A Scalable Memory Model for Low-Level Code *

Zvonimir Rakamarić and Alan J. Hu

Department of Computer Science, University of British Columbia, Canada
{zrakamar,ajh}@cs.ubc.ca

Abstract. Because of its critical importance underlying all other software, low-level system software is among the most important targets for formal verification. Low-level systems software must sometimes make type-unsafe memory accesses, but because of the vast size of available heap memory in today's computer systems, faithfully representing each memory allocation and access does not scale when analyzing large programs. Instead, verification tools rely on abstract memory models to represent the program heap. This paper reports on two related investigations to develop an accurate (i.e., providing a useful level of soundness and precision) and scalable memory model: First, we compare a recently introduced memory model, specifically designed to more accurately model low-level memory accesses in systems code, to an older, widely adopted memory model. Unfortunately, we find that the newer memory model scales poorly compared to the earlier, less accurate model. Next, we investigate how to improve the soundness of the less accurate model. A direct approach is to add assertions to the code that each memory access does not break the assumptions of the memory model, but this causes verification complexity to blow-up. Instead, we develop a novel, extremely lightweight static analysis that quickly and conservatively guarantees that most memory accesses safely respect the assumptions of the memory model, thereby eliminating almost all of these extra type-checking assertions. Furthermore, this analysis allows us to create automatically memory models that flexibly use the more scalable memory model for most of memory, but resorting to a more accurate model for memory accesses that might need it.

1 Introduction

Because of its critical importance underlying all other software, low-level system software is among the most important targets for formal verification. For example, the correct execution of even the most mundane software relies on a vast array of supporting system software: the compiler and linker during development, of course, but also all the OS services at runtime: application-level memory management and the underlying virtual memory system, context swaps and the underlying OS scheduler, device drivers for all I/O, etc. With the emergence of virtualization, the hypervisor becomes an even lower-level, even more critical layer that needs verification (e.g., [27]), as even the operating system relies on its correctness.

All formal software analysis must model memory in some way. At one extreme, the entire memory space could be modeled as a single, giant array of bytes/words (e.g., [14,

* This work was supported by a research grant from the Natural Sciences and Engineering Research Council of Canada and a University of British Columbia Graduate Fellowship.

11, 10], early versions of VCC [27] also supported byte-level reasoning). Doing so makes the verification completely accurate (sound and precise with respect to the effect of any memory access), but does not scale beyond very small segments of code. At the other extreme, we can restrict our analysis to handle only code that has no dynamic memory allocation and is completely type-safe (e.g., [6])[1]. Such an approach has scaled to millions of lines of code [6], but obviously precludes verification of typical mainstream software. Most software verification tools (e.g., [2, 20, 21, 8, 18]) try to strike a balance, assuming some degree of type-safety, e.g., assuming that pointers to different types of objects do not alias. Note that most tools do not check these assumptions — if the code violates the assumption, the tool might report wrong answers without any warning.

The choice of memory model is particularly challenging for low-level systems software, because such software must sometimes make type-unsafe memory accesses. For example, common idioms include casting a data structure from/into an array of bytes or integers for efficiency or to interface to hardware, and accessing a structure via differently-typed pointers as a way to implement sub-typing in C. Address arithmetic is also common, usually to offset before or after a given pointer in order to access a nearby data field. Verification tools for low-level software must find an intermediate memory model that assumes some type information to provide scalability, yet accurately captures the effects of lower-level, type-unsafe memory accesses.

In this paper, we develop such a model. The paper consists of two separate, but related parts. In the first part (Section 2), we compare a recently introduced memory model, specifically designed to more accurately model low-level memory accesses in systems code, to an older, widely adopted memory model. We find that the newer memory model scales poorly compared to the earlier, less accurate model. In the second part (Section 3), we investigate how to improve the soundness of the less accurate model. We first consider adding assertions to the code that each memory access does not break the assumptions of the memory model, but this causes verification complexity to blow-up. Then, we develop a novel, extremely lightweight static analysis that quickly and conservatively guarantees that most memory accesses safely respect the assumptions of the memory model, thereby eliminating almost all of these extra type-checking assertions. Furthermore, this analysis allows us to create automatically memory models that flexibly use the more scalable memory model for most of memory, but resort to a more accurate model for memory accesses that might need it. Experimental results show that the static analysis is very fast, maintaining the scalability of the less accurate memory model. Along the way, our tool found four bugs in real Linux device drivers, three of which were previously unreported.

2 Comparing Two Memory Models

Because of the vast size of available memory in today's computer systems, faithfully representing each memory allocation and access in a static verifier does not scale. Therefore, verification tools rely on memory models that trade precision for scalability,

[1] Astrée now supports type casts, but still does not support dynamically allocated memory [24].

and in turn, they define programming language operational semantics with respect to the chosen memory model. In this section, we introduce two memory models that are typically used in modular deductive verification tools, describe their advantages and drawbacks in the context of low-level code verification, and present empirical results on using the models to verify a number of Linux device drivers.

2.1 Monolithic Memory Model

Our first memory model is heavily influenced by the one used in early versions of HAVOC [9], and also similar to the one used in the first incarnation of VCC [27]. The main idea behind this memory model is to divide the memory into disjoint objects (or regions). Each object is identified by its reference, and has a fixed size determined when the object is allocated. A pointer in the memory model is therefore a pair, consisting of a reference and an offset; the reference uniquely defines the object into which the pointer points; the byte offset defines the byte in the object being pointed to.

To be able to translate a program into a representation that uses a memory model, we have to define the semantics of its source language with respect to the chosen memory model. In the monolithic memory model, the semantics of programs depends on three fundamental types: the uninterpreted type `ref` of object references, the type `int` of integers, and the type $\texttt{ptr} = \texttt{ref} \times \texttt{int}$ of pointers. For notational convenience, each variable in a program, regardless of its declared type, contains a pointer value: a pointer is a pair containing an object reference and an integer offset, and an integer value is encoded as a pointer value whose first component is the special constant `null` of type `ref`. Note that because of the integer offset component, the memory model can precisely capture byte offsets and low-level pointer arithmetic inside an object. On the other hand, since object references are uninterpreted, the objects are essentially "infinitely apart", and the memory model cannot model pointer arithmetic between objects.

The heap of a program is modeled using two map variables, `Mem` and `Alloc`, and a map constant `Size`:

$$\begin{aligned} \texttt{Mem} &: \texttt{ptr} \to \texttt{ptr} \\ \texttt{Alloc} &: \texttt{ref} \to \{\texttt{UNALLOCATED}, \texttt{ALLOCATED}\} \\ \texttt{Size} &: \texttt{ref} \to \texttt{int} \end{aligned}$$

The variable `Mem` maps pointers to pointers and represents the contents of memory at a location. The variable `Alloc` maps object references to the set {`UNALLOCATED`, `ALLOCATED`} and is used to model memory allocation. The constant `Size` maps object references to positive integers and represents the size of the object. For instance, the procedure call `malloc(n)` for allocating a memory buffer of size `n` returns a pointer $\texttt{Ptr}(o,0)$ where o is an object reference such that $\texttt{Alloc}[o] = \texttt{UNALLOCATED}$ and $\texttt{Size}[o] \geq n$ before the call, and $\texttt{Alloc}[o] = \texttt{ALLOCATED}$ after the call (ignoring the possibility of memory allocation failure, which could also be easily modeled).

2.2 Burstall's Memory Model

Our second memory model is a type-indexed memory model (also known as Burstall's memory model [7]) that has been commonly used in the deductive verification of type-safe languages [5, 19]. The main idea behind this model is that, apart from dividing

memory into disjoint objects as in the previous model, we also split the memory according to a set of possible *types* of memory locations. To achieve this splitting, a set of unique type constants of type `type` is introduced, which represent types in the original program. The common types found in a language, such as `int`, `int*`, `char`, etc., are going to be translated as type constants `$int`, `$intP`, `$char`, etc. Usually, apart from all of the commonly found types, the set of type constants also contains a unique type constant for each structure field. For instance, the structure

```
struct {
  int x;
  int y;
} foo;
```

introduces unique type constants `$foo#x` and `$foo#y`. It turns out that this "type-awareness" in the model, caused by adding type constants and splitting the memory according to those, is exactly what gives this model an edge when it comes to scalability over the monolithic model.

Our map `Mem` from the previous memory model is therefore, instead of mapping pointers to pointers, going to map type-pointer pairs to pointers. We also introduce in the model an additional map constant `Type` that maps pointers (memory locations) to types and represents the allocation type of memory locations. Each type in the memory model is a unique constant distinct from all other types. The type-indexed memory model therefore has four maps:

$$\begin{aligned} \texttt{Mem} &: (\texttt{type} \times \texttt{ptr}) \rightarrow \texttt{ptr} \\ \texttt{Alloc} &: \texttt{ref} \rightarrow \{\texttt{UNALLOCATED, ALLOCATED}\} \\ \texttt{Size} &: \texttt{ref} \rightarrow \texttt{int} \\ \texttt{Type} &: \texttt{ptr} \rightarrow \texttt{type} \end{aligned}$$

Adding types to the memory model makes proving programs easier and faster:

- One can conclude that updates to different fields of a structure don't influence each other without reasoning about integer offsets and pointer arithmetic, as would be needed in the monolithic memory model. Such reasoning is often hard in the presence of quantifiers.
- Memory locations of different fields of two distinct objects usually don't alias, which is nicely captured by this memory model. This also greatly simplifies the task of proving many interesting assertions.
- When a field is being updated, based on its type, only the corresponding submap of `Mem` changes, which simplifies proving frame axioms.

2.3 Experimental Results

We have implemented the preceding memory models as part of our tool SMACK (Static Modular Assertion ChecKer [26]), which is a modular, annotation-based, extended static property checker of C programs. In the spirit of modular verification, SMACK verifies programs annotated with procedure specifications and loop invariants. It uses the

Driver	LOC	Memory Model		Speedup
		Monolithic (s)	Burstall (s)	
ib700wd	346	45.7	14.6	3.1
w83877f_wdt	421	59.5	16.1	3.7
sc520_wdt	443	50.2	16.5	3.0
machzwd	494	71.0	18.1	3.9
wdt977	519	46.4	19.3	2.4
ds1286	633	70.8	20.3	3.5
efirtc	815	62.2	16.3	3.8
applicom	934	*3368.8	161.2	20.9

Table 1. Running times for checking correct locking behavior in device drivers from the Linux kernel. The column "LOC" given the number of lines of code; "Monolithic" gives the total running time of BOOGIE using the monolithic memory model; "Burstall" gives the total running time of BOOGIE using Burstall's memory model with assumed types; "Speedup" compares the running times. The * indicates that BOOGIE timed out on two procedures from the applicom driver (time out is set to 1200s).

LLVM compiler framework [22] to parse input programs and annotations. The LLVM output is translated by SMACK into a BoogiePL [16] program based on the operational semantics of C memory accesses according to the selected memory model. BoogiePL is the input language of the BOOGIE verifier [3], which, in turn, generates a verification condition (VC) from the input program whose validity implies partial correctness of the input. The VC generation in BOOGIE is performed using a variation [4] of the standard *weakest precondition* transformer [17]. We check the generated VC using the accompanying Z3 theorem prover [15]. We report only the running times of BOOGIE required to verify the examples since the transformation SMACK performs takes only a small fraction of that time.

We applied SMACK to check correct locking behavior of several device drivers from the Linux kernel. The source code of the examples, the models and stubs of the relevant kernel routines, and the test harness are taken from the DDVERIFY suite [29, 1]. Ensuring correct locking behavior amounts to checking that locks are initialized before they are used and that locks are alternately acquired and released. Table 1 lists the drivers and gives the running times for the verification using the monolithic and Burstall's memory models. All experiments were executed on an Intel Pentium D at 2.8Ghz.

Seven of the drivers were arbitrarily picked character device drivers that contain spinlocks, usually as one or two global variables. In addition, we handpicked the applicom driver, since this driver has a global array of structures where each structure is protected by its own spinlock. This makes it much more interesting and challenging to verify (see Figure 1), requiring from a tool the ability to reason precisely about such unbounded data structures. Current tools that are typically used in the verification of device drivers [2, 20, 21, 8, 11, 10] have trouble handling unbounded data structures. One of the goals of SMACK is to address that weakness.

From the running times, it can be seen that Burstall's memory model is the clear winner. It always outperforms the monolithic memory model on easier examples, and the speedup factor is from 2.4 to 3.9. Furthermore, using Burstall's memory model, we

```
1 struct applicom_board {
2   unsigned long PhysIO;
3   void __iomem *RamIO;
4   wait_queue_head_t FlagSleepSend;
5   long irq;
6   spinlock_t mutex;
7 } apbs[MAX_BOARD];
8
9 irqreturn_t ac_interrupt(int vec, void *dev_instance) {
10  for (i = 0; i < MAX_BOARD; i++) {
11    if (!apbs[i].RamIO) continue;
12    spin_lock(&apbs[i].mutex);
13    if(readb(apbs[i].RamIO + RAM_IT_TO_PC)) {
14      spin_unlock(&apbs[i].mutex);
15      i--;
16    } else {
17      spin_unlock(&apbs[i].mutex);
18    }
19  }
```

Fig. 1. Simplified code excerpt from the applicom Linux device driver illustrating the complexity of checking correct locking behavior. The loop on line 10 iterates over array elements. If the field `RamIO` of the element at index `i` is not null (line 11), the lock (field `mutex`) is acquired on line 12 and then later released. The verification requires checking complex invariants over all elements of the array (i.e. quantified) that involve values of the `RamIO` fields as well as the status of locks (initialized, locked, unlocked).

managed to verify the applicom example, which we couldn't do using the monolithic memory model since it timed out on two procedures. The example requires proving complex quantified invariants over fields from an array of structures. The key to successful verification of this example is structure field disambiguation: Burstall's memory model provides this for free, whereas in the monolithic model, it requires reasoning about offsets and pointer arithmetic.

However, the much better running times of Burstall's memory model come at a price: it relies on the assumption that memory is strongly typed. In the examples, when we use Burstall's model, we are assuming the type of a memory location before each memory access, which is unsound and can cause bugs to be missed in a type-unsafe setting such as C. In the next section, we describe how to deal with this problem.

3 Ensuring Soundness with Burstall's Memory Model

Burstall's memory model relies on the assumption that memory is strongly typed, as in type-safe languages such as Java. That means that a type of the object is established when it is created, via a call to `new`, and the object is always accessed using that original type. However, low-level languages like C allow reinterpretation of the original type and therefore type-unsafe memory accesses. Such operations are not uncommon in systems

```
1 typedef struct {
2   int x;
3 } S1;
4
5 typedef struct {
6   int a;
7   int b;
8 } S2;
9
10 void main() {
11   S2* s2 =
12     (S2*)malloc(sizeof(S2));
13   S1* s1 = (S1*)s2;
14
15   s2->a = 3;
16   s1->x = 4;
17
18   assert(s2->a == 3);
19 }
```

```
1 const unique $S1#x:type;
2 const unique $S2#a:type;
3 const unique $S2#b:type;
4
5 procedure main() {
6   var s1:ptr, s2:ptr;
7   call s2 := malloc(Ptr(null,8));
8   s1 := s2;
9
10  Mem[$S2#a,s2] := Ptr(null,3);
11  Mem[$S1#x,s1] := Ptr(null,4);
12
13  assert(Mem[$S2#a,s2] ==
14    Ptr(null,3));
15 }
```

Fig. 2. Example illustrating a simple upcasting in C that causes unsoundness in Burstall's memory model. The right column shows simplified BoogiePL code of the translation of the function `main`, assuming Burstall's model. Because of the assumption of type safety, the two assignments on BoogiePL lines 10 and 11 do not alias, resulting in the assertion incorrectly passing.

code and are typically done in C using casts or unions[2]. Often, casts don't reinterpret memory at the byte level, but are used to simulate object-oriented language features, such as inheritance, that are not supported directly in C. In fact, according to empirical studies [28, 13], more than 90% of the structure casts in C fall into that category.

Figure 2 gives a simple example illustrating "upcasting" in C. The structure `S2` is a subtype of the structure `S1`, and the cast on line 13 represents an upcast. The example shows how such a simple cast can cause Burstall's memory model to become unsound: the field update on line 16 overwrites the value that was written to the same memory location on line 15, and the assertion on line 18 fails. However, in Burstall's model this overwrite does not happen, since different field names (i.e. different unique types) denote different memory locations in the model: the write to `s2->a` is translated as the write to `Mem[$S2#a,s2]` on line 10 of the BoogiePL translation in the right column, while the write to `s1->x` is translated at the write to `Mem[$S1#x,s1]` on line 11, and doesn't overwrite the location `Mem[$S2#a,s2]` although the pointers `s1` and `s2` are equal.

A simple way of ensuring soundness in the presence of such casts is to syntactically analyze the source code and just give up on the verification if we find one (e.g., [18]). Our goal is to go a step further and verify the code even in the presence of type-unsafe structure casts, while preserving soundness. In the following sections, we'll describe three different techniques of how to achieve that goal.

[2] We can consider union a special case of cast.

```
1 const unique $S1#x:type;
2 const unique $S2#a:type;
3 const unique $S2#b:type;
4
5 procedure main() {
6   var s1:ptr, s2:ptr;
7   call s2 := malloc(Ptr(null,8));
8   assume(Type[s2] == $S2#a && Type[s2 + 4] == $S2#b);
9   s1 := s2;
10
11  assert(Type[s2] == $S2#a);
12  Mem[$S2#a,s2] := Ptr(null,3);
13  assert(Type[s1] == $S1#x); // Fails!
14  Mem[$S1#x,s1] := Ptr(null,4);
15
16  assert(Type[s2] == $S2#a);
17  assert(Mem[$S2#a,s2] == Ptr(null,3));
18 }
```

Fig. 3. Translation of the example from Fig. 2 with type-check assertions added before each memory access (lines 11, 13, and 16). The type-check assertion on line 13 will fail, indicating a violation of the assumption of type safety.

3.1 Guarding Memory Accesses with Type Assertions

A straightforward way of preventing unsoundness described in the previous section from happening in Burstall's memory model is to add *type checks* before each memory access. The checks are added in the form of assertions on the `Type` map. Every access to a memory location x with type $\$t$ is going to be preceeded with the assertion $assert(\texttt{Type}(x) == \$t)$ that will have to be discharged.

Figure 3 shows the translation of the example in Figure 2 with the inserted type checks. The map `Type` represents the compile-time allocation type of memory locations, and therefore the correct allocation type has to be assumed on line 8 after the allocation. Then, type check assertions are inserted before each memory access (lines 11, 13, and 16). The type check assertion on line 13 clearly will fail: $\texttt{s1} = \texttt{s2}$, and the type of `s2` is `$S2#a`, not `$S1#x`. Whenever a memory location is accessed through a type that is not the allocation type of the memory location, the added type check assertion will fail. This preserves the soundness of the verification in Burstall's model.

However, proving such type check assertions for each memory access in the program is a big overhead, as we'll show later on in the experiments in Section 3.4. Furthermore, discharging those assertions often requires adding more manual annotations to the code which poses an additional burden on the user. Both of these drawbacks are an unacceptable burden that is not justified since most parts of the code usually obey the type restrictions imposed by Burstall's memory model. Therefore, in the next section, we introduce a lightweight static analysis that eagerly removes most of the required type-check assertions by conservatively guaranteeing that those memory accesses safely respect the assumptions of the model.

3.2 Eagerly Eliminating Type Check Assertions

We'll start this section by giving some background information on the pointer analysis that is the starting point of our technique for eagerly eliminating type check assertions. Then, we'll describe our algorithm for eliminating type checks.

Data Structure Analysis (DSA). DSA [23] is a highly scalable and fast, context-sensitive (with full *heap cloning*), field-sensitive (even in a type-unsafe setting), conservative pointer analysis. The term "heap cloning" refers to a property important for achieving true context-sensitivity — heap objects are not distinguished just by allocation site, but also by (acyclic) call paths leading to their allocation, i.e. the calling context in which they were created. Support for data structure operations is often going to be encapsulated in a library used throughout the code, and therefore context-sensitivity is important to be able to handle such cases precisely.

DSA constructs a representation of the heap in the form of Data Structure Graphs (DS graphs); it creates one DS graph per procedure plus an additional one for global storage. The separate globals graph is a key optimization allowing procedure graphs to contain only the parts of global storage reachable from that procedure. A DS graph consists of a set of nodes (DS nodes) and a set of edges. As an example, a simplified part of the globals DS graph for the applicom device driver is shown in Figure 4. We distinguish two types of DS nodes: heap nodes with a number of fields at different offsets (e.g. rectangle nodes in the example graph), and pointer variable nodes that point into heap nodes (e.g. oval nodes in the example graph). A pointer variable node is named after the pointer variable it represents and has one edge. A heap node has one outgoing edge per pointer field. Each heap node has a type and represents a potentially unbounded number of objects in memory of that type. A DS graph edge is defined by its source node and offset (i.e. offset of the respective pointer field in the source node), and its end node and offset. For instance, if the word size is 4 bytes, the second edge coming out of the *genhd_registered* node is defined by $\langle genhd\_registered, 8\rangle \rightarrow \langle block\_device, 0\rangle$.

Instead of just providing the usual pairs of references that may alias (points-to/alias information), the explicit heap representation DSA constructs can be used to identify different instances of data structures and provide structural and type information for each identified instance. The key feature of DSA we take advantage of is the conservative type information for each heap object. In particular, if all accesses to objects that a node represents obey a consistent type, such node is called "type-homogeneous". Accesses are defined as operations on pointers that point into the node and actually interpret the type: load and store operations, and structure and array indexing operations on pointers. Operations such as memory allocation and pointer casts (e.g. from `void*`) are not counted as accesses and don't influence a node type. If accesses with incompatible types are found, the type of the node is marked as *Unknown*. Therefore, DSA tracks types precisely in the type-safe parts of the heap/program, while in the presence of type-unsafe operations it conservatively treats nodes as having an unknown type.

Eager Type Check Elimination Algorithm. The algorithm is relatively simple and straightforward, but as we'll show in the experiments in Section 3.4, extremely effective. First, we run the DSA on the code we are analyzing, outputting a DS graph for each procedure and the globals graph. Then, for each memory read or write through a pointer, we find the type of the memory location it points to using the appropriate DS

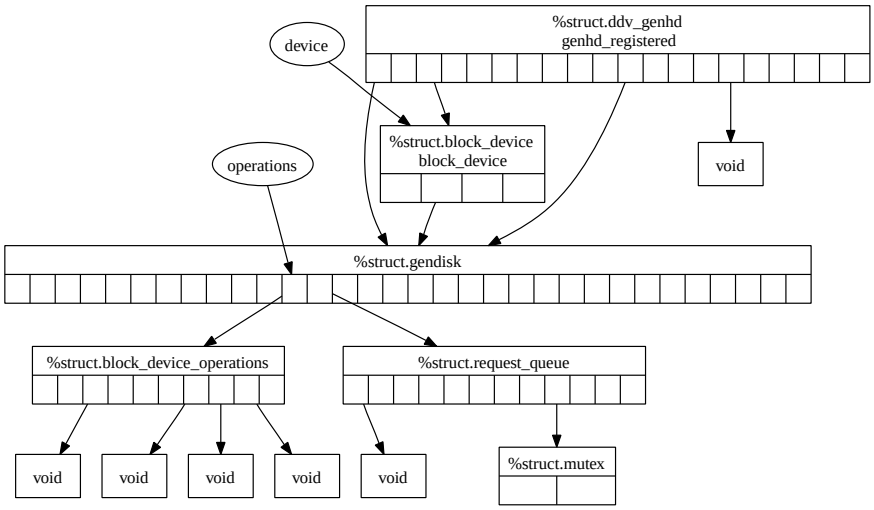

Fig. 4. An example of a Data Structure Graph. The graph shows a simplified part of the globals DS graph for the applicom device driver. Oval nodes in the graph are pointer variable nodes (e.g. *device* and *operations*); rectangle nodes are heap nodes (e.g. *genhd_registered* and *block_device*). Each heap node has a type. For instance, the type of the *genhd_registered* node is *struct.ddv_genhd*, the type of the *block_device* node is *struct.block_device*, etc. Pointer fields of heap nodes have outgoing edges, while fields of other types are just empty boxes.

graph. If the computed type is the same as the actual type of the pointer, we omit the type check (assertion) that would be otherwise generated. If the types are not the same or if the type of the node the pointer points to is *Unknown*, we will generate the type check assertion to preserve soundness.

Figure 5 illustrates the benefits of our technique, removing two type-check assertions compared to the code in Figure 3. However, the soundness is preserved, since the assertion on line 12 couldn't be safely eliminated and is going to fail again: According to DSA, pointer `s1` is going to point to the field `a` of structure `S2`, and therefore its type is going to be `$S2#a` and not `$S1#x` as expected by the memory access.

The algorithm essentially compares compile-time pointer types used by Burstall's memory model with the sound over-approximation of the run-time types that DSA generates: if the two agree, we can safely omit the type check; if not, which could happen either because of actual type-unsafe casts or because of the imprecision of DSA, the type check stays. To sum up, using the extremely fast, cheap, and yet relatively precise Data Structure Analysis, we are eagerly getting rid of most of the type checks that are usually hard and expensive to prove later on.

In order for the remaining assertions to be discharged, either the user has to provide additional manual annotations that will essentially unify the types, which is the approach taken in some related work [25, 12], or such types can be unified automatically, which is our approach described in the next section.

```
1 const unique $S1#x:type;
2 const unique $S2#a:type;
3 const unique $S2#b:type;
4
5 procedure main() {
6   var s1:ptr, s2:ptr;
7   call s2 := malloc(Ptr(null,8));
8   assume(Type[s2] == $S2#a && Type[s2 + 4] == $S2#b);
9   s1 := s2;
10
11  Mem[$S2#a,s2] := Ptr(null,3);
12  assert(Type[s1] == $S1#x); // Fails!
13  Mem[$S1#x,s1] := Ptr(null,4);
14
15  assert(Mem[$S2#a,s2] == Ptr(null,3));
16 }
```

Fig. 5. Translation of Fig. 2 using the eager type check elimination algorithm. Compared to Fig. 3, the unneeded type checks have been eliminated, but the type-safety violation will still be caught.

3.3 Eager Type Unification

The type check elimination algorithm from the previous section doesn't remove the type check assertion for which the compile-time type of a pointer and the one computed by DSA don't agree. Proving those left-over assertions might still require the addition of manual annotations by a user. Instead, we describe a simple, completely automatic technique that will soundly remove the left-over assertions.

For each memory access for which the type check elimination algorithm couldn't agree on types, we unify the two types. Unification simply means that the type constants are not unique anymore, which is in BoogiePL achieved by removing the keyword `unique`. There is an obvious tradeoff between the type check elimination algorithm and the type unification algorithm: the first one might require additional running time and manual annotations from a user to discharge the left-over assertions; the second one is completely automatic, but with each unification, the memory model is closer to the monolithic one and the performance might suffer (in the worst case, all types are unified and we essentially have the monolithic model).

Figure 6 shows the translation using the eager type unification algorithm. Instead of the type-check assertion on line 12 in Figure 5, the types `$S1#x` and `$S2#a` are unified and are not unique constants any more (lines 1 and 2). Now, `Mem[$S2#a,s2]` and `Mem[$S1#x,s1]` possibly refer to the same location, which is sound, and therefore the assertion on line 14 will fail. Note that only the types `$S1#x` and `$S2#a` involved in the actual type-unsafe access got unified, while the type `$S2#b` not involved in type-unsafe operations didn't. Therefore, the overapproximation caused by unification is localized only to the places that actually need it in order to preserve soundness. In the limit, eager type unification degenerates into the monolithic memory model, but for

```
1 const $S1#x:type;
2 const $S2#a:type;
3 const unique $S2#b:type;
4
5 procedure main() {
6   var s1:ptr, s2:ptr;
7   call s2 := malloc(Ptr(null,8));
8   assume(Type[s2] == $S2#a && Type[s2 + 4] == $S2#b);
9   s1 := s2;
10
11  Mem[$S2#a,s2] := Ptr(null,3);
12  Mem[$S1#x,s1] := Ptr(null,4);
13
14  assert(Mem[$S2#a,s2] == Ptr(null,3));
15 }
```

Fig. 6. Translation of Fig. 2 using the eager type unification algorithm. Instead of flagging the type-safety violation, this translation handles type unsafety by allowing `$S1#x` and `$S2#a` to be possibly the same type. Thus, the verifier will correctly catch the assertion violation on line 14.

code that is mostly type-safe, it should have most of the efficiency of Burstall's model and the soundness of the monolithic model.

3.4 Experimental Results

The results in Table 2 compare the running times for checking correct locking behavior while ensuring soundness using the three different approaches: guarding memory accesses with type assertions, eagerly eliminating type check assertions, and eagerly unifying types. The algorithm that inserts type checks for each memory access is a simple linear scan of the code and is extremely fast. Also, DSA scales to hundreds of thousands of lines of code in less than 4s [23]. Therefore, total running times are dominated by the verification done by BOOGIE, and those are the times we report.

As expected, blindly generating type check assertions for each memory access simply does not scale — verification times after using both eager techniques are roughly 30-40 times faster. Furthermore, both eager techniques give roughly the same verification times afterwards. The reason is, to our surprise, that none of the analyzed device drivers actually has type-unsafe structure casts. Therefore, both of the algorithms end up generating the same BoogiePL code. In the future, as we verify increasingly complex examples, we will be able to evaluate the trade-off between the two methods.

Bugs Found. While doing the experiments, we found a total of four bugs in the eight device drivers we checked from the Linux kernel. One bug is the rediscovered incorrect locking pattern in the `ds1286` driver that was also found earlier by the DDVERIFY checker. The other three bugs are previously unreported buffer-overflow bugs. We submitted the bugs to the Linux kernel development team, who confirmed all three bugs and issued patches to the standard Linux kernel.

Driver	Assuring Soundness			Speedup	Speedup
	Every Access (s)	Eager Elimination (s)	Eager Unification (s)	EA/EE	EA/EU
ib700wd	448.7	14.2	14.1	31.6	31.8
w83877f_wdt	683.5	15.3	15.2	44.7	45.0
sc520_wdt	632.5	16.7	16.0	37.9	39.7
machzwd	761.4	18.2	17.8	41.8	42.8
wdt977	466.2	18.1	18.3	25.8	25.5
ds1286	823.5	20.7	25.5	39.8	32.4
efirtc	576.2	15.5	15.3	37.2	37.7
applicom	*7487.4	173.5	172.0	43.2	43.5

Table 2. Total running times for checking correct locking behavior while ensuring soundness in Linux device drivers. The column "Every Access" gives the total running time of BOOGIE when checking type assertions on every access; "Eager Elimination" gives the total running time of BOOGIE when our eager elimination technique is used to soundly remove most of the required type checks; "Eager Unification" gives the total running time of BOOGIE when our eager unification technique is used to ensure soundness; "Speedup EA/EE" compares the running times of Every Access vs Eager Elimination; "Speedup EA/EU" compares the running times of Every Access vs Eager Unification. The * indicates that BOOGIE timed out on four and the memory blew up on one procedure from the applicom driver (time out is set to 1200s).

A natural question is how the different memory models affected the detectability of these bugs. The answer is not straightforward:

- First, as mentioned above, these device drivers turned out to be type-safe, in the sense that Burstall's model would be as accurate as the monolithic model. Thus, one might argue that the more accurate models are unnecessary. However, the type-safety is not at all obvious — this is C code, with type casts, pointer arithmetic, etc. With Burstall's model, we assume type safety and might catch some bugs, but we don't know whether the code is truly type-safe; with the monolithic model, we don't assume type safety, but the verification complexity blows up, so we can't catch any bugs anyway. Our new models ensure type safety but also scale well.
- The other issue is that, of the four bugs we found, only the previously discovered one was a direct violation of the locking-unlocking properties we were checking. The other three bugs were buffer-overflow bugs that were caught because of the type-checking assertions. These bugs perhaps could have been caught by a variety of methods, using many different memory models.

The key point is that our new memory models can ensure, rather than assume, type-safety, yet are scalable enough to handle real code that is sufficiently complex to contain significant bugs that have eluded previous detection.

4 Conclusion and Future Work

In the first part of the paper, we presented our experience with two memory models for low-level code. We introduced the monolithic memory model, which can handle soundly many common low-level idioms. Then, we presented Burstall's memory model,

which has typically been used in the verification of type-safe languages. We implemented both models as part of our verification tool SMACK. In the experiments, we checked correct locking behavior of a number of Linux device drivers, and showed that the performance using Burstall's model is much better than using the monolithic memory model, especially on more complex examples.

However, a straightforward translation of a program using Burstall's memory model cannot preserve soundness of type-unsafe operations found in low-level code. Therefore, in the second part of the paper, we describe three different techniques for ensuring soundness with Burstall's model: insertion of soundness checks before each memory access, our novel eager type check elimination algorithm based on a lightweight pointer analysis, and our novel eager type unification technique. We showed in the experiments that naively inserting checks is an unnecessary verification overhead, since most of the checks can be eagerly removed using our algorithms. During the verification effort, we found three previously unreported bugs.

In an upcoming paper [12], Condit et al. describe a novel memory model for low-level code that includes type information. Types can be checked using an SMT solver, and they also provide a decision procedure for checking type safety. Using these techniques, they type-checked a number of Windows device drivers. Their work is complementary to ours: we conservatively and eagerly remove as many type checks as possible, whereas they provide an efficient technique to prove type checks. Obvious future work is to combine the best of both approaches: quickly eliminating most type checks using our methods, and solving the remaining ones efficiently using theirs.

References

1. The DDVerify verification tool. `http://www.verify.ethz.ch/ddverify/`, 2007. Cited: August 15, 2008.
2. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pp. 203–213, 2001.
3. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. *Symp. on Formal Methods for Components and Objects (FMCO)*, pp. 364–387, 2005.
4. M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. *Workshop on Program Analysis For Software Tools and Engineering (PASTE)*, pp. 82–87, 2005.
5. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. *Intl. Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS)*, pp. 49–69, 2005.
6. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pp. 196–207, 2003.
7. R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.
8. S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *Intl. Conf. on Software Engineering (ICSE)*, pp. 385–395, 2003.
9. S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamarić. A reachability predicate for analyzing low-level software. *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 19–33, 2007.

10. E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 168–176, 2004.
11. E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI–C programs using SAT. *Formal Methods in System Design*, 25(2-3):105–127, 2004.
12. J. Condit, B. Hackett, S. Lahiri, and S. Qadeer. Unifying type checking and property checking for low-level code. *ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, 2009. To appear.
13. J. Condit, M. Harren, S. Mcpeak, G. C. Necula, and W. Weimer. CCured in the real world. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pp. 232–244, 2003.
14. D. W. Currie, A. J. Hu, S. Rajan, and M. Fujita. Automatic formal verification of DSP software. *37th Design Automation Conference*, pp. 130–135. ACM/IEEE, 2000.
15. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 337–340, 2008.
16. R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.
17. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18:453–457, 1975.
18. J.-C. Filliâtre and C. Marché. Multi-prover verification of C programs. *Intl. Conf. on Formal Engineering Methods (ICFEM)*, pp. 15–29, 2004.
19. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pp. 234–245, 2002.
20. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. *ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, pp. 58–70, 2002.
21. F. Ivančić, I. Shlyakhter, A. Gupta, M. K. Ganai, V. Kahlon, C. Wang, and Z. Yang. Model checking C programs using F-Soft. *Intl. Conf. on Computer Design (ICCD)*, pp. 297–308, 2005.
22. C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. *Symp. on Code Generation and Optimization (CGO)*, pp. 75–88, 2004.
23. C. Lattner, A. Lenharth, and V. S. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pp. 278–289, 2007.
24. A. Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. *ACM SIGPLAN/SIGBED Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pp. 54–63, 2006.
25. Y. Moy. Union and cast in deductive verification. *C/C++ Verification Workshop (CCV)*, pp. 1–16, 2007.
26. Z. Rakamarić and A. J. Hu. Automatic inference of frame axioms using static analysis. *IEEE/ACM Intl. Conf. on Automated Software Engineering (ASE)*, pp. 89–98, 2008.
27. W. Schulte, S. Xia, J. Smans, and F. Piessens. A glimpse of a verifying C compiler (extended abstract). *C/C++ Verification Workshop*, 2007.
28. M. Siff, S. Chandra, T. Ball, K. Kunchithapadam, and T. W. Reps. Coping with type casts in C. *European Software Engineering Conf./ACM SIGSOFT Symp. on the Foundations of Software Engineering (ESEC/FSE)*, pp. 180–198, 1999.
29. T. Witkowski, N. Blanc, D. Kroening, and G. Weissenbacher. Model checking concurrent linux device drivers. *IEEE/ACM Intl. Conf. on Automated Software Engineering (ASE)*, pp. 501–504, 2007.