Computer Languages, Systems & Structures 47 (2017) 104–129

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

An array content static analysis based on non-contiguous partitions[☆], [☆☆]

Jiangchao Liu[*], Xavier Rival

ENS, CNRS, INRIA, PSL, 45, rue d'Ulm 75230 Paris Cedex 05, France*

ARTICLE INFO

Article history:
Received 27 April 2015
Received in revised form
14 January 2016
Accepted 19 January 2016
Available online 28 January 2016

Keywords:
Array analysis
Abstract domain
Program verification

ABSTRACT

Conventional array partitioning analyses split arrays into contiguous partitions to infer properties of sets of cells. Such analyses cannot group together non-contiguous cells, even when they have similar properties. In this paper, we propose an abstract domain which utilizes semantic properties to split array cells into groups. Cells with similar properties will be packed into groups and abstracted together. Additionally, groups are not necessarily contiguous. This abstract domain allows us to infer complex array invariants in a fully automatic way. Experiments on examples from the Minix 1.1 memory management and a tiny industrial operating system demonstrate the effectiveness of the analysis.

© 2016 Elsevier Ltd. All rights reserved.

1. Introduction

Arrays are ubiquitous, yet their mis-use often causes software defects. Therefore, a large number of works address the automatic verification of array manipulating programs. In particular, partitioning abstractions [12,18,20] split arrays into sets of contiguous groups of cells (also called *segments*), in order to, hopefully, infer that they enjoy similar properties. A traditional example is that of an initialization loop, with the usual invariant that splits the array into an initialized zone (the segment from index 0 to the current index) and an uninitialized region (the segment from the current index to the end of the array).

However, when cells that have similar properties are not contiguous, these approaches cannot infer adequate array partitions. This happens for unsorted arrays of structures, when there is no relation between indexes and cell fields. Sometimes the partitioning of array elements relies on relations among cell fields. This phenomenon can be observed in low-level software, such as operating system services and critical embedded systems drivers, which rely on static array zones instead of dynamically allocated blocks [30]. When cells with similar properties are not contiguous, traditional partition based techniques are unlikely to infer relevant partitions/precise array invariants.

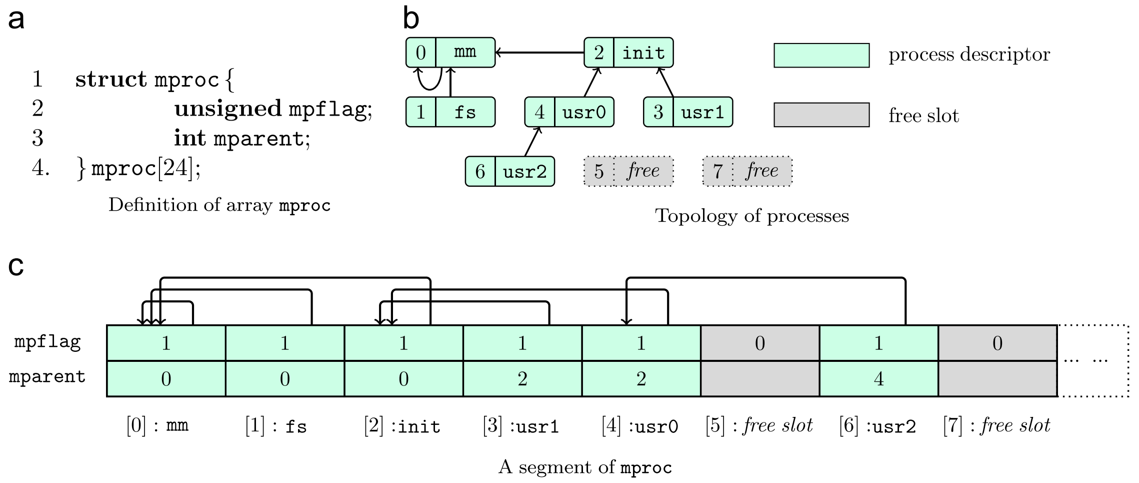
Fig. 1 illustrates the Minix 1.1 Memory Management Process Table (MMPT) main structure. The array of structures `mproc` defined in Fig. 1(a) stores the process descriptors. Each descriptor comprises a field `mparent` that stores the index of the parent process in `mproc`, and a field `mpflag` that stores the process status. Fig. 1(c) depicts the concrete values stored in `mproc` to describe the process topology shown in Fig. 1(b) (the whole `mproc` table consists of 24 slots, here we show only 8,

[☆] This document is a collaborative effort.

[☆☆] The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013), Grant agreement 278673, Project MemCAD, and from the ARTEMIS Joint Undertaking no. 269335 (see Article II.9 of the JU Grant agreement).

[*] Corresponding author.
E-mail addresses: jliu@di.ens.fr (J. Liu), rival@di.ens.fr (X. Rival).

http://dx.doi.org/10.1016/j.cl.2016.01.005
1477-8424/© 2016 Elsevier Ltd. All rights reserved.



```
struct mproc {
        unsigned mpflag;
        int mparent;
} mproc[24];
```

	[0] : mm	[1] : fs	[2] :init	[3] :usr1	[4] :usr0	[5] : *free slot*	[6] :usr2	[7] : *free slot*	
mpflag	1	1	1	1	1	0	1	0	... ...
mparent	0	0	0	2	2		4		

Fig. 1. Minix 1.1 memory management process table (MMPT) structure.

for the sake of space). An element of mproc is a process descriptor when its field mpflag is strictly positive and a free slot if it is null. Minix 1.1 uses the three initial elements of mproc to store the descriptors of the memory management service, the file system service and the init process. Descriptors of other processes appear in a random order. In the example of Fig. 1, init has two children whose descriptors are in mproc[3] and mproc[4]; similarly, the process corresponding to mproc[4] has a single child the descriptor of which is in mproc[6]. Moreover, Minix assumes a parent–child relation between mm and fs, as mm has index 0 and the parent field of fs stores 0. To abstract the process table state, valid process descriptors and free slots should be partitioned into *different groups*.

Traditional, contiguous partitioning cannot achieve this for two reasons: (1) the order of process descriptors in mproc cannot be predicted, hence is random in practice, and (2) there is no simple description of the boundaries between these regions (or even their sizes) in the program state. The symbolic abstract domain by Dillig, Dillig and Aiken [15] also fails here as it cannot attach arbitrary abstract properties to summarized cells.

In this paper, we set up an abstract domain to partition the array into non-contiguous groups for process descriptors and free slots so as to infer this partitioning and precise invariants (Section 2) automatically. Our contributions are:

- An abstract domain that partitions array elements according to semantic properties, and can represent non contiguous partitions (Section 4).
- Static analysis algorithms for the computation of abstract post-conditions (Sections 5 and 6), widening and inclusion check (Section 7).
- The implementation and the evaluation of the analysis on the inference of tricky invariants in excerpts of some operating systems (e.g. Minix 1.1) and other challenging array examples (Sections 8 and 9).

2. Overview

Minix is a Unix-like multitasking computer operating system [31]. It is a very small OS (with fewer than 10 000 lines of kernel), yet it greatly influenced the design of other kernels, including Linux. It is based on a micro-kernel architecture, with separate, lightweight *services* respectively in charge of *task scheduling* (in kernel), *memory management* and *file system*. Each service maintains a *process table* that describes the processes currently running. The tables of distinct services are consistent with each other. In the following, we consider the process table of the memory management service, which is very similar to that of the other services (and is quite representative of process table structures in operating systems kernels). This process table consists of an array mproc that stores the memory management information for each process in a distinct slot. As in all Unix operating systems, processes form a reversed tree, where each process has a reference to its parent (which created it) and is referred to by its children (which it created).

New processes can be created by the system call fork from a parent process. A process exits after it calls exit and its parent calls wait. These two system calls form a synchronization barrier and the process and its parent are set to be "hanging" and "waiting" respectively when they reach the barrier first. In Fig. 1(b), the process described by mproc[4] would be "hanging" after it calls exit if mproc[2] is not "waiting", and after mproc[2] calls wait, mproc[4] will exit. System calls fork, wait and exit are first handled by *memory management* and then passed on to *task scheduling* and *file system* if needed. One function and three system calls perform atomic changes on the process table structure:

- function mm_init is called when the operating system is initialized and constructs slots in mproc for the first three system level processes;