ADVANCES IN PARALLEL COMPUTING

A Research Annual

VOLUME 2 • 1992

ADVANCES IN PARALLEL COMPUTING

A Research Annual

Editor: DAVID J. EVANS
Parallel Algorithms Research Centre
Department of Computer Studies
Loughborough University of Technology

VOLUME 2 • 1992

JAI PRESS LTD

London, England *Greenwich, Connecticut*

LIST OF CONTRIBUTORS

Barbara Bleck	Department of Informatics Justus-Leibig University of Giessen Giessen, Germany
Calvin Ching-Yuen Chen	Department of Computer Science University of North Texas Denton, Tex., U.S.A.
Sajal K. Das	Department of Computer Science University of North Texas Denton, Tex., U.S.A.
David J. Evans	Parallel Algorithms Research Centre Department of Computer Studies Loughborough University of Technology Loughborough, England
A. M. Davies	Proudman Oceanographic Laboratory Bidston Observatory Birkenhead, England
R. B. Grzonka	Sowerby Research Centre Computational Physics Department British Aerospace (Dynamics) Ltd Filton, Bristol, England
G. Isern	School of Computing University of Venezuela Caracas, Venezuela
Henner Kroger	Department of Informatics Justus-Leibig University of Giessen Giessen, Germany

D. G. Maritsas	Department of Computer Engineering an Computer Technology Institute University of Patras Patras, Greece
G. M. Megson	Computer Laboratory University of Newcastle-upon-Tyne England
E. Montagne	School of Computing University of Venezuela Caracas, Venezuela
George T. Papaspyropoulos	Department of Computer Engineering an Computer Technology Institute University of Patras Patras, Greece
R. K. Sen	Department of Computer Science Hampton University Hampton, Va., U.S.A.
C. V. Stephens	NERC Computer Services Bidston Observatory Birkenhead, England
R. Suros	School of Computing University of Venezuela Caracas, Venezuela
P. Willet	Department of Information Studies University of Sheffield England

PREFACE

The demand for more powerful computers has increased unabated since the inception of the early von Neumann computers. This is undoubtedly due to the universality of the design of the general purpose digital computer and its widespread applicability to the scientific, technological and commercial problems which are encountered in present day society.

This need for additional processing power is likely to continue within the foreseeable future because of the increased complexity and sophistication of the software systems we use and expect nowadays. End-user environments based on graphics and agile windowing techniques, desk-top publishing, database applications and the use of expert systems and artificial intelligence in knowledge-based applications have all assisted in this growth. Further, fourth generation software tools provide increased user efficiency but do so by the liberal use of computational power.

The benefits of general purpose parallel architectures are all exposed in the papers presented in this volume. These can be summarized briefly as:

Cost: Since parallel machines can be constructed from multiple, standard components and offer an economical way of providing large amounts of processing power, they suffer less degradation in response time as user demand increases.

Growth: A system based on multiple components can be expanded simply by the addition of further components providing an economical and less disruptive upgrade evolutionary route with the additional benefits of fault tolerance if any of the processors fail.

The chapters in Volume 2 of *Advances in Parallel Computing* again cover a wide spectrum of parallel computing issues ranging from parallel searching in databases, parallel simulation, numerical and non-numerical parallel algorithms and dataflow, cellular and systolic parallel architectures.

Finally I should like to thank the contributing authors for their prompt and cooperative support and my secretary Mrs. Judith Poulton for her skillful organization in making this publishing venture a painless and worthwhile task.

David J. Evans
Series Editor

CONTENTS

APPLICATIVE CACHING AND A DATA FLOW COMPUTER

R. K. Sen

ABSTRACT

Applicative caching typically reduces reevaluation of functions in a recursive program. This can in effect be beneficial in reducing resource demand in a dataflow parallel computer. Here a discussion of implementing applicative caching in a list processing oriented dataflow computer has been presented. A stream based computation has been assumed and the necessary functions for applicative caching in this machine have been devised.

1. INTRODUCTION

Conventional programming languages and von Neumann computers have developed closely. On the other hand, at the beginning of the popularity of applicative or functional languages, first introduced in 1965 by McCarthy [32] and Landin [31], efficient computing support was not available.

With a wider usage of computers and the urgent need for better methods of software engineering, several bottlenecks of the von Neumann computer became explicit [13]. The operational semantics of programming languages associated with the von Neumann model promoted multiple assignment of

Advances in Parallel Computing, Volume 2, pages 1–22
Copyright © 1992 by JAI Press Ltd
All rights of reproduction in any form reserved.
ISBN: 1-55938-274-0

values to expressions. Although initially a useful notion, it causes difficulty in having a simple method for program verification and design [27].

The main idea of applicative programming, that programs can be built entirely from equations defining functions, began with Lisp. Lisp pioneered applicative programming and included some non-applicative features like assignment statements and eventually object-oriented programming [41]. Nevertheless, during the past two decades research in applicative programming has not been lacking, resulting in the design of languages such as Miranda by Turner [38].

Applicative programming provides a better mathematical basis for verification of programs [20]. This is because of the inherent cleaner semantics. It does not support multiple assignment thereby assuring that the value of an expression remains unchanged whenever it is used. This is called 'referential transparency'. Moreover, there is a growing body of knowledge on structural properties of programs together with the algorithmic aspects that has resulted in a renewed interest in the use of applicative programming today [20].

The basic mechanism of computation for processing applicative language program is known as reduction. Models of reduction based computation has been developed following Church's lambda calculus and Curry and Fey's scheme for single argument, which can itself be a function, function evaluation (combinator theory) [15].

There have been attempts to implement reduction based computation on von Neuman architectures. SKIM II is an efficient implementation of Turner's combinator reduction machine [39]. Kieburtz, in 'The G-machine' describes an implementation of the architecture of a reduction machine that uses supercombinators (programmable combinators) [23]. The G-machine, an abstract model for evaluating functional language program, was defined by Johnsson and Augustsson [12, 25]. The G-machine is an evaluation model for a complier for a dialect of ML [33] called Lazy ML. The abstract model represents an expression as a graph and through successive transformations, or reductions, modifies the graph until its form is that of a fully evaluated result.

A growing interest in parallel computing today, due to the dramatic reduction in cost of building machines with several processors together with the realization that the physical limit of computing speed can only be overcome by having several computers work on the same problem, has motivated workers in designing novel methods for exploiting parallel computers. The availability of considerable knowledge has led people to construct such computers by putting together multiple copies of the von Neumann computer. On the other hand, following a 'software first' approach alternative parallel architectures have also been designed [24].

Implicit parallelism in applicative programs is easier to detect than in conventional languages [19]. Considerable research in implementing applicative languages for parallel von Neumann architectures has been carried out [22].

In these approaches parallel computation is based on a process (task) model [21] and tends to favor large grain parallelism because of the overheads of the tasking mechanism. Moreover, the parallel version of programming language imposes the need for the programmer to know the operational aspects of the parallel computer which becomes intractable when hundreds of processors are employed.

Kennaway and Sleep [30] developed a general architecture model that consists of a number of processing elements and two pools of waiting and selected tasks. Selected tasks from the waiting pool are put into the selected pool to be chosen by the processing elements. The processing elements pick up subexpressions (selected tasks) for evaluation (reduction). In this model the conventional program counter of the von Neumann architecture is replaced by a set of tasks picked up by the processors for execution. The model specifies two alternative implementation schemes for beta substitution (a step in combinator reduction)—use of pointers or making copies of expressions.

Kenneway and Sleep categorized the new architectures as pipelined ring architectures, packet circulation ring architectures, physical tree architecture, and sequential reduction machines. The basis of the categorization are Manchester Dataflow machine [40] and MIT Static Dataflow machines, early ALICE implementation [39], FFP machine [39], AMPS [28], DDMI [39], GRIP [39] and SKIM, G-machine.

The static dataflow architecture first proposed by Dennis [34] with VAL [1] as the basic programming language mainly attempted to offer increased throughput for scientific applications with high inherent parallelism. Subsequently, methods of handling issues pertaining to function invocation, recursion, and structure handling were incorporated in the static dataflow architecture [17]. A VAL Interpreting Machine (VIM) that interprets the data flow program graphs has been built. It provides support for data structures like streams using early completion data structures and suspensions, and tail recursion is used to avoid unnecessary retention of activation records when iterations are to be implemented. In the dataflow architecture a computation is represented as a program graph. Every time a function is applied a copy of the function graph is incorporated into the calling program graph. This implies that for a computation involving several function calls a graph structure of unmanageable size may result. The dynamic dataflow architecture called Tagged Token Data Flow by Arvind [2, 7–9] instead allows different function invocation to be represented in terms of tagged (also called coloured) tokens thereby avoiding the cost of interpreting large program graphs. The dataflow architecture developed at Manchester University and known as the Manchester Dataflow architecture subscribes to the dynamic dataflow concept [39].

Dataflow machine can be used to implement graph reduction by having suitable compilation techniques to transform a functional program to dataflow graphs so that it implements graph-reduction semantics. The fundamental thrust is to implement graph-reduction by avoiding the need to build too large dataflow graphs. This is because building graphs is fundamentally an interpretive mechanism, and consequently expensive compared to compiled code. Work in compiling functional programs into Id (the dataflow language for the tagged token architecture at MIT) programs are being investigated by Nikhil [35]. It requires some extension to both the programming language Id as well as the dataflow architecture to support synchronizing updates to *I* structure memory (the structure memory of the tagged token architecture).

Amamiya et al. designed a dataflow machine for a list processing machine [4]. This architecture is basically data driven and can perform list processing in a highly parallel and pipelined fashion. The architecture supports pure Lisp, where except for **cons** no other function has side-effect. The **cons** is implemented in terms of lenient evaluation and called **lenient cons.** This allows partial evaluation of function. Partial function evaluation is related to the idea of reduction-based computation. A dataflow machine with partial function evaluation capability implicitly possesses the same abilities as reduction computers. Amamiya's architecture thus supports a form of reduction suitable for applicative programming.

Due to a highly parallel computing environment, dataflow computers, in general, adhere to the single assignment rule. Thus dataflow languages are applicative in nature. A salient feature of a data flow computer is that it has a decentralized control which is defined dynamically according to the availablity of data that enable, and hence fire instructions. The execution of a program is carried out in a highly concurrent manner where concurrence may be implicit. According to Sleep and Kenneway's categorization of parallel architectures for parallel reduction, dataflow systems appear towards the extreme end of fine granularity. It may be noted that fine granularity inherent in early dataflow systems were the subject of intense criticism due to the overall demand of resource management. Exposing parallelism tends to increase resource requirement [10]. Automatic program unfolding [5, 8, 39], if unconstrained, may expose far more parallelism than the machine can exploit and may suffer performance degradation. Moreover, unravelling and activation of all possible tasks may lead to deadlock [10]. Methods that can reduce the number of function activations need to be developed.

In an applicative programming environment for a recursive program a method for reducing resource demand involves eliminating recomputation. Keller and Sleep [29] presented syntactic and semantic mechanisms for implementing a scheme by which recomputation can be avoided. This is called applicative caching. Dataflow systems provide an applicative programming environment and recursive application programs are quite common. This prompts the use of applicative caching in a dataflow machine to reduce resource demand in general. The main issue in applicative caching is the efficient implementation of the cache. This entails the use of suitable data structures and enforcing applicative environment. As the dataflow systems do not inherently support the caching mechanism the problem of providing applicative caching with primitive dataflow instructions is not straightforward.

In order to study the essential issues involved in implementing applicative caching in a dataflow computer, the architecture of Amamiya seems to be the most appropriate. Amamiya's model supports list data structures, an essential component of a stream cache—the simplest applicative cache. The main reason for choosing Amamiya's architecture is the ease of encoding stream handling programs. Although other dataflow architectures handle structures quite efficiently, the basic list processing capabilities of this architecture seem more attractive for the purpose.

2. A DATA FLOW MACHINE FOR LIST POSSESSING

In a dataflow machine all operations can be executed concurrently when all of their operands are present (concurrency). The result of an operation depends on the values of its operands and is not affected by execution of other operations, and the history of the execution (functionality). The execution sequence of operations is decided based only on data dependency, and hence a centralized sequence control is unnecessary (distribution). By list processing one essentially means non-numeric applications. To apply dataflow mode of computation in non-numeric applications it is necessary to solve the problem of structured data manipulation with particular reference to lists. For preserving functionality the single assignment rule need to be enforced. This entails structures needing to be copied every time they are used.

Amamiya's dataflow architecture attempts to execute Lisp mainly because it provides the succinct basis for list processing. The brief description of this architecture is given below.

The architecture has five components: control modules (CMs); an inter-CMs communication network (CN); structure memory (SM); an instruction network (IN); and a result network (RN) as shown in Figure 1.

The CM is the kernel of the data flow execution. It consists of a memory that stores machine instructions. It also has the fetch mechanism for enabled instructions. The CN connects the CMs with each other. The SMs store the structured data such as lists. The IN and RN connect the CMs and SM.

List structured data are stored in the SM and their pointers flow in the machine as tokens. Here, by-reference mechanism [3] is used and a cell having attribute of a variable, list or pointer flow in the system.

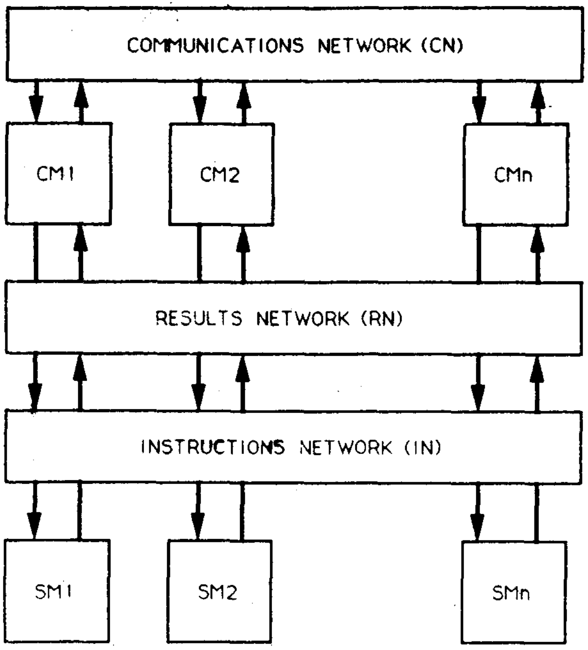

Figure 1. Amamiya's dataflow machine organization.

The architecture supports pure Lisp primitive instructions like **Cons, Car, Cdr**, **Atom** and **Equal**. All instructions except **Cons** all are side-effect free. **Cons** gets a new cell after communicating with the resource manager and returns a pointer to it. The car and cdr fields are written leniently. Hence the name **lenient cons**. There is no updating of a field and hence no multiple assignments. List processing is regarded as memory operations which mainly contains readout operations.

Memory contention and side effects are serious for exploiting parallelism in list processing. The memory contention is resolved by dividing the structure memory into banks. For each bank there are primitive operators, thereby enhancing parallelism among operations. An outline of the structure memory is shown in Figure 2.

Partial function body evaluation is inherently supported by this architecture. Partial function-body execution refers to the fact that the execution of a function is started whenever one of its argument value is obtained, and the execution of the function-body proceeds partially every time the value is passed in. Any function invocation overhead, such as assigning a function-body to a processing element, passing arguments and passing information regarding destination of a value etc., can be overlapped with the evaluation of arguments. Some functions can proceed with the execution using a part of

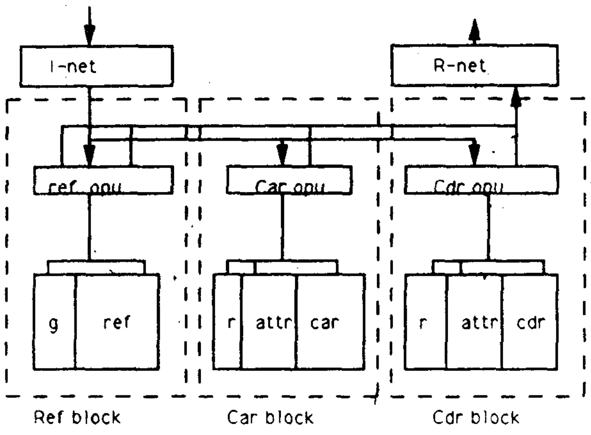

Figure 2. The structure memory.

the arguments, for example a **cond**. In addition, the function results can be returned to the caller as soon as they are obtained. This overlapping results in a more efficient function evaluation environment.

In order to process recursive function call and to support a higher order function the list-processing dataflow machine adopts a dynamic function-linkage mechanism using the link and rlink dataflow operators. Function bodies are shared using colored token approach. A token packet is expressed as

⟨instance-name, data⟩ destination

The call node is initiated by a signal token, and creates a new instantiation name. The call node creates the function body if it does not exist. Otherwise it creates only an instantiation name. When the function-body is ready to run, the token with new instantiation name is sent to *link* and *rlink* nodes of the dataflow graph. The *link* and *rlink* nodes are interface operators that creates packets for sending or receiving arguments and results. Each *link* node creates a packet with the new instantiation of the argument token and passes it to the input port of the function-body. The *rlink* node creates a packet with the new and current instantiation and the destination name along with the name of the output port of the function-body. Note, that this implements partial evaluation of function because whenever a result value (partial) is obtained the function need not wait for other results.

When list data structures are created with **cons** or **append** and then consumed dynamically, the consumer function, must wait until the producer creates the entire list. However, if each element of the list is returned every time it is generated, the execution which uses it can proceed without waiting. This results in the producer and consumer to overlap. For this the concept of lenient cons is introduced.

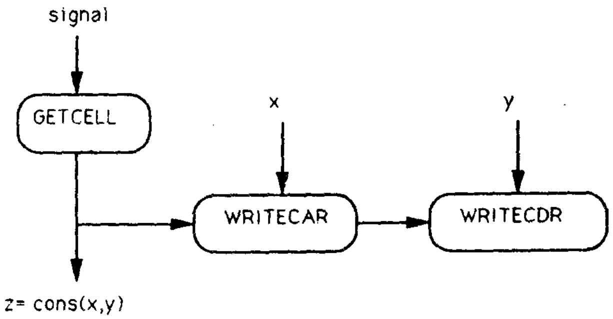

Figure 3. Cons operator.

The **cons** operator is decomposed into three primitive operators—getcell, writecell and writecdr, as shown in Figure 3. Figure 4 illustrates the data cell structure. Each data cell has a garbage tag, car-ready and cdr-ready tags. The garbage tag is used for reclamation of unused cells. The last two tags control the read access to the car and cdr fields.

The getcell is initiated on the arrival of a signal token, which is delivered when the new environment surrounding the cons operation is created. This operator creates a new cell and resets both ready tags to 'off' in order to inhibit access. It then sends the new cell address to the writecar, the writecdr and the nodes waiting for that cons value. When the ready tag is 'off' it means the value has not yet arrived.

The writecar (writecdr) writes the operand value *x*(or *y*) into the car field (or cdr field) and sets the ready tag to 'on' to allow read accesses. Although a cell may be referenced before a value is written (in such case the ready tag is 'off') the reference is suspended until the writing is completed (i.e. the ready tag is set 'on').

Lenient cons implements stream processing feature. It is effective in exploiting function-level parallelism. As can be seen it allows functions of a tree evaluation type (e.g. recursive divide and conquer algorithm), or a linear

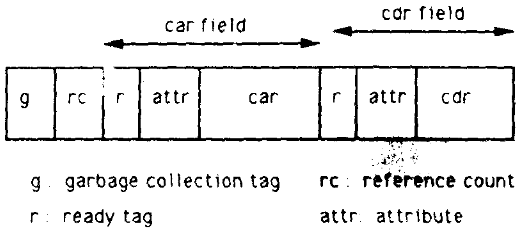

Figure 4. A memory cell.

evaluation type (e.g. simple tail recursion) to be c‸mputed in linear time in parallel. Since a function does not wait for the whole list to be completely computed it results in better resource utilization than in a situation without lenient cons.

Once a cell is created by a cons operation and a value is written into it the contents are never modified. Other operations (other than cons) only read from it. A new cell may be created at any location. The Car(Cdr) block has ready tag, attribute field and car(cdr) pointer field. The ready tag indicates whether the data has arrived or not. The attribute field indicates whether the data is an atom (number or literal) or pointer (to an SM cell address). The Ref block contains a garbage tag and reference count field. The Car(Cdr) block operation unit performs read/write operations from/to the car(cdr) field. The Ref block operation unit controls reference count management and performs the getcell operation. A lenient operation is decomposed into the operations of getcell, writecar, writecdr and gate (that finds lenient cons result), each of which is executed in the Ref, Car and Cdr operation and the functional unit of the CM respectively.

As list processing is considered as essentially a memory operation, efficient memory operations is a key element. Memory contention and side-effect are difficulties when attempting to exploit parallelism. In this architecture pipelined and parallel memory access is available.

The copying overhead is minimized by distributing access to lists. New cells are generated in such a way as to distribute cells uniformly in SM banks. The uniform distribution is a result of language features like blocks, functions, etc. Further details are available in [4, 5].

Independent memory access and pipelined list processing between execution control and memory operation can be achieved. The contention can be avoided by dividing the SM into many banks and including an operation unit for each memory cell. This is like the logic in memory concept. The structure memory is regarded as a part of the functional unit. It executes list operations except **Atom** and **eq** operations. The SM bank has three indpeendent blocks, Ref, Car and Cdr blocks. This allows primitive operation level overlapping. Each block has a specialized operation unit.

Efficient garbage collection in a data flow system is a difficult task. The pointers to list data may be scattered in many parts of the machine such as Instruction Memory, Operation Units, Networks, etc. It is difficult to extract an active cell without suspending execution. A reference count method is used in garbage collection, as there is no chance of getting any circular list. There is basic machine instructions that help garbage collection. The reference count field of a cell is explicitly undated by the manager corresponding to the increment and decrement operations generated by the compiler.