Experiments with Applicative Updating: Practical Results

Frances E. Hunt

Received July 1987; Revised December 1987

Large scale computing applications often consist of calculations that are repeated for many sets of input data. If the variance between the data sets is small, there may be portions of the computation which are not affected by the changes in the input values. The effort required for such systems can be reduced with efficient methods of recomputation. This paper presents an approach for efficient recomputation over function graphs. A change to an input value of a previously computed function initiates a process of retraction along the data paths dependent on that input. The retraction process consists of first tracing the paths from the input to the output, noting along the way that a change is impending, then tracing back from the output to the input, removing the previously computed values from the intermediate nodes. The modified input value is then released into the function graph, and the retracted portions of the graph are recomputed. Throughout this process, all nonretracted output values are an accurate reflection of the current input values. A retraction mechanism that allows efficient recomputation increases the usefulness of functional programming systems. We present the results of some experiments showing the time taken by the recomputation for two different types of problems. We propose the characteristics of function graphs for which this technique will be most effective.

KEY WORDS: Functional programming; demand-driven evaluation; incremental recomputation; applicative updating.

1. INTRODUCTION

As computing systems become more powerful, the classes of problems for which they are utilized become more complex. Programs often consist of large computations that are repeated each time there are modifications to

1 Department of Computer Engineering and Science, Case Western Reserve University.
the input data. The effort required for these programs would be reduced by an efficient mechanism for recomputation. An example of a process which would benefit from such a mechanism is VLSI circuit simulation, where it has been estimated that 90% of the transistors in a VLSI chip remain unchanged (do not require re-solution) in an average time step during the timing analysis.\(^1\)

We approach the problem of efficient recomputation by restricting the problem domain to functional programming systems. These systems are characterized by the lack of variables (i.e., locations in the storage of a machine, thereby giving it a "state"), and by the principle of referential transparency which ensures that multiple activations of a function with the same input values each time will result in the same output value each time. A program in a functional programming language is expressed as a function from its inputs to its outputs with no reliance on the notion of machine state.

Directed graphs may be used to describe functions and give the programmer a visual aid in understanding the data dependency and data flow characteristics of a program. Arcs of the function graphs represent the paths along which data flow between operators (and function invocations), which in turn, correspond to the nodes of the graphs.

Functional programming systems lend themselves naturally to implementation on distributed architectures. The lack of machine state allows any function which has its inputs ready to be executed in parallel with any other function(s) with all inputs available. An example of a distributed architecture is the Applicative Multi-Processing System, AMPS,\(^2\) a precursor to REDIFLOW.\(^3\)

An example of a functional programming language is the Function Equation Language FEL.\(^4\) The underlying graphical form of this language is FGL (Function Graph Language),\(^5\) which serves as the native language for AMPS. The AMPS architecture provides a multi-processor environment with a demand-driven evaluation scheme which executes FGL programs by means of simulated graph expansion and reduction. The language FEL (and its graphical counterpart FGL) will serve as a model for the concepts to be presented in this paper.

Since functional programming systems do not, in general, incorporate the notion of variables ("cells"), problem areas which involve history-sensitive computations, such as databases, require special programming approaches (e.g., stream-based values). Applying a single transaction to a database results in a new version of the database. The application of a stream of transactions to a database produces a stream of modified databases, each the accumulated result of the previous transaction(s). Since each new version of the database is a copy of the old database with the