Precise and Efficient Static Array Bound Checking for Large Embedded C Programs

Arnaud Venet
Kestrel Technology
NASA Ames Research Center
Moffett Field, CA 94035
arnaud@email.arc.nasa.gov

Guillaume Brat
Kestrel Technology
NASA Ames Research Center
Moffett Field, CA 94035
brat@email.arc.nasa.gov

ABSTRACT

In this paper we describe the design and implementation of a static array-bound checker for a family of embedded programs: the flight control software of recent Mars missions. These codes are large (up to 280 KLOC), pointer intensive, heavily multithreaded and written in an object-oriented style, which makes their analysis very challenging. We designed a tool called C Global Surveyor (CGS) that can analyze the largest code in a couple of hours with a precision of 80%. The scalability and precision of the analyzer are achieved by using an incremental framework in which a pointer analysis and a numerical analysis of array indices mutually refine each other. CGS has been designed so that it can distribute the analysis over several processors in a cluster of machines. To the best of our knowledge this is the first distributed implementation of static analysis algorithms. Throughout the paper we will discuss the scalability setbacks that we encountered during the construction of the tool and their impact on the initial design decisions.

Categories and Subject Descriptors

F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program Analysis*

General Terms

Algorithms, Languages, Verification

Keywords

Abstract interpretation, program verification, pointer analysis, array-bound checking, difference-bound matrices

1. INTRODUCTION

It is well-known that runtime errors plague the development of large mission-critical software. In 1996, the explosion of Ariane 501 shortly after launch was due to an overflow in an arithmetic conversion. This failure cost over $500 millions to the European space program. Classical verification techniques based on development process, code reviewing and testing were unable to detect that defect. This overflow could have been detected by employing static analysis techniques which can automatically inspect the text of a program and check the safety of all operations. As a matter of fact, the failure of Ariane 501 gave birth to a commercial static analysis tool called PolySpace Ada Verifier [22]. This tool can perform precise static analysis of large Ada programs (over 1 MLOC) and find runtime errors. In a previous article [5], we have reported our experience with C Verifier (the C version of Ada Verifier) on real NASA software. Unfortunately, we found that C Verifier does not scale as well as its Ada counterpart. In short, we had to limit our analysis to code pieces of 20 to 40 KLOC and we obtained 20% of warnings after 8 to 12 hours of analysis. This level of performance was not enough to convince NASA software developers to adopt the technology.

We analyzed the reasons for these limitations and we decided to address them by prototyping our own static analysis tool called C Global Surveyor (CGS). We believe that it is extremely hard to build a static analyzer that works well for any C programs. The precision of a static analysis tool is measured in terms of the percentage of operations in the program that can be decided as safe (or unsafe). Precision is the main metric for judging the quality of a static analyzer. Therefore, designing a static analyzer for any type of C programs forces the tool implementer to make tradeoffs that sacrifice scalability. We extensively experienced with PolySpace C Verifier on a variety of NASA programs and we observed that precision remained consistently around 80%. However, there was a huge discrepancy between execution times, from a couple of hours to days. Our driving philosophy is that designing a tool for specific coding style and software architecture allows us to make different tradeoffs that optimize execution time for the software family we target.

Cousot et al. [3] used a similar approach to build a static analyzer that is specialized for software developed by Airbus; it can analyze 75,000 lines of C code without producing any warnings. Our goal with CGS is not as ambitious. Whereas the software analyzed in [3] is safety-critical, single-threaded and uses a very restricted subset of C, we have to analyze programs that are multithreaded and use the full power of pointer arithmetic. Our main purpose is to achieve a level of precision comparable to that of PolySpace C Verifier with much lower execution times, since in our case this is the decisive factor for having the technology adopted by missions at NASA. C Global Surveyor checks for one type of runtime errors: out-of-bounds array accesses. This is probably the most critical category of runtime errors because it silently corrupts the memory, causing nondeterministic behaviors during the mission. CGS is specialized for the NASA software following the Mars Path Finder (MPF) legacy, which we call the MPF family. The flight software for the Deep Space One mission (DS1) and the Mars Exploration Rover mission (MER) all belong to the MPF family.

The programs of the MPF family share a unique feature in the field of embedded applications: they are written in an object-oriented style. This means that these programs contain a myriad of small generic functions which are passed pointers to the segments of data on which they shall operate. This has two consequences on the structure of the analyzer. First, context-sensitivity should be enabled in order to distinguish between hundreds of calls to the same function. Second, interprocedural propagation should be very efficient. All decisions made in the design of CGS originate from these two observations. We do not claim that the architecture of CGS represents the optimal solution to this problem. The experiments showed that some of our choices were justified and some others were questionable. This paper should be seen as the critical report of a practical experience in implementing a large scale static analyzer

The paper is organized as follows. In Sect. 2 we introduce the abstract interpretation framework underlying the architecture of CGS. In particular we define the semantic model of the memory in which the symbolic information produced by the pointer analysis interacts with the numerical invariants produced by the flow-sensitive analysis of loops. Section 3 defines the abstract semantics of memory accesses and the generation of semantic equations that are used during the interprocedural propagation phase. In Sect. 4 we describe the architecture of CGS and our implementation choices. Section 5 summarizes the experimental results obtained for the MPF and DS1 codes on a cluster of PC workstations. We give a critical interpretation of these results with respect to the design decisions. We discuss related work in Sect. 6 and we end the paper with concluding remarks.

2. ABSTRACT INTERPRETATION FRAMEWORK

Abstract Interpretation [7, 8, 10] is a theoretical framework for the systematic construction of provably correct static analyzers. Classically, the abstract interpretation of a program consists of attaching to each program point an abstract memory configuration that is a conservative approximation of the actual memory configuration for all executions of the program that reach that point. This information can be automatically inferred by associating an abstract semantic transformer to each basic operation of the program and computing the composition of these transformers along all possible executions paths in the control-flow graph. This is achieved in practice by constructing a system of abstract semantic equations that describes the flow of information in the program and by applying appropriate fixpoint algorithms for computing the solution of the system, usually with the help of widening/narrowing operators in order to ensure termination and/or rapid convergence.

In our case we are interested in discovering all possible addresses that can flow through each pointer variable in the program. Thus, we can check whether every memory read or write operation of the program occurs within the bounds of a memory block. We are not interested in checking whether a pointer is `NULL` or contains an undetermined value. This is a different problem that has to be treated with a separate analysis. Therefore, in our abstract semantic model the denotation of a pointer always contains `NULL` and any undetermined value. We can nevertheless flag an illegal memory access with certainty whenever our analysis discovers an empty points-to set.

The C language authorizes creating a pointer to an object inside a compound data structure, for example to the element of an array. This construct is heavily used in the MPF and DS1 codes, since data are organized in large structures which are modified via pointer references passed to generic functions. Therefore, our abstract memory model should represent references as a triple $(\mathbf{a}, \pi, s)$ where $\mathbf{a}$ is the address of a memory block, π is an access path into the block and s is the size of the block. An address is either the address of a variable `&A`, a constant character string $\mathbf{string}_\ell$ that appears in the program at the program location ℓ, or the dynamic allocation of a block $\mathbf{malloc}_\ell$ at the program location ℓ. Our model does not distinguish between instances of a `malloc` in a loop simply because this situation never occurs in the class of programs that we are considering, although techniques exist that can cope with this problem [29, 26].

Without access path information it is impossible to perform any precise array bound checking. We could use the type information contained in the C program for representing access paths symbolically. Unfortunately, the aggressive type casting mechanism of C combined to pointer arithmetic ruins this idea. Consider for example the following fragment of code:

```
struct MsgHeader {
  int id;
  int length;
};
struct Msg_X {
  struct MsgHeader header;
  Data_X data;
};
/* Thread 1 */
struct Msg_X *msg = malloc (...);
...
sendMsg (Thread_2, msg);
...
/* Thread 2 */
struct MsgHeader *msg = readMsg (...);
if (msg->id == ID_OF_X) {
  Data_X *data = (DataX *)(msg + 1);
  ...
```

This is in essence how the message passing mechanism for thread communication is implemented in the MPF family. All messages start with the same header which contains an id that uniquely determines the type of the message. The data are stored right after the header. The actual type of the data is only known after the message id has been read, which explains this seemingly odd construction. This piece of code illustrates the overall object-oriented design of the

MPF family software. Messages are considered as objects and this is nothing more than a manual encoding of virtual method dispatch. However, this makes the manipulation of symbolic access paths extremely difficult since we must keep track of the actual layout of structure components in memory in order to cope accurately with pointer arithmetic. Our solution to this problem consists of choosing a uniform offset-based representation of structure components instead of symbolic access paths. A reference is then represented by a triple $(\mathbf{a}, o, s)$ where o is an offset from the beginning of the block expressed in bytes. With this numerical model, type casting is no more an issue and becomes transparent for the analysis. All architecture-dependent problems entailed by this representation like memory alignment and padding are completely resolved by the C front-end. Therefore, there is no extra complexity in implementing this model.

For ensuring computability we approximate a set $\{(\mathbf{a}_i, o_i, s_i) \mid i \in I\}$ of memory references by an *abstract memory reference* $(\{\mathbf{a}_i \mid i \in I\}, O, S)$ where O and S are the smallest intervals such that $\forall i \in I : o_i \in O \ \&\ s_i \in S$. This corresponds to the notion of *attribute-independent* or *cartesian* approximation [9]. We can gain precision by considering the *reduced product* [8] between the powerset lattice of addresses and the lattice of intervals. The size of memory blocks is known at compile time for the address of a static memory block, i.e. an address of type `&A` or **string**. We denote by $\mathbf{sz}(\mathbf{a})$ the size of the block at address $\mathbf{a}$. If $\mathbf{a}$ is the address of a dynamically allocated block we set $\mathbf{sz}(\mathbf{a}) = [-\infty, +\infty]$. The reduced product consists of refininig the expressiveness of each lattice by bringing information from the other one. In practice this is performed by applying a *reduction operation* σ defined as follows:

$$\sigma(A, O, S) = \left(\{\mathbf{a} \in A \mid \mathbf{sz}(\mathbf{a}) \in S\}, O, S \cap \left(\bigcup_{\mathbf{a} \in A} \mathbf{sz}(\mathbf{a})\right)\right)$$

The effect of this operation is to remove spurious references and reduce the size range, which results in better accuracy. The reduction should always be performed on an abstract memory reference before any operation is applied to it. In practice reduction turned out to be very important, because in many cases the numerical information was too coarse to represent the size precisely.

An abstract memory configuration is thereby a couple (E, H) where E is an abstract environment mapping each local pointer variable of a function to an abstract memory reference and each local integer variable to an interval, and H is an abstract heap. An abstract heap is a set of points-to relations $\langle(\mathbf{a}, O) \mapsto (\mathbf{a}', O', S)\rangle$ where $\mathbf{a}, \mathbf{a}'$ are addresses and O, O', S are intervals. Such a relation expresses that there may be references in the block $\mathbf{a}$ within the range of offsets O to the elements within the range of offsets O' in the block $\mathbf{a}'$, the size of which lies in the interval S. Furthermore, we impose that there are no two distinct points-to relations in H with the same addresses $\mathbf{a}$ and $\mathbf{a}'$. We perform two additional approximations that are crucial for the tractability of this model:

1. Abstract environments are *field-insensitive*, i.e. we do not distinguish between the values of fields within a compound local variable.

2. Local variables that are address-taken (i.e. modified through a pointer reference) are *globalized*, i.e. they are represented in the abstract heap H. This means that we cannot distinguish the value of such a variable between different execution contexts.

These approximations ensure that the domain of an abstract environment E only contains variable names without any offset information, and that any modification of the value of a variable v in E may only be performed by an assignment operation in which v explicitly appears. Even though local structures and address-taken variables are quite common in the MPF family, they rarely concern variables that carry pointers. Thus the impact of these approximations on the precision is low whereas they greatly simplify the design of the analyzer.

As is, the classical abstract interpretation framework that assigns an abstract memory configuration (E, H) at each control point is not applicable to heavily multithreaded programs, since this requires considering all possible thread interleavings. A solution would be to use a flow-insensitive analysis, which can obviously cover all possible thread interleavings at a low cost. However, the array bound checking absolutely requires precise loop invariants, which cannot be obtained without flow-sensitivity. Our solution consists of using a mixed framework similar to that of [26] in which the abstract environment E is computed in a flow-sensitive way whereas the abstract memory heap H is constructed in a flow-insensitive way.

More precisely, let H be an abstract heap that is a conservative approximation of all possible heaps that can be generated by the program at any point of any execution. We perform a flow-sensitive analysis by keeping the second component of the abstract memory configurations equal to H. In this case we do not have to consider any thread interleaving at all since the variables in the domain of the abstract environments are thread-local. We simply analyze the program as if it were sequential, the initial states being given by all the thread entry points. During the analysis memory reads are always performed on H and memory writes are never taken into account. More precisely, if `p = *q` is a read operation that fetches a pointer from the heap, we get the abstract memory reference (A, O, S) associated to `q` at this stage of the computation. The result of the read operation is the join of all memory references $(\{\mathbf{a}'\}, O', S')$ for which there exists a points-to relation $\langle(\mathbf{a}, O'') \mapsto (\mathbf{a}', O', S')\rangle$ in H such that $\mathbf{a} \in A \ \&\ O \cap O'' \neq \emptyset$.

At the end of the analysis, we consider all memory write operations of the program. For each such operation `*p = q` that may carry a pointer we retrieve the abstract memory reference (A, O, S) associated to `p` and the abstract memory reference (A', O', S') associated to `q` at this point. For each **a** in A and each **a'** in A', we generate a points-to relation $\langle(\mathbf{a}, O) \mapsto (\mathbf{a}', O', S')\rangle$. We gather all the points-to relations generated this way to form a new abstract heap H'. The abstract heap H' satisfies two properties:

1. H' is a conservative flow-insensitive approximation of all actual heaps of the program.

2. H' *refines* H, denoted by $H' \sqsubseteq H$: for all $\langle(\mathbf{a}, O_1) \mapsto (\mathbf{a}', O'_1, S_1)\rangle$ in H', there is a points-to relation $\langle(\mathbf{a}, O_2) \mapsto (\mathbf{a}', O'_2, S_2)\rangle$ such that $O_1 \subseteq O_2, O'_1 \subseteq O'_2 \ \&\ S_1 \subseteq S_2$.

This provides us with a process for incrementally refining the abstract heap. We start with a coarse flow-insensitive approximation of the heap H_0 and we construct a decreasing sequence $H_0 \sqsupseteq H_1 \sqsupseteq \cdots \sqsupseteq H_n$ of abstract heaps with respect to the refinement order. We can use the pointwise extension of the narrowing of intervals to define a narrowing operation over abstract heaps. We can then automate this process, using the narrowing to enforce stabilization. Automatic stabilization is not implemented in the current version of CGS, the user must explicitly give the number of refinement steps that shall be computed.

To illustrate this mechanism, consider for example a program working on an array `A` of two pointers, a pointer variable `P` and two integer variables `I` and `J`, and made of three simple threads defined as follows:

```
void task1() {     void task2() {     void task3() {
  A[0] = &I;         P = A[0];          A[1] = &J;
}                  }                  }
```

Imagine that we are provided with a conservative field-insensitive approximation H_0 of the memory graph as follows:

$$H_0 = \left\{ \begin{array}{l} \langle \&\mathtt{A}, [-\infty,+\infty], [8,8]\rangle \mapsto \langle \&\mathtt{I}, [-\infty,+\infty], [4,4]\rangle, \\ \langle \&\mathtt{A}, [-\infty,+\infty], [8,8]\rangle \mapsto \langle \&\mathtt{J}, [-\infty,+\infty], [4,4]\rangle, \\ \langle \&\mathtt{P}, [-\infty,+\infty], [4,4]\rangle \mapsto \langle \&\mathtt{I}, [-\infty,+\infty], [4,4]\rangle, \\ \langle \&\mathtt{P}, [-\infty,+\infty], [4,4]\rangle \mapsto \langle \&\mathtt{J}, [-\infty,+\infty], [4,4]\rangle \end{array} \right\}$$

assuming pointers and integers occupy four bytes in memory on the architecture considered. After one step of iteration, the elements at indices 0 and 1 of array `A` are entirely determined, however the value of `P` is computed from the points-to information contained in H_0. Therefore we obtain the following memory graph:

$$H_1 = \left\{ \begin{array}{l} \langle \&\mathtt{A}, [0,0], [8,8]\rangle \mapsto \langle \&\mathtt{I}, [0,0], [4,4]\rangle, \\ \langle \&\mathtt{A}, [4,4], [8,8]\rangle \mapsto \langle \&\mathtt{J}, [0,0], [4,4]\rangle, \\ \langle \&\mathtt{P}, [0,0], [4,4]\rangle \mapsto \langle \&\mathtt{I}, [-\infty,+\infty], [4,4]\rangle, \\ \langle \&\mathtt{P}, [0,0], [4,4]\rangle \mapsto \langle \&\mathtt{J}, [-\infty,+\infty], [4,4]\rangle \end{array} \right\}$$

Note that the offset in the memory block `&P` has been solved because the assignment `P = A[0]` writes its lefthand side at the offset 0. After one more iteration step, the assignment to `P` in task 2 can be precisely solved, since the memory layout of `A` has been completely determined at the previous iteration step. We finally obtain:

$$H_2 = \left\{ \begin{array}{l} \langle \&\mathtt{A}, [0,0], [8,8]\rangle \mapsto \langle \&\mathtt{I}, [0,0], [4,4]\rangle, \\ \langle \&\mathtt{A}, [4,4], [8,8]\rangle \mapsto \langle \&\mathtt{J}, [0,0], [4,4]\rangle, \\ \langle \&\mathtt{P}, [0,0], [4,4]\rangle \mapsto \langle \&\mathtt{I}, [0,0], [4,4]\rangle \end{array} \right\}$$

It now remains the problem of bootstrapping the iterative process, i.e. obtaining the first approximation H_0. We first used Steensgaard's analysis [24] enhanced with Das' one-level flow edges optimization [13]. However the resulting abstract heap was too coarse, and there were spurious points-to relations introduced at that stage that remained in all subsequent refinement steps. One source of imprecision was due to the way message queues are allocated. The unique `malloc` call that creates a queue is nested within several function calls. Since in our memory model allocations can only be distinguished by the syntactic location of the corresponding `malloc`, all message queues were merged, resulting in an unrecoverable loss of precision. Adding an option to CGS allowing to inline the corresponding functions solved this problem. The idea is to treat isolated sources of imprecision manually in this way rather than complicating the pointer analysis in order to cover all special cases. The drawback is that this kind of instrumentation can only be done by a high-end user who perfectly knows the internals of the analysis and how to cope with this kind of situation (see also [3] for a discussion of this issue).

A substantial amount of the remaining spurious points-to relations was due to brutal unification operations in Steensgaard's analysis caused by pointers stored in global variables. The solution consisted of extending Das analysis in order to be able to handle n-level flow edges without sacrificing efficiency. We believe that scalable versions of Andersen's analysis [2] could have been considered as well for the bootstrap [18]. We unfortunately did not have the time to implement an inclusion-based analysis and compare the results.

This ends the presentation of the abstract interpretation framework implemented in CGS. We now have to present the details of the abstract semantic equations.

3. ABSTRACT SEMANTICS

The symbolic and numerical parts of an abstract memory reference are independent, which means that we can compute these two pieces of information separately. We just need to perform a reduction operation σ whenever there is a context change (function call) or an interaction with the abstract heap (memory read). The choice of performing a cartesian approximation for the abstract memory references was mainly motivated by this simplifying assumption in the abstract semantics.

We generate two separate sets of semantic equations for each function in the program, one for the symbolic part in the form of inclusion constraints between points-to sets, the second as a system of numerical constraints between offset and size variables. The resolution of these equations follows the call graph by propagating call contexts made of points-to sets and intervals. The symbolic and numerical systems associated to a function `f` are solved separately for all possible call contexts of `f` (depending on whether context-sensitivity is enabled for this function or not). The resolution of these two systems of equations is interleaved, interactions occurring whenever some information is retrieved from the environment, i.e. by a memory read. In this case we have to combine the numerical and symbolic information in order to query the memory graph H used at this step of the resolution.

3.1 Points-to Inclusion Constraints

Given a function `f` of the program, we associate a metavariable $A_{\mathtt{p}}$ to each local variable `p` of `f` that may carry a pointer (either a pointer variable itself or a compound variable with pointer-valued fields). These metavariables represent the first component of an abstract memory reference, i.e. a set of symbolic addresses. Following the model defined in [26] we associate an *anchor* metavariable A_ℓ to each location ℓ of a memory read operation or a function call that may return a pointer. The metavariable A_ℓ represents the set of addresses returned by the read operation or the function call. We similarly assign a special anchor metavariable $A_{\mathtt{x@f}}$ to each formal parameter `x` of `f` that may carry a pointer. This anchor denotes the points-to set of the argument passed to the function and is used during interprocedural propagation. Following Andersen's model [2] we use inclusion constraints of the form $A_{\mathtt{p}} \supseteq A_{\mathtt{q}}$ to relate the metavariables.

The generation of inclusion constraints is quite straightforward. For all assignments `p = q`, `p = q + n` (pointer arithmetic) or `p = (T *)q` (type cast), we generate a constraint $A_{\mathtt{p}} \supseteq A_{\mathtt{q}}$. For all memory read operation `p = *q` or function call `p = f (...)` at a location ℓ in the program we generate a constraint $A_{\mathtt{p}} \supseteq A_\ell$ and we record a semantic operation $\mathbf{read}(A_\ell, A_{\mathtt{q}})$ which is used during interprocedural propagation for querying the abstract memory graph. A memory write operation `*p = q` is not assigned an inclusion constraint, it is simply assigned a semantic operation $\mathbf{write}(A_{\mathtt{p}}, A_{\mathtt{q}})$ which is used at the end of an analysis pass to generate a new abstract heap, as described in the previous section. Similarly a `return p` statement is recorded separately as $\mathbf{return}(A_{\mathtt{p}})$ and is used for the construction of the transformers in the backward propagation phase described in Sect. 3.3. We must also add the constraints corresponding to the implicit binding relations between formal and actual parameters as follows: $A_{\mathtt{x}} \supseteq A_{\mathtt{x@f}}$, for all formal parameter `x` of `f`.

The resolution of these constraints differs from Andersen's algorithm [2] since **read** operations retrieve data from the abstract memory graph H and require some information about the offset at which the memory block is read. Our algorithm consists of a local fixpoint iteration that computes a set of symbolic addresses for each metavariable of `f` and launches the resolution of numerical constraints on demand whenever a memory read is encountered. For efficiency the resolution algorithm implemented in CGS first computes the directed acyclic graph of strongly connected components of the dependency graph of the system of inclusion constraints. The iterations are then performed locally on each strongly connected component following a weak topological ordering of the metavariables [4].

3.2 Numerical Constraints

Classically, when building an abstract interpretation of numerical computations, the abstract semantic equations follow the program structure [12]. A loop statement in the body of a function will appear as a recursive dependency in the equations. Solving the system precisely usually requires computing two fixpoint iterations, the first one with widening the second with narrowing. These calculations should be performed on the whole program, i.e. hundreds of thousands lines of C, at each step of the heap refinement process described in the previous section. In practice, we measured that at least five global iterations over the program are needed to achieve a good level of precision. It was unrealistic to perform a full-strength fixpoint iteration at each step; it would severely impair the efficiency of the analyzer. We decided to first compute a summary of each function of the program by using a relational numerical lattice as described in [11].

As for the points-to inclusion constraints, given a function `f` of the program, we associate two numerical metavariables $O_{\mathtt{p}}$ and $S_{\mathtt{p}}$ to each local variable `p` of `f` that may carry a pointer. The metavariables $O_{\mathtt{p}}$ and $S_{\mathtt{p}}$ represent respectively the offset and size ranges of the abstract memory reference carried by the variable. We also associate a metavariable $I_{\mathtt{n}}$ to each integer valued local variable `n`. Recall that local variables that are address-taken are globalized and never occur in an abstract environment. We also attach two anchor metavariables O_ℓ and S_ℓ to each location ℓ of a memory read/write operation or a function call that may return a pointer. The metavariables O_ℓ and S_ℓ represent respectively the offset and size ranges of the abstract memory reference returned by the operation at that point. We similarly attach special anchors $O_{\mathtt{x@f}}$ and $S_{\mathtt{x@f}}$ (resp. $I_{\mathtt{x@f}}$) to each pointer-valued (resp. integer-valued) formal parameter `x` of `f`. .

We could also attach anchor metavariables I_ℓ to each location ℓ of a memory read operation or a function call that returns an integer. CGS actually has command-line options to generate such anchors. The representation of integer values in the abstract heap is identical to that of pointers, i.e. it consists of mapping a memory location $\langle \mathbf{a}, O, S\rangle$ to an interval $[a, b]$. Some extra care is required when reading an integer from the heap in order to ensure that the offset of the read operation is aligned with the offset of the integer in the memory block, otherwise this would result into returning a truncated value. Similarly we have to make sure that the sizes match, for example if we try to read a byte from the location of an integer, otherwise the results would be inconsistent. We address these issues in a very simple way: whenever we encounter a read operation of an integer of size s from the address $\mathbf{a}$ at the offset O' and there is a mapping $\langle \mathbf{a}, O, S\rangle \mapsto [a, b]$ in the abstract heap, we return the interval $[a, b]$ if and only if O and S are singletons and $O = O', S = [s, s]$. We return $[-\infty, +\infty]$ otherwise. Surprisingly enough, the experiments showed no noticeable gain in precision on the MPF family with this option of CGS enabled.

Now we need to choose a relational abstract domain for representing relationships between the numerical metavariables. Consider for example the following function which is representative of the matrix computations performed in the programs of the MPF family:

```
void equate (double *p, double *q, int n) {
  int i;

  for (i = 0; i < n; i++)
    p[i] = q[i];
}
```

In the abstract syntax tree of this function the body of the loop is represented by the three following statements:

```
a = p + i;
b = q + i;
c = *b;
*a = c;
```

The variables `a`, `b` and `c` are internal names generated by the front-end. If we assume that the size of a `double` is 8 bytes, the exact loop invariant is given by

$$\begin{cases} S_{\mathtt{a}} = S_{\mathtt{p@equate}} \\ 0 \le O_{\mathtt{a}} - O_{\mathtt{p@equate}} \le 8 * I_{\mathtt{n@equate}} - 8 \\ S_{\mathtt{b}} = S_{\mathtt{q@equate}} \\ 0 \le O_{\mathtt{b}} - O_{\mathtt{q@equate}} \le 8 * I_{\mathtt{n@equate}} - 8 \end{cases}$$

where we have eliminated all metavariables associated to local integer variables of the function, since they are just used for storing the result of intermediate computations. It immediately appears in this simple example that we need general linear inequalities in order to be precise. The only abstract domain that is expressive enough for representing this kind of invariants is the lattice of convex polyhedra [12]. Unfortunately, because of the complexity of the underlying algorithms this lattice cannot be used for representing relationships between more than 20 variables in practice. The functions in the codes of the MPF family can be quite large and use many pointers simultaneously. We found that in some functions more than 30 pointers were active in the body of a loop. Moreover, the abstract syntax tree representation provided by the front-end introduces numerous internal variables since all statements are broken down into a 3-address format.

Some numerical relational lattices have been developed recently that showed good promises of scalability [20, 21]. However they are not expressive enough for representing the kind of linear inequalities in which we are interested. They can only express linear inequalities between two variables and the coefficients of these variables may only be 1 or -1. Our solution consists of modifying the form of our numerical constraints by introducing additional variables so that the overall expressiveness of a system of numerical constraints is kept constant, whereas the class of numerical relations required to achieve this expressiveness is simpler.

More precisely, it appears that the main source of complexity comes from the byte-based representation of offsets. An array access `p[i]` is transformed into an arithmetic expression in which we multiply the index by the size of an array element expressed in bytes. We extend the representation of a pointer `p` by attaching additional metavariables $\delta_1(\mathtt{p}), \ldots, \delta_k(\mathtt{p})$ and $u_1(\mathtt{p}), ..., u_k(\mathtt{p})$ for a fixed k. A pair $(\delta_i(\mathtt{p}), u_i(\mathtt{p}))$ represents an offset expressed in a different unit than the byte. $\delta_i(\mathtt{p})$ is the relative offset and $u_i(\mathtt{p})$ is the base. The actual offset in bytes denoted by this representation is given by the following formula:

$$O_{\mathtt{p}} + \sum_{i=1}^{k} \delta_i(\mathtt{p}) * u_i(\mathtt{p})$$

We call the representation $W_{\mathtt{p}} = \langle O_{\mathtt{p}}, (\delta_1(\mathtt{p}), u_1(\mathtt{p})), \ldots, (\delta_k(\mathtt{p}), u_k(\mathtt{p}))\rangle$ a *sliding window*. We call $O_{\mathtt{p}}$ the *base offset*. The associated sliding operation $\mathbf{slide}(W_{\mathtt{p}}, \delta, u)$ is defined as follows:

$$\mathbf{slide}(W_{\mathtt{p}}, \delta, u) = \langle O_{\mathtt{p}} + \delta_1(\mathtt{p}) * u_1(\mathtt{p}), (\delta_2(\mathtt{p}), u_2(\mathtt{p})), \ldots, \\ (\delta_{k-1}(\mathtt{p}), u_{k-1}(\mathtt{p})), (\delta, u)\rangle$$

The initial values of the sliding window for metavariables associated to inputs of the function, i.e. the parameters and the return values of a memory read or a function call, are set to 0 except for the base offset and u_k. The base offset is the one associated to the metavariable and u_k is the size of the element pointed to by the variable as it appears in the type inferred by the C front-end.

The sliding operation is used for handling a type cast operation $\mathtt{p} = (\mathtt{T}*)\mathtt{q}$. When analyzing this operation we first retrieve the range of $u_k(\mathtt{q})$ from the current system of inequalities. If it is a singleton and it is equal to the size t of `T` then $W_{\mathtt{p}} = W_{\mathtt{q}}$, otherwise $W_{\mathtt{p}} = \mathbf{slide}(W_{\mathtt{q}}, 0, t)$. This way u_k always represents the size of the element currently pointed-to by the variable. Whenever a pointer arithmetic operation $\mathtt{p} = \mathtt{q} + \mathtt{n}$ is analyzed, the sliding window $W_{\mathtt{p}}$ is equated to $W_{\mathtt{q}}$ except for $\delta_k(\mathtt{p})$ for which the constraint $\delta_k(\mathtt{p}) = \delta_k(\mathtt{q}) + I_{\mathtt{n}}$ is generated. Now if we analyze the function `equate` with sliding windows of size $k = 2$ and the abstract numerical domain of difference-bound matrices [20], we obtain the following system of constraints for the loop invariant:

$$\left\{\begin{array}{ll} S_{\mathtt{a}} = S_{\mathtt{p@equate}} & S_{\mathtt{b}} = S_{\mathtt{q@equate}} \\ O_{\mathtt{a}} = O_{\mathtt{p@equate}} & O_{\mathtt{b}} = O_{\mathtt{q@equate}} \\ \delta_1(\mathtt{a}) = u_1(\mathtt{a}) = 0 & \delta_1(\mathtt{b}) = u_1(\mathtt{b}) = 0 \\ 0 \leq \delta_2(\mathtt{a}) \leq I_{\mathtt{n@equate}} - 1 & 0 \leq \delta_2(\mathtt{b}) \leq I_{\mathtt{n@equate}} - 1 \\ u_2(\mathtt{a}) = 8 & u_2(\mathtt{b}) = 8 \end{array}\right.$$

We can express the exact loop invariant with a less powerful abstract lattice and more variables.

We chose the domain of difference-bound matrices [20] (DBMs for short) for expressing numerical constraints between variables. In this domain a constraint may only have the form $x - y \leq c$ where c is an integer. The fundamental operation on a DBM is the normalization that refines constraints by repeated application of the following rule:

$$\left.\begin{array}{l} x - y \leq c \\ y - z \leq c' \\ x - z \leq c'' \end{array}\right\} \Rightarrow x - z \leq \min(c + c', c'')$$

Our choice was motivated by the observation that DBMs have a sufficient expressiveness for our purpose and by the existence of an efficient quadratic algorithm devised by Johnson [6] for the normalization of sparse systems of constraints. We assumed indeed that the systems of constraints would be rather sparse, since it would be very unlikely to have all variables in a function related at the same time. Our first implementation used Floyd-Warshall's algorithm [6] for computing the normalization operation. The execution times were catastrophic. A simple function independently manipulating 20 pointer variables within a loop took more than 15 minutes to analyze. The execution time did not change at all when we tried Johnson's algorithm.

After a careful inspection of the results it appeared that the system of inequalities was always dense, i.e. all variables were related. Therefore the cubic worst case execution time was always attained. The reason was to be found in the way simple range constraints of the form $a \leq x \leq b$ are represented. A DBM always contains a dummy zero variable Z which has the value 0. Range constraints are translated into constraints of the form $a \leq x - Z \leq b$. Therefore all variables introduced in a DBM during the analysis become implicitly related as soon as a range constraint is involved, in other terms always. Thus completely independent variables become related from the moment they receive a constant (during initialization for example). This was a surprising and disappointing result.

Our response to this situation was to explicitly pack computationally dependent variables together, so that the analyzer works on a collection of smaller DBMs. A similar situation has been independently reported in [3]. In that work the authors pack variables in small groups using a syntactic criterion (all variables that appear within a same statement). In our case, such a simple criterion does not work. Pointer variables and loop counters can become related in a nontrivial way via the sliding window representation. We could not even use a dependency analysis because the application of the `slide` operation depends on the range of u_k which can only be known during the fixpoint iteration. Any dependency analysis performed beforehand would relate all variables of the sliding windows which would still lead to a high workload.

Our solution consisted of dynamically computing the dependency relation between metavariables during the execution of the analysis. We start with all metavariables being unrelated and we incrementally merge the DBMs whenever two of their variables become related by an operation of the program. We also merge the associated zero variables. We should also take care of implicit dependencies, i.e. the invisible dependencies between variables which are modified within a loop. If we do not consider these relations we lose all relations between array indices and loop counters for example. Therefore we first perform a rapid analysis of every loop in order to check the variables that can be modified in the body and we explicitly relate them before analyzing the loop. We are then able to infer all invariants that can be expressed with our abstraction. The function that took 15 minutes with the classic DBM domain could now be analyzed in about 10 seconds.

The domain of adaptive DBMs that we have constructed in that way is an order of magnitude of complexity beyond the original one. Fortunately it can be simply described as an instance of a *cofibered domain* [27, 28]. Cofibered domains were initially introduced to construct complex domains for pointer analysis. They enable the manipulation of dependent abstract domains, i.e. families of abstract domains indexed by the elements of a lattice. The domain of adaptive DBMs is exactly a cofibered domain: the indexing lattice is the set of all partitionings of the set of variables ordered by the refinement relation, and the abstract domain associated to one partitioning of the variables is the product of the family of DBM domains based upon each set in the partitioning. We measured that the average size of a partition of correlated variables was five elements. It would actually be an interesting experiment to use convex polyhedra instead of DBMs in the cofibered domain, since five is a tractable dimension for polyhedra, and compare the gain in precision.

3.3 Interprocedural Propagation

Function pointers are widely used in embedded programs for efficiency reasons. There are plenty of them in codes of the MPF family. We realized that a simple control-flow analysis based on Steensgaard's algorithm [24] was sufficient to solve exactly almost all computed calls. As a matter of fact, recent experimental evaluations showed that simple pointer analyses were sufficient to resolve computed calls in most applications [19]. We perform this simple control-flow analysis at the bootstrap prior to launching the interprocedural propagation phases. Having all computed calls resolved at bootstrap makes the design of the interprocedural propagation algorithms tremendously simpler. In order to achieve efficiency we break down the interprocedural propagation into two phases:

1. A backward propagation phase computes transformers relating the parameters of a function with its return value. These transformers are expressed using the domain of adaptive DBMs.

2. A forward propagation phase uses the transformers computed in the previous phase to propagate abstract memory references and ranges using the lattice of intervals.

The transformers computed during the backward propagation phase are used during the forward propagation to solve a function call without having to analyze the body of the called function. The **return** operations are used at this moment to propagate the constraints between the return value and the arguments of the call. A coarse version of the transformers are computed during the bootstrap in order to enable the first forward propagation phase. Using a classical resolution scheme would have implied iterating over interprocedural cycles induced by the two-way dependencies between a caller and a callee (function parameters/return value), which is completely unrealistic for large programs.

The interprocedural propagation phase of CGS can be context-sensitive. We implemented call-site sensitivity, i.e. the invariants of a function are duplicated depending on the syntactic call site. This level of context-sensitivity is sufficient for the MPF family, since it handles the common situation where a pointer to some part of a big structure (typically an array of `double` representing a vector or a matrix) is transmitted to a mathematical function. Context sensitivity is not applied uniformly, but only to functions which have a pointer in their signature, since this is the only situation where the analysis is able to distinguish between different call contexts. Context-sensitivity is extremely important for precision. Arrays of `double`, which are the main data structures manipulated by the MPF family codes, are usually transmitted together with an integer parameter containing the size of the array like in the `equate` example above. Since the numerical call contexts computed by CGS only are made of intervals, they cannot express a relation between the size of the array and the integer parameter. The only way to capture this information is to enumerate all call contexts. Hence, without context-sensitivity the tool would be unable to perform any precise array bound checking on this large family of functions.

4. ARCHITECTURE OF CGS

The algorithmic core of C Global Surveyor consists of 20,000 lines of C code. The tool is architected around three main phases:

1. **The build.** This phase computes the points-to constraints and the numerical inequalities for each function in the program.

2. **The bootstrap.** This phase performs a flow-insensitive pointer analysis and a coarse context-independent resolution of the numerical inequalities, in order to obtain a first approximation of all memory accesses. These results are used to construct the call graph and an initial approximation of the heap.

3. **The solve.** This phase consists of performing a forward or backward interprocedural propagation of numerical invariants. The results obtained at the end of this phase are used to compute a new abstract heap that refines the previous one. This phase should be repeated until a satisfactory level of precision has been attained.

There are two additional satellite phases:

- **The initialization**. This phase is performed at the very beginning and collects general information about the program, like the table of global variables, the table of functions, etc.

- **The array-bound check (abc)**. This phase can be executed at any time after the bootstrap and checks the safety of all memory accesses from the results of the analysis available at this moment. The precision computed at the end of this phase is the main criterion for deciding whether to continue refining the results or stop at this point.

A very important decision in the initial design of a static analyzer is the choice of the front-end. We chose the Edison Design Group's C/C++ front-end [15], a commercial front-end which supports a large variety of C dialects. Moreover, the Green Hills' compiler [17], which is widely used at NASA especially for developing flight software, is based on this front-end. This is a relevant factor when considering the application of the tool to other types of programs developed inside NASA.

CGS has been designed from the beginning with a distributed model of computation in mind. Therefore, we tried to parallelize all phases for which this makes sense, i.e. the build and the refinement, the nature of the algorithms used in the bootstrap precluding any attempt of parallelization. We chose the Parallel Virtual Machine (PVM) for implementing the distribution layer [16]. A major problem consisted of storing the artifacts of the analysis and transmitting them to the processes running on parallel. We decided to use a relational database for both the storage and the communication between processes of the artifacts, the PVM communication mechanism being merely used for sending commands to processes. We chose the PostgreSQL [25] database to work with CGS. The architecture of CGS is illustrated in Fig. 1. Note that each phase launches a master PVM process that in turn launches slave processes. Slave processes operate on each C file of the program for the initialization, the build and the array-bound check, whereas they operate on functions in the solve phase. The bootstrap is the only sequential phase.

It is not surprising to say that the cost of communications is the major limiting factor in designing a distributed application. CGS follows the same communication pattern for each job: all needed artifacts are retrieved from the database at the beginning of the job, the results are stored in internal memory until the job completes, then the results are written into the database. Two important algorithmic issues in designing the distribution of jobs in CGS are the *granularity* (which jobs should be executed in parallel) and the *scheduling* (in which order jobs should be executed).

The granularity of the build phase is the file: one PVM process is launched for generating the semantic equations of each source file. The scheduling of tasks in the build follows a metric calculated during the initialization phase which estimates the complexity of the fixpoint computation for each function of the program. Complex files are executed in priority in order to prevent the computation from being blocked by a big job that has been scheduled at the end of the worklist. The function-level granularity gave poor results because the analysis time of a single function is so short that the database becomes overwhelmed by numerous concurrent accesses.

The granularity of the solve phase is the function: one PVM process is launched for computing the invariant of each function. The scheduling follows a weak topological ordering [4] given by the call graph in each way (forward/backward): a function is added to the worklist whenever all its predecessors have been analyzed. We have limited control on the granularity and scheduling of the solve phase because of it is entirely bound to the structure of the call graph. The choice of the next function to schedule from the worklist turned out to be critical. In our first experiments we used simple heuristics that all led at some point to an almost sequential execution. Therefore, we should find a scheduling strategy that tries to maximize the parallelism. We chose a heuristic that consists of picking up the next function to schedule from the worklist that has the largest number of calls to functions which are not in the worklist yet. This heuristic is simple to compute and gives good results in terms of distribution.

Phase	*MPF (140 KLOC)*				
	1 cpu	*2 cpus*	*4 cpus*	*6 cpus*	*8 cpus*
init	232	187	113	78	67
build	1253	791	538	372	327
bootstrap	416	383	412	419	426
fwd solve	873	545	438	354	344
bwd solve	897	529	413	343	331
fwd solve	867	548	435	348	346
abc	274	211	374	697	880

Figure 2: Average analysis times (in seconds) per phase for MPF

Phase	*DS1 (280 KLOC)*				
	1 cpu	*2 cpus*	*4 cpus*	*6 cpus*	*8 cpus*
init	457	357	264	230	208
build	3678	1979	1480	1313	1155
bootstrap	711	663	780	777	686
fwd solve	1689	1075	914	860	771
bwd solve	1811	1062	885	803	688
fwd solve	1666	1080	954	853	767
abc	537	484	413	824	1022

Figure 3: Average analysis times (in seconds) per phase for DS1

5. EXPERIMENTAL RESULTS

This section shows two types of performance measures for CGS. First, we study the improvement of analysis times (for each phase) in function of the number of available CPUs. Note that all CPUs are identical (2.2 MHz with 1 GB of memory). Second, we show how the precision evolves with each solve phase. We distinguish between forward and backward interprocedural propagation in the solve phases. All experiments are conducted using two NASA mission software systems, i.e., the flight software of the Mars Path Finder missions (about 140 KLOC) and the Deep Space One mission (about 280 KLOC). Both are written in C and follow the same architectural and programming principles.

5.1 Analysis Time Measures

Figure 2 and 3 show the results of the evolution of the average analysis times of each phase for MPF and DS1 when the number of available processors varies. We distinguish between successive solve phases because the input data at each iteration are different. Fig 4 gives a synthetic view of these times on a graph plot. These number are averages over

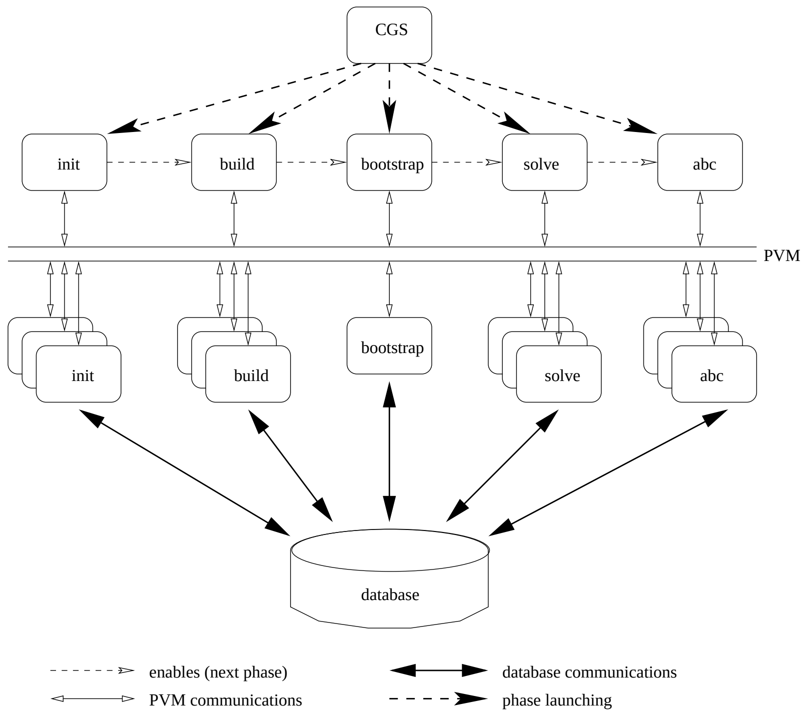


Figure 1: Architecture of C Global Surveyor

several measurements. We sometimes noticed a significant variation between trials which can be imputed to the network load at that moment, since we do not have a dedicated cluster of machines for running our experiments. Note that the differences between execution times for the bootstrap phase are not relevant since this phase is purely sequential.

The main conclusion is that contrarily to our expectations, parallelizing the algorithms does not bring a substantial payoff. Four CPUs seems to be the threshold beyond which the communication cost counterbalances the parallelization benefits. The execution times consistently decrease for all parallel phases except for the array-bound check. The explanation is that each slave process for the array-bound checking performs very simple computations over a large amount of data (the numerical invariants associated to all memory accesses in a C file), hence the execution time is dominated by the I/O with the database. This phase should definitely be made sequential like the bootstrap.

# solves	*MPF*		
	total checks	*warnings*	*precision*
1	37044	13248	64%
2	37044	9216	75%
3	37044	9216	75%

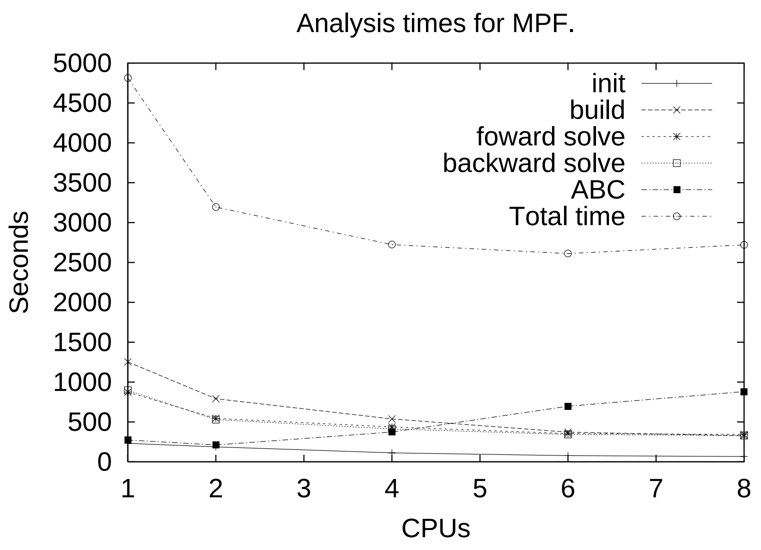
# solves	*DS1*		
	total checks	*warnings*	*precision*
1	72152	18878	74%
2	72152	15103	79%
3	72152	15103	79%

Figure 5: Evolution of the precision after successive pairs of backward-forward solve phases for MPF and DS1

5.2 Precision Measures

First, we study the precision in terms of ABC checks. All runs have been performed with context-sensitivity enabled. We count the number of ABC checks performed and compute the percentage of these checks that are not warnings (i.e. array-bound checks that could have been decided by CGS), which provides us with a measure of the precision of the analysis. We display the results in Fig. 5. Note that we group the solve phases by pairs backward-forward, since a backward interprocedural propagation which computes function transformers is of no use if there is no following forward propagation phase that uses the transformers to analyze function calls more precisely. Two passes seem to be the optimal configuration.

We also study the precision in terms of the number of points-to relations in the abstract heap computed for the program. As described in Sect. 2, each points-to relation carries three numerical invariants representing the offsets from the pointer and into the pointee, as well as the size of the memory block being pointed to. In Fig. 6 we display the evolution of the number of points-to relations after successive pairs of backward-forward solve phases for MPF. We also show the number of imprecise numerical invariants (i.e. intervals which have one of their bounds equal to $\pm\infty$) for the pointer/pointee/size information respectively. The last column represents the number of alias relations with an imprecise numerical invariant for either the pointer, or the pointee, or the size of the pointed memory block.



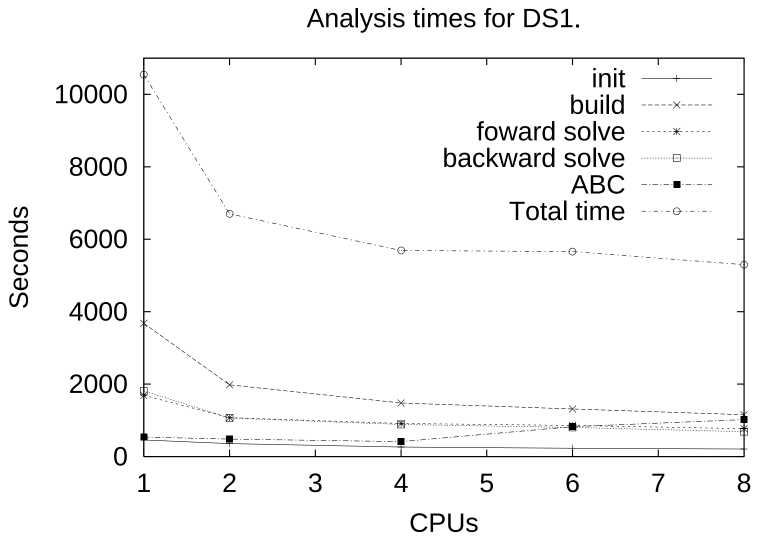


Figure 4: Average analysis times per phase and total time for MPF and DS1

# solves	*relations*	*pointer*	*pointee*	*size*	*any*
1	306	23	71	72	111
2	306	23	48	51	90
3	306	23	47	43	89

Figure 6: Evolution of the points-to relations after successive pairs of backward-forward solve phases for MPF

We notice that the number of points-to relations is constant. The major improvement concerns the numerical invariants. Although the points-to table may seem very small compared to the size of the code, the points-to relations recorded there are pervasively used throughout the code. During the development of the tool we noticed that improving the precision of few critical entries in this table could resolve thousands of checks at once.

6. RELATED WORK

There are two bodies of work that are directly related to our work. The first one is the commercial tool PolySpace C Verifier [22]. At the time of writing, the tool is available in three versions: C, C++ and Ada. We do not have any information about the C++ version. The Ada version seems to scale quite well, even though we do not have any practical experience with it. The C version however does not really scale. Our experiments, using PolySpace C Verifier, on MPF and DS1 showed that the tool could only process the code in chunks no bigger than 40 KLOC. Still, PolySpace Verifier was useful and found quite a few bugs (mainly uninitialized variables, out-of-bound array accesses, and overflows). Unfortunately, it also produced a large amount of warnings which deters developers.

The second body of work precisely addresses the problem of generating too many warnings. In [3] the authors describe a static analyzer (also based on abstract interpretation) that can analyze 75,000 lines of C code in a couple of hours with a high level of precision (11 false alarms on the code used for their experiment). Like for CGS, the authors specialized their algorithms for a family of software with the following characteristics: many global and static variables, no recursive functions nor gotos, and simple data structures. Furthermore, the authors mentioned than the alias information is trivial in the code they analyze.

There are many analyses that can now scale to large programs [24, 2, 1, 14, 18], but none of those offer the level of precision that can meet our requirements. For example, none of those analyses can track offsets (in arrays or complex data structures) with sufficient precision. Moreover, all these analyses have been designed for sequential programs. More precise analyses, such as those used in shape analysis [23], exist but they fail to scale to large programs. In fact, it is extremely difficult to design an analysis that scales with high precision for any C program. However, as we demonstrate here, high precision can be achieved on large programs that share the same basic structure.

7. CONCLUSION

We have shown in this paper that the array bound checking of large C programs can be performed with a high level of precision (around 80%) in nearly the same time as compilation. The key to achieve this result is the specialization of the analysis towards a particular family of software. Most importantly, this experience emphasizes the importance of specializing the algorithms (the domain of adaptive DBMs) and dismisses the use of general solutions (parallelization). This approach has a major drawback however: developing a specialized static analyzer is a huge effort that requires an important expertise, which limits the impact of these techniques in the software industry.

CGS is currently being applied to other kinds of NASA software. It has been recently run with success on several pieces of software operating in the International Space Station. This is an interesting process that will give us information on how a specialized analyzer behaves on programs that do not belong to its primary scope. The first results show noticeable variations in the precision. However, the scalability of the tool remains remarkably intact and CGS is able to analyze small programs of 20 KLOC in few minutes.

8. REFERENCES

[1] A. Aiken and M. Fähndrich. Program analysis using mixed term and set constraints. In *Proceedings of 4th International Static Analyses Symposium (SAS'97)*, 1997.

[2] L. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, 1994.

[3] 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. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*, pages 196–207, San Diego, California, USA, June 7–14 2003. ACM Press.

[4] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proceedings of the International Conference on Formal Methods in Programming and their Applications*, volume 735 of *Lecture Notes in Computer Science*, pages 128–141. Springer Verlag, 1993.

[5] G. Brat and R. Klemm. Static analysis of the mars exploration rover flight software. In *Proceedings of the First International Space Mission Challenges for Information Technology*, pages 321–326, 2003.

[6] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.

[7] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Symposium on Principles of Programming Languages*, pages 238–353, 1977.

[8] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.

[9] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992.

[10] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[11] P. Cousot and R. Cousot. Modular static program analysis, invited paper. In R. Horspool, editor, *Proceedings of the Eleventh International Conference on Compiler Construction (CC 2002)*, pages 159–178, Grenoble, France, April 6—14 2002. LNCS 2304, Springer, Berlin.

[12] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.

[13] M. Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation*, pages 35–46. ACM Press, 2000.

[14] M. Das, B. Liblit, M. Fähndrich, and J. Rehof. Estimating the impact of scalable pointer analysis on optimization. In *Proceedings of 8th International Static Analyses Symposium (SAS'01)*, pages 260–278, 2001.

[15] Edison Design Group. `http://www.edg.com`.

[16] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *Pvm 3 User's Guide And Reference Manual.* MIT Press, 1994.

[17] Green Hills Software. `http://www.ghs.com`.

[18] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of c code in a second. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 254–263, 2001.

[19] A. Milanova, A. Rountev, and B. G. Ryder. Precise and efficient call graph construction for c programs with function pointers. *Journal of Automated Software Engineering*, 2004.

[20] A. Miné. A new numerical abstract domain based on difference-bound matrices. In *Proceedings of the 2nd Symposium PADO'2001*, volume LNCS 2053, pages 155–172, 2001.

[21] A. Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001.

[22] PolySpace Technologies. `http://www.polyspace.com`.

[23] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis using 3-valued logic. In *Proceedings of Symposium on Principles of Programming Languages*, 1999.

[24] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of theACM Conference on Principles of Progamming Languages*, 1996.

[25] The PostgreSQL Global Development Group. `http://www.postgresql.org`.

[26] A. Venet. A scalable nonuniform pointer analysis for embedded programs. Submitted to publication.

[27] A. Venet. Abstract cofibered domains: Application to the alias analysis of untyped programs. In *Proceedings of SAS'96*, volume 1145 of *Lecture Notes in Computer Science*, pages 266–382. Springer Verlag, 1996.

[28] A. Venet. Automatic analysis of pointer aliasing for untyped programs. *Science of Computer Programming*, 35(2):223–248, 1999.

[29] A. Venet. Nonuniform alias analysis of recursive data structures and arrays. In *Proceedings of the 9th International Symposium on Static Analysis SAS'02*, volume 2477 of *Lecture Notes in Computer Science*, pages 36–51. Springer, 2002.