An Automated Approach for Estimating the Memory Footprint of Non-linear Data Objects

Sebastian Dreßler and Thomas Steinke

Zuse Institute Berlin
Takustraße 7
14195 Berlin
{dressler,steinke}@zib.de

Abstract Current programming models for heterogeneous devices with disjoint physical memory spaces require explicit allocation of device memory and explicit data transfers. While it is quite easy to manually implement these operations for linear data objects like arrays, this task becomes more difficult for non-linear objects, e.g. linked lists or multiple inherited classes. The difficulties arise due to dynamic memory requirements at run-time and the dependencies between data structures. In this paper we present a novel method to build a graph-based static data type description which is used to create code for injectable functions that automatically determine the memory footprint of data objects at run-time. Our approach is extensible to implement automatically generated optimized data transfers across physical memory spaces.

1 Introduction

Current programming models for heterogeneous systems with disjoint address spaces require an explicit data transfer. Therefore, data management and migration between devices must be implemented by the programmer. For linear data objects like arrays these tasks are rather simple to implement. Specific programming models, e.g. OpenACC or the Intel Compiler for Intel Xeon Phi, already enable implicit transfers. However, non-linear data objects are not sufficiently supported due to the need for serialization before transfers. This is caused by a lack of sophisticated automatic serialization.

We want to address implicit data movements and propose a three step solution. First, the memory footprint (MF) and transfer direction (intent) of data objects must be determined. Second, an automated serialization and deserialization scheme must be implemented. As third step, code executing the MF determination, serialization and the data transfer must be injected.

In this paper, we focus on the first step. We present an approach that analyzes non-linear data objects and estimates their MF dynamically at run-time alongside with the objects intent. Related methods focus on existing compilers or implement altered programming models. In contrast, our approach utilizes the Low Level Virtual Machine (LLVM) compiler framework. LLVM introduces

D. an Mey et al. (Eds.): Euro-Par 2013 Workshops, LNCS 8374, pp. 249–258, 2014.

an intermediate code representation (IR) where higher level abstractions are disclosed. Thus, information like data types become directly deducible. This enables language independence and removes the need for additional object metadata in our approach.

As we show in the evaluation section, we are able to process a wide range of simple and non-linear data objects, namely scalars, multi-dimensional arrays, and C++ classes with templates, multiple inheritance and iterators with both, static and dynamic allocation. The currently existing limitations concern overlapping pointers, function pointers, and re-allocation. The main contributions of the paper are: i) the design and implementation of two LLVM passes for analyzing data structures and to determine the data-direction of function arguments, and ii) a method for processing simple and non-linear data objects to derive run-time properties (MF) without adding additional metadata.

The paper is structured as follows. Next, we provide an overview on related work. Subsequently, we discuss our implemented analysis process. Then, the handling of different data objects is discussed in detail. That is, we show how they are internally represented and how their MFs are estimated. The evaluation section presents results of several synthetic tests and of real-world applications and discusses the run-time overhead for selected tests. We conclude the paper with a summary and an outlook.

2 Related Work

Determining the MF of data objects relates to type checking compilers. SAFECode [3] for instances provides array bounds checking. The MF of the array must be known to perform this task. Currently, only linear arrays are supported, while our approach also enables analysis of non-linear objects.

HiCUDA [5] targets heterogeneous systems. Allocations and transfers are performed automatically but require *#pragma* directives. Additionally, information regarding the shape of the data must be provided. We want to remove these constraints by performing an automated shape recognition.

Uchiyama et al. [10] propose an automated memory allocation method for OpenMPC [9]. However, for performance reasons CUDA as programming model is often favored. Therefore, the necessity to analyze data objects in existing applications remains. With the use of LLVM, our approach only requires support by a LLVM language frontend, e.g. Clang for C / C++ or DragonEgg for FORTRAN and other languages supported by GCC.

Jablin et al. [7] propose the first fully automatic CPU-GPU communication management. This approach enables the programmer to stay with CUDA. Furthermore, CPU pointers are mapped directly to GPU pointers which means that no serialization before transfers is required. However, once the programmer does not access GPU data in the same way as on the CPU, data must be re-ordered which implies serialization.

In our previous work [4] we focused on simple data objects in C. This paper focuses non-linear data objects with abstractions toward language independence.

3 The Data Type Analysis Process

This section describes the details of the process of analyzing object types and estimating their memory footprint (MF). Fig. 1 shows its general outline and also represents the outline of the discussion.

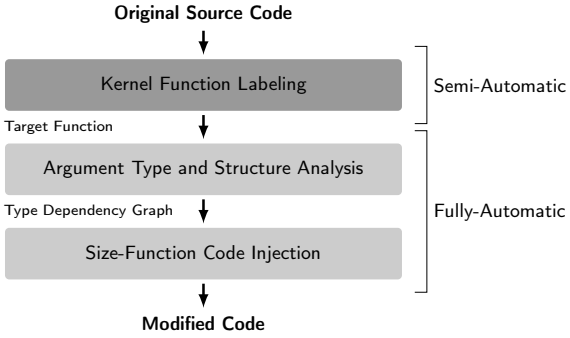

Fig. 1. Representation of the tool flow for automatically estimating the memory footprint of data objects

The analysis process and the subsequent creation of size-functions to estimate the MF both utilize LLVM [8]. In conjunction with LLVM we introduce a graph-based representation of the objects data structure which simplifies the subsequent analysis process.

3.1 Labeling of Kernel Functions

The labeling of a kernel function (KF) marks the function to be analyzed. This is a semi-automatic task since the developer manually chooses the KF. For labeling, source code *#pragma* statements are not an option, since they are not visible in LLVM IR. Instead we use the compiler attribute *annotate*: `__attribute__((annotate("kernel")))` is placed prior the KF. In LLVM IR the attribute is a global variable which we search for during analysis.

3.2 Analysis of the Structure of Function Argument Types

This step reveals the structural type layout of an argument of the selected KF. The goal of the analysis of the type structure is to enable the construction of formulas expressing the MF of instances of data types. For instance, consider the class type definition `class A { int x, y; }`. The MF of an object `A a;` is expressible as $2 \times \text{sizeof}(int)$ bytes. Type structure information are directly deduced from LLVM IR during type structure analysis. We store the structural layout in a graph, the *type dependency graph* (TDG), avoiding repetitive analysis.

The TDG is a directed graph. Its vertices represent data types and are labeled with the name of the type and its byte-width, if the data type is a scalar. The TDG may contain cycles to represent recursive type structures. Furthermore, the KF is added as root vertex. This ensures a common known entry point and also makes the TDG unique across different KFs.

If a vertex represents a non-linear data type it contains at least a single outgoing edge. These edges branch the non-linear data type into known subtypes. Edges are labeled with an index containing the position of a subtype in its source type. Fig. 2a depicts the TDG for the previously defined class `A`.

If a variable is a (multi-dimensional) pointer, its base type is extracted. The corresponding edge is labeled as pointer-edge. Its target vertex then represents the type that is pointed to. This approach enables an easier type determination in subsequent steps while retaining pointer information. Fig. 2b depicts the TDG of the function definition `void f(int *a, int b)`. In Section 4 we discuss TDG construction for different data types in detail.

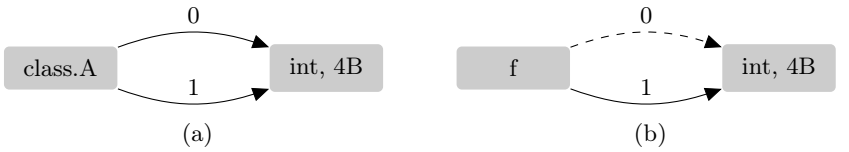

Fig. 2. Example TDGs for two different data type structures. The number at each edge represents the position of the member in the class. Dashed edges represent pointers. a) illustrates the TDG for `class A { int x, y; }`. b) illustrates the TDG for `void f(int *a, int b)`.

3.3 Injection of Size-Functions and Intent Estimation

For each KF argument, we inject an appropriate size-function (SF) and a call to it with LLVM. A SF returns the MF of an argument at run-time. It is generated by traversing the TDG and considering dynamic memory allocations. Section 4 discusses this process in detail.

During run-time all SFs are called and their results are gathered. Their summation occurs in three different sums to distinct between the arguments intent: i) *in*, representing read-only arguments, ii) *out*, representing write-only arguments, and iii) *in-out*, representing read-write arguments.

We used Andersen's algorithm for inter-procedural points-to analysis [1] as basis to determine data directions. In particular, we construct a *data dependency graph* (DDG) containing loads (`LD`) and stores (`ST`) related to the argument.

Kernels with side-effects are considered by tracing function calls and linking correlated arguments. That is, we extend the parent DDG by the one generated from the called function.

For construction, it also is crucial to distinct, whether `LD` and `ST` access an arguments value or only its address. For instance, in LLVM IR, argument addresses are commonly stored at the beginning of a function. These `ST` operations do not belong to the graph, because they do not write a value to the argument.

To determine the data direction, we consider the count of `LD` and `ST` instructions accessing the argument. If only `LD` instructions are recognized, the argument is labeled with *in*. In contrast, it is labeled with *out*, if only `ST` instructions exist. For all other cases, the argument direction is *in-out*.

Once the processing is finished, the altered LLVM IR is compilable to an executable. On execution, SFs are called and their results are gathered as described. The resulting sums can then be further processed, e.g. printed to the console or used for memory allocation.

4 Handling of Data Types

This section discusses how the type dependency graph is constructed for different data types and how a matching SF is derived.

4.1 Scalar Data Types

C / C++ scalar data types, e.g. `int`, utilize a single vertex. The byte-width of the vertex is obtained by calling the LLVM function `getBitWidth` and subsequent division by 8. Alignments are considered if the MF is affected. The SF returns the byte-width of the type.

4.2 Pointers

Pointers are represented by pointer-edges in the TDG. We assume, that the pointed-to value is either a scalar or a pre-allocated (multi-dimensional) array of scalars. The pointer itself does not belong to the MF of the argument. This relies on the assumption, that only the pointed-to object is of interest.

To determine the allocated size, we explicitly search for calls to allocation functions. Currently, only memory allocations by either *malloc* or *new* are supported. A `ST` can be found, if the argument is related to an allocation function. The DDG helps determining this relation. Once a matching allocation call was found, its argument is extracted. At run-time, this argument contains the allocated size in bytes.

If the pointed-to object is a scalar or a one-dimensional array this argument is returned by the SF. However, if a multi-dimensional array was allocated, our tool search for loops surrounding the identified allocation function. We use the LLVM *LoopInfo*[1] pass for this search. Currently, only linear multi-dimensional arrays are fully supported. The SF returns the allocated size in bytes multiplied with the loop count.

4.3 Classes and Structures

In LLVM IR, classes and structures are represented with the LLVM type `struct`, e.g. `class A { int x, y; }` is expressed as `%class.A = { i32, i32 }`. The

[1] `http://llvm.org/docs/doxygen/html/classllvm_1_1LoopInfo.html`

constructed TDG then contains sub-types resembling the structural layout of the class (or struct). Thus, a single vertex is created for the class type and multiple vertices are created for the contained types. Edges between class type and contained types are labeled with indices referring to their location. Fig. 3 provides an example.

```
class Coord {
  int x, y, z;
};

class Value {
  Coord coord;
  double value;
};
```

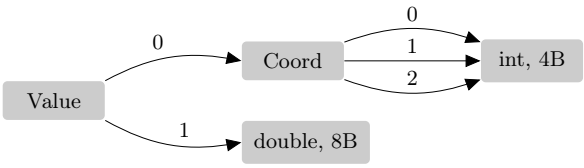

Fig. 3. Example for TDG construction for a small class

The SF of a class gathers the values of the SFs of the contained types. This method is also applicable on hierarchical class structures, classes with inheritance and with templates. These structures are evaluated at compile-time and thus are present in LLVM IR. Inheritance extends the TDG by another class. Types of templates are resolved and placed directly in LLVM IR, thus they are available in the TDG. Structures with recursion, e.g. linked lists, are currently only supported if they are implemented with STL containers. Since the SFs are called at runtime, dynamically typed objects from C++ interfaces are resolved correctly.

4.4 STL Containers

In general, STL containers can be treated like classes. The TDG construction thus follows the rules provided in the previous section. For most cases, type deduction can also be done like described previously. But for e.g. `std::map`, we use the destructor to identify the key / value types since they are obfuscated.

However, SFs use the provided iterators for object traversal. This eases the processing and enables support for the aforementioned recursive structures. The SFs of STL containers call the member `begin` and `end` of an instance and traverses through its elements. For each element, its corresponding size function is called and the returned value is summed. With this technique, element types with likely differing byte-width, e.g. a `vector` of `strings`, are considered. Once all elements were traversed, the SF returns the summed size. In future, our tool will be extended to also support custom iterators.

5 Evaluation

We evaluated our tool-chain with two different test scenarios. Synthetic tests were used to demonstrate the core functionality of the tool-chain. Additionally, we selected real-world applications, each providing different data structures. The automatically estimated MFs and directions were checked against the results of manually inserted computations in our test cases.

Table 1. Abbreviations and KF interfaces for the synthetic test cases (first half) and real-world application test cases (second half)

Name	Kernel Function Interface
NEW	`void f(int *a, Y *y)`
STRVEC	`void f(std::vector<std::string> x)`
NESTTPL	`void f(X<std::string> x)`
MAP	`void f(std::map<int, std::string> x)`
TermFrequency	`void tf(int k, std::string *genome, vector<string> *kmer, vector<int> *count)`
Octree	`void OctreeListToMatrixList(OctreeList *l, params *p)`
HPCCG	`int HPCCG(HPC_Sparse_Matrix *A, const double * const b, double * const x, const int max_iter, const double tolerance, int niters, double normr, double *times)`
miniFE	`void driver(const int global_box[][2], int my_box[][2], ComputeNodeType *compute_node, Parameters *params, YAML_Doc *ydoc)`

5.1 Synthetic Test Cases

Table 1 provides an overview on the synthetic test cases. It shows the name and the kernel function interface of every test case. The test cases do not have a function body, i.e. their data-direction is always marked as *in*. We will not discuss these test cases in detail, since their function arguments are self-explanatory. The custom class of the pointer `*y` is defined as `class Y { int x, y; }`. Implementation details on the custom class `X` are provided in Listing 1.1.

5.2 Real-World Application Test Cases

For the following description, we spare a detailed KF interface discussion and solely focus on noticeable data types. Table 1 additionally provides the complete KF interfaces. *TermFrequency* and *Octree* originate from the ENHANCE project [2], while *HPCCG* and *miniFE* were taken from the Mantevo project [6].

TermFrequency computes the amount of short genomic sequences (mers) contained in a genomic string. `mer` is of type `std::vector<std::starting>` and stores the mers. `count` is a `std::vector<int>` and stores the corresponding count of occurrences. This variable is passed as a pointer with data direction *out*.

```
template<class T>
class X {
public:
  Y<T> y;
};
```

```
template<class T>
class Y {
public:
  std::vector<
    Z<T>
  > z;
};
```

```
template<class T>
class Z {
public:
  std::vector<T> v;
};
```

Listing 1.1. Class layout definition for the synthetic test NESTED

Octree transposes a list of octrees into a list of matrices. Octrees are commonly used in computer graphics, its vertices always have zero or eight children. The function does not return any data but writes the matrices to disk. `params` is a class containing several values of type `double`. `list` is a custom class that inherits from the type `std::vector<Octree>`. `Octree` in turn contains a vector of child nodes, which again are custom classes containing `double` values. In summary, a nested class structure with inheritance has to be processed.

HPCCG calculates the conjugate gradient of a sparse matrix. The most interesting kernel function argument is the sparse matrix `A`, which is stored in CSR format with several dynamically allocated arrays. Furthermore, member allocation is done inside a separate function. Figure 4a depicts the TDG for `A`.

miniFE performs a FEM calculation. It uses YAML serialization to record performance data. We analyzed the class `YAML_Doc`, since it contains recursive structures. It derives from `YAML_Element` which contains in turn a STL vector recursing again to `YAML_Element`, e.g. `std::vector<YAML_Element*>`. Figure 4b depicts the TDG for `YAML_Doc`.

5.3 Evaluation Results

For each test, the evaluation procedure is as follows. First, we compile the test to LLVM IR. Next, we process and modify it with our tool-chain. Then, we compile the result to an executable binary. During execution, the estimated MFs and data directions are printed to the console alongside with the manually computed values.

The overall result is as follows. Synthetic tests and real-world applications revealed, that our implementation behaves correctly for these particular tests. That is, MFs and data directions were all estimated correctly. Additionally, we measured the run-time overhead for the synthetic test cases with the time measurement and analysis tool *elaps'd*[2]. The results are depicted in Table 2. These measurements revealed the following properties.

The run-time overhead for arrays containing constantly-sized elements is approximately constant (*NEW*). For these data types our tool multiplies the base byte-width of the element with the number of allocated elements.

[2] `http://sdressler.github.io/elapsd`

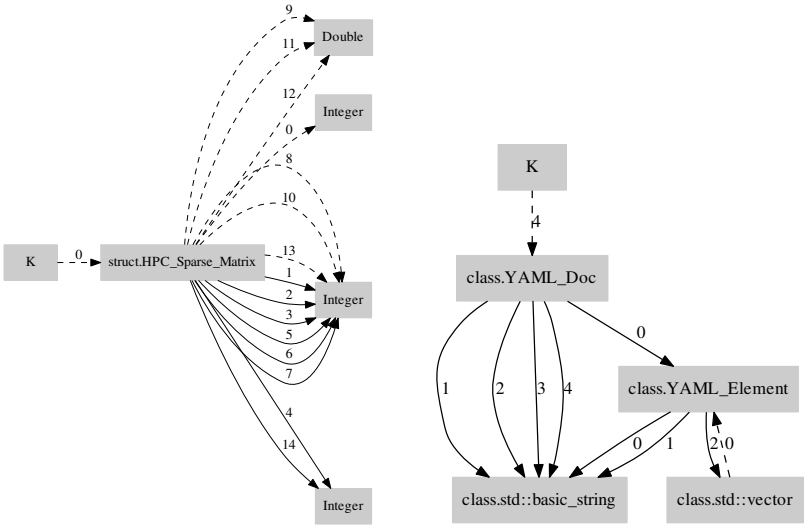

Fig. 4. (a) depicts the TDG of the sparse matrix utilized in HPCCG. (b) shows the TDG of the class `YAML_Doc` used in miniFE. The vertex K refers to the KF, the index on the outgoing edge represents the position of the argument in the KF.

Table 2. Measured overheads of three selected benchmarks with a linearly increasing number of elements. All measured times are in μs.

#Elements	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
NEW	0.92	0.86	0.69	0.76	0.54	0.71	0.68	0.78	0.70	0.63
STRVEC	8.14	12.33	22.78	27.76	35.45	38.24	42.12	47.46	53.35	59.23
MAP	5.81	7.47	9.13	13.45	19.53	22.91	27.84	34.17	43.47	52.51

For arrays containing elements with potentially varying sizes (*STRVEC* and *MAP*), the overhead correlates with the number of elements. This is a result from the need to call every size function. If the element itself contains elements with varying sizes, the effect strengthens, i.e. the overhead increases further. In contrast to *STRVEC*, the overhead for *MAP* must not increase linear, since the STL implementation of `std::map` uses a red-black tree. Consequently, its elements may be spread in memory which causes a memory access slowdown. For *STRVEC*, elements are contiguous in memory.

6 Conclusions and Future Work

We presented an automated approach to estimate the memory footprint of non-linear data objects. By using LLVM and a graph-based approach for information deduction, our workflow retains extensibility. To demonstrate its correctness and functionality, synthetic tests and real-world applications were used. Our evaluation showed, that our tool is able to analyze non-linear data structures like deeply nested classes with recursion.

Our future work will concentrate on extending the approach to improve support for objects not providing iterators. For this, the developer may provide additional object information, e.g. which member indicates the end of a list. Furthermore, we will use the presented method to automatically inject data layout transformations.

Acknowledgments. This is developed as part of the ENHANCE project, funded by German ministry for education and science (BMBF), grant No. 01|H11004G. The reference implementation of our tool is available at: `https://github.com/sdressler/objekt`, the terms of the BSD license apply. We would like to thank Florian Schintke for valuable discussions.

References

1. Andersen, L.O.: Program analysis and specialization for the C programming language. PhD thesis, University of Cophenhagen (1994)
2. ENHANCE Consortium. Enhance - enabling heterogeneous hardware acceleration using novel programming and scheduling models (January 2012)
3. Dhurjati, D., Kowshik, S., Adve, V.: SAFECode: enforcing alias analysis for weakly typed languages. ACM SIGPLAN Notices 41, 144–157 (2006)
4. Dreßler, S., Steinke, T.: A Novel Hybrid Approach to Automatically Determine Kernel Interface Data Volumes. Technical report, ZIB (2012)
5. Han, T.D., Abdelrahman, T.S.: hiCUDA: High-level GPGPU programming. IEEE Transactions on Parallel and Distributed Systems 22(1), 78–90 (2011)
6. Heroux, M.A., Doerfler, D.W., Crozier, P.S., Willenbring, J.M., Edwards, H.C., Williams, A., Rajan, M., Keiter, E.R., Thornquist, H.K., Numrich, R.W.: Improving performance via mini-applications. Sandia National Laboratories, Tech. Rep. (2009)
7. Jablin, T.B., Prabhu, P., Jablin, J.A., Johnson, N.P., Beard, S.R., August, D.I.: Automatic cpu-gpu communication management and optimization. ACM SIGPLAN Notices 46, 142–151 (2011)
8. Lattner, C., et al.: The LLVM compiler infrastructure (2010)
9. Lee, S., Eigenmann, R.: OpenMPC: Extended OpenMP programming and tuning for GPUs. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–11. IEEE Computer Society (2010)
10. Uchiyama, H., Tsumura, T., Matsuo, H.: An Automatic Host and Device Memory Allocation Method for OpenMPC (2012)