RC24995 (W1002-021) February 8, 2010
Computer Science

IBM Research Report

Increasing Accuracy of Pointer Analysis to Find Concurrency Bugs

Daniel Brand
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Marcio Buss
Department of Computer Science
Columbia University
New York, NY

Vugranam C. Sreedhar
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Increasing Accuracy of Pointer Analysis to Find Concurrency Bugs

Daniel Brand
IBM Research
Yorktown Heights, NY, USA
danbrand@us.ibm.com

Marcio Buss
Dept. of Computer Science
Columbia University
New York, NY, USA
marcio@cs.columbia.edu

Vugranam C. Sreedhar
IBM Research
Hawthorne, NY, USA
vugranam@us.ibm.com

ABSTRACT

We address the problem of statically finding bugs due to concurrency, and do so without reporting false positives. This problem is important because false reports of concurrent bugs are even less acceptable than those of sequential bugs. (Typically programmers need to examine larger amount of code in diagnosing the report of a concurrent bug than sequential one.) Avoiding false positives for concurrent bugs is more difficult, because techniques like symbolic execution, used for avoiding false reports of sequential bugs, become prohibitively expensive for parallel programs. Our proposed solution is pointer analysis accurate enough to report concurrent bugs without too many false positives. For our implementation the trade-off between accuracy and efficiency is driven by the needs of an algorithm for deadlock detection.

1. INTRODUCTION

It is often the case that static analysis needs to make trade-offs among cost of the analysis, accuracy or precision of the analysis, and the scope of program that is being analyzed. Static analysis of multi-threaded programs is more challenging than sequential ones for several reasons. First interleaving of threads greatly increases the number of execution paths. Second the presence synchronization, such as locks and unlocks, can impact how program paths are abstracted. Third detecting concurrent bugs on the average requires larger scope than detecting sequential bugs.

For the past few years we have been developing a static analysis tool called BEAM [1] for detecting software bugs. It accepts source code in C, C++, Java; throughout the paper our examples are all in the C language, using the pthread synchronization primitives. We will be using the terms "mutex" and "lock" interchangeably.

The BEAM tool strives for maximal accuracy so as to report many true positives with few false positives. For that reason we rely on symbolic execution in the case of sequential programs. For multi-threaded programs, however, unrestricted symbolic execution can quickly explode the set of paths that needs to be analyzed due to the presence of multiple threads that need to be interleaved. One approach to this problem is to precalculate sequences of instructions within which interleaving with other threads can be ignored [6, 7, 16, 10]. A different approach [8] uses imprecise analysis, generating many false positives, and then uses heuristics to rank the findings according to likely validity.

```
callee(data *p, data *q)
{
  pthread_mutex_lock(p->m);
  pthread_mutex_lock(q->m);
  ...
```

```
caller()
{
  callee(x, y);
  ...
  callee(y, x);
```

Figure 1: Is there a deadlock in the callee?

In this paper we address the problem of path explosion by relying on a more accurate pointer analysis, not on symbolic execution. Our approach collects information from each thread in isolation, and then combines the results to detect concurrency bugs across multiple threads of control. This is similar in spirit to [12].

This choice of pre-analyzing threads in isolation is related to the more general question of what is a bug. For example, consider Figure 1. In the whole program approach [9, 14], or the top-down approach [8], errors are normally associated with the actual failing statement; in Figure 1 that would be the call `pthread_mutex_lock(q->m)`, where execution might stall. In contrast, we follow the evidence-based approach [2], which declares the callee deadlock-free, but finds the `caller` in error. The reason is that two threads executing the `caller` could invoke the callee with arguments in reverse order. This approach is amenable to library analysis; in particular, we will present an algorithm that collects information about libraries for detecting deadlocks in clients using the libraries.

In any case, the chosen philosophy for what is a bug needs to be matched with pointer analysis. The whole program view requires whole program pointer analysis [13, 18]. That has to identify possible aliases of `p` and `q` in the `callee()` of Figure 1, and such relationship is established by callers. In contrast, the evidence based approach requires pointer

[0]The second author's collaboration was done while he was with Columbia University.

analysis that propagates information from callees to callers only [4].

The main focus of this paper is on fine-tuning pointer analysis for detecting concurrency bugs, rather than on new ways to detect such bugs. In the rest of this section we motivate the need for accuracy of the pointer analysis. In Section 2 we describe our approach to detecting deadlocks, and then the rest of the paper describes pointer analysis accurate enough for the task. Section 3 outlines the space of pointer analyses from which we chose a particular analysis described in Section 4. That choice is implemented in our tool and is used for other purposes as well – MOD, alias, escape analyses; but we found none of them as demanding in accuracy as detection of concurrent bugs.

Mutex Ordering Problem The example of Figure 2, illustrates two things: Lock variables are part of a structure, which necessitates field sensitive analysis. Secondly, it is important to respect the order of lock/unlock operations, which calls for some form of flow-sensitivity in the analysis. We will show pointer analysis that handles these two problems without the expense of a full-blown flow-sensitive analysis.

```
pthread_mutex_lock(a->m);
/* ... real work 1 ... */
if ( c) pthread_mutex_unlock(a->m);
/* ... real work 2 ... */
if (!c) pthread_mutex_unlock(a->m);
```

Figure 2: Mutex unlocked under several conditions

Conditional Mutex Problem A common programming pattern arises when a locked mutex is unlocked under different conditions, see Figure 2. In such cases, when tracking the relative ordering of lock variables we also need to take conditions into account. It is important to ensure that, when a mutex is unlocked under different conditions, such conditions are mutually exclusive and exhaustive.

Mutex Across Procedures Problem It is often the case that a mutex can be locked in one procedure and can get conditionally unlocked before returning. This would happen in Figure 2 if the last line were missing. Such situations typically happens when a procedure attempts to acquire a lock, but under some exceptional circumstances it fails and in that case releases the lock. This shows the need to propagate the conditions of lock acquisition inter-procedurally, and often also the need to relate the condition to the values of a return code.

To summarize the main contributions of this paper are:
(1) A simple abstraction to model relationship among lock variables and using the abstraction as the basis for detecting deadlocks. The resulting algorithm is suitable for library analysis.
(2) An efficient representation and manipulation of program conditions.
(3) Identifying a pointer analysis with a balance between accuracy and run-time performance, which is suitable for reporting concurrency bugs.
(4) Preliminary empirical results to validate the choice of analysis (3). While we did not find any deadlocks in any production code, the analysis is accurate enough to avoid false positives, and the accuracy comes with minimal loss of run-time efficiency.

2. DEADLOCK ALGORITHM

As an example of a concurrent bug whose discovery requires accurate pointer analysis we will discuss deadlock detection. There is a relationship between our approach and the collection of *locksets*, which can be dynamic [17] or static [8]. Those algorithms traverse a program in a forward fashion (caller to callee), recording locks held. The forward traversal facilitates disambiguation as to which global lock is pointed to by a variable at any point of time. We are solving the converse problem, namely library analysis. For each procedure we calculate its summary, including its effect on a global *locklist*. The entries on the locklist are not global lock addresses, but rather symbolic contents of global variables or procedure parameters, independently of how they may be set by a caller. We collect a list of locks instead of a set of locks, because we use the list do derive a lock hierarchy, namely the order in which locks are acquired. The calculated summary can then be plugged into callers to calculate their summaries.

There are two complications influencing our algorithm. First locks could be recursive, so not every cycle in a lock hierarchy indicates a deadlock. Secondly, in the C language locks can be released in an order unrelated to the order of acquiring them. Our approach is a static version of the following dynamic algorithm to deal with those complications.

The dynamic algorithm observes lock operations and keeps a list of acquired locks for each thread. The list may contain several instances of the same recursive lock. Whenever a lock is acquired, an instance of the lock is appended to the list. Whenever a lock is released the last instance of that lock is removed from the list.

To be specific, we implement lock instances as heap nodes allocated whenever a lock is acquired; they are placed into a doubly linked list (for easy removal). They are never discarded. To form a lock hierarchy, each new lock instance gets a *hierarchy* edge to its predecessor on the list; hierarchy edges remain even after a lock is released. Thus for any procedure, at the end of its execution we get a set of lock instances with hierarchy edges indicating which instances were acquired while still holding others.

Note that all these manipulations are performed in a single thread. The resulting hierarchy graph can be used for detecting recursive acquisition of non-recursive locks. To find deadlocks involving multiple threads we go to the second step, which is independent of the method used to generate the hierarchy among lock instances.

We form hierarchy edges between lock variables themselves. There is a hierarchy edge induced from lock variable x to y iff x and y each has an instance with a hierarchy edge between them. See Figure 3 with four lock variables a, b, c, d and their instances $a_1, a_2, b_1, c_1, c_3, d_2, d_3$. Solid arrows are hierarchy edges, a dotted line for each instance shows to which lock it belongs. The hierarchy chain $a \rightarrow b \rightarrow c$ is

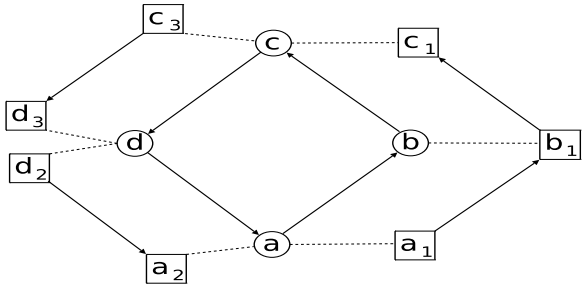

Figure 3: Hierarchy among lock instances (square nodes) inducing hierarchy among locks (round nodes)

due to $a_1 \rightarrow b_1 \rightarrow c_1$, which is within a single thread. If there is no hierarchy path from a_1 to c_3 then the chain $c_3 \rightarrow d_3$ could be made in a separate thread. If all three chains $a_1 \rightarrow b_1 \rightarrow c_1$, $d_2 \rightarrow a_2$, $c_3 \rightarrow d_3$ are pairwise unreachable, then we could have a deadlock in case the given procedure were executed concurrently by three threads. Figure 4 is an actual message issued by our tool explaining the deadlock. The explanation is a sequence of 6 attempts of acquiring a lock by three threads. After each of the 6 lines there is a chain of calls leading to the lock operation. (In our example each chain has only 1 call.)

```
Thread 1 acquires ‘c’ because
  "f.c", line 18: calling ‘pthread_mutex_lock’

Thread 2 acquires ‘a’ because
  "f.c", line 14: calling ‘pthread_mutex_lock’

Thread 3 acquires ‘d’ because
  "f.c", line 10: calling ‘pthread_mutex_lock’

Thread 1 blocks acquiring ‘a’ because
  "f.c", line 20: calling ‘pthread_mutex_lock’

Thread 2 blocks acquiring ‘d’ because
  "f.c", line 15: calling ‘pthread_mutex_lock’

Thread 3 blocks acquiring ‘c’ because
  "f.c", line 11: calling ‘pthread_mutex_lock’
```

Figure 4: Message issued for Figure 3

Up to now we have described the algorithm as dynamic, and it could be such. But, as any dynamic approach, it would be limited to those executions tried. Instead we perform the algorithm statically. It traverses a call graph bottom up; loops in the call graph imply iteration till convergence. The result on any given procedure will include the hierarchy graph of lock instances. This graph is propagated to callers when forming their lock hierarchy. In addition, the lock hierarchy for each procedure can be used for deadlock detection exactly the same way as if it were obtained dynamically.

This kind of approach to deadlock detection, or concurrent bugs in general, has several implications. First of all, the bottom up approach implies that information computed cannot be in terms of global locks, but rather in terms of local pointers to unknown locks. This implies the need for pointer analysis, in contrast to the top-down approach of [8].

The pointer analysis itself need not be aware of threads. The reason is that it is used to accumulate the hierarchy of lock instances in a single thread. (The second step of inter-thread analysis does not require pointer analysis.) Therefore we could use any pointer analysis for sequential programs, provided it is suitable for libraries. In the next section we will outline such a general pointer analysis.

The pointer analysis will be used for the usual purposes of alias, MOD, escape analysis. In addition, lock operations will be translated directly into corresponding transformations of the locklist, as if the locklist were an explicit data-structure in the program. The procedure summary built by pointer analysis for each procedure will then contain not only updates to explicitly declared variables, but also updates to the locklist, and hence the lock hierarchy.

3. POINTER ANALYSIS SPACE

It is well known [11] that the accuracy of pointer analysis has tremendous impact on “client analysis”, whether it is for optimizing code or detecting bugs or re-factoring code. One needs a balance between the precision of pointer analysis and the cost of doing it, and this balancing should take into consideration the needs of the client analysis. In our previous work [4, 5] we defined a space of pointer analyses suitable for such balancing. In this section we leave aside deadlock detection to briefly review those aspect of the space needed to explain our choice of the next Section – choice of pointer analysis driven by the needs of the client of Section 2.

3.1 Dimensions of Pointer Analysis

Our pointer analysis [4] uses a data structure called *Assign-Fetch Graph (AFG)* for representing pointer/alias information. The input into pointer analysis is set a of flow graphs obtained from a given program. Our flow graph looks like a traditional control-flow graph with the following important properties.
1) Besides control flow the graph also contains all the data operations.
2) All loops are converted into tail-recursive procedures. This results in a separate flow graph for each loop as well as each procedure. Each flow graph is acyclic, and the statements are in a partial order.
3) All memory accesses are broken down into two primitives: assign and fetch.
4) For each operation there is a condition for executing it, represented by an edge of the flow graph.

The result of pointer analysis is an AFG that summarizes pointer information for each flow graph. An AFG for a procedure contains all of the side-effects that the procedure could have on a caller, and so one can obtain pointer relations among locations visible to a caller. An interesting aspect of AFG is that it can model various kinds of summary information in a unified manner depending on a desired trade-off between accuracy, run-time complexity, and scope. To capture the right level of accuracy, in our previous work we identified three orthogonal dimensions: statement ordering, conditions, and strong updates as shown in Fig-

ure 6. They are illustrated using the code fragment of Figure 5. The objective of the analysis is to determine under what conditions will `y` contain which value.
Analyses that ignore statement ordering (e.g., flow insensitive analysis) would obtain the same result even if the three statements were rearranged.
Analyses that ignore conditions would obtain the same result even if the conditions in Figure 5 changed. Analyses that do not ignore conditions might obtain that `y` is assigned 0 under condition $c \wedge e$ and it is assigned 1 under condition $d \wedge e$.
The above is an example of a result obtained by ignoring strong updates. Analyses that do consider strong updates, in addition to statement ordering and conditions, would obtain the result that `y` is assigned 0 under condition $c \wedge \neg d \wedge e$ and it is assigned 1 under condition $d \wedge e$.

```
if (c) x = 0;
if (d) x = 1;
if (e) y = x;
```

Figure 5: One assignment conditionally kills another

When designing a pointer analysis, there is no need to chose extreme points along any of the three dimensions; intermediate points are the key to achieving the right trade-off. Therefore the space of possible analyses can be represented as a 3-dimensional cube as in Figure 6. There is no numeric scale associated with any of the axes; they only represent qualitative comparison. For sake of illustration, Figure 6 also shows the position of three commonly used analyses – flow-insensitive, flow-sensitive, and path-sensitive. Note that the flow-sensitive point is not at the bottom of the cube. This is because flow-sensitive analysis does not completely ignore conditions – it considers the two branches of an if-statement to be mutually exclusive. The corner of the cube just below the flow-sensitive analysis represents analysis that would treat conditional branches no differently than branches of a parallel construct.

As one moves away from the origin (flow-insensitive analysis) accuracy increases, but not necessarily run-time. In particular, it was shown in [4] that moving along the statement-ordering axis increases accuracy while decreasing run-time. Therefore there is no need to consider analyses outside the bold lined side in Figure 6. In the next section we will describe the point chosen for our tool, marked in Figure 6 as "BEAM analysis". It is more accurate than flow-sensitive in

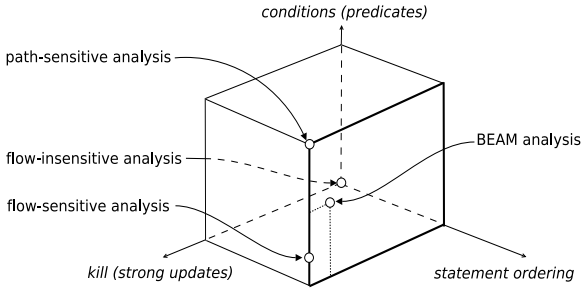

Figure 6: Space of possible pointer analyses

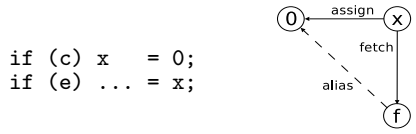

Figure 7: Assignment of x followed by a fetch of x

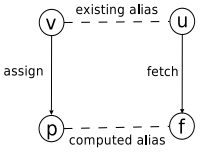

Figure 8: Assignment and fetch with aliased sources

treating conditions, but less accurate in strong updates.

3.2 Construction of Assign-Fetch Graph

Construction of Assign-Fetch Graph is the same independently of a choice of accuracy; that choice plays a role only in determining which assignment can *affect* which fetch. Nodes in an AFG represent values, edges represent contents of address nodes. There is a straightforward correspondence between a flow graph and an AFG. For each fetch operation of a flow graph there is a fetch edge in its AFG, whose sink node represents the value fetched. For each assign operation of a flow graph there is an assign edge in its AFG, whose sink node represents the value assigned. For example, see Figure 7. The solid arrows in Figure 7 are obtained from the flow graph. The dashed line is calculated by pointer analysis and represents aliasing between the fetch of `x` (labeled `f`) and the value 0. Analysis that considers conditions would label the assign with the condition c and the fetch with the condition e as obtained from the flow graph. It would then calculate the condition $c \wedge e$ for the alias edge between `f` and 0.

A general situation is in Figure 8. Assume that previous calculation discovered that `u` and `v` alias. Here `u` and `v` are not necessarily addresses of variables, but results of fetches and therefore they may alias. That triggers a new alias between the contents `f` of `u` and the value `p` assigned to `v`.

The sinks of an assign and a fetch edge alias if the assign can *affect* the fetch, written $\mathcal{A}(assign, fetch)$. The relation *affect* depends not just on the two given edges, but also on the rest of the graph. For example, in Figure 8 one might calculate
$\mathcal{A}(assign, fetch) = \mathcal{C}(assign) \wedge \mathcal{C}(fetch) \wedge \mathcal{C}(u, v)$,
where $\mathcal{C}(assign)$ is the condition of the assign, $\mathcal{C}(fetch)$ is the condition of the fetch, and $\mathcal{C}(u, v)$ is the condition of aliasing between u and v.

The relationship *affect* is in general not computable, and therefore different analyses are derived by different approximations. The choice of an approximation for *affect* determines the accuracy of the analysis. The next section describes our choice of an approximation.

As mentioned in Section 2 we rely on pointer analysis to

generate the lock hierarchy. That requires that lock acquisition and release operations trigger the same actions that the dynamic algorithm of Section 2 would perform.

When a call to a procedure acquiring a lock is being translated to the AFG, the call normally has an argument that is a pointer to a lock. That triggers the following additions to the AFG.
First a new heap node representing an instance of the lock is created. Second the AFG gets all the assignments and fetches that would be performed by appending the new instance to a global locklist. Thirdly a *hierarchy* edge is added from the new instance to the previous item on the locklist. The hierarchy edge is a *payload* edge. Payload edges are not interpreted by pointer analysis, they are just copied when their sources or sinks alias to other nodes. This is a general mechanism we use to make pointer analysis propagate client's information.
When a call to a procedure releasing a lock is being translated into the AFG, we add the assignments and fetches that would be performed by removing from the locklist the most recent instance of the lock. The hierarchy edges remain and become part of the procedure's summary.

4. POINTER ANALYSIS CHOICE

This section describes the actual implemented analysis, marked as "BEAM analysis", in Figure 6. It represents a certain compromise between efficiency and accuracy. The objective is an analysis as efficient as possible, while providing the accuracy needed for multi-threaded programs as explained in Section 1.

4.1 Statement order

As mentioned previously there is no advantage in compromising on the full partial order of statements. In addition, respecting statement order is essential, as explained under "Mutex ordering problem" of Section 1. This section describes how we represent that order efficiently.

The representation assigns a certain quantity *rank* to each operation edge. We will define a relation $\sqsubseteq$ on *ranks* so that the following property holds. (For two operations in a flow graph we write $op_1 \sqsubseteq op_2$ to mean that op_1 precedes op_2 in every execution.) Consider any two memory accesses op_1 and op_2 in the flow-graph, and let e_1, e_2 be the corresponding edges in the AFG.

$$\texttt{if } op_1 \sqsubseteq op_2 \texttt{ then } rank(e_1) \sqsubseteq rank(e_2) \quad (1)$$

$$\texttt{if } rank(e_1) \sqsubseteq rank(e_2) \texttt{ then } op_1 \sqsubseteq op_2 \quad (2)$$

The method described next ensures (1) for any flow graph, and it ensures (2) for planar control flow, which includes all practical programs.

We perform two topological sorts on the flow-graph of a given procedure. As topological sorting is allowed to make arbitrary choices at branch points (if-statements, switch-statements), our two sorts will make those choices differently – one sort will visit the branches of a conditional statements in left-to-right order, the other will do so in right-to-left order. Each of the two sorts will assign an order number to each operation of the flow-graph. The rank of an operation is then the pair of those two order numbers. The relation $\sqsubseteq$ is defined by
$[l_1, r_1] \sqsubseteq [l_2, r_2]$ `iff` $l_1 \leq l_2$ `and` $r_1 \leq r_2$, i.e., one operation precedes another if it precedes in both topological sorts.

The ranks of two edges can then be used in the decision whether an assignment edge can affect a fetch edge. If

$$rank(fetch) \sqsubseteq rank(assign) \quad (3)$$

then the $fetch$ cannot happen after the $assign$, so there can be no effect. In the usual case when all branching is due to conditionals, as opposed to parallel execution, an even stronger property holds: It is necessary that

$$rank(assign) \sqsubseteq rank(fetch) \quad (4)$$

for the $assign$ to affect the $fetch$. If neither (3) nor (4) holds then the two edges lie on different branches, and neither precedes the other topologically. Semantically an assignment could affect a fetch on a different branch only if the branching represents parallelism, not conditionals.

4.2 Conditions

Section 1 (under "Conditional mutex problem") explains why it is necessary to take conditions into account. It can be done to different degrees, as illustrated by the following two examples.

```
f(unsigned a)
{
  if (a >= 2) {...}
  else        {...}
  if (a >= 5) {...}
  else        {...}
}
```

(f)

```
g(unsigned a, unsigned b)
{
  if (a + b > 0) {...}
  else           {...}
  if (a     > 0) {...}
  else           {...}
}
```

(g)

Figure 9: Functions with four topological paths, but only three feasible paths

Traditional pointer analysis algorithms will consider all the topological paths in a program, as in Figure 9. In Section 1 we saw that this is inadequate for analysis of multi-threaded programs. Therefore we had experimented with letting our pointer analysis use the theorem prover that is part of our BEAM tool [3]. That lets pointer analysis consider only the three feasible paths in the functions of Figure 9. However, that approach is too expensive computationally. Therefore we adopted a more efficient compromise, which proved adequate for analysis of multi-threaded programs. That compromise allows our pointer analysis to consider only the three feasible paths of Figure 9(f), but it considers all four topological paths of Figure 9(g). This section then describes this handling of conditions.

As mentioned above the input into pointer analysis is a flow graph. A condition, such as $a + b > 0$, is represented in the flow graph by a pair – an edge and a range. In our example the edge would represent $a+b$ and the range would be $[1, \infty]$. The edge representing $a + b$ comes from an addition node whose inputs represent a and b. In our compromise for handling conditions we ignore the source of the edge $a + b$ and treat it as if it were independent of the edge representing a.

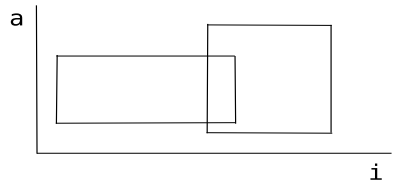

Figure 10: Two conditions represented as cubes in 2-dimensional space

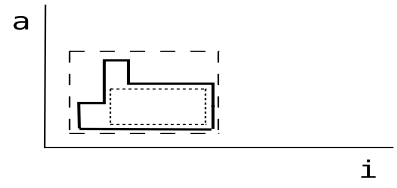

Figure 11: A general condition is an irregular shape (solid lines) approximated by an upper bound cube (dashed lines) and a lower bound cube (dotted lines)

This makes our handling of conditions more efficient than full path-sensitive approach, because it avoid the expense of a theorem prover for arithmetic and other operations.

This compromise allows a representation of conditions that is efficient for certain frequent manipulations, such as conjunction of two conditions. For each condition edge in the flow graph, we collect all the integer ranges the edge is tested against. In the example of Figure 9(f) a is tested against the ranges $[2, \infty]$ and $[5, \infty]$. That divides the entire range $[0, \infty]$ of possible values for a info three subranges: $[0, 1]$, $[2, 4]$, $[5, \infty]$. The three subranges are assigned a bit position in a bit vector of length 3. For example, 010 represents the condition $a \in [2, 4]$, and 101 represents the condition $a \in [0, 1] \cup [5, \infty]$.

In a procedure there are many values like a, each tested against several possible ranges. Each such value is assigned a bit vector as above. All the bit vectors are then concatenated into one long bit vector according to an arbitrary but fixed ordering of all the values. For example, suppose that a program, besides a contains a signed variable `i`, which is tested for being in range $[-\infty, 0]$ or in range $[1, \infty]$. And suppose that the ordering of values is a, i. Then the bit vector 10110 represents $a \in [0, 1] \cup [5, \infty]$ `and` $i \in [-\infty, 0]$. The bit vector 11111 represents no information about a or i (both are allowed any value). The bit vector 11100 represents a contradiction because i is allowed no value.

We can visualize conditions by considering each bit string as a cube in a n-dimensional space, where n is the number of values involved in all conditions of a procedure. For example, Figure 10 shows a 2-dimensional space for the two values a and i. A condition is a rectangle, or in general an n-dimensional cube. (Geometrically speaking it can be actually a union of several disjoint cubes, just like $[0, 1] \cup [5, \infty]$ is a union of two disjoint intervals. But from the point of view of logical operations a set of cubes behaves like a single cube.)

Below we will describe three operations including union (disjunction) and intersection (conjunction). While intersection of two cubes is always a cube, that is not true about a union. The result of several conjunctions and disjunctions then becomes an irregular shape as in Figure 11. We cannot represent it by a cube, so we approximate it from above and below.

Thus each operation edge in an AFG is assigned a pair of cubes $[u, l]$, i.e., a pair of bit vectors. The cube u is the upper bound on the true, but unrepresentable, condition; while l is the lower bound.

There are three operations that we need to define on conditions. For each the upper bound of the result must be an over-approximation of the true result, while the lower bound must be an under-approximation.

$$[u_1, l_1] \wedge [u_2, l_2] = [u_1 \wedge u_2,\ \ l_1 \wedge l_2] \tag{5}$$

$$[u_1, l_1] \vee [u_2, l_2] = [u_1 \vee u_2,\ \ l_1 \vee_l l_2] \tag{6}$$

$$[u_1, l_1] \setminus [u_2, l_2] = [u_1 \setminus_u l_2,\ \ l_1 \setminus_l u_2] \tag{7}$$

Conjunction (5) is the simplest, because the intersection of two cubes is always a cube. It can be carried out by bitwise AND on the bits representing the cubes. For example, consider a program condition `2 <= a && i > 0`. The condition `2 <= a` is represented by the bit vector 01111. (Out of the first three bits allocated for a two are on, and both bits allocated for i are on, representing no constraint on i.) Similarly the condition `i > 0` is represented by 11101. Their bitwise AND 01101 is then the representation of `2 <= a && i > 0`.

On the other hand, a union (6) of two cubes is not necessarily a cube. The upper bound can be obtained by simple bitwise OR on the two upper bounds. The lower bound, represented by the operation $\vee_l$, does not have such a simple implementation. A conservative under-approximation of the lower bound would pick one of the two given lower bounds to become the lower bound of the union. Our approach is less conservative, recognizing several common situations where the union of two cubes is a actually cube, yielding tighter bounds in the most common situations.

Conjunction and disjunction are the only operations needed to represent all the conditions of the program. (Negation is not necessary because of the special way conditions are represented in the flow graph.) Those two operations are also sufficient for calculating the effect $\mathcal{A}(assign, fetch)$. Moreover, the upper bounds alone are sufficient for that.

The need for lower bounds comes only if we want strong updates, as described in the next section. For that we have the operation of relative set complement (7). As is the case with disjunction, removing one cube from another need not yield a cube. We need a separate approximation $\setminus_u$ for the upper bound and $\setminus_l$ for the lower bound. The lower bound is simpler – it is sufficient to reduce only one dimension of l_1, chosen as as to maximize accuracy; and that operation can be performed by bitwise AND and bitwise complement. The upper bound $\setminus_u$ involves similar heuristics as for $\vee_l$.

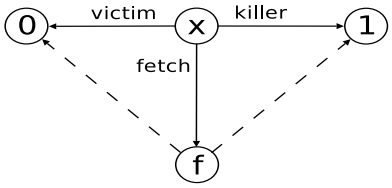

Figure 12: Under what conditions does the fetch node alias to 0 and 1?

4.3 Strong updates

Strong updates refer to the fact that a later assignment overwrites (kills) a previous assignment into the same variable. Analysis without strong updates would assume that the variable may contain values assigned in earlier as well as later assignments. That is inadequate for our purposes as illustrated by Figure 2. Locking a mutex is represented by assignments involved in adding a lock instance to the global locklist. Unlocking is represented by assignments that are supposed to restore the original state of the locklist. Without strong updates it would appear that unlocking does not completely undo the effect of locking.

We will first illustrate the computation on the simple example of Figure 5. The AFG representation is in Figure 12. The assignments of 0 and 1 are labeled "killer" and "victim" respectively, because the assignment `x = 1` may kill the assignment `x = 0`. Recall from Section 3 that the objective is to calculate under what conditions `y` will contain 0, under which it will contain 1, and under which it will retain its original value. That is, we need to calculate the aliasing conditions for the fetch node f, which represents the r-value of `x` in the assignment `y = x`.

First consider the condition $\mathcal{A}_0$ of the fetch being affected by either assignment as would be calculated by analysis ignoring strong updates.
$\mathcal{A}_0(victim, fetch) = \mathcal{C}(victim) \wedge \mathcal{C}(fetch) = c \wedge e$
$\mathcal{A}_0(killer, fetch) = \mathcal{C}(killer) \wedge \mathcal{C}(fetch) = d \wedge e$
In our example both conditions are exact, in the sense that upper and lower bounds are identical.

Now we calculate the effect $\mathcal{A}$ taking strong updates into account. Since nothing can kill the assignment `x = 1`,
$\mathcal{A}(killer, fetch) = \mathcal{A}_0(killer, fetch) = d \wedge e$.

When calculating the condition of aliasing to 0, we need to consider the possibility that the assignment *killer* may kill the assignment *victim* because the killer lies between the *victim* and the $fetch$. That is,

$$rank(victim) \sqsubseteq rank(killer) \sqsubseteq rank(fetch). \quad (8)$$

The $fetch$ is affected by the *victim*, provided it would be affected even without the *killer* (i.e., $\mathcal{A}_0(victim, fetch)$), and secondly that the $fetch$ is not affected by the *killer*. Thus we can write

$$\mathcal{A}(victim, fetch) = \mathcal{A}_0(victim, fetch) \setminus \mathcal{A}_0(killer, fetch) \quad (9)$$

The calculation (9) evaluates to $c \wedge \neg d \wedge e$.

In general, strong updates are taken into account whenever calculating $\mathcal{A}(victim, fetch)$ for any pair of assignment *victim* and any $fetch$ edge. First we calculate the conditions $\mathcal{A}_0$ without strong updates. Then for each assignment *victim*, we calculate $\mathcal{A}(victim, fetch)$ by repeatedly applying (9) for every *killer* assignment satisfying (8).

A pointer analysis with strong updates needs to address an issue with convergence of inter-procedural analysis. If the number of pointer relations inside a callee increases, then more assignments may be killed in the caller, potentially decreasing the number of pointer relations in the caller. That would violate the monotonic increase needed for convergence.

We deal with the issue of convergence when analyzing a strongly connected component of a call graph. When embedding a callee summary in a caller, there is a process of translating conditions coming from the callee into conditions in the caller. During this process callees from the same strongly connected component as the caller receive special treatment. Any condition from the callee gets its lower bound replaced by the empty cube. That prevents any edge from such callee from killing any assignment in the caller. No such changes are needed for conditions in callees outside the strongly connected components, because our inter-procedural analysis proceeds bottom-up through the call graph.

5. EXPERIMENTAL RESULTS

We have applied our implementation to the AIX kernel and to the Parsec benchmarks [15], without reporting any deadlock in either. While such benchmarks drove the accuracy of the pointer analysis, its sufficiency is based on manual inspection of the summaries, and the lack of false positives. However, we did not perform manual deadlock detection to see if there are false negatives.

In this section we will illustrate the trade-off between accuracy and run-time of our pointer analysis using the open source examples of Table 1. This is to show the costs associated with increased accuracy.

The column "lines" gives the line count (excluding comments and blank lines). The remaining columns give run times in increasing order of precision. The column "flow insens." is equivalent to Anderson style flow insensitive analysis, which excludes everything described in Section 4. The column "order sens." includes the effects of Section 4.1 only. The column "condition sens." includes the effects of Section 4.1 as well as Section 4.2. The column "kill sens." includes everything described in Section 4.

	lines	flow insens.	order sens.	condition sens.	kill sens.
Gzip	5,317	0:11	0:11	0:13	0:13
Ispell	6,445	0:13	0:14	0:16	0:16
Pcre	11,639	9:05	1:23	1:45	1:53
Make	15,054	2:27	1:26	1:56	1:55
Bison	19,318	1:48	0:54	0:56	0:56
Tar	21,651	1:21	1:11	1:18	1:20
Balsa	106,716	memory	2:37	2:53	2:53

Table 1: Run-times with different levels of precision

To understand the experimental results, we need to explain

the analysis flow of the BEAM tool. It is performed in several steps listed below. The run-times of Table 1 include steps 1) - 6), but not step 7).
1) Parse source code.
2) Build flow graphs for all procedures. These flow graphs are build under very conservative assumptions, as neither MOD nor alias analysis is available.
3) Based on the flow-graphs from step 2) perform interprocedural pointer analysis as described in this paper.
4) Using the results of 3) simplify the flow graphs. This involves computing MOD and alias information.
5) Discard all the inter-procedural results, and redo interprocedural analysis, including the pointer analysis described in this paper.
6) Report those errors whose determination is based on pointer analysis alone. This includes the deadlock detection described in this paper.
7) Report those errors that use data-flow analysis and symbolic execution.

The above benchmark runs differ from our normal production runs in one way. Some of the benchmarks consist of several files. Normally we build the flow graphs in step 2) for procedures in one file at a time. For the purposes of this paper we combined all the source files into one, which caused out-of-memory condition for Balsa under the least accurate pointer analysis.

Increased precision of pointer analysis also has effect on the search for sequential bugs in step 7), although that is not the subject of this paper. Step 7) performs expensive symbolic execution and theorem proving, whose run-time swamps the time spent in pointer analysis. The effect of larger precision is to increase or decrease the number potential bugs requiring the expensive checking, which causes a wide variation in run-time. In terms of the number of (sequential) bugs reported the picture is a little clearer. Adding strong updates (Section 4.3) on top of condition-sensitive analysis (Section 4.2) had no affect on the bugs reported in our benchmarks. Also adding order-sensitive analysis (Section 4.1) on top flow-insensitive analysis had little effect; in our benchmarks one false positive was eliminated. On the other hand, adding condition sensitivity increased the number of detected bugs substantially. The reason is our conservative suppression of errors in the face of uncertainty, which was reduced by more precise MOD analysis.

6. CONCLUSIONS

We have presented a pointer analysis that is accurate enough for detecting certain concurrency bugs without overwhelming the user with false positives, and without the need for symbolic execution. Its application to deadlock detection, which handles pthread and the AIX libraries, is now in production use.

The main strength of the pointer analysis is its ability to handle statement ordering, program conditions and strong updates, and do so without the usual expense of flow-sensitive analysis. The experimental results show that the increase in accuracy comes with very small penalty in run-time, and may even improve run-time.

7. REFERENCES

[1] D. Brand. A software falsifier. In *Proceedings of Software Reliability Engineering (ISSRE)*, pages 174–185, San Jose, California, Oct. 2000.

[2] D. Brand, M. Buss, and V. Sreedhar. Evidence based analysis and inferring preconditions for bug detection. In *23rd IEEE International conference on Software Maintenence*, pages 44–53, Paris, France, Oct. 2007.

[3] D. Brand and F. Krohm. Arithmetic reasoning for static analysis of software. Technical Report RC-22905, IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY 10598, Oct. 2003.

[4] M. Buss. *Summary-Based Pointer Analysis Framework for Modular Bug Finding*. PhD thesis, Columbia University, New York, New York, USA, Feb. 2008. CUCS–013–08.

[5] M. Buss, D. Brand, V. Sreedhar, and S. A. Edwards. Flexible pointer analysis using assign-fetch graphs. In *Proceedings of the Symposium on Applied Computing (SAC)*, pages 234–239, Fortaleza, Ceará, Brazil, Mar. 2008.

[6] J. Chow and W. H. III. State space reduction in abstract interpretation of parallel programs. In *Proceedings of the IEEE International Conference on Computer Language*, May 1994.

[7] J. Corbett. Using shape analysis to reduce finite-state models of concurrent java programs. In *Proceedings of the International Symposium on Software Testing and Analysis*, 1998.

[8] D. Engler and K. Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 237–252, 2003.

[9] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. In *Proceedings of International Symposium on Software Testing and Analysis*, pages 133–144, 2006.

[10] P. Godefroid and P. Wolper. A partial approach to model checking. In *Proceedings of the IEEE symposium on Logic in Computer Science*, Amsterdam, The Netherlands, 1991.

[11] C. Z. Guyer and C. Lin. Client-driven pointer analysis. In *Proceedings of the Static Analysis Symposium (SAS)*, pages 214–236, 2003.

[12] J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Transactions on Programming Languages and Systems*, 8(3):268–299, May 1996.

[13] W. Landi and B. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of Program Language Design and Implementation (PLDI)*, pages 235–248, San Francisco, California, July 1992.

[14] A. Loginov, E. Yahav, S. Chandra, S. Fink, N. Rinetzky, and M. Nanda. Verifying dereference safety via expanding-scope analysis. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 213–224, Seattle, WA, 2008.

[15] Princeton University. The princeton application repository for shared-memory computers.

http://parsec.cs.princeton.edu.

[16] R. Rugina and M. Rinard. Pointer analysis for multithreaded programs. In *Proceedings of Program Language Design and Implementation (PLDI)*, pages 77–90, Atlanta, Georgia, 1999.

[17] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, Nov. 1997.

[18] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of Program Language Design and Implementation (PLDI)*, pages 1–12, La Jolla, California, June 1995.