Function Distribution in Computer System Architectures

Harold W. Lawson

Universidad Politecnica de Barcelona*

NOTE: This is an electronic version (2000-01-14) of an invited paper appearing in the Proceedings of the Third Annual Symposium on Computer Architecture, Clearwater, Florida, January 1976. The author's current affiliation is Lawson Konsult AB, Lidingö, Sweden. He can be contacted at bud@lawson.se.

ABSTRACT

The levelwise structuring and complexity of a computer system is presented informally and as a general model based upon the notion of abstract machines (processors), processes and interpreters. The important domains of the computer architect are considered along with historical perspectives of certain stimulae and decisions that have affected the distribution of functions amongst the various levels of computer system implementations.

Keywords: Computer Architecture, Computer System Complexity, Computer History.

1. Introduction

In the early days of digital computers, the stratification of computer systems was, on the surface, quite simple. Two main levels were apparent, namely, hardware and programs (e.g. software). Growth in the sophistication of the application of computers to new areas, changing physical technologies, the man-machine interface, the economics of computer usage, production and investment, inherent and created complexities, and finally a better understanding of the structuring of hardware and software have all influenced the levelwise structuring of the computer systems architectures as we view it in the mid 1970's. It would be difficult to get an agreement on precisely how many levels exist (or should be described) in a modern computer system. It would even be difficult have agreement on the question: What is a modern computer system? In any event for purposes of this paper we shall begin with the leveling structure as introduced by Lawson and Magnhagen (1). This leveling is at least representative for supporting implementation of the "Third Generation Computer System" environment as presented by Denning (2).

2. Informal View of Function Distribution

The above mentioned leveling structure appears in Figure 1. The lowest level is purely physical whereas the higher levels are all organizational, realized by hardware algorithms or program algorithms (or combinations). While the various levels may vary in content, one thing is clear; each level (l) uses level (l-1) as a "tool" for level (l) composition. In the next section we formalize, as a general model, the composition of levels and inter-level relationships. Presently, we shall consider the implications of Figure 1 in an informal manner.

---

* Former address: Mathematics Institute, Linköpings Univerity, Linköping, Sweden

Computer System Architecture

Figure 1

First let us consider the question of complexities. In the paper by Lawson and Magnhagen (1), the notions of horizontal (intra-level) and vertical (inter-level) complexity were introduced. That is, there is an inherent complexity within each level and created complexities due to the mapping of level upon level. Many examples of created complexities can be sighted. Some will be presented later in the paper.

The levels of Figure 1 are shown as an inverted pyramid to illustrate that "in general" the lower the level, the fewer people involved in designing, and producing the tools of the levels, whereas, as we go toward higher levels the greater number of people involved in using lower levels as tools. That is to say, for example, that more people use integrated circuits than design and produce them, or, hopefully, more people use computers than design and produce computer hardwares and softwares. This relationship is an important factor in developing an informal notion of the cost function of vertical complexities to be specified shortly.

As a rather practical matter, there appears to be a few unwritten principles that one can extract from the short history of designing and producing computers that have been related to many project developments and individuals making decisions.

Principle 1:
"If you cannot solve the problem, give it to someone else".

Frequently the passing of the problem goes upwards in the leveling structure, thus increasing the number of people affected by the decision. Complexities become magnified.

Principle 2:
"If you cannot select an alternative, provide all possibilities".

That is, give your "users" general purposeness so that they can do anything they wish. While this may be useful at certain levels of the distribution, it can be disastrous at other levels and contribute heavily to magnified complexities. A simple measure of the application of this principle is the quantity of shelf-space for documentation of all possibilities (assuming the level is fully documented).

Of course these two principles are based upon the fact that the designers and/or implementers first realize that they have a problem or are aware of the alternatives. A lack in these directions can cause even greater magnified complexities in the eventual system.

Principle 3:
"If a tools exists that can be adapted to perform a function; use it".

This decision, normally by those responsible for the economics of system implementation, has frequently been catastrophic. The wrong tools are used to implement a level (l+1), thus forcing many complications upon the implementers and propagating complexities upwards. A professional plumber does not use carpenters tools to fix leaks in the plumbing, nor does he use general purpose tools to rectify a specific problem. Amateur plumbers due to the usage of the wrong tools or tools of too high a degree of general purposeness can create floods. Several analogies applied to computer system architectures may come to the reader's mind in which floods have been created.

Principle 4:
"If a design mistake is discovered during implementation, try to accomodate the mistake instead of fixing it".

This is, of course, an economic question of project investment that must be made by responsible project management in relationship to schedule slippage, penalties for late deliveries, etc. It is rare that the implementation procedure is reset to a point where the mistake can be corrected. Frequently, the end cost has been higher than the cost would have been for mistake correction. Many design mistakes have wound up being presented as "system features".

All four of the above principles have resulted in intra and inter level complexities. Now for the informal cost function. It should be obvious that if complexities are passed upwards, towards the users, the cost of complexity increases since the cost must be repaid for each usage. Whereas if complexities are passed downward, it is probable, but not always guaranteed, that total costs will be decreased.

To illustrate some concrete examples of the passing of complexity let us consider the following:

Passing complexity downwards:

- simplified job control language
- vertical (highly encoded) microinstruction formats
- tagged data and program object types
- virtual storage management by lower levels

Passing complexity upwards:

- user selection of a multiplicity of file accessing techniques

• complicated code generation decisions left to compilers
• using unsuitable microarchitectures for emulating foreign target systems

The author does not offer any rule-of-thumb for deciding upon the correct structuring of the levels of a computer system architecture. A quote from Horning and Randell (3), with which this author is in complete agreement, explains why:

"The appropriate use of structure is still a creative task, and is, in our opinion, a central factor of any system designers responsibility".

The author does venture to say that some of the key factors are the selection of an understandable amount of semantic content at each level and the appropriate balance of special purposeness versus general purposeness. As for the number of levels; that is system dependent. A process control system does not require as many levels as a multi-user access system where the users are performing different types of data processing.

3. A General Model of Function Distribution

Horning and Randell (3) have pointed out that processes can be used to model parts of a computer system. In this section, we build upon this notion. The main extension is upon the specification of program parts, which we consider as being composed of the executions sequence of programs utilizing one of more "abstract machines".

Two types of abstract machines form the notion of what we shall call "processors". One type of processor can service sequential processes, the other type of processor exists to take care of process interactions. The latter of these may be viewed as the controller of asynchronous concurrent events and for this type of processor we assume the notion of "monitors" as presented by Hansen (4) and further developed by Hoare (5) and Hansen (6). The important part is that the processors perform algorithms leaving out the notion as to whether they are hardware, software or combinations thereof. A processor can, for example, be an arithmetic and logical unit as well as a resource allocation monitor.

A processor which we shall refer to as (pr) responds to a program (p) and an instance of execution of (p) upon (pr) yields a process (ps).

We may state formally

$$ps = f(pr,p) \qquad (3.1)$$

A process is a function of a processor and a program.

If we think then that each level other than the lowest level in Figure 1 is a processor, then we can construct a computer system model generation formula.

$$pr(i) = ps(i) = f(pr(i-1), p(i-1)) \; i = 1,2,\ldots,n \qquad (3.2)$$

where n = number of levels above the physical circuit level of Figure 1 (i.e. the number of organizational levels). That is, a processor is defined as a process which is developed as a function of a lower level processor and its "program" where program does not only mean

stored program, it can mean simple sequencing. Note that this formula also serves as a formal definition of the notion of an interpreter or interpreter hierarchy.

Given this notion, we can now state what the role and responsibility of a computer architect is in terms of producing a computer system design.

> To find convenient mappings between each (pr, p) pair that permit convenient realizations of all levels and to distribute functions according to some goals amongst the various levels. Furthermore, to seek to minimize both intra and inter level complexities in all parts of the system.

In respect to system complexity, we can consider the following formalization as a measure of complexity (c).

$$c = \sum_{i=1}^{n} i^{k} \cdot \sum_{j=1}^{m} s_j \qquad (3.3)$$

Where: n = number of levels in the system
k = exponential growth of complexity between levels
m = number of potential state transitions within a level
s = a vector containing a measure of complexity of each potential state transition

We note that the complexity is weighted by the level, thus reflecting the increasing cost of complexity discussed in the previous section. It is obviously difficult to measure the complexity of each transition in a uniform way, however, it is indeed related to the semantic content of discrete activities and the potential interactions of the activities at each level whether they be timed sequences through a logic chain, micro instructions, target instructions, procedures or a run in a multi-phase application package.

The programming language concurrent Pascal as presented by Hansen (6) contains some interesting features for controlling the complexity of interrelationships of processes and monitors. The process and monitor functions to be performed are declared as abstract types rather than as absolute objects. Further, a concise statement of real instances of processes and monitors giving each instance only specific "access rights" to other processes and monitors is made by a global declaration. The interconnections within an access graph are made quite explicit. Other interconnections are automatically excluded by the programming language translator. This type of thinking would be extremely useful in constructing processors at all levels including micro code, Computer Aided Design, LSI layout, etc.

Having now considered the levelwise structuring and implication in a general model form, we shall finish by considering some historical events that affected the distribution of functions and various (pr,p) mappings.

4. Historical Perspective

We shall consider some, but certainly far from all the events that have caused changes in the distribution of functions in computer system architectures.

Let us first briefly consider the physical component technology changes since we shall later concentrate on the organizational aspects. That is, what we have built and how we have structured the systems we have built.

The first major step above the early use of relay and vaccum tube technologies for logic realization, was the invention and use of the transistor. On the memory side, the first major step was the invention of the magnet core storage. We have gone through several generations of transistors realized in different types of technologies with astounding success in miniaturization, packing densities and speed increases to the point where the transistor currently forms the basis for most memories and logic.

We shall not belabor these obvious physical changes, however, it is worth noting that the physical side has had an important impact upon what we have built. Up until 1970's when mini and later LSI microprocessors became increasingly important, we viewed the central processing units as well as the memories as expensive items.

Due to the early high costs, it is not hard to see why the first stored program[*] type architecture was so readily accepted. That is, uniform program – data stores, and a simple accumulator oriented processor logic.

Early attempts to move away from the first stored program type architectures generally resulted in very complex, expensive hardware structures for processors such as the Burroughs B5000 (8, 9). In any event, it was clear that certain architects such as Barton (10) did not consider the first stored program structure as the best solution that was cast in a Bible of stone. It is interesting to note that these early departing architectures would with today's integrated circuit technologies (and computer aided design techniques) be many orders of magnitude simpler to realize.

As the functional requirements for processors grew from simple functions to include features such as floating point arithmetic, decimal arithmetic and input/output control, it became obvious to Wilkes (11) that this increase in complexity required a better organizational hardware implementation technique. Wilkes thus proposed new levels in the function distribution, namely, microprogram architecture and microprograms.

On the software organizational side, it became obvious in the late 1940's and early 1950's that constructing programs directly in machine language was a nuisance, therefore, assembly languages were conceived. Further, the notion of utility programs and subroutines for computations as well as computer management functions like input/output codes evolved to give economic and psychological advantages to the field of programming. These were the humble beginnings of system softwares.

The scope of the system software level increased with the proposal by Hopper (12) to use higher level programming languages. It is interesting to note that in the early developments of programming languages that some implementers realized that the machine for implementation, usually following the early stored program computer concept, was not a convenient machine for the mapping of programming language programs. Therefore, the idea

[*] This concept is usually referred to as the von Neumann (7) type of architecture, but Professor Maurice Wilkes has informed the author that several people including the group at the Moore School of Electrical Engineering, University of Pennsylvania, contributed to the concept. Professor Wilkes proposes EDVAC type computer.

of inventing a pseudo machine for the language and constructing an interpreter program of the pseudo machine became popular.

During the mid 1950's many arguments occurred concerning the pro's and con's of using higher level languages versus assembly code on the first hand and, on the other hand, pseudo machines and interpreters as an implementation technique as opposed to compiling code. Unfortunately, compiling won out on the efficiency of object program arguments. Thus, since the mid 1950's we have spent large sums of money reconstructing compilers to generate codes for new hardwares. Many times it is difficult and sometimes impossible to decide upon the "best code" to generate for particular programming language features. With pseudo machines, there is usually only one best mapping. But at that time it would have been difficult to propose constructing more hardware-like pseudo machines. Machine architectures and microprogramming, of course, have now evolved to the point where this not only possible but economical, for example, see Wilner (13) and Lawson and Malm (14).

In the early 1950's a divergence in computer architecture occurred based upon the end use of systems. That is, processors, their related memories and input/output systems were made more special purpose and oriented towards scientific or commercial markets. Using this design strategy, certain (pr,p) mappings were better for certain classes of applications. However, it was frequently required to supply compilers of scientific languages for commercial oriented processors and vice-versa. These compiler mappings in many instances were extremely difficult. Like using carpenters tools to fix leaks in the plumbing.

One of the main arguments for moving to the System/360 type architecture in the early 1960's was to create a more general purpose architecture thus cutting down on the proliferation of different system softwares that arose from having several special purpose architectures. One system for all users, and, as was attempted, a uniform programming language for all, namely, PL/I. One of the main problems was that by adding general purposeness, the mappings ps = (pr,p) for most all areas, Fortran, Cobol, PL/I, etc. to System/360 became more difficult. This resulted in very complicated schemes of compiler code generation and optimizations of System/360 codes as well as providing the economic need for several levels of support. Did this reduce software proliferation?

While one may question to soundness of the target system architecture of System/360, it is important to note that this was the first wide scale use of Wilkes microprogramming concept. Various System/360 processors were designed and programmed to be compatible interpreters of the System/360 architecture. The microprocessors were not general purpose microprocessors, but designed for the special purpose of implementing System/360. In any event, since they were programmable they were used to produce "emulators" of IBM second generation equipment. Many of these ps = (pr,p) mappings, where pr was the microprocessor, were extremely painful and represent prime examples of the plumber using carpenters tools and creating floods.

On the operational (access) side of computers, it became evident in the mid 1950's that one user at a time exploiting such an expensive resource was uneconomical. Therefore, the idea of spooling programs, supervisors, schedulers, etc. eventually led to the notion of operating systems. As the reader is well aware, these properties led to profound changes in the way people use computers and led to special requirements upon many levels in the computer system architecture. Denning (2) does an excellent job of extracting the principles of operating systems as we know them in third generation computer systems.

In the mid 1960's several people were speculating that microprogramming would become an important media for aiding in programming language and operating system requirements. See, for example, Opler (15), Lawson (16), Wilkes (17) and Rosin (18). However, the necessary step to accomplish this was to have a more general purpose microprocessor. Such microprocessors started to be developed in the late 1960's, see Lawson and Smith (19) and have resulted in several interesting designs including those mentioned earlier, namely Wilner (13) and Lawson and Malm (14).

While redistribution of functions to a more general purpose microprocessor may have certain appeal in reducing complexity, it is a realistic fact of life that the huge investment in computer products (hardwares and softwares) of typical third generation products has slowed down the marketing of products based on different concepts for target system architectures and instruction sets. At least one system architecture by the Amdahl corporation has used redistribution of functions to better use Large Scale Integration parts in perpetuating System/360 and 370 type architectures.

One prime example of the redistribution of functions from software to hardware has been the wide spread use of the concept of virtual memory. This concept has helped us solve many complex problems with program overlays and file management by utilizing levels lower than the system software level. It is interesting to note that the idea and implementation existed for many years before it was widely implemented. See Kilburn, Edwards, Lanigan and Sumner (20).

On the hardware side, many manufacturers during the 1960's after strong resistance from "hardware artists" have accepted the use of Computer Aided Design techniques for accomplishing printed circuit board layout. This type of resistance is very similar to the argument of "programming artists" utilizing assembly language versus the utilization of higher level languages. That is, I can always do the job better without automation tools. But we may ask; at what global cost? In any event, just as with higher level languages, CAD can be extremely effective and certainly permits us to more quickly and clearly realize more sophisticated architectures at the lower organizational end of the function distribution.

In the 1970's we have experienced an awareness (certainly not too soon) of the complexities of using computer systems. Much is now written on structured programming and programming style, see for example Dahl, Dijkstra and Hoare (21), Weinburg (22) and Denning (23). These writings have influenced application and system software construction and should in the future hopefully start to influence lower levels of the organizational end of function distribution.

Hansens Concurrent Pascal (6) as mentioned earlier contains some interesting features for structuring and controlling "component" interrelationships. This work builds on that of Wirths Pascal (24). It is clear in these two cases that there is an attempt to redistribute functions in the distribution of Figure 1. By having the programming language translator "guarantee" that invalid relationships cannot exist, we can avoid constructing lower level hardware and/or microcode solutions to checking for invalid operations.

It is interesting to note that the concepts of structured programming and improved programming style require a restrictive type of programming, thus reducing semantics. Could this be a good general principle? Special purpose tools, with restricted semantics, can perhaps be used to solve, in a better manner, special problems!!

While this historical treatment has certainly not been exhaustive, it is hoped that the reader can see, in terms of the earlier informal and general model presentations of structuring and complexity, some of the implications of the events that have happened in the brief history of digital computer design, implementation and utilization.

Conclusions

It would be difficult to terminate this paper without saying a few words about possible directions for function distribution in the future. One thing is clear, the range of the distribution that a computer architect must cover has increased. He must be intimately familiar with all levels with the exception of the details of the physics of logic realization by particular technologies. Further, he must be prepared to specify an appropriate number of levels to solve the program at hand and assure that the ps = (pr,p) mappings are realizable, economic and convenient to utilize.

Current trends towards LSI logic parts at low costs due to high production volume will undoubtedly have a great impact upon the organizational levels, see Fuller and Siewiorek (25). Just make sure that the parts selected represent the correct tools to do the job, otherwise, look out for floods. The mass production of large scale integration parts may be insensitive to the complexity of the logic, but users of the components are not insensitive.

Due to the fact that processor physical structures have reduced in cost, we can expect more dedicated systems on the one hand and an attempt to utilize many processors in a distributed manner on the other.

As a practical matter for many areas of computer usage, we are saddled with our history due to large program investments. There will certainly be an impetus to seek solutions to new architectures that can accommodate this software investment. Most likely, these solutions will be combinations of several levels in the distribution of functions.

Acknowledgment

The author wishes to thank Professor Maurice Wilkes for reviewing this material, particularly to verify the historical aspects.

References

(1) H.W. Lawson and B. Magnhagen. *Advantages of Structured Hardware*, Proceedings of the Second Annual Symposium on Computer Architecture, Houston, Texas, January, 1975.
(2) P.J. Denning. *Third Generation Computer Systems*, Computing Surveys 3, 4, 1971.
(3) J.J. Horning and B. Randell. *Process Structuring*, Computing Surveys 5, 1, 1973.
(4) P.B. Hansen. *Operating System Principles*, Prentice Hall, Englewood Cliffs, NJ, 1973.
(5) C.A.R. Hoare. *Monitors: An Operating System Structuring Concept*, Communications of the ACM 17, 10, October 1974.
(6) P.B. Hansen. *The Programming Language Concurrent Pascal*, IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, June 1975.
(7) -. *Von Neumann's Collected Works*, A.H. Taub, Ed., Permagon, London, 1963.
(8) Burroughs Corporation. *B5000 Reference Manual*, Form 200-21014, Detroit, 1961.

(9) W. Lonergan and P. King. *Design of the B5000 System*, Datamation, 7, 5, May 1971.
(10) R.S. Barton. Ideas for Computer System Organization: A Personal Survey, Software Engineering, Vol. 1, Academic Press, New York, 1970.
(11) M.V. Wilkes. *The Best Way to Design an Automatic Calculating Machine*. Manchester University Inaugural Conference, 1951.
(12) G.M. Hopper. *Compiling Routine A-0*. Unpublished documentation, May 1952.
(13) W.T. Wilner. *Design of the B1700*. Proceedings of the FJCC, Anaheim, CA, 1972.
(14) H.W. Lawson and B. Malm. *A Flexible Asynchronous Microprocessor*, BIT 13, 2, June 1973.
(15) A. Opler. *Fourth Generation Software*, Datamation 13, 1, 1967.
(16) H.W. Lawson. *Programming Language Oriented Instruction Streams*. IEEE Transactions on Computers, C-17, 1968.
(17) M.V. Wilkes. *The Growth of Interest in Microprogramming: A Literature Survery*, Computing Surverys, 1, 3, 1969.
(18) R.F. Rosin. Contemporary Concepts in Microprogramming and Emulation. Computing Surverys 1, 4, 1969.
(19) H.W. Lawson and B. Smith, *Functional Characteristics of a Multi-Lingual Processor*, IEEE Transactions on Computers, Vol C-20, July 1971.
(20) T. Kilburn, D.B.G. Edwards, M.J. Lanigan and F.H. Sumner. *One-Level Storage System*, IEEE Transactions, EC-11, April 1962.
(21) O.J. Dahl, E.W. Dijkstra and C.A.R. Hoare. *Structured Programming*, Academic Press, London, 1972.
(22) G.M. Weinburg. *The Psychology of Computer Programming*, Van Nostrand Reinhold, 1971.
(23) P.J. Dening, ed. *Special Issue: Programming*, Computing Surverys, 6, 4, 1971.
(24) N. Wirth. *The Programming Language Pascal*, Acta Informatica, Vol. 1, No. 1, 1971.
(25) S.H. Fuller and D.P. Siewiorek. *Some Observations on Semiconductor Technology and the Architectures of Large Digital Modules. Computer*, October, 1973.