# Query-Directed Adaptive Heap Cloning for Optimizing Compilers

Yulei Sui Yue Li Jingling Xue

Programming Languages and Compilers Group
School of Computer Science and Engineering
University of New South Wales, NSW 2052, Australia
{ysui,yueli,jingling}@cse.unsw.edu.au

## Abstract

Andersen's pointer analysis becomes more precise when applied with full heap cloning but unscalable for large, heap-intensive programs. In contrast, $k$-callsite-sensitive heap cloning can be faster but less precise for some programs.

In this paper, we make one step forward by enhancing Andersen's analysis with QUery-Directed Adaptive (QUDA) heap cloning for optimizing compilers. The novelty of our analysis, called QUDA, lies in performing $k$-callsite-sensitive heap cloning iteratively, starting with $k = 0$ (without heap cloning), so that an abstract heap object is cloned at iteration $k = i + 1$ only if some may-alias queries that are not answered positively at iteration $k = i$ may now be answered more precisely. QUDA, which is implemented in Open64, has the same precision as the state-of-the-art, FULCRA, a version of QUDA with exhaustive heap cloning, but is significantly more scalable. For 10 SPEC2000 C benchmarks and 5 C applications (totalling 840 KLOC) evaluated, QUDA takes only $4+$ minutes but exhaustive heap cloning takes $42+$ minutes to complete. QUDA takes only 75.1% of the time that Open64 takes on average to compile these 15 programs under "-O2".

***Categories and Subject Descriptors*** D.3.4 [*Processors*]: Compilers; F.3.2 [*Semantics of Programming Languages*]: Program Analysis

***General Terms*** Algorithms, Languages, Performance

***Keywords*** Pointer analysis, heap cloning

## 1. Introduction

Pointer analysis is critical in driving optimizations in optimizing compilers. To sharpen its precision, heap cloning aims to statically distinguish objects dynamically created at different calling contexts to an allocation site. We speak of *$k$-callsite-sensitive heap cloning* if a context is identified by the last $k$ callsites in an acyclic call path (with each recursion cycle being collapsed). There are two important special cases: *full heap cloning* (by acyclic call paths) when $k$ is sufficiently large and *allocation-site-based heap abstraction* without heap cloning when $k = 0$. In the latter case, one abstract heap object is created per allocation site. In the former case, a heap object returned from a callee is cloned distinctly at its distinct callsites. As $k$ increases, pointer analysis becomes strictly no less precise but more costly to compute. With full heap cloning, the precision for some programs can be significantly improved but the analysis can be prohibitively expensive or even does not terminate [22, 23].

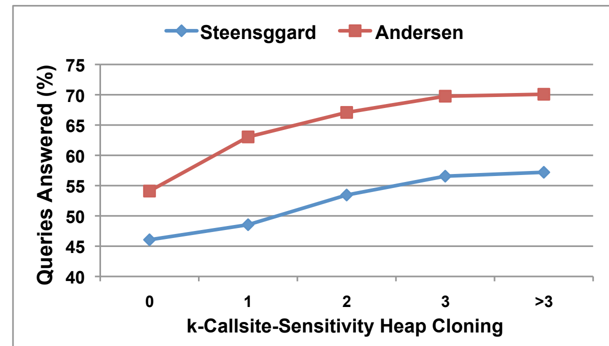



**Figure 1.** Percentage of must-not aliases disambiguated for `hmmer` (gene sequence search) among the machine-independent queries issued by WOPT (in Open64's backend) with $k$-callsite-sensitive heap cloning.

Steensgarrd's unification-based and Andersen's inclusion-based pointer analyses are commonly used in modern compilers. Earlier [16], Steensgarrd's analysis with full heap cloning is made scalable for C programs. As Andersen's analysis is significantly more precise but also significantly more costly to compute than Steensgarrd's, we have not seen the same success for Andersen's analysis with full heap cloning. In this paper, we make one step forward in solving this challenging problem. There are two motivations under-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*CGO '13* 23-27 February 2013, Shenzhen China.
978-1-4673-5525-4/13/$31.00 ©2013 IEEE. . . $15.00

pinning this work. First, Andersen's analysis is the most precise of all pointer analyses that are flow-insensitive (by not considering flow of control) and context-insensitive (by not distinguishing calling contexts for a callee). Due to recent advances, Andersen's analysis is now scalable for large programs [10, 25]. In the latest release (5.0) of the Open64 compiler, its pointer analysis is no longer unification-based but rather inclusion-based. Second, given the same heap cloning policy, Andersen's analysis can be more precise than Steensgarrd's in answering the alias queries issued by the compiler, as illustrated in Figure 1, for an application evaluated later in Section 5.

There are some earlier efforts on integrating heap cloning with Andersen's analysis. Full heap cloning for C is unscalable for large, heap-intensive programs [22] as the number of abstract heap objects cloned can be prohibitive in terms of both time and space. The demand-driven analysis formulated via context-free-language reachability [21, 27–29, 38] embraces naturally full heap cloning, but its scalability when deployed as a whole-program pointer analysis remains to be further investigated [36]. BDD-based techniques, while effective for solving context-sensitive pointer analyses [33, 39], may not scale for full heap cloning [18, 35, 36].

Among various heap cloning solutions, full heap cloning is the most precise but doing so blindly across all acyclic call paths can lead to uncontrollable overhead [22] with little benefit to support compiler optimizations [15, 17]. On the other hand, $k$-callsite-sensitive heap cloning for a small, fixed $k$ is also problematic as shown in Figure 10(a). The best value for $k$ precision-wise varies from program to program. For example, $k$ should be 2 for `sendmail` but 4 for `vortex`. As a result, fixing $k = 2$ for both is not precise for `vortex` but using $k = 4$ instead is unnecessarily over-costly for `sendmail`. Although it is tempting to tune a program manually by trying different values of $k$ to tradeoff precision and scalability [22, 35, 36], this is not systematic and can be as costly as full heap cloning since the program has to be reanalyzed with the prior results dropped and recomputed.

### 1.1 Our Solution

We show how to achieve the same precision of full heap cloning on top of Andersen's analysis efficiently for an optimizing compiler without actually performing full heap cloning. We achieve this by focusing our heap cloning efforts on improving the precision required for answering the pointer-related queries issued by the compiler. Our key insight is to enable heap cloning only where it is necessary to answer more alias queries precisely in an iterative manner.

As shown in Figure 2, we enhance Andersen's analysis with QUery-Directed Adaptive (QUDA) heap cloning. The novelty of the resulting analysis, called QUDA, lies in performing k-callsite-sensitive heap cloning iteratively, starting with $k = 0$ (without heap cloning), so that an abstract heap object is cloned at iteration $k = i + 1$ only if some alias

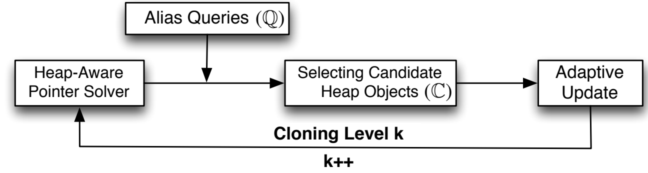



**Figure 2.** QUDA: Query-directed adaptive heap cloning.

queries that are not answered positively at iteration $k = i$ may now be answered more precisely.

Our analysis is heap-aware as each pointed-to target is guarded. For the subset of queries in $\mathbb{Q}$ that are detected to be may-aliases, the guards for their aliased locations reveal directly the set $\mathbb{C}$ of candidate heap objects to be further cloned so that these queries may be answered more accurately. The points-to relations affected by such candidates are removed. The pointer analysis in the next round is restarted, with each candidate in $\mathbb{C}$ being cloned as needed during the pointer resolution.

### 1.2 Contributions

- We propose a novel query-directed adaptive heap cloning (QUDA) approach that achieves the same precision as full heap cloning (subject to some stopping condition) for the same alias queries but is significantly more scalable.
- We present the first *heap-aware* pointer resolution method performed on a constraint graph that allows points-to relations to be efficiently removed (rather than just added).
- We have implemented QUDA in the Open64 compiler and evaluated it using the 10 largest SPEC2000 C benchmarks and 5 C applications (totalling 840 KLOC). By comparing QUDA with the state-of-the-art FULCRA [22] (a version of QUDA with exhaustive heap cloning), QUDA takes only $4+$ minutes but exhaustive heap cloning takes $42+$ minutes to complete. Indeed, QUDA takes only 75.1% of the time that Open64 takes on average to compile all these 15 programs under "-O2".

## 2. Background

We first describe the canonical representation used for a program, then introduce briefly Andersen's inclusion-based analysis, and finally, look at context-sensitive heap cloning.

### 2.1 Program Representation

There are four types of statements: $x = \&y$ (`address`), $x = *y$ (`load`), $*x = y$ (`store`) and $x = y$ (`copy`). Note that $x = **y$ can be transformed into $x = *t$ and $t = *y$ by introducing a new temporary variable $t$. Different fields of a struct are distinguished. However, arrays are considered monolithic. Every call contained in a procedure has the form $foo(r, a_1, \ldots, a_n)$, where the return variable $r$ and the actual parameter $a_1, \ldots, a_n$ are local variables in $foo$.

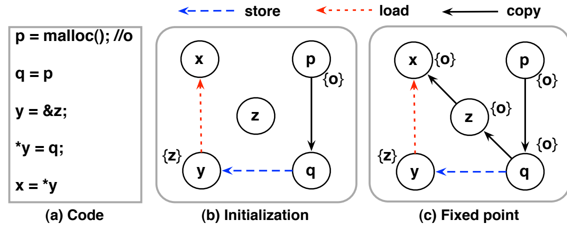



**Figure 3.** Pointer resolution in a constraint graph.

```
1: int main(){
2:  int *m,*n;
3:   m = bar(); //o2
4:   n = bar(); //o3
5: }
6: int* bar(){
7:  int* t = malloc(4);//o1
8:  return t;
9: }
```

(a) Explicit cloning

```
1: int main(){
2:   int**p,**q,*m,*n;
3:   p = &m; q = &n;
4:   bar(p); //o2
5:   bar(q); //o3
6: }
7: void bar(int** x){
8:   *x = malloc(4);//o1
9: }
```

(b) Indirect cloning

**Figure 4.** Two kinds of heap cloning. In either case, *m* and *n* point to *o1* without heap cloning and *m* points to *o2* and *n* points to *o3* with 1-callsite-sensitive heap cloning.

### 2.2 Constraint-based Andersen's Analysis

As illustrated in Figure 3, Andersen's analysis discovers points-to information by treating assignments as subset constraints using a constraint graph until a fixed point is reached. For the code in Figure 3(a), Andersen's analysis starts with the constraint graph given in Figure 3(b). For the address statements, `p=&o` and `y=&z`, the points-to information is directly recorded for their left-hand side variables. Then the analysis resolves `loads/stores` by adding new copy statements discovered. As `y` points to `z`, the two new copy statements related to $z$ are added as shown in Figure 3(c). The new points-to information discovered is propagated along the two edges. Finally, `x` is found to point to `o`.

### 2.3 Context-Sensitive Heap Cloning

Consider the two examples in Figure 4. Without heap cloning in either case, both `m` and `n` point to *o1*, which provides an abstraction for all objects created inside `bar`. So `*m` and `*n` are regarded as aliases. With 1-callsite-sensitive heap cloning, *o1* is cloned into *o2* and *o3* explicitly in Figure 4(a) via a return statement or indirectly in Figure 4(b) via a formal parameter. By distinguishing the two calling contexts to `bar` in either case, `m` points to *o2* and `n` points to *o3*. As a result, `*m` and `*n` are concluded not to alias precisely.

## 3. A Motivating Example

As shown in Figure 5(a), there is one allocation site in `getMem` and one in `allocMem`. If full heap cloning is enabled, explicit heap cloning happens at the callsites $\text{cs}_2$ and $\text{cs}_3$ in `readArray` when the side-effects of `getMem` are applied and indirect heap cloning happens at $\text{cs}_5$ and $\text{cs}_6$ in `allocArray` when the side-effects of `allocMem` are applied. Similarly, one more indirect cloning occurs at $\text{cs}_7$.

Let us see how QUDA, depicted in Figure 2, works given the two may-alias queries in $\mathbb{Q} = \{(*m, *n), (*p, *q)\}$ to be answered. There are two allocation sites in the program: $cs_1$ and $cs_4$. Each allocation site is associated with an HCT (Heap Cloning Tree), which captures the relations among all the objects created from the site, as will be explained later.

We focus only on how `readArray` is analyzed. There are two iterations illustrated in Figures 5(b) and (c):

$k = 0$ **(Figure 5(b))** No heap cloning occurs. In this case, the same heap object is returned in different calling contexts to an allocation site, as indicated by the HCTs. Our heap-aware pointer analysis proceeds as follows. Consider the leftmost box. Initially, `p` points to $(\mathit{true}, \texttt{x})$ unconditionally since `x` is guarded by *true*. When the side-effects of `getMem` are transferred to the callsites $cs_2$ and $cs_3$, `m` points to $(h_{o1}, o1)$ and `n` also points to $(h_{o1}, o1)$. In each case, the pointed-to target $o1$ may be spurious if $o1$ encoded by the guard $h_{o1}$ (indicated by its subscript) is further cloned. When the store `*m = p` is resolved, a copy edge $o1 \overset{h_{o1}}{\hookleftarrow} p$ is introduced except that it is guarded by the same condition in the pointed-to target $(h_{o1}, o1)$ of `m`. Similarly, when the load `q = *n` is resolved, a copy edge $q \overset{h_{o1}}{\hookleftarrow} o1$ is introduced. Our constraint resolution remains the same as the traditional one illustrated in Figure 3 except that all copy edges are guarded. The fixed point found is the one shown in the leftmost box.

We now try to answer the two alias queries issued. $\textsf{MayAlias}(\texttt{*m}, \texttt{*n}) = \mathit{true}$. Their aliased location $o1$ is guarded by $h_{o1}$, indicating that $o1$ should be selected as a candidate for cloning at $cs_2$ and $cs_3$ so that this query may be more accurately answered. $\textsf{MayAlias}(\texttt{*p}, \texttt{*q}) = \mathit{true}$ since `x` is their common target. While `x` is not a heap object, the guard $h_{o1}$ in the pointed-to target $(h_{o1}, \texttt{x})$ of `q` indicates that this points-to relation may be spurious if $o1$ is further cloned. Therefore, we obtain $\mathbb{C} = \{o1\}$.

Finally, the constraint graph is updated as shown in the middle box. Essentially, the points-to relations affected by $o1$ in `m`, `n`, `p` and `q` have been removed.

$k = 1$ **(Figure 5(c))** When the side-effects of `getMem` are transferred to $cs_2$ and $cs_3$, $o1$ is split into $o2$ and $o3$, respectively, as recorded in the HCTs. Thus, `m` points to $(h_{o2}, o2)$ and `n` to $(h_{o3}, o3)$. This time, we conclude that $\textsf{MayAlias}(\texttt{*m}, \texttt{*n}) = \mathit{false}$. Similarly, $\textsf{MayAlias}(\texttt{*p}, \texttt{*q}) = \mathit{false}$ as `q` no longer points to `x`. Since both queries are answered positively, no more iterations are needed.

Finally, as shown in the HCTs in Figure 5(c), QUDA clones only where it is necessary to answer the two alias queries. In contrast, full heap cloning is blind, resulting in more heap objects cloned unnecessarily with no benefit.

```
int foo(){
        char* x; char** arr = &x;
cs7:    allocArray(arr); //o7
        readArray(arr);
}
void readArray(char **p){
cs2:    char** m = getMem(); //o2
cs3:    char** n = getMem(); //o3
              *m = p;
        ...
        char*  q = *n;
}
char* getMem(){
cs1:    char* t = malloc(10); //o1
        ...
        return t;
}
void allocArray(char **arr){
        char* y; char** str = &y;
cs5:    allocMem(arr); //o5
cs6:    allocMem(str); //o6
        ...
}
void allocMem(char** mem){
cs4:    char* c = malloc(10); //o4
        *mem= c;
}
```

(a) code

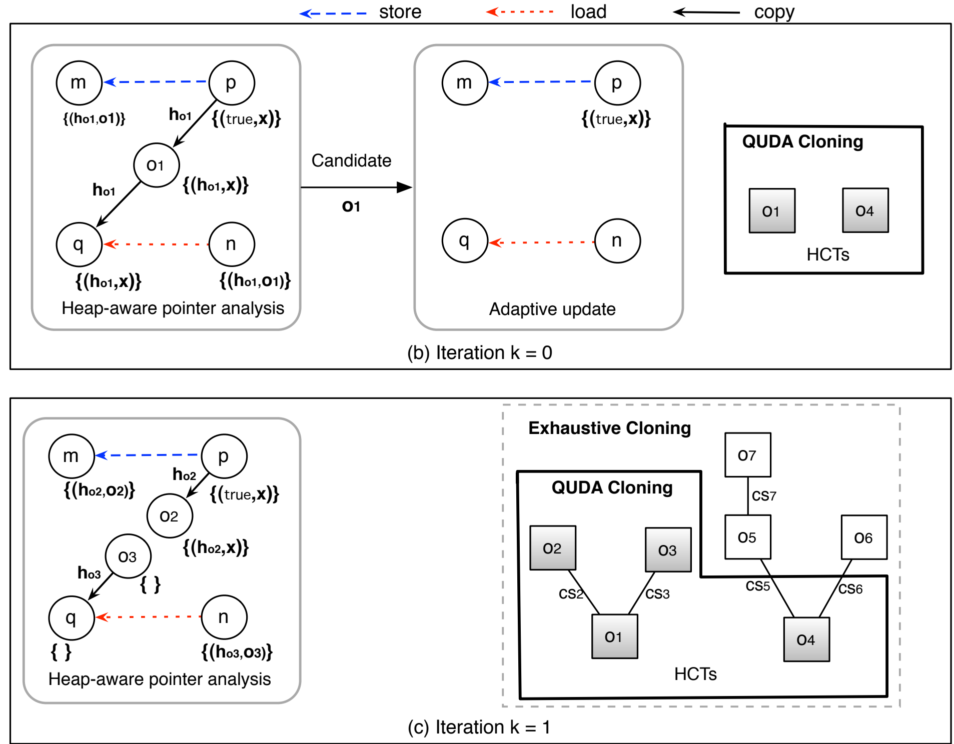



**Figure 5.** A motivating example (illustrating the analysis of `readArray` in its constraint graph).

## 4. Query-Directed Adaptive Heap Cloning

We describe our QUDA approach by discussing how to integrate it with an Andersen's analysis that is context-sensitive by acyclic call paths for method calls (i.e., their local variables) but 0-callsite-sensitive for heap cloning. As both components are orthogonal, we will focus only on describing the key phases of QUDA depicted in Figure 2.

Without loss of generality, our analysis is summary-based. As a result, the analysis interleaves a top-down phase and bottom-up phase iteratively until a fixed point is reached. During a top-down phase, the pointer information is propagated from a caller to its callees. During a bottom-up phase, the side-effects of a callee are summarized and promoted to its callsites for further pointer resolution.

As shown in Figure 2, QUDA is adaptive, guided by the alias queries issued by the compiler. Therefore, QUDA starts with an interprocedural context-sensitive heap-aware pointer analysis without heap cloning (Section 4.1). Then, the points-to information obtained is used to answer the alias queries generated by the compiler. The heap objects that can be further cloned to potentially improve the precision of some queries are selected as candidates for further heap cloning (Section 4.2). Finally, the constraint graph of a program is updated by removing the points-to relations (and constraints) affected by the candidate heap objects selected (Section 4.3).

### 4.1 Heap-Aware Pointer Analysis

This section presents our heap-aware pointer analysis for a program. Section 4.1.1 focuses on the intraprocedural analysis by describing how an individual procedure is analyzed without considering the side-effects of its callsites. Section 4.1.2 discusses the interprocedural analysis for a procedure by considering the side-effects of its callsites and performing context-sensitive heap cloning at its callsites.

#### 4.1.1 Intraprocedural Analysis

There are two tasks involved when analyzing a procedure, *heap-aware pointer resolution* and *side-effect summarization*. The former discovers the points-to sets of the pointers in the procedure with annotated heap information. The latter computes the side-effects of the procedure to be used by its callers for interprocedural context-sensitive analysis.

Due to flow-insensitivity, global variables are treated context-insensitively in a separate constraint graph in the standard manner [16, 20, 22]. Therefore, only the locals and heap objects are subject to context-sensitive analysis.

**Definition 1** (Points-to Sets)**.** *For a variable $p$, $PtrSet(p)$ is a set of locations possibly pointed to by $p$.*

During pointer resolution, each pointed-to location is guarded by a Boolean condition, which specifies the set of all heap objects whose further cloning (if possible) may render the pointed-to location spurious.

| | Statements | Points-Relations | Statements | Points-Relations |
|---|---|---|---|---|
| Input: | *v = x; | $\mathsf{PtrMap}(x) = \{(\mathit{true}, g)\}$ | t1 = *v; | $\mathsf{PtrMap}(v) = \{(h_{o3}, o3)\}$ |
| | y = *w; | $\mathsf{PtrMap}(v) = \{(h_{o1}, o1)\}$ | *t1 = x; | $\mathsf{PtrMap}(o3) = \{(h_{o4}, o4)\}$ |
| | *m = y; | $\mathsf{PtrMap}(w) = \{(h_{o1}, o1)\}$ | t2 = *w; | $\mathsf{PtrMap}(x) = \{(\mathit{true}, g)\}$ |
| | z = *n; | $\mathsf{PtrMap}(m) = \{(h_{o2}, o2)\}$ | z = *t2; | $\mathsf{PtrMap}(w) = \{(h_{o3}, o3)\}$ |
| | | $\mathsf{PtrMap}(n) = \{(h_{o2}, o2)\}$ | | |
| Output: | $\mathsf{PtrMap}(z) = \{(h_{o1} \wedge h_{o2}, g)\}$ | | $\mathsf{PtrMap}(z) = \{(h_{o3} \wedge h_{o4}, g)\}$ | |

(a) Value-flow via a sequence of objects (b) Value-flow via a hierarchy of nested objects

**Figure 6.** Two examples illustrating heap-aware pointer analysis.

**Definition 2** (Points-to Maps)**.** *The points-to map of a variable $p$, PtrMap(p), is a set of guarded pointed-to locations $(h, a)$, where $h$ is a Boolean condition and $a \in \mathit{PtrSet}(p)$.*

It is understood that each pointed-to location appears only once in $\mathsf{PtrMap}(p)$. If $(h, o)$ and $(h', o)$ are added to it together, then both are merged into $(h \vee h', o)$.

$$\texttt{[ADDR-HEAP]}\quad \frac{p = \&o \text{ (malloc(\dots))} \quad o \text{ is a heap object}}{\mathsf{PtrMap}(p) \cup= \{(h_o, o)\}}$$

$$\texttt{[ADDR-VAR]}\quad \frac{p = \&q \quad q \text{ is a local/global variable}}{\mathsf{PtrMap}(p) \cup= \{(\mathit{true}, q)\}}$$

$$\texttt{[COPY-DIR]}\quad \frac{p = q}{p \xleftarrow{\mathit{true}} q}$$

$$\texttt{[COPY-PROP]}\quad \frac{p \overset{h}{\hookleftarrow} q \quad (h_{q'}, q') \in \mathsf{PtrMap}(q)}{\mathsf{PtrMap}(p) \cup= \{(h_{q'} \wedge h, q')\}}$$

$$\texttt{[LOAD]}\quad \frac{p = *q \quad (h_{q'}, q') \in \mathsf{PtrMap}(q)}{p \xleftarrow{h_{q'}} q'}$$

$$\texttt{[STORE]}\quad \frac{*p = q \quad (h_{p'}, p') \in \mathsf{PtrMap}(p)}{p' \xleftarrow{h_{p'}} q}$$

**Table 1.** Rules used for heap-aware pointer resolution.

***Heap-Aware Pointer Resolution*** There are a total of six rules in Table 1. These rules differ from those used in the traditional constraint resolution (introduced briefly in Section 2.2) only in that all copy edges are now guarded.

There are two rules for address statements in the program. If $p$ receives directly what is returned from an allocation site, then the guard $h_o$ encodes the proposition that $o$ is the object returned from the allocation site, establishing the condition under which the points-to relation holds (`[ADDR-HEAP]`). Otherwise, when what is pointed to is a local or global, the points-to relation is unconditional, since it cannot be changed by heap cloning (`[ADDR-VAR]`).

For a load statement, a new copy edge guarded by $h'_q$ is generated from each pointed-to target $q'$ of $q$ to $p$ (`[LOAD]`). We do likewise for a store statement (`[STORE]`).

There are two rules for copy assignments. For a copy statement available in the program, `[COPY-DIR]` applies. For a copy edge guarded by $h$ (i.e., an indirect assignment created from a load/store), each guarded location pointed by $q$ is propagated to $p$ under this extra guard, $h$ (`[COPY-PROP]`).

**Example 1.** *Let us consider our example. In Figure 5(b), PtrMap$(p) = \{(\mathit{true}, x)\}$ by `[ADDR-VAR]` and PtrMap$(m) =$ PtrMap$(n) = \{(h_{o1}, o1)\}$ by first applying `[ADDR-HEAP]` to the malloc statement in `getMem` and then transferring the side-effects of `getMem` to $\text{cs}_2$ and $\text{cs}_3$ without heap cloning. When resolving `*m = p` by `[STORE]`, a copy edge $o1 \overset{h_{o1}}{\hookleftarrow} p$ is added. When resolving `q = *n` by `[LOAD]`, another copy edge $q \overset{h_{o1}}{\hookleftarrow} o1$ is added. When further resolving the two copy edges by `[COPY-PROP]`, we obtain PtrMap$(q) = \{(h_{o1}, x)\}$.*

*Similarly, in Figure 5(c), where heap cloning is applied, after having resolved all the constraints, we find that $\{(h_{o1}, x)\} \notin$ PtrMap$(q)$. Recall that $o1$ created at $\text{cs}_1$ is split into $o2$ and $o3$ at $\text{cs}_2$ and $\text{cs}_3$, respectively.*

Figure 6 illustrates our rules intuitively with two representative cases in terms of object containment. In Figure 6(a), $h_{o1} \wedge h_{o2}$ specifies the fact that $g$ flows (directly) into $z$ through a sequence of two objects (or data structures), $o1$ and $o2$. In Figure 6(b), $h_{o3} \wedge h_{o4}$ specifies the fact that $g$ flows into $z$ through a hierarchy of nested objects (or components) such that $g$ flows into $o4$, which flows into or is nested inside $o3$. So $g$ flows into $o3$ indirectly.

**Definition 3.** *Given a guarded points-to location $(h, a)$, an heap object $o$ is* encoded *by $h$ if $h$ contains $h_o$.*

In Figure 6(a), $(h_{o1} \wedge h_{o1}, g) \in \mathsf{PtrMap}(z)$. The points-to relation is conditional on how the two encoded objects $o_1$ and $o_2$ are further cloned. In Figure 6(b), $(h_{o3} \wedge h_{o4}, g) \in \mathsf{PtrMap}(z)$. The two encoded objects are $o_3$ and $o_4$.

If these two cases are combined, then $\mathsf{PtrMap}(z) = \{(h_{o1} \wedge h_{o2} \vee h_{o3} \wedge h_{o4}, g)\}$ since our analysis is flow-insensitive. In general, each pointed-to location is guarded by a disjunction of conjuncts that encodes the set of heap

objects to be further cloned (if possible) if the points-to location is to be further disambiguated.

**Example 2.** *Figure 7 shows an example when performing guarded points-to propagation across copy edges. Eventually,* `o` *is propagated from* `p`*'s points-to set into q's points-to set guarded by* $(h_1 \wedge h_2) \vee h_3$. *If either* $h_1 \wedge h_2$ *or* $h_3$ *is* true, *then* $(\text{true}, o) \in \mathit{PtrMap}(q)$. *This means that the pointed-to relation is unconditional, as it cannot be changed by how different heap objects are cloned in future iterations.*

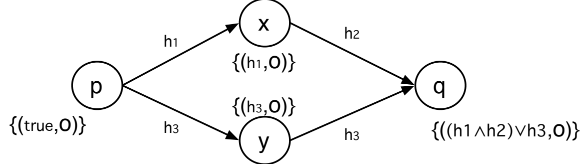



**Figure 7.** Guarded propagation of points-to information.

***Side-Effect Summarization*** First, a lightweight escape analysis is performed for each procedure to determine the objects that may escape into and out of this procedure. Then the side effects of each procedure is summarized, including all its statements that may potentially modify any escaped object in the standard manner as described in [22]. To deal with the callsites in the procedure, their side-effects must be accommodated also, as discussed below.

#### 4.1.2 Interprocedural Analysis

QUDA performs its adaptive heap cloning iteratively across the call graph of a program, first top down and then bottom up, as shown in Algorithm 1. The input is $\mathbb{C}$, the set of heap objects selected in the previous iteration for further cloning in this iteration. $\mathbb{C} = \emptyset$ when $k = 0$. The call graph for a program is the one built with context-sensitive Andersen's analysis by acyclic call paths for method calls without heap cloning. All recursion cycles in the call graph are collapsed into SCCs so that the procedures in an SCC are analyzed context-insensitively. QUDA resembles FULCRA [22] except that their heap cloning strategies are different: the former is adaptive in order to answer alias queries more accurately while the latter is exhaustive.

**Algorithm 1:** INTERPROCEDURALANALYSIS($\mathbb{C}$)

```
1 repeat
2 |  Top-down_Analysis
  |  // Bottom-Up Analysis
3 |  foreach procedure (or SCC) f in reverse topological
  |  order of the program's call graph do
4 |  |  APPLYSUMMARIESANDCLONING(f, C)
5 |  |_ INTRAPROCEDURALANALYSIS(f)
6 until a fixed point is reached;
```

During a top-down phase, pointer resolution is performed on the constraint graph of a program with the side-effects of callsites ignored. During a bottom-up phase, a procedure is analyzed by first applying the side-effect summaries of its callees at their callsites in the procedure *with adaptive heap cloning enabled* (Algorithm 2) and then performing the intraprocedural analysis discussed in Section 4.1.1.

***Heap Cloning Trees (*HCTs*)*** There is one HCT per allocation site, with the root denoting the heap object created at the allocation site. The root objects for different allocation sites are distinct. For each procedure that contains a (direct or indirect) call to an allocation site, a heap object returned by the procedure may be partitioned, i.e., cloned into distinct child nodes at the distinct calling contexts of the procedure.

Given a program, the HCTs currently used at an iteration of our analysis collectively define the heap model currently used for the program at this particular iteration.

**Definition 4** (Location Aliasing)**.** *Two locations (variables or heap nodes) o and* $o'$ *are aliases if* $o = o'$ *or o and* $o'$ *are from the same HCT such that one is an ancestor of the other.*

**Example 3.** *In Figure 5(c), there are two* HCTs *that would be created with full heap cloning for the two allocation sites at* $cs_1$ *and* $cs_4$ *shown in Figure 5(a). For the* HCT *on the left,* $o_1$ *is the root and* $o_2$ *and* $o_3$ *are its clones created at callsites* $cs_2$ *and* $cs_3$*, respectively. For the* HCT *on the right,* $o_4$ *is the root,* $o_5$ *and* $o_6$ *are its clones created at callsites* $cs_5$ *and* $cs_6$*, respectively, and* $o_7$ *is a clone of* $o_5$ *created at* $cs_7$.

***Summary Application and Adaptive Heap Cloning*** A procedure $f$ is analyzed interprocedurally by Algorithm 2. There are two tasks. First, the side-effects of all invoked callees are applied to their callsites in $f$ in the standard manner (lines 1 – 3). Second, every heap object $o$ returned as a (guarded) pointed-to target $(h, o)$ of a variable $p$ at a callee $g$ that is invoked at a callsite $cs$ is cloned into a child node $o'$ in its HCT if $o$ was cloned in an earlier iteration (lines 7 – 8) or inserted into $\mathbb{C}$ in the previous iteration for it to be cloned now at the current iteration (lines 9 – 11). Once $o$ is cloned into $o'$, every occurrence of the old guard $h_o$ in $h$ is replaced by a new one, $h_{o'}$ (lines 12 – 13). Essentially, $h_{o'}$ encodes the proposition that the heap object returned from the callee $g$ that calls (directly or indirectly) the unique allocation site associated with $o$ is now represented as $o'$.

**Example 4.** *Let us return to our example, by considering how* `readArray` *is analyzed at* $k = 1$ *in Figure 5(c) given* $\mathbb{C} = \{o1\}$. *There are two callsites,* $cs_2$ *and* $cs_3$*, to* `getMem`*. When the side-effects of* `getMem` *is applied to the two callsites,* $\mathit{PtrMap}(m) = \mathit{PtrMap}(n) = (h_{o1}, o1)\}$. *As* $o1 \in \mathbb{C}$, $o1$ *is cloned into* $o2$ *and* $o3$ *at* $cs_2$ *and* $cs_3$*, respectively. As a result,* $\mathit{PtrMap}(m) = \{(h_{o2}, o2)\}$ *and* $\mathit{PtrMap}(n) = \{(h_{o3}, o3)\}$. *Before cloning, the* HCT *is the one given in Figure 5(b). After cloning, it becomes the one in Figure 5(c).*

### 4.2 Selecting Candidate Heap Objects for Cloning

Once the points-to information is obtained, a set of alias queries issued by the compiler are processed. A set of heap

**Algorithm 2:** ApplySummariesAndCloning($f$,$\mathbb{C}$)

```
1  foreach callsite cs in f do
2      foreach callee g invoked at cs do
3          Apply the side-effect summaries of g to cs
4  foreach variable p accessed in f s.t. (h, o) ∈ PtrMap(p),
   where o is returned from a callee g at a callsite cs in f do
5      Let T be the HCT that o belongs to
6      if o's clone at (cs, g) is already on T or o ∈ ℂ then
7          if o's clone at (cs, g) is already on T then
8              Let o' be the clone of o at (cs, g) on T
9          else // o ∈ ℂ
10             Let o' be a new clone of o at (cs, g)
11             Add o' to T
12         Let h' be obtained from h with every occurrence of
           h_o being replaced by a new guard h_o'
13         PtrMap(p) ← (PtrMap(p) \ (h, o)) ∪ (h', o')
```

objects is selected for cloning by Algorithm 3 when doing so may cause some queries to be answered more precisely.

For an expression $e$, we write $\mathcal{A}(e)$ for the set of guarded pointed-to locations aliased with $e$. For example, $\mathcal{A}(*p) = \mathsf{PtrMap}(p)$. Two expressions $e$ and $e'$ may alias, denoted $\mathcal{A}(e) \approx \mathcal{A}(e')$, if there exist $(h, a) \in \mathcal{A}(e)$ and $(h', a') \in \mathcal{A}(e')$ such that $a$ and $a'$ are aliased locations by Definition 4.

**Algorithm 3:** CandidateSelection

```
   input  : ℚ: a set of alias queries
   output: ℂ: a set of heap objects for further cloning
1  ℂ = ∅
2  foreach (e, e') ∈ ℚ such that A(e) ≈ A(e') do
3      foreach (h, a) ∈ A(e) and (h', a') ∈ A(e') such that a
       and a' are aliased locations by Definition 4 do
4          if (h ∧ h') ≠ true then
5              foreach heap object o encoded by h ∧ h' do
6                  if o is further cloneable in its own HCT
                   along some acyclic call paths then
7                      ℂ ← ℂ ∪ {o}
```

Algorithm 3 is straightforward. We choose to clone all heap objects encoded by $h$ and $h'$ (greedily) for two reasons. First, each guard is a disjunction of conjuncts (Section 4.1.1). Due to flow-insensitivity, each conjunct must be “cloned”, i.e., refined in order to render its guarded location spurious. Second, 90% of the guards in the 15 benchmarks used on average encode one or two heap objects each.

**Example 5.** *Given the points-to information in Fgiure 5(b) and $\mathbb{Q} = \{(*m, *n), (*p, *q)\}$. By Algorithm 3, we find that $\mathbb{C} = \{o1\}$. So $o1$ is selected to be cloned as in Fgiure 5(c).*

### 4.3 Adaptive Update

The constraint graph of a program is updated by Algorithm 4. First of all, a points-to relation is removed if some heap objects encoded by its associated guard are to be further cloned. In addition, a copy edge is also removed if its guard encodes some heap objects that are to be cloned.

**Algorithm 4:** AdaptiveUpdate

```
   input  : ℂ: a set of heap objects for further cloning
   output: Constraint graph updated
1  foreach o ∈ ℂ do
2      foreach (h, a) ∈ PtrMap(p) s.t. o is encoded by h do
3          PtrMap(p) ← PtrMap(p) \ {(h, a)}
4      foreach copy edge p ←h→ q s.t. o is encoded by h do
5          Remove it from the constraint graph
```

In our implementation, a data structure is maintained so that each $o$ in line 2 (line 4) is directly linked to the points-to relations (copy edges) processed in line 3 (line 5).

**Example 6.** *When analyzing `readArray` by moving from Figure 5(b) to Figure 5(c), with $\mathbb{C} = \{o1\}$, `o1` is removed from the points-to sets of `m` and `n` and `x` from the points-to sets of `q` and `o1`. In addition, the two copy edges guarded by $h_{o1}$ are also removed. This gives rise to the constraint graph shown in the middle box given in Figure 5(b).*

## 5. Evaluation

The objective is to show that our query-directed heap cloning approach is significant faster than full heap cloning subject to some stopping condition (by about an order of magnititude) while achieving the same precision in answering the same alias queries issued by the compiler.

We have selected 15 C programs (totalling 840 KLOC), with 10 being the largest SPEC CPU2000 benchmarks and 5 being open-source applications: `hmmer-2.3` (gene sequence search), `icecast-2.3.1` (a streaming media server), `jpeg2000` (image compression), `rasta` (speech analysis), and `sendmail-8.14.2` (an internet email server). Their characteristics are given in Columns 2 – 5 in Table 2.

Our platform is a 2.8GHz quad-core Intel Xeon running Redhat Enterprise Linux 5 (2.6.18) with 8GB memory.

### 5.1 Methodology

We compare QUDA with Fulcra [22], a state-of-the-art Andersen's analysis with full heap cloning. QUDA and Fulcra differ only in terms of their heap cloning algorithms used. Both analyses are context-sensitive by acyclic call paths for method calls (i.e., local variables). The call graph for a program is the one built using Andersen's analysis that is context-sensitive by acyclic call paths for method calls but without heap cloning. All recursion cycles in a program are merged into distinct SCCs so that the procedures inside are analyzed context-insensitively. There is one common stopping condition for heap cloning in both cases. As soon as the number of side-effect-causing statements promoted (i.e., transferred) from an SCC to its a caller exceeds

| Program | Program Characteristics | | | | QUDA | | | | Analysis Times (secs) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | KLOC | #Procs | #Pointers | #Callsites | #Queries | #Iters | #Nodes | #Edges | FULCRA | QUDA | Speedup |
| ammp | 13.4 | 182 | 9829 | 1201 | 38893 | 2 | 11690 | 10267 | 0.38 | 0.38 | 1.00 |
| crafty | 21.2 | 112 | 11883 | 4046 | 15545 | 3 | 13725 | 10887 | 0.84 | 0.15 | 5.60 |
| gap | 71.5 | 857 | 61435 | 5980 | 101187 | 2 | 61856 | 70719 | 22.20 | 22.20 | 1.00 |
| gcc | 230.4 | 2256 | 134380 | 22353 | 128713 | 6 | 135785 | 153220 | 1321.33 | 92.78 | 14.24 |
| hmmer | 48.1 | 3868 | 645 | 2446 | 645 | 7 | 55285 | 63197 | 128.52 | 11.65 | 11.03 |
| icecast | 22.3 | 15098 | 603 | 468 | 7774 | 2 | 24085 | 24331 | 28.30 | 2.23 | 12.69 |
| jpeg | 26.4 | 377 | 19623 | 1177 | 8978 | 3 | 26884 | 43845 | 108.90 | 7.31 | 14.90 |
| mesa | 61.3 | 1109 | 44582 | 3611 | 144328 | 5 | 101228 | 141379 | 651.08 | 93.73 | 6.95 |
| parser | 11.4 | 327 | 8228 | 1782 | 9529 | 2 | 9155 | 8560 | 0.96 | 0.96 | 1.00 |
| perlbmk | 87.1 | 1079 | 54816 | 8470 | 98090 | 4 | 59341 | 64127 | 117.17 | 11.06 | 10.59 |
| rasta | 26.9 | 299 | 33387 | 2782 | 10892 | 4 | 33383 | 42963 | 30.10 | 2.83 | 10.64 |
| sendmail | 115.2 | 107242 | 2656 | 16973 | 144398 | 3 | 42887 | 46934 | 85.60 | 5.23 | 16.37 |
| twolf | 20.5 | 194 | 20773 | 2074 | 92917 | 2 | 24313 | 31672 | 2.56 | 1.20 | 2.13 |
| vortex | 67.3 | 926 | 40260 | 8522 | 81542 | 5 | 49678 | 50391 | 78.65 | 7.21 | 10.91 |
| vpr | 17.8 | 275 | 7930 | 1995 | 23557 | 4 | 9914 | 10902 | 1.32 | 0.10 | 13.20 |
| total | 840.8 | 134201 | 451030 | 83880 | 906988 | | 659209 | 773394 | 2577.91 | 259.02 | |

**Table 2.** Program statistics and experimental results.

50, either analysis becomes context-insensitive for the SCC. This is the same limit imposed in the original FULCRA implementation released in the Trimaran compiler (4.0).

#### 5.1.1 Implementation

We have implemented QUDA in the Open64 compiler (5.0), at its IPA (interprocedural analysis) phase. IPA performs global analysis by combing information from its IPL (Local part of its Interprocedural Phase, which collects summary information local to a procedure). We have implemented FULCRA by following strictly its algorithms [22].

All the analyses are offset-based field-sensitive. However, arrays are considered monolithic. All heap wrappers are identified and treated as allocation sites.

Wave Propagation [24] is used to perform constraint resolution. Global variables are handled context-insensitively in the standard manner as in [20, 22]: they are separately tracked without participating in side-effect summarization.

#### 5.1.2 Experimental Setting

We focus on the alias queries generated in IPA (global analysis) and WOPT (backend optimizations) for machine-independent optimizations. We did not include those issued in CG (code generation) for two reasons. First, the precentage of queries from CG is small ($< 5\%$ across our programs tested). Second, once adaptive heap cloning for IPA and WOPT is completed, the alias information is expected to be precise enough for CG without a need for further iterations.

We use the *Alias Tag* mechanism provided internally in Open64 to answer alias queries. Each *Alias Tag*, attached to an expression, represents the memory locations accessed. Only the queries that needs pointer information are issued.

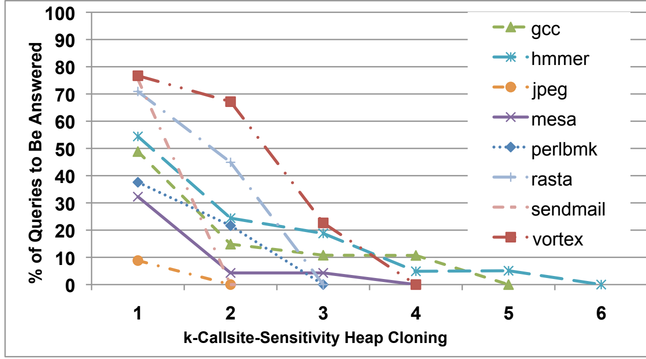
Let us explain how the analysis times are calculated for the two analyses. FULCRA performs its pointer analysis before it is used in IPA and WOPT. So its analysis time is simply how long it takes to complete its analysis.

QUDA performs its $k$-callsite-sensitive heap cloning analysis iteratively, starting from $k = 0$. During each iteration, the pointer analysis is first performed and then the alias queries are issued. The queries issued in different iterations may differ due to different optimizations enabled. Ideally, the analysis results for IPA and WOPT in all iterations should be kept in memory. However, in Open64, IPA and WOPT are two separate modules, with WOPT invoked after IPA. During each iteration, we use the SUMMARY mechniasm provided internally in Open64 to write the points-to information obtained in IPA into auxilairy ELF files (*.I and *.G) so that it can be fed into WOPT. The results from the alias queries and the constraint graph between two adjacent iterations are all communicated using ELF files, which consumes little time (less than 0.1 secs for each program). Finally, the time that QUDA takes to analyze a program requiring $k$ iterations to complete is measured to be $(k-1)\times$ (the time that Open64 spends on its optimizations in IPA and WOPT) + (QUDA's analysis time).

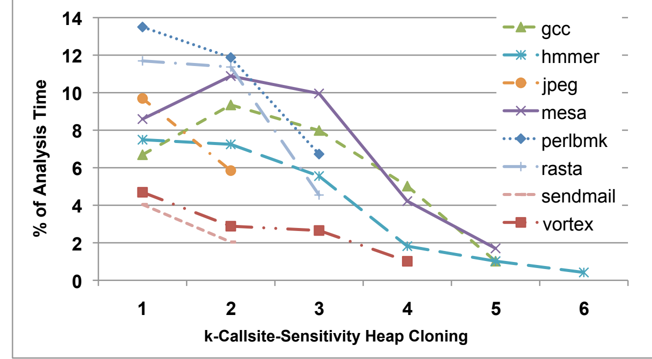
### 5.2 Results and Analysis

We first compare QUDA with FULCRA to demonstrate the performance improvements achieved by adaptive heap cloning. We then analyze our benchmarks to provide some insights on understanding these results.

***Comparing with*** FULCRA  As shown in the last three columns in Table 2, QUDA is 8.82 times faster on average than FULCRA while achieving exactly the same precision by design in terms of the alias queries issued by Open64 (Column 6 in Table 2). To analyze all the 15 programs, QUDA spends only 259.02 seconds (4+ minutes) while FULCRA spends a total of 2577.91 seconds (42+ minutes).




(a) Alias queries to be answered at each iteration




(b) Analysis time per iteration over the total

**Figure 10.** The alias queries and analysis times across different iterations of QUDA.

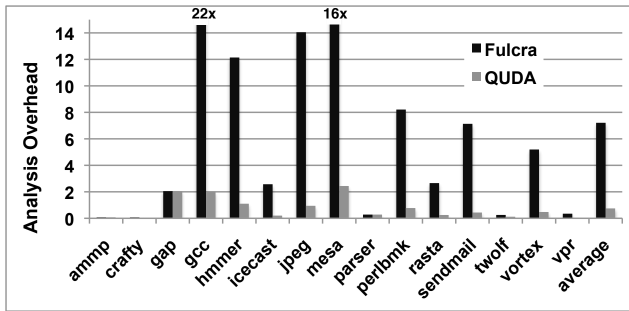



**Figure 8.** Analysis times of FULCRA and QUDA normalized with respect to Open64's compile times under "-O2".

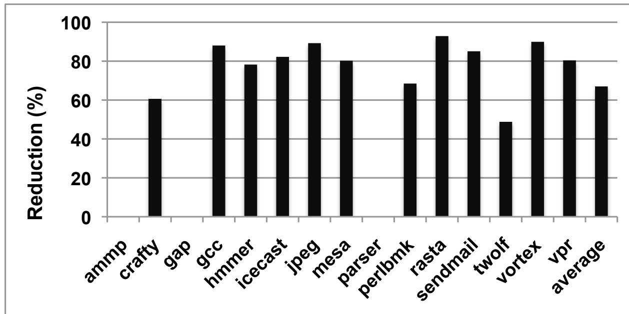



**Figure 9.** Number of heap objects reduced by QUDA over FULCRA in percentage terms.

Figure 8 compares QUDA and FULCRA in terms of their analysis times relative to Open64's compile time for a benchmark. On average, QUDA takes only 75.1% of the time that Open64 spends on compiling the 15 benchmarks but FULCRA takes 721.0%. This means that exhaustive heap cloning is not feasible for most of the programs. In contrast, query-directed adaptive heap cloning is very promising. For programs, such as `gcc`, `hmmer`, `jpeg`, `mesa` and `perlbmk`, that take Open64 relatively long times to compile, QUDA's analysis times are comparable. For the others, QUDA's analysis times are small in absolute terms.

Let us take a look at Table 2 again. The analysis for most programs converges in 1 – 4 iterations, except for `gcc`, `hmmer`, `mesa` and `vortex` (Column 7). For the programs achieving good speedups, their performance results are consistent with the percentage reductions achieved for heap objects created as shown in Figure 9. For `ammp`, `gap`, and `parser` that are not heap-intensive, QUDA is not effective (with no speedup over FULCRA in each case). In `ammp`, the objects allocated are used either locally or globally. The other two benchmarks have few allocation sites. In `gap`, its loads/stores are mostly on global data structures. In `parser`, a large pool is allocated and used throughout.

***Understanding* QUDA** We analyze the eight representative benchmarks in Figures 10 and 11. The remaining ones are either too small (`crafty` and `vpr`), or not heap-intensive (`ammp`, `gap` and `parser`) or able to finish all the issued queries in two itrerations (`ammp`, `icecast` and `twolf`).

Figure 10 gives the percentage of queries answered (starting from $k = 1$) and the percentage of analysis time elapsed in an iteration. An alias query for two expressions $e$ and $e'$, $\mathsf{MayAlias}(e, e')$, is considered as being answered if (1) $e$ and $e'$ are must-not aliases or (2) $e$ and $e'$ may alias with a guarded location but the heap objects encoded by its guards cannot be cloned further. Note that if $e$ and $e'$ unconditionally alias with a common location (which must be non-heap) at $k = 0$, then the alias cannot be disambiguated at any later iteration. Thus, all such may aliases, which cannot be improved with heap cloning, are not included in our statistics.

From Figure 10(a), we see that the analysis for most of the benchmarks finishes in up to 5 iterations, except for `gcc` and `hmmer`. With $k = 1$, more than half of the alias queries can be answered in `gcc`, `jpeg`, `mesa` and `perlbmk` (as in `ammp`, `icecast` and `twolf` not shown here). However, the same does not happen for the other four benchmarks, `hmmer`, `rasta`, `sendmail` and `vortex`. In the case of `sendmail`,

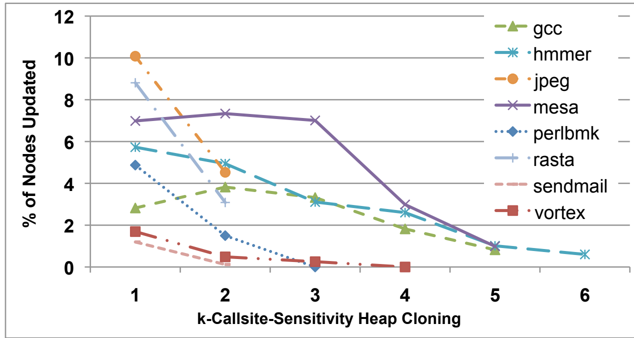



**Figure 11.** Percentage of nodes updated (with its points-to relations or incident edges changed) in constraint graphs.

almost all queries can be answered with $k = 2$. For the other three, some more iterations are needed in order to distinguish more heap objects created along different acyclic call paths.

Let us move to Figure 10(b) to examine the distribution of a program's analysis time across different iterations. $k = 0$ (without heap cloning), which is not shown but implied, is the most expensive. Each subsequent one is much faster ($<$ 15% of the total), because only small parts in the constraint graph of a program are updated, as observed in Figure 11.

As $k$ increases, the analysis time per iteration for a program tends to decrease. There are two reasons behind. First, fewer and fewer queries need to be answered, as shown in Figure 10(a). Second, less and less heap cloning is performed, resulting in fewer and fewer changes to the constraint graph of the program, as shown in Figure 11. However, there are some exceptions as in `gcc` and `mesa`, which cost more to analyze when $k = 2$ and $k = 3$ than $k = 1$. This is due to a large number of heap objects cloned when $k = 2$ and $k = 3$, causing relatively large parts of their constraint graphs to be updated, as shown in Figure 11.

Finally, `vortex` has a large number of queries to answer when $k = 2$ and $k = 3$ but the times elapsed in these two iterations are relatively small. This is because the changes made to its constraint graph are small as shown in Figure 11. A similar phenomenon occurs for `sendmail`.

## 6. Related Work

There are many prior studies on context-sensitive pointer analysis. Different types of precision are distinguished if calling contexts are identified by full call paths [13, 37], assumed aliases at callsites [5, 14], acyclic call paths (with recursion cycles collapsed into SCCs) [13, 37] and approximated call paths within recursion cycles [8, 25, 33, 34].

In most of these earlier efforts, the heap is modeled only 0-callsite-sensitively (without heap cloning). There are a few attempts considering context-sensitive heap cloning in whole-program analysis [7, 16, 22, 33, 35, 36].

DSA [16] represents a Steensgarrd's unification-based pointer analysis with full heap cloning. The authors showed that full heap cloning can greatly recover the precision loss due to pointer unificaiton. Compared to Steensgarrd's analysis, Andersen's analysis is the most precise of all flow- and context-insensitive pointer analyses. In [22], the authors introduced FULCRA as a context-sensitive Andersen's analysis by promoting the side-effect-causing statements from a calee to its callsites. Earlier [23], they showed empirically on top of their analysis that full heap cloning can improve the analysis precision, but can also incur uncontrollable overhead. In this paper, QUDA makes full heap cloning significantly more scalably for Andersen's analysis by being query-quided. As compared in Figure 1, QUDA can be more precise than DSA for some programs.

Some analyses incorporate $k$-callsite-sensitive heap cloning [22, 35, 36]. As discussed in Section 1, different programs require different values of $k$ to make the best tradeoffs between precision and scalablity. The one-size-for-all solution with a fixed $k$ for all programs is not an acceptable solution. In addition, it is difficult to tune each program manually to find the best $k$ for two reasons. First, this process can be costly as it involves reanalyzing a program many times. Second, the best $k$ may be elusive as the precision improvement may be unchanged for some consecutive values of $k$ and suddenly improves at the next value (Figure 10(a)). In contrast, QUDA achieves the precision of exhaustive heap cloning iteratively, guided by the queries issued, and is fast since only small parts of a constraint graph are updated.

In [9], a client-driven pointer analysis is introduced that adjusts its precision in terms of flow and context sensitivity according to the needs of client analyses. However, they did not address heap sensititivity as we do in this work.

In the demand-driven alias/pointer analysis based on context-free language reachability [21, 27–29], full heap cloning is naturally supported. in future research, whether it scales when performed as a whole-program analysis for large programs remains to be further investigated [36].

Shape analysis [3, 4, 26] reasons about pointer-related properties concerning recursive data structures. The state-of-the-art techniques do not scale for large programs.

Recently, sparse pointer analysis [11, 12, 19, 31, 32, 37] discovers pointer information more quickly across def-use chains rather than iteratively in a data-flow framework. Our adaptive heap cloning complements this existing research.

Binary Decision Diagrams (BDDs) have been used to reduce the time for handling the exponential growth of calling paths [2, 33, 39]. Procedure-cloning-based algorithms are not fast enough for compilers in analyzing large programs. In [7], strong updates are exploited in a flow-sensitive analysis with full heap cloning for program verification.

## 7. Conclusion

In this paper, we make one step forward by solving a challenging problem in making Andersen's pointer analysis with full heap cloning more scalable for optimizing compilers.

By using the alias queries issued from the compiler to guide heap cloning, our analysis achieves the precision of full heap cloning while being significantly more scalable.

This work has opened up some new research opportunities. As an example, our adaptive approach allows heap cloning efforts to be focussed on improving the precision of the queries ranked in order of their importance (e.g., in hot vs. non-hot procedures). As another example, iterative optimization finds the best optimization sequence through repeated runs on data sets with heuristics [1, 6, 30]. Our approach may help their techniques converge more quickly.

## Acknowledgments

Thanks to the reviewers for their helpful comments on the work. This research project is supported by Australian Research Grants, DP0987236 and DP130101970.

## References

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, and C. Williams. Using machine learning to focus iterative optimization. In *CGO '06*, pages 295–305.

[2] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. *PLDI '03*, pages 103–114.

[3] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *POPL '09*, volume 44, pages 289–300.

[4] B. Chang and X. Rival. Relational inductive shape analysis. In *POPL '08*, volume 43, pages 247–260.

[5] R. Chatterjee, B. Ryder, and W. Landi. Relevant context inference. In *POPL '99*, pages 133–146.

[6] Y. Chen, Y. Huang, L. Eeckhout, L. Peng, G. Fursin, O. Temam, and C. Wu. Evaluating iterative optimization across 1000 data sets. In *PLDI '10*, pages 448–459.

[7] I. Dillig, T. Dillig, A. Aiken, and M. Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In *PLDI '11*, pages 567–577.

[8] R. Emami, M. Ghiya and J. Hendren. Context-sensitive interprocedural points-to analysis in presence of function pointers. In *PLDI '94*, pages 242–256.

[9] S. Z. Guyer and C. Lin. Client-driven pointer analysis. In *SAS'03*, pages 1073–1073.

[10] B. Hardekopf and C. Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *PLDI '07*, pages 290–299, .

[11] B. Hardekopf and C. Lin. Semi-sparse flow-sensitive pointer analysis. In *POPL '09*, pages 226–238, .

[12] B. Hardekopf and C. Lin. Flow-Sensitive Pointer Analysis for Millions of Lines of Code. In *CGO '11*, pages 289–298, .

[13] V. Kahlon. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In *PLDI '08*, pages 249–259.

[14] W. Landi and B. Ryder. A safe approximate algorithm for interprocedural aliasing. *PLDI '92*, 27(7).

[15] C. Lattner and V. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *PLDI '05*, pages 129–142.

[16] C. Lattner, A. Lenharth, and V. Adve. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World. In *PLDI'07*, pages 278–289.

[17] O. lhoták. Context-sensitive points-to analysis: is it worth it? In *CC '06*, pages 47–64.

[18] O. Lhoták and L. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. *TOSEM '08*, 18(1), 2008.

[19] L. Li, C. Cifuentes, and N. Keynes. Boosting the performance of flow-sensitive points-to analysis using value flow. In *FSE '11*, pages 343–353.

[20] D. Liang and M. Harrold. Efficient computation of parameterized pointer information for interprocedural analyses. *SAS '01*, pages 279–298.

[21] Y. Lu, L. Shang, X. Xie, and J. Xue. An incremental points-to analysis with CFL-reachability. In *CC'13*.

[22] E. Nystrom. *FULCRA pointer analysis framework*. PhD thesis, University of Illinois at Urbana-Champaign, 2006.

[23] E. Nystrom, H. Kim, and W. Hwu. Importance of heap specialization in pointer analysis. In *PASTE '04*, pages 43–48.

[24] F. Pereira and D. Berlin. Wave propagation and deep propagation for pointer analysis. In *CGO '09*, pages 126–135.

[25] R. Rupesh Nasre. Prioritizing constraint evaluation for efficient points-to analysis. In *CGO '11*, pages 267–276.

[26] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS '02*, 24(3):217–298.

[27] L. Shang, Y. Lu, and J. Xue. Fast and precise points-to analysis with incremental CFL-reachability summarisation: preliminary experience. In *ASE '12*, pages 270–273, .

[28] L. Shang, X. Xie, and J. Xue. On-demand dynamic summary-based points-to analysis. In *CGO '12*, pages 264–274, .

[29] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for java. *PLDI '06*, pages 387–400.

[30] M. Stephenson, S. Amarasinghe, M. Martin, and U. O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. *PLDI '03*, pages 77–90.

[31] Y. Sui, S. Ye, J. Xue, and P. Yew. SPAS: Scalable path-sensitive pointer analysis on full-sparse SSA. In *APLAS '11*, pages 155–171.

[32] Y. Sui, D. Ye, and J. Xue. Static memory leak detection using full-sparse value-flow analysis. In *ISSTA '12*, pages 254–264, 2012.

[33] J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04*, pages 131–144.

[34] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. *PLDI '95*, pages 1–12.

[35] X. Xiao and C. Zhang. Geometric encoding: forging the high performance context sensitive points-to analysis for Java. In *ISSTA '11*, pages 188–198.

[36] G. Xu and A. Rountev. Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis. In *ISSTA '08*, pages 225–236.

[37] H. Yu, J. Xue, W. Huo, X. Feng, and Z. Zhang. Level by level: making flow-and context-sensitive pointer analysis scalable for millions of lines of code. In *CGO '10*, pages 218–229.

[38] X. Zheng and R. Rugina. Demand-driven alias analysis for C. In *POPL '08*, pages 197–208.

[39] J. Zhu and S. Calman. Symbolic pointer analysis revisited. In *PLDI '04*, pages 145–157.