Fast Condensation of the Program Dependence Graph

Nick P. Johnson Taewook Oh Ayal Zaks† David I. August

Princeton University, Princeton, NJ
{npjohnso, twoh, august}@princeton.edu

†Intel Corporation, Haifa, Israel
ayal.zaks@intel.com

Abstract

Aggressive compiler optimizations are formulated around the Program Dependence Graph (PDG). Many techniques, including loop fission and parallelization are concerned primarily with dependence cycles in the PDG. The Directed Acyclic Graph of Strongly Connected Components (DAG_{SCC}) represents these cycles directly. The naïve method to construct the DAG_{SCC} first computes the full PDG. This approach limits adoption of aggressive optimizations because the number of analysis queries grows quadratically with program size, making DAG_{SCC} construction expensive. Consequently, compilers optimize small scopes with weaker but faster analyses.

We observe that many PDG edges do not affect the DAG_{SCC} and that ignoring them cannot affect clients of the DAG_{SCC}. Exploiting this insight, we present an algorithm to omit those analysis queries to compute the DAG_{SCC} efficiently. Across 366 hot loops from 20 SPEC2006 benchmarks, this method computes the DAG_{SCC} in half of the time using half as many queries.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers

Keywords Demand-driven analysis; Dependence analysis; Program Dependence Graph; Strongly connected components.

1. Introduction

Users expect compilers to be fast; one study [34] indicates that programmer productivity drops when compilation takes more than a few seconds. If the benefit of aggressive optimizations does not outweigh the cost of long compile times, users will avoid them. Making matters worse, analysis precision drastically affects optimization quality [3, 8, 28, 29, 36], and precise analyses tend to be more expensive than their less-precise counterparts [7, 15] and scale poorly [16, 26]. Despite the performance potential of state-of-the-art transformations, common compilers optimize small, intra-procedural scopes, instead favoring short compilation times.

To extend the benefits of precise analysis to the wider community, we must first address compiler scalability. We envision a future where common compilers feature aggressive optimizations such as automatic parallelization [4, 24, 28–30, 35, 36] by default. The critical path to this end is the precise analysis of large program scopes.

Many compiler techniques are formulated around the Program Dependence Graph (PDG) [6]. Many of those techniques (*clients* of the PDG) focus primarily on dependence cycles, identified as the Strongly Connected Components (SCCs) of the PDG (Figure 1(b)).

The Directed Acyclic Graph of the SCCs (*DAG_{SCC}*) or *condensation* of the PDG is a representation that makes dependence cycles explicit (Figure 1(c)). The DAG_{SCC} contains enough information to support a broad class of compiler techniques. For instance, automatic parallelization [4, 24, 28–30, 35, 36] asks: *can two operations execute concurrently without races, or is synchronization needed?* Determine whether their respective components are ordered in the DAG_{SCC}. Program slicing [17, 37, 39] asks: *Which earlier operations may affect an operation?* All operations in the same component and operations in components ordered before it in the DAG_{SCC}. Loop fission [2, 20] asks: *Can a loop be split?* Ensure that no operations in the second half belong to the same component as, or any component ordered before components from the first half.

The DAG_{SCC} holds less information than the PDG and should be cheaper to compute. Yet, standard practice wastefully builds the full PDG before condensing it to a DAG_{SCC}. The number of potential PDG edges grows quadratically with the scope size (in vertices). Each potential edge adds a quantum of analysis effort (a *query*) to determine whether that edge exists. The running times of these queries sum to make DAG_{SCC} construction prohibitively expensive, especially since precise analyses are costly [7, 15, 16, 26].

Compiler authors should not sacrifice analysis precision for cost since imprecision limits optimization [3, 8, 28, 29, 36]. Instead, they should use the most precise analyses and reduce compilation time by exploiting the reduced information of the DAG_{SCC}.

This paper presents a technique that computes the DAG_{SCC} more efficiently than by finding SCCs of the full PDG. Using partial dependence information, the algorithm identifies dependence edges which cannot affect the clients of the DAG_{SCC}. Next, the algorithm uses a Demand-Driven Analysis framework [13, 32, 40] to elide those analysis queries and thus expend effort only on important analysis queries rather than the whole program. This improvement is orthogonal to speeding up each query; it reduces DAG_{SCC} construction time yet maintains high analysis quality since no analysis algorithms change. With these savings, compiler authors may pursue more aggressive and costlier analyses while providing the same quality of service to compiler end-users. This work contributes:

- a fast client-agnostic DAG_{SCC} algorithm (Section 3.2);
- an extension of that algorithm for PS-DSWP [28] (Section 3.3);
- a proof of correctness (Section 4); and,
- evaluation of the performance benefit (Section 5).

Averaged across 366 hot loops from 20 SPEC2006 benchmarks, the proposed method computes the DAG_{SCC} with 49.0% fewer queries, reducing construction time by 53.1% (Section 5.2). These savings dramatically affect compiler scalability. The proposed method analyzes half of the hot loops (weighted by coverage) in 456.6s on average, saving 111.5s over the baseline (Section 5.3). These savings broaden the class of transformations that fit the user's time constraints without reducing analysis precision.

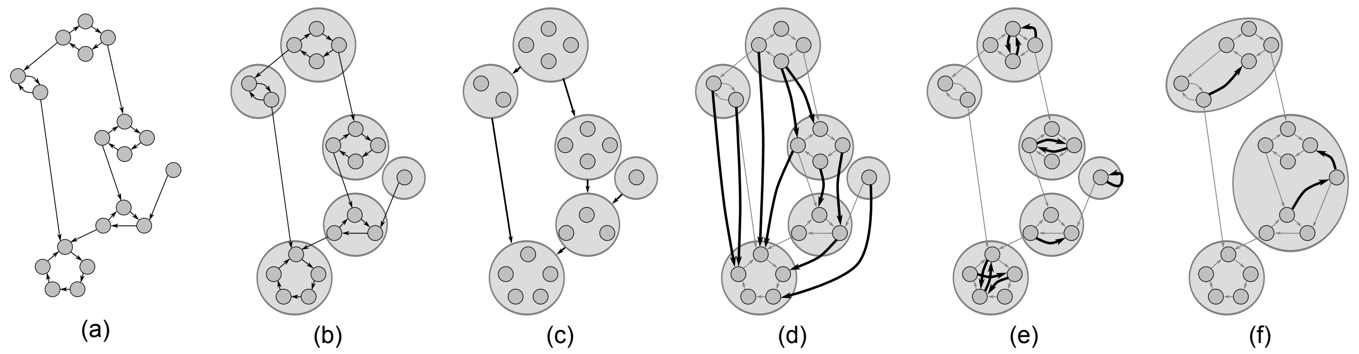


Figure 1: (a) Example PDG; (b) Strongly Connected Components; (c) Condensation of the example; (d) Additional edges which redundantly order components do not change the condensation, thus have no marginal value; (e) Additional edges within a component do not change the condensation, thus have no marginal value; (f) Additional edges which violate ordering are valuable since they may change condensation.

2. Insight

To construct the DAG_{SCC} of a loop, a naïve compiler considers the presence or absence of a dependence edge between every pair of vertices (operations), and condenses that into a DAG_{SCC}. However, not all dependences in the PDG contribute equally to the structure of the DAG_{SCC}. Once a PDG is partially computed, some edges have no marginal value since they do not affect the structure of the DAG_{SCC} and thus cannot affect the answer to optimization questions. By eliminating these redundant dependence edges, a compiler computes the DAG_{SCC} with fewer dependence queries in less time. Compiler authors may spend these savings on costlier analyses in pursuit of aggressive optimization.

An ideal algorithm would perform queries only for those edges found in a transitive reduction of the PDG (to *join* components), as well as queries to ensure the absence of back edges (to *separate* components). This, however, leads to a problem: the compiler does not know the PDG *a priori*, and so it cannot distinguish redundant edges from important ones. Instead, this paper proposes an approximation of that ideal.

Figure 1(d) demonstrates one class of redundant dependences: edges that order two vertices whose components are already ordered. This is a large class of dependences, which grows quadratically in the number of components and quadratically in component size. Across SPEC2006, empirical study indicates that two-thirds of all loops have 5–968 SCCs and two-thirds of all components have 8.4–1118.0 vertices.

Another class of redundant dependences are demonstrated by Figure 1(e): edges within a component other than a minimum cycle that spans the component. This class grows quadratically in component size and linearly in the number of components.

The only dependences which contribute to finding the condensation graph are the class which join separate components, demonstrated in Figure 1(f). These grow quadratically in the size of components and quadratically in the number of components. Although this is a large class, any one dependence between a pair of components will constrain the entire component. Conversely, the absence of these dependences also has value, since only after analysis returns *negative* results can the algorithm confidently report that the separate components are separate.

By periodically interrupting PDG construction to recompute strongly connected components, the proposed algorithm eliminates queries for dependences which are definitely in classes (d) and (e) while focusing on those dependence queries which seem to be within class (f). This approach is informed by the following heuristic: if the compiler can build large components quickly, it can safely exclude more edges. Further, this technique performs more computation to actively search for opportunities to elide queries. This strategy will not be faster in the worst case since the overhead of recomputing components may overwhelm the benefits for loops with a very low average component size. However, the common case is more amenable to this strategy; experiments show that the proposed method is faster for all but 14 of 366 loops.

3. Efficient DAG_{SCC} Construction Algorithm

In a Program Dependence Graph (PDG), each instruction in the loop is represented as a vertex. Edges are drawn to represent control and data dependences. Figure 2(a) shows an example of a PDG.

Control dependences represent cases where one instruction may prevent another instruction from executing; for instance, an `if`-statement controls its `then`- and `else`-clauses. Data dependences represent the flow of data between instructions. We distinguish *register* data dependences from *memory* data dependences. Register and control dependences are computed quickly in practice.

Memory dependences represent data *flow* through a memory location, or additional constraints such as *anti*- and *output*-dependences. Exactly determining memory dependences is undecidable. To mitigate this, standard practice allows analyses to fail: given a pair of operations which access memory, report *no*- or *must*-depend, or fall-back to *may*-depend when an answer cannot be decided. Clients interpret may-depend conservatively.

We use `Query`$(v_1.\text{inst}, v_2.\text{inst}, \text{type})$ to denote a demand-driven analysis query that determines whether there is a memory dependence from the instruction associated with vertex v_1 to the instruction associated with vertex v_2; type is either *Loop-Carried* or *Intra-Iteration*. Any analysis—no matter its internal structure or operation—can be packaged to provide this query interface.

In the algorithms below, `TarjanSCC` refers to Tarjan's Algorithm for Strongly Connected Components [33]. Tarjan's algorithm reports SCC structure as well as a topological sort of those components and runs in time linear in the number of vertices and edges.

3.1 Baseline Algorithm

The baseline algorithm (Algorithm 1) builds a full PDG, including all register, control and memory dependences. To find memory dependences, it queries every pair of vertices (corresponding to instructions in the IR) which access memory to determine if there is a loop-carried or intra-iteration memory dependence. It then computes the strongly connected components of that PDG.

Algorithm 1: Baseline computeDagScc(V)

```
let E := computeRegisterDeps($V$) $\cup$ computeControlDeps($V$);
foreach vertex $v_{src} \in V$ which accesses memory do
    foreach vertex $v_{dst} \in V$ which accesses memory do
        if Query($v_{src}$.inst, $v_{dst}$.inst, Loop-Carried) then
            let E := E $\cup$ {⟨$v_{src}$, $v_{dst}$, Loop-Carried⟩};
        end
        if Query($v_{src}$.inst, $v_{dst}$.inst, Intra-Iteration) then
            let E := E $\cup$ {⟨$v_{src}$, $v_{dst}$, Intra-Iteration⟩};
        end
    end
end
return TarjanSCC($V$, $E$);
```

3.2 Client-Agnostic Algorithm

We first present a client-agnostic version of the proposed algorithm, listed in Algorithm 2. This algorithm is written without destructive updates to simplify the proofs in Section 4, though a real implementation may save space by overwriting old values. We label points "X" and "Y" to clarify those proofs.

Similar to the baseline algorithm, the client-agnostic method starts by computing register and control dependences. This yields a PDG at Point X which is only partially computed since it lacks memory dependences. Next, it performs queries only between the vertices of select components in `withTheGrain` and `againstTheGrain`. These components are selected so that they will quickly cause components to merge into larger components. This leads to a savings in the number of memory dependence queries since dependences between vertices in a common component cannot further constrain the DAG_{SCC}.

For simplicity of implementation, we chose to recompute the SCCs by invoking `TarjanSCC` multiple times rather than using an incremental component maintenance algorithm. This reduces the amount of code to write and allows us to use simpler data structures internally. The cost of computing SCCs does not have a significant impact on overall performance.

Algorithm 2: Client-Agnostic computeDagScc(V)

```
let E := computeRegisterDeps($V$) $\cup$ computeControlDeps($V$) ;
let TopSort$_0$ := TarjanSCC($V$, $E$) ;
• (Point X)
let $E_0$ := E $\cup$ withTheGrain(E, TopSort$_0$) ;
for i = 1 to ∞ do
    • (Point Y)
    let E$'$ := againstTheGrain(TopSort$_{i-1}$) ;
    if $E' = \emptyset$ then
        return TopSort$_{i-1}$ ;
    end
    let E$'_i$ := E$_{i-1}$ $\cup$ E$'$ ;
    let TopSort$_i$ := TarjanSCC($V$, E$'_i$) ;
    let E$_i$ := E$'_i$ $\cup$ withTheGrain(E$'_i$, TopSort$_i$) ;
end
```

The routine `withTheGrain` (Algorithm 3) considers pairs of components c_{early} and c_{late} where c_{early} appears *before* c_{late} in the topological sorting of components. We exploit the feature that Tarjan's algorithm provides a topological sorting of the components with no additional computation. Figure 2(a) shows a topological sort. `withTheGrain` only performs queries that flow along topological order (i.e. from c_{early} to c_{late}), and only between components that are not already immediately ordered.[1] Such queries neither cause separate components to merge, nor invalidate the topological sorting of components, as illustrated in Figure 2(b).

Algorithm 3: withTheGrain(E$_0$, TopSort)

```
let E$'$ := $\emptyset$;
let $N$ := size(TopSort);
for i = N-1 down to 0 do
    let $c_{\text{late}}$ := TopSort($i$);
    for j = i-1 down to 0 do
        let $c_{\text{early}}$ := TopSort($j$);
        if $\neg$hasEdge($c_{early}$, $c_{late}$, $E_0$) then
            let E$'$ := E$'$ $\cup$ findOneEdge($c_{\text{early}}$, $c_{\text{late}}$);
        end
    end
end
return E' ;
```

The routine `againstTheGrain` (Algorithm 4) searches for dependences between pairs of components. Unlike `withTheGrain`, `againstTheGrain` only performs queries which may add edges that violate topological sort order, i.e. those from a vertex in a component c_{late} to a vertex in a topologically-earlier c_{early}. The rationale is that such queries quickly form larger components (Figure 2(c)). Large components have a compounding effect, further reducing the number of queries performed later. This routine performs enough queries to test every absence of an edge if none exists, allowing the algorithm to report that two components are separate.

It may surprise some readers that `againstTheGrain` has a `break` statement in its inner loop. This `break` stops searching for edges *from* a given component after it finds one, allowing the algorithm to recompute SCCs before resuming the search. This `break` is not necessary for correctness, however, it benefits performance. By recomputing SCCs, this formulation of `againstTheGrain` merges SCCs early, actively searching for opportunities to skip more queries. Later invocations of `againstTheGrain` perform fewer queries because there are fewer, larger components.

Algorithm 4: againstTheGrain(TopSort)

```
let $E$ := $\emptyset$;
let $N$ := size(TopSort);
for i = N-1 down to 0 do
    let $c_{\text{late}}$ := TopSort($i$);
    for j = i-1 down to 0 do
        let $c_{\text{early}}$ := TopSort($j$);
        let E$'$ := findOneEdge($c_{\text{late}}$, $c_{\text{early}}$);
        let E := E $\cup$ E$'$;
        if $E' \neq \emptyset$ then
            break;
        end
    end
end
return E;
```

The routine `findOneEdge` (Algorithm 5) performs queries from a source component to destination component. It stops after it finds the first edge between them since additional edges would order those two components redundantly.

3.3 Client-Aware DAG_{SCC} Constructions

The DAG_{SCC} guides clients such as DSWP [29] or loop fission [2, 20]. Some clients want more information than the DAG_{SCC} offers. The proposed algorithm may be extended to needs of particular clients. Despite these additional requirements, one can implement these extensions while achieving comparable performance improvements over the baseline. Two dimensions characterize client-specific extensions of the algorithm: additional requirements of dependence information and opportunities to abort early.

[1] This test should exclude components that are *ordered*. We use *immediate ordering* as a fast approximation to avoid transitive closure. Consequently, `withTheGrain` may conservatively add some edges from class 1(d).

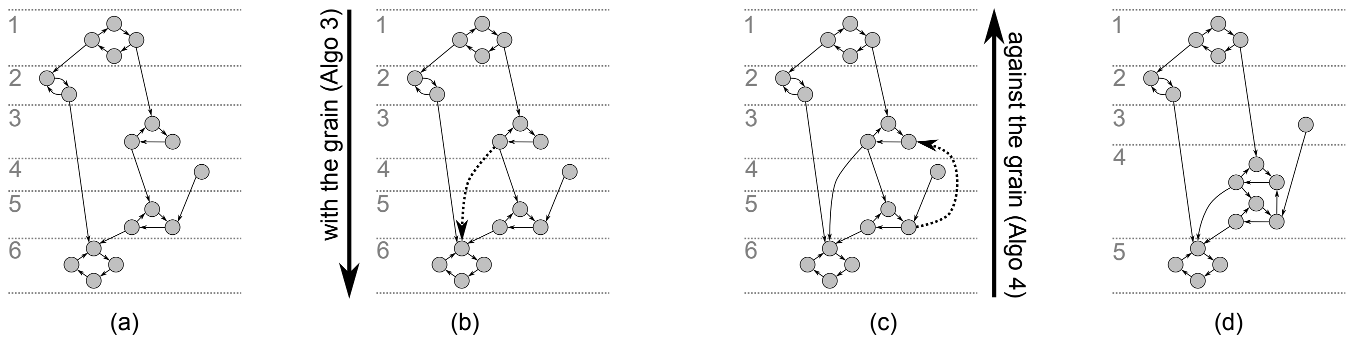


Figure 2: A partially computed PDG. (a) Topological sort (grey lines) imposes a total order on the partially ordered components. (b) `withTheGrain` (Algorithm 3) performs queries to discover edges between components with increasing position. Such edges neither cause SCCs to merge nor invalidate the topological sort. Here, a new edge is discovered from component three to six. (c) `againstTheGrain` (Algorithm 4) performs queries to discover edges between components with decreasing position. Here, a new edge is discovered from component five to three. (d) When `againstTheGrain` discovers new edges the topological sort is invalidated and components may merge.

Algorithm 5: findOneEdge(c_{src}, c_{dst})

```
foreach vertex $v_{src} \in c_{src}$ which accesses memory do
    foreach vertex $v_{dst} \in c_{dst}$ which accesses memory do
        if Query($v_{src}$.inst, $v_{dst}$.inst, Loop-Carried) then
            return {⟨$v_{\text{src}}$, $v_{\text{dst}}$, Loop-Carried⟩};
        end
    end
end
foreach vertex $v_{src} \in c_{src}$ which accesses memory do
    foreach vertex $v_{dst} \in c_{dst}$ which accesses memory do
        if Query($v_{src}$.inst, $v_{dst}$.inst, Intra-Iteration) then
            return {⟨$v_{\text{src}}$, $v_{\text{dst}}$, Intra-Iteration⟩};
        end
    end
end
return ∅;
```

Parallel Stage Decoupled Software Pipelining (PS-DSWP) is an illustrative example of such a client. PS-DSWP is an automatic thread-extraction technique with great performance potential [28, 36]. PS-DSWP partitions the DAG$_{\text{SCC}}$ into pipeline stages such that all communication and synchronization flow forward in pipeline order (i.e. forbidding cyclic communication among worker threads). PS-DSWP delivers scalable speedups when a large *parallel stage* is available; conversely, PS-DSWP does not transform the code when no significant parallel stage is present.

PS-DSWP requires slightly more dependence information than is present in the DAG$_{\text{SCC}}$, thus creating a meaningful evaluation scenario. Beyond the DAG$_{\text{SCC}}$, PS-DSWP classifies each SCC as either *DOALL* or *Sequential* according to the absence or presence of loop-carried dependences. Parallel stages are assembled from the DOALL SCCs such that no loop-carried dependence exists among the operations in the parallel stage. Algorithm 2 does not guarantee that such queries will be performed. To support PS-DSWP, the algorithm must perform additional queries to classify each SCC as DOALL or Sequential. These additional queries are still fewer than the full PDG and DAG$_{\text{SCC}}$ guides the compiler to search for such queries. Furthermore, DAG$_{\text{SCC}}$ construction may abort as soon as no significant parallel stage is possible.

We extend Algorithm 2 for the needs of PS-DSWP in Algorithm 6. The routine `checkReflexiveLC` checks for loop-carried dependences from any operation in a DOALL SCC to itself, stopping after it finds one. `checkWithinSccLC` checks for loop-carried dependences from any operation located in a DOALL SCC to any other operation in the same SCC. The latter contains the former, but experience suggests that prioritizing reflexive queries tends to exclude many components from the parallel stage after only a linear number of queries, whereas querying in `checkWithinSccLC` is quadratic. At the end, the algorithm invokes `checkWithinSccLC` again since components have grown, potentially including more loop-carried dependences. These checks are cheaper than full PDG construction since they only query among DOALL SCCs.

Algorithm 6: PS-DSWP-Aware computeDagScc(V)

```
let E := computeRegisterDeps($V$) ∪ computeControlDeps($V$);
let TopSort := TarjanSCC($V$, $E$);
abortIfPsInsubstantial(V,E,TopSort);
let E := E ∪ checkReflexiveLC($V$);
abortIfPsInsubstantial(V,E,TopSort);
let E := E ∪ checkWithinSccLC(TopSort) ;
abortIfPsInsubstantial(V,E,TopSort);
let E := E ∪ withTheGrain(E, TopSort);
while true do
    let E′ := againstTheGrain(TopSort);
    if $E' = \emptyset$ then
        break;
    end
    let E := E ∪ E′;
    let TopSort := TarjanSCC($V$, $E$);
    abortIfPsInsubstantial(V,E,TopSort);
    let E := E ∪ withTheGrain(E, TopSort) ;
end
let E := E ∪ checkBetweenDoallSccs(TopSort) ;
abortIfPsInsubstantial(V,E,TopSort);
let E := E ∪ checkWithinSccLC(TopSort) ;
return TopSort ;
```

Loop-carried dependences between DOALL SCCs prevent the simultaneous assignment of those components to the parallel stage. The routine `checkBetweenDoallSccs` performs queries to find such dependences. These checks are cheaper than full PDG construction, since they only consider pairs of DOALL SCCs. `abortIfPsInsubstantial` cancels construction if no substantial parallel stage is present whenever the upper bound on the parallel stage may change. For evaluation, we say a stage is "substantial" if it contains memory accesses or calls.

4. Proof of Correctness

We present a proof that our proposed method (Algorithm 2) produces a DAG$_{\text{SCC}}$ that is equivalent to the one produced by the baseline method (Algorithm 1), both in terms of partitioning the set of vertices V into the same SCCs, and in terms of drawing the same edges between SCCs.

Both algorithms partition the same set of vertices V. Let C_B, C_P represent the components returned by the baseline and proposed algorithms, respectively. Each algorithm computes its own set of edges E_B and E_P, respectively, between pairs of vertices in V. Two components in the DAG$_{\text{SCC}}$ are connected with an edge if there exists an edge between members of those components: for any components $c_1, c_2 \in C_B$, we write $c_1 \to_B c_2$ iff there is an edge $\langle v_1, v_2\rangle \in E_B$ such that $v_1 \in c_1$ and $v_2 \in c_2$. Similarly, for any components $c_1, c_2 \in C_P$, we write $c_1 \to_P c_2$ iff there is an edge $\langle v_1, v_2\rangle \in E_P$ such that $v_1 \in c_1$ and $v_2 \in c_2$.

Let $B(v) \in C_B$ denote the strongly connected component which contains vertex v as reported by the baseline algorithm. Let $P(v) \in C_P$ denote the strongly connected component which contains v as reported by the proposed algorithm.

We state our equivalence in Theorems 1 and 2.

Theorem 1 (C_B and C_P induce the same partition of V)**.** *For every $t, u \in V$, $B(t) = B(u)$ iff $P(t) = P(u)$.*

Proof. Follows immediately from Lemmas 3 and 5. ☐

Theorem 2 ($\langle C_B, \to_B\rangle$ is isomorphic to $\langle C_P, \to_P\rangle$)**.** *For every $t, u \in V$, $B(t) \to_B B(u)$ iff $P(t) \to_P P(u)$.*

Proof. We construct a correspondence $\Psi = B(v) \mapsto P(v)$.
Lemmas 3 and 5 show that Ψ is a bijective function.
Lemmas 4 and 6 show that $t \to_B u$ iff $\Psi(t) \to_P \Psi(u)$. ☐

We prove both Theorems using the following lemmas.

Lemma 1 (Forward Preservation of Edges, Simplified)**.** *Ignoring the* `break` *in Algorithm 4, if $\langle t, u\rangle \in E_B$ then $P(t) \to_P P(u)$.*

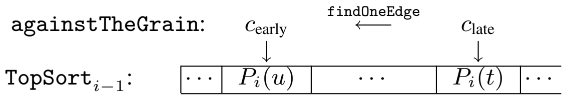
Proof. During an invocation of the proposed method (Algorithm 2), execution will necessarily reach Point Y.

Components evolve during the execution of the proposed algorithm; to avoid confusion we refer to specific versions of the components. Let $P_i(v)$ denote the strongly connected component which contains vertex v at Point Y in the i-th iteration of the loop. In other words, $P_i(v)$ finds the component that contains v within the variable TopSort$_{i-1}$. Note that $P(v)$ is the value of $P_i(v)$ during the final iteration.

We consider three cases based on the relative positions of $P_i(t)$ and $P_i(u)$ in the topological sort of components reported by `TarjanSCC`, observed at Point Y.

Case 1: During any iteration i, $P_i(u)$ appears before $P_i(t)$ in the topological sort.

During that iteration, the invocation of `againstTheGrain` (Algorithm 4) necessarily reaches an iteration during which $c_{\text{late}} = P_i(t)$. It visits every earlier component c_{early}, invoking `findOneEdge` on each until an edge is discovered. Ignoring the `break` statement in Algorithm 4, we will reach an iteration in which $c_{\text{early}} = P_i(u)$.



`findOneEdge` (Algorithm 5) will perform queries between the elements of $P_i(t)$ and $P_i(u)$ until an edge is found.

During the execution of the baseline algorithm, the call to Query(t.inst, u.inst, f) returns `true` given that $\langle t, u\rangle \in E_B$. Note that `Query` depends only on its arguments, so it behaves the same during the execution of the proposed algorithm.

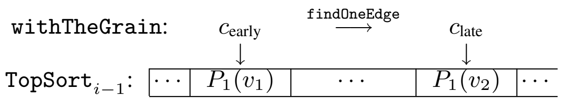
If `findOneEdge` reaches the iteration where $(v_{\text{src}}, v_{\text{dst}}) = (t, u)$, then Query($t$.inst, u.inst, f) will again return `true`, thus adding the edge $\langle t, u\rangle$. The only case it may not reach that iteration is when `findOneEdge` finds some other edge between those components. Thus, $P_i(t) \to_P P_i(u)$.

Case 2: During any iteration i, $P_i(t)$ appears at the same position as $P_i(u)$ in the topological sort. That is, $P_i(t) = P_i(u)$.

By reflexivity, $P_i(t) \to_P P_i(u)$.

Case 3: $P_i(u)$ never appears before or at the same position as $P_i(t)$ in the topological sort during any iteration.

$P_1(t)$ appears before $P_1(u)$ in the topological sort of components during the first iteration of the loop. The topological sort is not updated between Point X and Point Y in the first iteration, so $P_1(t)$ appears before $P_1(u)$ in the topological ordering before the invocation of `withTheGrain` (Point X in Algorithm 2).



The algorithm `withTheGrain` necessarily reaches an iteration during which $c_{\text{early}} = P_1(t)$ and $c_{\text{late}} = P_1(u)$. If there is not already an immediate ordering relationship $P_1(t) \to_P P_1(u)$, `withTheGrain` passes those components to `findOneEdge`. Since $\langle t, u\rangle \in E_B$, we know that Query($t$.inst, u.inst, f) returned `true`. Thus, `findOneEdge` must find an edge (either $\langle t, u\rangle$ or an earlier one) between these components: $P_1(t) \to_P P_1(u)$.

In all cases, we have $P_i(t) \to_P P_i(u)$ for some i.

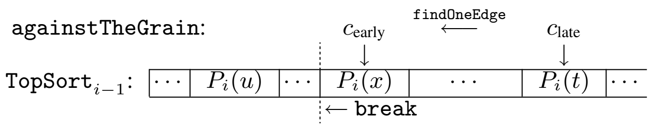
Observe that the proposed algorithm may add edges to the graph, but never removes edges from the graph. Adding edges may cause two separate components to merge into one, but never splits a component. Thus, for any vertex v and iteration i: $P_{i-1}(v) \subseteq P_i(v)$. Since $P(v)$ is the value of $P_j(v)$ in the final iteration j, it follows that $P(t) \to_P P(u)$. ☐

Lemma 2. *Considering the* `break` *in Algorithm 4, if $\langle t, u\rangle \in E_B$ then $P(t) \to_P P(u)$.*

Proof. The only difference between the simplified and proposed algorithms occurs in Lemma 1, Case 1: during iteration i, $P_i(u)$ appears before $P_i(t)$ in the topological sort.

The invocation of `againstTheGrain` (Algorithm 4) necessarily reaches an iteration during which $c_{\text{late}} = P_i(t)$. It visits every earlier component c_{early} invoking `findOneEdge` until an edge is discovered.

Suppose there is an intervening component $P_i(x) \neq P_i(u)$ such that `findOneEdge` discovers an edge $\langle w, x\rangle$ from $w \in P_i(t)$ to $x \in P_i(x)$. This edge causes the loop to break before visiting $c_{\text{early}} = P_i(u)$.



After the new edge is found, the algorithm recomputes components and may change their relative positions. Either $P_{i+1}(u)$ precedes $P_{i+1}(t)$ in the topological sort TopSort$_{i+1}$, or they merge, or $P_{i+1}(t)$ precedes $P_{i+1}(u)$. In the latter case, the subsequent invocation of `withTheGrain` immediately detects an edge from a vertex in $P_{i+1}(t)$ to a vertex in $P_{i+1}(u)$. Thus we need only consider the case in which they maintain their relative topological order.

We argue inductively that such an iteration of Algorithm 2 will be followed by another iteration that falls into Case 1, yet has one fewer intervening component. Assume that $P_{i+1}(u)$ precedes

$P_{i+1}(t)$. Observe that the component $P_{i+1}(x)$ cannot appear before $P_{i+1}(t)$ because of the newly discovered edge $\langle w, x\rangle$. Consequently, there is one fewer intervening component that could cause an later invocations of `againstTheGrain` to break. As a new edge $\langle w, x\rangle$ was found, the loop in Algorithm 2 will perform at least one more iteration. Thus, in the next iteration `againstTheGrain` will be one `break` closer to Lemma 1. After sufficient iterations, all intervening components have been eliminated and Lemma 1 Case 1 applies.

□

Lemmas 1 and 2 demonstrate that edges in E_B will order components in C_P. We next strengthen this statement to show that edges between components in C_B will order components in C_P in Lemma 4, but first we prove the following.

Lemma 3 (Wholeness of Components, Forward)**.** *For any vertices $t, u \in V$, if $B(t) = B(u)$ then $P(t) = P(u)$.*

Proof. Vertices t and u belong to the same strongly connected component of C_B, so there is a path from t to u:

$$\langle t, t_1\rangle, \langle t_1, t_2\rangle, \ldots, \langle t_{j-1}, t_j\rangle, \langle t_j, u\rangle \in E_B$$

and a path from u to t:

$$\langle u, u_1\rangle, \langle u_1, u_2\rangle \ldots, \langle u_{k-1}, u_k\rangle, \langle u_k, t\rangle \in E_B.$$

By Lemma 2 this implies that there is a cycle across the corresponding components of C_P: $P(t) \rightarrow_P P(t_1) \rightarrow_P \ldots \rightarrow_P P(t_j) \rightarrow_P P(u) \rightarrow_P P(u_1) \rightarrow_P \ldots \rightarrow_P P(u_k) \rightarrow_P P(t)$.

This, in turn, implies that $P(t) = P(t_1) = \ldots = P(t_j) = P(u) = P(u_1) = \ldots = P(u_k)$. □

Lemma 4 (Preservation of Structure, Forward)**.** *For any vertices $t, u \in V$, if $B(t) \rightarrow_B B(u)$ then $P(t) \rightarrow_P P(u)$.*

Proof. By definition of $\rightarrow_B$, there is an edge $\langle x, y\rangle \in E_B$ such that $x \in B(t)$ and $y \in B(u)$. By Lemma 2 we know $P(x) \rightarrow_P P(y)$.

Since components are a partition of all vertices, $x \in B(t)$ implies $B(t) = B(x)$. Similarly, $B(u) = B(y)$.

By Lemma 3, $P(t) = P(x)$ and $P(u) = P(y)$.

Thus, $P(t) \rightarrow_P P(u)$. □

Lemma 5 (Wholeness of Components, Reverse)**.** *For any two vertices $t, u \in V$, if $P(t) = P(u)$ then $B(t) = B(u)$.*

Proof. Vertices t and u belong to the same strongly connected component of C_P, so there is a path from t to u:

$$\langle t, t_1\rangle, \langle t_1, t_2\rangle, \ldots, \langle t_{j-1}, t_j\rangle, \langle t_j, u\rangle \in E_P$$

and a path from u to t:

$$\langle u, u_1\rangle, \langle u_1, u_2\rangle \ldots, \langle u_{k-1}, u_k\rangle, \langle u_k, t\rangle \in E_P.$$

Since the baseline performs *all* queries, $E_P \subseteq E_B$, and the same a cycle connects the corresponding components of C_B: $B(t) \rightarrow_P B(t_1) \rightarrow_P \ldots \rightarrow_P B(t_j) \rightarrow_P B(u) \rightarrow_P B(u_1) \rightarrow_P \ldots \rightarrow_P B(u_k) \rightarrow_P B(t)$.

This, in turn, implies that $B(t) = B(t_1) = \ldots = B(t_j) = B(u) = B(u_1) = \ldots = B(u_k)$. □

Lemma 6 (Preservation of Structure, Reverse)**.** *For any vertices $t, u \in V$, if $P(t) \rightarrow_P P(u)$ then $B(t) \rightarrow_B B(u)$.*

Proof. By definition of $\rightarrow_P$, there is an edge $e = \langle x, y\rangle \in E_P$ such that $x \in P(t)$ and $y \in P(u)$. Since components are a partition of vertices, $P(x) = P(t)$ and $P(y) = P(u)$. By Lemma 5, it follows that $B(x) = B(t)$ and $B(y) = B(u)$. Since $E_P \subseteq E_B$, $e \in E_B$ and therefore $B(x) \rightarrow_B B(y)$. By substitution we obtain that $B(t) \rightarrow_B B(u)$ as desired. □

5. Empirical Validation

To evaluate this technique, we implement the baseline (Section 3.1), client-agnostic (Section 3.2), and PS-DSWP-aware (Section 3.3) algorithms in the LLVM infrastructure [21] revision 164307. Each algorithm is augmented with a 30 minute timeout.

Each algorithm uses the same data structures to represent the program dependence graph and strongly connected components. The PDG data structure is a sorted adjacency-list representation, which performs well since PDGs tend to be sparse graphs. The data structure is capable of representing partial knowledge of memory dependences: between any pair of vertices, a memory dependence is *present*, *absent*, or *unknown*. Thus, none of the algorithms will ever perform the same query more than once. The cost of manipulating the data structure had negligible effect on most experiments.

We evaluated these techniques on 20 SPEC 2006 benchmarks [31]. The experiments exclude eight FORTRAN benchmarks because the front-end supports only C and C++. Each benchmark was compiled under two optimization regimens. The less-optimized regimen uses `clang -O1`. The more-optimized regimen is designed to create larger scopes that are harder to analyze. Specifically, we apply internalization,[2] devirtualization of indirect calls, and `-O3`.

We profile each benchmark to identify 366 hot loops. Hot loops are those loops whose running time consumes at least 5% of application running time, and which perform at least five iterations per invocation, on average. The hot loops found among the benchmarks are summarized in Table 1. It is not always possible to correlate hot loops between the less- and more-optimized regimens; optimization may break a hot loop into several, or reduce the execution time of a loop below the threshold.

Experiments run on an eight core 1.6GHz Xeon E5310. The machine has 8GB RAM and runs 64-bit Linux 2.6.32. All benchmarks are compiled to 64-bit, little-endian code. In this section, we use *instruction* to refer to an LLVM virtual instruction. All measurements experienced negligible variance.

5.1 Evaluation Analysis Framework

The overall performance benefit of the proposed algorithms depends greatly upon the performance characteristics of the underlying analysis framework. Many algorithms implement dependence analysis [9, 10, 13, 18, 22, 23, 32, 38, 40], yet these algorithms are not easily compared. Each occupies a distinct niche in the precision-efficiency trade-off [14].

We fix a single dependence analysis framework across all experiments. This framework combines separate analyses under a common, demand-driven interface. The interface accepts *queries* about the intra-/inter-iteration dependence relationship between two memory operations with respect to a loop of interest. As each query enters the analysis framework, it passes through each analysis in turn to find the most optimistic answer; thus, the combination features the strengths of each member.

The evaluation analysis includes nineteen separate analyses developed internally at Princeton. These analyses are designed to support automatic thread-extraction in general purpose codes such as SPEC INT [31]. Thus, our analyses emphasize precision in codes with linked-data structures and are sensitive to loops. The suite of analyses includes control- and data-flow sensitive analyses, calling-context sensitive analyses, induction-variable analyses, analyses specializing in external functions from the C and C++ standard libraries, a rudimentary shape analysis, and analyses which reason about call sites. These analyses are either purely demand-driven or

[2] Internalization asserts that the input program is the *whole* program, i.e. that no external libraries reference any of the program's exported symbols. It is similar to marking all global symbols with C's `static` keyword.

	Less-Optimized Regimen					More-Optimized Regimen				
	Hot	Coverage		Size		Hot	Coverage		Size	
Benchmark	Loops	Hottest	Coldest	Largest	Smallest	Loops	Hottest	Coldest	Largest	Smallest
400.perlbench	4	25.6%	5.5%	163 (#1)	9 (#4)	3	25.8%	11.1%	266 (#2)	66 (#3)
401.bzip2	9	73.5%	8.5%	597 (#7)	7 (#9)	9	71.5%	5.6%	2236 (#9)	7 (#8)
403.gcc	16	79.2%	5.0%	5800 (#1)	7 (#12)	11	79.8%	5.4%	11326 (#1)	40 (#2)
429.mcf	7	99.9%	6.3%	81 (#4)	26 (#1)	8	99.7%	8.6%	1352 (#1)	47 (#8)
433.milc	9	52.5%	7.5%	159 (#1)	12 (#9)	15	32.5%	5.4%	298 (#1)	19 (#7)
435.gromacs	5	99.9%	18.6%	671 (#1)	23 (#5)	8	99.4%	6.2%	10191 (#1)	72 (#7)
444.namd	16	99.9%	5.2%	1266 (#10)	9 (#2)	21	100.0%	6.1%	1271 (#14)	66 (#10)
445.gobmk	20	100.0%	5.0%	3868 (#7)	12 (#11)	20	99.9%	5.3%	3099 (#7)	39 (#13)
447.dealII	20	100.0%	5.5%	140 (#17)	10 (#10)	16	100.0%	5.6%	788 (#4)	6 (#6)
450.soplex	6	50.7%	6.4%	118 (#5)	15 (#6)	9	69.4%	5.5%	1034 (#4)	15 (#7)
453.povray	6	99.9%	28.8%	90 (#5)	23 (#6)	7	99.9%	5.6%	258 (#1)	13 (#7)
456.hmmer	6	100.0%	6.4%	277 (#2)	11 (#4)	6	100.0%	7.2%	240 (#1)	13 (#5)
458.sjeng	7	100.0%	9.5%	779 (#4)	147 (#1)	9	99.9%	5.4%	3359 (#7)	13 (#8)
462.libquantum	15	74.2%	4.9%	49 (#4)	5 (#6)	12	94.6%	5.7%	97 (#1)	9 (#11)
464.h264ref	8	100.0%	6.7%	680 (#8)	159 (#3)	8	100.0%	11.1%	1483 (#8)	128 (#3)
470.lbm	2	99.8%	99.1%	475 (#2)	23 (#1)	2	99.6%	99.0%	1175 (#1)	475 (#2)
471.omnetpp	2	100.0%	13.2%	23 (#1)	23 (#1)	2	100.0%	19.0%	37 (#2)	22 (#1)
473.astar	9	65.4%	5.8%	61 (#3)	9 (#9)	12	56.6%	6.7%	238 (#1)	17 (#6)
482.sphinx3	10	95.0%	6.6%	429 (#2)	12 (#10)	8	94.5%	5.0%	2170 (#2)	12 (#1)
483.xalancbmk	2	98.0%	7.2%	28 (#2)	12 (#1)	1	97.6%	97.6%	36 (#1)	36 (#1)

Table 1: Hot loops from SPEC2006. "Coverage" is the percent of running time spent in the loop. "Size" is the number of LLVM IR instructions contained in the loop. "Largest" and "smallest" also contain the loop id, where #1 is the hottest loop, and #n is the coldest.

are largely demand-driven, i.e., a significant portion of analysis effort is performed in response to a query, not ahead of time.

Analysis services most queries quickly: half of all queries take less than 287.6μs (460K cycles); two thirds take less than 601.3μs (962K cycles); 90% take less than 1.0ms (2M cycles). Differences in query running time are due to differences in query complexity: for instance, analyzing a call site is generally more expensive than analyzing a `load` instruction. Across multiple runs, the running time of any one query exhibits negligible variance, suggesting that noise has minimal impact on timing results.

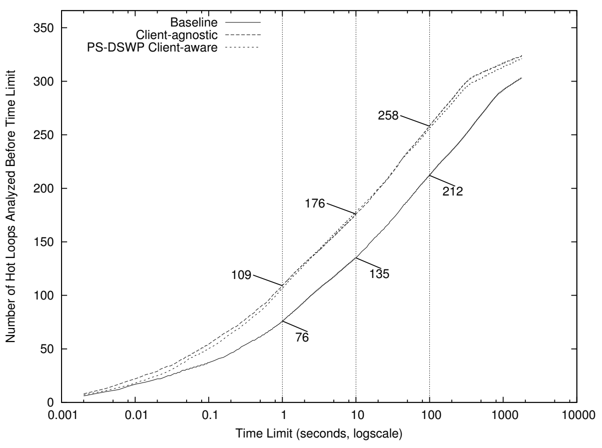


Figure 3: Largest sequence of hot loops analyzed before timeout.

5.2 Performance Improvement

The most direct impact of the proposed algorithm is a reduction in $\mathrm{DAG}_{\mathrm{SCC}}$ construction latency.

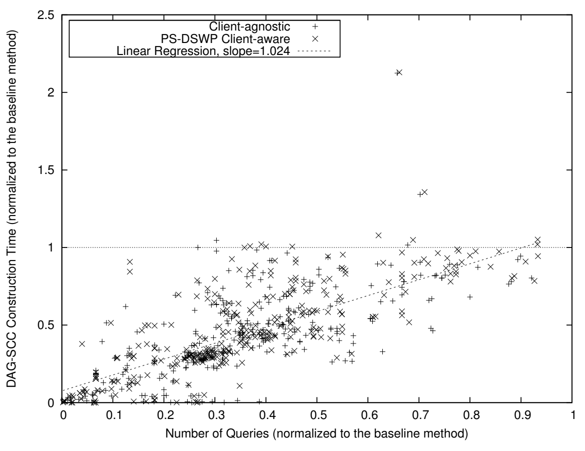
Figure 4(a) shows the time required to construct a $\mathrm{DAG}_{\mathrm{SCC}}$ for both the client-agnostic and PS-DSWP-aware algorithms. Each point represents a loop from the less- or more-optimized regimen, normalized to the running time of the baseline algorithm (smaller is better). The client-agnostic method is faster for all but 14 of 366 loops.

Performance improvements are due primarily to a reduction in the number of dependence analysis queries. Empirical results concur with the claim that the client-agnostic algorithm normalized running time is linear in the normalized number of queries. The Pearson's Correlation between the normalized construction time and normalized number of queries is 0.63.

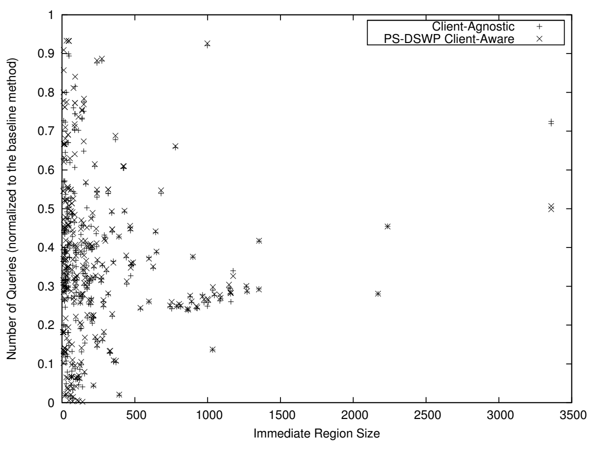
Figures 4(b)–(d) show factors which contribute to the reduction in queries. The fraction of queries performed by the client-agnostic method is related to both the average size of SCCs as well as the number of SCCs, yet is only mildly affected by the size of the region. This is because the algorithm elides queries for a class of redundant edges that grows with both average SCC size and number of SCCs (illustrated in Figure 1(d)–(e)). Empirical results concur with the claim that the client-agnostic method elides a greater fraction of queries in loops with fewer or larger components. The Spearman's Rank[3] between the average SCC size and normalized number of queries is -0.52. The Spearman's Rank between the number of SCCs and the normalized number of queries is 0.24.

One extreme outlier experiences more than 2$\times$ slowdown: the fourth-hottest loop from `458.sjeng`, located in function `std_eval`. In that loop, the proposed methods decrease the number of queries and the time spent on analysis queries. The cost of computing SCCs several times is less than the savings from fewer queries. However, the overhead of manipulating the sparse graph data structure is exceptionally high for this loop, canceling the savings. Further engineering work could reduce this overhead.

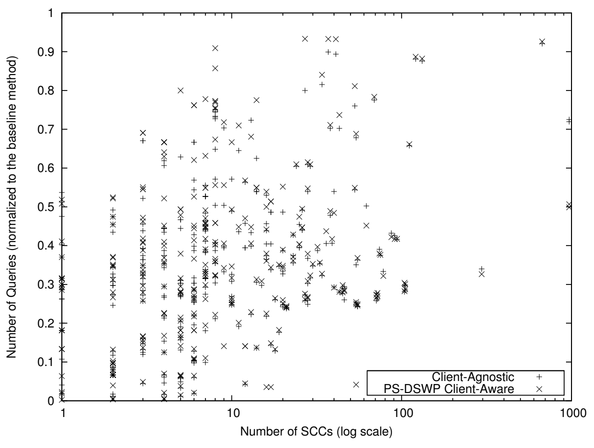
[3] Spearman's Rank is a measure of statistical dependence [19]. We use Spearman's Rank to support the claim of a *monotone* relationship, which is strictly weaker than a *linear* relationship indicated by Pearson's Correlation.



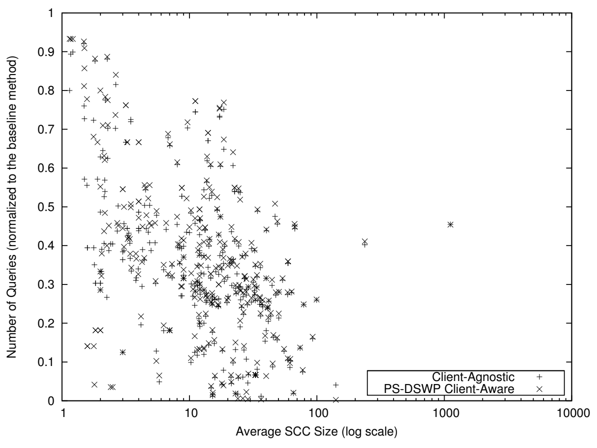
(a) Improvement in Time vs Improvement in Queries



(b) Improvement in Queries vs Size of Region



(c) Improvement in Queries vs Number of SCCs (log scale)



(d) Improvement in Queries vs Average SCC Size (log scale)

Figure 4: Improvement in running time and in number of queries, normalized to the baseline method.

5.3 Effect on Compiler Scalability

Analysis of whole benchmarks matters more than analysis of single loops. Compilers consider hot loops to plan an optimization strategy. We consider the question: *what is the largest sequence of loops which can be analyzed?* both in terms of the total number of loops, and in terms of the sum of execution coverage.

Figure 3 considers the largest sequences of loops that can be analyzed before varying limits on analysis time (log scale). To simulate a very large application, this experiment allows the compiler to select any of the loops from the entire benchmark suite. The client-agnostic method analyzes more loops than the baseline under the same time constraints. The PS-DSWP client-aware extensions cause a slight performance degradation from client-agnostic yet are still more efficient than the baseline.

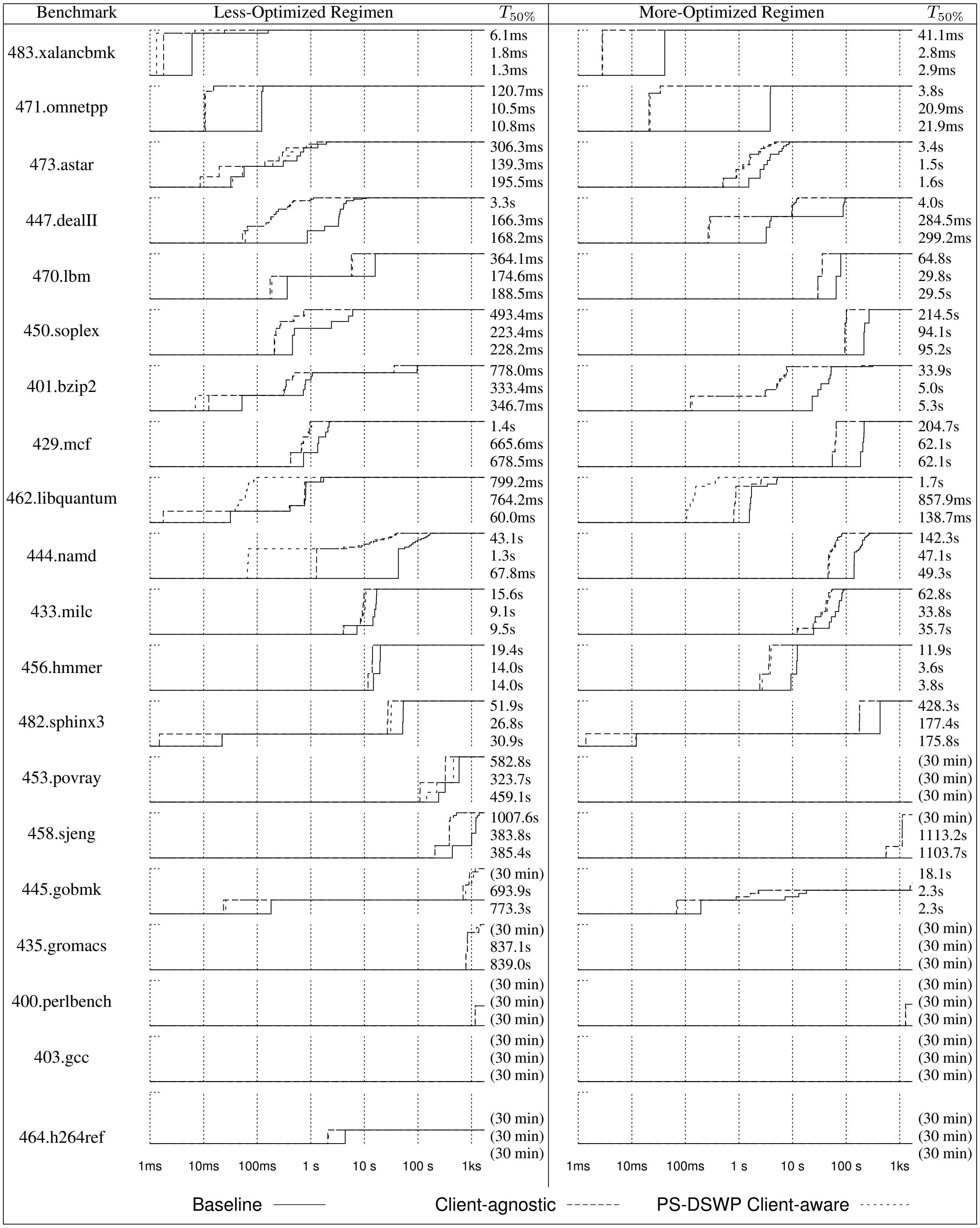
Not all loops are equally valuable. Amdahl's law encourages compilers to ration their time budget towards hot loops. Figure 5 explores how many hot loops (weighted by coverage) each method analyzes by a certain time. The compiler considers each loop from hottest to least-hot. On average, the client-agnostic method analyzes 50% of the hot loops 111.5s before the baseline, and the client-aware method achieves that 106.1s before the baseline. The proposed methods allow an optimizing compiler to analyze the code which most contributes to running time in shorter use cycles.

6. Related Work

Harrold et al. [12] present an algorithm to efficiently compute control dependences for a PDG. This technique does not address memory dependence analysis. In our setup, control analysis is cheap; memory analysis dominates construction time. There is theoretical evidence that precise analysis *must* be expensive [16, 26].

Scalability of memory analysis is an area of ongoing research. Approaches can be classified along levels of abstraction: low-level approaches improve the efficiency of the analysis, without considering the client; mid-level approaches assume some properties of the client to improve efficiency; high-level approaches modify the client so it more judiciously employs analysis or restructure the code so analysis will perform better.

Low-Level Many analyses are implemented as Maximum Fixed Points (MFP) of a set of data-flow equations. Several works optimize how programs are reduced to data-flow equations. Johnson et al. observe that formulating these equations with respect to the control flow graph is wasteful, proposing instead to formulate them along the Data-Flow Graph thereby reducing the number of identity relationships [18]. Similarly, Duesterwald et al. attempt to optimize the set of equations by identifying congruent equations through idempotency and common sub-expression elimination prior to computing the MFP [5]. Both techniques eliminate



Benchmark	Less-Optimized Regimen $T_{50\%}$	More-Optimized Regimen $T_{50\%}$
483.xalancbmk	6.1ms	41.1ms
	1.8ms	2.8ms
	1.3ms	2.9ms
471.omnetpp	120.7ms	3.8s
	10.5ms	20.9ms
	10.8ms	21.9ms
473.astar	306.3ms	3.4s
	139.3ms	1.5s
	195.5ms	1.6s
447.dealII	3.3s	4.0s
	166.3ms	284.5ms
	168.2ms	299.2ms
470.lbm	364.1ms	64.8s
	174.6ms	29.8s
	188.5ms	29.5s
450.soplex	493.4ms	214.5s
	223.4ms	94.1s
	228.2ms	95.2s
401.bzip2	778.0ms	33.9s
	333.4ms	5.0s
	346.7ms	5.3s
429.mcf	1.4s	204.7s
	665.6ms	62.1s
	678.5ms	62.1s
462.libquantum	799.2ms	1.7s
	764.2ms	857.9ms
	60.0ms	138.7ms
444.namd	43.1s	142.3s
	1.3s	47.1s
	67.8ms	49.3s
433.milc	15.6s	62.8s
	9.1s	33.8s
	9.5s	35.7s
456.hmmer	19.4s	11.9s
	14.0s	3.6s
	14.0s	3.8s
482.sphinx3	51.9s	428.3s
	26.8s	177.4s
	30.9s	175.8s
453.povray	582.8s	(30 min)
	323.7s	(30 min)
	459.1s	(30 min)
458.sjeng	1007.6s	(30 min)
	383.8s	1113.2s
	385.4s	1103.7s
445.gobmk	(30 min)	18.1s
	693.9s	2.3s
	773.3s	2.3s
435.gromacs	(30 min)	(30 min)
	837.1s	(30 min)
	839.0s	(30 min)
400.perlbench	(30 min)	(30 min)
	(30 min)	(30 min)
	(30 min)	(30 min)
403.gcc	(30 min)	(30 min)
	(30 min)	(30 min)
	(30 min)	(30 min)
464.h264ref	(30 min)	(30 min)
	(30 min)	(30 min)
	(30 min)	(30 min)

Figure 5: The client-agnostic, client-aware and baseline methods analyzing each benchmark. The horizontal axis measures time (seconds) from 1ms to 30 minutes on a log scale. The vertical axis is the fraction of loops analyzed before a that time, weighted by loop coverage. The compiler analyzes loops from hottest to coldest. $T_{50\%}$ shows the times when the Baseline, Client-agnostic, and PS-DSWP Client-aware methods reach 50% or *(30 min)* if they time out first. The client-agnostic method reaches 50% on average 111.5s earlier than baseline.

unnecessary relationships among program points, reducing MFP convergence time.

Calling-context sensitivity is a challenge for scalability: procedures exhibit drastically different memory access behavior depending on actual parameters and global state. One response has been the use of "procedure summaries," i.e. functions which compute a points-to set for a procedure as a function of its parameters and globally accessible state [9, 22]. In such solutions, there are still exponentially many calling contexts, yet summaries allow large numbers of contexts to be evaluated quickly. A complementary approach employs Datalog solvers which scale to the huge number of calling contexts using Binary Decision Diagrams [38]. This approach is restricted to analyses expressible in Datalog.

Mid-Level Demand-driven analyses employ an orthogonal approach [13, 32, 40]. Under the observation that many clients need only a fraction of all analysis queries, a demand-driven approach shifts effort from preprocessing into each query. This scales better as it only expends effort on queries which affect the client.

However, these approaches do not inform compiler authors of how to (re-)structure analysis clients as to perform the minimum number of analysis queries. The algorithms proposed in this paper have a synergistic relationship with demand-driven analyses since the proposed algorithms actively attempts to perform only select queries while still supporting advanced clients.

The Pruning-Refinement method [23] considers a spectrum of Datalog dependence analyses ranging from cheap-yet-imprecise to expensive-yet-precise. It first applies cheaper analyses to queries and, if successful, returns that answer to the client. Otherwise, it runs the slower, more precise analyses, using a byproduct of cheap analyses to prune extraneous inputs and reduce running times. This method has dramatic benefits on memory usage, but its running time improvements are less pronounced. This method is only applicable to analyses which can be formulated in Datalog.

High-Level The structure of input code, in particular the division into procedures, has an effect on analysis and optimization. Program restructuring techniques can improve analyzability. Procedure inlining and partial inlining not only reduce the overhead of a procedure call, but also improve memory analysis by disambiguating the relationship of call sites and callees [1, 25]. Region formation chooses scopes independently of procedure boundaries to make interprocedural analysis and optimization scale [11].

Ohata et al. [27] propose merging program statements before building a PDG to reduce memory consumption and analysis time. They modify analysis to treat groups of statements as one, leading to faster convergence, similar to Duesterwald et al. [5]. However, this approach is imprecise and falsely reports that some statements belong to a slice because they are merged with a statement truly in the slice. This technique can only merge non-call statements which are adjacent in the program source code and control equivalent, and requires tight integration with analysis. Our technique calls for no modifications to analysis.

Client-driven approaches [10] use cheap analyses first and retroactively apply precise analyses when imprecision limits the client. This approach requires analysis to track imprecision due to *polluting assignments*, and clients to request greater precision in important cases. This approach improves performance only when those important cases are less common than polluting assignments. In terms of the DAG_{SCC}, an augmented client must identify those conservative edges whose removal would split a component into several. In contrast, our method always uses the most precise analysis available and only queries edges that may merge components.

When used to improve analysis quality, the high-level approaches share a common failing: they conflate the separate concerns of analysis and transformation and break abstractions that are useful for the development of compilers.

7. Conclusion

The DAG_{SCC} provides strong insight into dependence structure over a large program scope and is sufficient to drive a large class of compiler optimizations. This paper demonstrates that the DAG_{SCC} can be computed much more efficiently than the naïve method of computing the strongly connected components of the full PDG. The savings from this technique allow a compiler to analyze larger scopes while still providing short turnaround times to the compiler's user. This makes aggressive optimization of large scopes palatable to a wider audience, and contributes to the universal deployment of aggressive whole-program optimization.

Acknowledgments

We thank the entire Liberty Research Group for their support and feedback during this work. We also thank the anonymous reviewers for their insightful comments. Additionally, we thank CJ Bell, Sid Sen, Sushant Sachdeva, and Chris Monsanto for commenting on early drafts. This material is based on work supported by National Science Foundation Grants 0964328 and 1047879. All opinions, findings, conclusions, and recommendations expressed throughout this work are those of the authors and do not necessarily reflect the views of the aforementioned funding agencies.

References

[1] R. Allen and S. Johnson. Compiling C for vectorization, parallelization, and inline expansion. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI)*, pages 241–249, June 1988.

[2] U. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer Academic Publishers, Norwell, MA, 1993.

[3] T. Chen, J. Lin, W. Hsu, and P. Yew. An empirical study on the granularity of pointer analysis in C programs. *Languages and Compilers for Parallel Computing (LCPC)*, pages 157–171, 2005.

[4] R. Cytron. DOACROSS: Beyond vectorization for multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing (ICPP)*, pages 836–884, 1986.

[5] E. Duesterwald, R. Gupta, and M. L. Soffa. Reducing the cost of data flow analysis by congruence partitioning. In *In International Conference on Compiler Construction*, pages 357–373. Springer-Verlag, 1994.

[6] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9:319–349, July 1987.

[7] J. S. Foster, M. Fähndrich, and A. Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *Proceedings of the 7th International Symposium on Static Analysis (SAS)*, pages 175–198, London, UK, UK, 2000. Springer-Verlag.

[8] R. Ghiya, D. Lavery, and D. Sehr. On the importance of points-to analysis and other memory disambiguation methods for C programs. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI)*, pages 47–58. ACM Press, 2001.

[9] B. Guo, M. J. Bridges, S. Triantafyllis, G. Ottoni, E. Raman, and D. I. August. Practical and accurate low-level pointer analysis. In *Proceedings of the 3rd International Symposium on Code Generation and Optimization (CGO)*, March 2005.

[10] S. Z. Guyer and C. Lin. Client-driven pointer analysis. In *In International Static Analysis Symposium*, pages 214–236. Springer-Verlag, 2003.

[11] R. E. Hank, W. W. Hwu, and B. R. Rau. Region-based compilation: An introduction and motivation. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 158–168, December 1995.

[12] M. J. Harrold, B. Malloy, and G. Rothermel. Efficient construction of Program Dependence Graphs. In *Proceedings of the 1993 ACM*

SIGSOFT international symposium on Software testing and analysis (ISSTA), pages 160–170, New York, NY, 1993.

[13] N. Heintze and O. Tardieu. Demand-driven pointer analysis. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation (PLDI)*, pages 24–34, New York, NY, 2001.

[14] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 2001.

[15] M. Hind and A. Pioli. Evaluating the effectiveness of pointer alias analyses. In *Science of Computer Programming*, pages 31–55, 1999.

[16] S. Horwitz. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Transactions on Programming Languages and Systems*, 19(1), January 1997.

[17] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *In proceedings of the Fourteenth International Conference on Software Engineering (CSE)*, pages 392–411, 1992.

[18] R. Johnson and K. Pingali. Dependence-based program analysis. In *In Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation (PLDI)*, pages 78–89, 1993.

[19] M. G. Kendall. *Rank Correlation Methods*. Charles Griffin and Company, Limited, London, 1948.

[20] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[21] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the Annual International Symposium on Code Generation and Optimization (CGO)*, pages 75–86, 2004.

[22] C. Lattner, A. Lenharth, and V. Adve. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Diego, California, June 2007.

[23] P. Liang and M. Naik. Scaling abstraction refinement via pruning. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[24] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. POSH: a TLS compiler that exploits program structure. In *PPoPP '06: Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 158–167, 2006.

[25] R. Muth and S. Debray. Partial inlining. Technical report, Department of Computer Science, University of Arizona, 1997.

[26] R. Muth and S. Debray. On the complexity of flow-sensitive dataflow analyses. In *In Proc. ACM Symp. on Principles of Programming Languages*, pages 67–80. ACM Press, 2000.

[27] F. Ohata, A. Nishimatsu, and K. Inoue. Analyzing dependence locality for efficient construction of program dependence graph. *Information and Software Technology*, 42(13):935 – 946, 2000.

[28] E. Raman, G. Ottoni, A. Raman, M. Bridges, and D. I. August. Parallel-stage decoupled software pipelining. In *Proceedings of the Annual International Symposium on Code Generation and Optimization (CGO)*, 2008.

[29] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August. Decoupled software pipelining with the synchronization array. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 177–188, September 2004.

[30] L. Rauchwerger and D. A. Padua. The LRPD test: Speculative runtime parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel Distributed Systems*, 10:160–180, February 1999.

[31] Standard Performance Evaluation Corporation. http://www.spec.org.

[32] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for java. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*, pages 59–76, New York, NY, 2005.

[33] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[34] A. J. Thadhani. Factors affecting programmer productivity during application development. *IBM Systems Journal*, 23(1):19 –35, 1984.

[35] C. Tian, M. Feng, V. Nagarajan, and R. Gupta. Copy or discard execution model for speculative parallelization on multicores. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 330–341, Washington, DC, 2008. IEEE Computer Society.

[36] H. Vandierendonck, S. Rul, and K. De Bosschere. The Paralax infrastructure: Automatic parallelization with a helping hand. In *Proceedings of the 19th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 389–400, 2010.

[37] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, (ICSE), pages 439–449, Piscataway, NJ, 1981.

[38] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, (PLDI), pages 131–144, New York, NY, 2004.

[39] *The Wisconsin Program-Slicing Tool, Version 1.1*, 2000. http://research.cs.wisc.edu/wpis/slicing_tool/.

[40] X. Zheng and R. Rugina. Demand-driven alias analysis for C. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 197–208, New York, NY, 2008.