Forgotten Ideas in Computer Architecture: It's Time to bring them Back!

Dennis, Jack

Abstract—*Future computing systems will comprise multiple processors to achieve high performance. The issues surrounding the programming of such systems offer a major opportunity for innovative research in computer system architecture. The ideas of global virtual memory, functional programming, and data-cognizant control, which have long been ignored by mainstream computer architects, are relevant to addressing and solving the problems of formulating a satisfactory programming model for parallel computing, and developing computing systems that support such a model and the modular construction of software. The Fresh Breeze Project offers one approach to this goal.*

Index Terms—*Computer system architecture, functional programming, concurrent processing, parallel programming models, software modularity, virtual memory.*

1. Introduction

A revolution is in the making in the architecture of computer systems. Computer processing chips are being built with from two to hundreds of processor cores. Yet there is no generally accepted approach to writing programs that can exploit the parallelism offered by these devices. There is no agreement about addressing the "memory wall" – the problem raised by the fact that the time to fetch data from off-chip memory is now hundreds of times greater than the time to execute an instruction. A major weakness in the way parallel programs are currently written is that they are not "composable" – it is not possible to use a parallel program as a component in building a larger program without understanding and revising internal details of the component program – a violation of basic principles of modular programming.

This article reviews the field of computer system architecture based on my experiences as a participant in the field since the 1950s. It emphasizes concepts that, although they have had a degree of respect and acceptance in the past, have been ignored for many years by computer architects. Some of these concepts have a strong relevance to the current crisis in computer architecture: global virtual memory, functional programming, and data-cognizant control. Many projects and research efforts have contributed to a wealth of knowledge about these forgotten ideas – a gold mine waiting to be tapped and exploited. The time is right to look at them again.

Manuscript received January 2007. This work was supported in part by the U.S. National Science Foundation under Grant CNS-0509386.

J. Dennis is Professor Emeritus at the MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, Massachusetts, USA (e-mail: dennis@csail.mit.edu).

2. Virtual Memory

In the history of computer architecture no innovation has had the impact on programming that virtual memory and paging has had. Two ideas have contributed to the evolution of virtual memory. The better known is from the Atlas machine[15] which introduced the use of a relatively small main memory divided into fixed-size page frames loaded from a backing store on demand by a run-time control program. The need to load a new page was detected by failure to match the page number in a set of tag comparators, one for each page frame.

The second idea is the concept of segment, a contiguous area of memory space allocated to hold a particular data or code object. In the Rice University Computer, *code words* were introduced[14] that defined the limits of a data segment so run-time routines could move segments about in the main memory and provide contiguous space for new data segments.

The use of paging has become universal in commercial computers, and has made the programmer's task far easier than before. Before paging the programmer had to decide which data and code objects should reside in the main memory through the course of program execution, and include code to explicitly move pieces of code and data between main memory and the "backing store" (tape, drum, or disk) to meet the needs of the computation. (These programming gyrations have been reintroduced with the advent of the IBM Cell Processor.)

Paging implements a "virtual memory" in that the programmer writes as though programs and data are laid out in an address space much larger than the physical main memory of the computer. The paging mechanism operates so the "working set" of code and data objects resides in main memory, and those objects not

currently in active use are automatically relegated to the backing store. This makes it possible for the programmer to practice the most fundamental technique of modular programming -- calling a procedure module without the caller being concerned about the modules' ability to access the data it needs to perform its function. In other words, paging hardware has provided the support needed for dynamic management of the memory resource, a feature that relieves the programmer of the task of planning the static allocation of memory. This has become an accepted standard of computer system engineering.

Segmentation has had a less felicitous history. The concept was introduced to permit a data structure to grow (within a data segment) according to the needs of the specific computation being performed, not constrained by the presence in memory of other data objects. The idea of code words from the Rice University machine was adopted in the descriptors of the Burroughs B5000 system[7]. This feature allowed the automatic operating and scheduling program of the B5000 to load into main memory just those code and data segments needed to run the current constellation of user jobs. This opened up a new degree of power in modular programming because data objects (segments) of arbitrary size could be passed to and from a called procedure module.

The Multics computer system[3] carried the idea of segmentation further. Its memory management hardware provided each user process with the capability of linking to as many as 2^{18} data or code segments, each of which could hold as many as 2^{18} words of information. A segment could be created of arbitrary size, and could grow or shrink as computation progressed. Segments were divided into fixed-size pages to make storage management simple, and so only the active pages of segments need be kept in main memory. This effectively implemented a virtual memory space sufficiently large that there was no need for a separate means for accessing "files": A file in Multics is just a segment, so accessing a "file" was just like accessing data in "memory". These features of Multics gave its users a level of modular programming capability that did not exist before, and has not been seen since in commercial systems.

Multics did not however provide the most ideal environment for modular programming for at least two reasons: First, because the linking of segments was done independently for each process, addresses (e.g. representing data objects) could not be passed between processes without losing their meaning. Moreover, if the owner of a process would like it to run continuously, linking to new objects as necessary, it was necessary to invoke mechanisms to unlink unneeded segments from the process – a violation of modularity principles. More relevant to the theme of this article, Multics did not support any means for encapsulating a computation by several processes into a module that could be used as a component of a larger program. Encapsulation could not be applied to parallel programs. Overcoming these defects in Multics remains a challenge for future system architects.

3. *Functional Programming*

The main ideas of functional programming became of interest to computer scientists in the 1960s through the efforts of such workers as Landin, Strachey, Burge[5], and others, largely motivated by the mathematical logic of Alonzo Church[8]. Functional programming is a "declarative" style of expressing computation, in contrast to the imperative style characteristic of conventional programming languages. A declarative style expresses "what is to be done" rather than "how it is to be done". This means that functional languages do not use labels for memory locations, but only as names referring to values. The values to be calculated are defined in terms of intermediate values, which are, in turn, defined by other defined values, and so on, back to the input values for the calculation. Recursion is used as a natural way of defining sets of values generated by a series of similar computation steps, instead of iteratively "updating" loop variables. In this way, functional programs are much closer to the conventional expression of an algorithm in mathematical notation than is a program in, say, Fortran or C.

Early workers in functional programming concentrated on the expressiveness and semantics of functional notations for programs. The functional programming community also studied type systems for programming languages such that one could guarantee that a successfully compiled program would never encounter a type error during execution. These ideas regarding types have finally made it into popular languages such as Java.

One important property of functional programming languages is the absence of "side effects" of procedure calls. In procedural languages such as Pascal, C, and their successors, the procedure body may contain assignments to variables external to the procedure. (In some languages such assignments are limited to assignments to procedure arguments passed "by reference".) In a "pure" functional language such assignments cannot be made. This property gives purely functional languages several important benefits. One is that of "referential transparency" – an expression of the language has the same

meaning (represents the same value) regardless of where it appears within the body of a function. A related property is that of "implicit parallelism" – any set of separate subexpressions of a function body may be evaluated in parallel; that is, their order evaluation does not affect the result. Programming languages that permit side effects forego this useful property, and it is necessary to employ sophisticated analysis techniques to discover sections of code that may be correctly executed concurrently.

An often-mentioned limitation of functional programming is that side effects are used in programs to accomplish tasks that are not otherwise expressible – tasks involving the notion of "state". One class of such tasks arises in situations where a sequence of values is being processed to generate a result sequence, and each element of the result is dependent on all input values that have been seen so far. A program that performs such a task must maintain an internal state that holds whatever information about past input values is needed to calculate future results. A purely functional language can express such tasks if the power of stream data types[11] is included. So far, important languages such as ML and Haskell have not adopted this enhancement.

4. Data-Cognizant Control

Data flow concepts of computation led to a flurry of interest in the 1970s and 1980s. Data flow offers an appealing idea of parallelism: the availability of data initiates action on the data. In particular, the joint presence of two operands can signal the time to apply an operation to two input values. This seems conceptually simpler than the software synchronization gyrations needed to start a third process just when two others have completed subsidiary tasks. Indeed, data flow principles are applied in superscalar processors to indicate when an instruction may correctly issue to a function unit.

The right combination of control and data activated computation in a programming model can yield simplicity, elegance, and other benefits such as avoiding explicit locking and the problems of cache coherence. Two examples of combining control and data flow are part of current practice: One is message-passing where in the reception of a message a new activity is initiated; the other is the remote procedure call that starts a new process and provides its initial data in a single action. The concept can be made more universal with better architectural support.

5. Modular Construction of Software

Modularity in software concerns the ability to assemble large software systems from separately designed software components. Some principles for the practice of modular software construction were laid down many years ago. Parnas[17] identified "information hiding" as an essential element, and Boebert[4] emphasized context independence and the importance of using hierarchy. In Modula 2, Niklaus Wirth[18] introduced the practice of separating module inputs and outputs and precisely specifying their data types.

A weakness of these early ideas about program modules is that they failed to deal with data objects represented as linked structures in memory. Although Parnas understood the desirability of encapsulating representations and operations on data structures, it was only with Simula 67 and Clu that these ideas became incorporated in programming languages, and it became clear that a garbage-collected heap is essential to achieve the full benefit of support for modular software. The advent of Java makes these ideas finally available in a popular programming language.

Recently this author recognized a Secure Arguments principle[10], which requires that a software module not be able to modify input data in any way that would affect operation of another module presented with the same input concurrently. This is clearly important in a system supporting concurrent execution of program modules, where two modules developed independently might operate simultaneously using the same input object. Yet it is also important in sequential programming because it should not make any difference which of two independent modules is executed first when they share only input data.

A summary of these ideas has been presented as six principles of modular software construction[10]:

> **Information Hiding Principle:** The user of a module must not need to know anything about the internal mechanism of the module to make effective use of it.
>
> **Invariant Behavior Principle:** The functional behavior of a module must be independent of the site or context from which it is invoked.
>
> **Data Generality Principle:** The interface to a module must be capable of passing any data object an application may require.
>
> **Secure Arguments Principle:** The interface to a module must not allow side-effects on arguments supplied to the interface.
>
> **Recursive Construction Principle:** A program constructed from modules must

be useable as a component in building larger programs or modules.

Resource Management Principle: Resource management for program modules must be performed by the computer system and not by individual program modules.

Although it seems obvious that the architecture of a computer system would affect the ability of its users to practice modular software construction, computer architects have paid little heed to these principles. A module must be a unit of software that has well-defined interfaces with other modules and implements functionality independent of the context of use. To satisfy the Resource Management Principle, the system must implement resource management rather than permitting modules to make private decisions about use of shared resources. To satisfy the Data Generality Principle, it must be possible to pass arbitrary data structures held in heap storage as module arguments and results. This implies the necessity of a global address space. The Secure Arguments Principle implies that the language in which modules are written should not permit "side effects" through input data.

6. ARCHITECTURE EVOLUTION

Since the introduction of RISC architecture in around 1980, the main driving forces in computer architecture have been performance of programs written for the sequential programming model, and the ability to execute legacy code. This led to the ever-increasing complexity of processor architecture in a quest to execute more instructions per clock cycle. The added complexity usually increased power consumption; that was not an issue because a system would have only one processor chip and would include lots of memory and communication logic that would still dominate the power equation. In fact, any scheme that would utilize silicon area to achieve even a slight (single-thread) performance gain was worth the investment: register renaming and wide instruction windows to increase the amount of instruction-level parallelism the hardware could discover in the instruction stream; logic to by-pass use of the register file for results that could be immediately used by successor instructions; and mechanisms to ensure that results are not committed too early when instructions are issued out of their order in the instruction stream. The schemes included mechanisms such as branch prediction that permitted the processor to continue executing instructions following a branch instruction even though it might turn out that the branch is taken and the speculative execution has no value. This process of buying small performance gains for significant investments of silicon area has now run out of steam.

Users interested in high performance for scientific computations have, since the "Attack of the Killer Micros"[6], relied on the most economical way of achieving supercomputer levels of performance: connecting large numbers of commercial microprocessor chips together and using message-passing models to write parallel applications. This approach has never been acceptable in commercial data processing because running legacy software packages has been a fundamental requirement. Instead, the client/server model allowed users to spread their workloads over multiple machines without disturbing the underlying software paradigm.

Now the tables have turned. The speed of logic in future silicon devices is no longer expected to increase with each new generation of processor chip. The myriads of transistors it is now possible to fabricate in one chip must be put to better use than heroic attempts to improve single thread performance. Instead, the way forward is through concurrency, using multiple threads of computing to carry out multiple activities simultaneously. Power consumption has also become a big issue. Complex superscalar processors are now seen as an exorbitant waste of power and silicon area for the performance achieved. Using multiple copies of simpler processors is clearly the better choice; the pressing question is how to program them.

7. IMPEDIMENTS TO PARALLEL PROGRAMMING

Much has been made of the difficulties of writing parallel programs. Taking a sequential program and finding the parts that may be executed concurrently is a challenging task for a person, and the search for automatic means for uncovering opportunities for parallel computing has had disappointing results. On the other hand, experience has shown that parallel computers can be successfully used: most computing tasks can be analyzed and programmed to exploit the parallel execution capabilities of multiprocessor systems. However, there are at least four issues: First is the programming effort required to coordinate the activities of multiple threads of computing. Second is the cost in performance due to the additional instructions and processor cycles needed to implement the coordination. Third, the programming of thread coordination is rife with opportunities for errors that can lead to data races and nondeterminate behavior. Finally, data structures with shared access from several parts of a program are often a source of complexity because the program parts are running in different addressing environments and

their interaction requires translation of each communication into the address space of the receiver.

These problems all arise from the absence of a satisfactory *program execution model* for parallel computing. The application program interfaces (API) provided by multi-core chip running conventional operating systems and providing library software for parallel computation violate many principles of modular programming. When one attempts to use a parallel program as component of a larger software unit, one finds that:

- Processors are committed.
- Data structures are distributed.
- Files may be used, but only their names may be used in an interface.
- The procedure call is an inadequate module interface, but the best available.
- Side effects are rampant.
- Naming conflicts arise.

A common attitude is that these difficulties can be overcome through improvements in software: better languages, smarter compilers, and advanced run-time systems. Yet even the vendors of the new multi-core chips admit that new programming models are needed for parallel computing and are looking to the research community for ideas.

Thus there is a new search for better ways of expressing parallelism. However, the search often presumes that good solutions will be found in more advanced and sophisticated software tools. I believe this search is futile because the source of the difficulties is the inadequacies of the application programming interface (API) presented by current multiprocessor systems and their operating software.

8. The Opportunity

The theme of this article is that the difficulties of programming parallel computations can be eliminated by choosing a good programming model and supporting it with matching system architecture.

> **It is possible to construct systems that exploit the parallelism present in applications and honor all six principles of modular software construction … and they can have superior performance.**

A first requirement is that all threads of control use names (addresses) for objects that have the same meaning. A second requirement is that processing resources not be committed by a program module. Processors must be virtualized so they are available to perform any work that is ready to be done. A third requirement is that the means of thread coordination not present the programmer with choices that can result in needlessly nondeterminate behavior. Means that combine data flow and control flow concepts can provide a solution.

The need for a major change is appreciated by leaders in industry. Dr. John McCalpin of IBM has said[16]:

> **"Hardware must support dramatically superior programming models to enable effective use of parallel systems."**

Workers on shared memory multiprocessor systems have long been concerned with the "memory model" presented by their systems to the user. In contrast, I believe that designing a memory model is solving the wrong problem. Computer systems are built to run programs, and it is the program execution model that determines whether a program behaves correctly according to the semantics of the language in which it is expressed. A memory model is relevant only as a component of a program execution model with which it must be consistent.

9. The Fresh Breeze Project

An example of a forward-looking project is the Fresh Breeze Project being pursued by the author and students at the MIT Computer Science and Artificial Intelligence Laboratory. This project is evaluating a multi-core chip architecture that aims to support efficient parallel execution of large programs constructed hierarchically from modular components. A Fresh Breeze computer system will honor all six principles for modular software construction. To achieve this goal the Fresh Breeze architecture will use simultaneous multithreading processors, and provide a global memory consisting of fixed-size chunks that are created and initialized by threads, but become read-only once they are shared with other threads. The Fresh Breeze multithread program execution model is an extension of the Cilk model[12], with hardware-implemented futures to build producer/consumer style compositions of modules, and a special *guard* memory object to provide a transaction capability. A Fresh Breeze program runs with determinate behavior unless it contains uses of guards for transaction processing applications.

The Fresh Breeze architecture provides an alternate machine-level view of computation – one consistent with principles of functional programming: Starting with given data, compute new values and store in fresh memory cells; don't overwrite the given data – it may be in use by other concurrent activities. Achieving this requires the flexible and agile handling of memory and processor allocation that is built into the Fresh Breeze architecture.

Further benefits of the Fresh Breeze system architecture include:

- The Fresh Breeze execution model does not permit creation of cycles of chunks, so reference count garbage collection may be used, realizing rapid, distributed identification of unused memory chunks.
- Conventional files are supported as structured collections of memory chunks, and so may be freely passed between program modules.
- Fine-grain management of threads is supported by an on-chip thread scheduler that provides flexible allocation of processing resources.
- Many functions performed by the kernel of a conventional operating system are performed by Fresh Breeze hardware, eliminating a large source of software overhead.

The Fresh Breeze multithread processor will, we believe, achieve competitive, even superior, performance for the area and power used.

10. Conclusion

The time is ripe for many innovative projects to explore new design spaces for computer architecture. The tools are available for constructing prototypes at modest cost, including the FPGA prototyping board developed by the RAMP Project[1]. A recent Berkeley Report[2] observes that the best choice along the line of simple to complex is not independent of the application domain and context of use, so there are many design points that would be worthwhile to explore. The Fresh Breeze Project makes some radical choices. Other design points may be capable of achieving similar goals.

There is no longer an excuse for computer architecture research to evaluate minimal adjustments to conventional processor designs. Changes are needed in the nature of our computing infrastructure. Let us seize this opportunity to build a new foundation for computing, one that will have the power to change the way we construct software for the better. The coming years should see a blossoming of innovative work in computer system architecture.

Acknowledgment

The ideas presented here have evolved through research efforts with students and colleagues in the Computation Structures Group of the MIT Computer Science and Artificial Intelligence Laboratory, and with professional, academic and industrial colleagues throughout the world.

References

[1] Arvind, K. Asanovic, D. Chiou, J. C. Hoe, C. Kozyrakis, M. Oskin, D. Patterson, J. Rabacy, and J. Wawrzynek. RAMP: Research Accelerator for Multiple Processors - A Community Vision for a Shared Experimental HW/SW Platform. Technical Report UCBCSD-05-1412, University of California, Berkeley, September 2005.

[2] Asanovic, K., et al. *The Landscape of Parallel Computing Research: A View from Berkeley.* Technical Report UCB/EECS-2006-183. University of California at Berkeley, 2006.

[3] Bensoussan, A. Clingen, C. T., and Daley, R. C. The Multics virtual memory. In *Proceedings of the Second Symposium on Operating System Principles*, ACM, 1969, pp. 484-492.

[4] Boebert, W. E. Toward a modular programming system. In *Proceedings of a National Symposium: Modular Programming.* Cambridge, MA: Information Systems Press, 1968.

[5] Burge, W. H. Recursive Programming Techniques. Reading, Massachusetts: Addison-Wesley, 1975.

[6] Brooks, Eugene. Attack of the Killer Micros. Presentation at Supercomputing 89. Lawrence Livermore National Laboratory.

[7] Burroughs Corporation. The Descriptor--a definition of the B5000 information processing system, Detroit, Michigan, 1961.

[8] Church, A.. *The Calculi of Lambda Conversion*. (AM-6: Annals of Mathematics Studies) Princeton University Press, 1941.

[9] Dennis, J. B. Fresh Breeze: A multiprocessor chip architecture guided by modular programming principles. *ACM SIGARCH Computer Architecture News 31*, 1 (March 2003) pp. 7-15.

[10] Dennis, J. B. A parallel program execution model supporting modular software construction. In *Third Working Conference on Massively Parallel Programming Models*, IEEE Computer Society, 1998, pp. 50-60.

[11] Dennis, J. B. Stream data types for signal processing. In *Advances in Dataflow Architecture and Multithreading*. J.-L. Gaudiot and L. Bic, eds., IEEE Computer Society, 1995.

[12] Frigo, M., Leiserson, C., and Randall, K. H. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming language design and Implementation. ACM SIGPLAN Notices 33*, 5 (May1998).

[13] Gosling, J., Joy, B., and Steele, G. *The Java Language Specification.* Addison-Wesley, 1996.

[14] Iliffe, J. K., and Jodeit, J. G. A dynamic storage allocation scheme. *Computer Journal 5* (3): pp. 200 - 209, October 1962.

[15] Kilburn, T., Edwards, D.B.G., Lanigani, M.J., and. Sumner, F. H One-Level Storage System. *IRE Trans. Electronic Computers EC-11,* April 1962.

[16] McCalpin, John D. Perspectives on the Memory Wall. Powerpoint presentation, February 18, 2004. IBM, Austin, TX.

[17] Parnas, D. L. Information distribution aspects of design methodology. In *Information Processing 71.* North-Holland, 1972, pp 339-344.

[18] Wirth, N. The development of procedural programming languages: personal contributions and perspectives. In *Lecture Notes in Computer Science, Volume 1897: Modular Programming Languages.* Springer-Verlag, 2000, pp 1-10.