International Journal of Parallel Programming, Vol. 32, No. 4, August 2004 (© 2004)

A Framework for Incremental Extensible Compiler Construction

Steven Carroll[1,3] and Constantine Polychronopoulos[2]

Received December 9, 2003; revised January 13, 2004; accepted March 15, 2004

Extensibility in complex compiler systems goes well beyond modularity of design and it needs to be considered from the early stages of the design, especially the design of the Intermediate Representation. One of the primary barriers to compiler pass extensibility and modularity is interference between passes caused by transformations that invalidate existing analysis information. In this paper, we also present a callback system which is provided to automatically track changes to the compiler's internal representation (IR) allowing full pass reordering and an easy-to-use interface for developing lazy update incremental analysis passes. We present a new algorithm for incremental interprocedural data flow analysis and demonstrate the benefits of our design framework and our prototype compiler system. It is shown that compilation time for multiple data flow analysis algorithms can be cut in half by incrementally updating data flow analysis.

KEY WORDS: Compiler construction; incremental compilation.

1. INTRODUCTION

Compiler design concerns both the effectiveness of the analysis and code optimization methods as well as the efficiency of the compilation process itself. The latter has been the subject of research (along with compiler optimization algorithms) primarily with respect to the complexity of individual analysis or optimization algorithms. However, a global approach to

[1] Microsoft Corporation, 1 Microsoft Way, Redmond, WA 98052, USA. E-mail: scarroll@microsoft.com.
[2] Center for Supercomputing Research and Development, University of Illinois, 1308 W. Main, Urbana-Champaign IL 61801, USA. E-mail: cdp@csrd.uiuc.edu
[3] To whom correspondence should be addressed.

compiler efficiency that considers the entire ensemble of analysis and optimization passes is a considerably more complex subject that has attracted much less attention by the research community.

The complexity of compiler systems, especially those that include program restructuring and code optimization for microarchitectures that support instruction level parallelism (ILP), is such that their development time and, correspondingly, their life-cycle far outlast the life-cycle of processors and computer systems. Typical production compilers represent, at a minimum, tens of man-years of development effort. It is typical to expect a plethora of enhancements to a commercial compiler, including new optimizations, revisions of existing modules, extensions to the internal representation (IR), interfacing with DLLs and multithreading libraries, extensions that are necessary to facilitate porting to various architectures, etc. Leveraging powerful restructuring (and optimizing) compilers across processor and system generations is the norm and the large life-cycle of a compiler almost guarantees that the original developers are never those performing the majority of modifications, extensions and maintenance. Arguably the most commonly used and extended compiler is gcc, which has been modified and ported to a variety of systems since 1987 (or presently 16 years).(1) The result of the various modifications and re-adaptations of a typical compiler over the years is, more often than not, patchwork code that reuses old modules in new contexts and adapts old compilers to work with new language idioms and machine architectures.

In this paper, we present a compiler design framework that facilitates three main goals: (1) extensibility, (2) compilation efficiency, and (3) code optimization. We present our design approach and show that designing for extensibility also improves code optimization and dramatically reduces compilation time.

Our extensible compiler prototype is PROMIS(2) which is a multilingual parallelizing compilation system targeting a host of RISC and CISC architectures, including MIPS and x86 ISAs. One of the goals of the PROMIS project is to develop a compiler framework that allows compiler researchers to develop new analyses and transformations for new architectures without deep knowledge of the underlying IR, data flow and common optimization passes. A library of common passes is provided that the compiler developer can deploy in any order without any chance of interference from new passes.[4] In this paper, we present an overview of our design methodology, describe the implementation of specific analyses and optimization passes that take advantage of the extensibility features, and discuss experiments that clearly demonstrate the advantages of

[4]The PROMIS source code can be downloaded at http://promis.csrd.uiuc.edu/.

the proposed design framework. Eventhough many of the advantages and benefits are of a qualitative nature, our experiments provide strong quantitative evidence about the benefits of our framework with respect to dramatic reduction in compilation time.

In Section 2, we will present background on incremental analysis and the PROMIS IR. Next, in Section 3, we present the architecture of the callback mechanism and the API for simplifying incremental algorithm design. In Section 4, we describe our implementation of a self-maintaining data flow analysis. In Section 5, we present quantitative evaluations of the benefits of this technique in terms of compile time improvement and ease of implementation. Finally, we present related work, future work, and conclusions.

2. BACKGROUND AND OVERVIEW

Throughout the paper we use the terms pass, transformations, optimization, and module interchangeably to indicate an implementation of an analysis or optimization pass. In the case of our prototype compiler, PROMIS, a pass can be any module that involves a total or partial walk through the IR and results in either transforming the IR or gathering data or control flow (CF) information about the target program.

We consider passes to be of one of two categories: analysis passes or transformation/optimization passes. Many traditional compiler modules can be broken into an analysis pass and a transformation pass. For instance, the common subexpression elimination (CSE) pass first calculates which subexpressions are available at each point in the program. That part of the original pass is called the available expression analysis pass. Next, the CSE transformation pass uses the available expression analysis information to replace the common subexpressions with copies.

Frequently, analysis information is used multiple times throughout a complete compilation. For instance, CSE creates new copies that can be copy propagated, which in turn creates more opportunities for CSE. However, any transformation pass that runs in-between the initial calculation of the analysis information and a reuse of that analysis information, can potentially modify the program, thereby invalidating the analysis. If this occurs, it would be necessary to recalculate the analysis information or modify the transformation to maintain the analysis information. Since it would be impossible for the designers of the libraries in PROMIS to predict all possible orderings and new passes that could be added to the system, it would be necessary for the user's configuration of PROMIS to either recalculate needed analysis every time a transformation pass occurs or modify the library code to maintain the information. The former is inefficient and the latter requires the compiler writer to modify unfamiliar code.

The Extensible Compiler Interface (ECI) feature of PROMIS attempts to address these problems. It provides an API that allows passes to track the changes in the IR transparently and independently from the transformation passes using a callback mechanism. Analysis passes can update the state of their analysis information using the changes recorded by the call backs and incremental algorithms. Incremental algorithms are algorithms where the state can be updated to reflect changes in the input. Order of magnitude improvements in compile time have been observed(3) through incremental compilation. In this paper, we present a novel organization for a compiler that significantly improves the modularity and extensibility of a compiler. In addition, the ECI framework provides novel support to ease the job of designing incremental algorithms, which have been notoriously difficult to implement and maintain.

2.1. PROMIS IR

The basics of the PROMIS IR are presented here to the extent necessary to understand the algorithms below. The PROMIS IR is based on a modified Hierarchical Task Graph (HTG).(2) The details of the IR can be found at the PROMIS web site.(4) The HTG has two kinds of nodes, Statement nodes and Compound nodes. Statement nodes are the simple statements that make up the program like Call statements, Assignment statements, Pointer statements and Branch statements. Compound nodes are nodes that contain other nodes. For instance, Loop nodes contain subnodes that represent the body of that loop. Each Loop node has an implicit back edge that connects the end of the body with its beginning. Hierarchical basic blocks are similar to basic blocks in that they contain consecutive nodes without branches. However, the consecutive nodes can be compound nodes that contain branches within them. Figure 1 depicts an HTG. The outermost rectangle is the function block, which is the "root" of the HTG. Ovals represent the loop nodes.

The HTG's nodes are connected by two kinds of arcs: control flow (CF) arcs and data dependence (DD) arcs. The CF arcs are always present and the DD arcs can be constructed if desired. The HTG nodes and the two types of arcs form the *Core IR*. All transformation passes are required to maintain the Core IR structures.

In order to improve the extensibility of the PROMIS compiler, several mechanisms are provided for adding new instructions, types, expressions and analysis information. For this paper, we will focus on how to add new analysis information to the IR. New analysis information called

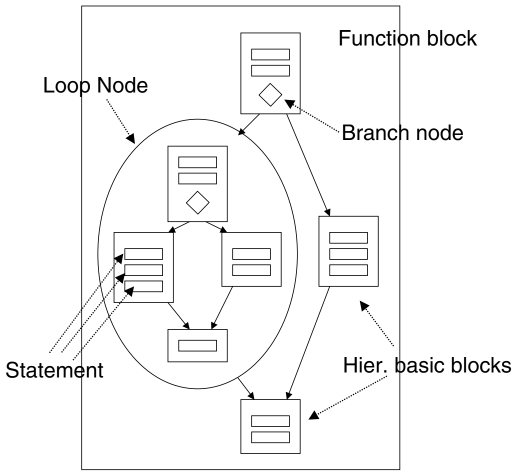


Fig. 1. Hierarchical task graph example.

External Data Structures (EDSs) can be attached to any data structure in the IR. For example, data flow information can be attached to each statement node. An EDS attached to a variable can describe whether that variable is privatizable. They are equivalent to annotations in the SUIF compiler.$^{(5)}$

3. ARCHITECTURE

There are two layers to the ECI architecture in PROMIS. The lowest level consists of the callbacks that are invoked on each modification to the IR. This level provides maximum flexibility to developers of new passes. In addition, there is an API built on top of the raw callback level that provides commonly needed functionality for incremental analysis described in Sections 3.2 and 3.3. It improves ease-of-use at the cost of some generality.

3.1. Callbacks

Callback functions can be registered for a number of different events in the IR. When an event occurs, all callbacks registered for that event are invoked with all relevant parameters. For instance, when a new CF arc is added, the "*add Control Flow Arc*" callbacks are invoked with the new CF arc as a parameter. Since all modifications to the IR go through a protected API, all transformations to the IR will transparently trigger the appropriate callbacks.

The minimum set of callbacks necessary to track changes to a program in the PROMIS IR is node addition and removal, and CF arc addition and removal. The node addition and removal callbacks track not only when new nodes are added or existing nodes are removed, but also when existing nodes are modified. For instance, if a statement node contains the expression $x = 3 + 3$ and it is transformed into $x = 6$ by constant folding, this modification triggers one node removal and one node addition callback. Correctness is assured because any change to an existing node is equivalent to the removal of that node and the addition of a new node with the new expression.

However, tracking node addition and deletion is not sufficient to track all meaningful changes to a program. In order to track so-called *structural* changes, the control flow arcs must also be tracked. A structural change is any change to the IR that can potentially change basic block boundaries or the program ordering of those basic blocks. Removing a branch with a constant branch condition would be one example of a transformation that structurally modifies a program, as would function inlining. The addition or removal of call statements is another example (although these do not require CF arc tracking to detect).

The callbacks presented so far are sufficient to track any change in the program, but a larger set of callbacks is provided to make incremental analysis more convenient and efficient. For instance, any new variables added to the IR can be detected by examining new nodes and modifications of nodes, but we also provide a callback for variable creation to make those events easier to track.

3.2. Event Queues

The callbacks are sufficient to maintain any type of analysis since they can track any change to the IR, but there is a lot of overlap in needed functionality for good incremental algorithms. For instance, it is always best to postpone updating the analysis information until that information is actually queried. This lazy update scheme ensures that valuable compilation time is not wasted updating analysis that will never be used. In this subsection, we present the various APIs that are provided to the compiler developer for managing the changes to the IR. Each of these interfaces is implemented as a set of pre-written callbacks that can be registered for a new analysis pass. When the developer instantiates one of these structures, the callbacks are automatically registered, presenting a simple interface to access change information.

The simplest interface is a dirty flag. The dirty flag is set if any of a list of provided modification types occur (i.e., CF arcs, nodes, or combinations). This interface is useful if the developer of the new pass is interested in whether or not the IR has been modified, but is not interested in how it was modified. A first step for a new analysis might be to simply use the dirty flag to determine if the IR has changed since the initial calculation and then recalculate if necessary.

However, if a compiler developer wants to develop an incremental analysis pass, one available interface is the Event Queue. An Event Queue is simply a list of recorded modifications to the IR in the order in which they occurred. An *event* in our system is a triple of the form {TYPE,ACTION,ID}. TYPE identifies what type of IR structure has been modified (i.e., CF arc, node, etc). ACTION is either an add or a remove. ID is a unique identifier that indicates which instance of the TYPE was modified. The event data structure in PROMIS also stores a pointer to the actual instance referred to by the ID, although they must be treated with care (as shown below). Events of the arc TYPE also store a pointer to the sink and source of the arc.

Event Queues provide a number of extra features. First, *event pruning* is available to remove redundant and intermediate events. For example, if a node is created by one transformation and then eliminated by another between initial calculation and update, that node's events can be safely ignored. The pruning functions will remove these events from the queue. Pruning solves the memory de-allocation problem as well. If a node has been removed permanently, it is likely that its memory has also been de-allocated and is, therefore, invalid. If the last event in a queue for a given *ID* was a *remove* event, all events with that *ID* are marked to signal that the instance is no longer available (and the pointer to it is invalid). The pruning function also labels all source and sink nodes of an arc as removed if the chronologically last event modifying those nodes was a *remove*. One example of the use of these labels is that a CF arc whose sink node was eventually deleted can be ignored in forward data flow problems because data cannot flow to a node that no longer exists.

3.3. Hierarchical Event Queues

Hierarchical Event Queues (HEQs) are an enhancement to event queues that also store the location in the IR where the modifications have occurred. In HEQs, the event queue is distributed throughout the hierarchical structure of the IR. Each compound node in the IR (i.e., loops, blocks) begins with an empty sub-queue. When an event occurs, the event is placed in the sub-queue of the parent compound node where the event occurred. The sub-queue is stored as an EDS attached to the compound node. For instance, if a node is removed, the event for that removal is

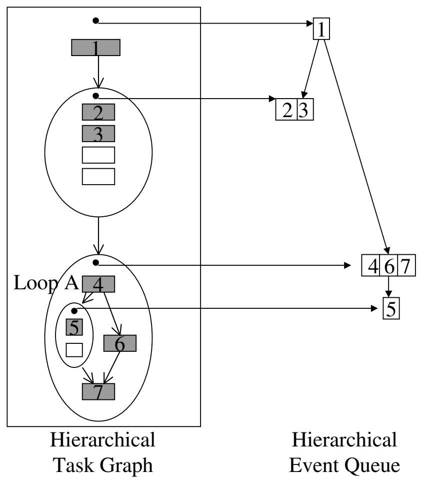


Fig. 2. Hierarchical event queue example.

stored in the sub-queue of the original parent node of the removed node. For events of ARC type, a copy of the event is placed in the parent of the arc's sink and source node. When a compound node is deleted, all the events in that compound node are promoted to the parent node's sub-queue, so that no location information is lost.

An example of an HEQ can be seen in Fig. 2 Several new nodes (shaded and numbered) have been added to the IR since the HEQ began tracking. An event corresponding to each addition is stored in the parent node of each new node, which is the rectangle immediately surrounding it. When the HEQ is accessed a tree is constructed from the compound nodes with non-empty sub-queues up to the root node (the outermost function block). Each compound node can be queried for changes within it. For instance, if queried, the loop node labelled "Node A" will return the four events contained within it. This is useful for demand driven updates since many parallelization analyses are only interested in modifications within loops.

4. APPLICATION: DATA FLOW ANALYSIS

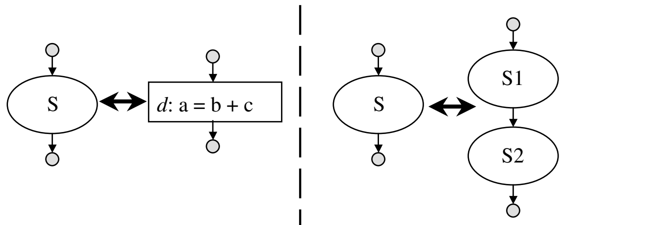
In this section, we give an overview of the data flow algorithms that were developed to showcase the power of the ECI system for extensibility. Several data flow analyses have been implemented in our data flow framework including available constants and copies, and available expressions. Both intraprocedural and interprocedural calculations were implemented.

4.1. Initial Calculation

We chose a structural elimination based approach to data flow calculation for our implementation to take advantage of the loop properties of the HTG. HTG construction restructures highly irregular loops as described in Ref. 6. Our approach is an extension of the structured approach to data flow presented in Ref. 7. First, *gen* and *kill* sets are calculated. The *gen* set enumerates the expressions, constants or copies which are generated (defined) by a statement or block. The *kill* set describes which of these are killed (re-written) by the statement or within the block. The *gen* and *kill* sets are calculated from the bottom statement levels of the HTG up to the root of the HTG. The *in* and *out* sets, on the other hand, are calculated from the root of the HTG down to the statement level.

Figures 3 and 4 give the equations for available copies which are straightforward applications of the standard data flow equations. One technique for applying our technique in a more traditional IR based on a pure CFG would be to extend the handling of loops in the call graph presented below to irregular loops.

To extend our implementation to work interprocedurally, we first sort the call graph into strongly connected components (SCCs). Then, the SCCs are topologically sorted. The calculation is again split into two phases: a bottom-up *gen–kill* calculation and a top-down *in–out* calculation. The first phase proceeds in reverse topological order of SCCs from the functions with no call statements up to the main function. This ordering assures that the *gen* and *kill* set of all called functions (out of the SCC) will already be evaluated.



$$gen[S] = \{d\}$$
$$kill[S] = D_a - \{d\}$$

$$out[S] = gen[S] \cup (in[S] - kill[S])$$

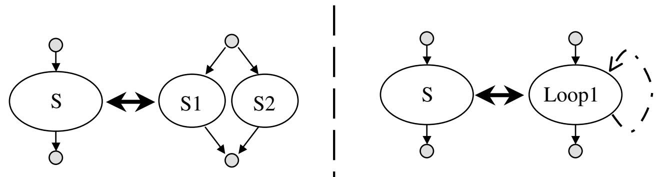
(a)

$$gen\,[S] = gen\,[S2] \cup (gen\,[S1] - kill\,[S2])$$
$$kill\,[S] = kill\,[S2] \cup (kill\,[S1] - gen\,[S2])$$

$$in\,[S1] = in\,[S]$$
$$in\,[S2] = out\,[S1]$$
$$out\,[S] = out\,[S2]$$

(b)

Fig. 3. Data flow equations (I).



(a)

$gen[S] = gen[S1] \cup gen[S2]$
$kill[S] = kill[S2] \cap kill[S2]$

$in[S1] = in[S]$
$in[S2] = in[S]$
$out[S] = out[S1] \cup out[S2]$

(b)

$gen[S] = gen[Loop1]$
$kill[S] = kill[Loop1]$

$in[Loop1] = in[S] \cup gen[Loop1]$
$out[S] = out[Loop1]$

Fig. 4. Data flow equations (II).

The calculation of *gen* and *kill* for each function is similar to the intra-procedural calculation except for call statements. When a call statement is evaluated, the *gen* and *kill* sets are equal to the *gen* and *kill* sets of the topmost node after they are masked to eliminate all the bits that represent expression/copies that cannot be propagated between procedures. For instance, copies of local variables cannot be propagated. We call the masked versions of the topmost (root) node's *gen* and *kill* sets the "function *gen* and *kill* sets."

An SCC in the call graph with more than one function signifies recursion. We use an iterative approach to determine the solution for each function in the SCC in these cases. An initial solution to the *gen* and *kill* sets is determined by evaluating the functions intra-procedurally once with empty *gen* and *kill* sets for the call statements to functions within the same SCC. In a recursion cycle, it can be assumed that each function is called, so the final estimate of the *kill* set for the intra-SCC call statements is the union of the *kill* sets from the initial calculations. The *gen* and *kill* sets are updated with the new estimates and the functions are re-evaluated until the estimates are stable.

After all *gen* and *kill* sets are calculated, the *in* and *out* sets are calculated in the forward topological order of SCCs (from the main function down to the functions with no call statements). The input to the main function is empty, but the input set to all other functions is the intersection of the *in* sets to the call statements to that function. The intersection of those *in* sets is then masked to eliminate non-IP propagatable bits. Like the *gen* and *kill* calculations, for SCCs with multiple functions (recursion), the calculation starts with a conservative estimate and is iterated to improve that estimate. In this case, the initial estimate is an empty *in* set.

4.2. Update Phase

Pseudocode for the update algorithm is presented in Figs. 5 and 6, but an overview is presented here. Our incremental update algorithm uses a HEQ for lazy updates. The HEQ is set to track control flow arcs and nodes. When the analysis information is accessed, the HEQ is pruned and each remaining event is processed. This implementation propagates all changes in a single batch for signi ficant performance gains over evaluating each change individually. For instance, two changes in a single basic block will likely propagate their effect to many of the same nodes. If the effects of two changes propagate together, each affected node will only be re-evaluated once. The algorithm has three phases. In the first phase, the nodes that have been potentially modified are determined from the HEQ. In the second phase, the effects of these changes on the *gen* and *kill* sets and finally, the *in* and *out* sets are updated.

A node stack is used to hold all modified nodes. It has a stack for each level of the HTG up to the maximum depth of the HTG. When a node is pushed on the node stack it is put at the level equivalent to its depth in the HTG. Node stacks are useful because the top-down and bottom-up nature of the computation means that all nodes must be processed level by level.

There are two node stacks for each function, a *gen–kill* stack and an *in* stack. The *gen–kill* stack contains all nodes that need to have their *gen* and *kill* sets re-evaluated. The *in* stack contains the nodes whose inputs have changed. Events in the HEQ are processed as follows.

Node Add events add to the *gen–kill* stack. If node is a new copy or expression, add all the nodes that can kill that node to the *gen–kill* stack.

Node Remove events no action; handled by CF arc.

CF Arc Attach events add the sink node to the *gen–kill* stack.

CF Arc Remove events the sink node and the sink node's parent must be re-evaluated; add the sink node where the event is stored in the HEQ to the *gen–kill* stack. If the sink was not removed, add it to the *in* stack.

The second phase begins by starting at the deepest level of the *gen–kill* stack and re-evaluating the *gen* and *kill* set for each node at that level. If and only if the new *gen* and *kill* have changed, the parent compound node is pushed on the node stack at the next higher stack level and the node is pushed on the *in* stack. This process continues upwards until the node stack is empty or the root node is reached.

The events in the *in* stack are processed from the top level down. Each node is re-evaluated and the input changes propagate forward at that level. If and only if a node is a compound node and its *in* set changes, the

```
void
UpdateInterprocedural(HierEventQueueMap heqs) {
    heqs.PruneAll();                    // prune all the HEQs to remove redundant and dead events
    if (IsCallGraphDirty(heqs)) {
        sccs = CalcSCCs();                    // calculates the SCCs are sorts them in topological order
    }

    // Phase 1: Find Affected Nodes, Build Node Stacks
    NodeListFunctionMap modified_nodes;        // nodes that have been added or modified
    NodeListFunctionMap removed_nodes;         // parent nodes with removed children
    foreach HierEventQueue heq in heqs {
        // find all effected nodes from the HEQ, divide into modified and affected lists
        AffectedNodes(heq,modified_nodes[func],removed_nodes[func]);
    }
    NodeStackMap genkill_ns_map;
    NodeStackMap rem_ns_map;
    genkill_ns_map = ConvertToNodeStack(modified_nodes);
    rem_ns_map = ConvertToNodeStack(removed_nodes);

    // Phase 2: Update Gen and Kill sets
    foreach SCC scc in sccs (reverse topological order){
        if (scc.size > 1) {
            // this scc has multiple functions, requires iteration
            DataFlowSet scc_kill;
            scc_kill = NULL;
            foreach Function f in scc {
                UpdateGenKill(f,genkill_ns_map,rem_ns_map);
            }
            scc_kill = CalcSCCKill(scc);
            bool changes = true;
            while (changes) {
                genkill_ns_map.AddIntraSCCCalls(scc);
                UpdateGenKill(f,genkill_ns_map,rem_ns_map,scc_kill);
                DataFlowSet new_scc_kill = CalcSCCKill(scc);
                changes = (new_scc_kill == scc_kill);        // iterate until scc_kill stabilizes
                if (changes) {scc_kill = new_scc_kill};
            }
        } else {
            // a single function scc
            UpdateGenKill(f,genkill_ns_map,rem_ns_map);
        }
    }

    // Phase 3: Update In and Out sets
    foreach SCC scc in sccs (topological order){
        if (scc.size > 1) {
            bool changes = true;
            // get a list of all the call statements with targets in this scc
            CallStatementList intra_scc_calls = IntraSCCCalls(scc);
            CallStmtInDFMap cs_in_map;
            foreach Call c in intra_scc_calls {
                cs_in_map[c] = c.GetInDFSet();
            }
            while (changes) {  // iterate until stable
                foreach Function f in scc {
                    UpdateInOut(f,rem_ns_map);
                }
                changes = CheckInSetForChanges(cs_in_map);  // reset the map and check for changes
            }
        } else {
            UpdateInOut(f,rem_ns_map);  // update in/out sets
        }
    }
}
```

Fig. 5. Update interprocedural algorithm.

```
void
UpdateInOut(Function func, NodeStackMap ns_map) {
    NodeStack ns = ns_map[func];
    for level (0 to ns.DeepestLevel()) { // top level to bottom level
        foreach Node node in level {      // each affected node at this level
            Node parent = GetParent(node)
            foreach Node node in parent.Subnodes (in topological order) {
                if (level.IsElement(node)) {
                    level.RemoveNode(node);
                    bool changed = RecalcInOut(node);
                    if (changed) {
                        // add the successors of this node to stack for recalc
                        ns.AddNodes(node.Successors(),level);
                        if (node.IsCompoundNode()) {
                            // add the first child of this compound node for recalc
                            ns.AddNode(node.FirstSubNode(),level+1);
                        }
                        if ((interprocedual) && isCallStmt(node)) {
                            // if this is a call stmt, push the top node of the called functions on their node stack
                            foreach Function called_f of node.CallTargetFunctions() {
                                ns_map[called_f].Add(TopNode(called_f),0);
                            }
                        }
                    }
                }
            }
        }
    }
}

void
UpdateGenKill(Function func, NodeStackMap genkill_ns_map,
              NodeStackMap inout_ns_map) {
    NodeStack genkill_ns = genkill_ns_map[func];
    NodeStack inout_ns = inout_ns_map[func];
    for level (genkill_ns.DeepestLevel() to 0) {  // bottom level to top level
        foreach Node node in level {
            Node parent = GetParent(node);
            bool changed = RecalcGenKill(parent);
            if (changed) {
                // add the parent to the gen-kill node stack for re-calc
                genkill_ns.AddNode(parent,level-1);
                // add this node to the in-out node stack for re-calc
                inout_ns.AddNode(node,level);
            }
        }
    }
}
```

Fig. 6. Update interprocedural algorithm (cont.).

in set of the child node is set to the new *in* set of its parent and the node is pushed onto the next lower level of the *in* set.

In the interprocedural case, the SCC ordering is used to determine ordering in which functions are re-evaluated. A change to a function root node's *gen* or *kill* set is reflected by pushing all call statements to that function on the *gen–kill* stack that corresponds to the call statement's function. Changes to the *in* set of a call statement result in the function's root node being pushed on to the *in* stack. Reiteration of the functions within the SCC is performed in the case of recursion until a stable solution is determined.

5. RESULTS

Three key objectives of our design approach are to allow passes to be reordered, to allow new passes to be added without invalidating existing analyses, and to facilitate the development of incremental analyses. In this section, the compile time improvement of incremental algorithms and ease of implementation of incremental analyses are quantitatively evaluated.

5.1. Compile Time

We apply common transformations to several known benchmark codes in order to quantify the compilation time benefits of our approach. Previous quantitative evaluations of incremental algorithms have been based on random call graphs and random modifications.(3) While those results are interesting, we believe it is more important to determine the actual reduction in compilation time in realistic situations for standard benchmarks. The SPEC95 benchmarks were used for our experiments.

The PROMIS compiler was built using Microsoft Visual Studio version 6. It was executed in a Windows XP environment on a dual processor system with 2.66 GHz Xeon processors with hyperthreading enabled. The test system has 1.5 GB of RAM. A high resolution timer library was implemented that uses the processor hardware counters. The PROMIS pass console automatically uses these timers to determine the amount of compile time passed between the start and completion of each compiler pass.

The compile time overhead of executing the callbacks to push events on to the queue was measured by comparing the runtime of the transformation for CSE with and without the callbacks turned on. Callbacks were turned off using a dynamic flag. The overhead never exceeded 2% and was usually too small to be measurable. This was the expected result, since HEQ callback functions were designed to be $O(1)$ for node attach and detach operations.

Figures 7 and 8 depict the speedup of incremental update versus. recalculation for intraprocedural available copy analysis. The initial calculation of intraprocedural available copy analysis is performed. Next, the copy propagation transformation is invoked, changing a certain number of nodes. The number of nodes changed is equivalent to the number of copies propagated because each propagation changes a single node. Then, the available copy information is incrementally updated or cleaned and re-evaluated. Speedup is calculated as the time to recalculate divided by the time to update incrementally. The time to recalculate is the timer reading for a second calculation of the available copies initial calculation after

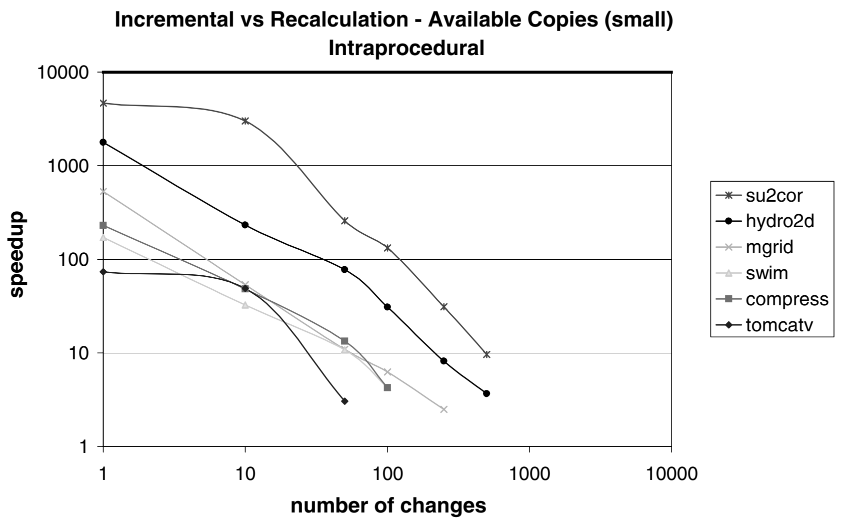


Fig. 7. Intraprocedural speedup — available copies

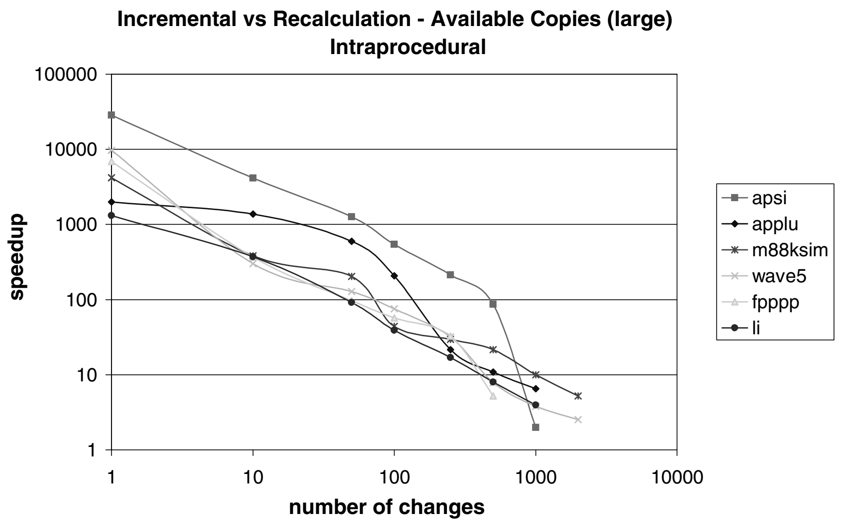


Fig. 8. Intraprocedural speedup — available copies.

the original data has been cleaned from the IR. The cleaning time was not included in the speedup. The time to update is defined as the timer reading for the data flow update pass, which processes the HEQ and then updates the *gen*, *kill*, *in*, and *out* sets.

The x axis depicts the number of changes, which is equivalent to the number of nodes changed by the copy propagation pass. The y axis is speedup as defined above. Each line in the graph represents a different input benchmark from SPEC95. The data is broken up between large benchmarks and smaller benchmarks so that the trend for small benchmark could be seen more clearly. A log–log graph was used because the speedup for a small number of changes is orders of magnitude larger than the speedup for larger numbers of changes.

As can be seen, it is always faster to incrementally update the available copy information after copy propagation rather than recalculate. It is an order of magnitude faster to update for small numbers of changes. Note that each benchmark's data ends when there are no further copies to propagate. It is also important to note that the code generated for the incremental version and recalculation code were compared to confirm the precision of our approach. The emitted code was identical for each benchmark as verified by the different application, which compares each line of the two versions and reports whether or not they are identical.

Figures 9 and 10 depicts the same speedup comparison for interprocedural analysis. Changes in the interprocedural analysis can propagate further than the intraprocedural case because, for example, a new copy that is generated in the *main* function can be available in each function called from *main*. Therefore, the speedup curves drop off faster than in intraprocedural analysis. However, even in the interprocedural case, thousands of changes can be tolerated.

The speedup graphs for intraprocedural and interprocedural incremental available expression analysis are presented in Figs. 11–14. The shapes of the curves are largely similar to the available copies analysis with up to a thousand changes being quicker to incrementally update instead of recalculating. One interesting data point in Fig. 14 is the crossover point where recalculation becomes faster than update. This point is reached in two large benchmarks, *li* and *m88ksim*. It is, therefore, determined that for large benchmarks in the sample used here, the maximum number of changes that should be updated instead of recalculated is approximately 1000 changes.

An optimization driver pass was written to repeatedly execute the constant folding (CF), constant/copy propagation (CP), and CSE passes. In the first half of this pass, CF and CP are run repeatedly until no further changes are possible. In the second half, CSE and CP are run repeatedly until no further changes occur. The driver has two modes: *recalculate* and *update*. In recalculation mode, the data flow information is cleaned and recalculated after each pass that made changes to the IR. In update mode, the data flow analysis is incrementally updated. Speedup results are

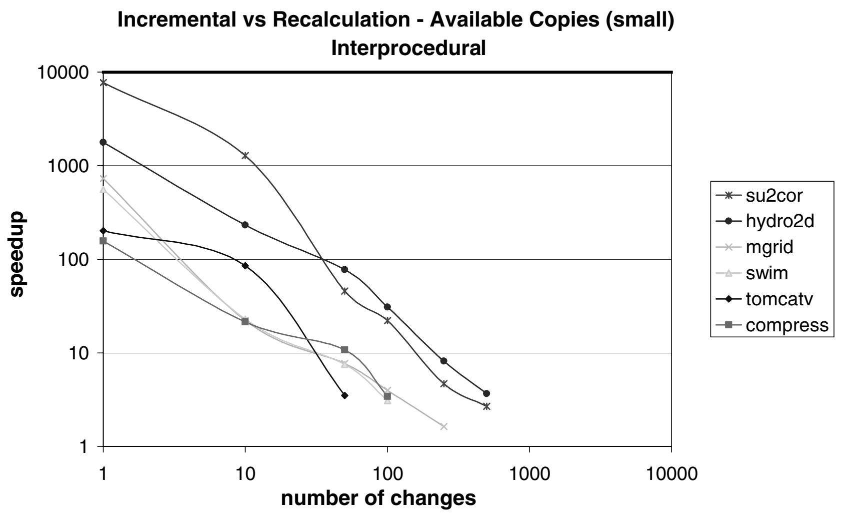


Fig. 9. Interprocedural speedup — available copies.

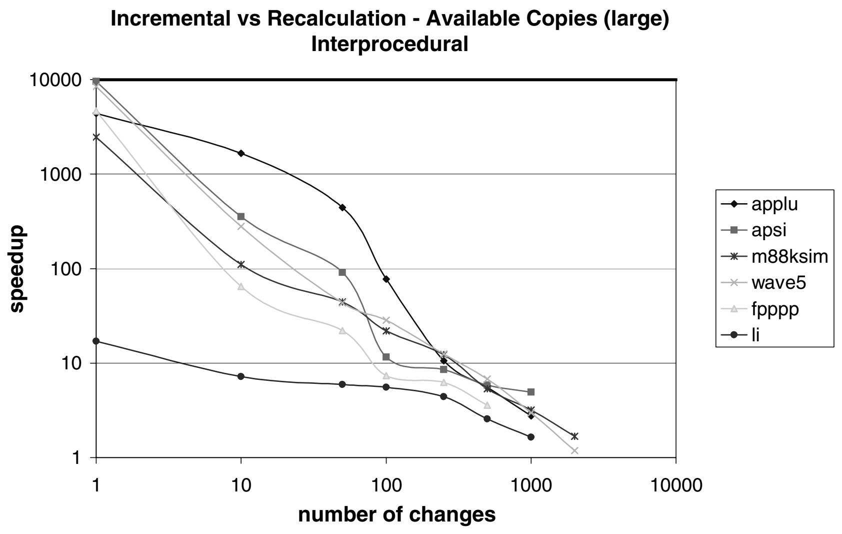


Fig. 10. Interprocedural speedup — available copies.

presented in Table I. Note that *ipupdate* and *iprecalc* represent the analysis time using interprocedural analysis. On average, incremental analysis was about twice as fast as recalculation.

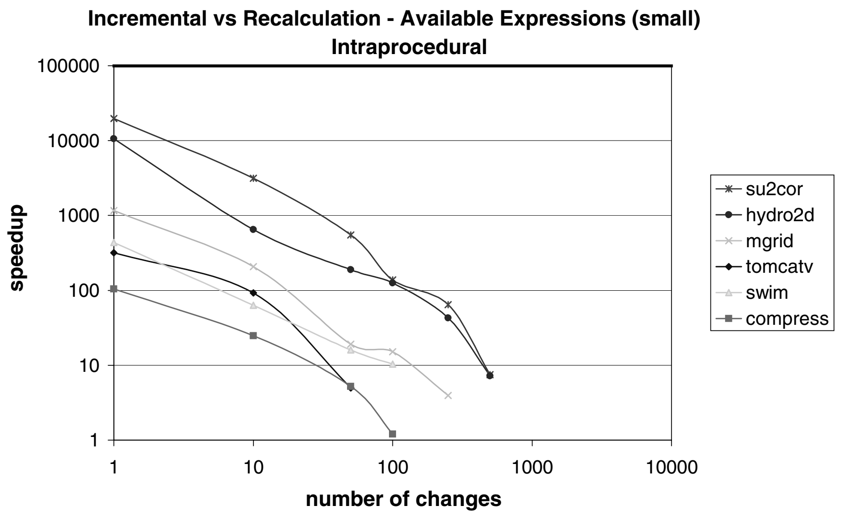


Fig. 11. Intraprocedural speedup — available expressions.

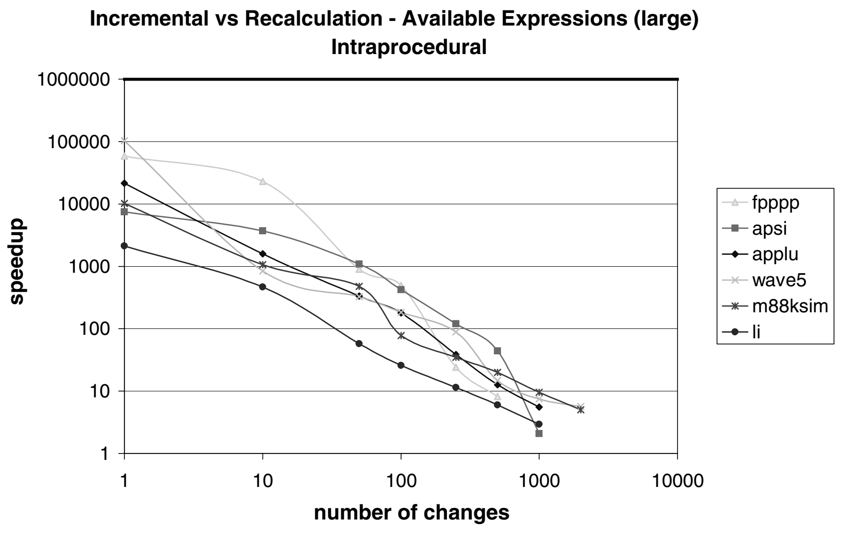


Fig. 12. Intraprocedural speedup — available expressions.

A graphical comparison of the compile time for the two modes in intra procedural and inter procedural data flow drivers can be found in Figs. 15–17. In these graphs, the x axis is now the benchmark. Each benchmark has four bars representing the total analysis computation time

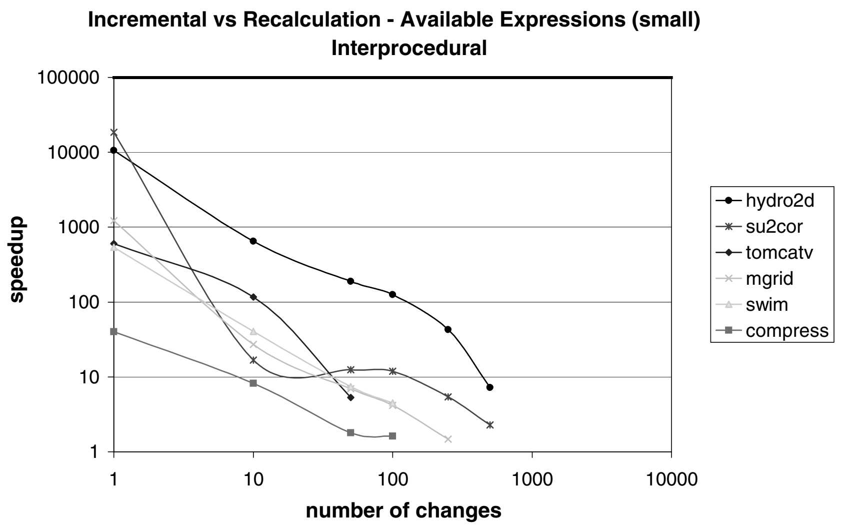


Fig. 13. Interprocedural speedup — available expressions.

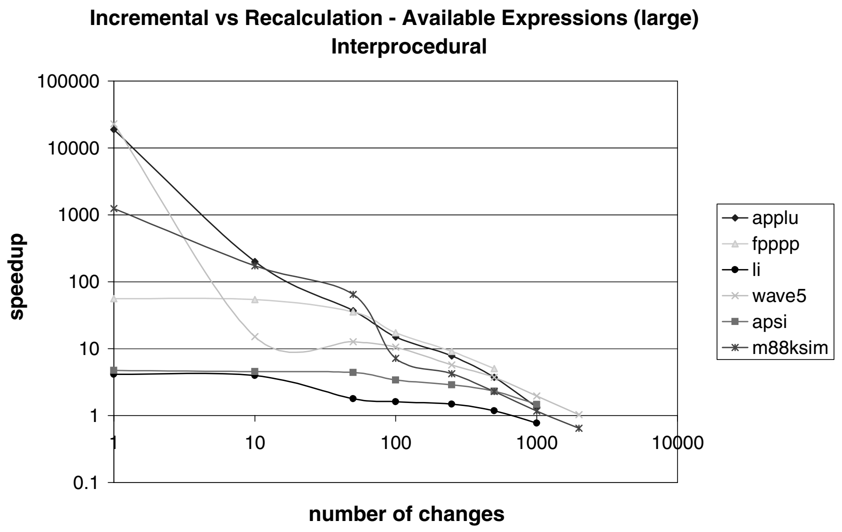


Fig. 14. Interprocedural speedup — available expressions.

for recalculation mode and update mode for both the standard intraprocedural mode and the interprocedural mode. The pass timing data for each benchmark was generated by the timers, and the sum of execution time spent in passes related to analysis (either calculation, update, or cleaning of the data flow analyses during the driver) was calculated. The *y* axis is

Table I. Total Analysis Speedup for Optimizations

Units Benchmarks	Secs Update	Secs Recalc	Speedup	Secs Ipupdate	Secs Iprecalc	Ipspeedup
Tomcatv	0.892078	1.743996	1.95	0.99307	1.951276	1.96
Compress	0.290018	0.833402	2.87	0.600084	1.525409	2.54
Swim	1.654695	3.419882	2.07	2.316679	4.510543	1.95
Mgrid	2.155619	4.795379	2.22	5.171289	8.680112	1.68
Hydro2d	8.226787	17.65996	2.15	20.68031	35.97532	1.74
Su2cor	17.21337	50.09315	2.91	39.11354	88.97242	2.27
Applu	46.1024	136.0613	2.95	72.85377	184.3312	2.53
Fpppp	63.79205	127.8026	2.00	91.73171	179.6379	1.96
Apsi	53.8326	217.615	4.04	315.098	798.2923	2.53
Wave5	58.81569	162.2093	2.76	375.071	679.635	1.81
M88ksim	17.66136	110.161	6.24	375.867	931.7761	2.48
Li	3.29864	26.65209	8.08	116.3783	263.7807	2.27
Average			3.35			2.14

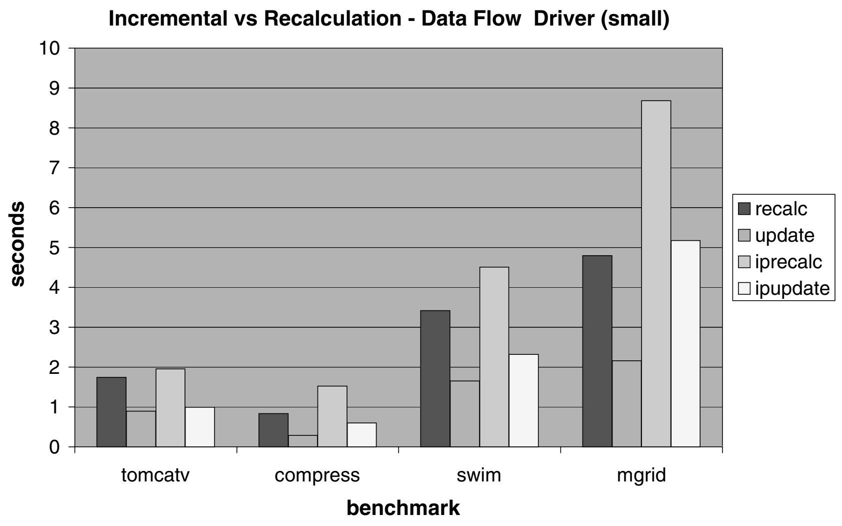


Fig. 15. Incremental versus recalculation — data flow driver (small).

the number of seconds spent on this analysis. It can be seen clearly in the graph that the length of the bars associated with recalculation mode are much longer than the corresponding update bar.

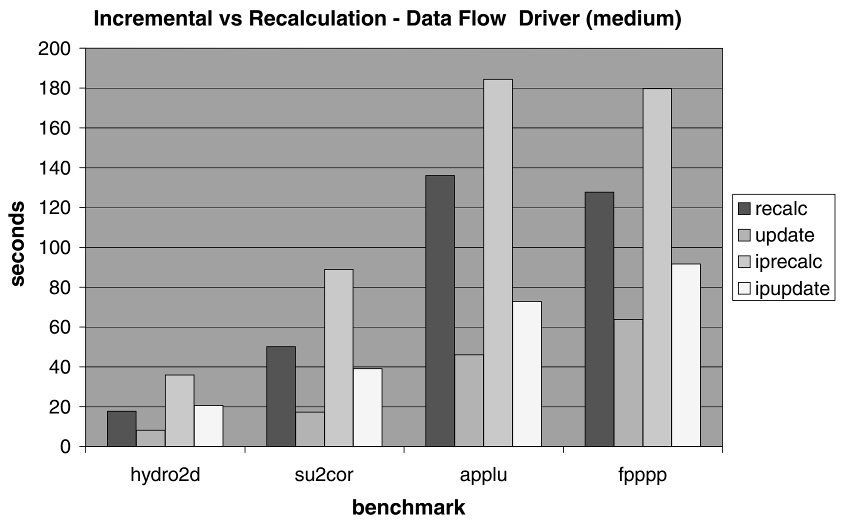


Fig. 16. Incremental versus recalculation — data flow driver (medium).

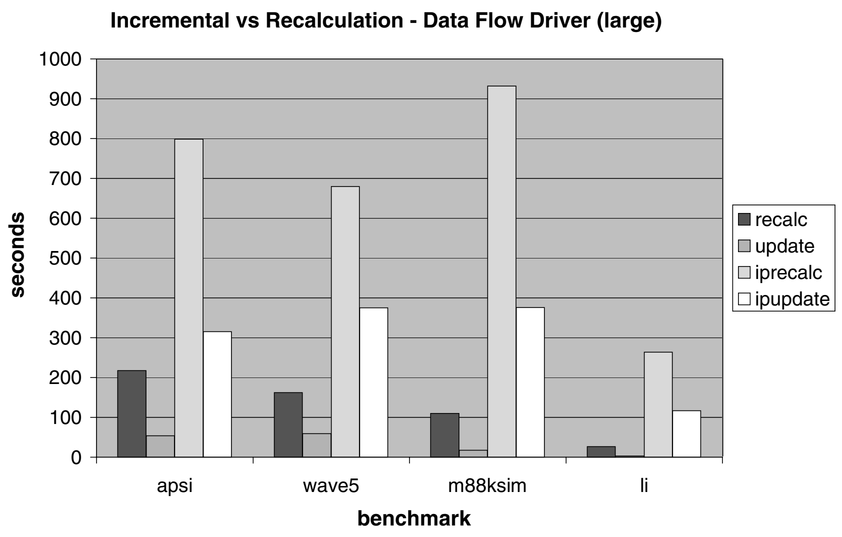


Fig. 17. Incremental versus recalculation — data flow driver (large).

5.1.1. Effectiveness of Event Pruning

Tables II and III present the number of events pruned during the execution of the data flow driver pass. The charts also list the number of constants folded, constants propagated, copies propagated, common subexpressions eliminated and finally the number of passes (CF, CP, CSE) that were executed by the driver. First, it should be pointed out that the number of events pruned is actually smaller in some cases for the interprocedural calculation. This reduction in the number of events pruned is caused by the call graph pruning done by the interprocedural solver. The call graph pruning function removes any function in the source code that is not reachable *via* any call arc from the *main* function. Therefore, the number of functions that are optimized is frequently smaller for the interprocedural analysis when compared to the intraprocedural analysis. The statistics concerning the number of changes made are presented for two reasons. First, they provide outside researchers with data to gauge the strength of the PROMIS optimizer compared to their own optimizers. Second, the statistics demonstrate a general trend that the number of events pruned increases as the number of changes increase.

Pruned events occur in the data flow driver in several situations. The first cause of pruned events occurs in the first half of the data flow driver when the CP and the CF transformations are being alternated. For the following block of code, the available copies are calculated, and it is determined that A = 4 at the beginning of statement 2

Table II. Event Pruning and Optimization Statistics (Intraprocedural)

Bench	Constants Folded	Constants Propagated	Copies Propagated	CSEs Pruned	# Events	# Passes
Tomcatv	2	43	66	16	0	9
Compress	102	86	164	8	66	13
Swim	38	67	138	29	0	9
Mgrid	57	452	278	264	218	11
Turb3d	270	303	848	142	100	15
Su2cor	447	667	1009	260	160	11
Hydro2d	123	298	759	42	68	9
Applu	542	410	1671	144	178	11
Fppp	293	227	440	53	18	9
Apsi	449	2665	1675	1072	678	15
Wave5	417	2744	1552	1084	406	11
M88ksim	938	3237	1870	1302	1672	23
Li	90	1438	609	384	704	25

Table III. Event Pruning and Optimization Statistics (Interprocedural)

Bench	Constants Folded	Constants Propagated	Copies Propagated	CSEs Pruned	# Events	# Passes
Tomcatv	2	43	66	16	0	9
Compress	102	85	150	7	66	13
Swim	38	69	138	29	0	9
Mgrid	55	453	270	214	264	11
Turb3d	275	326	862	140	110	15
Su2cor	445	517	973	169	96	11
Hydro2d	123	293	756	42	68	9
Applu	548	410	1788	144	190	11
Fpppp	294	240	447	53	20	9
Apsi	445	2343	1607	915	592	15
Wave5	416	2832	1449	1053	448	13
M88ksim	931	3102	1803	1254	1604	23
Li	91	1428	611	378	698	25

```
1: A = 4;
2: X = 3 + A;
```

CP modifies statement 2. As discussed earlier, a modification is represented as a detach event on node 2 followed by an attach event on the same node

```
1: A = 4;
2: X = 3 + 4;
```

Next, CF executes and again modifies node 2, creating 2 more nodes.

```
1: A = 4;
2: X = 7;
```

Since changes have occurred, the CP pass must be executed again, and therefore available copy information must be updated and the event pruning algorithm will prune the two extra events on node 2.

Another case where events will be pruned is when a node is copy propagated, creating a new common subexpression which is then eliminated. In the following sequence of code, note that statement 3 is modified twice. The extra modification events will be pruned.

```
1: X = A + B;
2: Y = B;
3: Z = A + Y;

1: X = A + B;
2: Y = B;
3: Z = A + B;
```

1: $t1 = A + B$;
4: $X = t1$;
2: $Y = B$;
3: $Z = t1$;

5.2. Ease of Implementation

One of the main reported disadvantages of incremental compilation algorithms (according to other studies) is the complexity of implementation and maintenance.(8) We argue that our callback system addresses the maintenance issue by making the incremental passes completely independent of transformation passes. In this section, we address the issue of ease of implementation. This is a notoriously difficult factor to evaluate quantitatively, and our approach relies on references to lines of C++ code. The source files for general data flow, available copies, and available expressions were divided into incremental and non-incremental parts. Approximately 2500 lines of code were related to incremental update out of a total of approximately 8000 lines of code. Therefore, about 31% of the code is related to incremental updates. The amount of time invested in developing an incremental update algorithm for the analysis passes was about 50–75% of the amount of time it took to develop the initial calculation pass thanks, in large part, to the extensive reuse of code between initial calculation and update and the functionality provided by the HEQ. Almost all of the code that was developed for the initial calculation was reused for the update algorithm.

6. RELATED WORK

A comprehensive catalog of previous research in general incremental analysis can be found in Ref. 9 By far the most wellresearched incremental compiler framework is abstract grammars. Attribute grammars became very popular for incremental algorithms because of Reps' optimal algorithm for incremental attribute evaluation.(10) Our work is more flexible than incremental attribute evaluation because analyses need not be based on the attribute grammar formalism. This generality is necessary because it is impossible to predict whether future analyses in PROMIS will be implementable in the many types of attribute grammars. The incremental evaluator algorithms implemented using the HEQs can be specialized for each analysis. In addition, a circular attribute grammar would be necessary to handle the interprocedural data flow analysis.(11)

Carle and Pollock(12) developed an incremental source to source parallelizer framework. Like the PROMIS ECI system presented here, a compiler is broken into phases (passes) that communicate only through data attached to the IR. Their approach is also based on an attribute grammar system.

SUIF(5,13) is one of the most commonly used extensible compilers in the research community. It is designed to provide a library of passes that can be used by the compiler designer. SUIF annotations are similar to the PROMIS EDS because they can both be attached to structures in the IR to give more information about them. However, SUIF lacks a system for maintaining these annotations and they can be invalidated by transformation passes that do not understand them. We believe our system is considerably more flexible than SUIF in terms of pass order flexibility and ease of design of incremental analysis passes.

The Montana compiler(14) also has "observers" which is related to our callback idea in that both of them dispatch events to registered observers. However, the paper on the Montana extension mechanisms makes no claims about using the callbacks to perform incremental compilation. Also, the granularity of the Montana callbacks appears much coarser grained than the PROMIS API.

An early framework for incremental data flow analysis information was presented by Tan and Lemone.(15) Like PROMIS, all transformations are required to go through a specially defined interface for moving nodes so that the changes can be tracked. Pollock and Soffa(16) built a compiler re-optimization framework that uses incremental data flow algorithms to trigger when certain optimizations are safe after small edits to the source program.

Ryder *et al.* have produced a series of incremental data flow analyses(17–22) exploring elimination algorithm, iterative algorithms and hybrid elimination—iterative algorithms. Sreedhar *et al.*(23) presented an incremental data flow algorithm to fit the specifics of their "DJ" graph, much like the algorithm in this paper is optimized for the HTG. Any of these algorithms could be implemented in the ECI framework, but a custom algorithm was developed to take advantage of the loop structures in the HTG.

Lerner *et al.*(24) proposed a way of combining separate data flow algorithms into a single driver while maintaining modularity. Each component analysis is extended to allow them to return "replacement graphs" that reflect the improvements made by one module in addition to the *out* set for the node. This technique improves the quality and runtime of data flow analysis, but appears to be restricted to working only with other data flow analyses, and therefore not suited to our more general purposes.

7. FUTURE WORK

We are developing several other analyses in the ECI framework to prove it is sufficiently general, robust and easy to use. Analysis modules that drive automatic parallelization are being developed in order to demonstrate that the ECI system is equally effective at high end analyses as it is at low level analyses like data flow. Specifically, we will be developing an incremental version of array privatization. We are also developing demand driven versions that calculate partial updates of the analysis information at certain regions of the program.

8. CONCLUSION

In this paper, we presented an extensible compiler design framework which facilitates an easy unification of extensive analysis, restructuring and code generation and optimization, resulting in substantial reduction of compilation time as well as in fully modular design and development. We demonstrated the viability of our approach by outlining our implementation of the PROMIS research prototype which was designed explicitly for extensibility. The callback system allows passes to be reordered without concern for invalidating the analysis information. We believe that the callback/event queue design could easily be transferred to improve the extensibility of other compiler infrastructures.

Our design also addresses the difficulties commonly associated with designing incremental algorithms by providing a simple interface for tracking changes. Our tracking system also postpones any incremental maintenance until the information is next accessed in a lazy fashion so that valuable compile time will not be wasted updating analysis information that will not be reused. In addition to the improved modularity of passes and compiler extensibility, the incremental passes implemented in our system also provide considerable compile-time speedups. A new incremental data flow algorithm has been implemented in the incremental framework that is capable of combining the effect of several changes into a single update pass. Order of magnitude changes have been demonstrated for small changes to the IR for data flow analysis. Finally, this research work demonstrated a 100% improvement in compilation time for updating data flow across repeated execution of the CSE, CF, and constant/CP transformation passes on well-known benchmarks instead of random graphs and random modifications like previous works.(3)

9. ACKNOWLEDGMENTS

This work was supported in part by NSF Grant EIA 99-75019 with research support from NSA, and a grant from Intel Corp. The views of this paper do not necessarily reflect the views of the funding agencies. The authors would also like to thank members of the PROMIS group past and present for their contributions to this work.

REFERENCES

1. The GNU Project, GCC Releases, http://www.gnu.org/software/gcc/releases.html.
2. H. Saito, N. Stavrakos, S. Carroll, and C. Polychronopoulos, The Design of the PROMIS Compiler, *Compiler Construction Conference, Lecture Notes in Computer Science*, **1575**:214–228 (1999).
3. B. G. Ryder, W. Landi, and H. D. Pande, Profiling an Incremental Data Flow Analysis Algorithm, *IEEE Transactions on Software Engineering*, **16**(2):129–140 (1990).
4. C. Polychronopoulos, The PROMIS Compiler Web Page, http://promis.csrd.uiuc.edu/.
5. R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S. Liao, C. Tseng, M. Hall, M. Lam, and J. Hennessy, *The SUIF Compiler System: A Parallelizing and Optimizing Research Compiler*, Technical Report CSL-TR-94- 620, Stanford University, Stanford, CA (May 1994).
6. M. Girkar and C. D. Polychronopoulos, The Hierarchical Task Graph as a Universal Intermediate Representation, *International Journal of Parallel Programming*, **22**(5):519–551 (October 1994).
7. A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison Wesley (1986).
8. R. S. Sundaresh and P. Hudak, Incremental Computation via Partial Evaluation, *Eighteenth Annual ACM Symposium on Principles of Programming Languages*, Orlando, Florida, pp. 1–13, New York: ACM (1991).
9. G. Ramalingam and T. Reps, A Categorized Bibliography on Incremental Computation, *Conference record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Charleston, South Carolina, pp. 502–510 (1993).
10. T. Reps, T. Teitelbaum, and A. Demers, Incremental Context-dependent Analysis for Language-Based Editors, *ACM Transactions on Programming Languages and Systems*, **5**(3):449–477 (July 1983).
11. L. G. Jones, Efficient Evaluation of Circular Attribute Grammars, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **12**(3):429–462 (1990).
12. A. Carle and L. Pollock, Modular specification of Incremental Program Transformation Systems, *Proceedings of the 11th International Conference on Software Engineering* (1997).
13. The National Compiler Infrastructure Project, The National Compiler Infrastructure Project, http://www.suif.stanford.edu/suif/NCI (January 1998), also at http://www.cs.virginia.edu/nci.
14. D. Soroker, M. Karasick, J. Barton, and D. Streeter, Extension Mechanisms in Montana, *Proceedings of the 8th IEEE Israeli Conference on Software and Systems* (1997).
15. Z. Tan and K. A. Lemone, A Research Environment for Incremental Data Flow Analysis, *Proceedings of the 1985 ACM Thirteenth Annual Conference on Computer Science*, pp. 356–362, ACM Press, NY (1985).
16. L. L. Pollock and M. L. Soffa, Incremental Global Reoptimization of Programs, *ACM Transactions on Programming Languages and Systems*, **14**(2):173–200 (April 1992).
17. B. G. Ryder and M. C. Paull, Incremental Data-flow Analysis Algorithms, *ACM Transactions on Programming Languages and Systems*, **10**(1):1–50 (January 1988).

18. M. G. Burke and B. G. Ryder, A Critical Analysis of Incremental Iterative Data Flow Analysis Algorithms, *IEEE Transactions on Software Engineering*, **16**(7):723–728 (1990).
19. T. Marlowe and B. Ryder, Hybrid Incremental Alias Algorithms, *Proceedings of the Twentyfourth Hawaii International Conference on System Sciences* (1991).
20. M. Carroll and B. Ryder, Incremental Data Flow Update via Attribute and Dominator Updates, *ACM SIGPLANSIGACT Symposium on the Principles of Programming Languages*, pp. 274–284, ACM Press (1988).
21. M. Carroll and B. Ryder, Incremental Data Flow Analysis via Dominator and Attribute Update, *Proceedings of the Conference on Principles of Programming Languages*, IEEE, NY (1997).
22. J.-S. Yur, B. G. Ryder, and W. Landi, An Incremental Flowand Context-Sensitive Pointer Aliasing Analysis, *Proceedings of the 21st International Conference on Software Engineering (ICSE-99)*, pp. 442–452, ACM Press, NY (May 16–22 1999).
23. V. C. Sreedhar, G. R. Gao, and Y.-F. Lee, A New Framework for Exhaustive and Incremental Data Flow Analysis Using DJ Graphs, *SIGPLAN Conference on Programming Language Design and Implementation*, pp. 278–290 (1996), URL citeseer.nj.nec.com/sreedhar95new.html.
24. S. Lerner, D. Grove, and C. Chambers, Composing Dataflow Analyses and Transformations, *Proceedings of the Conference on Principles of Programming Languages*, ACM, NY (2002).