flash" in this section.

GEM enables custom code placement to accommodate systems with varied memory shapes. Figure 5 presents examples of valid (flash, SRAM) divisions supported by GEM for each platform. Note that flash has a lower bound of ap-

Table 3. Intra-Platform Migration, Sizes (KB)

Migration				Size Before			Size After			% Change
Benchmark	Flash Cap Before	Flash Cap After	De-duplicated?	SRAM	Flash	Total	SRAM	Flash	Total	Total
Stellaris_ahd	∞	80.0	No	34.2	109.4	143.7	64.1	80.0	144.1	0.3%
Stellaris_ahd	∞	80.0	Yes	34.2	109.4	143.7	61.2	80.0	141.2	-1.7%
Stellaris_ahd	80.0	∞	No	60.5	80.0	140.5	60.8	109.4	170.2	21.2%
Stellaris_ahd	80.0	∞	Yes	60.5	80.0	140.5	30.7	109.4	140.1	-0.3%

Figure 5. Valid (Flash, SRAM) Configurations



proximately 10 KB, as a small set of modules needed to boot the virtual machine must reside in flash.

GEM's heterogeneous compilation process de-duplicates not only within flash, but across SRAM and flash. Therefore, the total amount of space consumed by a given platform's library code is constant regardless of the flash/SRAM breakdown; there is no net disadvantage to moving code to SRAM. Further, collecting the data in Figure 5 revealed that the greedy algorithm described in Section 4.3 successfully partitions the objects without wasting memory. Out of 30 datapoints, flash was always filled to within 4 B of capacity, and was filled exactly to capacity 83.3% of the time.

5.4 Transparent Migration

GEM facilitates migration across different memory architectures and across different platforms. Such heterogeneity complicates migration; since the contents of flash may be different at the source and destination, blindly migrating the SRAM may result in missing or duplicated objects. As described in Section 4.4, GEM handles missing objects by allocating these objects within the SRAM component of the graph prior to overwriting the heap. Redundancies can in turn be eliminated by de-duplicating just before memsetting.

Table 3 shows the results of migration across the same platform (Stellaris), but between different memory architectures. It presents the memory distribution of the artificial horizon display application before and after migration between a system with sufficient space to fit all of the code in flash memory (flash cap of ∞) and a system with only 80 KB of flash. Further, it shows the resultant memory layout with and without de-duplication. Note that for these experiments the library modules were previously de-duplicated during compilation; all savings due to de-duplication during migration result from consolidating runtime structures.

Even without de-duplication, migration from a system with more flash to one with less introduced virtually no overhead (0.3%). Objects in the source flash that were absent from the destination flash were simply placed in the destination SRAM; thus, while the memory breakdown at the destination is different, the total consumption is roughly the same. The negligible overhead comes from a small amount of build-specific information. Performing de-duplication during migration re-gained 2.9 KB, more than compensating for this overhead.

In contrast, migration from a system with less flash to one with more flash resulted in a 21.2% overhead without de-duplication. This is because the complete contents of the source heap were placed at the destination, with no regard for the fact that much of what was relegated to SRAM on the source already resided in flash at the destination. This was easily solved with de-duplication, which eliminated the extraneous modules as well as some redundant runtime structures, reclaiming over 30 KB for a net gain upon migration.

Migration of the artificial horizon display application between the Stellaris and STM32 platforms highlights GEM's cross-platform capabilities. Chosen for its realistic workload, this application requires hardware peripherals at the source and destination. However, the peripherals at the source and destination need not be identical, so long as they are both compatible with the application. For these experiments the same model of LCD display was used for both Stellaris and STM32, but the Stellaris board was connected to an external accelerometer whereas the STM32 board utilized its on-board accelerometer. Table 4 presents the results of this migration with and without de-duplication. Without de-duplication, migration added an overhead of approximately 1–2%; however, de-duplication once again regained all of the overhead and then some.

Table 4. Inter-Platform Migration, Sizes (KB)

Migration		Size Before			Size After			% Change
Benchmark → Platform	De-duplicated?	SRAM	Flash	Total	SRAM	Flash	Total	Total
Stellaris_ahd → STM32	No	34.4	128.2	162.5	37.2	128.5	165.7	2.0%
Stellaris_ahd → STM32	Yes	34.4	128.2	162.5	33.0	128.5	161.5	-0.6%
STM32_ahd → Stellaris	No	34.3	128.5	162.8	37.4	128.2	165.5	1.7%
STM32_ahd → Stellaris	Yes	34.3	128.5	162.8	33.2	128.2	161.4	-0.8%

6. Related Work

Considerable past work has used memory graphs for program visualization [8, 16, 26, 30, 32, 43]. However, GEM's visualizer diverges from past work by catering to a different domain and audience.

First, existing tools were designed to profile applications running on systems with extensive resources. Such applications may have over a million live objects on the heap [10]. Thus, many existing tools abstract away details to make the graph manageable. Some do not include nodes for primitive types [8, 26]; others collapse individual objects of the same type into a single node [32]. In contrast, GEM targets resource-constrained embedded systems, which have far fewer live objects. With a minimum object size of 12 B, Owl's default 80 KB heap fits only a few thousand objects. This makes it tractable for GEM to provide a finer level of detail by including a node for each object on the heap. Yet, GEM still provides aggregated views of compound objects.

Second, the primary audience of existing tools is the application developer. While GEM also benefits the application developer, it primarily targets the system developer. Many design choices made by existing tools are not optimal for GEM's target audience. For instance, one tool constructs a graph using logging data that is unavailable at boot-time [30]. This is suitable for application profiling, but insufficient for system profiling, as it excludes objects created during start-up that persist unmodified. Another extracts the heap by querying GDB from the roots, thereby excluding garbage and unused code [43]. GEM includes both, as they highlight opportunities for system-level improvements.

Those tools that do capture a full snapshot of the heap perform the aforementioned abstractions. Sacrificing detail for simplicity is a reasonable choice when presenting a million-object heap to a programmer who has no interest in system-level details. However, this conceals information that, when displayed by GEM, inspired improvements such as the de-duplicated code object format. Further, since the programmer has no control over the layout of the code, existing tools uniformly focus on the runtime data and only snapshot the heap [8, 16, 26, 30, 32, 43]. GEM captures a snapshot of both the data on the heap and the code in flash.

Additionally, GEM provides a mechanism by which the user can obtain a snapshot at any point in execution, without advance warning. Several existing tools provide similar flexibility, but at the cost of slowing execution by continuously logging [16, 30], or excluding unreachable objects [43]. Other tools sacrifice flexibility to avoid such pitfalls, and instead either automatically select points to snapshot based on memory utilization [26], or require that the application developer specify where to snapshot in advance [8].

Though GEM's memory visualizer differs significantly from prior visualization tools, the primary novelty of GEM lies in its use of memory graphs to not only inspect memory but transform it. GEM uniquely structures memory mutations as graph transformation passes, and uses these transformation to impact system memory. While GEM's versatile framework for applying these transformations is unique, several individual transformation passes leverage past work.

In particular, GEM's techniques for compaction and reference shifting based on an address mapping closely resemble mark-compact garbage collection, which slides in-use objects towards one end of the heap by maintaining a "break table" which it uses to update references [11, 14, 27, 37]. Likewise, GEM's reference shifting based on an offset builds on existing techniques for portable migration. One such technique first converts all references to offsets relative to the beginning of the checkpoint, and then converts them back to absolute addresses once the checkpoint has been migrated [31]. GEM achieves the same result without this intermediate step, shifting references by the difference between the base addresses of the source and target heaps.

Similarly, the three transformative use cases built upon GEM adapt past work to the domain of embedded systems. Operating systems and storage systems commonly employ de-duplication at runtime [21, 24, 25, 35]. They eliminate duplicates at either the file level or the block level, since large-scale duplication results from storing multiple versions of the same file. Programs exhibit different patterns of duplication from storage systems. Large blocks are not redundant; duplication occurs at the granularity of individual objects. Instead, GEM's de-duplication is similar to runtime interning, which is supported by runtime systems such as the Oracle JVM and CPython [3, 19, 23]. However, runtime interning misses opportunities to consolidate objects within the code. GEM uniquely performs de-duplication at compile-time, eliminating the longest-lasting redundancies.

Likewise, snapshots and migration are well-studied. Within the domain of embedded systems, existing snapshot techniques were primarily designed for rollback and recovery, and feature design decisions which run contrary to GEM's dual goals of preserving memory and enabling mi-

gration. They either suffer from large memory overheads by storing a complete copy of the memory space on the device [15], or achieve space efficiency by generating partial snapshots [36, 41] which are insufficient for migration.

Alternative migration techniques have been developed for mobile agents which operate only at the moment of migration, as opposed to checkpointing continuously during execution [17, 20, 34]. These techniques instrument the source code, inflating code size. GEM's migration requires no changes to the application code, and only the addition of a single bytecode to the runtime system. Additionally, these techniques for mobile agents migrate one thread at a time, whereas GEM migrates the entire runtime state, including all threads and scheduling information.

Other work has similarly expanded the unit of migration, to an entire virtual machine [13, 40] or operating system [18]. Further, shadow drivers have been used to transparently migrate between platforms with equivalent, but not identical, hardware devices [22]. However, the mid-range microcontrollers which GEM targets lack sufficient resources for this extra layer of abstraction. To enable migration between devices with different images in flash, GEM instead performs a series of off-line transformations, requiring no additional resources at runtime and capitalizing on the ease of manipulating a graph representation.

The final use case presented in this paper, compilation for heterogeneous memory architectures, accepts upper bounds on SRAM and flash that can be allocated to the Python library code. Other compilers for embedded systems similarly accept code-size parameters [29]. Rather than using this information to divide the code between different memory regions, as GEM does, they instead carefully craft code that will fit within a single memory space.

No other tool simultaneously supports all four use cases of GEM. At best, efforts have been made to combine two: prior work integrated de-duplication into migration to minimize latency [12, 33, 42]. Designed to hasten migration, the impact that these techniques have on memory is fundamentally different from that of GEM. One requires round-trip migration in order to see any space savings [42]; the others de-duplicate during a single migration, but at a page granularity rather than GEM's object granularity [12, 33].

7. Conclusions

Memory analysis and optimization in embedded systems is complicated by resource constraints and heterogeneity. By imposing structure on memory, managed runtime systems enable the development of tools to automate and simplify these tasks. This paper has presented one such tool, GEM, which leverages structured memory to refashion memory as a graph, facilitating its transformation. Designed for generality and synergy, GEM's flexible framework substantially eases the burden of memory optimization by allowing the same underlying graph transformation passes to be combined to implement a multitude of capabilities.

Four such capabilities have been implemented and evaluated in this paper: visualization, de-duplication, heterogeneous compilation, and transparent migration. Though the primary contribution of GEM is its versatile infrastructure and novel low-level transformations, the sample use cases built upon it exemplify the value that it brings to real challenges of managing embedded memory. GEM's interactive visualizer has facilitated the identification of system-level inefficiencies; its de-duplication has reclaimed up to 24% of the space consumed by the Python library code; its heterogeneous compilation has broadened the range of memory architectures within which the virtual machine can fit; and its transparent migration has enabled the migration of a running program, even amidst hardware and software incongruities between the source and destination.

GEM was implemented for an existing embedded runtime system, Owl. However, the concept of modeling memory as a graph, as well as the transformations that memory graphs facilitate, transcend any specific system. Thus, GEM was designed to be portable. In particular, the parser and unparser which translate between the system-specific object model and GEM's much more generic graph representation are auto-generated. With only minor changes to the parser generator, GEM could be ported to other managed runtime systems with well-defined type systems and memory organization.

8. Acknowledgements

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. 1450681.

References

[1] Linux graphical disk usage analyser. `https://wiki.gnome.org/Apps/Baobab`.

[2] elua. `http://www.eluaproject.net/`.

[3] Java language and virtual machine specifications. `http://docs.oracle.com/javase/specs/`.

[4] owl. `http://www.embeddedpython.org/`.

[5] p14p. `http://code.google.com/p/python-on-a-chip/`.

[6] pycparser. `http://pypi.python.org/pypi/pycparser/`.

[7] Micro python. `http://micropython.org/`.

[8] E. E. Aftandilian, S. Kelley, C. Gramazio, N. Ricci, S. L. Su, and S. Z. Guyer. Heapviz: Interactive heap visualization for program understanding and debugging. In *ACM Symposium on Software Visualization*, 2010.

[9] T. W. Barr, R. Smith, and S. Rixner. Design and implementation of an embedded python run-time system. In *USENIX Annual Technical Conference*, 2012.

[10] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee,

J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Weidermann. The dacapo benchmarks: Java benchmarking development and analysis. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2006.

[11] C. J. Cheney. A non-recursive list compacting algorithm. In *Communications of the ACM*, volume 13, 1970.

[12] J.-H. Chiang, H.-L. Li, and T. cker Chiueh. Introspection-based memory de-duplication and migration. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2013.

[13] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Symposium on Networked Systems Design and Implementation*, 2005.

[14] J. Cohen and A. Nicolau. Comparison of compacting algorithms for garbage collection. In *ACM Transactions on Programming Languages and Systems*, volume 5, 1983.

[15] A. Cunei and J. Vitek. A new approach to real-time checkpointing. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2006.

[16] A. Erkan, T. VanSlyke, and T. M. Scaffadi. Data structure visualization with latex and prefuse. In *SIGCSE Conference on Innovation and Technology in Computer Science Education*, 2007.

[17] S. Fünfrocken. Transparent migration of java-based mobile agents. In *Personal Technologies*, volume 2, 1998.

[18] J. G. Hansen and E. Jul. Self-migration of operating systems. In *Workshop on ACM SIGOPS European Workshop*, 2004.

[19] M. Horie, K. Ogata, K. Kawachiya, and T. Onodera. String deduplication for java-based middleware in virtualized environments. In *ACM SIGPLAN/SIGOPS International Conference on Virutal Execution Environments*, 2014.

[20] T. Illman, T. K. F. Kargl, and M. Weber. Transparent migration of mobile agents using the java platform debugger architecture. In *Mobile Agents*, 2001.

[21] K. Jin and E. L. Miller. The effectiveness of deduplication on virtual machine disk images. In *Israeli Experimental Systems Conference*, 2009.

[22] A. Kadav and M. M. Swift. Live migration of direct-access devices. In *The First Workshop of I/O Virtualization*, 2008.

[23] K. Kawachiya, K. Ogata, and T. Onodera. Analysis and reduction of memory inefficiencies in java strings. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2008.

[24] H. S. Koppula, K. P. Leela, and A. Agarwal. Learning url patterns for webage de-duplication. In *ACM International Conference on Web Search and Data Mining*, 2010.

[25] P. Kulkarni, F. Douglis, J. LaVoie, and J. M. Tracey. Redundancy elimination within large collections of files. In *USENIX Annual Technical Conference*, 2004.

[26] M. Marron, C. Sanchez, Z. Su, and M. Fahndrich. Abstracting runtime heaps for program understanding. In *IEEE Transactions on Software Engineering*, volume 39, 2013.

[27] P. McGachey and A. L. Hosking. Reducing generational copy reserve overhead with fallback compaction. In *International Symposium on Memory Management*, 2006.

[28] M. F. Mergen, V. Uhlig, O. Krieger, and J. Xenidis. Virtualization for high-performance computing. In *ACM SIGOPS Operating Systems Review*, volume 40, 2006.

[29] M. Naik and J. Palsberg. Compiling with code-size constraints. In *ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, 2002.

[30] S. Pheng and C. Verbrugge. Dynamic data structure analysis for java programs. In *IEEE International Conference on Program Comprehension*, 2006.

[31] B. Ramkumar and V. Strumpen. Portable checkpointing for heterogeneous architectures. In *International Symposium on Fault-Tolerant Computing*, 1997.

[32] S. P. Reiss. Visualizing the heap to detect memory problems. In *Visualizing Software for Understanding and Analysis*, 2009.

[33] C. P. Sapuntzakis, R. Chandra, B. Pfaff, and J. Chow. Optimizing the migration of virtual computers. In *USENIX Symposium on Operating Systems Design and Implementation*, 2002.

[34] T. Sekiguchi, H. Masuhara, and A. Yonezawa. A simple extension of java language for controllable transparent migration and its portable implementation. In *Coordination Languages and Models*, 1999.

[35] P. Sharma and P. Kulkarni. Singleton: System-wide page deduplication in virtual environments. In *International Symposium on High-Performance Parallel and Distributed Computing*, 2012.

[36] R. Smith and S. Rixner. Surviving peripheral failures in embedded systems. In *USENIX Annual Technical Conference*, 2015.

[37] S. Stanchina and M. Meyer. Mark-sweep or copying? a 'best of both worlds' algorithm and a hardware-supported real-time implementation. In *International Symposium on Memory Management*, 2007.

[38] *STM32F20xx / STM32F207xx*. STMicroelectronics, 2013.

[39] *STM32F373xx*. STMicroelectronics, 2014.

[40] T. Suezawa. Persistent execution state of a java virtual machine. In *ACM Conference on Java Grande*, 2000.

[41] H. Tabkhi, S. G. Miremadi, and A. Ejlali. An asymmetric checkpointing and rollback error recovery scheme for embedded processors. In *IEEE International Symposium on Defect and Fault Tolerance of VLSI Systems*, 2008.

[42] K. Takahashi, K. Sasada, and T. Hirofuchi. A fast virtual machine storage migration technique using data deduplication. In *International Conference on Cloud Computing, GRIDs, and Virtualization*, 2012.

[43] T. Zimmerman and A. Zeller. Visualizing memory graphs. In *Lecture Notes in Computer Science – Software Visualization*, 2002.