Call-Graph Caching: Transforming Programs into Networks

Mark Perlin
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract*

There are computer programs that use the same flow of control when run on different inputs. This redundancy in their program execution traces can be exploited by preserving suitably abstracted call-graphs[1] for subsequent reuse. We introduce a new programming transformation *Call-Graph Caching* (CGC) which partially evaluates the control flow of sets of such programs into a network formed from their call-graphs. CGC can automatically generate efficient state-saving structure-sharing incremental algorithms from simple program specifications. As an example, we show how a straightforward, inefficient LISP program for conjunctive match is automatically transformed into the RETE network algorithm. Simple and understandable changes to elegant functional (and other) programs are automatically translated by CGC into new efficient incremental network algorithms; this abstraction mechanism is shown for a class of conjunctive matching algorithms. We establish criteria for the appropriate application of CGC to other AI methods, such as planning, chart parsing, consistency maintenance, and analogical reasoning.

1 Introduction

Caching is a general efficiency mechanism for exploiting redundancy in computation by reusing previously computed information. For example, AI systems often cache problem solving experience learned over time, or *knowledge*, as a set of programs. Intelligent *interactive* systems apply this program set over repeated cycles of interaction with external data input. We define as *persistent* those intelligent interactive systems that apply all knowledge to all data on every cycle. Straightforward implementations of persistent systems are often inefficient, since only some programs in the knowledge base are relevant each cycle, and potentially reusable intermediate computations are discarded.

We have identified a key source of computational redundancy: a program's flow of control, or "call-graph" structure. This control-flow redundancy can be exploited by caching call-graphs.

In this paper, we present a new program transformation *Call-Graph Caching* (CGC) that partially evaluates and then reuses a program's control structure. We:

1. Provide applicability conditions for the use of CGC.
2. Present a working prototype EVAL' which performs partial evaluation of LISP programs into executable call-graphs.
3. Show how CGC helps in the conceptual and implementational derivation of efficient state-saving structure-sharing incremental network algorithms.
4. Illustrate how CGC exploits the fixed call-graph structure to reverse the flow of computation: data can flow up the call-graph, instead of always moving top-down from the program. Thus program execution can be driven from incremental changes in input data, instead of being rigidly preset.

The utility of CGC is demonstrated by:

1. Using CGC to transform a simple, inefficient LISP program for conjunctive matching into the classic RETE match algorithm [Forgy, 1979]. This also demonstrates the use of our automatic transformation prototype EVAL'.
2. Showing other applications of CGC in AI, such as indicating how small changes in our conjunctive match LISP program can mechanically generate alternative network join topologies. Such transformations enable researchers to spend minutes modifying short functional programs, instead of months engaged in low-level network programming.
3. Suggesting that CGC is uniquely suited to AI, in that it potentially increases the efficiency of persistent knowledge-based systems.

CGC differs from ordinary data or instruction caching [Baer, 1980] in that control decisions, not data or actions, are stored. Unlike function caching [Pugh, 1988], CGC caches network traces, not computed function values. CGC observes, records, and removes the control decisions of a program (executing on some input), whereas program dependence graph methods [Ferrante, 1987] do not diminish control flexibility.

After first introducing the CGC transformation (Section 2), we show how to transform matching programs into RETE networks (Section 3). We then present a number of other uses of CGC in AI (Section 4).

*This work was supported in part by grant R29 LM 04707 from the National Library of Medicine, and by the Pittsburgh NMR Institute.

[1]A call-graph is a trace of an executing program's flow of procedural control. With recursive languages like LISP, this is the explicit tree of a program's dynamic procedure calls.

2 Call-Graph Caching

Call-graph caching is a program transformation from arbitrary programs[2] into network programs. After defining our notion of "program", and establishing what "network programs" are, we motivate how the control flow of a program can be cached into a network. We then present the mechanics of the transformation, and describe an implementation.

2.1 Programs

Programming languages provide *action* constructs that operate on input *data*, sequenced according to some *control* organization. The actions are applied to the data by an *evaluator* (such as LISP's EVAL), which may also partially determine the control sequencing. Program execution proceeds by traversing the program's actions according to the control organization, and evaluating the actions on the data. For example, LISP's control mechanism is recursion, which coordinates the λ-expression actions on data arguments, such as symbols and lists. Traversal of a LISP program's code tree recursively executes EVAL on the λ-expressions and arguments at each node in the tree.

2.2 Network Programs

There are programs that operate by traversing an acyclic network. The *actions* of such programs reside and are executed at network nodes, while their *control* organization is represented by network links. The links of the directed acyclic[3] graph (DAG) encode a partial ordering of the nodes. Programs whose network traversals are guaranteed to respect this partial ordering belong to the class NETWORK. Partial order enumeration is usually implemented with some version of topological ordering [Knuth, 1973], and assures that every DAG node is visited exactly once: after each of its predecessors have been visited.

When input *data* is presented to the leaves, computations propagate through the network. The results of local node computations can be locally stored in node memory. A node uses memories of predecessor nodes in bottom-up NETWORK computation similarly to a stack frame's use of recursive return values in usual programming languages. Since the network persists between cycles, finite differencing [Paige, 1982] can reduce redundant computations at a node. To implement this incremental strategy, node memory is divided into a short-term *buffer* and a longer-term *store*, the use of which is shown below.

NETWORK programs are very efficient, with overhead at most linear in |DAG|, the number of nodes and edges [Hoover, 1987]. Propagating from all the leaves, and keeping newly computed values in nodes' buffer memory, topological ordering assures that each node is recomputed exactly once [Perlin, 1988a]. If changes are made to only a subset of DAG leaves, using the store memory (which persists between change cycles) the computation need only be propagated to the transitive closure DAG subset AFFECTED, |AFFECTED| ≤ |DAG|. This produces *state-saving* algorithms that perform minimal recomputation, directed from just the *changes* to their input.[4] When coupled with finite differencing, state-saving *incremental* algorithms result.

The price for this efficiency is inflexible control structure: NETWORK program DAGs have the restricted form of basic blocks [Aho, 1986]. Despite this limitation, NETWORK finds extensive application. Efficient incremental spreadsheets use the algorithm sketched above. Conjunctive matching programs are often cast in network form for efficiency. Other potential applications include the representation of dependency relations, multiple inheritance, consistency maintenance reasoning [Doyle, 1979], and incremental attribute grammar evaluation [Reps, 1983].

2.3 Caching Control Flow

Let P(x,y) be a program in some language, having arguments x and y. Suppose that P's control structure is independent of y. Then partial evaluation [Futamura, 1971] of P(x,y) with respect to some fixed x_0 will yield a new program $P'_{x_0}(y)$ of one argument having a *unique* call-graph.

An interactive program's input either depends on external factors, or is independent of external interactions. Now suppose that program P(x,y) is interactive, and its argument x is independent of external factors. Then P'(y)

1. completely characterizes P(x,y)'s interactions with the external environment, and
2. has a unique call-graph.

We define an interactive program P(x,y) to be **basic** when x is independent of external factors, and P's control structure is independent of y.

P'(y) has a unique call-graph which can be cached as a NETWORK program for later reuse. The program's actions are encoded in the call-graph nodes; each node is allocated local memory. The program's control flow decisions are recorded as call-graph edges. Applying input data to the call-graph leaves, a bottom-up graph traversal (respecting the nodes' partial ordering represented by the edges) can correctly execute the program. We therefore have a new NETWORK program operationally equivalent to the partially evaluated P, as illustrated in Figure 1.

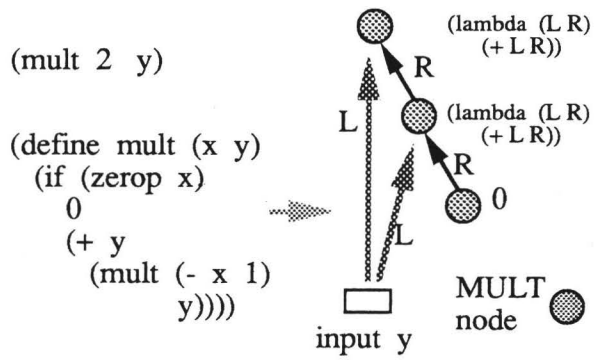


Figure 1: Transforming a program into a network.

[2]Our initial focus in Sections 2 and 3 is on functional-style LISP programs. In Section 4 we look at other programming language styles, such as rules.

[3]Graphs with cycles are also admissible, as long as the cycles are broken at run time; i.e., the graph *acts* like a DAG.

[4]Further efficiency gains are possible by restricting the DAG traversal to the subgraph influencing only select node computations.

2.4 The Transformation

We describe the Call-Graph Caching (CGC) transformation in several loosely coupled steps. The first step assembles the call-graph from its subgraph components. The second collects a set of call-graphs into a network. The resulting cached call-graph network structure is used (and reused) as a data cache. Pedagogical examples on simple polynomials are detailed in [Perlin, 1988b].

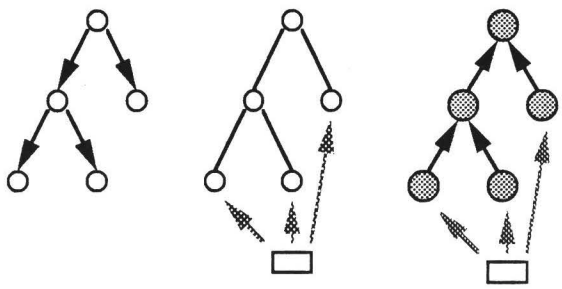
2.4.1 Building the Call-Graph

Control-Flow Caching is an algorithm which builds the call-graph of a program on some input. The construction takes as

- **input** either
  1. the compile-time text of a program, together with its partial input, or
  2. the run-time program executing on its complete input, and
- **outputs** the call-graph of that program.

The procedure employs an auxilliary data structure, the *Control-Flow Cache*, which is used in the assembly of the final call-graph structure. There is also an optional argument specifying the key execution steps to cast into graph nodes.

Control-flow caching proceeds as three separate steps. **First**, with a program, (partial) input, and a user-definable set of the key steps to abstract[5] a trace is formed of the program's execution. Each node in the resulting call-graph represents one (key) step in the program's trace. The actual formation of the call-graph is facilitated by specific control-flow cache management strategies. One such strategy is the above abstraction mechanism of recording only the "key" steps as nodes. Another strategy, used in chart parsing [Winograd, 1983], is exploiting the constraints posted in the control-flow cache to help reduce the executing program's computation, i.e., dynamic programming. Yet another, say for a LISP program, would be to passively cache the succession of execution branches into a full call tree. Regardless of the specific strategies, the resulting call-graph captures (in space) the program's execution over time, as shown in Figure 2, step 1.



(1) (2) (3)

Figure 2: Control-Flow caching: assembling the call-graph.

Program P(x,y) has thus far been evaluated on input x_0, with the partially evaluated P'(y) preserved as a call-graph. The **second** step of Control-Flow Caching connects the external input y to the program's graph. As shown in Figure 2, step 2, the resulting call-graph of the complete program is a DAG[6].

The **third** step *operationalizes* the call-graph into a usable data structure. For example, graph nodes can be augmented with the requisite buffer and store memory with a system-specific representation.

At this point, saving and reusing *just this single* call-graph provides a fully functional state-saving NETWORK program. Directing input y through the graph in a partially ordered node enumeration, with the node memories as a data cache, will perform the computations of $P(x_0,y)$. More efficient *incremental* evaluation via finite differencing is effected by using the local buffer memory to (1) record intracycle computations and (2) differentially update the local intercycle store memory.

2.4.2 Collecting Call-Graphs

The Call-Graph Caching transformation is completed by collecting a *set* of $P(x_0,y)$ call-graphs, for a variety of P's and x_0's. This set is called the *Call-Graph Cache*, and, like other caches, usually employs efficient cache management strategies. For example, spreadsheets and conjunctive matchers exploit common shared prefix structure, with trie-like [Aho, 1983] merging of call-graphs into a single connected network. In allocating the often limited resource of space, another common strategy is to perform a cost/benefit analysis, determining which call-graphs stay in the cache, and which are removed.

2.4.3 Using the Call-Graph Cache

After building the call-graphs of $\{P(x_i,y) \mid i \in I\}$ and collecting them into a call-graph cache, the resulting network is used (and reused) as a data cache. Values or sets of values of y propagate bottom-up through the network, employing the buffer memory within each propagation cycle, and store memory between cycles. For efficiency and correctness, the network traversal control mechanism is partial order enumeration (implemented with a topological sort).

This completes the transformation of a finite set of basic programs (in any programming language) into an efficient state-saving NETWORK program.

2.5 An Implementation

To demonstrate the workability of Call-Graph Caching, we implemented in Common LISP a simple partial evaluator EVAL', which transforms a large class of LISP programs into their corresponding call-graphs. The input to EVAL' is the symbolic LISP expression representing "$P(x_0,y)$", for some P and x_0, and a set of labels denoting the key execution steps to cache. The output is a call-graph, where each node specifies

- the label of the node type;
- a lambda expression containing all the information required to execute the node's computation when applied to the values of its immediate predecessor nodes;

[5]This user-definable set ranges from the empty set to all possible steps.

[6]This is because the caching of the program's execution over time breaks (i.e., unravels) any cycles present in the flow of control.

- recursively, the nodes of its immediate predecessors.

EVAL' performs the following computations:

1. Arrive at (the label of) a key node (i.e., LISP function or symbol) to be abstracted.
2. Perform EVAL' on the unevaluated arguments to the function.
3. Substitute these values into the function, and then execute EVAL' on the LISP function's code tree.

This delayed evaluation is done recursively, caching the control structure into a call-graph. The call-graph's nodes abstract out the set of labelled functions, preserving the local actions required for later execution.

We have also developed a variety of network structure-sharing programs for merging call-graphs, and a partial order network traversal toolkit for executing these cached Call-Graph networks as programs. Our working implementations have demonstrated the efficacy of transforming simple LISP specifications into efficient incremental network programs on AI examples such as RETE matching.

3 RETE Networks: An Example of Call-Graph Caching

RETE matching is a state-saving structure-sharing incremental algorithm used in OPS-5 and other production systems for conjunctively matching many patterns against many objects. Because it provides excellent average-case behavior for an important NP-hard AI problem, it has been extensively studied and varied. It is also a good example of the Call-Graph Caching program transformation, illustrating nontrivial usage of the Control-Flow Cache and the Call-Graph Cache.

3.1 Rule Matching

Forward-chaining Rule Systems (or "production systems") such as OPS-5 are programming languages with *match* as their control element. Program data is organized into a set of rules, having left-hand-side (LHS) *tests* and RHS actions. External input (often called "working memory") comes from a slowly varying set D of data *objects*. If a rule's tests match objects, the rule becomes a candidate for firing; when executed, its actions serve to modify D.

Following common practice, we fix the form of the rules' LHSs to be a set of conjunctive *conditions*, each condition containing tests restricting the set of matchable objects. An *instantiation* of a rule having n LHS conditions is an n-tuple of objects satisfying the rule's LHS tests. On every interaction cycle, the rule evaluator must try to match each rule against all possible combinations of objects in D, forming its set of instantiations

$$\{inst \in D^n \mid \forall test \in Rule,\ test(inst)\},$$

where D^n is DxDx...xD, n times. Letting TEST be the set of allowable tests, and TESTS be its power set 2^{TEST}, *Match* may be described as a function which filters the set of possible instantiations D^n into those satisfying some tests in TESTS,

$$Match_1(TESTS, D \times D \times D ... D) \rightarrow D \times D \times D ... D.$$

Since all conditions are matched against all objects, we may rewrite *Match* in the more convenient form

$$Match(TESTS, D) \rightarrow D \times D \times D ... D,$$

which applies a set of tests to a set of objects, producing a filtered set of n-tuple instantiations.

Production systems are *persistent*, in that they

1. maintain their knowledge in a finite set of experientially derived programs (the rule set), and
2. apply all programs to all available data on each interactive cycle.

They are also *inefficient*. Consider just one rule having n conditions matching against only two data objects- the set of candidate instantiations D^n grows exponentially in n. In fact, conjunctive rule matching is NP-hard [Minton, 1988]. Generally, however, only a small fraction of the object set changes each cycle. So instead of rigidly applying *all* rule programs to the data, perhaps the incremental *changes* to *data* should drive the matching computation.

For each rule program in the rule set specifying some tests in TESTS, *Match*(tests,D) is the computationally expensive subprogram. Observe that with

P = *Match*, x = tests, and y = D,

1. x is independent of the external data input D, and
2. the conjunctive matcher P(x,y) is programmable so that P's control is independent of y (e.g., Section 3.2).

Therefore P is *basic*, and Call-Graph Caching will generate an efficient state-saving structure-sharing incremental *Matching* algorithm.

3.2 Transforming Rule Matching into RETE Networks

We illustrate the use of Call-Graph Caching by generating the RETE network from a simple functional programming specification of the matching function.

1. We start from an easily specified, though inefficient, set of functional programs.
2. Using an auxilliary Control-Flow Cache, partial evaluation of the basic program *match*(T_0,D) produces a call-graph capturing *match'*(D)'s control flow. This call-graph is usable as an incremental data-driven *state-saving* NETWORK program which can store processed input data as intermediate matching results.
3. The Call-Graph Cache merges the individual call-graphs in order to conserve space, and achieve some speedup. This is done by *test sharing*: nodes with common test prefixes are combined to form a single trie data structure.

We now detail this construction of the RETE network algorithm.

3.2.1 Rule Matching as LISP Programs

A rule specifies a fixed set of tests T_0 for its match component. The conjunctive match program *Match*(T_0,D) can be formulated so that its control is independent of working memory D. We now write such a filter *Match* as a functional-style LISP program.

For efficiency on a serial processor, we first impose a fixed ordering on the rules' conditions. Each test examines one or more objects in a candidate n-tuple $\in D^n$; these objects are now ordered by the condition ordering. We now order the test set: associate to each test the number of the

last object it examines, and arrange the tests with respect to this index. For efficiency, the match is performed by a conditional AND, testing a candidate n-tuple against the first test subset, then the second, and so on through the nth. Within the kth test subset, $k \leq n$, the tests may be further grouped into two classes:

A. *alpha* tests on a single object $\in$ D, and

B. *beta* tests on more than one object, i.e., k-tuples $\in D^k$.

There are many ways to write the LISP code for this simple filter (e.g., as one function, iteratively, recursively, etc.). While the CGC transformation is independent of programming style, for clarity, we present *Match* using linear recursion.

```
; Match rule's tests against the data.
(define match (tests data)
   (beta-join
      (first tests)
      (second tests)
      data))
; Join together the preceding sift and
; join sets with a filtering beta test.
(define beta-join (A B D)
   (if (null A)
      '(())
      (filter (first B)
         (set-product
            (alpha-sift (first A) D)
            (beta-join
               (rest A) (rest B) D)))))
; Sift the objects with alpha tests.
(define alpha-sift (A D)
   (set-filter A D))
```

Match takes a preordered set of tests, and a set of data objects as its arguments. The key interesting function is *beta-join*, which merges the simple *alpha-sift* filter with further recursive calls to *beta-join*; this produces a linearly recursive call-graph. Note how the tests in *match*'s *tests* argument are deposited locally at each level of filtering, and that no control decisions are made using the *data* argument.

The auxilliary function *set-filter* returns the subset satisfying some predicate tests, while the function *set-product* operates similarly on a pair of sets.

3.2.2 Building the Call-Graph

STEP 1. For any rule r, calling EVAL' on *match*($tests_r$,D) with the set of labels {match, alpha-sift, beta-join} will save the calling structure of the rule's tests. The control-flow cache is used with the functional program *match* to store to the growing call tree. As shown in Figure 3 A, the call-graph has a linear spine, with the appropriate tests localized at each node.

STEP 2. In Figure 3 B, the free input variable *data* is attached to the call-graph as an input source, turning the call tree into a DAG.

STEP 3. The graph structure of this single rule's match component can now be completed. Memory for the intra-cycle buffer and the inter-cycle store (and other information) can be specialized into a specific graph representation. This call-graph can be reused as a bottom-up NETWORK filtering program. In Figure 3 C, the domains of the filtered objects are shown.

Using the buffer memory, partial order traversal of the call-graph from the data computes the filtered instantiation subset of D^n. If the nodes' longer-term store memories are initially loaded with D (and then continually updated), only

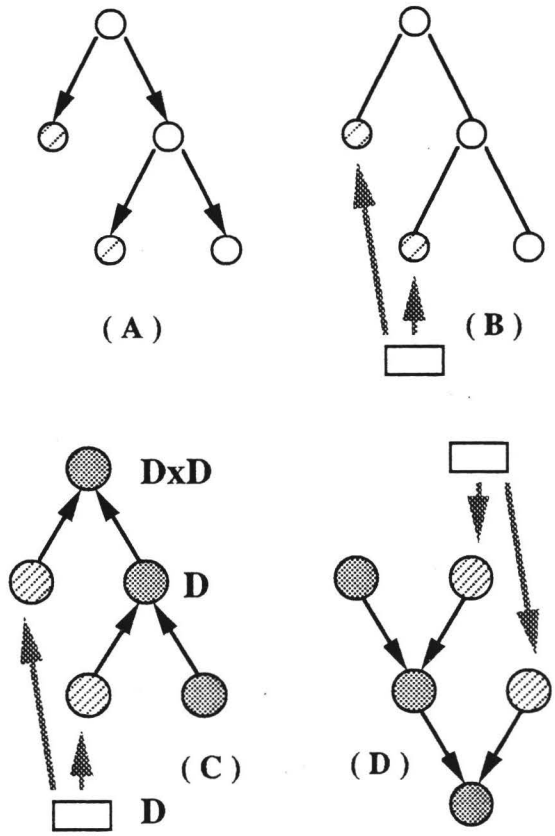


Figure 3: CGC on a linearly recursive conjunctive matcher.

changes to the object set ΔD are needed for computing further instantiations. That is, the call-graph is a *incremental* state-saving data-driven NETWORK program for computing the state of a single rule's match.

This is not surprising: Figure 3 D shows the RETE network beta join topology, which is isomorphic to the *Match* call-graph in Figure 3 C.

3.2.3 Collecting Call-Graphs

A rule system is comprised of a finite *set* of rules; we therefore form the corresponding set of call-graphs, one for each rule's match component. This set is the Call-Graph Cache. One cache management strategy for conserving cache space (with some associated speedup) is to *merge* the call-graph DAGs into one connected network. The matcher's behavior is unchanged if, proceeding from the data input source, nodes are merged based on prefix *sharing of tests*. A succession of call-graphs merging into a common beta-join node trie is depicted in Figure 4.

When an alpha discrimination net is added to the beta join trie, the classical RETE match network is generated. We implemented this addition by modifying the *alpha-sift* LISP function to perform its tests tail recursively. This illustrates how CGC readily produces new desired network topologies from small changes in LISP program specifications.

3.2.4 Using the Call-Graph Cache

Partial Order traversal of the RETE network will perform the match of the rule set (cached as a shared set of call-graphs) against working memory input D. The intracycle buffer and intercycle store memories at each node are used as a *data cache* to preserve the partial match computations

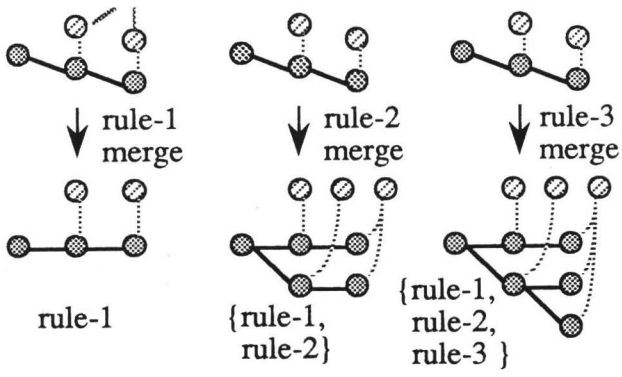


Figure 4: Prefix sharing to merge Call-Graphs.

(within and between) each cycle. Working memory elements are then incrementally added or deleted from these memories.

3.3 Alternate Join Topologies

Call-Graph Caching generates more than RETE networks: one application is the generation of families of efficient conjunctive matchers. By making simple variations in *match*'s LISP specification, and changing which key function names are cached into call-graph nodes, many different join topologies can be designed, easily specified, and automatically constructed. Also, there are other Call-Graph Cache merging strategies besides trie-based prefix sharing.

The RETE example was described above: a linearly structured call graph. One known alternative approach is to *not* cache the beta join nodes [Miranker, 1987]. Another is to structure the call graph as binary tree [Stolfo, 1982], reducing the long linear chains problematic in RETE. We are currently exploring and assessing a variety of new join topologies using CGC as a rapid prototyping tool. These topologies can be custom tailored to task-specific requirements, such as learning or parallelism.

4 Other Uses

With general recursive computation, actions (e.g., floating point arithmetic, file access) are often expensive, whereas actions' control organization and input data can be uninteresting. With intelligent systems, however, actions (e.g., precomputed motions, user queries) are commonly mundane; it is the **sequencing** of such actions to achieve concrete goals that is complex and computationally difficult. Since the CGC method explicitly records such control decisions for subsequent modification and reuse, it provides both a conceptual framework and an implementation strategy. In script-based planning [Schank, 1977], for example, knowledge-based action sequences are retrieved, and each simple action step is replayed. Analogical reasoning [Carbonell, 1983] can extend this retrieval by transformations of retrieved plans. Other applications of CGC in AI are discussed below.

4.1 Control-Flow Caching

The control-flow cache may simply record the unraveling of an execution tree over time, as in the RETE example, or take a more active role, such as enabling constraint-directed dynamic programming. For example, in efficient context-free parsing, each graph node represents an individual firing of a grammar production; the control-flow cache (or "chart") records past firings to constrain future ones.

If rule firings are recorded as CGC nodes, the call-graph of a rule system's problem solving instance forms a trace of the rules' executions. This record (DAG) of the rule and data dependencies may then aid in consistency maintenance analysis for exploring alternative reasoning scenarios. Here the call-graph would form a NETWORK program utilizing its data cache, with ground instance changes incrementally propagated via DAG traversal.

4.2 Call-Graph Caching

Persistent processes maintain their knowledge in a program cache; this cache is augmented or modified as the knowledge changes over time. When the cached programs have call-graphs with sufficient redundancy (e.g., are *basic*), the knowledge may be compiled into an efficient Call-Graph Cache network. For example, consider the rule trace discussed above: DAGs representing arbitrary rule execution are not likely to share similar morphology, and, therefore, they are not usually cached as networks. Rather, such call-graphs are abstracted into networks having a *single* inner node (or "chunk") using EBG [Mitchell, 1986] or some other execution trace generalization method. These reformed networks have sufficient operationality to then be reused in the program cache, resulting in potential efficiency improvements. (In some systems [Laird, 1986], they may be compiled into RETE networks at a lower level of abstraction for further efficiency gains.)

There are many common persistent processes comprised of basic programs. For example, a (multiple) inheritance network will be automatically generated as the cached call-graph set from the process of successive subset classification on some input set. As another example, window systems may be thought of as caching an inefficient "painter's algorithm" redisplay execution into a call-graph based on the *in-front-of* relation. Efficiencies accrue since, in general, the data inside the windows is independent of window redisplay.

4.3 Extensions

CGC is applicable when *persistent* interactive intelligent systems are comprised of *basic* programs. Since not all programs are basic, we consider how to use CGC under weaker assumptions.

As in Section 2.3, let P'(y) be a partially evaluated program. P'(y) need not have a unique call-graph for CGC to be useful. As long as its set of call-graphs is manageable, some caching strategy could succeed. For instance, P'(y) might have only a small finite number of call-graphs. Alternatively, a skewed distribution of P'(y)'s call-graphs could probabilistically ensure manageability. Extending the RETE match example, when disjunction (i.e., choice) is introduced into primarily conjunctive rules, there is still much redundancy in control flow. Though weaker, this redundancy is effectively exploited in RETE-based OPS-5 via copying and conflict resolution.

4.4 Future Work

The CGC transformation lets us reexamine many network algorithms as *partially evaluated programs* which have their call-graphs preserved. Further, as with RETE, it may be the case that reformulation of the network into a new program in some appropriate language leads to clearer specification of the algorithm. Since NETWORK-like efficiency is guaranteed by the CGC transformation, improvements can

then be effected in the abstracted programming language, rather than in the low-level NETWORK language.

Conversely, given a clear specification of a set of programs in some language, when control-flow redundancy is present (whether guaranteed by the "basic" property, or simply empirically observed) CGC becomes another route for improving performance. Possibilities include:

1. Refining classic state-saving incremental network algorithms where call-graph redundancy has already been observed.
2. Reexamining inefficient AI architectures for reusable redundancy in control-flow.
3. Developing new and efficient persistent interactive processes by starting from precise, inefficient programs that have sufficient control-flow redundancy for the CGC transformation to succeed.

5 Conclusion

The Call-Graph Caching program transformation is simpler than many modern compilation techniques. Nonetheless, CGC can mechanically transform programs having sufficient control-flow redundancy into highly efficient incremental counterparts. CGC's chief practical use is in rapidly specifying (and testing) complex network algorithms as simple programs in ordinary programming languages. This was demonstrated with the RETE matching example.

Intelligent systems, unlike fully general computation, must often rely on viable sequences of actions in order to solve their problems. To the extent that such systems exploit redundancies in their control (i.e., sequencing) decisions as they evolve over time, CGC can help in the analysis and implementation of this aspect of intelligence. We therefore suggest that the caching and reuse of call-graphs could prove applicable to a broad range of problem areas and techniques in AI.

Acknowledgments

Jaime Carbonell provided much assistance in the initial formulation of these ideas. Peter Lee, David Steier, and Milind Tambe also contributed to the ideas' evolution.

References

[Aho, 1983] Aho, A.V., Hopcroft, J.E., and Ullman, J.D. *Data Structures and Algorithms*. Addison-Wesley, Reading, Massachusetts, 1983.

[Aho, 1986] Aho, A.V., Sethi, R., and Ullman, J.D. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, Massachusetts, 1986.

[Baer, 1980] Baer, J. *Computer Systems Architecture*. Computer Science Press, Rockville, Maryland, 1980.

[Carbonell, 1983] Carbonell, J. G. Learning by Analogy: Formulating and Generalizing Plans from Past Experience. In R. S. Michalski, J. G. Carbonell and T. M. Mitchell (editors), *Machine Learning, An Artificial Intelligence Approach*. Tioga Press, Palo Alto, CA, 1983.

[Doyle, 1979] Doyle, J. A Truth Maintenance System. *Artificial Intelligence*, 12:231-272, 1979.

[Ferrante, 1987] Ferrante, J., Ottenstein, K.J., and Warren, J.D. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319-349, July 1987.

[Forgy, 1979] Forgy, C.L. *On the Efficient Implementation of Production Systems*. PhD thesis, Department of Computer Science, Carnegie Mellon University, February, 1979.

[Futamura, 1971] Futamura, Y. Partial evaluation of computation process -- an approach to a compiler-compiler. *Computer Systems Controls*, 2(5):45-50, 1971.

[Hoover, 1987] Hoover, R. *Incremental Graph Evaluation*. PhD thesis, Cornell University, May, 1987.

[Knuth, 1973] Knuth, D.E. *Fundamental Algorithms*. Addison-Wesley, Reading, Massachusetts, 1973.

[Laird, 1986] Laird, J. E., Rosenbloom, P. S. and Newell, A. Chunking in SOAR: The Anatomy of a General Learning Mechanism. *Machine Learning*, 1, 1986.

[Minton, 1988] Minton, Steven M. *Learning effective search control knowledge: An explanation-based approach*. PhD thesis, Carnegie Mellon University, February, 1988.

[Miranker, 1987] Miranker, D.P. *TREAT: A New and Efficient Match Algorithm*. PhD thesis, Columbia University, January, 1987.

[Mitchell, 1986] Mitchell, T. M., Keller, R. M. and Kedar-Cabelli, S. T. Explanation-Based Generalization: A Unifying View. *Machine Learning*, 1:47-80, 1986.

[Paige, 1982] Paige, R., and Koenig, S. Finite Differencing of Computable Expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402-454, 1982.

[Perlin, 1988a] Perlin, M.W. Reducing Computation by Integrating Inference and User Interface. Technical Report CMU-CS-88-150, Carnegie Mellon University, Pittsburgh, PA, June, 1988.

[Perlin, 1988b] Perlin, M.W. Transforming Programs into Networks: Call-Graph Caching, Applications and Examples. Technical Report CMU-CS-88-202, Carnegie Mellon University, Pittsburgh, PA, December, 1988.

[Pugh, 1988] Pugh, W.W. *Incremental Computation and the Incremental Evaluation of Function Programs*. PhD thesis, Cornell University, August, 1988.

[Reps, 1983] Reps, T., Teitelbaum, T., and Demers, A. Incremental context-dependent analysis for language-based editors. *ACM Trans. Prog. Lang. Sys.*, 5(3):449-477, July 1983.

[Schank, 1977] Schank, R. C. and Abelson, R. P. *Scripts, Goals, Plans and Understanding*. Hillside, NJ: Lawrence Erlbaum, 1977.

[Stolfo, 1982] Stolfo, S.J., and Shaw, D.E. DADO: A Tree-structured Machine Architecture for Production Systems. In *Proceedings of National Conference on Artificial Intelligence*, pages 369-388, August, 1982. AAAI.

[Winograd, 1983] Winograd, T. *Language as a Cognitive Process, Volume I: Syntax*. Addison-Wesley, 1983.