Code Optimizations Using Formally Verified Properties

Yao Shi
NICTA and
University of New South Wales
yao.shi@nicta.com.au

Bernard Blackham
NICTA and
University of New South Wales
bernard.blackham@nicta.com.au

Gernot Heiser
NICTA and
University of New South Wales
gernot@nicta.com.au

Abstract

Formal program verification offers strong assurance of correctness, backed by the strength of mathematical proof. Constructing these proofs requires humans to identify program invariants, and show that they are always maintained. These invariants are then used to prove that the code adheres to its specification.

In this paper, we explore the overlap between formal verification and code optimization. We propose two approaches to reuse the invariants derived in formal proofs and integrate them into compilation. The first applies invariants extracted from the proof, while the second leverages the property of program safety (i.e., the absence of bugs). We reuse this information to improve the performance of generated object code.

We evaluated these methods on seL4, a real-world formally-verified microkernel, and obtained improvements in average runtime performance (up to 28%) and in worst-case execution time (up to 25%). In macro-benchmarks, we found the performance of para-virtualized Linux running on the microkernel improved by 6–16%.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.3.4 [*Programming Languages*]: Processors; D.4.8 [*Operating Systems*]: Performance

Keywords formal verification; micro-kernel; optimization

1. Introduction

Formal verification is the only way to ensure that the implementation of a software system of non-trivial size meets its specification. In the field of operating systems, formal verification has been advocated and attempted since the 1970s [9, 25]. The seL4 microkernel was the first general-purpose operating system (OS) kernel to be successfully verified by theorem proving. The expected cost of a new formally verified kernel would be only twice the cost of a traditional system with no assurance, and would be half as expensive as the industry rules-of-thumb of \$1k/LOC for Common Criteria EAL 6 certification with less assurance [13]. Hence, verification is already cost-competitive for high-assurance software, and, with on-going improvement in verification techniques, we can expect its use to become more widespread. A number of other OS verification projects are now underway [1, 10], and an argument has been made for "pervasively" verifying applications [8].

While program correctness is obviously an important goal, and the primary driver of formal verification efforts, in this paper we explore another potential benefit – that of improved code optimization. A program which has been verified by theorem proving has many precisely-known properties: about 150 invariants were proved about the models which describe the $\sim$9,000 lines of C code of seL4 [13]. Most of these are quite deep and require significant insight, and are therefore infeasible to find by static analysis, and are therefore out of reach of the analysis performed by a compiler.

We propose the use of such properties in the optimization phase of the compiler. Specifically, we leverage two kinds of properties resulting from formal verification: (1) *explicit invariants* which can be extracted from the verification artefacts and fed into the compiler, and (2) an overall *program safety* property, which implies that a potentially unsafe operation contained in the code can be trusted to be safe. Our techniques can also be applied to unverified software, using informal reasoning about a program's behaviour.

We report our experience with a prototype framework for verification-assisted compiler optimization built around GCC. Our approach uses highly aggressive inlining to perform call-site-specific optimizations. We find performance improvements of up to 28% on average in seL4 micro-benchmarks and 6%–16% for some I/O intensive applications on a Linux system running virtualized on seL4. We observe similar improvements in the worst-case execution time of seL4 system calls.

OOPSLA '13, October 29–31, 2013, Indianapolis, Indiana, USA.
Copyright © 2013 ACM 978-1-4503-2374-1/13/10…\$15.00.
http://dx.doi.org/10.1145/2509136.2509513

This paper makes the following contributions:

- we demonstrate the extraction of useful properties from formal verification and their insertion into the compilation work flow;
- we show how these properties can be used to aid compiler optimization without modifying the source code of the program to be compiled;
- we show that, combined with inlining, this allows us to perform optimizations specific to particular calling contexts;
- we apply this approach to a real-world operating-system kernel, and show significant improvements in average- and worst-case execution times.

2. Background

Klein et. al. successfully demonstrated the first full functional correctness proof of a complete real-world microkernel, called seL4 [13]. They prove that the C implementation of seL4 adheres to an abstract high-level specification of the kernel. We leverage their work and proof effort, using knowledge obtained from the proof to create a compiler plugin that performs code transformations and optimizations. This section provides relevant background on seL4.

2.1 seL4 Proof and Invariants

The *abstract specification* of seL4 is written in the formal theorem proving language Isabelle/HOL and is primarily concerned with the user-visible behavior of the kernel – i.e., *what* the kernel does. There is a large amount of non-determinism at the abstract level, leaving the precise implementation details unspecified. However, several kernel-wide invariants are specified at this level, and any correct implementation of the specification will not violate them. Invariants here include statements about kernel integrity, from local properties such as "every non-NULL pointer points to an object of the correct type", to broader ones such as "all page directory objects have a consistent view of the kernel's mappings".

Conversely, the *C implementation* precisely specifies the details of *how* kernel operations are carried out. This includes the layout of data structures in memory and the algorithms used to achieve a high-performance microkernel. There is very little non-determinism in the C implementation. However, due to the immense detail and precision of the C source, it is difficult to reason about high-level properties and system-wide invariants based on the C code alone.

The relationship between the abstract specification and the C implementation is demonstrated in Figure 1. Over 200,000 lines of hand-written, machine-checked proof text form the *refinement proofs*, which prove that the C implementation correctly implements the abstract specification. These refinement proofs are a classic method of code verification.

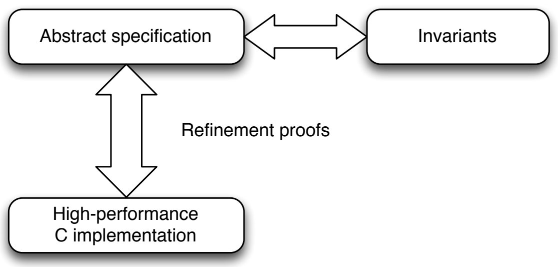

Figure 1. seL4 uses refinement proofs to show that the C implementation adheres to an abstract specification. Most invariants are given on the abstract specification.

```
void deleteCallerCap(tcb_t *receiver) {
   callerSlot = TCB_PTR_CTE_PTR(receiver, 3);
   deleteCap(callerSlot);
}
void deleteCap(slot_t* slot) {
   ...
   ret = finalizeCap(slot->cap, ...);
   ...
}
int finalizeCap(cap_t cap, ...) {
   switch (getCapType(cap)) {
      case cap_null_cap:
      case cap_reply_cap:
         ...
         return 0;
      case cap_endpoint_cap:
         ...
      case cap_async_endpoint_cap:
         ...
   }
   ...
}
```

Figure 2. An example of C code which can be optimized using program invariants. Note the shorter path for the two bolded cases.

For our purposes, we seek to utilize the information available at the abstract and executable specifications, such as kernel-wide invariants, and apply them to code transformations of the C implementation. It is difficult to automatically transfer knowledge at the abstract level down to the C code, so we instead create a database of relevant *translation rules* that have been proven at the abstract level, represented in terms of C expressions. A significant challenge is minimizing effort in creating this database, as there is a wealth of knowledge available at the abstract level but not all of it is relevant or useful for optimization. We will address this issue in Section 4.1 when discussing **demand-driven rule generation**.

2.2 Motivating Examples

Let us consider the C code shown in Figure 2. The function `finalizeCap` can be called from different contexts, and its

```
tcb_cnode_index 3 ↦
   (λ _ thread_state .
   case thread_state of
      BlockedOnReceive e d → (op = NullCap)
      | _ → (op ∈ {ReplyCap, NullCap}))
```

Figure 3. Part of an abstract invariant which applies to the program in Figure 2. The invariant specifies that the "cap at index 3 of a TCB cnode" will be one of the types shown in bold.

```
int setThreadPriority(...) {
  ...
  return setThreadParams(..., NULL);
}
int setThreadParams(..., slot_t *slot) {
  if (...) {
     cap = slot->cap;
     ...
  }
  ...
}
```

Figure 4. An example of code where the program safety invariant can be applied.

behavior is affected by the type of the "cap", which is given by the value of `getCapType(cap)`.

The kernel invariant shown in Figure 3 tells us that the "cap at index 3 of a TCB cnode" (the precise meaning of this expression is unimportant) will be either of type "NullCap" or "ReplyCap", depending on some state. Under the calling context of `deleteCallerCap` → `deleteCap` → `finalizeCap`, we know that we are looking specifically at the "cap at index 3 of a TCB cnode", as this is exactly the object being accessed in the first line of `deleteCallerCap`.

In this context, only the first case of the switch in `finalizeCap` is relevant, as our invariant tells us the only two cap types possible are `cap_null_cap` and `cap_reply_cap` (as represented in C). By generating a specialized version of the function for this context, we can improve the performance of this code path by eliminating the invocation of `getCapType()`, the branching of the switch statement, and the unused remainder of the function. Furthermore, the specialized function is then small enough to be inlined and may benefit from subsequent optimizations. Our work seeks to apply invariants such as these to improve performance.

A second method to improve performance of verified programs is to utilize knowledge of program safety. We leverage the proven guarantee that a program is free of errors such as NULL-pointer dereferences, uninitialized memory reads, memory leaks and buffer overflows, in order to perform much more aggressive code optimizations. Any code path through a program that violates a safety guarantee of the kernel is deemed infeasible and can be eliminated. We note that compilers are already permitted to optimize away

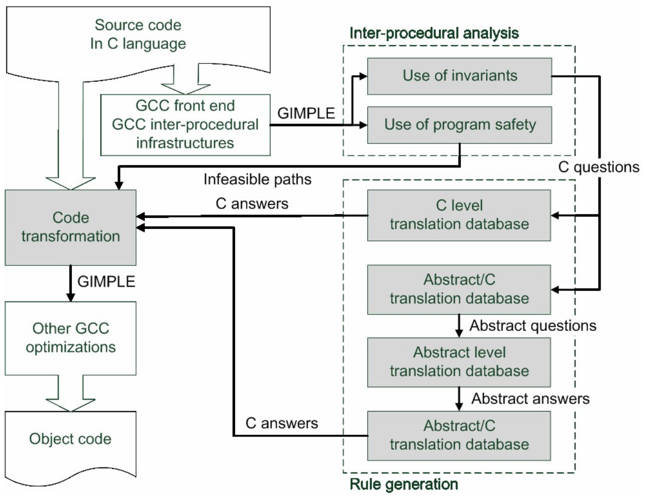

Figure 5. Overview of the framework. The shaded blocks are our implementation.

undefined behaviour, but in practice they are conservative in what optimizations are performed. We discuss this further in Section 7.

Such a case can be seen in Figure 4. The function `setThreadPriority` calls `setThreadParams` with `NULL` as the last argument `slot`. In `setThreadParams`, the pointer `slot` is dereferenced within a conditional block. If the condition was true, the code would dereference a NULL pointer. However, the program is guaranteed to be free of NULL-pointer dereferences, so we can infer that the path from the call site of `setThreadParams` to the pointer dereference is infeasible. In this case, the condition on the if statement must be false and therefore the if statement and its entire body can be removed under this context.

Note that in this scenario, the branch condition does not decide if the pointer `slot` is `NULL`. So the conditional block is not obvious dead code that traditional compilers can eliminate.

3. Architectural Overview

Our framework is implemented as a plugin for the GCC compiler. We interpose our framework into the early stages of GCC's inter-procedural analysis, immediately after the creation of the call graph. At this critical point, we can make best use of the inter-procedural analysis, and influence subsequent inter- and intra-procedural optimizations – in particular, inlining.

The architecture of the framework is shown in Figure 5. There are three key stages to our framework: inter-procedural analysis, rule application and code transformation.

The inter-procedural analysis stage serves two purposes: (1) it identifies "questions" about specific expressions in C code, which may be later answered by proof invariants (described further in Section 4.1); and (2) it detects NULL-

Question

Expression relative to `finalizeCap()`:
```
    getCapType(cap)
```
Expression relative to `deleteCallerCap()`:
```
    getCapType(TCB_PTR_CTE_PTR(
            receiver,3)->cap)
```
$\mathcal{C} = [\texttt{deleteCallerCap}_4, \texttt{deleteCap}_4, \texttt{finalizeCap}_7]$

Answer

Possible values: {`cap_null_cap`, `cap_reply_cap`}
Impossible values: {}

Figure 6. An example of a question, its context ($\mathcal{C}$) and its answer. The subscripts following each function name identify the specific invocation to disambiguate cases where there are multiple such calls within a function.

pointer dereferences which could be eliminated by leveraging knowledge of program correctness (described further in Section 4.3). The questions generated are passed to the next stage, while the detected NULL-pointer dereferences are used to determine infeasible paths which can be eliminated.

In the second stage, proof invariants are identified to answer as many questions as possible by creating translation rules. Some difficult questions may generate proof obligations – these are possible invariants which a verification engineer can either prove correct or reject as bogus. As most invariants apply to higher-level abstractions than the C code, we use translation rules to map high-level statements onto C expressions, and accumulate this information in a database. Our compiler plugin searches the database to find possible answers to the questions. The developer maintains the database and creates the translation rules corresponding to the questions offline. This keeps the entire compilation/optimization sequence uninterrupted. Figure 6 shows an example question and answer pair. We discuss this in further depth in Section 4.2.

Finally, in the third stage, we have a set of answered questions (many questions may have gone unanswered), and a set of infeasible paths derived from the NULL-pointer dereference detection. Our GCC plugin clones functions called under different contexts and applies inter-procedural code transformations, before applying its existing optimizations. We describe this further in Section 4.4.

4. Optimizations

In this section, we elaborate on the algorithms used to perform our code optimizations. First, we must introduce some terminology and definitions, as follows.

- A **context** $\mathcal{C}$ denotes a call stack represented as an ordered list of call sites, which is $\text{NumLevels}(\mathcal{C})$ levels deep. $\mathcal{C}_i$ is the ith call site in the list ($1 \leq i \leq \text{NumLevels}(\mathcal{C})$).
  $\mathcal{F}_i$ is the callee function of $\mathcal{C}_i$. $\mathcal{F}_0$ is the root caller of the context (but $\mathcal{C}_0$ is undefined).
  Questions hold the full context from the root function while answers and rules contain only partial contexts.
- An **expression** $\mathcal{E}$ is derived from a branch condition.
- A **set of values** $\mathcal{V}$ specifies the range an expression may evaluate to.
- A **question** $\mathcal{Q} = (\mathcal{C}, \mathcal{E}_0, \mathcal{E}_1)$ asks "under the context $\mathcal{C}$, what are the possible (or impossible) values of the expression $\mathcal{E}_1$?" Here, $\mathcal{E}_0$ is the expression in $\mathcal{F}_{\text{NumLevels}(\mathcal{C})}$ while $\mathcal{E}_1$ is the equivalent expression in the root caller $\mathcal{F}_0$. Later, we will need to know the possible values of $\mathcal{E}_0$.
- An **answer** $\mathcal{A} = (\mathcal{C}, \mathcal{E}, \mathcal{V}_0, \mathcal{V}_1)$ denotes that, under the context $\mathcal{C}$, the expression $\mathcal{E}$ (in $\mathcal{F}_{\text{NumLevels}(\mathcal{C})}$) may evaluate to a value in $\mathcal{V}_0$ and will never evaluate to a value in $\mathcal{V}_1$. As it is impossible for $\mathcal{V}_0$ to enumerate all possible values, we use the empty set to denote the universal set. Thus, an empty $\mathcal{V}_0$ means that $\mathcal{E}$ can be anything except for the values in $\mathcal{V}_1$, and vice versa for an empty $\mathcal{V}_1$.
- A **basic block** $\mathcal{B}$ represents a branch-free sequence of instructions with only one entry point and one exit point. $\text{BasicBlock}(\mathcal{S})$ refers to the basic block that the statement $\mathcal{S}$ belongs to. The statement $\mathcal{S}$ may be a call site or an expression.
  $\text{FunctionEntry}(\mathcal{F})$ is the basic block containing the entry point of the function $\mathcal{F}$.
- A **path** $\mathcal{I} = (\mathcal{C}, \mathcal{B})$ denotes a specific basic block in the function $\mathcal{F}_{\text{NumLevels}(\mathcal{C})}$, under the calling context $\mathcal{C}$.
- The function $\text{Unmodified}(\mathcal{E}, \mathcal{F}, \mathcal{B})$ is true iff the value of the expression $\mathcal{E}$ is not modified between $\text{FunctionEntry}(\mathcal{F})$ and $\mathcal{B}$ (where $\mathcal{B}$ must be contained in $\mathcal{F}$).

4.1 Applying Invariants

Formally applying invariants from the abstract level to C source code requires a substantial effort by a verification engineer – although there is nothing inherently difficult in doing so, it is something that is not yet automated in the verification of seL4. We instead use translation rules to apply the invariants onto the C code. These rules are presently constructed by hand, but can be proved correct to ensure a total proof of correctness.

The abstract level contains a large number of invariants, yet only a small handful of these are useful for code optimization. Our challenge is to find the useful invariants automatically and represent them at the C level. We propose a solution we call *demand-driven rule generation*.

As shown in Figure 5, the inter-procedural analysis stage generates questions about the C source. These questions provide hints about what would assist code optimizations and are used to manually create translation rules. A question $\mathcal{Q}$ consists of the calling context, the original expression in the leaf callee, and the equivalent expression in the root caller. A question is derived from branch conditions on conditional branches.

The equivalent expression in the root caller is computed by a bottom-up traversal of the call graph: given an expression, if all variables in the expression are parameters to the function and are not modified between function entry and where it is used in the branch condition or call site, we can represent the expression as the equivalent form in the caller function.

Function	Expression
finalizeCap	`getCapType(cap)`
deleteCap	`getCapType(slot->cap)`
deleteCallerCap	`getCapType(TCB_PTR_CTE_PTR( receiver, 3)->cap)`

Table 1. Forms of the same expression in different functions.

Consider our motivating example program in Figure 2. Table 1 shows the different forms of the expression `getCapType(cap)` in function `finalizeCap` at different levels of the call stack. If we can apply proof invariants to compute the possible (or impossible) values of any one of these expressions, then this knowledge can be applied to all expressions, as they are equivalent. In this case, we can use invariants of the expression in `deleteCallerCap` to compute `getCapType(cap)` in `finalizeCap`, and use it to perform code transformation.

The algorithm used to find questions which can possibly be answered by invariants is shown in Algorithm 1. In summary, an expression may be represented in different forms at many levels of the call stack, which are all equivalent. At this stage, we do not know which one may be solved, so we pass all forms with their contexts as questions to the rule generation stage. A question can be answered if any one of its equivalent questions is answered.

4.2 Rule Generation

A translation rule consists of a question and its answer. We use three rule databases, which accumulate translation rules as they are added by the developer. These databases are the C rule database, the abstract rule database, and the abstract/C translation rule database.

There will be simple questions that the developer can confidently answer directly in the C language, which can be immediately added to the C rule database as a translation rule. More difficult questions may require the developer to use the abstract specification to find the answer. In these cases, the question should be converted to the abstract level according to rules in the abstract/C translation rule database. The developer can then prove the answer at the abstract level and add the question and answer to the abstract rule database. Finally, the answer at the abstract level is translated back to C using the abstract/C translation rule database.

Algorithm 1: Finds candidate questions to apply invariants to.

```
Output: Questions: $\mathcal{Q} = (\mathcal{C}, \mathcal{E}_0, \mathcal{E}_1)$
1  function RaiseQuestion($\mathcal{C}$, $\mathcal{E}_0$, $\mathcal{E}_1$, $\mathcal{L}$) begin
2      $\mathcal{C}' \leftarrow (\mathcal{C}_{\mathcal{L}}, \ldots, \mathcal{C}_{\text{NumLevels}(\mathcal{C})})$;
3      $\mathcal{Q} \leftarrow (\mathcal{C}', \mathcal{E}_0, \mathcal{E}_1)$;
4      Output $\mathcal{Q}$;
5  end
6  function Analyze($\mathcal{C}$, $\mathcal{E}_0$, $\mathcal{E}_1$, $\mathcal{L}$) begin
7      RaiseQuestion($\mathcal{C}$, $\mathcal{E}_0$, $\mathcal{E}_1$, $\mathcal{L}$);
8      if $\text{Unmodified}(\mathcal{E}_1, \mathcal{F}_L, \text{BasicBlock}(\mathcal{E}_1))$ and all
       variables in $\mathcal{E}_1$ are parameters and $\mathcal{L} > 0$ then
9          $\mathcal{E}' \leftarrow$ the corresponding form of $\mathcal{E}_1$ at the upper
           level $\mathcal{F}_{\mathcal{L}-1}$;
10         Analyze($\mathcal{C}$, $\mathcal{E}_0$, $\mathcal{E}'$, $\mathcal{L} - 1$);
11     end
12 end
13 function FindCandidateQuestions() begin
14     foreach $\mathcal{C}$ do
15         foreach $\mathcal{E}$ in branch condition do
16             Analyze($\mathcal{C}$, $\mathcal{E}$, $\mathcal{E}$, NumLevels($\mathcal{C}$));
17         end
18     end
19 end
```

Figure 7 shows some examples of translation rules. The syntax of the rules is designed to be simple and familiar for developers. A rule consists of a pattern to match C or abstract expressions, and can optionally include arguments and a context. In the first example of Figure 7, the C level macro `TCB_PTR_CTE_PTR` has two arguments `$1` and `$2` and corresponds to the expression `($1, tcb_cnode_index($2))` in abstract level. If a translation rule applies over the whole program, the context is unnecessary. Otherwise, the developer needs to specify the context in which it applies. This is expressed as a list of functions denoting the call stack. Consider the last example of Figure 7. Here, $\texttt{sendIPC}_1$ and $\texttt{scheduleTCB}_1$ indicate a call stack where `sendIPC` calls the first invocation of `scheduleTCB`, which in turn calls the first invocation of `isRunnable` within it.

Note that the databases are searched automatically for translation rules. Inserting translation rules into the databases is done manually, offline.

The developer does not need to answer all questions. They may not know the answer, or the question may be impossible to answer. In fact, most of the questions do not need to be answered. For example, consider the expressions in Table 1. Each of these expressions, when taken in the context of the given function, forms a question ("what are the possible values of the expression under the given context?"). The first two cannot be answered in general, so can be ignored.

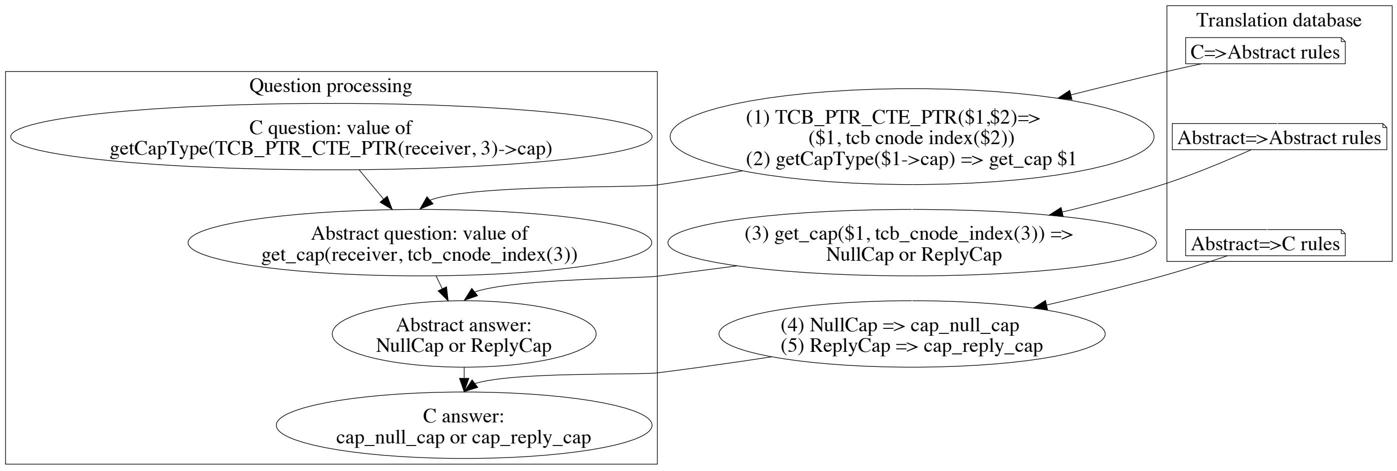
Question processing
C question: value of getCapType(TCB_PTR_CTE_PTR(receiver, 3)->cap)
Abstract question: value of get_cap(receiver, tcb_cnode_index(3))
Abstract answer: NullCap or ReplyCap
C answer: cap_null_cap or cap_reply_cap
(1) TCB_PTR_CTE_PTR($1,$2)=> ($1, tcb cnode index($2))
(2) getCapType($1->cap) => get_cap $1
(3) get_cap($1, tcb_cnode_index(3)) => NullCap or ReplyCap
(4) NullCap => cap_null_cap
(5) ReplyCap => cap_reply_cap
Translation database
C=>Abstract rules
Abstract=>Abstract rules
Abstract=>C rules

Figure 8. An example of question processing.

C → Abstract

`TCB_PTR_CTE_PTR($1,$2)` $\Rightarrow$
`($1,tcb_cnode_index($2))`

Abstract → Abstract

`get_cap($1, tcb_cnode_index(3))` $\Rightarrow$
`NullCap or ReplyCap`

Abstract → C

`NullCap` $\Rightarrow$ `cap_null_cap`

`ReplyCap` $\Rightarrow$ `cap_reply_cap`

C → C (including the context)

`isRunnable(thread)` $\Rightarrow$ `false`
$\mathcal{C} = [\texttt{sendIPC}_1, \texttt{scheduleTCB}_1]$

Figure 7. Examples of translation rules.

However, we have information at the abstract level shown in Figure 3 which can provide an answer for the expression `getCapType(TCB_PTR_CTE_PTR(receiver, 3)→cap)` in the function `deleteCallerCap`. More importantly, we can prove that this expression is either `cap_reply_cap` or `cap_null_cap` at abstract level. That means the expression `getCapType(cap)` in `finalizeCap`, which is specified in the source code, is either `cap_reply_cap` or `cap_null_cap` in this context. This information is passed to the code transformation stage.

Figure 8 demonstrates the process of answering the above question using the translation rules. The compiler asks the C question "what is the value of `getCapType(TCB_PTR_CTE_PTR(receiver,3)→cap)`?" Extracting rules (1) and (2) from the "C⇒Abstract" database and applying these to the C question, it obtains the abstract question for the value of `get_cap(receiver, tcb_cnode_index(3))`. Rule (3) from "Abstract⇒Abstract" database produces the abstract answer `NullCap or ReplyCap`. Finally, applying rules (4) and (5) from the "Abstract⇒C" database yields the C answer, `cap_null_cap or cap_reply_cap`.

4.3 Use of Program Safety

As mentioned in Section 2.2, we can leverage the verified guarantee of program safety. If we detect an unsafe path in the program, we can eliminate it because the program is known to be safe. In this paper, we only use the safety of pointer dereferences, allowing us to eliminate any path which dereferences a NULL pointer. We could make use of other verified safety properties, such as the absence of uninitialized memory reads, buffer overflows and memory leaks.

Our algorithm identifies code paths that will not be executed under the program safety assumption (i.e., those paths that dereference NULL pointers), and eliminates them from the program. This alone does not achieve significant gains, as such code paths are rare. However, there is much more scope for optimization when considering infeasible paths across function boundaries. We can create specialized versions of callee functions that are optimized for their calling context. Algorithm 2 identifies infeasible paths through a bottom-up traversal of each context's call stack. This can be applied to our motivating example shown in Figure 4.

This demonstrates an easy way of leveraging program safety. More precise and complex detection methods may lead to larger optimization improvements. We do not discuss this further as our focus is on leveraging formal verification artefacts.

4.4 Code Transformation

We can apply the information obtained by the techniques in the previous sections to perform code transformations.

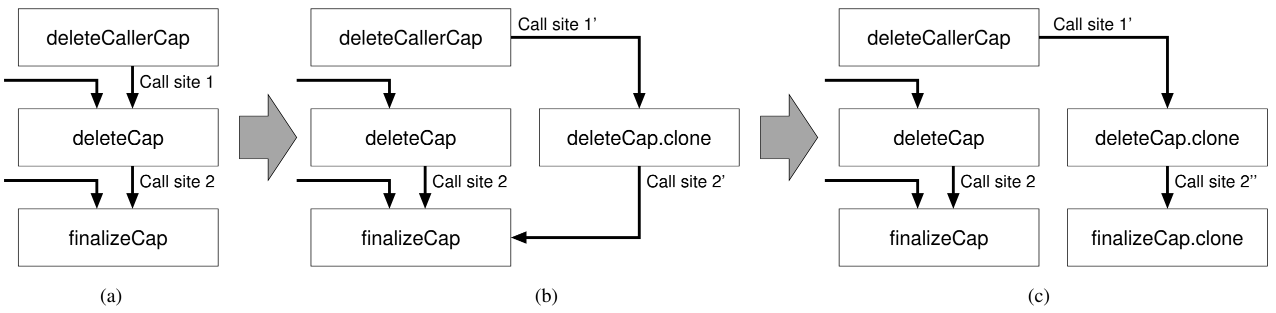

Figure 9. Context separation by function cloning, showing the call graph at each stage of the process.

Algorithm 2: NULL-pointer dereference detection.

```
   Output: Infeasible paths $\mathcal{I} = (\mathcal{C}, \mathcal{B})$
1  function ReportInfeasiblePath($\mathcal{C}$, $\mathcal{B}$, $\mathcal{L}$) begin
2      $\mathcal{C}' \leftarrow (\mathcal{C}_{\mathcal{L}}, \ldots, \mathcal{C}_{\mathrm{NumLevels}(\mathcal{C})})$;
3      $\mathcal{I} \leftarrow (\mathcal{C}', \mathcal{B})$;
4      Output $\mathcal{I}$;
5  end
6  function Backtrace($\mathcal{C}$, $\mathcal{B}_0$, $\mathcal{P}$, $\mathcal{L}$, $\mathcal{B}_1$) begin
7      if Unmodified($\mathcal{P}, \mathcal{F}_{\mathcal{L}}, \mathcal{B}_1$) and $\mathcal{L} > 0$ then
           /* $\mathcal{P}'$ is the equivalent form of $\mathcal{P}$
              in the calling function $\mathcal{F}_{\mathcal{L}-1}$   */
8          if $\mathcal{P}'$ is $NULL$ then
9              ReportInfeasiblePath($\mathcal{C}$, $\mathcal{B}_0$, $\mathcal{L}$);
10         else
11             Backtrace($\mathcal{C}$, $\mathcal{B}_0$, $\mathcal{P}'$, $\mathcal{L} - 1$,
               BasicBlock($\mathcal{C}_{\mathcal{L}}$));
12         end
13     end
14 end
15 function NullPtrDetect() begin
16     foreach $\mathcal{C}$ do
17         foreach $\mathcal{E}$, where $\mathcal{E}$ is the dereferencing of
           parameter pointer $\mathcal{P}$ do
18             Backtrace($\mathcal{C}$, BasicBlock($\mathcal{E}$), $\mathcal{P}$,
               NumLevels($\mathcal{C}$), BasicBlock($\mathcal{E}$));
19         end
20     end
21 end
```

4.4.1 Context Separation

The information we obtain from the invariants is only correct under the relevant context. This means we cannot apply code transformations to existing functions which may be called from anywhere. We solve this using an approach called **context separation**, which creates clones of functions to be called from different contexts.

Specifically, given a context $\mathcal{C} = (\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n)$, we clone the functions $\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_n$ to generate the corresponding functions $\mathcal{F}'_1, \mathcal{F}'_2, \ldots, \mathcal{F}'_n$, and replace the respective call sites with calls to the newly cloned functions. Algorithm 3 describes the details of this method. A concrete example of context separation applied to the program from Figure 4 is shown in Figure 9.

Algorithm 3: Context separation.

```
   Input: Context $\mathcal{C} = (\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n)$
   Output: New context $\mathcal{C}' = (\mathcal{C}'_1, \mathcal{C}'_2, \ldots, \mathcal{C}'_n)$
1  function CloneFunction($\mathcal{C}$) begin
2      $\mathcal{F}'_0 \leftarrow \mathcal{F}_0$;
3      for $i \leftarrow 1$ to $n$ do
4          Clone $\mathcal{F}_i$ to $\mathcal{F}'_i$;
5          Let $\mathcal{C}'_i$ in $\mathcal{F}'_{i-1}$ call $\mathcal{F}'_i$;
6      end
7      $\mathcal{C}' \leftarrow (\mathcal{C}'_1, \mathcal{C}'_2, \ldots, \mathcal{C}'_n)$;
8      return $\mathcal{C}'$;
9  end
```

As context separation replicates many functions, it unsurprisingly increases total code size. Both intuition and past research suggest that this may incur a performance penalty due to the larger instruction cache footprint [15]. In practice, we have not observed any negative performance impact in our experiences. We discuss this aspect further in Section 5.

4.4.2 Applying Answers

We have used the questions and invariants to create a set of answers which describe the possible or impossible values for a given expression. The use of these answers for transforming code depends on the type of answer:

- *Only one possible value.* In this case, we can directly replace the expression with the value. Subsequent optimizations propagate the information for this context.
- *A set of possible values.* This information can be propagated to switch statements by removing labels that are not members of the set. It can also be applied to branch conditions by testing the condition against each of the possible values of the expression. If all tests fail, the condition is set to false. If all tests succeed, the condition is set to true.

- *A set of impossible values.* As above, switch statements can be modified by removing labels that are members of the set. If a branch condition requires the expression to evaluate to one of a set of possible values in order to be true, and this is a subset of the impossible values, then the condition is set to false. Similar logic can check if the branch condition can be set to true.

4.4.3 Applying Infeasible Paths from Program Safety

The output of Algorithm 2 gives an infeasible path which is represented by a context $\mathcal{C} = (\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n)$ and a basic block $\mathcal{B}$ within the function $\mathcal{F}_n$ invoked by $\mathcal{C}_n$.

If $\mathcal{B}$ is not executed on every path through $\mathcal{F}_n$, i.e. $\mathcal{B}$ does not *post-dominate* the entry of $\mathcal{F}_n$, we can simply remove $\mathcal{B}$. Otherwise, the entire function $\mathcal{F}_n$ cannot be executed under the given context. In this case, we recursively work backwards through the calling context, removing the infeasible basic blocks and functions. Algorithm 4 describes the method in detail.

Algorithm 4: Remove infeasible path.

```
   Input: Context $\mathcal{C} = (\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n)$
   Input: Basic block $\mathcal{B}$
1  function RemovePath ($\mathcal{C}$, $\mathcal{L}$, $\mathcal{B}$) begin
2      if $\mathcal{B}$ post-dominates FunctionEntry($\mathcal{F}_\mathcal{L}$) then
3          RemoveFunction($\mathcal{F}_\mathcal{L}$);
4          RemovePath($\mathcal{C}$, $\mathcal{L} - 1$, BasicBlock($\mathcal{C}_\mathcal{L}$));
5      else
6          RemoveBasicBlock($\mathcal{B}$);
7      end
8  end
9  function RemoveInfeasiblePath($\mathcal{C}$, $\mathcal{B}$) begin
10     RemovePath($\mathcal{C}$, NumLevels($\mathcal{C}$), $\mathcal{B}$);
11 end
```

4.5 Effects for Subsequent Optimizations

After our code transformations, the compiler continues its normal analysis and optimization routines, ultimately generating executable objects. Many of our code transformations provide performance gains only after subsequent optimization passes in GCC. We describe the key steps below.

4.5.1 If-Switch Conversion and If-Conversion

In many cases, when a switch expression is optimized, the resulting statement has only a few possible labels. The best case is that only one label remains. In this case, the switch can be eliminated entirely, leaving just the code of the remaining case. If there are two or three labels, the compiler can transform the switch statement to an if statement. This avoids a jump table, reducing code size. It also exposes the opportunity for compilers to apply "if-conversion" during low-level code generation on the ARM architecture, which eliminates further branches by using predicated instructions.

4.5.2 Dead Code Elimination

The code transformations above may modify branch conditions. This can cause related branch conditions (e.g. alternate switch cases) to become unreachable, allowing those code sections to be eliminated. Additionally, some code may be eliminated indirectly. For example, consider the program in Figure 2. The switch statement is simplified to the single case, which ends in a return statement that terminates the function. The remainder of the function becomes unreachable and can be eliminated.

4.5.3 Redundant Parameter Elimination

After performing context separation, some function parameters become unnecessary. For example, a given context may enforce only one possible value for a parameter. Therefore, this parameter can be replaced by a constant and eliminated under this context. This optimization reduces the code size and reduces register pressure around call sites during code generation, making it less likely that expressions will spill onto the stack. It is especially crucial for architectures such as ARM which use general-purpose registers in their standard calling conventions.

4.5.4 Inlining

Inlining is known to be one of the most important optimizations in the modern compiler, as it converts inter-procedural issues into intra-procedural issues that are easier to analyze. In general, the inlining decision is based on the trade-off between the code size (i.e. the size of the callee function) and the benefit after inlining.

For our optimizations, after dead code elimination, the size of some functions may be significantly smaller. This provides opportunities to inline many functions that were originally prohibitively large. For example, in the program in Figure 2, under the context of the caller `deleteCallerCap`, the function `finalizeCap` which originally has tens of statements is reduced to a simple return statement which can be inlined.

5. Evaluation and Discussion

We implemented our techniques as a plugin for GCC 4.5.2, cross-compiling from an x86 host to an ARM target. Our target platform is a BeagleBoard-xM with a TI DM3730 processor. This processor features a 1 GHz ARM Cortex-A8 CPU core. It also contains separate L1 data and instruction caches, each 32 KiB 4-way set-associative, as well as a 128 KiB unified L2 cache. We ran the analysis and compilation on our host platform which is a 2.66 GHz Intel Xeon system with 10 GiB of memory, running Linux.

We measured the impact on runtime performance of three sets of benchmarks:

- micro-benchmarks of seL4: we measure the critical factor in microkernel performance – inter-process commu-

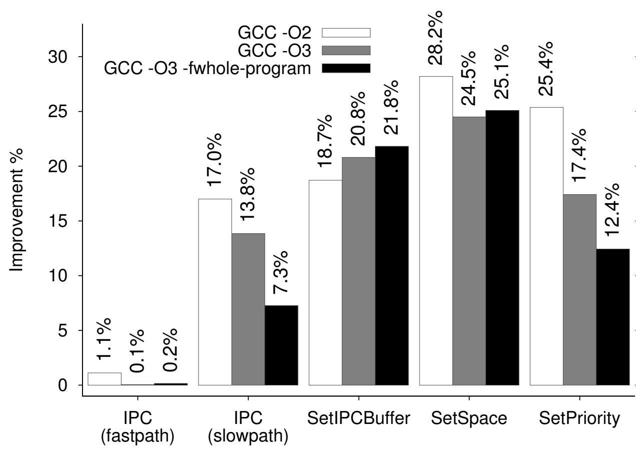

Figure 10. Relative performance of seL4 micro-benchmarks against an unmodified compiler, after our optimizations.

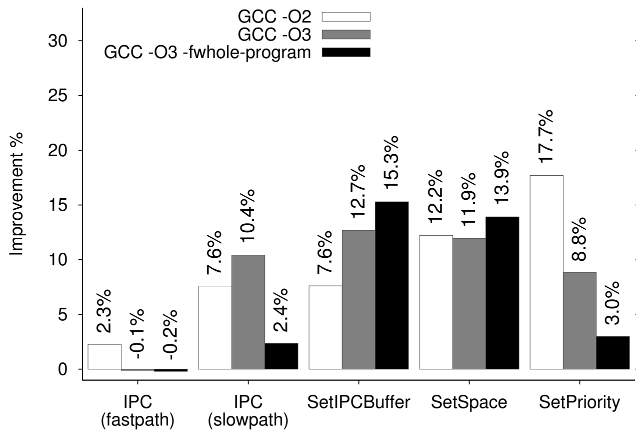

Figure 11. Relative performance using only context separation and inlining, without our invariant-based optimizations.

nication (IPC) – as well as several other microkernel operations.

- micro-benchmarks of virtualized Linux running on seL4: these are obtained by running the LMbench 3.0 test suite.
- macro-benchmarks on virtualized Linux running on seL4: we specifically selected I/O intensive applications (rather than CPU-bound ones), as their performance is significantly affected by the virtualization platform due to frequent context switching.

As seL4 is primarily designed to support different OS personalities, "native" seL4 applications are uncommon. We note that micro-benchmarks on Linux can be considered macro-benchmarks for the seL4 API.

The virtualized Linux platform we tested is based on the Linux 2.6.38 kernel, and is paravirtualized to run on seL4.

All of our tests were run using the `-O2` optimization level in GCC. We also ran the micro-benchmarks at the `-O3` optimization level, and also with the `-fwhole-program` flag. As these introduce extra optimization effort from the compiler, we present these results to show that our analysis improves performance beyond what the unmodified compiler achieves at its most aggressive optimisation level.

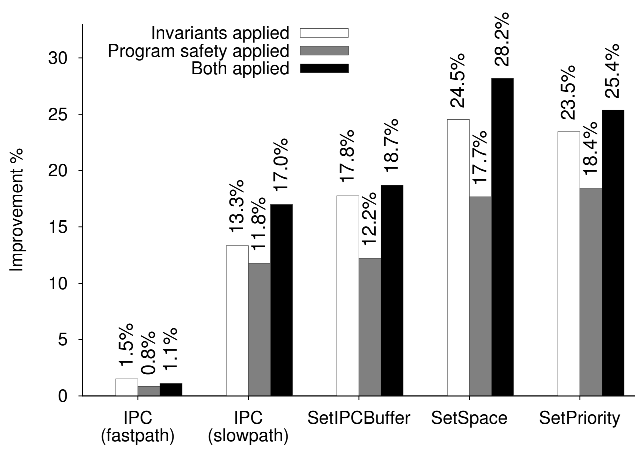

Figure 12. Relative performance of each approach against an unmodified compiler individually, generated with GCC -O2 optimization level.

5.1 seL4 Micro-benchmarks

Figure 10 shows the improvement in performance of seL4's micro-benchmarks using our optimizations. The first benchmark in this graph is the seL4 *IPC fastpath* – this is a highly-tuned C code path for handling the most frequently exercised part of the kernel as quickly as possible. Significant effort was previously devoted to optimizing this code path [4]. As such, there is little room for improvement here. The remaining items in the graph show *slowpath* IPC (the less common case), as well as several typical configuration primitives used to configure threads in seL4. These generally gain 17%–28% improvements when compiled with -O2.

As noted previously, our optimizations clone many functions due to context separation. These cloned functions are typically invoked only once in any execution and may be inlined by the compiler. This has an impact on performance distinct to that of our own optimizations. We quantified this impact separately by measuring the runtime performance after applying only context separation and inlining. None of the proof-based optimizations were used for these measurements.

These results are shown in Figure 11. We see that in almost all cases the improvements are much smaller than in Figure 10. After applying our optimizations, we see performance improvements in all cases.

Finally, we quantify the impact of applying invariants and using program safety separately, shown in Figure 12. Both techniques in isolation improve performance by between 12%–25%, and unsurprisingly are even better when combined.

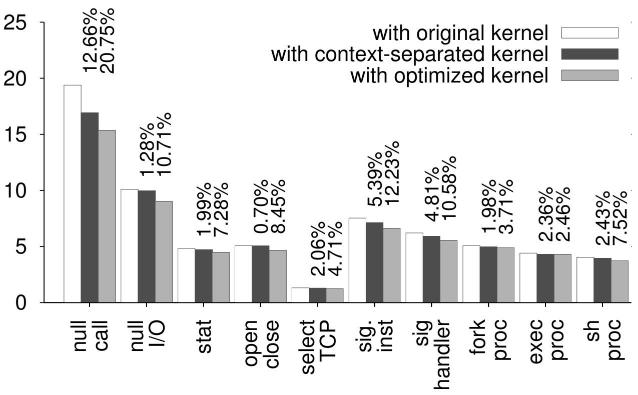

Figure 13. LMbench: relative latency of Linux operations virtualized on seL4, normalized to a baseline of 1 for native Linux (smaller is better).

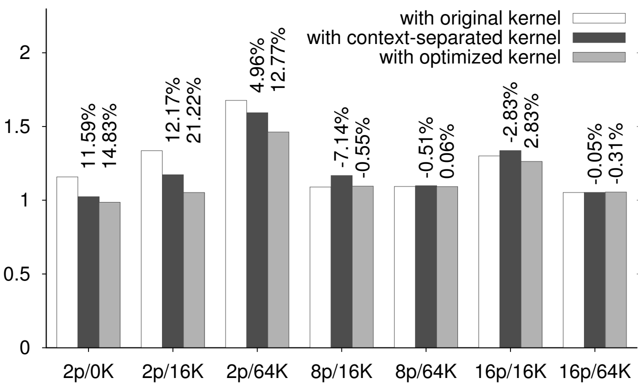

Figure 14. LMbench: relative latency of context switching in Linux virtualized on seL4, normalized to a baseline of 1 for native Linux (smaller is better).

5.2 Virtualized Linux Micro-benchmarks

Running a virtualized Linux machine in parallel with a mission-critical application is a common use-case of microkernels. As such, the LMbench suite provides some insights into the performance of virtualized Linux, as it heavily exercises the Linux kernel and the microkernel/hypervisor. The results of LMbench on virtualized Linux are shown in Figures 13 to 17. All results are normalized to a baseline of 1 representing native (un-virtualized) Linux. We also show the performance improvement obtained using only context separation and inlining, to clearly identify the improvement obtained by our invariant-based optimization. Some performance improvements using only context separation are definitely negative, which reflect the intuition that more inlining is not always better.

In these tests, only the seL4 microkernel was optimized – no changes were made to the Linux kernel or user level programs. As the execution of seL4 system calls are only a part of the entire execution of a test case, the performance im-

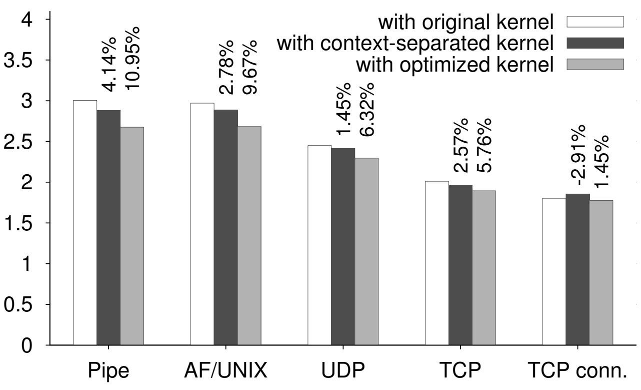

Figure 15. LMbench: relative communication latency in virtualized Linux on seL4, normalized to a baseline of 1 for native Linux (smaller is better).

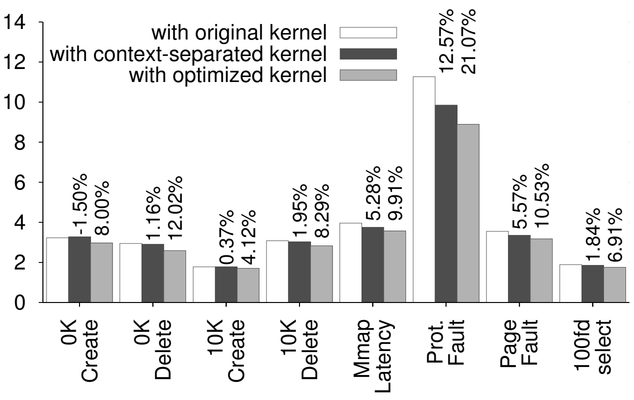

Figure 16. LMbench: relative latency of filesystem operations in virtualized Linux on seL4, normalized to a baseline of 1 for native Linux (smaller is better).

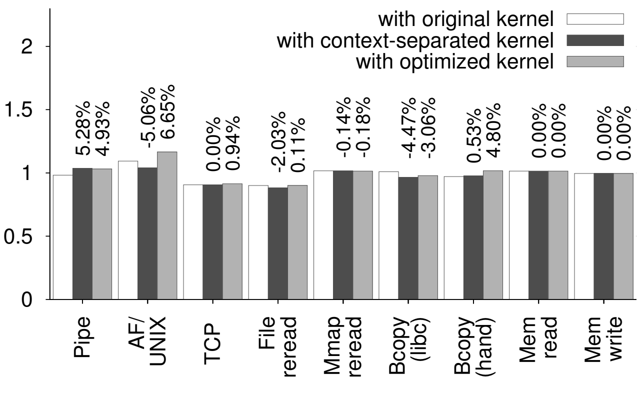

Figure 17. LMbench: relative bandwidth of operations in virtualized Linux on seL4, normalized to a baseline of 1 for native Linux (bigger is better).

provements are naturally less than those of the seL4 micro-benchmarks.

On all latency micro-benchmarks from LMbench, we see improvements of up to 21% with our optimized version of seL4. The improvement in bandwidth throughput (Figure 17) is not as dramatic because these benchmarks are primarily memory-bound.

The observant reader will notice that the “AF/UNIX” benchmark shows better performance virtualized than non-virtualized, both with the normal build as well as with our optimized seL4. This is due to fragility in the benchmark, which also exists in the “pipe” benchmark. These two benchmarks pass messages over UNIX sockets and pipes, respectively. However, neither of these transports are designed to preserve message boundaries – the message boundaries may be influenced by variations in scheduling and interrupts. Therefore, the number of operations required to send a message of a given size varies. Scheduling under virtualized Linux consistently favours better throughput for these benchmarks, as despite the fact operations are more expensive virtualized, fewer operations are required.

5.3 Macro-benchmarks

Finally, we evaluated some typical applications running in the virtualized Linux environment on seL4. These macro-benchmarks give a realistic, tangible measurement of real-world use cases. We examined the performance impact on I/O-intensive applications which make frequent interactions with the Linux kernel. We tested the performance of `tar` and `cp` operations, which are common I/O-intensive commands on Unix systems.

We ran `tar` and `cp` on a single file, whilst varying the size of this file. The average improvements of tar and cp are both 6.4%. See Figure 18.

We also ran `tar` and `cp` on batches of files and directories. With 25 empty files in each directory, we varied the number of directories to exercise the filesystem, with results shown in Figure 19. The average improvements of tar and cp for batches of files and directories are 16.0% and 12.9%, respectively.

Figure 18 and Figure 19 also show the performance improvement from context separation and inlining alone. The results indicate that the main contributor to improved performance is our invariant-based optimizations.

5.4 Worst-Case Execution Time

The worst-case execution time (WCET) of a program is the theoretical upper bound on its execution time. Knowing the WCET of a program is essential for designing hard real-time systems. A smaller WCET reduces the margin by which hardware must be over-provisioned by in order to guarantee correct and timely behaviour in all circumstances. WCET can be computed using offline static analysis or through measurement-based analysis [28].

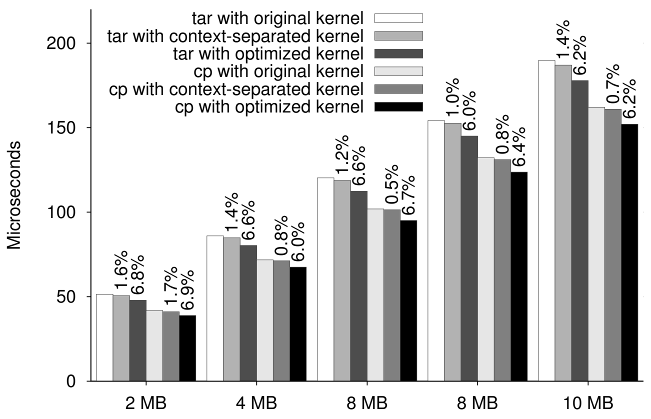

Figure 18. Performance of tar and cp for a single file on Linux virtualized on seL4. The geometric means of the improvements of tar and cp are both 6.4%.

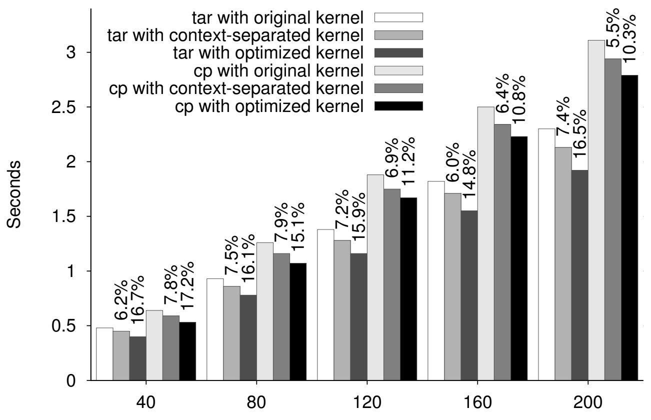

Figure 19. Performance of tar and cp for batch of files and directories on Linux virtualized on seL4. The geometric means of the improvements of tar and cp are 16.0% and 12.9%, respectively.

The seL4 kernel has previously had its WCET computed via static analysis with a view towards supporting safety-critical hard real-time systems [5]. We examined the effects of our optimizations on the WCET of seL4. We found that the WCET was improved in all cases, as shown in Figure 20. The WCET of our optimized kernel is 25.5% less than that of the original kernel for a normal system call. The improvement can be mostly attributed to eliminated code which reduces branch mispredictions and cache misses.

5.5 Impact on Code Size

The compiled seL4 kernel increased in size due to context separation and inlining. This is quantified in Figure 21 which shows the overall kernel size, including code, data and heap sections produced by the compiler, as well as the effect on the code size alone. For each optimization level, we show the size of the original and optimized kernels, as well as for

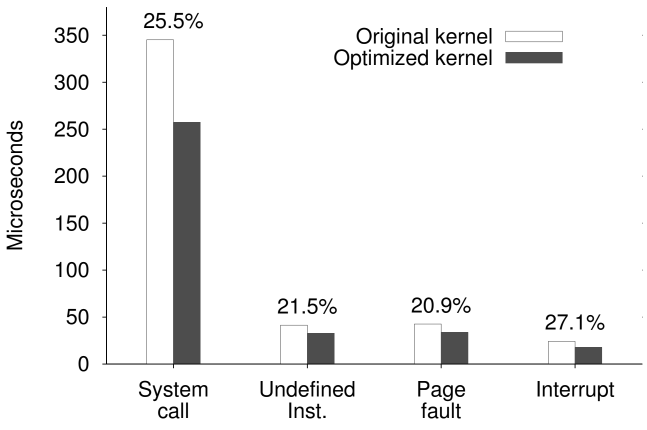

Figure 20. Worst-case execution time of seL4 before and after invariant-based optimization (smaller is better).

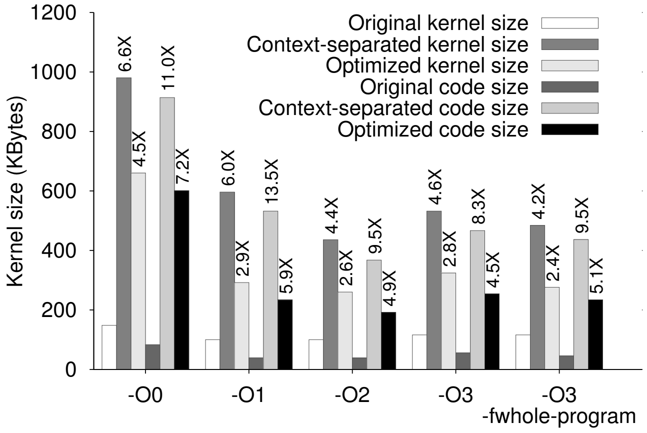

Figure 21. seL4 kernel size (text, data and heap sections) and code size (text section only) when compiled with different GCC optimization levels.

a kernel compiled with context-separation and inlining. This demonstrates that our invariant-based optimizations eliminate a significant amount of dead code. In all cases where any optimization was used (-O1 or higher), the overall kernel size increased to at most 2–3$\times$ of the original.

Despite the increases in kernel and code size, we see better performance. We believe this to be for the following reasons:

- The code size of a single function is usually reduced, never expanded. That means for any single function, the local cache performance will be improved using our optimizations.
- Context separation clones the entire call chain which actually improves the likelihood of cache hits. Consider a function a which calls function b, and b is cloned to b'. The scenario detrimental to cache performance occurs when b is in the cache but a calls b'. However, this is not a common scenario. Additionally, modern compilers implement mitigating techniques which reorder functions and basic blocks [16]. These techniques reorder the position of b' or the basic blocks in b' after inlining to increase the probability of cache hits. This may make up for the performance loss due to increased instruction cache size by code expansion.

Lines of C code	9525
Number of functions	276
Number of contexts	217439
Number of processed expressions	106334
Inter-procedural analysis time	12.22 seconds
Rule generation time	0.04 seconds
Code transformation time	5.23 seconds
Total compilation time	201.09 seconds
Original total compilation time	16.89 seconds
Number of C questions	10223
Number of C answers	1540
Number of rules	125
Number of separated contexts	4671

Table 2. Properties of seL4 and our optimizations. The times are measured with GCC -O2.

Given this, we assert that a code base of seL4's size is well suited to context separation. The increase in code size of less than 200 KiB (at -O2 or above) is insignificant on today's protected-mode platforms (even embedded ones). On large applications or resource-constrained devices, heuristic-based inlining becomes necessary to ensure that the code size increases remain manageable. However, this is a complex topic beyond the scope of this paper.

5.6 Scalability

Table 2 gives some basic information on the seL4 code base used and our optimizations. We counted the number of functions using the GCC intermediate representation (GIMPLE) rather than in the source code. The number of expressions counts multiple occurrences several times if they occur under different contexts. The number of contexts is greater than the number of expressions because some functions (and therefore contexts) do not have a branch or only branch conditions that cannot be processed. These numbers show that it is possible to enumerate all contexts and process them in a reasonable amount of time. For programs much larger than seL4, users may select the appropriate contexts or use a BDD-based system [27] for context processing.

The table also shows the time required for each stage of the analysis. The inter-procedural analysis time includes both applying invariants and using program safety. Most of this time is spent enumerating all contexts and expressions. The compilation time is significantly increased, disproportionately so compared to the expansion in code size. However, given the benefit in runtime performance, we believe this is acceptable for many use cases.

The last section in Table 2 shows some statistics of the rule generation phase. For the seL4 micro-kernel, applying invariants raises 10,223 questions in the C language. 1540 of them can be answered with our 125 manual translation rules. This shows that the rules are effective at answering many questions in practice.

Although we separated 4671 contexts, most of the cloned functions were eliminated in subsequent optimizations. This reinforces the code size results seen in Figure 21: that with no optimizations, the code size is significantly larger.

6. Limitations and Applicability

6.1 Implementation Correctness

At present, although we make use of invariants derived from the formal verification, the translation rules themselves are not formally verified. Thus, human error in writing the rules can lead to an incorrect binary. This is not an inherent limitation in our approach, as it should be possible to formally verify each of the translation rules, eliminating this weakness from the trusted computing base.

Bugs in the compiler or our plugin can lead to incorrect object code being produced. GCC implements hundreds of optimizations, none of which are formally verified. Without any correctness guarantees, verified translation rules can be easily undermined.

There are at least two possible solutions to these issues. The first is to implement our approach in a formally verified compiler such as CompCert [7, 14]. However, the state-of-the-art in verified compilers cannot currently match the performance achieved by mainstream optimizing compilers such as GCC and LLVM.

A second solution is to verify the compiled assembly code directly against the C code. This was recently achieved by Sewell et al. for the seL4 microkernel [18]. By integrating this verification step with the formal proof of seL4, we can apply the existing proven invariants, which would wholly remove the compiler, our plugin, and the translation rules from the trusted computing base, thereby avoiding any reliance on their correctness. The proof would guarantee that the binary is correct (with respect to the kernel specification), in the presence of all optimizations.

6.2 Applicability

The main components of our approach, i.e., applying invariants and program safety, are only loosely coupled with compiler internals. This makes our approach applicable to both verified and non-verified compilers. There are two scenarios for the potential application of our idea.

Firstly, for software that demands the strong assurances of formal verification, the developer can input the invariants into a verified compiler such as CompCert. In addition to implementing the code transformation within the compiler, the correctness of the transformations needs to be proven using the invariants. Such a proof may require extra developer effort, but is required to maintain the strong assurances of formal verification. However, as mentioned previously, CompCert does not currently achieve the same levels of performance as mainstream optimizing compilers.

On the other hand, our work can also be applied to software that is not formally verified. For unverified software, if a developer informally believes a program invariant to be true, our techniques can use this to optimize the compilation process. Most applications today use optimizing compilers for performance, and rely on industry standard testing practices to catch bugs. The impact of an incorrect invariant in these circumstances is comparable to any regular program bug where a developer had assumed the invariant to be true.

The requirements of a specific application dictate the acceptable trade-offs between correctness and performance. If high assurance is required, system designers may choose to sacrifice performance for verified correctness.

6.3 Human Effort

Another limitation of our work is the human effort required to generate the translation rules. Although the process of applying rules at compile time is automatic and transparent, the entire optimization process cannot be performed without this manual work.

In our approach, questions have full calling contexts while rules and answers have only partial contexts. Therefore, one partial context may match several full contexts. For example, of the 10,223 questions (as shown in Table 2), many are identical except for the context. 117 questions are similar to the example in Table 1 and can be answered by a single rule.

Furthermore, if a question cannot be answered, we can filter out many similar questions with different contexts to reduce human effort. For seL4, we translated 125 invariants into rules, which answer 1,540 questions. It took us 2-3 days in total to review all 10,223 questions and provide the necessary translation rules (given existing familiarity with the codebase). Providing these rules proved much easier than manual code optimization.

In our experience, it is easy to maintain most of the translation rules when the source code changes. Many rules are relatively primitive and context-independent so that source code changes usually do not affect these rules. Let us consider the translation rules in Figure 7. The first three rules are primitive and context-independent. So the code changes generally do not affect the correctness of these rules except for the changes of the infrastructure. However, the last rule is context-dependent which may not hold after some general code changes.

Despite all this, the human effort required is still a limitation of our approach. We plan to investigate methods to derive these rules automatically in the future.

7. Related Work

Since the 1970s, formal program verification, and in particular OS verification, has been an active research area. As performance is a key consideration in most practical operating systems, many verification approaches have focused on implementations in low-level languages such as C/C++ [2, 11–13, 19, 25]. Optimization of these systems traditionally focuses on modifications to the source code – e.g. using better algorithms.

Vandevoorde describes a prototype compiler for the Speckle programming language which makes use of a formal specification written in the Larch Shared Language [24]. In that work, the formal specification is targeted for optimizations, not for a proof of program correctness. Thus the specification only guarantees the correctness of the optimizations performed, but not that of the original program. The specification also requires multiple implementations for general purpose and more specific optimizations. In contrast, our motivation is to leverage the existing specifications for the program correctness, which are not necessarily designed for optimizations. We do not need to modify the specification and its implementation, and do not need multiple implementations. Furthermore, Vandevoorde's work is not based on a full-featured programming language and omits some important features such as non-local variables. It measured only one simple benchmark of "AC-unification" and gained 11% improvement comparing with an unknown baseline. In comparison, our approach is more practical and has successfully optimized a real-world microkernel written in C.

Recently, Blackham and Heiser investigated optimizing the IPC fastpath in the seL4 microkernel [4]. Unlike traditional fastpaths which are hand-coded in assembly, the seL4 fastpath is written in C. They hand-tuned C code to provide the compiler with more optimization opportunities. They used knowledge from kernel invariants to ensure that reordered code was still safe (e.g. that pointers could be safely dereferenced earlier).

The CompCert project investigates the formal verification of realistic compilers [6, 7, 14]. They have successfully created the formally verified CompCert C compiler, and have verified several optimization algorithms [3, 17, 21–23]. However, none of these apply formal verification of their inputs to improve the performance of generated binaries.

To the best of our knowledge, no formally verified real-world software has been automatically optimized using information obtained from the formal verification process.

We note that C compilers are permitted to treat undefined behaviour however they choose. Wang et al. found numerous bugs in critical software due to compilers optimizing code that accidentally used undefined constructs [26]. They demonstrate a case where a NULL-pointer dereference in Linux caused security-critical checks to be omitted by the compiler. Exploiting undefined behaviour can improve the optimizations available to a compiler, but compiler writers must be cautious, as many existing programs unknowingly invoke undefined behaviour which can result in serious bugs or security vulnerabilities [20]. In this paper, because seL4 is proven to be free of undefined behaviour, we can use much more aggressive optimizations without fear of introducing such bugs.

8. Conclusion

Formal verification is traditionally only used to demonstrate program correctness. We have demonstrated a technique which can leverage existing proof effort to improve code optimizations. We proposed two approaches, applying invariants and program safety, both of which are pervasive for formally verified programs.

We have shown how to reuse information such as explicit invariants from the formal verification of a program and integrate it into the compilation of the code in order to improve its runtime performance. We also leverage the guarantee of program safety provided by formal verification, to perform further optimizations by detecting infeasible code paths. Invariants may also be derived through informal reasoning and applied to unverified software.

We evaluated our approaches and applied them to the formally-verified seL4 microkernel. Integrating explicit invariants gave a performance increase of up to 24% in some micro-benchmarks. When combined with the guarantees of program safety, the maximum performance gains increase to 28%. On I/O-heavy macro-benchmarks, we observe up to 16% improvements. Worst-case execution time was also reduced by 20–25%.

Our experiments have shown our techniques to be practical and effective for real-world applications. Constructing translation rules in order to make use of the invariants is currently a necessary but tedious process. Future work will investigate how to automatically extract the relevant information from a formal specification.

Acknowledgements

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

[1] E. Alkassar, M. Hillebrand, D. Leinenbach, N. Schirmer, and A. Starostin. The Verisoft approach to systems verification. In N. Shankar and J. Woodcock, editors, *VSTTE 2008*, volume 5295 of *LNCS*, pages 209–224. Springer, 2008.

[2] J. Andronick, B. Chetali, and C. Paulin-Mohring. Formal Verification of Security Properties of Smart Card Embedded Source Code. In J. Fitzgerald, I. J. Hayes, and A. Tarlecki, editors, *FM*, volume 3582 of *LNCS*, pages 302–317, Newcastle, UK, Jul 2005. Springer.

[3] Y. Bertot, B. Gregoire, and X. Leroy. A structured approach to proving compiler optimizations based on dataflow analysis.

In *Types for Proofs and Programs, Workshop TYPES 2004*, volume 3839 of *LNCS*, pages 66–81, 2006.

[4] B. Blackham and G. Heiser. Correct, fast, maintainable – choose any three! In *3rd APSys*, pages 13:1–13:7, Seoul, Korea, Jul 2012.

[5] B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser. Timing analysis of a protected operating system kernel. In *32nd RTSS*, pages 339–348, Vienna, Austria, Nov 2011.

[6] S. Blazy and X. Leroy. Formal verification of a memory model for C-like imperative languages. In *International Conference on Formal Engineering Methods, vol. 3785 of LNCS*, pages 280–299, 2005.

[7] S. Blazy, Z. Dargaye, and X. Leroy. Formal verification of a C compiler front-end. In *14th FM*, volume 4085 of *LNCS*, pages 460–475. Springer, 2006.

[8] M. Daum, N. W. Schirmer, and M. Schmidt. From operating-system correctness to pervasively verified applications. In *IFM*, volume 6396 of *LNCS*, pages 105–120, Nancy, France, 2010. Springer.

[9] R. J. Feiertag and P. G. Neumann. The foundations of a provably secure operating system (PSOS). In *AFIPS Conf. Proc., 1979 National Comp. Conf.*, pages 329–334, New York, NY, USA, Jun 1979.

[10] L. Gu, A. Vaynberg, B. Ford, Z. Shao, and D. Costanzo. CertiKOS: A certified kernel for secure cloud computing. In *2nd APSys*, 2011.

[11] C. L. Heitmeyer, M. Archer, E. I. Leonard, and J. McLean. Formal specification and verification of data separation in a separation kernel for an embedded system. In *CCS*, pages 346–355, Alexandria, VA, USA, 2006.

[12] M. Hohmuth and H. Tews. The VFiasco approach for a verified operating system. In *2nd PLOS*, Glasgow, UK, Jul 2005.

[13] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *22nd SOSP*, pages 207–220, Big Sky, MT, USA, Oct 2009. ACM.

[14] X. Leroy. Formal certification of a compiler back-end, or: Programming a compiler with a proof assistant. In J. G. Morrisett and S. L. P. Jones, editors, *33rd POPL*, pages 42–54, Charleston, SC, USA, 2006. ACM.

[15] J. Liedtke. On μ-kernel construction. In *15th SOSP*, pages 237–250, Copper Mountain, CO, USA, Dec 1995.

[16] A. Ramirez, J. l. Larriba-pey, C. Navarro, J. Torrellas, and M. Valero. Software trace cache. In *13th Intl. Conference on Supercomputing*, 1999.

[17] S. Rideau and X. Leroy. Validating register allocation and spilling. In *Compiler Construction*, volume 6011 of *LNCS*, pages 224–243, 2010.

[18] T. Sewell, M. Myreen, and G. Klein. Translation validation for a verified OS kernel. In *PLDI*, pages 471–481, Seattle, Washington, USA, Jun 2013. ACM.

[19] H. Tews, T. Weber, and M. Völp. A formal model of memory peculiarities for the verification of low-level operating-system code. In R. Huuck, G. Klein, and B. Schlich, editors, *3rd SSV*, volume 217 of *ENTCS*, pages 79–96, Sydney, Australia, Feb 2008. Elsevier.

[20] The GCC Team. GCC 4.8 release series: Changes, new features, and fixes. `http://gcc.gnu.org/gcc-4.8/changes.html`, 2013.

[21] J.-B. Tristan and X. Leroy. Formal verification of translation validators: A case study on instruction scheduling optimizations. In *35th POPL*, pages 17–27, 2008.

[22] J.-B. Tristan and X. Leroy. Verified validation of lazy code motion. In *2009 PLDI*, pages 316–326, 2009.

[23] J.-B. Tristan and X. Leroy. A simple, verified validator for software pipelining. In *37th POPL*, pages 83–92, 2010.

[24] M. T. Vandevoorde. Specifications can make programs run faster. In *Theory and Practice of Software Development, LNCS*, volume 668, pages 215–229, 1993.

[25] B. J. Walker, R. A. Kemmerer, and G. J. Popek. Specification and verification of the UCLA Unix security kernel. *CACM*, 23(2):118–131, 1980.

[26] X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Undefined behavior: what happened to my code? In *3rd APSys*, pages 9:1–9:7, New York, NY, USA, 2012. ACM.

[27] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *2004 PLDI*, 2004.

[28] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Emb. Comput. Syst.*, 7(3):1–53, 2008. ISSN 1539-9087.