Parallelizing Top-Down Interprocedural Analyses

Aws Albarghouthi *
University of Toronto
aws@cs.toronto.edu

Rahul Kumar
Microsoft Corporation
rahulku@microsoft.com

Aditya V. Nori
Microsoft Research India
adityan@microsoft.com

Sriram K. Rajamani
Microsoft Research India
sriram@microsoft.com

Abstract

Modularity is a central theme in any scalable program analysis. The core idea in a modular analysis is to build summaries at procedure boundaries, and use the summary of a procedure to analyze the effect of calling it at its calling context. There are two ways to perform a modular program analysis: (1) top-down and (2) bottom-up. A bottom-up analysis proceeds upwards from the leaves of the call graph, and analyzes each procedure in the most general calling context and builds its summary. In contrast, a top-down analysis starts from the root of the call graph, and proceeds downward, analyzing each procedure in its calling context. Top-down analyses have several applications in verification and software model checking. However, traditionally, bottom-up analyses have been easier to scale and parallelize than top-down analyses.

In this paper, we propose a generic framework, BOLT, which uses MapReduce style parallelism to scale top-down analyses. In particular, we consider top-down analyses that are demand driven, such as the ones used for software model checking. In such analyses, each intraprocedural analysis happens in the context of a reachability query. A query Q over a procedure P results in query tree that consists of sub-queries over the procedures called by P . The key insight in BOLT is that the query tree can be explored in parallel using MapReduce style parallelism – the map stage can be used to run a set of enabled queries in parallel, and the reduce stage can be used to manage inter-dependencies between queries. Iterating the map and reduce stages alternately, we can exploit the parallelism inherent in top-down analyses. Another unique feature of BOLT is that it is parameterized by the algorithm used for intraprocedural analysis. Several kinds of analyses, including may-analyses, must-analyses, and may-must-analyses can be parallelized using BOLT.

We have implemented the BOLT framework and instantiated the intraprocedural parameter with a may-must-analysis. We have run BOLT on a test suite consisting of 45 Microsoft Windows device drivers and 150 safety properties. Our results demonstrate an average speedup of 3.71x and a maximum speedup of 7.4x (with 8 cores) over a sequential analysis. Moreover, in several checks where a sequential analysis fails, BOLT is able to successfully complete its analysis.

* This author performed the work reported here during a summer internship at Microsoft Research India.

PLDI'12, June 11–16, Beijing, China.
Copyright © 2012 ACM 978-1-4503-1205-9/12/04…$10.00

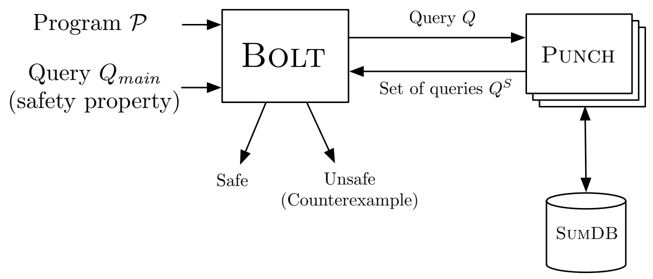


Figure 1. Black box view of the BOLT framework.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification– assertion checkers, correctness proofs, formal methods, model checking; D.2.5 [*Software Engineering*]: Testing and Debugging– symbolic execution, testing tools; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs– assertions, pre- and post-conditions

General Terms Parallelism, Testing, Verification

Keywords Abstraction refinement; Interprocedural analysis; Software model checking

1. Introduction

Scalable program analyses work by exploiting the modular structure of programs. Almost every interprocedural analysis builds summaries at procedure boundaries, and uses the summary of a procedure at its calling contexts, in order to scale to large programs. Broadly, interprocedural analyses can be classified as either top-down or bottom-up, depending on whether the analysis proceeds from callers to callees or vice-versa.

Bottom-up analyses have been traditionally easier to scale. They work by processing the call graph of the program upwards from the leaves. In a bottom-up analysis, before a procedure P_i is analyzed, all the procedures P_j that are called by P_i are analyzed, and for each P_j its summary S_{P_j} is computed, typically without considering the calling contexts of P_j. Then, during the analysis of P_i the summary S_{P_j} is used to calculate the effects of calling P_j, instead of the body of P_j. One of the significant advantages of bottom-up analyses is their decoupling between callers of a procedure P and the analysis of the body of P, which enables parallelization. For instance, the designers of the SATURN tool [1], a scalable bottom-up analysis, say that "*another advantage of analyzing procedures separately is that the process is easily parallelized, with parallelism limited only by analysis dependencies between different procedures. We use compute clusters of 40-100 cores to run Saturn analyses in parallel and normally achieve 80-90% efficiency*".

In contrast, a top-down analysis starts from the root of the call graph and proceeds downward, analyzing each procedure in its calling context. Top-down analyses have several applications in verification and software model checking [2, 3, 19], as well as dynamic test generation [16, 30]. However, top-down analyses have been more challenging to scale and parallelize. During a top-down analysis, the analysis of a procedure P_i is done separately for each calling context, leading to repeated analysis of each procedure, and fine grained dependencies between analysis instances. Hence, top-down analyses are harder to scale. However, top-down analyses do have precision advantages. Since each analysis of a procedure P_i is done with respect to a calling context, the summary built for that context can be more precise. Thus, software model checkers such as SLAM [4] and BLAST [10], which employ a top-down analysis, are able to be very precise, and have very low false error rates, but only scale to programs of size 100K lines of code [7].

Thus, it is natural to ask if we can parallelize and scale top-down analyses. In particular, we consider top-down analyses that are demand driven, such as the ones used for software model checking. In such analyses, each intraprocedural analysis happens in the context of a query. A query Q over a procedure P results in sub-queries over procedures called by P. The query Q is in a "blocked" state (that is, it cannot make progress) until at least one of these sub-queries can be answered using a summary. When such a summary is available for one or more of the sub-queries, the parent query Q transitions to a "ready" state where it can continue to execute, and may produce new sub-queries and get "blocked" again. When the analysis of a query Q is conclusive, it moves to a "done" state. It is important to note that (this will be illustrated later) a query Q can move to a "done" state, even before all of its sub-queries are done. Therefore, in this situation, the remaining (transitive) sub-queries of q can be stopped and garbage collected. Thus, the dependencies between query instances are more intricate and detailed during a top-down analysis.

In this paper, we propose a generic framework, BOLT, which uses MapReduce [13] style parallelism to scale top-down analyses. The key insight in BOLT is that the query tree can be explored in parallel as follows:

- The *map stage* is used to run a set of "ready" queries in parallel, which can potentially result in a new set of sub-queries, and
- the *reduce stage* is used to manage interdependencies between queries, assess which parent queries can be moved to the "ready" state from the "blocked" state, and which queries can be garbage collected because they are no longer necessary for the parents that originated these queries.

By iterating the map and reduce stages alternately, BOLT can exploit the parallelism inherent in any top-down analysis.

BOLT is parameterized by an intraprocedural analysis algorithm, which we shall henceforth call PUNCH, used to analyze a single procedure. BOLT initially receives a program $\mathcal{P}$ and a reachability query Q_{main} over the main procedure *main* of $\mathcal{P}$ and it employs PUNCH to process Q_{main}. PUNCH explores paths in *main* either forward or backward, or using combination of both. It can use an overapproximate analysis, an underapproximate analysis, or a combination of both. Whenever it encounters a method call P_i, PUNCH formulates a sub-query Q_i for P_i, which it needs to know about P_i in order to answer the query it was asked, namely Q_{main}. We say that Q_i is a *child* of the *parent* query Q_{main}. PUNCH first looks for a summary which can answer Q_i in a database of summaries SUMDB. If a suitable summary is found, it answers Q_i using that summary and moves on. If not, PUNCH sets the query Q_i to the ready state and adds it to the set of queries it will return, and explores other paths in *main*, repeating the same strategy to handle any procedure calls it encounters on these paths. The PUNCH call on the query Q_{main} finishes when PUNCH cannot perform any further analysis in *main* without getting answers to queries it has made to callees of *main*. At this point, it returns all the sub-queries it has generated (which are all in the ready state), and Q_{main} itself, which it sets to blocked state. The next map stage applies PUNCH to all the queries that are in the ready state in parallel, and the subsequent reduce stage then processes the answers produced by the map stage, removing completed and redundant queries, and informing parent queries when their children have produced answers. This process continues until Q_{main} returns an answer. Figure 1 shows an overview of the BOLT framework which uses multiple instances of PUNCH in parallel and a database of procedure summaries SUMDB that stores results produced by PUNCH to avoid recomputing similar queries over the same procedure.

In our exposition of BOLT, we provide a formal specification of the parameter PUNCH that any correct instantiation of BOLT should respect. PUNCH can be instantiated to capture different kinds of analyses. An instantiation of PUNCH to a "may-analysis" or an overapproximation-based analysis, using "may" summaries, can parallelize analyses such as SLAM [4] and BLAST [10]. An instantiation of PUNCH to a "must-analysis" or an underapproximation-based analysis, using "must" summaries, can parallelize analyses such as DART [17] and CUTE [35]. We present an instantiation of PUNCH that uses a "may-must-analysis", combining overapproximations and underapproximations, to parallelize algorithms in the family of DASH [9, 19].

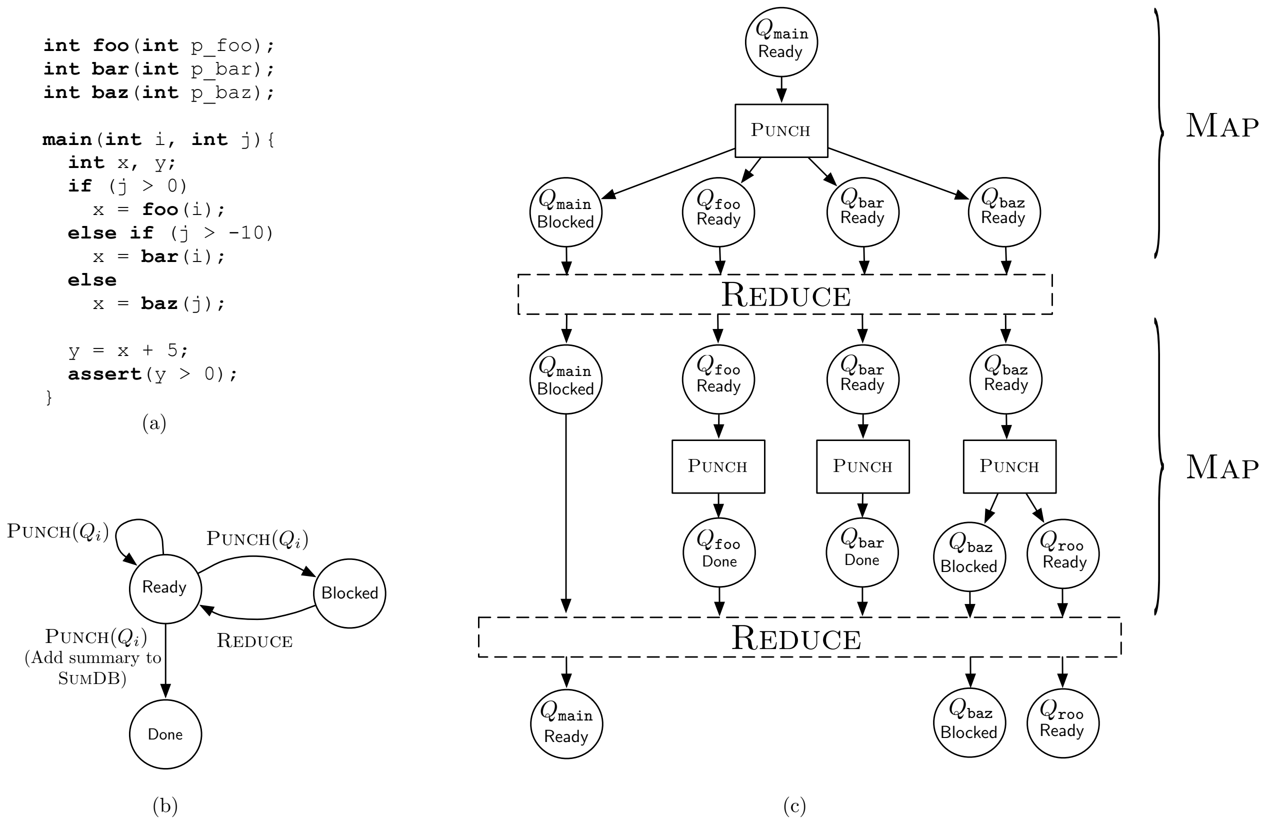
Our contributions are summarized as follows.

– BOLT: a generic framework for parallelizing top-down interprocedural analyses. In particular, the framework targets demand driven analyses such as the ones used in software model checkers.

– An instantiation of the BOLT framework with an intraprocedural analysis algorithm that uses a "may-must" analysis combining testing (in the style of DART and CUTE) and abstraction (in the style of SLAM and BLAST).

– A modular implementation of BOLT, where different intraprocedural algorithms can be plugged-in and parallelized seamlessly; and an extensive experimental evaluation of BOLT, using our may-must-analysis on a number of Microsoft Windows device drivers and safety properties, that demonstrates an average speedup of 3.71x and a maximum speedup of 7.4x using 8 cores (in comparison with a sequential analysis). Using BOLT, we have been able to analyze several driver-property pairs that sequential analyses have been unable to analyze previously.

The rest of the paper is organized as follows: Section 2 provides an overview of BOLT and motivates its style of parallelism. Section 3 is a formal description of the BOLT framework. Section 4 discusses the various instantiations of the intraprocedural parameter PUNCH. Section 5 discusses our implementation and evaluation of BOLT. Section 6 compares BOLT with related work. Finally, Section 7 concludes the paper and outlines directions for future research.



```
int foo(int p_foo);
int bar(int p_bar);
int baz(int p_baz);

main(int i, int j){
  int x, y;
  if (j > 0)
    x = foo(i);
  else if (j > -10)
    x = bar(i);
  else
    x = baz(j);

  y = x + 5;
  assert(y > 0);
}
```

Figure 2. (a) Procedure `main` of example program, (b) State machine of a query Q_i, and (c) Illustration of BOLT on (a).

2. Motivation

In this section, we illustrate the operation of BOLT on a toy program and also motivate BOLT's style of parallelism by examining a real-world application.

2.1 Illustration of BOLT on a Toy Program

Consider the program shown in Figure 2(a) with main procedure `main`. Procedure `main` invokes three other procedures, `bar`, `foo`, and `baz`, which only have their signatures shown. Our goal is to check if there exists some input to `main` that violates the assertion "`assert(y > 0)`" at the end of the procedure. This check is encoded as the following query (formally defined in Section 3) over the procedure `main` .

$$Q_{\texttt{main}} = \langle \mathit{true} \overset{?}{\Longrightarrow}_{\texttt{main}} \texttt{y <= 0} \rangle \tag{1}$$

This query asks the question if there is an execution through the procedure `main` starting in any input state (denoted by the precondition true) and ending in a state satisfying the error condition `y <= 0`.

As shown in Figure 2(c), BOLT alternates between the map and reduce stages in the style of the MapReduce framework [13]. Specifically, BOLT operates on this example as follows.

The first stage is the map stage where BOLT applies PUNCH to the main query $Q_{\texttt{main}}$, which is initially in the Ready state, i.e., ready to be processed (the state machine for a lifespan of a query is shown in Figure 2(b)). This results in the new queries $Q_{\texttt{foo}}$, $Q_{\texttt{bar}}$ and $Q_{\texttt{baz}}$, which are children of $Q_{\texttt{main}}$, all of which are in the Ready state:

$$Q_{\texttt{foo}} = \langle \mathit{true} \overset{?}{\Longrightarrow}_{\texttt{foo}} \texttt{ret <= -5} \rangle \tag{2}$$

$$Q_{\texttt{bar}} = \langle \mathit{true} \overset{?}{\Longrightarrow}_{\texttt{bar}} \texttt{ret <= -5} \rangle \tag{3}$$

$$Q_{\texttt{baz}} = \langle \texttt{p\_baz <= -10} \overset{?}{\Longrightarrow}_{\texttt{baz}} \texttt{ret <= -5} \rangle \tag{4}$$

Here, we assume that the intraprocedural analysis instantiated by PUNCH is able to ascertain that the assertion "`assert(y > 0)`" in the procedure `main` holds if and only if each of the procedures `foo`, `bar`, and `baz`, return a value greater than -5. Note that $Q_{\texttt{baz}}$ has the precondition `p_baz <= -10`, since `baz` is only called with inputs less than or equal to -10.

The query $Q_{\texttt{main}}$ is now in the Blocked state, because it needs an answer from at least one of its child sub-queries before it can make progress. The reduce stage analyzes if any interdependencies between the queries have been resolved, and in this case, none are resolved, so all the queries remain in their respective states (the first reduce stage is essentially a no-op).

The second map stage applies PUNCH, *in parallel*, to each of the Ready queries $Q_{\texttt{foo}}$, $Q_{\texttt{bar}}$, and $Q_{\texttt{baz}}$. We assume that the first two queries, $Q_{\texttt{foo}}$ and $Q_{\texttt{bar}}$, complete at this stage (perhaps due to `foo` and `bar` being leaves in the call-graph) and therefore move to the Done state, and $Q_{\texttt{baz}}$ moves to the Blocked state and generates a new query $Q_{\texttt{roo}}$. The Done state implies that the analysis of the query is complete. BOLT stores the results of Done queries as procedure summaries in a summary database SUMDB, in order to avoid recomputing them. A summary can be either a *must* summary, representing an underapproximation of the procedure and containing a path to error states, or a *not-may* summary, representing an overapproximation of the procedure and excluding paths to error states [19].

Since $Q_{\texttt{foo}}$ and $Q_{\texttt{bar}}$ have moved to a Done state, the reduce stage now sets $Q_{\texttt{main}}$ to a Ready state, and thus enables it to be processed by PUNCH in the next map stage, now that some of its

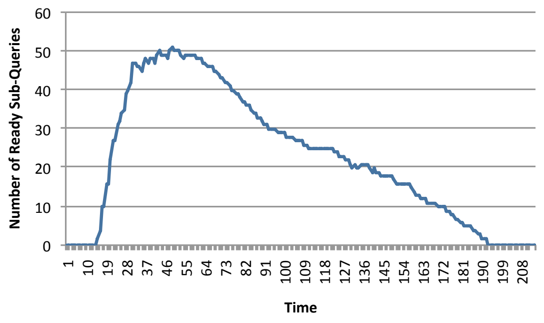


Figure 3. Potential of parallelism illustrated on a device driver.

child sub-queries have returned results (summaries). The reduce stage also deletes all Done queries (and their descendants). In this case, it deletes $Q_{\texttt{bar}}$ and $Q_{\texttt{foo}}$.

In subsequent stages (not shown in the figure), it may so happen that $Q_{\texttt{main}}$ completes due to the answers it gets from the sub-queries $Q_{\texttt{foo}}$ and $Q_{\texttt{bar}}$. If that happens, the next reduce stage will garbage collect the remaining queries such as $Q_{\texttt{baz}}$ and $Q_{\texttt{roo}}$, since their answers are no longer required (i.e., we have been able to answer $Q_{\texttt{main}}$ even without requiring the answers to these sub-queries).

To summarize, BOLT alternates between two phases: the parallel map stage, which applies PUNCH in parallel to verification queries, and the reduce stage, which performs housekeeping, getting rid of unnecessary queries and reactivating blocked ones. The process continues until the main query $Q_{\texttt{main}}$ is answered.

2.2 Opportunity for Parallelism

Since our goal is to parallelize top-down analyses, it is important to ascertain the amount of parallelism available in the query tree induced by a top-down analysis for real-world programs. Thus, for these programs, it is useful to know the number of Ready queries in each successive stage of the MapReduce process in order to understand the level of parallelism that is available.

To get a rough idea of the amount of parallelism available in analyzing device drivers, which are our target application, we instrumented a sequential top-down analysis and recorded the total number of Ready sub-queries over the lifetime of a query for a dispatch routine (the *main* procedure) in a device driver. Figure 3 shows this number plotted against time, for a typical device driver. At its peak, the query has around 50 Ready sub-queries. On this driver, a sequential top-down analysis would analyze each of these Ready sub-queries one at a time. In contrast, BOLT exploits the fact that these Ready queries can be analyzed independently, and, therefore, parallelizes execution of these sub-queries on the available processor cores. Once these sub-queries run in parallel, they will in turn generate more sub-queries with more opportunity for exploiting parallelism. Ultimately, the speedup obtained by BOLT depends on several other factors (see Section 5.1 for a thorough evaluation), but the above methodology is useful to understand the amount of parallelism inherently available in any top-down analysis, and the gains we can expect out of parallelizing the analysis using BOLT.

3. The BOLT Framework

In this section, we describe the BOLT framework together with its intraprocedural parameter PUNCH for solving reachability queries.

3.1 Preliminaries

Programs A program $\mathcal{P}$ is a set of procedures $\{P_0, \ldots, P_n\}$, where P_0 is the main procedure (entry point). A procedure P_i is a tuple $(V_i, N_i, E_i, n_i^0, n_i^x, \lambda_i)$, where

- V_i is the disjoint union of the set of local variables V_i^L of P_i and the set of global variables V_G of $\mathcal{P}$.
- N_i is the set of control nodes (locations).
- $E_i : N_i \times N_i$ is the set of edges between control nodes.
- $n_i^0, n_i^x \in N_i$ are the entry and exit locations, respectively.
- $\lambda_i : E_i \longrightarrow \mathsf{Stmt}$ is a labeling function, where Stmt is the set of program statements over V_i. Statements in Stmt are either *simple statements* or *call statements*. A simple statement in a procedure P_i is an assignment statement `x = ` E or an assume statement `assume`(Q), where `x` is a variable in V_i, E is an expression over the variables V_i, and Q is a Boolean expression over the variables V_i. A call statement to procedure P_j is of the form `call` P_j.

We assume, w.l.o.g., that communication between procedures is performed via the global variables V_G, and for each procedure P_i, there does not exist a node $n \in N_i$ such that $(n_i^x, n) \in E_i$.

Program Model A *configuration* of a procedure P_i is a pair (n, σ), where $n \in N_i$ and the *state* σ is a valuation of variables V_i of P_i. The set of all states of P_i is denoted by Σ_{P_i}. Every edge $e \in E_i$ is a relation $\Gamma_e \subseteq \Sigma_{P_i} \times \Sigma_{P_i}$ defined by the standard semantics of the statement $\lambda_i(e)$.

The initial configurations of a procedure P_i are $\{(n_i^0, \sigma) \mid \sigma \in \Sigma_{P_i}\}$. From a configuration (n, σ), P_i can execute a statement by traversing some edge $e = (n, n') \in E_i$ and reaching a configuration (n', σ'), where $(\sigma, \sigma') \in \Gamma_e$. We say that a configuration (n, σ) can *reach* another configuration (n', σ'), where $n, n' \in N_i$ if and only if there exists a sequence of edges in $(n, n_1), (n_1, n_2), \ldots, (n_m, n') \in E_i$ which if executed from state σ leads to state σ'.

Procedure Summaries For any procedure P_i, let φ_1 and φ_2 be formulae representing sets of states in $2^{\Sigma_{P_i}}$. Then, we have two types of summaries for P_i, *must summaries* and *not-may summaries* defined as follows [19].

- *Must Summary*: $\langle \varphi_1 \overset{must}{\Longrightarrow}_{P_i} \varphi_2 \rangle$ is a must summary for P_i if and only if every exit configuration (n_i^x, σ'), where $\sigma' \in \varphi_2$, is reachable from some initial configuration (n_i^0, σ), where $\sigma \in \varphi_1$.
- *Not-may Summary*: $\langle \varphi \overset{\neg may}{\Longrightarrow}_{P_j} \varphi_2 \rangle$ is a not-may summary for P_i if and only if every initial configuration (n_i^0, σ), where $\sigma \in \varphi_1$, cannot reach any exit configuration (n_i^x, σ'), where $\sigma' \in \varphi_2$.

Queries A *query* Q_i over some procedure P_j is defined as a 4-tuple $(q_i, s_i, p_i, \mathcal{O}_i)$, where

- q_i is a *reachability question* of the form $\langle \varphi_1 \overset{?}{\Longrightarrow}_{P_j} \varphi_2 \rangle$, asking if a procedure P_j starting in a configuration in $\{(n_j^0, \sigma) \mid \sigma \in \varphi_1\}$ can reach a configuration in $\{(n_j^x, \sigma) \mid \sigma \in \varphi_2\}$.
- $s_i \in \{\mathsf{Ready}, \mathsf{Blocked}, \mathsf{Done}\}$ is the *query state*.
- p_i is the index of the parent query Q_{p_i} of Q_i.
- $\mathcal{O}_i$ is a *verification object* that maintains the internal state of a query. The exact nature of this object depends on the kind of analysis being performed by BOLT (may-analysis/must-analysis/may-must-analysis). We will formally describe this in Section 4.

A procedure summary $\mathcal{S}$ can be used to *answer* a reachability question $\langle\varphi_1 \overset{?}{\Longrightarrow}_{P_j} \varphi_2\rangle$ in either of the following ways:

- *Answer = "yes"*, if $\mathcal{S} = \langle\hat{\varphi}_1 \overset{must}{\Longrightarrow}_{P_j} \hat{\varphi}_2\rangle$, where $\hat{\varphi}_1 \subseteq \varphi_1$ and $\varphi_2 \cap \hat{\varphi}_2 \neq \emptyset$,
- *Answer = "no"*, if $\mathcal{S} = \langle\hat{\varphi}_1 \overset{\neg may}{\Longrightarrow}_{P_j} \hat{\varphi}_2\rangle$, where $\varphi_1 \subseteq \hat{\varphi}_1$ and $\varphi_2 \subseteq \hat{\varphi}_2$.

Intuitively, a must-summary $\mathcal{S}$ answers a reachability question $\langle\varphi_1 \overset{?}{\Longrightarrow}_{P_j} \varphi_2\rangle$ with a *"yes, there is an execution from a state in φ_1 to a state in φ_2 through P_j"*. On the other hand, if $\mathcal{S}$ is a not-may summary, then it answers the reachability question with a *"no, there are no executions through P_j from any state in φ_1 to any state in φ_2"*.

A *verification question* for a program $\mathcal{P}$ is a query $Q_0 = (q_0, s_0, p_0, \mathcal{O}_0)$ over its main procedure P_0, where $q_0 = \langle\varphi_1 \overset{?}{\Longrightarrow}_{P_0} \varphi_2\rangle$, φ_2 describes undesirable (error) states, and p_0 is undefined, since the initial query Q_0 does not have any parent queries.

3.2 PUNCH: The Intraprocedural Parameter

BOLT is parameterized by an intraprocedural analysis algorithm PUNCH for manipulating queries. PUNCH takes a query Q_i in the Ready state, and the goal is to either compute a summary that answers the reachability question of Q_i or produce new queries required to answer Q_i. PUNCH stores procedure summaries that it computes in a database SUMDB. PUNCH also queries SUMDB for procedure summaries in order to avoid recomputing answers to queries. The formal specification of PUNCH is described below.

Input: $Q_i = (q_i, s_i, p_i, \mathcal{O}_i)$.

Output: Set of queries R.

Precondition: $s_i = \mathsf{Ready}$.

Postcondition: $R = \{Q_i'\} \cup C$, where $Q_i' = (q_i, s_i', p_i, \mathcal{O}')$ and

1. $(s_i' = \mathsf{Done}) \implies (C = \emptyset)$, and
2. $(s_i' \in \{\mathsf{Blocked}, \mathsf{Ready}\}) \implies$
   $\forall (q_j, s_j, p_j, \mathcal{O}_j) \in C \cdot p_j = i \wedge s_j = \mathsf{Ready}$

PUNCH takes a query $Q_i = (q_i, s_i, p_i, \mathcal{O}_i)$ as input and returns a set of queries R. If PUNCH successfully analyzes Q_i, it returns a copy Q_i' of Q_i in a Done state (formula 1 of the above postcondition), and *adds a summary that answers q_i to* SUMDB, as a side effect. Otherwise, it returns a copy Q_i' of Q_i that is Ready or Blocked, and a set of child sub-queries C of Q_i' (formula 2 of the above postcondition). Every child sub-query $Q_j = (q_j, s_j, p_j, \mathcal{O}_j) \in C$ is uniquely identified by its index j. If a query Q_i is in the Blocked state, PUNCH cannot make any progress with Q_i and can only continue when one of its children has an answer (i.e., the child reaches Done state and adds a summary to SUMDB). If Q_i is in the Ready state, PUNCH can perform more processing on Q_i. Note that the only side-effect of PUNCH is the addition of summaries to SUMDB.

BOLT expects PUNCH to operate as follows. First, PUNCH attempts to answer a query Q_i on some procedure P_j by analyzing P_j using the summaries of the procedures P_j calls that are stored in SUMDB. If it fails to find appropriate summaries for those procedures, it moves Q_i to a Blocked state and produces a number of new sub-queries C. The query Q_i remains Blocked until one of its sub-queries is Done (and, therefore, has a summary in SUMDB). Alternatively, PUNCH may decide to preempt an ongoing analysis of Q_i and return Q_i in a Ready state. In Section 4, we present several instantiations of PUNCH.

```
1:  function BOLT (Program $\mathcal{P}$, Query $Q_0 = (q_0, s_0, p_0, \mathcal{O}_0)$)
2:      QSet = $\{Q_0\}$
3:      while $\neg\exists(q_i, s_i, p_i, \mathcal{O}_i) \in \mathsf{QSet} \cdot s_i = \mathsf{Done} \wedge q_i = q_0$ do
          MAP:
4:          QSet' ← $\biguplus\{\text{PUNCH}(Q_i) \mid Q_i \in \mathsf{QSet} \wedge s_i = \mathsf{Ready}\}$
5:          QSet ← QSet' $\cup \{Q_i \mid Q_i \in \mathsf{QSet} \wedge s_i \neq \mathsf{Ready}\}$
          REDUCE:
6:          for all $Q_i = (q_i, s_i, p_i, \mathcal{O}_i) \in$ QSet do
7:              if $s_i = \mathsf{Done}$ then
8:                  if $s_{p_i} = \mathsf{Blocked}$ then set $s_{p_i}$ to Ready
9:                  (* remove subtree rooted at $Q_i$ from QSet *)
10:                 QSet ← QSet \ Descendants($Q_i$)
11:     if there exists a must summary for $q_0$ in SUMDB then
12:         return "Error Reachable"
13:     else
14:         return "Program is Safe"
```

Figure 4. BOLT algorithm

3.3 BOLT: The Parallel Top-Down Verification Framework

BOLT uses MapReduce [13] and is formally described in Figure 4. BOLT takes as input a program $\mathcal{P}$ and a verification question Q_0 over the main procedure P_0 of $\mathcal{P}$. The algorithm starts with a set of queries QSet that is initialized to the verification question (line 2). Each iteration (lines 3 – 10) is divided into two stages:

1. The MAP stage (lines 4 – 5): Applies PUNCH, *in parallel*, to each query $Q_i \in$ QSet that is in Ready state. QSet' is then assigned the union of all of the results returned by all calls to PUNCH. This is denoted by parallel union symbol $\biguplus$. The only resource shared by parallel instances of PUNCH is the summary database SUMDB.
2. The REDUCE stage (lines 6 – 10): Removes redundant and Done queries from QSet. The function *Descendants*(Q_i) is used to denote the image of the transitive closure of the parent-child relation starting from Q_i. For every query Q_i s.t. $s_i =$ Done, all descendants of Q_i, including Q_i, are removed from QSet, since they were added to QSet to help answer Q_i, and now that $s_i =$ Done, they are no longer required. Additionally, the REDUCE stage sets parents of Done queries to Ready state, as new results about their child queries have been added to SUMDB by PUNCH, potentially enabling parent queries to be Done in the next MAP stage.

The algorithm keeps iterating and executing the MAP and REDUCE stages until q_0 is answered. For a query Q_i, when $s_i =$ Done, SUMDB either contains a must summary or a not-may summary that answers q_i (by definition of PUNCH). Therefore, when BOLT exits the loop at line 3, we know that there exists a summary that answers the reachability question q_0. If q_0 is answered by a must summary, then BOLT returns "Error Reachable", as there is an execution to the error states defined in q_0. On the other hand, if q_0 is answered by a not-may summary, then BOLT returns "Program is Safe", since the not-may summary precludes any execution to an error state in q_0,

Example 1. Recall the example from Section 2.1. In the second iteration of BOLT, the MAP stage applies PUNCH to the Ready queries in QSet: $Q_{\texttt{foo}}$, $Q_{\texttt{bar}}$ and $Q_{\texttt{baz}}$. That is, in the second iteration, QSet is assigned as follows:

$$\mathsf{QSet}' \leftarrow \text{PUNCH}(Q_{\texttt{foo}}) \cup \text{PUNCH}(Q_{\texttt{bar}}) \cup \text{PUNCH}(Q_{\texttt{baz}})$$

$$= \{Q'_{\texttt{foo}}\} \cup \{Q'_{\texttt{bar}}\} \cup \{Q_{\texttt{roo}}, Q'_{\texttt{baz}}\}, \text{ and}$$

$$\mathsf{QSet} \leftarrow \mathsf{QSet}' \cup Q_{\texttt{main}}$$

Note that PUNCH($Q_{\texttt{foo}}$), PUNCH($Q_{\texttt{bar}}$), and PUNCH($Q_{\texttt{baz}}$), are computed in parallel. Subsequently, the REDUCE stage realizes that $Q'_{\texttt{foo}}$ and $Q'_{\texttt{bar}}$ are Done and, therefore, sets $Q_{\texttt{main}}$ to a Ready state and removes $Q'_{\texttt{foo}}$ and $Q'_{\texttt{bar}}$ from QSet.

4. Instantiations of PUNCH

In this section, we will describe how any must-analysis, may-analysis, and may-must-analysis can be suitably modified to meet the specification of PUNCH given in Section 3.2. For a detailed exposition of must-, may-, and may-must-analyses, the reader is referred to [19].

Assume that PUNCH, is given a query $Q_m = (q_m, s_m, p_m, \mathcal{O}_m)$, where $q_m = \langle \varphi_1 \overset{?}{\Longrightarrow}_{P_i} \varphi_2 \rangle$ and $s_m =$ Ready. We start by defining a *must-map* and a *may-map* over procedure P_i as follows:

- **Must-map**: A must-map $\Omega : N_i \rightarrow 2^{\Sigma_{P_i}}$ maps locations $n \in N_i$ of P_i to sets of states, representing an underapproximation of the set of reachable states at that location from states in φ_1 at n_i^0. For each node $n \in N_i$, we use Ω_n to denote $\Omega(n)$. Initially, $\Omega_{n_i^0} = \varphi_1$, and for all $n \in N_i \setminus \{n_i^0\}$, $\Omega_n = \emptyset$.
- **May-map**: A may-map $\Pi : N_i \rightarrow 2^{2^{\Sigma_{P_i}}}$ maps locations $n \in N_i$ of P_i to sets of sets of states (*partitions*), which together represent an overapproximation of the set of states that can reach φ_2 at that location. For each node $n \in N_i$, we use Π_n to denote $\Pi(n)$. Initially, $\Pi_{n_i^x} = \{\varphi_2, \Sigma_{P_i} \setminus \varphi_2\}$, and for every $n \in N_i \setminus \{n_i^x\}$, $\Pi_n = \{\Sigma_{P_i}\}$.

For a node $n \in N_i$, we treat sets of states Ω_n and $\varphi_n \in \Pi_n$ as formulas and use the notation Ω_n^G and φ_n^G to denote versions of Ω_n and φ_n where all local variables are existentially quantified.

In what follows, we sketch how different analyses populate these maps to answer the reachability question q_m.

Must-Analysis A must-analysis explores a subset of the behaviors, or an underapproximation, of a given program, and is therefore useful for proving the presence of errors. For example, DART [16] and CUTE [35] use a combination of symbolic and concrete executions to explore an underapproximation of a program.

In a must-analysis, PUNCH progressively propagates sets of reachable states along edges of the procedure P_i. If at any point $\Omega_{n_i^x} \cap \varphi_2 \neq \emptyset$, then the postcondition φ_2 of q_m is reachable from a state in φ_1, and, therefore, a must-summary that answers q_m can be generated and stored in SUMDB. The verification object $\mathcal{O}_m$ for a must-analysis is the must-map Ω.

The main difference from a typical must-analysis is the way in which PUNCH propagates reachable states over call statements. Given an edge $e = (n, n') \in E_i$ such that $\lambda_i(e)$ is a call statement `call` P_j, PUNCH encodes reachability over this call as the reachability question $\langle \Omega_n^G \overset{?}{\Longrightarrow}_{P_j} \Sigma_{P_j} \rangle$, and first checks whether a must-summary that answers this question is available in SUMDB. If such a must-summary exists in SUMDB, it uses the summary to update the set of reachable states $\Omega_{n'}$ at location n', the destination location of the call-edge e. On the other hand, if a must-summary is unavailable, PUNCH creates a child query Q_k, where $q_k = \langle \Omega_n^G \overset{?}{\Longrightarrow}_{P_j} \Sigma_{P_j} \rangle$, and adds it to R (the set of sub-queries that PUNCH returns, which contains an updated copy of Q_m). In contrast, a regular must-analysis would analyze the procedure P_j and compute reachability information.

If PUNCH successfully computes all reachable states, then it terminates analysis of Q_m. But since a must-analysis is not guaranteed to converge, PUNCH continues to analyze Q_m up to some time limit or an upper-bound on the number of explored paths before it stops analysis and returns a set of child sub-queries R of Q_m. This is to ensure that the MAP stage always terminates. When PUNCH stops its analysis of Q_m, the state of PUNCH, which is the must-map Ω, is saved in $\mathcal{O}_m$, so that the next time Q_m is processed by PUNCH, it can continue exploration from the saved state $\mathcal{O}_m$.

May-Analysis A may-analysis explores an overapproximation of a program's behaviors, and is therefore used to prove absence of errors. For example, software model checkers such as SLAM [5] and BLAST [10] overapproximate the set of states reachable in a program using predicate abstraction [20].

In our case, the goal of a may-analysis is to prove that no execution can reach a state in φ_2 at n_i^x from a state in φ_1 at n_i^0. For every edge $e = (n, n') \in E_i$, we assume that there exists an *abstract edge* between every $\psi_n \in \Pi_n$ and every $\psi_{n'} \in \Pi_{n'}$ (denoted by $\psi_n \rightarrow_e \psi_{n'}$). The may-analysis proceeds by eliminating infeasible abstract edges in order to prove that φ_2 is unreachable. Eliminated abstract edges are stored in the set $\bar{E}$, which is initially empty.

Suppose that for edge $e = (n, n')$, $\lambda_i(e)$ is a simple statement, and that there exists an abstract edge $\psi_1 \rightarrow_e \psi_2$. A may-analysis checks if ψ_1 can reach a state in ψ_2 by taking edge e. In case it cannot, ψ_1 is split into two partitions: $\psi_1 \wedge \theta$ and $\psi_1 \wedge \neg\theta$, where $\mathit{pre}(\lambda_i(e), \psi_2) \subseteq \theta$ and $\mathit{pre}(\lambda_i(e), \psi_2)$ is the preimage of the set of states ψ_2 w.r.t the statement $\lambda_i(e)$. Since no state in $\psi_1 \wedge \neg\theta$ can reach ψ_2, $\bar{E}$ is updated with the edge $(\psi_1 \wedge \neg\theta, \psi_2)$. Intuitively, the partition ψ_1 is refined into a partition that may reach ψ_2, and another one that may not.

Now suppose that $\lambda_i(e)$ is a call statement to some procedure P_j. Then, PUNCH encodes the reachability question $\langle \psi_1^G \overset{?}{\Longrightarrow}_{P_j} \psi_2^G \rangle$. If there exists exists a not-may summary $\langle \widehat{\psi_1} \overset{\neg may}{\Longrightarrow}_{P_j} \widehat{\psi_2} \rangle$ that answers this reachability question, then we know that there are no executions from ψ_1 to ψ_2. Therefore, PUNCH splits ψ_1 into $\psi_1 \wedge \theta$ and $\psi_1 \wedge \neg\theta$, where $\theta \subseteq \widehat{\varphi_1}$, and adds $(\psi_1 \wedge \theta, \psi_2)$ to the set $\bar{E}$. Otherwise, if there does not exist such a summary, PUNCH adds a child query Q_k, where $q_k = \langle \psi_1^G \overset{?}{\Longrightarrow}_{P_j} \psi_2^G \rangle$, to the set R.

As discussed, a may-analysis maintains the map Π and the set of eliminated edges $\bar{E}$. Therefore, when PUNCH returns Q_m in a Ready or Blocked state, $\mathcal{O}_m$ is set to $(\Pi, \bar{E})$. A may-analysis sets the query Q_m to Done when all partitions of n_i^0 intersecting with φ_1 cannot reach a partition of n_i^x intersecting with φ_2, where reachability is defined via abstract edges. As with a must-analysis, for fairness, PUNCH may decide to terminate analysis prematurely and store the state of the analysis in $\mathcal{O}_m$.

May-Must-Analysis Finally, may-must-analyses combine a must-analysis with a may-analysis in order to efficiently find errors as well as prove their absence. The SYNERGY [22] and DASH [9] algorithms are examples of may-must-analyses. Both use testing, symbolic execution and abstraction to check properties of programs. Interpolation-based software model checking algorithms, such as [2, 24, 31], can also be seen as may-must-analyses. Such algorithms also use symbolic executions to error locations to find bugs and, in case of infeasible executions, use interpolants derived from refutation proofs to create an abstraction that eliminates a large number of potential counterexamples.

For the query Q_m, a may-must-analysis maintains Π, Ω, and $\bar{E}$. That is, if PUNCH returns Q_m in a Ready or Blocked state, it sets $\mathcal{O}_m$ to $(\Pi, \Omega, \bar{E})$.

A may-must-analysis only analyzes an abstract transition $\psi_1 \rightarrow_e \psi_2$, where $e = (n, n') \in E_i$ and $\lambda_i(e)$ is a call to some procedure P_j, if $\Omega_n \cap \psi_1 \neq \emptyset$ and $\Omega_{n'} \cap \psi_2 = \emptyset$. That is, only abstract transitions which have been reached by the must analysis, but not taken, are analyzed. These transitions are called "frontiers" in [9, 22].

A may-must-analysis handles such abstract transitions as follows:

```
let bolt () =
  while (Q_0.isNotDone()) do
    QSet := Async.AsParallel
      [for Q_i in QSet -> async {return punch Q_i}];
    reduce ();
  done;
  ...
```

Figure 5. F# implementation of BOLT's main function `bolt`.

1. If there exists a must summary $\langle \widehat{\psi_1} \overset{must}{\Longrightarrow}_{P_i} \widehat{\psi_2} \rangle$ that answers the query $\langle \Omega_n^G \overset{?}{\Longrightarrow}_{P_j} \psi_2^G \rangle$, then we know that there exists an execution from Ω_n to ψ_2 through P_j, and, therefore, PUNCH updates $\Omega_{n'}$ to be $\Omega_{n'} \cup \theta$, where $\theta \subseteq \widehat{\psi_2}$ and $\theta \cap \psi_2 \neq \emptyset$.
2. If there exists a not-may summary $\langle \widehat{\psi_1} \overset{\neg may}{\Longrightarrow}_{P_i} \widehat{\psi_2} \rangle$ that answers the query $\langle \Omega_n^G \overset{?}{\Longrightarrow}_{P_j} \psi_2^G \rangle$, then we know that there are no executions from Ω_n to ψ_2, and, therefore, PUNCH splits region ψ_1 into $\psi_1 \wedge \neg\theta$ and $\psi_1 \wedge \theta$, where $\theta \subseteq \widehat{\varphi_1}$ and $\neg\theta \cap \Omega_n = \emptyset$. Thus, the edge $(\psi_1 \wedge \theta, \psi_2)$ is added to $\bar{E}$.
3. Finally, if neither kind of summaries exist, then a child query Q_k, where $q_k = \langle (\Omega_n \wedge \psi_1)^G \overset{?}{\Longrightarrow}_{P_i} \psi_2^G \rangle$, is added to R.

In a may-must-analysis, PUNCH continues processing a query Q_m until a must summary is produced, a not-may summary is produced, or all abstract edges have been analyzed and child queries have to be answered to continue processing. Similar to may- and must-analysis, PUNCH may decide to terminate analysis prematurely.

In summary, we have shown how PUNCH can be instantiated with various classes of analyses, which encompass a large number of already published algorithms from the literature. In the following section, we discuss a may-must implementation of PUNCH, based on DASH [9], and describe our empirical evaluation of BOLT on a number of Microsoft Windows device drivers.

5. Implementation and Evaluation

We have implemented BOLT in the F# programming language and integrated it with the Static Driver Verifier toolkit (SDV) [6] for Microsoft Windows device drivers. Our implementation adopts a pluggable architecture, where different instantiations of PUNCH that adhere to the specification in Section 3.2 can be easily integrated into our tool. The main function `bolt` of our implementation is partially shown in Figure 5. Our implementation assumes that `punch` is a pure function (except for its communication with SUMDB) that takes a query as input and returns a set of queries. As a result, the only resource shared between different threads of execution is the summary database SUMDB. This provides a natural framework where program analysis designers can plug in their intraprocedural analyses and automatically produce parallelized interprocedural analyses, without having to be faced with the intricacies of parallelizing individual analyses.

The instance of PUNCH that we have implemented is a variant of the may-must DASH algorithm [9] as described in Section 4. Our implementation of PUNCH uses the Z3 SMT solver [12] for satisfiability checking and can handle C programs with primitive datatypes, structured types, pointers, and function pointers. Checking if a summary answers a query is done via a call to the SMT solver.

5.1 Experiments

We now present our experimental setup and results for the BOLT algorithm.

Statistic	
Total time taken (sequential)	26 hours
Total time taken (parallel)	7 hours
Average observed speedup	3.71x
Maximum observed speedup	7.41x

Table 2. Cumulative results for BOLT on 50 checks (#threads=64, #cores=8).

Experimental Setup Our goal is to study the scalability of BOLT. We do this by measuring the speedup of BOLT over a sequential may-must top-down analysis. To precisely measure and study the scalability of the algorithm, we introduce an artificial throttle that allows us to limit the number of threads that can perform queries in parallel. The artificial throttle has the effect of limiting the total number of physical cores available to the algorithm, when the number of maximum threads is less than the total physical cores available, which also imposes a bound on the number of Ready queries processed by PUNCH in the MAP stage. In cases where the number of allowed maximum threads is greater than the number of physical cores available, the .NET environment (since BOLT is implemented using F#), thread scheduling, and contention play an important part in determining the final overall speedup. It should be noted that the theoretical limit of the speedup that can be achieved on a given test machine is N, where N is the total number of available physical cores, unless super linear speedup can be achieved by the parallel algorithm eliminating work that the sequential algorithm must perform.

We ran our experiments on an HP workstation with 8 Intel Xeon 2.66 GHz cores and 8 GB of memory. We set our initial number of maximum concurrent threads to be 1 (representing a sequential analysis) and doubled it for every subsequent run. The upper limit for the maximum number of concurrent threads is 128. All the experiments were run with resource limitations of 3000 seconds (wall clock time) and 1800 MB of memory.

Our evaluation of BOLT was over a test suite consisting of 45 Microsoft Windows device drivers and 150 safety properties[1]. For the purposes of reporting results, we select all checks (that is, driver-property pairs) where the sequential version of BOLT requires at least 1000 seconds to generate a proof. These are interesting checks (total of 50 checks) where a lot of computation is required and, in some cases, the sequential analysis is unable to produce a result. Therefore, they serve as challenge problems for BOLT. It turns out that all these checks were cases where the program satisfies the property (and thus a proof is reported by BOLT). For a study of effects of may-must summaries on the efficiency of the analysis, we refer the reader to [19, 33].

Results The cumulative results are presented in Table 2. For the 50 checks, where the sequential algorithm takes at least 1000 seconds for generating a proof, the average observed speedup using BOLT was 3.71x. The maximum observed speedup was 7.41x. These results were obtained using 8 cores, and a maximum of 64 threads.

We now analyze a sample of the 50 checks in greater detail. Table 1 shows the detailed results for 6 checks. Each row represents a single check and the time/speedup observed for different configurations of the maximum number of concurrent threads allowed. The speedup is calculated as the ratio of the time taken by the parallel version of BOLT (with 2 threads, 4 threads, etc.) to that of the sequential version of BOLT (with 1 thread).

[1] A subset of our benchmarks is available as part of the SDV-RP toolkit: `http://research.microsoft.com/en-us/projects/slam`.

Property / Max. Number of Threads	1	2		4		8		16		32		64		128	
	Time	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
Driver: `toastmon` (25KLOC)															
`PendedCompletedRequest`	1006	328	3.07	345	2.91	373	2.70	297	3.39	221	4.55	219	4.59	223	4.51
`PnpIrpCompletion`	2224	929	2.39	827	2.69	1041	2.14	599	3.71	300	7.41	300	7.41	300	7.41
Driver: `parport` (2KLOC)															
`MarkPowerDown`	1821	688	2.65	573	3.18	543	3.35	460	3.96	313	5.82	326	5.58	328	5.55
`PowerDownFail`	1916	673	2.85	566	3.38	524	3.65	398	4.81	305	6.28	315	6.08	318	6.03
`PowerUpFail`	2040	718	2.84	691	2.95	689	2.96	565	3.61	306	6.67	315	6.48	300	6.80
`RemoveLockMnSurpriseRemove`	1794	678	2.65	576	3.11	538	3.33	405	4.43	311	5.77	315	5.69	315	5.69

Table 1. Average observed time (in seconds) and speedup of parallel BOLT compared to sequential BOLT for varying number of maximum concurrent threads (#cores=8).

Driver	Property	Result		Time
		Seq	Parallel	
`daytona`	`IoAllocateFree`	TO	Proof	2800
`mouser`	`NsRemoveLockMnRemove`	TO	Proof	2743
`featured1`	`ForwardedAtBadIrql`	TO	Proof	2966
`incomplete2`	`RemoveLockForwardDeviceControl`	TO	Proof	1205
`selsusp`	`IrqlExAllocatePool`	TO	Proof	1951

Table 3. Driver and property combinations where BOLT was able to produce a proof (#cores = 8) and the sequential (Seq) version ran out of time (TO).

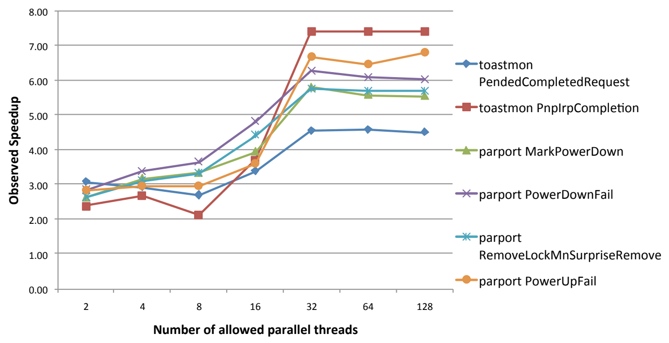


Figure 6. Measured speedup of the BOLT algorithm relative to the sequential version (#cores=8).

For the `toastmon` driver and the `PnpIrpCompletion`[2] property, we see that the speedup achieved with a maximum of 2 concurrent threads is 2.39. As the number of threads is increased to 128, the observed speedup of 7.41 reaches close to the theoretical maximum speedup achievable on the test machine (8 cores). For other checks in the table, we see that, in general, the observed speedup increases as the number of maximum concurrent threads is increased. Figure 6 illustrates the speedups reported in Table 1. The super linear speedup observed in some cases is related to the query processing order, which is discussed in the latter part of this section. In general, we find that the BOLT algorithm always achieves speedup, with the possibility of providing super linear speedup in some cases.

Table 3 shows checks where BOLT successfully produces a proof, whereas the sequential analysis runs out of resources. As can be seen from the table, in many cases, the time taken by BOLT is very close to the timeout limit of 3000 seconds, indicating the degree of difficulty and high amount of computation that is necessary to successfully complete the verification. In general, the speedups achieved by BOLT are due to parallelism, as well as the scheduling of queries. That is, sequential implementations tend to have a fixed deterministic exploration strategy of the query tree and are therefore unable to discover invariants in many cases. On the other hand, BOLT, due to its inherent parallel exploration, may compute invariants that the sequential version cannot discover.

In-depth Analysis To further understand the behavior of BOLT, we make the following measurements in our experiments.

- *The number of queries that are processed as verification progresses*. Figure 7 shows the number of queries processed in parallel for varying numbers of maximum threads (2 – 128) on the `PnpIrpCompletion` property of the `toastmon` driver. Note that, in Figure 7(f), there are two lines present on the chart, but due to the fact that they are exactly equivalent, it appears as only a single line.

  From the graphs, we see that when the number of maximum concurrent threads is less than 16, there is always a constant amount of work that is to be performed and all the threads are always close to 100% utilization. When the number of maximum threads is increased to more than 16, depending on the nature of the input program, mostly, an insufficient number of queries are produced. This results in only a few threads being utilized in an efficient and maximal manner, which explains the lower than expected observed speedup. Figures 7(e) and (f) clearly illustrate this fact. We can see that the number of queries processed in parallel is very inconsistent and never reaches the allowed maximum of 64 or 128. In particular, since a small number of queries need to be made, the graph for 64 and 128 is exactly the same.

- *Total number of queries that have to be performed when varying the degree of allowed concurrency/parallelism*. As the number of maximum concurrent threads is increased, the order of the queries performed changes. This can have two possible outcomes. First, the order of the queries may impact the verification positively, since an important fact can be learned earlier in the verification, which in turn reduces the number of queries that have to be performed. The net effect is that in some cases super linear speedup is observed (as seen in Table 1). Second, the order of the queries may result in an increased number of queries (due to redundancy), which elongates the verification task and reduces the observed speedup relative to the maximum theoretical speedup. Table 4 shows the total number of queries made for different numbers of maximum concurrent threads allowed for the two properties listed for the `toastmon` driver. As we can see in the the case of the `PnpIrpCompletion` property, the algorithm performs only 1.2 times as many queries for 128 threads as it did for 2 threads. But for the `PendedCompletedRequest` property, the algorithm performs 3.5 times as many queries. This fact directly relates to the lower observed speedup for the `PendedCompletedRequest` check, in comparison with the `PnpIrpCompletion` check (shown in Table 1).

[2] Visit http://msdn.microsoft.com/en-us/library/ff551714.aspx for a list of properties.

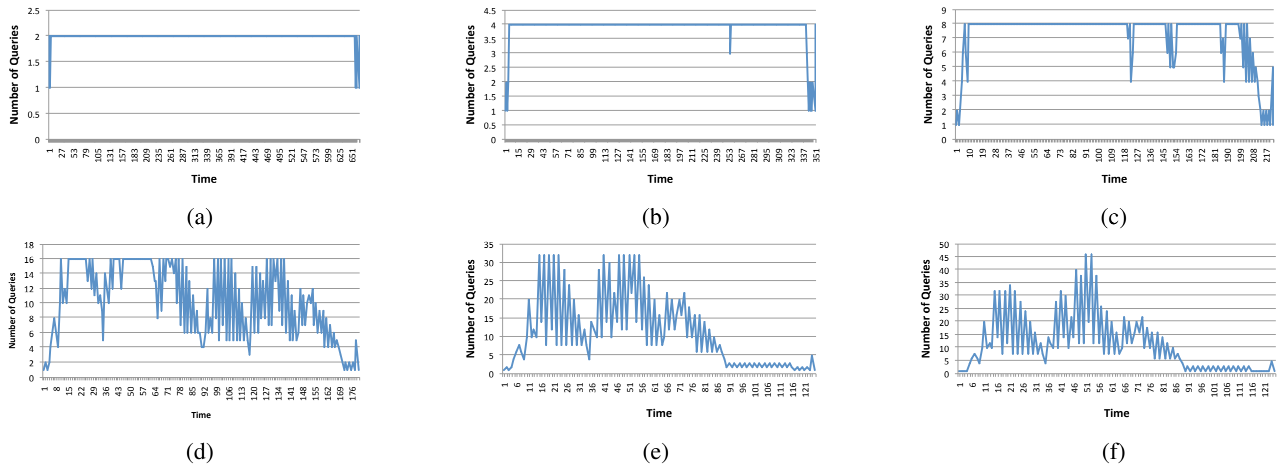


Figure 7. Number of concurrent queries performed over time for the driver `toastmon` and the property `PnpIrpCompletion` (#cores=8). Maximum number of threads for subfigures (a), (b), (c), (d) , (e) and (f) are 2, 4, 8, 16, 32, and 64 respectively. For 128 threads, the results are identical to 64 threads.

	Maximum Number of Threads						
Property	2	4	8	16	32	64	128
`PendedCompletedRequest`	873	1374	2213	2479	2529	3005	3078
`PnpIrpCompletion`	1198	1369	1614	1854	1383	1429	1429

Table 4. Total number of queries performed during verification for various degrees of parallelism on the `toastmon` driver (#cores=8).

Discussion. Our implementation and experiments shows that for may-must-analysis BOLT provides an average speedup of 3.7x and maximum speedup of 7.4x with 8 cores. This has enabled us to complete verification runs that sequential analyses have been unable to analyze previously. In particular, we have been able to complete every driver-property benchmark we have with BOLT, several of which we have previously found impossible to complete. Furthermore, each run of PUNCH needs to load only the procedure under analysis into memory, except for whole program information such as alias analysis, which can be stored in the database. Thus, we get significant advantages in memory savings, in addition to savings in time.

Our detailed investigation into the amount of parallelism available showed that, for the current set of benchmarks we have, increasing thread-level parallelism stops speeding up the analysis after 64 threads, since each MAP stage analyzes small number of queries. To this end, we see two avenues for efficiently utilizing a larger number of cores: (1) getting larger benchmarks with potential for even larger parallelism; and (2) designing a speculative extension to BOLT, where we can speculate on potential queries that will be made and analyze them even before the queries are actually created, thereby, generating more parallelism using speculation. We leave these directions for future work.

6. Related Work

Parallel and distributed algorithms for static analysis and testing is an active area of research [15, 29, 32, 34]. The unique feature of the BOLT framework is that it parallelizes interprocedural analyses that work in a top-down and demand-driven manner. Also, in contrast to earlier efforts to parallelize static and dynamic analysis, BOLT offers a pluggable architecture which inherits the nature of its underlying intraprocedural analysis (as described in Section 4). In this section, we place BOLT in the context of related work. Specifically, we compare BOLT with other parallel static analyses, as well as parallel finite state verification (model checking) and exploration techniques.

Parallel/Distributed Static Analysis Techniques As discussed in Section 1, bottom-up interprocedural analyses are amenable to parallelization due to the decoupling between callers and callees. For example, the SATURN software analysis tool [1] employs a bottom-up analysis and has been shown to scale to the entire Linux kernel, both in the interprocedurally path-insensitive [37] and path-sensitive settings [14]. In comparison, BOLT targets demand-driven top-down analyses, and is parameterized by the algorithm used for intraprocedural analysis. Therefore, BOLT can be easily applied to parallelizing existing software model checking and dynamic test generation techniques, for example, as we have shown with our instantiation of PUNCH to a DASH-like [9, 19] may-must algorithm.

Microsoft's Static Driver Verifier toolkit uses manually created harnesses [7], which specify a set of independent device driver entry points in order to create an embarrassingly parallel workload. On the other hand, BOLT is automatically able to exploit parallelism that occurs at finer levels of granularity. We believe that both techniques complement each other and their combination has the potential for greater scalability.

Lopes et al. [29] propose a distributed tree-based [25] software model checking algorithm based on the CEGAR [11] framework. The tree unrolling of the control flow graph of a program, consisting of a single procedure, is distributed amongst multiple machines. We summarize the differences between [29] and BOLT as follows: (1) [29] is intraprocedural and does not reuse procedure summaries as BOLT does. (2) BOLT is not restricted to a specific intraprocedural analysis. In fact, we believe that the distributed algorithm of [29] can be easily adapted to be an implementation of the intraprocedural parameter PUNCH. (3) The instantiation of PUNCH we present here does not use predicate abstraction and, therefore, skips the expensive step of computing an abstract post transformer required in [29].

In [32], Monniaux describes a parallel implementation of the Astrée abstract interpretation-based static analyzer. At certain branching locations (dispatch points), instead of analyzing program paths in sequence, they are analyzed in parallel. On three embedded software applications, the experiments showed around 2x speedup on 5 processors.

Several other parallel static analysis techniques have been recently proposed. One prominent example is EigenCFA [34], where

GPUs are used to parallelize higher-order control-flow analysis. The authors report up to 72x speedup, when compared to other sequential techniques.

Parallel/Distributed Exploration Techniques In the finite state verification arena, several parallel/distributed model checking algorithms have been proposed, e.g., [8, 21, 36]. At a high level, the methodology adopted by these techniques involves partitioning the state space and distributing the search to several threads or processors. Both [8] and [36], propose distribution strategies for explicit state model checking, where each node is responsible for exploring a partition of the state space. The authors of [27] propose a load balancing strategy to improve the performance of the state space partitioning algorithms to achieve higher speedup. Similarly, the technique in [21] partitions BDDs in symbolic model checking and distributes them to several threads. In [26], a multi-core algorithm along with load balancing techniques geared towards shared memory systems is proposed. In [15], a method is proposed for distributed randomized state space search in the context of Java Path Finder (JPF) [23]. In contrast to the above techniques, BOLT exploits a program's procedural decomposition in order to efficiently parallelize top-down demand-driven analyses.

7. Conclusion and Future Work

Modularity plays a central role in most scalable program analyses. Modular analyses are either top-down, where analysis starts at the main procedure and descends through the call-graph, or bottom-up, where analysis starts with the leaves of the call-graph and goes upwards to the main procedure. Top-down analyses are extensively used in software model checking and test generation. However, unlike their bottom-up counterparts, they are hard to parallelize due to the fine-grained interactions between analysis instances (queries).

In this paper, we have presented BOLT: a parameterized framework for parallelizing top-down interprocedural analyses. BOLT adopts a MapReduce-like strategy for parallelizing processing of reachability queries over multiple procedures in a top-down analysis. We have shown how a number of may-, must-, and may-must-analyses can be parallelized using BOLT. We have also demonstrated the strength of BOLT by parallelizing a may-must analysis. Our experimental results on device drivers showed that BOLT scales well on an 8-core machine, and is able to verify several checks where the sequential version runs out of resources.

We see a number of opportunities for continuing research in this direction:

- **Distributed BOLT** : BOLT's MapReduce architecture permits it to be easily implemented as a distributed application using programming models for large scale distributed systems [38]. As noted in [1], the limiting factor for scaling any analysis to very large programs is memory and not time, and this forms the primary motivation for performing a bottom-up analysis. Indeed, our experience with BOLT also indicates that its memory usage is significantly smaller than that the corresponding sequential top-down analysis. Therefore, we believe that we can distribute the parallelism that BOLT exposes across large-scale clusters of machines and, thereby, scale top-down analysis to very large programs.
- **Deriving value out of larger number of cores**: Our investigation showed that, on our set of benchmarks, increasing thread-level parallelism stops speeding up the analysis beyond 64 threads, due to limitations in the number of queries available for the MAP stage. Thus, we see two directions for efficiently utilizing a larger number of cores and, potentially, distributing the analysis: (1) getting larger benchmarks with potential for even larger parallelism, and (2) designing a speculative extension to BOLT, where we can predict what queries will be made and analyze them even before the queries are actually created, thereby, generating more parallelism using speculation.
- **Other instantiations of BOLT** : We would also like to experiment with other instantiations of PUNCH. For example, it would be interesting to apply BOLT for fuzz testing large scale systems in the style of SAGE [18]. Another interesting application for BOLT is the verification of concurrent programs. Recent research on reducing concurrent analysis to a sequential analysis [28] makes it possible to instantiate PUNCH so that BOLT is able to verify properties of concurrent programs.

Acknowledgments

We thank Tony Hoare, Vlad Levin, Jakob Lichtenberg and Robert Simmons for many useful discussions that shaped our work and the anonymous reviewers for their insightful comments and suggestions.

References

[1] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, and P. Hawkins. An overview of the Saturn project. In *Program Analysis for Software Tools and Engineering (PASTE 2007)*, pages 43–48, 2007.

[2] A. Albarghouthi, A. Gurfinkel, and M. Chechik. Whale: An interpolation-based algorithm for interprocedural verification. In *Verification, Model Checking, and Abstract Interpretation (VMCAI 2012)*, 2012.

[3] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN Workshop on Model Checking of Software (SPIN 2000)*, pages 113–130, 2000.

[4] T. Ball and S. K. Rajamani. The SLAM toolkit. In *Computer Aided Verification (CAV 2001)*, pages 260–264, 2001.

[5] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic Predicate Abstraction of C Programs. In *Programming language design and implementation (PLDI'01)*, pages 203–213. ACM, 2001.

[6] T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In *Integrated Formal Methods (IFM 2004)*, pages 1–20, 2004.

[7] T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with SLAM. *Communications of the ACM*, 54:68–76, July 2011.

[8] J. Barnat, L. Brim, and J. Stribrna. Distributed LTL model-checking in SPIN. In *SPIN Workshop on Model Checking of Software (SPIN 2001)*, pages 200–216, 2001.

[9] N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from tests. In *International Symposium on Software Testing and Analysis (ISSTA 2008)*, pages 3–14, 2008.

[10] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. "The Software Model Checker BLAST". *STTT: International Journal on Software Tools for Technology Transfer*, 9(5-6):505–525, 2007.

[11] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Computer Aided Verification (CAV 2000)*, pages 154–169, 2000.

[12] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, pages 337–340, 2008.

[13] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Operating Systems Design and Implementation (OSDI 2004)*, pages 137–150, 2004.

[14] I. Dillig, T. Dillig, and A. Aiken. Sound, complete and scalable path-sensitive analysis. In *Programming Language Design and Implementation (PLDI 2008)*, pages 270–280, 2008.

[15] M. B. Dwyer, S. G. Elbaum, S. Person, and R. Purandare. Parallel randomized state-space search. In *International Conference on Software Engineering (ICSE 2007)*, pages 3–12, 2007.

[16] P. Godefroid. Compositional dynamic test generation. In *Principles of Programming Languages (POPL 2007)*, pages 47–54, 2007.

[17] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Programming Language Design and Implementation (PLDI 2005)*, pages 213–223, 2005.

[18] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Network and Distributed System Security Symposium (NDSS 2008)*, 2008.

[19] P. Godefroid, A. Nori, S. Rajamani, and S. Tetali. Compositional may-must program analysis: Unleashing the power of alternation. In *Principles of Programming Languages (POPL 2010)*, pages 43–56, 2010.

[20] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Computer Aided Verification (CAV 1997)*, pages 72–83, 1997.

[21] O. Grumberg, T. Heyman, N. Ifergan, and A. Schuster. Achieving speedups in distributed symbolic reachability analysis through asynchronous computation. In *Correct Hardware Design and Verification Methods (CHARME 1995)*, pages 129–145, 2005.

[22] B. Gulavani, T. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. SYNERGY: A new algorithm for property checking. In *Foundations of Software Engineering (FSE 2006)*, pages 117–127, 2006.

[23] K. Havelund and T. Pressburger. Model Checking Java Programs Using Java Pathfinder. *International Journal on Software Tools for Technology Transfer*, 1999.

[24] M. Heizmann, J. Hoenicke, and A. Podelski. Nested interpolants. In *Principles of Programming Languages (POPL 2010)*, pages 471–482, 2010.

[25] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Principles of Programming Languages (POPL 2002)*, pages 58–70, 2002.

[26] G. J. Holzmann and D. Bosnacki. The design of a multicore extension of the SPIN model checker. *IEEE Transactions on Software Engineering*, 33:659–674, October 2007.

[27] R. Kumar and E. G. Mercer. Load balancing parallel explicit state model checking. *Electronic Notes in Theoretical Computer Science*, 128:19–34, 2005.

[28] A. Lal and T. W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design (FMSD)*, 35(1):73–97, 2009.

[29] N. P. Lopes and A. Rybalchenko. Distributed and predictable software model checking. In *Verification, Model Checking, and Abstract Interpretation (VMCAI 2011)*, pages 340–355, 2011.

[30] K. McMillan. Lazy annotation for program testing and verification. In *Computer Aided Verification (CAV 2010)*, pages 104–118, 2010.

[31] K. L. McMillan. Lazy abstraction with interpolants. In *Computer Aided Verification (CAV 2006)*, pages 123–136, 2006.

[32] D. Monniaux. The parallel implementation of the astrée static analyzer. In K. Yi, editor, *APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 86–96. Springer, 2005. ISBN 3-540-29735-9.

[33] A. V. Nori and S. K. Rajamani. An empirical study of optimizations in Yogi. In *International Conference on Software Engineering (ICSE 2010)*, pages 355–364, 2010.

[34] T. Prabhu, S. Ramalingam, M. Might, and M. Hall. EigenCFA: accelerating flow analysis with GPUs. In *Principles of Programming Languages (POPL 2011)*, pages 511–522, 2011.

[35] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Foundations of Software Engineering (ESEC-FSE 2005)*, pages 263–272, 2005.

[36] U. Stern and D. L. Dill. Parallelizing the Murphi Verifier. In *Computer Aided Verification (CAV 1997)*, pages 256–278, 1997.

[37] Y. Xie and A. Aiken. Scalable error detection using Boolean satisfiability. In *Principles of Programming Languages (POPL 2005)*, pages 351–363, 2005.

[38] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. In *Operating Systems Design and Implementation (OSDI 2008)*, pages 1–14, 2008.