Electronic Communications of the EASST
Volume 33 (2010)

Proceedings of the
Fourth International Workshop on
Foundations and Techniques for
Open Source Software Certification
(OpenCert 2010)

Open Source Verification under a Cloud

Peter T. Breuer and Simon Pickin

20 pages

Guest Editors: Luis S. Barbosa, Antonio Cerone, Siraj A. Shaikh
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Open Source Verification under a Cloud

Peter T. Breuer[1] **and Simon Pickin**[2]

[1] ptb@cs.bham.ac.uk
Dept. Comp. Sci., University of Birmingham, Birmingham, UK

[2] spickin@it.uc3m.es
Dpto. Ing. Telemática, Universidad Carlos III de Madrid, Leganés (Madrid), SPAIN

Abstract: An experiment in providing volunteer cloud computing support for automated audits of open source code is described here, along with the supporting theory. Certification and the distributed and piecewise nature of the underlying verification computation are among the areas formalised in the theory part.

The eventual aim of this research is to provide a means for open source developers who seek formally backed certification for their project to run fully automated analyses on their own source code. In order to ensure that the results are not tampered with, the computation is anonymized and shared with an ad-hoc network of volunteer CPUs for incremental completion. Each individual computation is repeated many times at different sites, and sufficient accounting data is generated to allow each computation to be refuted.

Keywords: Formal Methods, Software Verification, Static Analysis, Open Source, Cloud Computing, Distributed Computation

1 Introduction

We have developed a fledgling volunteer cloud computing system for the formal verification and static analysis of large open source software code bases, and performed experiments on some millions of lines of C code with it. What has motivated this development is the vision of a future in which a formal verification problem can be sent out to a cloud of volunteer solvers somewhere on the Internet for completion. Hopefully, those supporters of an open source project who do not have the skills to provide help at first-hand will instead contribute by lending their CPU cycles to the task of certifying their favourite new release free from certain semantic errors – or detecting them if they exist. They might also contribute extra regression tests or novel verification procedures.

In this kind of framework, the bottleneck presented by the certification authority in a traditional approach is removed. Moreover, the abundance of available CPU cycles allows the calculation to be duplicated many times over for reliability, while enough intermediate results are stored for accounting processes to check the computation as may be required. Our prototype software provides a skeleton for a possible 'open certification' method, in other words. While neither the design nor the implementation is perfect and complete, it is to be hoped that the initiative stimulates better efforts and further progress in this direction.

An ‘open verification cloud’ as prototyped here physically consists of a database back-end and its servers plus volunteer clients running the bespoke verification solver software. The clients have volunteered to help perform the computations that resolve the verification problems stored on the cloud’s database. The cloud-computation nature of the process is manifested in the fact that no client knows about the other clients currently helping the computation and none knows where the servers are physically located.

The code treated in the work reported here is ANSI C [ANSI89, ISO99] with embedded assembler, and no significant restrictions. There is no inherent limitation to a particular language, however. It is of course universally realized that (unrestricted) C is an inherently intractable candidate for verification because of its indirections via pointers and other infelicitous language features, and those obstacles are overcome in this approach by using deliberately approximate (but sound) verification logic [BG04, BP06a, BP06b].

1.1 Context and related work

The verification technology used in the work reported here falls in the class of ‘lightweight’ verification technologies. It is based at the top level on a symbolic programming logic [BP06b] and at the very bottom level on decision procedures using mixed integer linear programming implemented using the GNU Linear Programming Kit (GLPK). The GLPK is intended for solving large-scale linear programming, mixed integer programming, and other related problems. It is a set of routines itself written in ANSI C and organised as a library. It is available as part of the GNU Project and is released under the GNU General Public License [Bob01, Gou99]. That is particularly appropriate here because the principal target for our technology has historically been the open source C code of the Linux operating system kernel (see for example [BG04]).

Other lightweight verification technologies in the same class include Splint [EL02] (derived from Larch [GH91]), also ESC/Java and Spec# [BRLS04]. All these tools make some sacrifices in the area of completeness or precision in order to be useful on the undecorated original source codes, and some require expert annotations to be added to the source. And while the C language is always a particularly difficult target for such technologies, some notable attempts at it have been made.

David Wagner and collaborators in particular have been active in the area (see for example [JW04], where Linux user space and kernel space memory pointers are given different types, so that their use can be distinguished, and [WFBA00], where C strings are abstracted to a minimal and maximal length pair and operations on them abstracted to produce linear constraints on these numbers). That research group often uses model-checking [CES86].

Their approach in [WFBA00, JW04] makes use of both model-checking and abstract interpretation [CC77] (abstraction is used in general in order to ‘airbrush out’ the more unsavoury aspects of C from the formal view of it), and therefore contrasts with contributions like Jeffrey Foster’s work with CQual [FTA02], which seek to extend the type system of C in a more controllable direction. In particular, CQual has been used to detect “spinlock-under-spinlock”, a sub-case of one of the analyses routinely performed by the tools used in the experiment reported here.

The SLAM project [BR02] originating at Microsoft also analyses C programs using a mixture of model-checking, abstract interpretation and deduction. That technology is intrinsically an order or more of magnitude slower than the basic technology used in the experiment described here, but it also works by creating an abstraction of the program code, and it also generates intermediate state descriptions mechanically.

The Coverity checker [ECCH00] has also come to be used in the context of the Linux kernel source code. Coverity is a commercial tool based on an user-extensible version (a *meta-compiler*) of the GNU C compiler, *gcc* [Gri02]. Coverity itself is proprietary, and its innards are not accessible to review, but it may be guessed that the staff of the company can configure into the compiler framework any finite state machine-based computation for the purposes of a custom analysis that they have in mind. It is a less abstract technological solution than the one used here, but shares with it the characteristic of customizability.

Efforts to distribute verification computations to a large number of solvers organised in a well-defined topology are made regularly – see [Abu09], for example – and it is a recognised conference topic. Researchers have particularly sought to distribute model-checking problems onto grid-based machinery. Holzmann defines the notion of ‘swarm verification’ to describe the technique [HJG08a], adapting the SPIN [Hol03] model-checking tool to the paradigm. In passing, it may be noted that the verification technology used here seems to be part of a recent trend observed by Holzmann in [HJG08b] towards verification of an abstraction of the actual code rather than of a design model. However, the work reported here aims to accommodate the lower performance targets obtainable from zero-cost volunteer CPU cycles available out on the Internet.

Existing infra-structure projects support so-called ‘volunteer computing’ -type projects. See for example the BOINC software [And04, AKW05] from Berkeley. It is not clear at the time of writing if that software would have been a significant aid to our exploratory project, because BOINC clients expect a single data file and return a single result file to the database server, rather than engaging in a substantially continuous interchange, as is the case here. Nevertheless, it may in the future be very helpful in the organisation of the architecture in a full-scale project, particularly in terms of the organisation of the permissions for access and the classification of the provenance and reliability of the data returned.

Peter Lee [NL96] has approached the problem of automatically checking the trustworthiness of machine code to be executed by an operating system. The idea in *proof carrying* approaches is that incoming code snippets carry a proof that a desired security property is satisfied, and the operating system automatically checks the syntactic relationship of the machine code to the proof. Our approach is to check the source code instead and is designed to handle large code bases such as the Linux kernel in reasonable time.

1.2 Contents

This article is organised as follows. Section 2 formally describes the process of certification from the top down. After describing certification properties, the section describes in more detail the process of analysis that produces the certification here, showing in particular how the calculation is adapted to the exigencies of a part-time volunteer cloud-computing context. Section 3 succinctly presents the programming logic used in order to provide a self-contained account of the technology here, and readers may wish to skip that section if they are not interested in formal logic. Section 4 describes an experiment performed on about a million lines of C source code.

That experiment was previously conducted using monolithic analysis tools [BP06a] and it has been repeated in the volunteer cloud computing trial [BP09].

2 Certification

In this section a global view of the certification process is set out and related to the procedure implemented.

2.1 Certification in the abstract

This section describes formally what certification means as implemented in the prototype project.

Three characteristics stand out. Firstly, certification is a process and it produces both a result and a certificate. Secondly, the certificate has the property that it can be checked to have been generated by following the purported process applied to the purported source code, generating the purported result. Thirdly, the result apports certain guarantees about the code.

Consider then the certification process in the abstract. An automated procedure M takes a software code base C and, in the presence of a list L of known defects, produces a certificate X that says that the list is complete. That is, the process takes code C and (sometimes – the alternative is that the certification process fails) produces a certificate X:

$$CM = X$$

Moreover, if L_d is the sub-list of L of defects of kind d, and we write d_p to mean that there is a defect of kind d at point p in the code, then L_d contains all the points p in the code at which a defect of kind d arises. That is:

$$\{p \in C \mid d_p\} \subseteq L_d$$

Putting those two together, one gets a fundamental description of what certification means:

$$X = CM \;\Rightarrow\; \forall p \in C - L_d : \neg d_p \tag{1}$$

I.e. code that has more defects than stated does not get a certificate.

2.2 What is a defect?

In our implementation, a defect d_p is defined by a condition expressed in symbolic logic as $D_p(\mathbf{x})$ that is deduced to be possibly reachable after p. That is, logical analysis deduces a post-condition

$$\ldots\ p\ \{\ \mathrm{post}_p\ \}$$

for p and checking the formula using a model-based technique shows there is a non-empty intersection of the post-condition with $D_p(\mathbf{x})$. That is:

$$d_p \;\Leftrightarrow\; \exists \mathbf{x}.\, D_p(\mathbf{x}) \wedge \mathrm{post}_p \tag{2}$$

for some values of the logical variables $\mathbf{x}$. An example follows in Subsection 2.3 immediately below.

2.3 Example

An example of an interesting defect condition is

$$D_p^{\text{ex}} = \begin{cases} x > 1, & p \text{ a lock call} \\ x < 0, & p \text{ an unlock call} \\ \text{false}, & \text{otherwise} \end{cases} \tag{3}$$

where x is a logical (i.e., non-program) variable which counts the number of stacked locks taken in the program. The variable x is incremented by lock calls and decremented by unlock calls. The pre-/post-condition logic describing x for the analysis is:

$$\begin{array}{l} \{\phi[x+1/x]\}\ \textbf{lock( )}\ \{\phi\} \\ \{\phi[x-1/x]\}\ \textbf{unlock( )}\ \{\phi\} \end{array} \tag{4}$$

Where this particular defect condition checks out as feasible in the sense of (2), it indicates either that (a) a lock might have been taken twice by that point without a release between the two takes; or (b) a lock may have been released twice by that point without a lock attempt between. These defects d_p^{ex} can by definition (3) only be detected at the sites of a lock or unlock call p. Certification in this case means that the code C has been scanned and defects d_p^{ex} have been ruled out. That is, no lock can be taken twice in a row, nor unlocked twice in a row, without an unlock, respectively a lock, operation occurring between the two.

2.4 False positives

Note that there may be codes C which are flagged as having a defect in the sense of (2) but which are nevertheless semantically correct ('false positives'), in the sense of never in practice triggering the condition $D_p(\mathbf{x})$.

That is the rationale for in practice maintaining a list L_d of detected defect sites – they have individually to be signed off by the developer as 'false positive' or 'noted for correction in the next release', or 'noted but no solution yet'. The certificate certifies that it is unequivocally the case that the defined defect cannot arise anywhere other than the sites listed.

False positives generally fall into two classes. In the first class, a guard condition such as $y^2 < 0$ cannot in practice be breached, but the analysing logic does not know that, and explores a factually impossible code trace as though it were possible. That kind of semantic 'inexactness' is a result of the deliberately approximating nature of the symbolic logic used in any real life analysis. The analysing logic has to be less exact ('more alarmist') than reality or the computation would never finish in practice.

A typical instance of the second class of false positive arises naturally in the context of the example above in Subsection 2.3. It occurs when two *different* locks are taken in sequence in the code, without an unlock between them. A defect will be detected. The fault here is purely a definitional one. The situation is factually harmless in itself, but it is captured by the defect definition. The problem may be said to be rooted either in the poverty of the analysis language – different counts for different locks may be difficult to define – or in the poverty of the analysis logic – one may not be able to reliably distinguish references to different locks in C. The latter is the case here. Different pointers may point to the same underlying lock, and the same pointer may point to different locks at different times.

2.5 Accountability

It is important for a certification procedure that it can be checked that the certificate X produced relates the certified code C and the method M used to certify it. That is, there is a checking procedure K such that

$$K(X,C,M) = \begin{cases} \text{true}, & \text{if } X = CM \\ \text{false}, & \text{otherwise} \end{cases} \tag{5}$$

How is that guaranteed?

The answer is, in our procedure, via digital signatures, namely, the following: (a) $\sigma(T^\ddagger)$ is generated from a printout of intermediate results $T^\ddagger$ of the analysis in M; (b) $\sigma(\mathscr{A})$ is generated from the short ASCII file that configures the analysis; (c) $\sigma(\mathscr{H})$ is generated from the ASCII file that expresses the defect condition(s) being scanned for; (d) $\sigma(L)$ is generated for the list of allowed defect exceptions L; (e) $\sigma(C)$ is generated for the code C; (f) $\sigma(\mathscr{P})$ is generated from the file that configures the code parse $\mathscr{P}$. Those digital signatures comprise X, as will be detailed in the following paragraphs.

Then, provided the code developer holds on to a copy of the intermediate results, a copy of the analysis method configuration, and a copy of the original code, then any part of the computation via M can be repeated at will for the benefit of anyone that doubts it. That is the procedure K, modulo checking the digital signatures to confirm the veracity of the three components. That is to say

$$K(X,C,M) \ \Leftrightarrow \ CM = X$$

as required in (5). That means that

$$K(X,C,M) \ \Leftrightarrow \ \forall p \in C : \neg d_p \vee p \in L_d$$

according to (1). The important idea here is that a part of the calculation can be repeated as needed in order to check the result, and that in order to be sure that the repeated calculation starts from the right place (and finishes in the right place) the digital signatures in the certificate are necessary – as is the data signed, but that has to be stored separately. It is not present in the certificate. Where the data is kept is a separate question.

To explain how the calculation can be reconstructed when required, the calculation needs first to be described in more detail. The certification method M consists first of a parse $\mathscr{P}$ to give a syntax tree T:

$$T = C\mathscr{P} \tag{6}$$

Next an analysis $\mathscr{A}$ is applied to the tree T to decorate it with symbolic logic expressions, giving the decorated tree $T^\dagger$:

$$T^\dagger = T\mathscr{A} \tag{7}$$

Then a checker $\mathscr{H}$ is applied which further decorates the tree with evaluations of the logic to see if defects are feasible:

$$T^\ddagger = T^\dagger\mathscr{H} \tag{8}$$

The list of sites p within the code C at which defects d_p are detected is what is basically of interest to developers and consumers alike, and it is supposed to be covered by the list L of knowns.

$$L_d \supseteq \{p \in C : d \text{ decoration of } T^{\ddagger} \text{ at } p \text{ is not false}\} \tag{9}$$

The certificate X consists exactly of the digital signatures of the code, (printed out) tree decorations, and the configuration used for the parser, for the analysis and for the checker, and the list of known defects:

$$X = (\sigma(C), \sigma(\mathscr{P}), \sigma(T^{\ddagger}), \sigma(\mathscr{A}), \sigma(\mathscr{H}), \sigma(L)) \tag{10}$$

Every step of this procedure can be repeated unambiguously. For example, to get to $T^{\ddagger}$, one needs to repeat at least the step (8). That starts from $T^{\dagger}$. But $T^{\dagger}$ is just $T^{\ddagger}$ with some of the decoration dropped. So it can be unambiguously obtained from $T^{\ddagger}$, which is signed. The configuration for $\mathscr{H}$ is signed and available, and so $\mathscr{H}$ can be applied unambiguously to check the derivation of $T^{\ddagger}$ from $T^{\dagger}$.

2.6 Analysis and evaluation

The analysis procedure $\mathscr{A}$ is organised in detail according to the structure of the code. It generates a pre-/post-condition pair for each program fragment p:

$$\{\text{pre}_p\}\ p\ \{\text{post}_p\}$$

The pair is computed from the results for the component fragments $p_i \in p : p = \underset{i}{P}(p_i)$ where P is the constructor (`if`, `while`, etc.) that produces p from the p_i. That is

$$(\text{pre}_p, \text{post}_p) = \underset{i}{[P]}(\text{pre}_{p_i}, \text{post}_{p_i}) \tag{11}$$
$$\text{where } \{\text{pre}_{p_i}\}\ p_i\ \{\text{post}_{p_i}\} \text{ for } p_i \in p$$

and $[P]$ is the appropriate generator of symbolic logic. It is specified for the source language (here C) being treated in the configuration file for the analysis. 'Appropriately' here means that the logic is sound with respect to the semantics of the language, in that for each pre-/post-condition pair generated by the above formula (11):

$$\text{pre}_p \ \Rightarrow\ \mathbf{wp}[p](\text{post}_p) \tag{12}$$

where **wp** is the semantic *weakest precondition* constructor for the language.

That (12) is an implication and not necessarily an equivalence means that the symbolic logic generated by the scheme (11) is *approximate* (but sound, according to (12)). That gives rise to the name *symbolic approximation* [BP06b] for the general technique. In practice, a slightly different customized approximate symbolic logic is used for each defect analysis.

Note that some complexity reduction *is* performed by our tools via lightweight automatic theorem-proving techniques at the stage of producing the tree $T^{\dagger}$ with the symbolic logic annotations. For example, a formula of the form

$$p \wedge q$$

will be reduced to q if $p \to q$ is proved on the fly as the formula is generated. Similarly for $p \vee q$. That has proven very effective in reducing complexity. What our tools are not good at is reducing formulae of the general shape $\bigvee_i \bigwedge_j q_{ij}$ to a simpler expression p when there is one, such as in the case of $p \wedge q \vee p \wedge \neg q$. The inadvertent and unrecognised splintering of simple logical expressions into multiple complex cases in this style is the most significant source of the computational explosions that are occasionally encountered during processing. In principle the situation could be detected and repaired at the checking stage of the process when $T^\ddagger$ is generated (all the atomic propositional forms here are linear inequalities and one could detect when dropping one failed to relax the problem), but that is not done, because the extra computation is usually prohibitively expensive and apparently only rarely productive in practice.

In the final phase that produces $T^\ddagger$, the volunteer clients in the cloud apply a modelling technique to decide whether

$$\mathrm{post}_p \wedge D_p(\mathbf{x})$$

is satisfiable at any node p of the abstract syntax tree. Since all questions of satisfiability for the predicates in our logic can be reduced to questions of the feasibility of systems of linear inequalities in integer variables, the evaluation is performed using mixed linear integer programming. The implementation uses only open source libraries, principally GNU's Linear Programming Kit (GLPK). A non-negative answer to the question asked by the evaluation procedure indicates a *possible* defect d_p.

2.7 The cloud computation

The analysis $\mathscr{A}$ and evaluation calculations $\mathscr{H}$ are incremental, stateless, and can be broken off and restarted from the break-off point, as well as repeated either partially or wholly. That is the basis for performing the computation via a cloud and the following paragraphs describe the properties that permit that implementation in formal terms.

Let the constructs p (the nodes and leaves of the abstract syntax tree T produced by the parse) that appear in the code C be $p = \underset{i}{P}(p_i)$ for a syntactic constructor P and components p_i. The constructions define a *dependency* pre-order:

$$p = \underset{i}{P}(p_i) \;\Leftrightarrow\; p_i < p \tag{13}$$

which extends uniquely to a minimal partial order via transitivity. In the partial order, one code construct 'depends on' (is greater than) another if the second is a component or subcomponent, etc., of the first. For example, `if(x<0){ x++; y++; }` depends on the component `x++; y++` and on its component `x++`.

The operations $\mathscr{A}$ and $\mathscr{H}$ can then be split up into fragments $\mathscr{A}_p$ and $\mathscr{H}_p$ at $p \in C$ as follows:

$$\mathscr{A} = \underset{p}{\circ}\mathscr{A}_p \tag{14}$$

$$\mathscr{H} = \underset{p}{\circ}\mathscr{H}_p \tag{15}$$

where the order of the compositions is constrained only by the dependencies $p_i < p$. Formally, operations on different parts of the tree can be performed in any order:

$$\mathscr{A}_{p_1} \circ \mathscr{A}_{p_2} = \mathscr{A}_{p_2} \circ \mathscr{A}_{p_1} \tag{16}$$
$$\mathscr{H}_{p_1} \circ \mathscr{H}_{p_2} = \mathscr{H}_{p_2} \circ \mathscr{H}_{p_1} \tag{17}$$

where $p_1 \not\leq p_2$ and $p_2 \not\leq p_1$ in the dependency relationship. Also, since $\mathscr{A}$ and $\mathscr{H}$ work on different decorative features on the tree

$$\mathscr{A}_{p_1} \circ \mathscr{H}_{p_2} = \mathscr{H}_{p_2} \circ \mathscr{A}_{p_1} \tag{18}$$

whenever $p_1 \neq p_2$. When $p_1 = p_2$ then $\mathscr{H}$ requires the decoration produced by $\mathscr{A}$ first.

In practice, the computation of the symbolic logic forms and their evaluation is performed at the same time, because the former is usually a computationally cheap task relative to the latter. That is, the computation

$$\mathscr{A} \circ \mathscr{H} = \underset{p}{\circ}(\mathscr{A}_p \circ \mathscr{H}_p) \tag{19}$$

is performed. (16, 17, 18) justify the reordering of the components in (19).

That the computation can be broken off and restarted means only that (19) can be further reordered via (16, 17, 18) as

$$\mathscr{A} \circ \mathscr{H} = \underset{p\in P}{\circ}(\mathscr{A}_p \circ \mathscr{H}_p) = \underset{p\in P_1}{\circ}(\mathscr{A}_p \circ \mathscr{H}_p) \underset{p\in P_2}{\circ}(\mathscr{A}_p \circ \mathscr{H}_p) \tag{20}$$

where P_1, P_2 is a partition of the full set of code fragments $P = P_1 \uplus P_2$ such that

$$p_1 \in P_1 \wedge p_2 \in P_2 \;\Rightarrow\; p_2 \not\leq p_1 \tag{21}$$

I.e., P_1 already contains all the pre-dependencies $p_2 < p_1$ for any $p_1 \in P_1$. P_1 is the set of code fragments that have been completely analyzed at the time of the break, and P_2 is the remainder at that time.

Moreover, the computation can be broken off and re-started any number of times. That is, the equation (20) may be extended to match with any partitioning $P = P_1 \uplus \cdots \uplus P_n$ that respects the dependency order, as in (21). P_1 is the set of code fragments completely analysed at the time of the first break, $P_1 \uplus P_2$ the set completely analysed at the time of the second break, and so on. The enabling conditions are (16, 17, 18).

3 Logic

Readers uninterested in formal logic may wish to skip this section, which is included to provide a self-contained account here. It is not needed by what follows after.

The program logic used to generate the assertions which decorate the syntax tree $T^\dagger$ from (7) is called NRBG, for ‘normal, return, break, goto’, the principle kinds of program flow treated. In a program there is the normal program flow, which passes from the beginning of a statement through to its (normal) end, and there are exceptional flows, the break, return and goto flows (and also others in other languages), which exit a statement before it ends normally. The logic considers the interaction of these flows though each code construct.

For example, the rule for sequential statements states that either $a;b$ may terminate normally with condition r or it may terminate exceptionally with condition x. On the way to doing so, a may either terminate normally with condition q and b continue from q to the required termination conditions, or else a may terminate exceptionally with condition x right away. That is:

$$\frac{\{p\}\ a\ \{Nq \vee \mathscr{E}x\} \quad \{q\}\ b\ \{Nr \vee \mathscr{E}x\}}{\{p\}\ a;b\ \{Nr \vee \mathscr{E}x\}}$$

where $\mathscr{N}$ stands for 'normal' and $\mathscr{E}$ stands for any of the R ('return'), B ('break'), G_l ('goto') exceptional flows, where l is not a label defined in a or b.

The logic has been presented and explained many times over the years. See for example [BP06a]. The presentation given here is innovative in that it introduces the N, R, B, G_l as modal operators. The earlier presentations used a set of interacting logics N, R, B, G. The advantage of the new presentation is that the number of logical rules falls to about seven from about twenty.

The rule for a do-forever loop says that breaking from the body of the loop with condition q is the same as terminating the loop normally with q. That is:

$$\frac{\{p\}\ a\ \{Bq \vee Np \vee \mathscr{E}x\}}{\{p\}\ \texttt{while}(\texttt{true})\ a\ \{Nq \vee \mathscr{E}x\}}$$

where $\mathscr{E}$ stands for any of R, G_l, where l is not a label defined in a. The rule also captures the idea that exiting exceptionally in another way than through break (that is, with either return or goto) from the body of the loop with condition x means exiting the whole loop with condition x too. The normal termination condition p for the body of the loop that appears in the rule is a *fixpoint*, and finding a useful fixpoint (lower than 'true') in practice is a non-trivial feat of leger-de-main.

Exceptional modal conditions $R\,p$, $B\,p$, $G_l\,p$ are generated uniquely by the corresponding statements, `return`, `break` and `goto` respectively:

$$\frac{}{\{p\}\ \texttt{return}\ \{Rp\}} \qquad \frac{}{\{p\}\ \texttt{break}\ \{Bp\}} \qquad \frac{}{\{p\}\ \texttt{goto}\ l\ \{G_l\,p\}}$$

The rule for conditionals is, unsurprisingly:

$$\frac{\{p \wedge c\}\ a\ \{Nq \vee \mathscr{E}x\} \quad \{p \wedge \neg c\}\ b\ \{Nq \vee \mathscr{E}x\}}{\{p\}\ \texttt{if}(\texttt{c})\ a\ \texttt{else}\ b\ \{Nq \vee \mathscr{E}x\}}$$

where $\mathscr{E}$ stands for any of R, B, G_l, where l is not a label defined in a or b.

A suitable assignment rule is always:

$$\frac{}{\{q[e/x]\}\ x = e\ \{Nq\}}$$

but in practice some weaker rule is usually implemented, with special cases that depend on the form of the expression e. From the point of view of the correctness of the logic, it does not matter what weaker rule is implemented because it will be sound. The practical effect of a weaker implementation is eventually to generate more 'false positive' alerts for defects than would otherwise have been the case. For example, non-linear update expressions are typically described in practice by very approximate logic such as:

$$\frac{}{\{x>0\wedge|x||y|<2^{31}\}\ x=x*y\ \{\mathrm{sign}(y)=\mathrm{sign}(x)\}}$$

(the preconditions avoid overflow).

The rule for a labelled statement $l:b$ says that an initial condition p is required that is the same as the exit condition p from all the `goto` l statements within b. I.e., p is a fixpoint:

$$\frac{\{p\}\ b\ \{Nr\vee\mathscr{E}x\vee G_l\,p\}}{\{p\}\ l:\ b\ \{Nr\vee\mathscr{E}x\}}$$

Compare the rule for while forever loops.

A derived rule for labelled statements deals with the more general situation where a label occurs in the middle of a sequence of statements $a;l:b$, rather than at the beginning. There it is the case that the entry condition q for $l:b$ must not only be the normal exit condition from a, but also the condition that arises from the 'forwards pointing' gotos within a, as well as the 'backwards pointing' gotos in b:

$$\frac{\{p\}\ a\ \{G_l\,q\vee Nq\vee\mathscr{E}x\}\qquad\{q\}\ b\ \{Nr\vee\mathscr{E}x\vee G_l\,q\}}{\{p\}\ a;l:b\ \{Nr\vee\mathscr{E}x\}}$$

where $\mathscr{E}$ stands for any of R, B or $G_{l'}$ where l' is not a label defined in a or b.

There is a more convenient way to deal with the goto computations, however. It consists of loading the rules with prior 'assumptions' $G_l\,p_l$ (written to the left of a $\triangleright$ in Gentzen style) about the exit condition p_l that will be imposed by the `goto` l statements encountered within the program. The initial estimates are modified upwards by the conditions p found at the sites where the gotos are located in the program. The initial guess p_l needs to be loosened to $p\vee p_l$, and so on round and round until a fixpoint is found. A goto fixpoint achieved in practice is not usually the least fixpoint, but it is generally a useful and nontrivial one.

$$\frac{p\ \Rightarrow\ p_l}{G_l\,p_l\ \triangleright\ \{p\}\ \mathtt{goto}\ l\ \{G_l\,p\}}$$

The fixpoint p_l is available as an entry condition at the point in the code where the label l is sited:

$$\frac{G_l\,p_l\ \triangleright\ \{p_l\}\ a\ \{Nr\vee\mathscr{E}x\vee G_l\,p_l\}}{\triangleright\ \{p_l\}\ l:\ a\ \{Nr\vee\mathscr{E}x\}}$$

The model underlying the logic is of individual states s which assign values to the program variables, and links between them that are 'coloured' N, R, B or G_l according to whether the transition is as a result of respectively a $\underline{\mathrm{n}}$ormal program termination, hitting a $\underline{\mathrm{r}}$eturn statement, hitting a $\underline{\mathrm{b}}$reak statement, or hitting a $\underline{\mathrm{g}}$oto. The following diagram is of a R-coloured ('return coloured') transition from state s_1 to state s_2:

$$\begin{tikzcd} s_1\bigcirc \arrow[r, dash, "R"] & \bigcirc s_2 \end{tikzcd}$$

Each state has only one exit in the present (deterministic) setting, but there may be many entries to any state.

The semantics of the N, R, B, G_l operators is that, for example, a modal statement like Rp holds at the *pair* of a state s_2 *and* an arc e *entering* the state. For example, Rp holds at (e,s_2) if p holds at s_2 and e is coloured with R. In general:

$$(e,s) \models \mathscr{E}p \quad \Leftrightarrow \quad s \models p \wedge e \text{ is coloured by } \mathscr{E}$$

for $\mathscr{E}$ any of N, R, B, G_l.

An atomic programming language statement a causes a change from a state s_1 to a state s_2 via a link e that is of a ‘colour’ that is normally N, but is exceptionally R, B or G_l, depending as the statement executed in a is a return, break or goto respectively.

Suppose that for the (non-atomic) statement a, the sequence

$$\underset{s_0}{\bigcirc}\overset{e_1}{\text{---}}\underset{s_1}{\bigcirc}\overset{e_2}{\text{---}}\dots\overset{e_n}{\text{---}}\underset{s_n}{\bigcirc}$$

is a sequence of states run through by the execution of a. Then $\{p\}\ a\ \{q\}$ means

$$\begin{aligned}&\{p\}\ a\ \{q\} \ \Leftrightarrow \\ &\quad \forall s_0, e_1, s_1, \dots, e_n, s_n.\ p(s_0) \ \Rightarrow\ (e_n, s_n) \models q\end{aligned} \tag{22}$$

By convention, not specifying a ‘colour’ means that colours are ignored, i.e.:

$$p \ \Leftrightarrow\ Np \vee Rp \vee Bp \vee G_l p \vee \dots \tag{23}$$

for all possible labels l in the program. Then (22) can more symmetrically be written as

$$\forall e_0, s_0, e_1, s_1, \dots, e_n, s_n.\ (e_0, s_0) \models p \ \Rightarrow\ (e_n, s_n) \models q$$

Making (23) work requires a few axioms for the modal operators. Firstly, repetition of modal ‘colouring’ operators has no effect:

$$\mathscr{E}p \ \Leftrightarrow\ \mathscr{E}\mathscr{E}p \tag{24}$$

for $\mathscr{E}$ any of N, R, B, G_l. Also, an arc cannot be two colours at the same time:

$$\mathscr{E}_1 p \wedge \mathscr{E}_2 q \ \Rightarrow\ \text{false} \tag{25}$$

for $\mathscr{E}_1$, $\mathscr{E}_2$ from N, R, B, G_l and $\mathscr{E}_1 \neq \mathscr{E}_2$. Similarly

$$\mathscr{E}_1 \mathscr{E}_2 p \ \Rightarrow\ \text{false} \tag{26}$$

for $\mathscr{E}_1 \neq \mathscr{E}_2$. And colouring a (positive) formula is the same as colouring its parts:

$$\mathscr{E}(p \vee q) \ \Leftrightarrow\ \mathscr{E}p \vee \mathscr{E}q \tag{27}$$

$$\mathscr{E}(p \wedge q) \ \Leftrightarrow\ \mathscr{E}p \wedge \mathscr{E}q \tag{28}$$

for $\mathscr{E}$ from N, R, B, G_l. Together, (23), (24), (25), (26), (27), (28) mean that all modal formulae have the form

$$Np_N \vee Rp_R \vee Bp_B \vee G_l p_{G_l} \vee \dots$$

for non-modal formulae p_N, p_R, etc.

4 Implementation and experiment

We report briefly here on our experience [BP09] in converting what were originally a set of monolithic semantic analysis tools [BG04, BP06a] for C code to the service of the volunteer cloud computing approach, and the re-running on the cloud of an experiment that had previously been run locally.

The exact goal of the experiment was to solve a particular large formal verification problem via an ad-hoc distributed network of automated solvers. The task consisted of an analysis of the Linux kernel source code (written in C [ANSI89, ISO99] and assembler) to detect a particular kind of runtime deadlock in the operating system as compiled for multi-CPU 32-bit Intel (IA32) platforms. Those faults detected during the experiment are not intrinsically specific to the Intel platform, however, because 80-90% of the code is shared with and common to the 15 other major architectural types supported by the Linux kernel, and any faults found in a common section are relevant to the other platforms too. The deadlock is known as 'sleep under spinlock'. It happens when a thread of computation sets a 'spinlock' (one which another thread will wait busily for, spinning in a tight loop until it is released), but then 'sleeps' (is ejected from the CPU). Since the ejected thread is the only one that will eventually release the lock, if a thread enters the CPU meanwhile and spins waiting on the spinlock the situation is deadlocked. The CPU is occupied by a spinning thread that will do nothing except keep the only thread that will release the lock out of the CPU. The experiment detects about three such faults per million lines of code. It also simultaneously checks for other similar deadlock possibilities (notably 'spinlock under spinlock under spinlock ... '), which are detected at close to the same frequency. The average lifetime from appearance to elimination in the source code of the faults detected appears to be about six months, checking against the version histories.

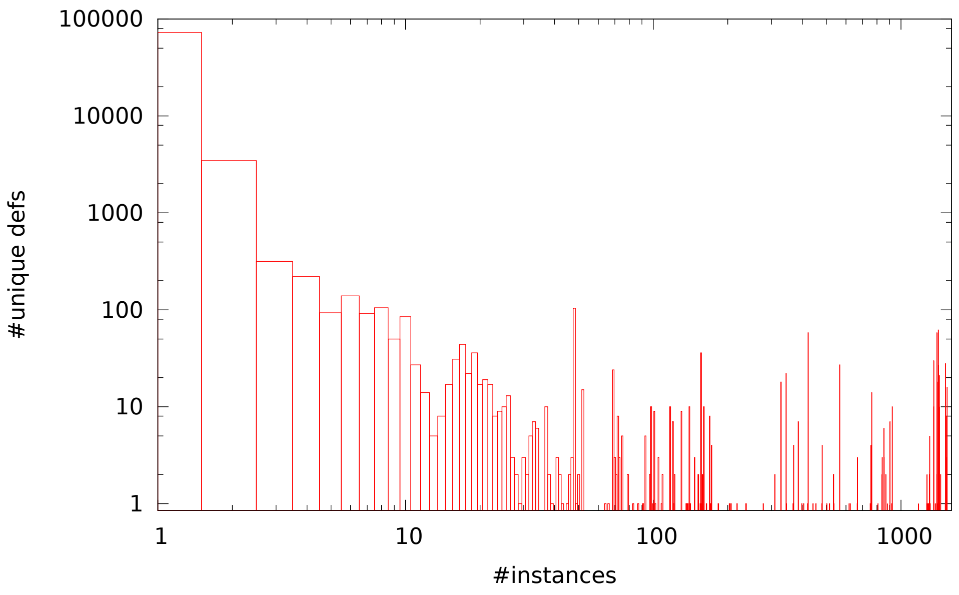
4.1 Populating the database

The first practical task for any submitter in presenting a problem to the cloud for solution consists of parsing the source code and storing the resulting syntax tree into the cloud's remote database. But it is a considerable logistical problem, and it turned out to be too difficult to treat naively.

In our experiment [BP09], a million or so lines of Linux kernel source code was offered up for analysis, and that gave rise to over ten million syntax tree nodes. Each insertion involved several relational database updates on the (postgresql [Dou05]) back-end and the acknowledgement and locking requirements slowed the transactions down to as much as a second or more across the network (the average time was a tenth of a second or so). There are only 86400 seconds in a day, and no developer is going to wait on the order of weeks to upload their problem. The efficiency might have been improved tenfold with effort, but it would have still been too slow for multi-million line source code bases.

This 'population problem' was eventually overcome by writing the parse data to a fast local non-relational data store (a GNU DBM 1.8.3 based store was used), then copying it to the remote database site in one lump, and converting it to relational database format in situ at the remote database. That got the job done in a day. It may be expected that incremental updates will comfortably handle new point releases of the source code base from there on. Attempts to use local and remote database replication pool services to upload the data in trickle mode failed.

Table 1: Top-level definitions with multiple instances ($\sum xy = 746844$)



Each stream tended to stop completely while the database was otherwise in use, and the end result was slower overall. Clearly in the future source code will have to be uploaded whole to an extra cloud service from where it can be transferred into the database from close by.

4.2 Pruning the analysis

A single analysis task downloaded for solution by a volunteer client in the cloud usually consists in practice of the analysis of a single top-level functional unit. However, it turned out in our experiment that many of the function definitions from common header files had effectively been duplicated up to thousands of times through being declared *static* and *inline*, a combination which, in C, signals local scope and context at every implantation site. See Table 1 for a count of the number of implanted definitions. At right in the table are represented the dozens of function definitions implanted into more than a thousand different sites. The number of analysis tasks was reduced tenfold overall by choosing to analyse only one representative from each class of syntactically identical functional definitions.

There is a potential problem in that the semantics of some of these apparent duplicates might have been modified unexpectedly by the differing contexts into which they were copied. It was supposed that that did not happen. The assumption was made that no two syntactically identical definitions captured identically named but different external references. This is a good assumption for well-written code, but it was not checked systematically. The numbers were certainly prohibitive - there were three quarters of a million top level function definitions in the database. Only seventy-two thousand of them corresponded to non-duplicates, and those were the ones eventually allowed to proceed to analysis.

4.3 Improving performance

Fetching data from the database in the cloud to a client as needed turned out to be far too inefficient as a general strategy. The latency of each database transaction was sufficient that the computation as a whole proceeded about one thousand times as slowly as it would have done on locally stored data (that experiment had already been tried [BP06a]).

The situation was improved firstly by avoiding downloading syntax trees (node by node) in favour of downloading the relevant source code text in one lump and re-parsing it locally on the volunteer client. The issue of generating the same database keys locally as remotely was handled by storing an elaborated version of the source decorated with extra annotations, among them the in-database key for each identifier reference (each reference appeared in the elaborated text as 'x@123456', where 'x' was the label in the original source, and '123456' the primary database key indexing the reference to 'x' at that line and column in that source code file). The primary provides enough information to generate all other keys locally too.

Secondly, a persistent cache was added on the client side just atop the database interface. The cache scored hits around the 95% level, with the corresponding order-of-magnitude-and-more speed-up.

Thirdly, the few database interactions that turned out to take minutes each – queries involving complex searches and aggregates across millions of database entries, such as calculating new priorities for the remaining work tasks after each task completion by a volunteer client – were amortised by calculating up to five hundred results ahead of time and then doling them out as needed. That implied that work task priorities in particular were never quite what they should have been according to theory, but the effect was not significant in practice.

Finally, significant reductions in the complexity of the logical formulae generated during the processing were achieved by building in automatic theorem-proving techniques to the mechanisms that generated the formulae in the first place. There is a trade-off between expending time to reduce complexity and gaining time through the reduced complexity, but there were huge gains made by the simplest reduction techniques, based essentially on automatic deduction in the symbolic logic in order to remove extraneous terms from the formulae. If the automatic deduction failed to obtain an improvement, abstract interpretation and finally mixed integer linear programming were used first to see if a reduction in complexity could possibly be achieved and then to check definitively [BP09].

4.4 Allocation and management strategy

Allocating work to volunteer clients required an allocation strategy. Naively sending out the next work task in alphabetical order would have eventually gotten all working clients stuck executing very hard tasks with no appreciable progress being made. The group of volunteer clients makes more progress overall if they complete the easy work tasks first. But which are the easy tasks? There is no definite way to tell other than by trying and seeing.

The size of analysis task taken on by volunteer clients was initially set to 'one complete functional unit', i.e. a top-level function definition. Each functional unit was initially assumed to be equally as hard to analyse as every other. Each volunteer was initially given $T_0 = 10$ minutes of CPU time (normalised to a 1GHz CPU) in which to complete the work task. If the limit was exceeded, the client abandoned the task, reported back the incompletion statistic to the cloud's database, and moved on to a different work task. The task's estimate of intrinsic difficulty was raised, as reflected by an increased timeout value $T_1 > T_0$ now associated with it.

It was intended by this means to tamp down as much as possible on the total concurrent interactions with the cloud's database. Network bandwidth is a finite bound that cannot be exceeded, and the database has a limit on the number of transactions per second it can absorb. The cache at each client served to prevent 90% of that client's database transactions from escaping onto the net but work task startup and shutdown are points where large amounts of novel information are exchanged with the cloud. Giving volunteer clients by default a relatively large work unit to chew on reduces the number of data requests flying about the network and thus in principle helps the computation overall. The downside is that clients may be given more than they can deal with, plugging progress overall. Unplugging by imposing a timeout was the simplest cure. The downside is that it implies the loss of the data accumulated by the client up to the point of abandonment.

Every time a work task was abandoned uncompleted, the estimated time required to complete it was increased by 50% (i.e., $T_{n+1} = 1.5\,T_n$), so that the next client to take it on would spend longer on it before abandoning. Moreover, tasks with a higher timeout were handed out with lower frequency (i.e., with lower priority) so that clients would tend to take the easier tasks first.

In the end, the 'hard' work tasks that took longer than the initial 10 minutes turned out to comprise only 0.5% (three hundred-odd) of the total. One might argue that the tasks handed out initially were not difficult enough since 99.5% failed to prevent their host from emitting significant noise on the network for less than ten minutes at a time.

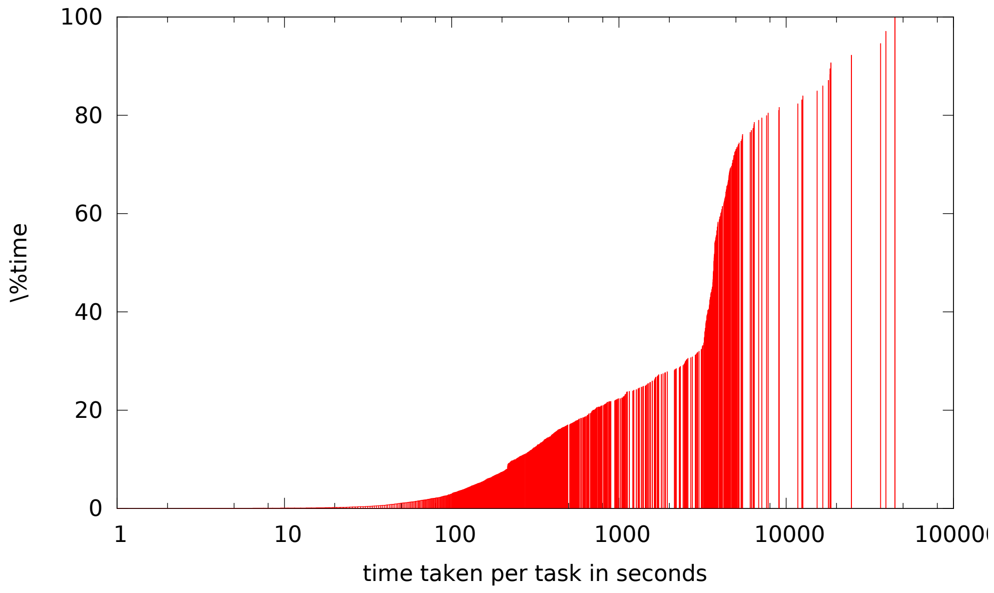
However, of the hard tasks, two thirds (about two hundred) eventually did complete in an hour or less, availing of lengthened timeouts. Abandonment wastes the earlier effort put in, but the time taken overall is still dominated by the successful final stint, so at most three hours of computation time were spent for each hour-long completion here.

Those 'very hard' work tasks that still were taking longer than an hour without completion (a hundred or so, or 0.15% of the original total) were dealt with in accord with the theory developed in Section 2. That is to say, the incremental progress in the client dealing with them was checkpointed to the cloud's database every minute. That pushed up the number of remote database transactions, but in return for guaranteed progress. Any volunteer client could take up the work where another had left off.

That eventually successfully dealt with all but 0.03% of the original set of seventy-two thousand functional units submitted for analysis. Twenty or so 'ultra hard' exemplars remained intractable. A few of these contained constructs particular to GNU C that could not be handled by the parser, 'interior' (local) function definitions within other function definitions being the most significant such. The rest were most notably characterised by the presence of generated symbolic logical assertions of great complexity, containing more than 40,000 terms each. Clearly, making progress on those requires better techniques with which to reduce the complexity of the symbolic logic expressions encountered or else better techniques for reducing the granularity of the calculations still further.

See Table 2 for a graph of the time taken per task against the percentage of the overall time taken. This graph shows inflexion points at about the 3600s point (corresponding to the one hour mark at which tasks were shifted to checkpointing execution) and also at about the 4500s mark, if not also the 60s mark. The cause behind the latter two inflections is not known.

Table 2: Percentage of total time taken per analysis task (cumulative)



The graph shows that the tasks taking up to ah hour of computation time comprised only abut 30% of the total time taken. Two thirds of the computation time overall was spent on those only a hundred or so ‘very hard’ tasks that in number comprised only 0.15% of the total numbers. That is a very surprising observation.

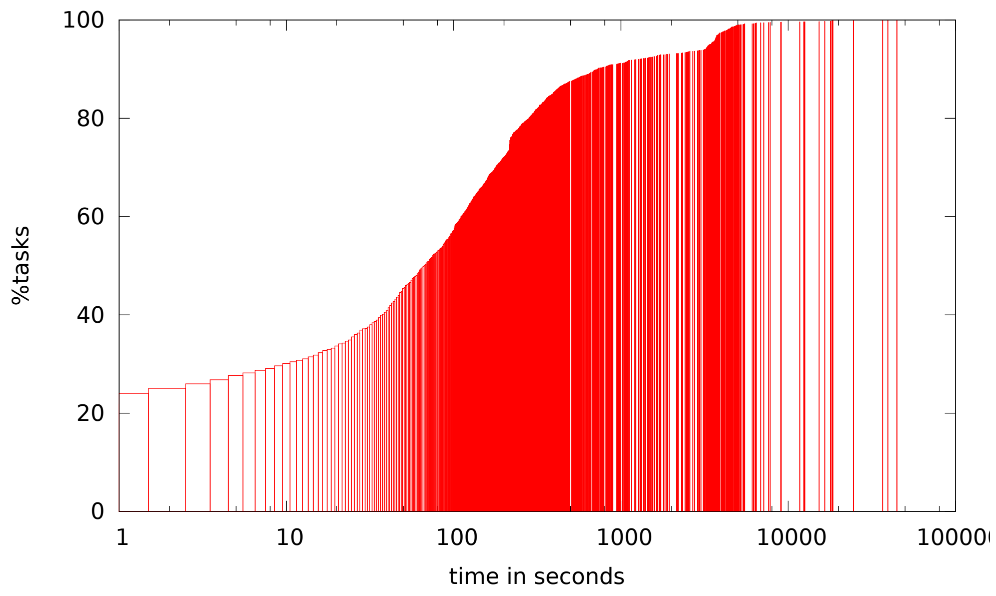
4.5 Statistical Inferences

It should take around 500 1GHz volunteer clients in the cloud to complete the work undertaken in the experiment in under six hours.

A rough average time needed overall for processing per top-level functional unit in the source code was 116 seconds on a notional 1GHz CPU. The ‘very hard tasks’ (taking more than an hour), though they accounted for not much more than 0.15% of the numbers, required around 8% of the processing time. See Table 3 for a straightforward graph of the timing spreads.

Regarding the CPU load expressed on a volunteer, one may conclude that it is only significant in the case of the hard work tasks, which comprise approximately 0.5% of the total number. For these cases, CPU load could in the future be limited by throttling the software automatically. It was not limited during the experiment undertaken, and CPU load was rarely more than a few percent for 99.5% of the tasks undertaken, rising towards maximum levels only on the hard tasks. The implication is that the clients were generally I/O bound, or CPU load would have routinely been much higher. The relatively infrequent queries to the cloud database that penetrated the local caches apparently stalled the client software significantly. The clients could be observed averaging between 150-500 accesses per second to the local cache layer on a 1GHz system (about 90% reads, 10% writes), leaving about 10 transactions per second per client to wend their way out to the rest of the cloud and back.

Table 3: Time taken per analysis task (cumulative count)



The back-end database fanout is presently limited to about 10 clients per server in the cloud, though that figure could be improved with better client-side caches. A fanout of around 40 would appear to be feasible by doubling the number of cores per server and increasing server RAM to 64GB (our experiment used a single core 1.8GHz Athlon with 3GB RAM), since a large proportion of real server load appears to come about through paging data to disk and back in order to accommodate database images that exceeded the available RAM. So between 12-50 servers in the cloud are needed to support the 500 volunteer clients projected as necessary to analyse a million lines of source code in 6 hours.

How does the cloud computation compare to the original monolithic computation from which it is descended? The short answer is ‘about 50 times as slow’, at present, making parity with respect to the original experiment occur about the 50-client mark now. But the original computation threw away all its intermediate calculations as it produced answers, meaning that accountability meant repeating the whole computation from scratch. It was not a scalable solution, while the cloud-computing approach is scalable.

5 Summary

The computation of a certificate guaranteeing the absence of formally defined defects in an open source code base has been formally described.

It has been shown that the computation may be handled incrementally by a distributed ‘volunteer cloud’ of client CPUs each taking a fragment of the work upon themselves at a time. An experiment in which the cloud was organised to analyse about a million lines of C code (requiring about nine million seconds of standardised 1GHz CPU time) has validated the idea.

Bibliography

[Abu09] F. Abujarad, B. Bonakdarpour and S. Kulkarni. Parallelizing Deadlock Resolution in Symbolic Synthesis of Distributed Programs. In *Proc. 8th Intl. Workshop on Parallel and Distributed Methods in Verification*, Nov. 2009 (with Formal Methods 2009).

[And04] D. P. Anderson, BOINC: A System for Public-Resource Computing and Storage. In *Proc. 5th IEEE/ACM Intl. Workshop on Grid Computing*, Nov. 2004.

[AKW05] D. P. Anderson, E. Korpela and R. Walton. High-Performance Task Distribution for Volunteer Computing, In *Proc. 1st IEEE Intl. Conf. on e-Science and Grid Technologies*, Dec. 2005.

[ANSI89] *American National Standard for Information Systems – Programming Language C.* ANSI X3.159-1989. American National Standards Institute. 1989.

[BR02] T. Ball and S. K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *Proc. 29th ACM Symp. on Principles of Programming Languages*, Jan. 2002.

[Bob01] P. K. Bobko. Open-Source Software and The Demise Of Copyright. *Rutgers Computer & Technology Law Journal* 51, 2001.

[BRLS04] M. Barnett, K. Rustan, M. Leino and W. Schulte. The Spec# programming system: An overview. In *Proc. Intl. Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices*, Mar. 2004. LNCS 3362, Springer, 2004.

[BG04] P. T. Breuer and M. García Valls. Static Deadlock Detection in the Linux Kernel. In A. Llamosí and A. Strohmeier (eds.), *Reliable Software Technologies – Ada-Europe 2004, Proc. 9th Ada-Europe Intl. Conf. on Reliable Software Technologies*, June 2004. LNCS 3063, Springer, 2004.

[BP06a] P T. Breuer and S. Pickin. One Million (LOC) and Counting: Static Analysis for Errors and Vulnerabilities in the Linux Kernel Source Code. In L. M. Pinho and M. González Harbour (eds.), *Reliable Software Technologies – Ada-Europe 2006, Proc. 11th Ada-Europe Intl. Conf. on Reliable Software Technologies*, June 2006. LNCS 4006, Springer, 2006.

[BP06b] P. T. Breuer and S. Pickin. Symbolic Approximation: An Approach to Verification in the Large. *Innovations in Systems and Software Engineering* 2(3-4), Dec. 2006.

[BP09] P. T. Breuer and S. Pickin. A Formal Nethod (a Networked Formal Method). *Innovations in Systems and Software Engineering*, 6(4), Dec. 2010.

[CES86] E. Clarke, E. Emerson and A. Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications. *ACM Trans. on Programming Languages and Systems*, 8(2), 1986.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM Symp. on Principles of Programming Languages*, Jan. 1977.

[Dou05] K. Douglas. PostgreSQL. Sams Publishing (2nd ed.), 2005.

[ECCH00] D. Engler, B. Chelf, A. Chou and S. Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *Proc. 4th Symp. on Operating System Design and Implementation*, Oct. 2000.

[EL02] D. Evans and D. Larochelle. Improving Security using Extensible Lightweight Static Analysis. *IEEE Software* 19(1), Jan/Feb 2002.

[FTA02] J. S. Foster, T. Terauchi, and A. Aiken. Flow-Sensitive Type Qualifiers. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, June 2002.

[Gou99] R. W. Gomulkiewicz. How Copyleft Uses License Rights to Succeed in the Open Source Software Revolution and the Implications for Article 2B. *36 Houston Law Review 179*, 1999.

[GH91] J. V. Guttag and J. J. Horning. *Introduction to LCL, A Larch/C Interface Language*. http://ftp.digital.com/pub/Compaq/SRC/research-reports/abstracts/src-rr-074.html.

[Gri02] A. Griffith. *GCC: The Complete Reference*. McGrawHill/Osborne, 2002.

[Hol03] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, Sep. 2003.

[HJG08a] G. J. Holzmann, R. Joshi1 and A. Groce. Swarm Verification. In *Proc. 23rd IEEE/ACM Intl. Conf. on Automated Software Engineering*, Sep. 2008.

[HJG08b] G. J. Holzmann, R. Joshi1 and A. Groce. Model driven code checking. *Automated Software Engineering*, 15(3-4), Dec. 2008.

[ISO99] *ISO/IEC 9899-1999, Programming Languages – C*. International Standards Organisation, 1999.

[JW04] R. Johnson and D. Wagner. Finding User/Kernel Pointer Bugs With Type Inference. In *Proc. 13th USENIX Security Symp., 2004*, Aug. 2004.

[NL96] G. C. Necula and P. Lee. Safe Kernel Extensions without Run-Time Checking. *SIGOPS Operating Systems Review* 30, SI, Oct. 1996.

[WFBA00] D. Wagner, J. S. Foster, E. A. Brewer and A. Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proc. Network and Distributed System Security Symp.*, Feb. 2000.