Overview of Rediflow II Development†

Robert M. Keller  
Quintus Computer Systems, Inc.  
Mountain View, CA 94041

Jon W. Slater  
ESL, Inc.  
Sunnyvale, CA 94088

Kevin T. Likes  
University of Utah  
Salt Lake City, UT 84112

Abstract A status report is provided on the design of Rediflow II, a proposed multiprocessor architecture based on graph reduction. We emphasize recent work on hardware design to support graph reduction, communication, and load balancing, as well as indicating an evaluation model oriented toward larger grain computation than in our previously-reported work.

Introduction

Rediflow is a proposed multiprocessing system based on the concept that parallelism should be achieved in as transparent a fashion as possible. This is accomplished by using functional languages as a base for programming, normal order evaluation, and a dynamic load balancing technique [Keller 84a-b, 85a-b, Lin 85]. It is the most recent in a line of development based on graph reduction which began with work reported in [Keller 79]. The reader is invited to review the papers mentioned in this paragraph to motivate the work, as we do not do so here.

Parallel normal order evaluation is implemented by suspending and resuming functions based on the availability of arguments to the functions. The advantages to normal order evaluation it facilitates asynchronous parallelism, only the code necessary for evaluation is executed, and a natural and efficient implementation of streams and other infinite data structures is supported [Keller 85a].

Normal order evaluation may be implemented using a "graph reduction" approach, in the sense that nodes representing unevaluated expressions are replaced by their value. However, the approach being used currently does not take this idea to its ultimate limit, as was proposed in [Keller 79]. Rather, graph reduction is used only selectively, with sequential code being used where graph reduction is not really needed. This dichotomy is achieved by compile-time preprocessing. The sequential code model also appears to provide a more natural approach toward the inclusion of sequential processes communicating via streams ("Kahn processes"[Kahn 74]) than did an earlier proposal for Rediflow [Keller 84a].

The "II" in the title indicates that, in the past year, three performance improvements have been made to the Rediflow design: the Redilisp language, an approach to graph reduction called PSCED, and the Redilink architecture [Slater 86]. This paper gives an overview of these developments. Like its predecessor, Rediflow II employs a dynamic load-balancing mechanism which serves the purpose of evenly distributing the available work among the processors, using the gradient model for task distribution [Lin 85]. It also uses saturation control in an attempt to retain locality of migrational tasks [Keller 84b].

† This work was performed in part while the authors were resident at the University of Utah, where it was supported in part by a grant from the IBM corporation. We also wish to thank Sperry, Inc. for providing a prototyping facility and parts for the prototype of the Redilink intercommunication switch module.
Redilisp

Redilisp is a Lisp dialect used to program Rediflow. It descended from our earlier FEL [Keller 82], but has Lisp, rather than "English" [Landin 64], syntax. It is a lexically-scoped language with higher-order functions and procedures, with emphasis on a declarative programming style. Parallel evaluation occurs by default, rather than by required introduction of special constructs, but the latter may also be introduced by the programmer for greater control. An objective of Redilisp is to allow the programmer to integrate functional, imperative, and object-oriented styles of programming into one language. As such, Redilisp provides destructive operators, however the user is encouraged to program in a fully functional subset.

PSCED

In order to improve performance over pure graph-reduction, a PSCED approach has been adopted in the new simulator and architectural specification. This also allows an efficient method for the integration of sequential code. This idea is similar to the Multilisp implementation and the SECD-m effort [Halstead 86, Abramsky 85]. In this scheme, we can achieve normal-order evaluation using what seems to be a much more efficient approach than our former, more-interpretive, graph reduction model. With this decreased complexity, it becomes possible to construct support hardware for handling run time function demands and suspend processes.

Landin's original "SECD" abstract machine derived from four registers in an abstract virtual machine, called "stack", "environment", "control", and "dump" [Landin 64]. We began using the modified name "SCED" as a more pronounceable acronym ("sked"), which evolved to "PSCED" (still pronounceable as "sked") to emphasize the added feature of parallel evaluation, and also to suggest a process identifier component. Meanwhile, the stack component has become a register file, for greater efficiency in accessing local values, but the component is still identified as "S". The PSCED approach of multiplexing among various states is consistent with the multiple register windows being proposed for certain RISC architectures [Katevenis85]. Likewise, we have made an effort to keep the complexity of the architecture down to a small number of machine instructions, and the heavy use of vectors has been helpful in this regard.

As mentioned above, the components of a "state" of the machine are

- **P** The index of the process (for descriptive purposes).
- **S** A register file for holding temporary values. Every S is fixed in its number of registers at process-creation time. We currently assume no upper bound on the size of S among processes, although this might have to be modified for certain processor architectures.
- **C** A Control string, which specifies a sequence of machine instructions to be executed. In the current implementation, this is a linked list of instructions. New control strings are installed as the result of certain instructions, such as those for application and conditional branching.
- **E** An environment, which is an implementation of a mapping from variable identifiers to values. E is implemented so that adding new environment layers, as necessary when creating new functions, is easy to effect. In a parallel model such as this one, it is also desirable to achieve a high degree of sharing and locality of access. Such considerations led us to choose a linked-list of vectors as the environment implementation. Each vector is a "frame" which comes into existence as the result of an application or related construct. For example, the parameters of a procedure are constructed within such a frame, which becomes the top layer of the environment when the procedure code is installed.

Although the method described imparts a sequential component to the access of a frame, such components are optimized by storing (pointers to) selected frames within local registers. This capability was part of the motivation behind use of a register file, rather than a stack.

- **D** The Dump is used as a means for saving state for later return. Effectively the dump embodies a continuation, and can be used to implement continuation objects in the language.