A Parallel Abstract Interpreter for JavaScript

Kyle Dewey Vineeth Kashyap Ben Hardekopf

University of California, Santa Barbara

{kyledewey,vineeth,benh}@cs.ucsb.edu

Abstract

We investigate parallelizing flow- and context-sensitive static analysis for JavaScript. Previous attempts to parallelize such analyses for other languages typically start with the traditional framework of sequential dataflow analysis, and then propose methods to parallelize the existing sequential algorithms within this framework. However, we show that this approach is non-optimal and propose a new perspective on program analysis based on abstract interpretation that separates the analysis into two components: (1) an embarrassingly parallel state exploration of a state transition system; and (2) a separate component that controls the size of the state space by selectively merging states, thus injecting sequential dependencies into the system. This perspective simplifies the parallelization problem and exposes useful opportunities to exploit the natural parallelism of the analysis.

We apply our insights to parallelize a JavaScript abstract interpreter. Because of JavaScript's dynamic nature and tricky semantics, static analysis of JavaScript is difficult to scale—one of our benchmarks with only 2.8 KLOC takes over 22 hours to analyze using the sequential JavaScript abstract interpreter. Thus, JavaScript is an excellent case study for the benefits of our approach. Our resulting parallel implementation sees significant benefits on real-world JavaScript programs, with speedups between 2–4× on average with a superlinear maximum of 36.9× on 12 hardware threads.

1. Introduction

JavaScript is prevalent on a wide variety of platforms, including the web, mobile phones, desktops, and servers. Static analysis for JavaScript is a necessity to help build developer tools to construct and review secure, fast, maintainable, and correct JavaScript code. In order to be useful, such JavaScript analyses need to be precise and to run within a reasonable amount of time. However, JavaScript's inherently dynamic nature makes precise static analysis very expensive. As an anecdotal example, we have observed a particular 2,800 line JavaScript program[1] on which a sequential JavaScript analysis that computes data and control dependencies takes over 22 hours to complete. While in the early days of its introduction JavaScript programs tended to be small, simple scripts, today there are many complex JavaScript applications with tens to hundreds of thousands of lines of code. Thus, there is a need to increase JavaScript analysis performance while maintaining high levels of precision.

A heretofore unexplored option is to *parallelize* the JavaScript analysis, thus exploiting the prevalence of modern multicore architectures. The idea of parallel program analysis is not novel; there are many existing parallel program analysis frameworks [10, 14, 15, 23, 24, 26–28, 32]. However, most of these efforts are aimed at first-order, statically-typed, highly imperative languages such as C or Fortran; JavaScript presents new challenges that must be addressed. Our novelty lies not only in the first parallel JavaScript static analysis, but also in the approach with which we design our parallel analysis, which potentially could benefit parallel analysis of other languages.

Key Insight. We focus on parallel analyses that are flow- and context-sensitive, because we need a high level of precision to successfully analyze JavaScript. Almost all such precise parallel analyses in existing work are based on traditional dataflow analysis (DFA) [20, 22]. Our key insight is that the DFA framework inextricably mixes decisions about synchronization and granularity with the definition of the analysis itself, thus limiting opportunities to fully exploit possible parallelism in such analyses. We identify an alternate approach to program analysis more amenable to parallelization, based on ideas from abstract interpretation. Using this approach, we can phrase an analysis as two separate and independent components:

- An abstract semantics that represents the static analysis as a state transition system (STS). The analysis is defined as a reachable-states computation on the STS: given a program and its initial state, the analysis finds all abstract program states potentially reachable from that initial state. This reachable-states computation is embarrassingly parallel in nature, because each state is inherently independent from all other states.
- A separate mechanism for selectively merging multiple abstract states into a single abstract state by over-approximating the information in the states being merged

[1] `linq_aggregate` with `stack-5-4`

2015 IEEE/ACM International Symposium on Code Generation and Optimization
978-1-4799-8161-8/15/$31.00 ©2015 IEEE

together. This merging takes place during the reachability computation and is used to bound the reachable state space in a sound manner. This mechanism effectively merges branches of the computation tree formed by the STS, turning it into a DAG and thus adding sequential behavior (and synchronization points) into the otherwise parallel reachable-states computation.

From this perspective the static analysis itself is trivially a massively parallel problem; this parallelism is then limited by a strategy that determines how and when states are merged (introducing synchronization into the analysis). Using this approach, opportunities for parallelizing the analysis become more obvious than the previous DFA-based approaches. In fact, existing approaches can be re-defined as limited instances of our framework. While it is possible to derive this overall insight purely from a DFA standpoint, it is not possible to act upon it within the DFA framework because DFA intertwines and conflates the two above-described components in an inseparable way.

Our new perspective provides a useful framework for designing parallel analyses, but there is still a large design space to be explored. The strategy for merging states controls the level of synchronization required by the analysis, as well as the size of the state space being explored; thus it has a strong impact on parallelism. In addition, while the normal reachable-states computation is embarassingly parallel, that does not necessarily mean that taking full advantage of its inherent parallelism is the best course—there are many different possible levels of granularity which may provide performance benefits and tradeoffs.

In essence, in our approach the problem of parallelizing an analysis boils down to two decisions: determining a strategy for merging states, and selecting a particular level of granularity at which to operate. In this paper we explore several such design points, discussing their rationales and implications. We include a novel parallelization strategy based on function contexts.

Contributions. The specific contributions of this work are the following:

1. A new perspective on the design of parallel program analyses, based on formulating the analysis as a state transition system plus a separate state merging strategy. (Section 3.1)
2. A language-agnostic exploration of the design space of this parallelization framework, including a novel parallelization strategy based on function contexts. (Section 3.2)
3. Our implementation of these ideas for JSAI, an abstract interpreter for JavaScript that computes a fundamental analysis for JavaScript—performing a combination of type inference, alias analysis, control-flow analysis, and string, numeric, and boolean value analysis. (Section 4)
4. An evaluation of our resulting parallel JavaScript abstract interpreter. Speedups are typically in the 2-4× range on 12 hardware threads, ranging as high as 36.9× . (Section 5)
5. A publicly available implementation.[2]

2. Background and Related Work

In this section we provide a brief background on sequential dataflow analysis (DFA) and describe related work on parallelizing program analysis, much of which is based on DFA.

2.1 Sequential Dataflow Analysis

DFA-based analysis is carried out on the program's *control-flow graph* (CFG), which is a directed graph $G = \langle N, E \rangle$ where N is a finite set of nodes corresponding to program statements and $E \subseteq N \times N$ is a set of edges corresponding to the possible control-flow between statements. The possible analysis solutions are structured into a lattice $\mathcal{L} = (Solns, \sqsubseteq, \sqcup, \sqcap)$, where the most-precise solution is at the bottom $\bot$ of the lattice and the least-precise solution is at the top $\top$ of the lattice.[3]

Each node k of the CFG maintains two lattice elements corresponding to the analysis solutions immediately before and immediately after that statement: IN_k represents the incoming solution, and OUT_k represents the outgoing solution. At the beginning of the analysis $\text{IN}_k = \text{OUT}_k = \bot$ for every k. Each node k has a transfer function $\mathcal{F}_k$ that transforms IN_k to OUT_k. For all nodes k, the analysis iteratively computes the following two functions until the analysis reaches a fixpoint:

$$\text{IN}_k = \bigsqcup_{x \in pred(k)} \text{OUT}_x \quad (1)$$

$$\text{OUT}_k = \mathcal{F}_k(\text{IN}_k) \quad (2)$$

In other words, for each node merge the outgoing information from all immediate predecessor nodes (using the lattice join operator) to get that node's incoming solution, and then apply that node's transfer function to get that node's outgoing solution. The fixpoint computation is usually performed using a worklist. The worklist is initialized to contain the program's entry node; the analysis iteratively performs the following actions until the worklist is empty (signaling the fixpoint has been reached): pop a node k from the worklist; compute IN_k and OUT_k; if OUT_k is changed from its previous value then put all successor nodes of k onto the worklist.

The Importance of Node Ordering. The order in which the worklist processes nodes is irrelevant in terms of correctness, i.e., the analysis will compute the same solution

[2] `www.cs.ucsb.edu/~pllab` under Downloads.

[3] This is actually opposite of the convention usually used by DFA, which reverses the lattice described above; we do this to be consistent with the abstract interpretation convention used later in the paper.

regardless of node ordering. However, it turns out to have significant impact on analysis performance. Intuitively, a bad node ordering can cause paths in the CFG to be redundantly recomputed many times. Suppose a node k is computed to have $\text{OUT}_k = \ell$ where lattice element $\ell \in \mathit{Solns}$, and this information is propagated by the worklist down the CFG paths starting from k. Later the worklist processes a node that is a predecessor to k, causing k to be processed again, and now $\text{OUT}_k = \ell'$ where $\ell \sqsubseteq \ell'$. Then this new information must be propagated down the CFG again, subsuming the previous solutions along those paths. In the worst case those paths could be recomputed h times where h is the height of the lattice. Thus, a good node ordering is important for performance.

2.2 Parallelizing Program Analysis

We categorize the related work on parallelizing flow- and context-sensitive program analysis into three general approaches. We leave out work on parallelizing flow- or context-*insensitive* analysis, such as that by Méndez-Lojo et al. [26, 27], Edvinsson et al. [15], and Nagaraj et al. [29].

Worklist-Based Parallelism. This parallelization strategy operates by processing all nodes currently enqueued on the analysis worklist in parallel. Dwyer et al. [14] discuss a worklist-parallel implementation of the FLAVERS DFA toolset [13] for C. They start a new thread for each node in a global worklist and each thread enqueues its result back in that worklist. The authors report average speedups of 3.8× on 6–9 hardware threads. However, the paper's evaluation is problematic in two respects, making it difficult to interpret the results: (1) the sequential analysis they compare against used an arbitrary node ordering for the worklist, which in our experience can cause slow-downs from 2–5× relative to a more optimized node ordering strategy; and (2) their evaluation reports analysis runtimes rather than speedups.[4]

Nondeterminism-Based Parallelism. This parallelization strategy looks for nondeterministic branch points in the analysis (e.g., conditional guards with indeterminate truth values) and executes the branches in parallel until control-flow merges again (e.g., after the conditional is finished). This approach is taken by Monniaux [28], who describes a parallel implementation of the Astrée static analyzer [11] for embedded controller code written in C. The parallel implementation exploits the fact that, in this particular application domain, programs often contain dispatch loops over a `switch` statement, and each `case` within the `switch` requires significant analysis effort and is independent of all other cases. Thus each case is analyzed in parallel, achieving speedups between 2–3× on five processors. The usefulness of this method is highly specific both to C and to idioms common in the C programs that Astrée targets. Monniaux claims that a version for general-purpose programs was attempted which parallelized at arbitrary nondeterministic points, and the results were dissappointing [28].

Partition-Based Parallelism. This parallelization strategy partitions the analysis in some way and computes the analysis of each partition in parallel. This strategy is extremely general, with a number of distinct instantiations in the literature.

Lee et al. [24, 25] partition their parallel Fortran analysis by strongly-connected components (SCC) in the program's CFG. Each SCC is analyzed in parallel using separate worklists; the SCC solutions are combined using elimination-based techniques [30]. They achieve an average speedup of 4.6× in 8 threads. However, the speedups were relative to their parallel analysis running on a single thread rather than to a specialized sequential version of the analysis.

Weeks et al. [32] partition a parallel analysis for a custom purely-functional language (used to write concurrent applications) using dynamically-discovered dependencies. If statement s_1 is found to be dependent on statement s_2, then s_1 will be put into s_2's partition (unless this would increase s_2's partition size beyond some threshold). The authors report runtimes, but we were able to compute speedup from the provided data. These average 9.4× on 16 threads on two trivial benchmarks handcrafted by the authors.

Albarghouthi et al's parallel C analysis. [10] is query-based (i.e., they do not compute a solution for the entire program, only enough to answer a specific query). They frame the analysis in terms of MapReduce [12], with a parallel map phase and a sequential reduce phase. During the map phase, multiple functions are analyzed intraprocedurally in parallel. If a function call is encountered, then the call is enqueued to be analyzed later. During the reduce phase, sequential dependencies are accounted for. The process is repeated on the enqueued function calls until a fixpoint is reached. They achieve an average speedup of 3.71× on 8 hardware threads.

2.3 Problems with the DFA Approach for Parallelism

A number of the existing approaches to parallelizing analysis, as described above, require a CFG as input. For languages such as C and Fortran this is a reasonable assumption; however, for a language like JavaScript it is not reasonable at all. Javascript's higher-order functions, prototype-based inheritance, implicit type conversions and implicit exceptions, and other language features mean that a computing a useful CFG requires extensive, precise, and costly analysis—the very kind of analysis we are trying to optimize via parallelization.

In addition, the DFA approach itself can make it more difficult to see opportunities for parallelization. The traditional formulation of DFA is inherently sequential. We observe that equations (1) and (2) in Section 2.1 implicitly impose synchronization points into the very definition of the analysis itself, as they require multiple nodes to cooperate in order to merge and propagate information between themselves. Syn-

[4] Speedups speak of both speed and scalability whereas runtimes tell us *only* about how fast something went.

chronization is (almost) unavoidable for a tractable analysis, but the DFA framework makes it difficult to separate synchronization out as a separate concern from the analysis itself.

3. Designing for Parallelism

Our key insight is that by designing and implementing the program analysis in a certain way, the design space of parallelization strategies becomes clearer and implementing the parallelization strategies effectively becomes easier. In particular, we take advantage of an approach to program anaysis based on abstract interpretation which we call $\text{STS}_\triangledown$; this approach divides the analysis into two separate components: an embarrassingly parallel reachability computation on a state transition system, and a strategy for selectively merging states during that reachability computation. We describe this program analysis approach below, and then discuss the parallelism design space exposed by this anaysis perspective.

3.1 The $\text{STS}_\triangledown$ Approach to Program Analysis

The basis for the $\text{STS}_\triangledown$ approach to program analysis is described in [16]; we summarize the approach in this section. Note that everything in this section refers to a completely sequential definition of program analysis; there is no parallelism. Fundamentally, the $\text{STS}_\triangledown$ model specifies a static analysis in two parts: (1) the underlying analysis itself, described as a state transition system (STS); and (2) a strategy for when to merge states together, used to bound the reachable state space while maintaining the soundness of the analysis.[5] The solution to the analysis is the set of reachable states in the STS from some given initial state; the state merging strategy specifies the *control flow sensitivity* of the analysis, i.e., its path-, flow-, context-, and heap-sensitivity. Thus, the analysis and its sensitivity are treated as two separate and independent concerns. The key insight of this paper, as opposed to [16], is that this separation of concerns can greatly benefit parallelism in a way described in later sections of this paper.

An abstract machine-based smallstep operational semantics is a useful way to describe a static analysis [31], and can easily be seen as a STS. Such a semantics defines a notion of *abstract state* (e.g., a program point together with the current abstract values of all variables in scope at that program point) and a set of *transition rules* that uses the semantics of the program statement at that program point to map an abstract state to a new abstract state. For example, if the abstract state is $\langle \texttt{pp3}, [x \mapsto 1] \rangle$ and the statement at program point 3 is "`x += 1`", then the next state would be $\langle \texttt{pp4}, [x \mapsto 2] \rangle$. The exact definition of an abstract state and the transition rules would vary depending on the language being analyzed and the analysis being defined. Without going into details on the exact state definition and transition rules for a particular language and analysis, we can formalize this idea as the following:

$$\hat{\varsigma} \in \Sigma^\sharp \qquad \text{abstract states}$$
$$\mathcal{F}^\sharp \subseteq \Sigma^\sharp \times \Sigma^\sharp \qquad \text{transition relation}$$

The abstract states form a lattice $\mathcal{L} = (\Sigma^\sharp, \sqsubseteq, \sqcap, \sqcup)$, where $\sqsubseteq$ is the ordering relation, $\sqcap$ is the meet operator, and $\sqcup$ is the join operator. The solution to the program analysis is defined as the least fixpoint (**lfp**) of the abstract semantics from some set of initial states $\Sigma_I^\sharp$. Define the operator $\mathring{:}$ so that for set S and any function on sets $\mathcal{F}$, $\mathring{\mathcal{F}}(S) = S \cup \mathcal{F}(S)$. Then the analysis solution $[\![P]\!]^\sharp$ for program P is defined by:

$$[\![P]\!]^\sharp = \textbf{lfp}_{\Sigma_I^\sharp} \mathring{\mathcal{F}}^\sharp$$

An operational view of this least fixpoint definition is as a worklist algorithm: it initializes the worklist with the states in $\Sigma_I^\sharp$, then iteratively it (1) removes the current states from the worklist; (2) applies $\mathcal{F}^\sharp$ to them to get a set of new states; (3) filters out any states it has seen already; and (4) puts the remaining states into the worklist. This continues until the worklist is empty, at which point it has computed the entire set of possible states, thus concluding the program analysis.

However, the analysis as defined is intractable (in fact, potentially uncomputable). The issue is control-flow, specifically, the nondeterministic choices that must be made because of the analysis' over-approximations: which branch of a conditional should be taken, whether a loop should be entered or exited, which (indirect) function should be called, etc. The number of abstract states grows exponentially with the number of nondeterministic choices, and is potentially unbounded. We must extend the analysis to control this behavior.

Therefore, we apply a *widening operator* $\triangledown$ to the analysis which bounds the abstract state space by selectively merging abstract states, thus losing precision but making the analysis tractable. This widening operator will, at each step of the fixpoint computation: (1) partition the current set of reachable states into disjoint sets; (2) for each partition, merge all of the abstract states in that partition into a single abstract state that over-approximates the entire partition; (3) union the resulting abstract states together into a new set that contains only a single abstract state per partition. This allows us to limit the number of states by fixing a particular number of partitions. By defining different strategies for partitioning the abstract states, we can control how states are merged and thus control the precision and performance of the analysis. As shown in [16], this partitioning strategy is actually synonymous with the analysis control flow sensitivity.

Operationally, this means that we modify the worklist algorithm so that it maintains a memoization table with one entry (i.e., abstract state) per partition. At each step the algorithm selects a state from the worklist, uses $\mathcal{F}^\sharp$ to compute a

[5] As described in [16] this strategy is a widening operator $\triangledown$ in the abstract interpretation sense.

new set of states, merges them into the appropriate partition entries using $\triangledown$, and if any of those partition entries have changed due to the newly-merged information, adds them back into the worklist. In pseudocode, this operational view of the $\text{STS}_{\triangledown}$ model looks like the following:

Algorithm 1 The sequential worklist algorithm

```
put the initial abstract state ŝ0 on the worklist
initialize map memo : Partition → Σ♯ to empty
repeat
    remove an abstract state ŝ from the worklist
    for all abstract states ŝ′ in next_states(ŝ) do
        if memo does not contain partition(ŝ′) then
            memo(partition(ŝ′)) = ŝ′
            put ŝ′ on worklist
        else
            ŝold = memo(partition(ŝ′))
            ŝnew = ŝold ⊔ ŝ′
            if ŝnew ≠ ŝold then
                memo(partition(ŝ′)) = ŝnew
                put ŝnew on worklist
            end if
        end if
    end for
until worklist is empty
```

The `next_states` function applies the state transition rules to determine the next abstract state(s) reachable from the given abstract state—this entails the computational core of the analysis logic. The `partition` function maps an abstract state to its partition as defined by the state merging strategy. The algorithm computes the analysis fixpoint exactly as described earlier.

3.2 Parallelism Design Space

The $\text{STS}_{\triangledown}$ program analysis model provides a useful perspective for parallelizing analysis, because it boils the problem down to two questions: (1) what strategy should we use to merge states during the reachability computation (thus injecting synchronization points); and (2) what granularity should we use to parallelize the reachability computation itself?

Recall that the state merging strategy is synonymous with the flow- and context-sensitivity of the analysis—merging fewer states means greater sensitivity and thus greater precision, while merging more states means less sensitivity and thus less precision. With respect to parallelization, there is a tradeoff between merging strategies that merge fewer states (reducing synchronization but increasing the number reachable states), versus strategies that merge fewer states (increase synchronization but reducing the number of reachable states). We explore a small part of this space in our evaluation, however, there is interesting future work in exploring this trade-off further.

Besides state merging, the remaining question is granularity, which we explore in the rest of this section. We first discuss an obvious point in this space, the worklist-parallel strategy, and why it is not a satisfactory solution. We then introduce a novel point in this space, the per-context strategy, that has not been explored before.

Worklist-Parallel Strategy. The most straightforward granularity strategy is to parallelize the worklist loop by processing each node on the worklist in parallel. In essence, we explore the reachability of each node independently until the various states reach some merge point specified by the merge strategy (but not necessarily the same merge point for all states), whereupon the merged states are inserted back into the global worklist for the process to be repeated. The pseudocode of the analysis for this strategy looks like the following:

Algorithm 2 The worklist parallel algorithm

```
put the initial abstract state ŝ0 on the worklist
initialize templist to empty
initialize map memo : Partition → Σ♯ to empty
repeat
    for all abstract states ŝ in the worklist do in parallel
        for all abstract states ŝ′ in next_states(ŝ) do
            begin thread-safe
                if memo does not contain partition(ŝ′) then
                    memo(partition(ŝ′)) = ŝ′
                    put ŝ′ on templist
                else
                    ŝold = memo(partition(ŝ′))
                    ŝnew = ŝold ⊔ ŝ′
                    if ŝnew ≠ ŝold then
                        memo(partition(ŝ′)) = ŝnew
                        put ŝnew on templist
                    end if
                end if
            end thread-safe
        end for
    end parallel for
    swap worklist and templist
until worklist is empty
```

The **thread-safe** block is run atomically using synchronization primitives.

There are three major drawbacks to this strategy. First, it can cause a great deal of redundant computation because of node ordering issues (as described in Section 2.1). If multiple states are being processed in parallel but one subsumes the others, then the parallel computations are not actually useful and there is no gain in performance. Second, all of the parallel computations must be synchronized together, even those that reach different merge points (and hence are independent). This is because the analysis doesn't know which threads will reach which merge points, and thus must wait until all threads reach some merge point before it can continue at any one merge point. Finally, this strategy introduces a large number of short-lived threads, which can be detrimental to performance.

Per-Context Parallel Strategy. We propose a novel point in the granularity design space based on function contexts, one that attempts to address some of the issues of the worklist-parallel strategy and is motivated by empirical observation. We want to reduce node ordering issues, limit

synchronization between independent parts of the analysis, and increase the granularity of the thread computations. Context-sensitive analyses have desirable properties which can be exploited for these goals. Context-sensitive analyses *clone* functions based on some notion of abstract calling context (the exact definition of "context" defines the particular type of context-sensitivity used by the analysis). Each clone is specialized to a particular context and, most importantly, analyzed separately. Different clones can be analyzed in parallel, while analysis of a single clone can be done sequentially. This strategy allows a more optimal node ordering, because within each context we can sequentially analyze nodes in reverse postorder (the best possible node ordering). Different contexts are independent of each other, which limits synchronization. Finally, threads now compute an entire function rather than a single statement or basic block, increasing work granularity per thread and reducing thread management overhead. The pseudocode of the analysis using this strategy is as follows:

Algorithm 3 The per-context parallel algorithm

```
procedure analysisThread(ctxt)
    move abstract states from backlog(ctxt) to worklist
    repeat
        remove an abstract state ŝ from the worklist
        for all abstract states ŝ′ in next_states(ŝ) do
            if context(ŝ′) ≠ ctxt then
                process(context(ŝ′), ŝ′)
            else if memo does not contain partition(ŝ′) then
                memo(partition(ŝ′)) = ŝ′
                put ŝ′ on worklist
            else
                ŝ_old = memo(partition(ŝ′))
                ŝ_new = ŝ_old ⊔ ŝ′
                if ŝ_new ≠ ŝ_old then
                    memo(partition(ŝ′)) = ŝ_new
                    put ŝ_new on worklist
                end if
            end if
        end for
    until worklist is empty
    if backlog(ctxt) is empty then
        mark this thread as potentially done
    else
        analysisThread(ctxt)
    end if
end procedure

procedure process(ctxt, ŝ)
    begin thread-safe
        enqueue ŝ into backlog(ctxt)
        if no thread corresponding to ctxt is running then
            analysisThread(ctxt)
        end if
    end thread-safe
end procedure

procedure main
    initialize map memo : Partition → Σ♯ to empty
    analysisThread(context(ŝ0))
end procedure
```

In the above algorithm, a unique thread is used to run analysisThread per context. The global map `backlog` maps each context to a synchronized queue, while `worklist` is local to each thread. The function `context` extracts the context under which an abstract state needs to be analyzed. Note that no synchronization is required on access to `memo` (because each thread is run sequentially and multiple threads do not access same parts of `memo`). The procedure process checks if no thread corresponding to context is running, which can happen under two circumstances: (1) the context has never been seen before, thus a new thread is used to run analysisThread with that context (2) the thread corresponding to the context has marked itself as potentially done, in which case the thread is unmarked and woken up back again to run analysisThread. The analysis begins by calling main, and the analysis ends when each of the threads mark themselves as potentially done and each of the backlog queues are empty.

While a per-context strategy has been previously mentioned in the literature [15], to our knowledge this is the first time it has ever been detailed and implemented. Additionally, thanks to the $\text{STS}_\triangledown$ representation of the analysis, using the per-context strategy is simple. Instead of a global worklist, use one worklist per context encountered during the analysis. Each worklist has a dedicated thread computing a fixpoint. When a thread processes a function call leading to a new context, it passes the resulting state on to the appropriate thread and continues processing its own worklist. The only synchronization required is this thread communication. When all worklists are empty, the analysis has reached a global fixpoint.

4. Parallel JavaScript Analysis

In this section we briefly describe the JavaScript language and the existing sequential JavaScript analysis that we adapted for our parallel analysis. We then describe the modification to that sequential analysis necessary to implement our parallel analysis design.

4.1 JavaScript Features

JavaScript is an imperative, dynamically-typed language with objects, prototype-based inheritance, higher-order functions, implicitly applied type-conversions, and exceptions. JavaScript programs only have two scopes (global scope and function scope), though variables and functions are allowed to be defined anywhere; these declarations (but not the corresponding initializations, except for functions) are automatically hoisted to the appropriate scoping level. JavaScript is designed to be as resilient as possible: when a program performs some action that doesn't make sense (e.g., accessing a property of a non-object, or adding a boolean and a function together) JavaScript uses implicit conversions and default behaviors when possible in order to continue the execution without errors rather than raising an exception.

Objects are the fundamental JavaScript data structure.[6] Object properties can be dynamically inserted and deleted, and when performing a property access the specific property being accessed can be computed at runtime. JavaScript features such as the `for..in` loop and the `in` operator allow for reflective introspection of an object's contents. Object inheritance is handled via delegation: when accessing a property that is not present in a given object *obj*, the property lookup algorithm determines whether *obj* has some other object *proto* as its prototype; if so then the lookup is recursively propagated to *proto*.

These features have two important implications for static analysis: (1) computing a precise CFG requires careful and costly analysis, because higher-order functions, prototype-based inheritance, implicit type-conversions, and implicit exceptions make control-flow non-obvious, thus analysis techniques based on the CFG are problematic; and (2) JavaScript's inherent dynamism means that high precision is important to get useful results, implying that any useful analysis will be expensive.

4.2 Sequential JSAI

We build on an existing sequential abstract interpreter for JavaScript called JSAI [21]. The analysis performs type inference, control-flow analysis, pointer analysis, and numeric, string, and boolean value analysis. JSAI is designed and implemented as an abstract machine-based smallstep operational semantics, which can be thought of as a state transition system. Rather than baking in a specific flow-, context-, and heap-sensitivity strategy, JSAI is designed around the STS_∇ model in order to have configurable control flow sensitivity [16]: the basic analysis computes the reachable states of the STS defined by the abstract semantics, while a separate modular component determines a strategy for selectively merging states. States are represented as tuples holding relevant components such as the values on the stack and heap, the current continuation, and a trace recording the execution history. The set of states forms a lattice; states are merged using the lattice join operator which operates pointwise on the state components. The choice of which states to merge and when is determined by JSAI's merging strategy, which can be chosen independently from the rest of the analysis. A given merging strategy determines the flow-, context-, and heap-sensitivity of the analysis; indeed, merging strategies and sensitivities are synonymous.

JSAI is formally specified and the code is designed to have a close correspondence with the formalisms, using immutable states and written using mostly pure functional style, making it easy to follow and manipulate. Alternatively, we could have used TAJS [17–19], a competing sequential JavaScript analysis framework, whose runtimes are in the same order of magnitude as JSAI (between 0.3× and 1.8×). However, TAJS does not use the STS_∇ model, does not offer configurable sensitivity, and lacks a formal specification.

[6] Even functions and arrays are just special kinds of objects, and can be used in the same ways as other objects.

4.3 Parallelism Strategies

We implement two specific parallelism strategies as discussed in Section 3: the worklist-parallel strategy and the per-context strategy. We describe for each one the necessary changes to JSAI, which were minimal in both cases. Our experience is that implementing different strategies is a simple task, making it easy to explore the design space of the STS_∇ model. For each strategy we use a single global thread pool [1] of a fixed size and create new thread tasks for the pool on demand. We also replace JSAI's memoization table (which holds the computed solution as the analysis executes, mapping program points to the abstract states computed at that program point so far) with a thread-safe version that requires no locking for table lookups [2].

Worklist-Parallel Strategy. This strategy is the simplest to implement. Rather than iteratively popping elements off of the worklist and processing them sequentially, instead we pop *all* elements of the worklist and process each element in parallel, having them enqueue the resulting abstract states back onto the global worklist. This strategy is, in concept, the same strategy used by Dwyer et al [14], and we implement it to use as a comparison point for our novel proposed strategy given below.

Per-Context Parallel Strategy. We observe that for the JavaScript benchmarks we have tested, if N states are on a non-empty worklist, and those N states are members of M contexts (where $1 \leq M \leq N$), then typically $M >> 1$. In other words, many contexts are typically enqueued for processing at once. This indicates that the per-context strategy described in Section 3 has promise for anayzing JavaScript. Instead of a global worklist, we use one worklist per context, with one thread for each worklist. Each worklist has an associated non-blocking, lock-free [3] *backlog* queue that other threads use to enqueue work for that thread; whenever a thread runs out of elements in its worklist, it puts its backlog queue into its worklist and continues. When a thread processes a function call that belongs to a new context, it puts the resulting abstract state into that context's backlog queue. The analysis has reached a fixpoint when all worklists and backlog queues are empty. The memoization table is global; because contexts are independent from each other, there will never be a conflict between threads when updating the memoization table. We also tried an alternative to the backlog queue strategy for thread communication, wherein threads directly enqueued work into other threads' worklists; we saw results ranging from 30% faster to 18% slower performance relative to the backlog queue implementation, with most benchmarks being slower; thus we only use the backlog queue implementation in our evaluation.

5. Evaluation

We evaluate our parallel JSAI implementation using a set of real-world JavaScript benchmarks, detailed in Section 5.2. We decribe the benchmarks and our experimental methodology, then present and discuss our results for the worklist-parallel strategy and the per-context strategy.

5.1 Experimental Methodology

System Under Test. Our testbed is equipped with two 6-core Intel Xeon processors running at 1.9 Ghz with hyper-threading enabled. We only report data for 1-12 threads, with one thread per core. While utilizing hyperthreading with 13-24 threads usually did yield better speedups, these tended to be minimal and with high variability. The machine is equipped with a total of 32 GB of memory, and we ran with a maximum JVM heap size of 25,600 MB for all experiements. During the course of the experiments, we had exclusive access to the machine, and all non-essential services were disabled.

Calculating Speedups. The speedups we report are relative to the sequential JSAI implementation, as per the usual definition of speedup. This is an important point for comparing against related work. In several cases, authors have instead focused on runtimes [32], speedups relative to the framework itself [24, 25, 27], percent improvement in performance [23], or atypical presentations of speedups [28].

Configuration Focus. Previous experiments [21] have shown that stack-based CFA tends to work well for JavaScript, both in terms of precision and performance. We specifically use `stack-k-h` CFA, in which the top k callsites on the call stack are used as the context (this is the standard "callstring context sensitivity" used in DFA). The parameter h controls the heap sensitivity, which distinguishes each abstract object allocation by a context of depth h, in addition to its program location. We show results for `stack-5-4` (most precise), `stack-3-2`, and `stack-1-0` (least precise) for comparison.

Testing. In order to test the correctness of our implementation, we annotated the benchmarks with special statements to print out the final abstract values for certain program points. In all cases, the solutions from our parallel implementation were equivalent to the solutions produced by the sequential interpreter. In addition, we ran several hundred handcrafted tests on both the sequential and parallel analyzers to compare their results; in all cases they agree.

5.2 Benchmarks

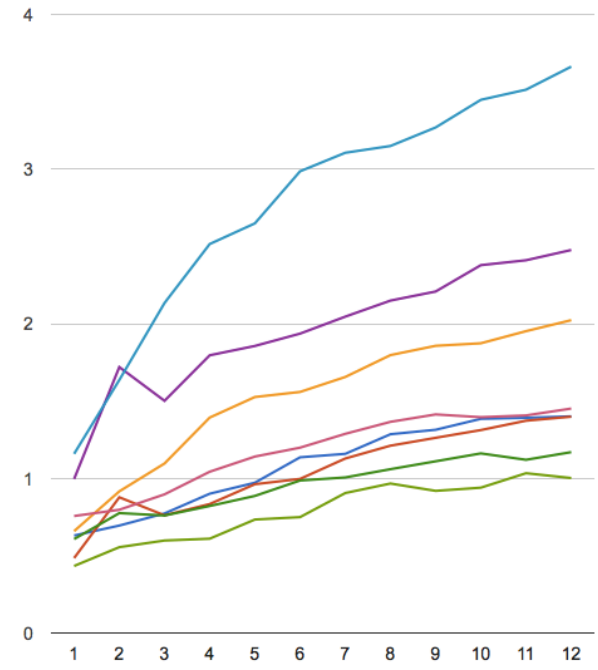
We focus on ECMA3-compliant JavaScript programs which do not exercise the document object model (DOM). While SunSpider and other concrete performance benchmark suites meet the above criteria, most of their constituents complete analysis within seconds. Given that short-running benchmarks can be improved little by the addition of parallelism, such benchmarks have been omitted from our evaluation.

In an attempt to derive more complex benchmarks which are more time-consuming for our analysis to handle, we have turned to open source JavaScript programs in the wild. This allows us to benchmark against a more realistic suite. Additionally, we have intentionally selected benchmarks representing a variety of coding styles, with both imperative and functional code. This allows us to determine whether or not our parallel analysis performance is dependent on coding style, which is important considering that JavaScript allows for very different styles to be used and to coexist. A complete description of our benchmark suite is given in Table 1.

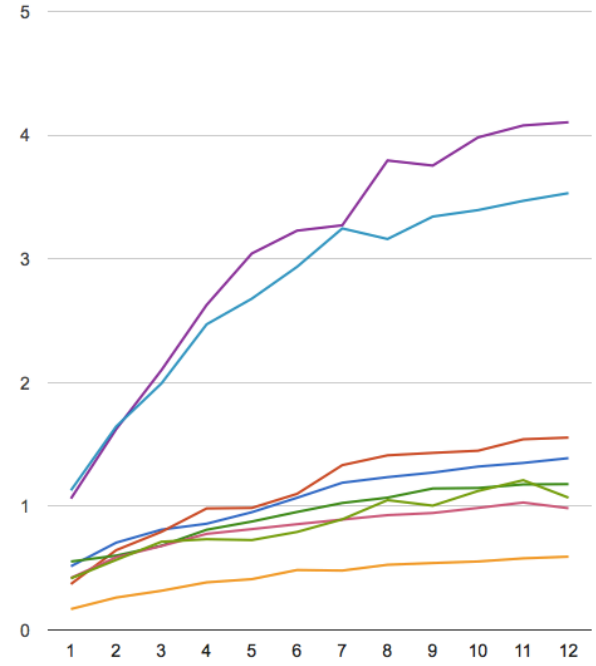
5.3 Worklist-Parallel Results

The performance results for the worklist-parallel strategy on the configurations `stack-1-0`, `stack-3-2` and `stack-5-4` in Figures 1(a), 1(b), and 1(c). Under `stack-1-0`, most benchmarks barely even reach a 2× speedup, though `linq_aggregate` and `linq_functional` manage to cross this bound by a small margin. With `stack-3-2`, the distance between the combination of `linq_aggregate` and `linq_functional` and the rest of the benchmarks begins to increase, as both comfortably break a 3× speedup. Another point of interest is that `linq_aggregate` and `linq_functional` are fairly comparable in terms of performance, whereas under `stack-1-0` `linq_functional` outperformed `linq_aggregate` by a significant margin.

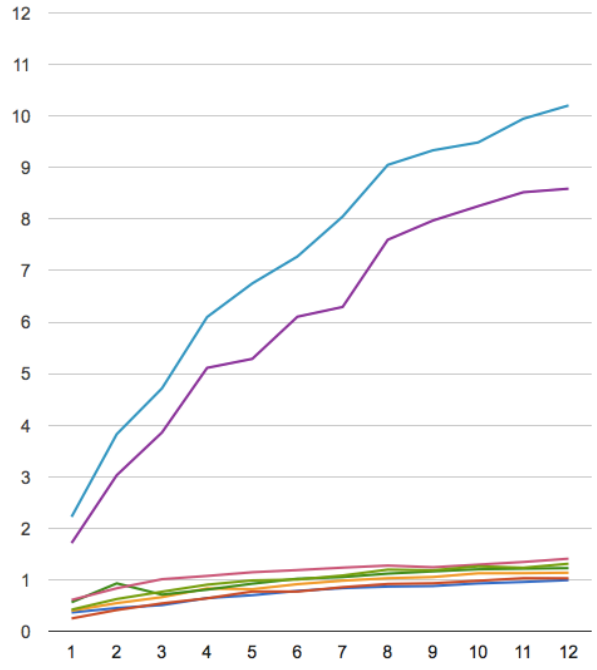
It is under `stack-5-4` that the worklist-parallel results show real promise with the `linq_aggregate` and `linq_functional` benchmarks. `linq_aggregate` outperforms `linq_functional`, in contrast with the behavior underneath `stack-1-0`. Moreover, both these benchmarks show superlinear speedup for less than 10 hardware threads. The superlinear speedup seems to be the result of two factors. First, both the `linq_aggregate` and `linq_functional` benchmarks are highly functional in implementation style, containing more than one function per ten lines of code. Moreover, many of these functions are used in a higher-order style, are largely independent of each other, and are of small to moderate length. Intuitively, this leads to many contexts of a granularity level well-suited to parallel processing. Second, node ordering is also probably a factor. The sequential analyzer enforces an arbitrary, possibly far-from-optimal ordering between states in different contexts (it uses reverse post-order *within* a context, but without the results of the analysis it isn't possible to order *between* contexts). It is possible that our worklist-parallel implementation just happens to choose better node orderings on these benchmarks. The somewhat erratic nature of our speedup curves serves as further evidence of these ordering issues. It is unclear how to measure the effects of ordering in a structured way, therefore, we do not have any experiments that can backup our conjecture. There are many examples from the literature where either superlinear or otherwise better than predicted performance has been recorded [10, 14, 15, 24], and different node orderings were commonly cited as the reason.



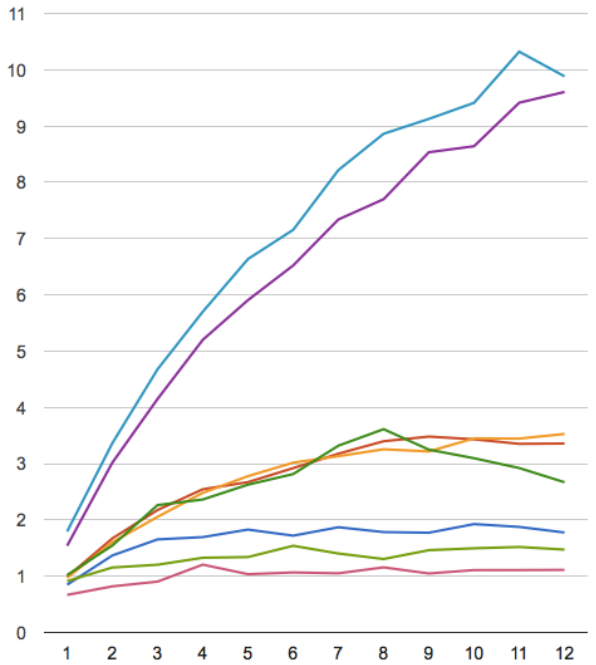
(a) Worklist-parallel speedups for `stack-1-0`



(b) Worklist-parallel speedups for `stack-3-2`

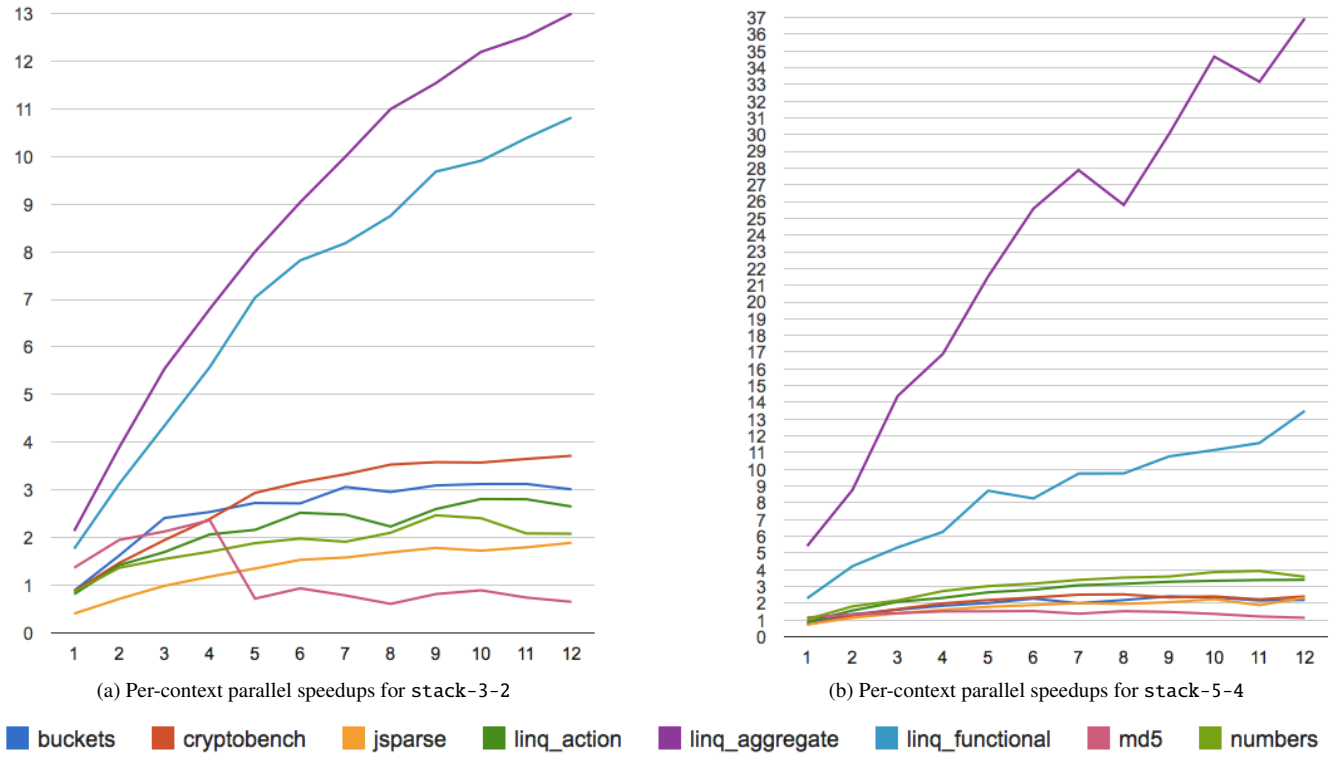


(c) Worklist-parallel speedups for `stack-5-4`



(d) Per-context parallel speedups for `stack-1-0`

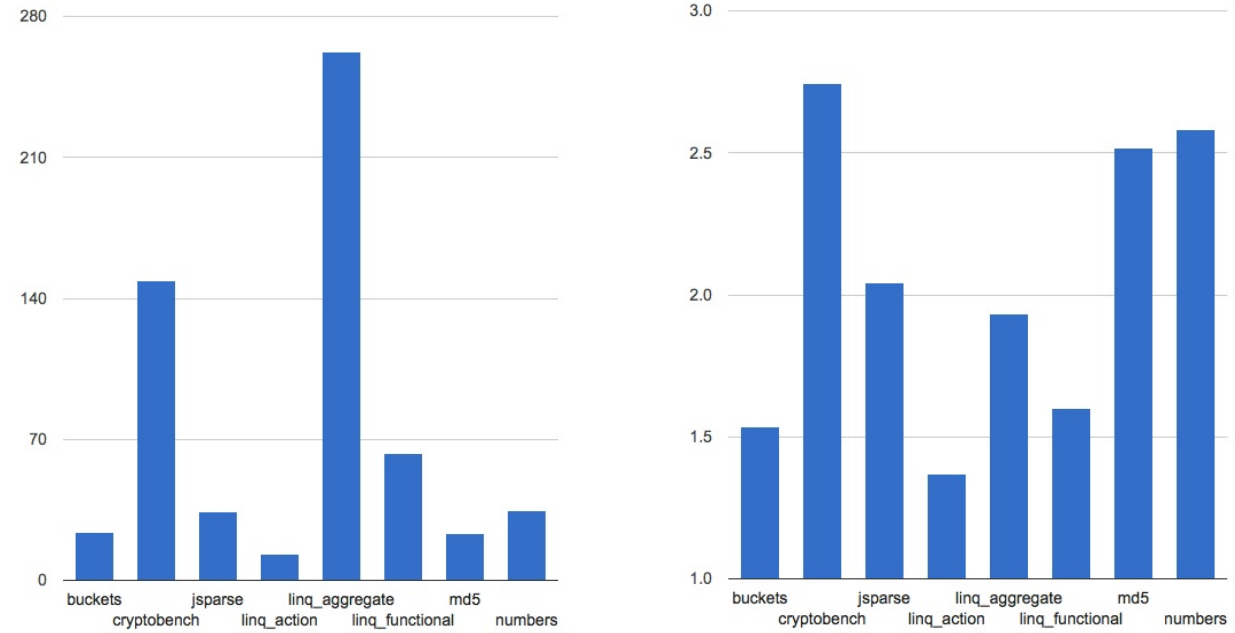
Figure 1: Speedups for our per-context parallel and worklist-parallel implementations on various traces. The number of hardware threads used is on the `x` axis, and the speedup is on the `y` axis.



(a) Per-context parallel speedups for `stack-3-2`

(b) Per-context parallel speedups for `stack-5-4`

Figure 2: Speedups for our per-context parallel implementations on configurations `stack-3-2` and `stack-5-4`. The number of hardware threads used is on the `x` axis, and the speedup is on the `y` axis.



(a) Average no. of available contexts for `stack-5-4` on the worklist-parallel implementation

(b) Average ratio of the no. of available states to the no. of available contexts for `stack-5-4` on the worklist-parallel implementation

Figure 3: Graphs to test the hypotheses presented in Section 5.5.

Benchmark	Derived From	General Kind	Number of Functions	LOC	Sequential Runtime Under `stack-5-4` (s)
`cryptobench`	[4] (SunSpider origin)	imperative	132	1699	508.082
`md5`	[5]	imperative	37	365	778.061
`buckets`	[6]	mixed	168	2472	73.801
`numbers`	[7]	mixed	90	1870	145.082
`jsparse`	[8]	functional	74	878	515.239
`linq_action`	[9]	functional	381	2783	32.097
`linq_aggregate`	[9]	functional	396	2830	80722.088
`linq_functional`	[9]	functional	378	2790	4516.588

Table 1: A summary of our benchmark suite. The `linq*` benchmarks all execute different APIs from the same common library in a manner that causes vastly different code paths to be analyzed between the three benchmarks. Benchmarks of the *mixed* kind have both imperative and functional characteristics based on subjective observation.

In stark contrast to the excellent speedups of `linq_aggregate` and `linq_functional` on `stack-5-4`, the rest of the benchmarks see rather dismal performance. No other benchmark was able to reach a speedup higher than 1.45×, irrespective of the number of hardware threads used. This implies that for these benchmarks, the worklist-parallel approach duplicates a significant amount of work. One exception to this seems to be `linq_action`, which was derived from the same codebase as `linq_aggregate` and `linq_functional`. Given that `linq_action` has a fairly short sequential runtime at around 32 seconds, it seems that it is simply too short to see much improvement from the worklist-parallel strategy.

5.4 Per-Context Parallel Results

Speedups for the per-context parallel implementation with our `stack-1-0` trace on our benchmark suite are shown in Figure 1(d). Once again, `linq_aggregate` and `linq_functional` stand out, unconditionally showing higher speedups in all cases than any other benchmark. Moreover, both benchmarks show superlinear behavior for less than 10 hardware threads, presumably for the same reasons as detailed in the previous section. Of particular interest is that based on the performance results, it appears that there are three buckets in which data can be distributed based on their relative performance to each other. This bucketing shows that functional programs tend to perform better than non-functional programs, with all of our functional benchmarks being either in the top-performing or moderate-performing bucket. However, this is not to say that non-functional programs can not perform well; the moderately-performing bucket holds `cryptobench`, a highly imperative benchmark.

With the trend of functional programs generally performing better, speedups underneath `stack-3-2` in Figure 2(a) and `stack-5-4` in Figure 2(b) are particularly interesting. These speedups do not illustrate the same sort of buckets as seen with `stack-1-0` in Figure 1(d). However, the `linq_aggregate` and `linq_functional` benchmarks both show superlinear speedup, with `linq_aggregate` seeing speedups higher than 34× on 10 cores. `linq_aggregate` shows strong evidence that node ordering is to blame, given the sudden drop in performance at 8 hardware threads under `stack-5-4`. It seems likely that the sequential interpreter chooses a particularly poor context ordering for `linq_aggregate` underneath the `stack-5-4` trace.

This same sort of performance drop is also seen with the `numbers` benchmark underneath `stack-5-4` and `md5` underneath `stack-3-2`. The `md5` benchmark underneath `stack-3-2` is especially interesting, considering the severity of the drop at 5 hardware threads and the fact that its performance never improves after this drop. It seems likely that in this case, certain well-performing context orderings are commonly available for thread schedules involving 5 or less threads. However, after this point, poor context orderings become far more likley. As such, context ordering plays a significant role, even for the parallel abstract interpreter.

5.5 Discussion of Per-Context Granularity

As shown by our speedup results in Figures 1(d), 2(a) and 2(b) our per-context granularity tends to yield decent speedup results. However, we have found evidence that even better levels of granularity exist for JavaScript, at least on a theoretical level. One would expect that if context-level granularity were the best level of granularity, then the following two propositions would be true:

1. As the number of available contexts increase, so do the speedups in the per-context parallel implementation, because there would be more opportunity for parallelism.
2. As the number of distinct contexts decrease and the overall number of states increase, the worklist-parallel approach becomes less successful. In other words, if the majority of states share the same few contexts, then the worklist-parallel speedups decrease. This follows from the assumption that states within the same context tend to be dependent on each other, and so processing many states in the same context in parallel would yield lots of wasted work.

We gathered data to test these propositions, and found that these tend not to be true. As shown in Figure 3(a), number of available contexts is only roughly correlated to the speedup of the per-context parallel implementation.

Figure 3(b) shows for each parallel program step in the worklist-parallel implementation, the ratio of total number of available states over the number of states with distinct contexts. It is expected that as this number becomes higher, more and more states are dependent on each other, and so the amount of lost work increases in the worklist-parallel implementation. However, there appears to be no correlation between how high or low a ratio is and how well or poor the worklist-parallel implementation performed.

Given that these propositions appear to be untrue, there likely exists an even better level of granularity for our JavaScript benchmarks, or at least a level that piggybacks off of our per-context granularity. An advantage of the $\text{STS}_\triangledown$ approach is that it exploring new parallelisation strategies to find these hypothetical improvements is relatively easy, because of way the analysis is structured.

6. Conclusions

We have presented a alternative program analysis model to the more usual DFA approach, called $\text{STS}_\triangledown$. This framework makes it easy to reason about and explore different parallelization strategies, as well as being more applicable than DFA to languages with difficult control-flow that make the program CFG hard to compute. Using this framework we have implemented a parallel analysis for JavaScript and explored two points in the parallel design space: a naive worklist-parallel strategy and a novel per-context strategy. Our results show that our parallel implementation provides speedups comparable or better than the speedups reported in our related work for realistic JavaScript programs.

Acknowledgments

NSF CCF-1319060 and CCF-1117165 supported this work.

References

[1] `http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html`.

[2] `http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/ConcurrentHashMap.html`.

[3] `http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html`.

[4] `http://octane-benchmark.googlecode.com/svn/latest/crypto.js`.

[5] `https://github.com/blueimp/JavaScript-MD5`.

[6] `https://github.com/mauriciosantos/buckets`.

[7] `http://github.com/sjkaliski/numbers.js`.

[8] `https://github.com/doublec/jsparse`.

[9] `http://linqjs.codeplex.com/`.

[10] A. Albarghouthi, R. Kumar, A. V. Nori, and S. K. Rajamani. Parallelizing top-down interprocedural analyses. In *ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*, 2012.

[11] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The astrée analyzer. In *European Symposium on Programming (ESOP)*, 2005.

[12] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[13] M. B. Dwyer and L. A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 1994.

[14] M. B. Dwyer and M. Martin. Practical parallelization: Experience with a complex flow analysis. Technical Report KSU CIS TR 99-4, Kansas State University, 1999.

[15] M. Edvinsson, J. Lundberg, and W. Löwe. Parallel points-to analysis for multi-core machines. In *International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC)*, 2011.

[16] B. Hardekopf, B. Wiedermann, B. Churchill, and V. Kashyap. Widening for control-flow. In *International Conference on Verification, Model-Checking, and Abstract Interpretation (VMCAI)*, 2014.

[17] S. H. Jensen, P. A. Jonsson, and A. Møller. Remedying the Eval that Men Do. In *International Symposium on Software Testing and Analysis*, 2012.

[18] S. H. Jensen, A. Møller, and P. Thiemann. Type Analysis for Javascript. In *International Symposium on Static Analysis*, 2009.

[19] S. H. Jensen, A. Møller, and P. Thiemann. Interprocedural Analysis with Lazy Propagation. In *International Symposium on Static Analysis*, 2010.

[20] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:309–317, 1977.

[21] V. Kashyap, K. Dewey, E. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf. JSAI: A static analysis platform for javascript. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2014.

[22] G. A. Kildall. A unified approach to global program optimization. In *Symposium on Principles of Programming Languages (POPL)*, 1973.

[23] W. Le and M. L. Soffa. Parallel path-based static analysis. Technical Report CS-2010-6, University of Virginia, 2010.

[24] Y.-F. Lee and B. G. Ryder. A comprehensive approach to parallel data flow analysis. In *ACM SIGARCH International Conference on Supercomputing (ICS)*, 1992.

[25] Y.-F. Lee, B. G. Ryder, and T. J. Marlowe. Experiences with a parallel algorithm for data flow analysis. *The Journal of Supercomputing*, 1991.

[26] M. Méndez-Lojo, M. Burtscher, and K. Pingali. A gpu implementation of inclusion-based points-to analysis. In *ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP)*, 2012.

[27] M. Méndez-Lojo, A. Mathew, and K. Pingali. Parallel inclusion-based points-to analysis. In *ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2010.

[28] D. Monniaux. The parallel implementation of the astrée static analyzer. In *Asian Symposium on Programming Languages and Systems (APLAS)*, 2005.

[29] V. Nagaraj and R. Govindarajan. Parallel flow-sensitive pointer analysis by graph-rewriting. In *International Conference on Parallel Architectures and Compilation Techniques*, 2013.

[30] B. G. Ryder and M. C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys (CSUR)*, 1986.

[31] D. Van Horn and M. Might. Abstracting abstract machines. In *ACM SIGPLAN International Conference on Functional Programming*. ACM, 2010.

[32] S. Weeks, S. Jagannathan, and J. Philbin. A concurrent abstract interpreter. *Lisp and Symbolic Computation*, 1994.