A CELLULAR, LANGUAGE DIRECTED COMPUTER ARCHITECTURE

(Extended Abstract)

Gyula A. Magó

University of North Carolina at Chapel Hill

Abstract If a VLSI computer architecture is to influence the field of computing in some major way, it must have attractive properties in all important aspects affecting the design, production, and the use of the resulting computers. A computer architecture that is believed to have such properties is briefly discussed.

I. Introduction

One would expect that microelectronics, having affected other application areas, should influence the way we design computers, and such sentiments have already been articulated in the literature [5]. So far, however, all ideas on how to organize a very large number of logic circuits (or for that matter, microprocessors) into a computing machine have had serious drawbacks in one or more respects. As a consequence, computer design has not yet been able to exploit successfully the ever-increasing capabilities of semiconductor technology.

In the first part of this paper, we argue that a VLSI computer architecture, in order to have a major impact on the computing field, (1) should be cellular in design, and (2) should have its design be guided by consideration of an appropriate language. The remainder of the paper outlines the major characteristics of one computer architecture which has these properties. A detailed description of this architecture is being published elsewhere.

II. Desiderata for VLSI Architectures

We start this sequence of arguments by acknowledging that for the forseeable future, only large production volumes can make it economically attractive to manufacture whatever VLSI chips we need for our computer designs. From this point of view, all conceivable computer designs requiring Q amount of hardware for their realization (measured in any meaningful way) can be linearly ordered: at one end of the scale one finds designs that require the whole Q amount of hardware to be designed into different, one-of-a-kind chips, whereas moving towards the other end one finds designs (we shall call them cellular) in which only a fraction, Q/n amount, of the hardware has to be designed into chips, and the computer is obtained by taking n copies of these chips and interconnecting them in some regular fashion. If one can create sensible computer deisgns that are cellular, then, all other things being equal, increasing n will make the situation more and more attractive by decreasing the number of necessary chip designs and increasing production volume of each remaining chip type.

We infer from the above that a VLSI computer architecture should have a cellular structure, that is, it should be obtained by interconnecting simple component processors (we shall call them cells) in a regular fashion. (At this stage we can conceive of a cell occupying more than one chip, or alternatively a chip containing many cells. In this latter case, the chips should also be connected in a regular fashion.) We shall take cellularity also to imply that in an organizational sense such an architecture should be expansible, and that there should be only physical limits to the size of such machines.

In a cellular computer, the cells are expected to operate concurrently, carrying out component computations, and the problem of designing such a computer is ultimately a problem of algorithm decomposition: how to prescribe for each cell what to do so that the totality of their behaviors adds up to the required global behavior, such as the execution of a particular user program. Since computations in a cellular computer must be able to unfold over potentially very large collections of cells (thousands or millions of them), often in a data dependent manner, it would be most natural for the details of the above mentioned decomposition to be worked out at run-time. The remaining question is how is the decomposition determined and at what cost?

The inherent nature of a cellular computer--it is expansible, and arbitrarily large collections of cells must be able to operate efficiently--appears to exclude any reliance on some central, global agent, e.g., a "master" computer, that would inspect the user program, decompose it, and then determine what each cell should do. The alternative to such a global control is to let individual cells cooperate in decomposing the user program into parts (small or large), carry out partial computations, and (still using only limited local information) cooperate in combining the partial results to obtain the desired overall result. The more straightforward, direct, and simple this process can be made, the more efficient the resulting computer can become. In the extreme case, the cellular network can be thought of as directly executing an appropriate user language. It appears most promising then to require that a VLSI computer architecture be language directed, meaning that it be able to execute directly a programming language. With this approach, the programming language specifies the global behavior the cellular network is to exhibit, which in turn can be used to derive the behavioral and structural specifications for the individual cells. The above requirement really urges an integrated, top-down design for both the software and the hardware of such a cellular computer, in sharp contrast with the usual practice of separately designing software and hardware for uniprocessors. (It should be mentioned that Dennis [3] and other advocates of the data flow approach have also argued for language directed architectures in the context of parallel, though not necessarily cellular, computers.)

In addition to the above considerations, an acceptable computer design for VLSI technology must meet many other criteria, such as ease of programming, efficient execution of user programs, and cost-effectiveness in an overall sense. In [4], a computer architecture is described that seems to meet these criteria. In this paper, we explore what a cellular, language directed architecture can offer by examining how some of these criteria are met by the architecture described in [4].

III. Main Characteristics and a Brief Evaluation of a Cellular, Language Directed Architecture

The architecture described in [4] is capable of directly executing certain applicative languages (e.g., reduction languages [1], FP and FFP systems [2]) recently introduced by Backus. In these languages, programs are expressions, and a program is executed by evaluating its expression. A particular class of expressions, called applications, specifies computations. Because these languages allow all innermost applications to be evaluated simultaneously, they are able to express parallelism in a very natural fashion.

The architecture of [4] is obtained by interconnecting cells in the form of a full binary tree, and additionally connecting the leaf cells into a linear array. The leaf cells (i.e., those in the linear array) are all identical and are called L cells. The remaining cells are different from the L cells but identical to each other; these are called T cells. The L cells store the expression to be evaluated (although they also have some processing capabilities), whereas the T cells are used for routing and processing purposes. In order to make the cells as small as possible, each L cell is used to store a single symbol of the source program.

A user program and its data form a single expression in these languages; this expression is a linear string of symbols, and is mapped onto the L array, one symbol per L cell. The innermost (hence executable) applications are contained in disjoint segments of the L array, and disjoint portions of the T network are used to evaluate (i.e., execute) them. Consequently, this cellular architecture is capable of unbounded parallelism on the source language level by simultaneously evaluating all innermost applications, the only limitation being the size of the L array.

In addition, there is parallelism below the level of the source language: since cells of L hold only single symbols of the source language, even the simplest primitive operations of the source language, such as adding two numbers, involve the cooperation of several cells of the network. This low-level parallelism makes feasible direct implementation of complex primitive operations of the language, such as vector operations and typical associative processor operations.

Since the above architecture is unusual, one can evaluate it or compare it with other architectures only on the basis of global, overall criteria, for ultimately its viability will be judged on the basis of such criteria. We consider four main categories of properties of the architecture.

1. Programmability

1.1. The applicative languages on which the architecture is based support a functional style of programming. The advantages of such a style over that permitted by conventional languages are discussed at length by Backus in his 1977 Turing Award Lecture [2].

1.2 These languages allow and encourage the introduction of a high degree of parallelism into the programs without requiring detailed planning on the part of the programmer (for example, the programmer does not have to initiate and terminate execution paths explicitly).

1.3 Execution times of programs can often be predicted analytically (see 2.). As a result, when a program is being written, efficient execution can be a criterion, and time-space tradeoffs are commonly available.

1.4. The primitive operations of the language are not wired into the cells, and consequently the architecture allows great flexibility as far as the primitives of the applicative languages are concerned, even within a single machine.

2. Efficiency of Program Execution

The processor is designed to be able to execute with acceptable efficiency any program written in an applicative language. Most important in this respect is that the processor initiates and terminates the execution of all innermost applications (i.e., execution paths) with little or no overhead (beyond the work required by individual innermost applications), thereby adjusting easily to the widely varying degrees of parallelism found in most programs.

The execution time of an innermost application can be predicted analytically. Some primitive operations require O(logN) time, where N is the number of cells in L. More complex primitives, such as reversing a list of an arbitrary number of elements, require time proportional to the number of data items that have to be moved. Based on the execution times for primitives, upper and lower bounds can be derived for the execution times of programs written in the applicative language; this is especially easy for programs with few data dependent branches. Many important classes of algorithms have been and are being analyzed, such as vector and matrix algorithms, algorithms for dense and sparse linear systems, Fourier transforms, solution methods for partial differential equations, and multidimensional search problems. These analyses show that not only can the machine execute sequential programs in an acceptable manner, but also the massive parallelism both on and below the source language level can result in greatly increased execution time efficiencies. The details of these analyses are to be published elsewhere.

3. System Software

Since the cellular network directly responds to an applicative language, the need for many typical components of present-day uniprocessor and multiprocessor software has been removed, and their function taken over by hardware. For example, in the presently envisioned system, there is no need for

(a) a compiler (there is only need for a very simple preprocessor to change the external representation of programs to an internal one),
(b) a software interpreter,
(c) memory management software,
(d) software to detect parallelism in user programs,
(e) software to assign processors to tasks.

Since the cost of software is the dominant component in the cost of present-day general purpose computers (and not only is it expensive to develop such software, but in addition it has to be stored in the machine, and it ties down hardware resources while executing), the possibility of trading complex system software for cellular hardware is a very welcome development.

4. Hardware Related Issues

Some of the advantages of cellularity have already been discussed under the desiderata.

4.1. Both the L and T cells are rather small, mainly because they need only a few dozen registers as local storage. As a result, as VLSI technology advances, whole subtrees of cells may be put on a single chip. (Any cell of T or L has to be connected to at most three other cells in the network. Furthermore, any complete subtree whose leaves are L cells communicates with the rest of the processor through at most three such points.)

4.2 Since a collection of user programs is again an expression in the applicative language, the cellular processor needs no extra machinery to deal with several user programs at a time. As a result, increasing the size of the processor can yield arbitrarily large throughput values without any need for reprogramming. (In fact, all other things being equal, throughput increases at least as fast as N/logN, where N is the number of L cells.)

4.3 Actual throughput values are influenced by many design parameters, such as speed of logic circuits and widths of data paths but most importantly by the parallelism present in user programs.

For programs that offer high degrees of parallelism, throughput values can become very high. As an example, for N=1 million L cells, 5,000 - 10,000 MIPS can realistically be estimated as the highest attainable, where an innermost application is counted as an "instruction." (Note that if an innermost application involves a complex primitive, such as finding the maximum of an arbitrary number of data items, what we count as one instruction may be equivalent to the computational work performed by many uniprocessor operations.)

References

[1] J. Backus, Programming Language Semantics and Closed Applicative Languages. IBM Research Report RJ1245, Yorktown Heights, N.Y. July 1973.

[2] J. Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," Communications of the ACM, Vol. 21, No. 8, August 1978, pp. 613-641.

[3] J. B. Dennis, "Programming Generality, Parallelism and Computer Architecture," Information Processing 68, North-Holland Publishing Co., 1969, pp. 484-492.

[4] G. A. Magó, "A Network of Microprocessors to Execute Reduction Languages," International Journal of Computer and Information Sciences (to appear)

[5] I. E. Sutherland and C. A. Mead, "Microelectronics and Computer Science," Scientific American, Vol. 237, No. 3, September 1977, pp. 210-228.