ADVANCES IN CYBER-PHYSICAL SYSTEMS

Vol. 2, No. 1, 2017

A NEW COMPUTATIONAL MODEL FOR REAL GAINS IN BIG DATA PROCESSING POWER

Conrad S. M. Mueller

Computer Science Department of the University of the Witwatersrand, Johannesburg, 2050 South Africa
Author e-mail: *cmueller@acm.org*

Submitted on 25.09.2017

Abstract: **Big data and high performance computing are seen by many as important tools that will be used to advance science. However, the computational power needed for this promise to materialize far exceeds what is currently available. This paper argues that the von Neumann computational model, the only model in everyday use, has inherent weaknesses that will prevent computers from achieving the envisaged performance levels. First, these weaknesses are explored and the properties of a computational model are identified that would be required to overcome these weaknesses. The performance benefits of implementing a model with these properties are discussed, making a case that a computational model with these properties has the potential to address the needs of high performance computing. Next, the paper presents a proposed computational model and argues that it is a viable alternative to the von Neumann model. The paper gives a simplified outline of an architecture and programming language that express the proposed computational model. The main feature of this computational model is that it processes variables as they become defined. These variables can be processed in any order and simultaneously, avoiding bottlenecks and enabling high levels of parallelism. Finally, the computational model is evaluated against the properties identified as desirable, showing that it is possible to design an architecture and programming language that do not have the weaknesses of the currently dominant von Neumann model. The paper concludes that the weaknesses which limit the performance of current computers can be overcome by exploring alternative computational models, architectures and programming languages, rather than by working towards incremental improvements to the existing dominant model.**

Index Terms: **Big data, Computer architecture, Computational models, High-performance computing, Programming language.**

INTRODUCTION

I recently attended a workshop discussing the computational needs of the European Organization for Nuclear Research (CERN) and the Square Kilometer Array (SKA) for large, data-intensive computations. These needs echoed those expressed in what I consider the “landscape papers” (the views of the Berkeley group [3], the 21st Century study [4] and the US Presidential Science and Advisory recommendations [5]). These papers argue that current technology is inadequate to meet the rapidly-increasing demands for transmitting, storing and manipulating large data sets. The landscape papers call for new models of computation to meet these demands. Even so, research and resources continue to be directed towards incremental improvements in the instruction and address-based architectures that express the dominant von Neumann computational model [3, 6]. Little effort is being directed at conceptualizing alternatives.

Those calling for new computational models, as well as those working to improve the current architectures, have identified concerns with limitations such as the memory wall, excessive power consumption, complexity of programming, reliability, and the cost of hardware development [1–6]. This paper argues that these are symptoms, and not the underlying reasons why the instruction- and address-based architectures cannot meet the demands of large data and intensive computing. This paper explores the inherent limitations of the dominant von Neumann computational model, and explains how an alternative computational model and its expression in an appropriate computer architecture and programming language, can avoid them.

THE COMPUTATIONAL NEEDS

While I had a good idea of what the computational needs were for handling large data, I was surprised by what the CERN and SKA researchers identified as priorities. Very high on their priorities was to limit the cost of hardware, as well as running costs, and optimising the application code and operating system were seen as critical to achieving this. A particular concern was that such optimisation is problem- and technology-specific, resulting in a moving target as technologies change. They were also concerned about power consumption and how “green” the technology is. Concerns with data and instruction bottlenecks did get mentioned in discussing the more technical aspects of parallelism and input and output. The impression I gained from the workshop was that current technology is a long way from being able to address the computational needs of both the SKA and CERN projects.

The landscape papers [3–5] look at the viability of continuing to scale computer hardware. The 21st Century Computer Architecture paper [4] and the Workshop on Advancing Computer Architecture

Research [1] conclude that (1) increasing the power per chip and the reliability are not sustainable; (2) communication between chips cannot cope with increased speeds; and (3) new designs are prohibitively costly. Ideal properties required for high-performance computing are identified as: performance, security, parallelism, improved communication, reduced energy requirements and programmability. Amongst the recommendations are to (1) contain energy requirements by moving from serial to parallel, (2) minimise communication, (3) better manage the memory hierarchy, (4) develop a new programming model that enables better management of resources, (5) revisit the program stack, (6) co-design hardware and software, (7) provide functionality and performance across a wide range of architectures, and (8) improve the verifiability and reliability of both hardware and software. Doesn't this extensive list suggest something seriously wrong with the whole computational model?

The earlier Berkeley paper [3] echoes many of the sentiments of the 21st Century paper [4], however one of their proposals is to focus more on programming.

> *Since real world applications are naturally parallel and hardware is naturally parallel, what we need is a programming model, system software, and a supporting architecture that are naturally parallel. Researchers have the rare opportunity to re-invent these cornerstones of computing, provided they simplify the efficient programming of highly parallel systems [3].*

These papers argue for a programming model that bridges the gap between applications and hardware. They also examine the memory wall and delays in memory access, as well as the need to shift focus from upping the speed of the clock, to greater parallelism. Other important aspects to be addressed are: coherence, synchronisation, programming models that are independent of the number of processors and address a rich set of data types and sizes, resource management, and how to improve the applications code and the operating system.

These kinds of concerns are not new. As far back as 1977, Backus [6] identified the need to liberate programming from the instruction and address-based model and explored potential solutions. He identified what he called the von Neumann bottleneck.

> *The task of a program is to change the contents of the store in some major way; when one considers that this task must be accomplished entirely by pumping single words back and forth through the von Neumann bottleneck, the reason for its name becomes clear. ... Ironically, a large part of the traffic in the bottleneck is not useful data but merely names of data, as well as operations and data used only to compute such names. [6]*

Backus predicted that non-von Neumann languages would dominate in the long term, but made the mistake of trying to address the bottleneck with a programming language implemented on a von Neumann architecture [6].

In these papers we see almost all the research effort being directed towards improving the instruction and address-based architecture, without questioning whether it is a suitable architecture.

WHY THE VON NEUMANN MODEL IS AN OBSTACLE

Much has been written about the efficiency cost of the memory hierarchy [7–10], but there has been little reflection on the causes of this inefficiency. Consider the probability that a particular memory address will be accessed in the next 1000 instructions. That probability is less than 1000/(size of memory), which is close to zero, even if we only consider the cache memory. If we consider the addressable memory, this probability is negligible. This is one example of how the design of the von Neumann computational model is inherently inefficient and the efficiency cost of the memory hierarchy is a symptom of this inefficiency. This simple example led me to think that we are approaching research into computing performance incorrectly, focusing on symptoms rather than causes.

In this section I examine key design flaws in the von Neumann model that need to be addressed if we are to achieve orders-of-magnitude improvements in performance. The von Neumann computational model makes use of instructions to process data, with the instructions and data stored in addressable memory. Instructions and memory addressing are two fundamental aspects of the model. For each, I explore the implications for speed of execution, cost of hardware, energy consumption and reliability. By unpacking the causes of these limitations in the von Neumann computational model, I argue for the need to explore alternatives.

A. The limitations of addressable memory

I have described above the inefficiency in the number of specific memory addresses that are accessed by a particular piece of program code. The manner in which memory storage is used is also inefficient. Most instructions result in information moving in and out of memory: getting the instruction, getting the data and storing the result. Instruction and data blocks may need to be swapped in and out of levels of the memory hierarchy as processing time is allocated to different processes following events like interrupts, page faults and process scheduling. A dirty cache block results in the further cost of having to save the cache block before bringing in the new block. These inefficiencies are inherent in the address-based model; they cannot be addressed by improved communication speeds, alternative memory hierarchies or the introduction of more processors.

Addressable memory appears to have three inherently problematic properties: (1) a finite address space, (2) numeric addresses which lead to linearity and (3) information-poor address referencing. All of these increase the complexity in programming [3–6].

A finite address space means that for non-trivial computations, some memory locations have to be reused. The program will thus need to know when a value can be stored in a particular address, and when that address has been reallocated. The consequences of this are that the program code has to manage the sequence and timing of the allocation of values to memory locations. That is, the code must manage what memory has to be allocated, for what purposes, and at what stage in the computation, as well as the allocation of values to the appropriate memory locations.

Numeric addresses results in a linear address space. This has implications for how data structures are stored. A consequence of a linear address space is stamp-coupling which requires all the components of a structured variable, such as an array, to be stored as a complete entity. Memory must be allocated for the entire structure, even if only one reference to one component is still needed. Referencing this memory for the structure is also inefficient, including global variables, parameters and indirect references. The components of the structure have to be accessed using offsets or pointers. Using offsets incurs the cost of calculating the offset and indexed addressing. Pointers require two levels of addressing as well as managing the allocation and deallocation of memory.

A second consequence of linearity is that we cannot be efficient about what is stored in active memory. As discussed above, a major overhead in the memory hierarchy is moving instructions and data around. Ideally we would want to predict what variables and instructions are required in the upcoming computations and load only these into memory. While one can predict the instructions likely to be executed and, given these, possibly predict the variables required, it is not possible to separate out and load only those instructions and variables; linear addressing requires continuous sequences of memory locations to be moved at once.

When people perform computations naturally, they make use of semantic information that aids in this process. For example, if I am subtracting liabilities from assets, the word “liabilities” is both a place-holder for a value and it communicates the meaning of that value. Although a memory address is a place-holder for a value, it lacks the semantic information that might aid in understanding the uses to which the value can be put. In the von Neumann model, the meaning of a value in a memory location is a complex composition of factors: the instruction currently executing, the statement in the program that relates to that instruction, the variable in that statement that is related to the memory address, and which instance of the variable is currently being referred to, which in turn depends on the state of execution of the program. The lack of semantic information means that the memory address provides no information as to how to process the value contained in the memory location, nor does it contribute to preventing accidental or malicious use or alteration.

The use of addressable memory requires the program to carry out the computations relating to a given value of a variable while that value remains in the memory location. State is thus critical. Not only does a program have to make correct computations with values, but it also must ensure that these computations take place in the correct sequence and at the appropriate time. This requires a programmer to apply temporal reasoning in designing a program, which considerably increases the complexity of the task.

Addressable memory results in a finite address space that is organised in a linear fashion and referenced by meaningless numeric addresses. These limitations lead to processing overheads that result in poor performance and the inefficient use of hardware and energy, as well as complexity which contributes to unreliability. The biggest inefficiencies occur as a result of data access: getting an operation, retrieving data, storing a result and incrementing a program counter. Each instruction involves, at best, one memory access and at worst, three memory accesses, as well as an index and a register access. The execution of an instruction can also escalate into page faults, operating system interventions and so forth.

Before turning to the weaknesses that result from the inherent properties of instructions, it remains to point out that addressable memory imposes an instruction-based architecture because it requires instructions to perform the functions involved in managing memory resources: to allocate, access and reallocate memory locations.

B. The limitations of instructions

Similarly, the instruction aspect of the von Neumann model has properties that result in inherent limitations. These are that (1) instructions are inherently sequential; both in terms of how each is processed and in relation to other instructions, (2) a sequence of instructions must fully occupy a processor while it is being executed, (3) instructions are themselves stored in memory in the same way as data, and (4) instructions are devoid of semantic information.

Instructions are inherently sequential in two different senses. Firstly, the manner in which the von Neumann model processes instructions, the instruction cycle, is sequential. The steps in the instruction cycle – getting the instruction, getting the data, combining the instruction with the data, performing the operation and saving the result – are executed in sequence. This model makes it impossible to break up the steps in the instruction cycle and have them carried out concurrently and asynchronously. This would allow the most time consuming steps to be handled by different hardware and buffering where the duration of a step may vary depending on the instruction.

Because the steps in the instruction cycle have to be carried out in sequence, the von Neumann model needs a clock to allocate time to each step. This clock allocates the same time to each step, meaning that hardware lies idle when a simple step is performed and the speed of the clock is limited by the slowest step in the instruction cycle. The only way to speed up processing is to speed up the clock. Chip manufacturers struggle with increasing power consumption and reliability as the clock speed is increased [8]. Having a clock in control makes hardware design more complex and harder to verify [7]. The design would be considerably simpler if each hardware function could be independent and did not have to be synchronised.

At any one time, only one sequence of instructions is being executed by a processor, until it relinquishes control back to the operating system, either by terminating or as the result of an interrupt. If interrupts are disabled, the executing code can keep control until the processor is reset. Even worse, with supervisor mode enabled, the executing code can potentially make changes outside the scope of its intended operation, including to the operating system. At best this weakness results in the potential for instability, but at worst it is exploited for malicious ends. The inclusion of third party code, such as drivers, in the operating system and the complexity of applications and the operating system, makes it almost impossible to verify that such errors cannot occur.

The consequence of this design, and a key feature of the von Neumann architecture, is that instructions are themselves stored in memory in the same way as data. The processor cannot easily differentiate between instructions and data in order to treat them differently. Program instructions can be downloaded, created or modified, and program code is thus vulnerable to errors and attacks. The processor does not control which instruction is to be executed next and what data is required. This can result page faults as well as context switches that require the swapping in and out of cache, as well as memory blocks, resulting in memory bottlenecks [8].

The nature of processing instructions means that the processor cannot be used efficiently. The processor is a critical resource which ideally should be fully occupied with application-related operations. Running a single set of instructions will typically result in a large amount of idle time due to delays from caching and input or output. One way to reduce idle time is for a number of sets of instructions to share the processor (multi-tasking), but this requires intervention by the operating system which comes at a cost. Another way to reduce idle time is to express a task as a number of concurrent sets of instructions (parallel processing). Typically these concurrent sets of instructions need to share data. Running them concurrently and the sharing of data has to be managed by the application, or by the operating system and this increases the amount of time that the processor is dealing with non-computational or overhead tasks. Whichever way, there are inefficiencies.

Efficient parallel processing requires work to be distributed evenly across processors. In most cases this requires passing or sharing of data across processors which results in problems such as deadlocks and race conditions that need to be addressed at a cost. There are potential delays in synchronising between processes such as waiting for data to be passed and to be read, or for a semaphore to enable access to data. Some of these problems can be mitigated by dealing with larger chunks of data, but this decreases the granularity of the parallelism.

In the same way that semantic information creates understanding of the meaning of values in memory, it can also illuminate the meaning of computations. A simple statement such as area = length * breadth expresses a straightforward computation. Area can be calculated once the values of length and breadth are defined. The instruction and address-based model introduces complexity into such a computation. The variables length and breadth may take on different values. Which values does the statement refer to? The risk of errors could be lowered by semantic information that relates the values and operations.

I have argued here that instructions have inherent properties that limit performance and reliability. The processing of an instruction requires a series of steps to be completed in sequence that necessitates the use of a clock, instructions are designed to be carried out in sequence, they fully occupy a processor while being executed, and they are expressed in a manner that lacks semantic information. These inherent properties of instructions result in poor performance and the inefficient use of hardware and energy, as well as introducing complexity and opportunities for errors and abuse which contribute to unreliability.

FEATURES DESIRABLE FOR HIGH PERFORMANCE

So we are now aware of the inherent properties of addressable memory and instructions that result in weaknesses in the von Neumann computational model. What features of a computational model are desirable in order to avoid these weaknesses?

To achieve high performance the aim is to complete each computation in the shortest possible time. The time taken to complete a computation can be reduced by (1) avoiding delays due to the inefficiency of the memory hierarchy (2) increasing parallelism, (3) fully utilising the processor and removing the overheads of the operating system. Speeding up the processing should, ideally, (4) reduce energy consumption, (5) simplify hardware design, and (6) simplify programming while making systems less vulnerable to malicious attack, all without introducing additional costs.

In the von Neumann model, the processing of each instruction involves a number of steps that are completed in sequence and controlled by a clock. The hardware remains idle if a step takes less time than the clock step. Hardware could be better utilised if these idle times were reduced or eliminated. One way to do this would be to decouple the steps in the cycle so that each step is self-contained and does not need the central clock to synchronise. The hardware for each step can then execute independently (for example, the arithmetic logic unit would not have to remain idle during a “load” step). Buffering between these components can be used to negate delays and to enable the parts of the execution cycle to execute in parallel. By decoupling the steps in the execution cycle, the design of the hardware could be simplified as circuits for each execution step could be designed separately and optimised for a particular step in the execution cycle. *So the first desirable attribute is that the computational model should allow for decoupling of the steps involved in executing an instruction.*

Parallelism increases performance and the more fine-grained the parallelism can be (without increasing the coordination overhead), the greater the performance improvement. If parallelism were taken to the level of each operation, this would mean that any operation should be able to be computed as soon as the operand values become available. Each operation may result in a value that needs to be stored, so in order not to limit the degree of parallelism, there should be no limit on how many such variables can be created and, in order to distinguish between them, each should be able to be uniquely referenced. That is, each variable should be referentially transparent throughout the computation. *So the second desirable attribute is that the computational model should allow for unlimited parallelism and unlimited, uniquely referenced variables.*

The von Neumann computational model has the operating system using valuable processor time to allocate both processor and memory resources. The demand from applications for resources is dynamic and cannot be determined a priori. Since the objective is to optimise the use of the processors, these processors should not be involved in resource management. Better ways of determining how to allocate resources are needed and the task can be delegated to a separate unit. *So our third requirement is that the computational model should separate resource management and the processing of steps in the execution cycle and have different hardware circuits for these functions.*

A computation is made up of a number of different steps: identifying the operation to perform, determining the data involved, performing the operation, and determining what to do with the result. Further there are additional tasks such as input and output of data. If each of these steps could be carried out independently then (1) the circuitry for each could be optimally used, (2) the complexity of the circuits could be considerably reduced (not have to synchronise with the clock), (3) the wire lengths within circuits can be reduced as each component is self-contained, facilitating increased speeds at lower temperatures and (4) where there is larger demand for some components these can be duplicated.

Memory access for both instructions and data should not impact on the performance of the CPU. In particular, an application should not have to wait because required instructions or data are not in memory. An address access costs more than a register access, both in time and hardware, and should be avoided. *So, the fourth desirable attribute is that the computational model should be able to provide the processing circuits with what they need “just-in-time”.*

One factor often overlooked in discussions of computational models is the cost of developing and maintaining software. Operating systems, utilities and development tools are complex so that their development requires considerable investment in time and skilled programmers. Optimising these tools to get the best hardware performance for a given application requires considerable skill and experimentation. When performance is critical, each software application needs to be tailored and this requires an in-depth understanding of the hardware, the operating system, utilities and development tools on the part of the programmer. In addition high performance can often not be achieved without modifying program code for specific instances of hardware and operating system. Programming that requires temporal reasoning and the ability to imagine all the possible states of a system requires high level cognitive skills that are in short supply. *So, the fifth desirable attribute is a computational model that simplifies programming and optimisation of programs.*

We have discussed several features of the von Neumann model that result in vulnerabilities. An incorrect index value can result in data or code being corrupted, whether accidentally or maliciously. Of concern is that such failures often go undetected with unknown consequences. *So, the sixth and final desirable feature is that the computational model should result in robust software that is not vulnerable to errors in the code or to attack.* In particular these vulnerabilities can be avoided if code and data cannot be changed during execution.

The next section explores whether it is possible to design an alternative computational model that has these features.

ARE THERE ALTERNATIVE COMPUTATIONAL MODELS?

It is easy to find fault with a system and more difficult to propose alternatives. However in this case a computational model already exists without the identified weaknesses that can form the basis for an alternative architecture. This paper compares the instruction and address-based computational model with the natural way in which people compute. I argue that the semantic information that people associate with the elements of computation could be mimicked to gain efficiencies in computer architecture.

C. *The broad principles of the model*

Space limitations prevent a full description of the implementation of the model, so the paper describes the principles and outlines the implementation.

The one alternative computational model is simple, manual arithmetic calculation. Simple mathematical notation has been used to describe such calculation. For example, the sum of a series of numbers can be expressed as:

$$s_0=0,\ s_{i+1}=s_i+n_i$$

(This expression allows the computation of any set of numbers, even an unlimited set.) Given a set of numbers:

$$n_0=5,\ n_1=3,\ n_2=-4$$

the computation can be done as follows:

$$s_0=0,\ s_1=s_0+n_0=0+5=5,\ s_2=s_1+n_1=5+3=8,$$
$$s_3=s_2+n_2=8+(-4)=4$$

The computation described relies on:
- a generalised relation si+1=si+ni that holds for a class of variables;
- variable identifiers (in this case s and n) used to determine which generalised relation applies to the variable;
- variable indices used to instantiate specific variables;
- a value associated with each variable, for example the value 3 with the variable n1;
- each variable having only one value; so that n1 is only ever 3 and does not take on other values as it would it an instruction-based program;
- the relating of variables expressed in a relation; and
- the computation of a result using the relation.

To automate such a computation would require:
- a means to record generalised relations,
- a means to identify (a possibly infinite number of) variables and their associated values,
- some way of identifying the relations that a variable is part of,
- a way to compute the operations in the relation; and
- some process that drives the computation in a similar way to the instruction cycle.

In this model a variable is not a place in memory. Rather a variable consists of an identifier (possibly including an index), and a value. Variables become available when their identifiers and values are known. Each variable can have only one value throughout the execution of the program. This means that each variable identifier is associated with something in the problem domain. The variable identifier thus tells us about that thing in the problem domain, effectively associating semantic information with the value. It may be helpful to think of a variable in the mathematical sense, of an element in a domain that has defined relationships with other elements.

This model does not have instructions. Rather the process is driven by which variables are available at any given time and which relations the variables are part of. To automate the manual process, variables (with their values) are "processed" as they become available. Each variable identifier is matched with the relations it forms part of. If all the necessary variables for a defined relation are available the result is calculated and becomes available as a new variable. Once a variable has been processed for all the relations that it is part of, it is discarded.

The inherent parallelism is that any number of variables can be processed at the same time. As soon as the variables for any relation are available, the result can be computed.

D. Accumulating variables for a relation

For any particular relation, all the variables needed may not become available at the same time. The design challenge for the architecture is to accumulate the variables that form part of a relation so that once all the variables (with their values) are available, the computation can be carried out. This section argues that all relations can be reduced to unary, binary or indexed relations and gives examples of how this accumulation could be done for each of these three cases. (Selection, which is required for Turing completeness, can also be reduced to a binary operation, but this is beyond the scope of the paper.)

Like with conventional compilers and computers, a relation can be broken up into simple relations consisting of a single operator with one or two operands [2]. In this way, the problem is simplified by only having to deal with relations that consist of unary or binary operations. The unary operations are straight-forward as the operation is applied to only the variable currently being processed and there is no need to identify additional variables. The operand is applied to value of the variable being processed and a new variable is created. For example, given the relation $n=-p$ and the variable $p=5$, when the variable p is processed, the identifier p is used to establish that the variable is used in the relation $n=-p$ and a new variable $n=-5$ is created by applying − to the value 5 of p. The resulting variable, $n=5$ is now available to be processed.

The binary operation is more complex. To compute the relation $v=a+b$, the values of the two variables a and b need to be available. However variables may become available independently of one another; there is no synchronising of the variables. One approach is to get rid of operations requiring two operands and replace them with unary operations that apply to a tuple t=[a;b]. The relation $v=a+b$ can now be expressed as v=+t. We create the tuple t using two operations: $\ltimes$ to create the left part of the tuple and $\rtimes$ to create the right part of the tuple. So we also have the relations t=$\ltimes$a and t=$\rtimes$b.

So when the variable a is processed, it forms the left part of a tuple t and when the variable b is processed it forms the right part of the tuple t. When both a and b have been processed and both parts of the tuple t exist, the operation can be applied and the relation computed. For example, with the above relations and the variables b=5 and a=3 processed one after the other, the result is:

Variable	Relation	Partial tuple	Variable
b=5	t=$\rtimes$b	t=[;5]	
a=3	t=$\ltimes$a	t=[3;]	t=[3;5]
t=[3;5]	v=+t		v=8

The matching of the partial tuples is handled by a dedicated tuple processor. This approach allows the two operands to be processed in any order and the variable identifier can be used to match the two halves.

A significant difference in this model is that variables are associated with a single value for the duration of the program. The requirement that variables each have only one value means that we need to have an unlimited number of variables to express programs that are non-trivial. To achieve this, indices are introduced as shown in the initial example. For this reason a variable identifier may have an index associated with it and we need to be able to increment the indices. Note that the index (if there is one) is part of the variable and not a separate variable. The sum expressed as $s_{i+1}=s_i+n_i$ can be broken down into the following relations:

$$s'i = \ltimes si$$
$$s'i = \rtimes ni$$
$$s''i = +s'i$$
$$si+1 = [+1]\ s''i$$

where [+1] is the operation to add 1 to the index.

Given the variable values above, the computation can proceed as follows:

Variable	Relation	Partial tuple	Variable
s0 = 0	s′i = $\ltimes$si	s′0 = [0,]	
n0 = 5	s′i = $\rtimes$ni	s′0 = [0,]	
		s′0 [,5]	s′0 = [0,5]
s′0 [0,5]	s″i = +s′i		s″0 = 5
s″0 = 5	si+1 = [+1]s″i		s1 = 5
s1 = 5	s′i = $\ltimes$si	s′1 = [5,]	
n1 = 3	s′i = $\rtimes$ni	s′1 = [5,]	
		s′1 [,3]	s′1 = [5,3]
s′1 = [5,3]	s″i = +s′i		s″1 = 8
s″1 = 8	si+1 = [+1]s″i		s2 = 8
s2 = 8	s′i = $\ltimes$si	s′2 = [8,]	
n2 = -4	s′i = $\rtimes$ni	s′2 = [8,]	
		s′2 [,-4]	s′2 = [8,-4]
s′2 = [8,-4]	s″i = +s′i		s″2 = 4
s″2 = 4	si+1 = [+1] s″i		s3 = 4

Note that the variables can be processed in any order. The sequence above is to aid with following the computation process. Note too that the indices in the relations play no role in the computation and the relations could equally be expressed as:

$$s' = \ltimes s$$
$$s' = \rtimes n$$
$$s'' = +s'$$
$$s = [+1]\ s''$$

E. *The computational cycle*

The computational cycle has similarities to the traditional instruction cycle but the steps are decoupled from one another. The steps are:

- get the next available variable, including the unique identifier and value (analogous to getting the data)
- get the relations that apply to the variable (analogous to getting an instruction)
- apply the operation of each relation to the value to create a new variable (analogous to performing the instruction)
- store the newly created variable (analogous to storing the result)

The architecture would then need the following four units to perform these functions:

The *Mapping unit* pops a variable (x, i, 5) off a variable queue and uses the identifier x to identify a list of relations that involve this variable. For each relation (for example v=-x), an augmented variable is created which consists of the identifier of the new variable to be created by the relation (v), the index i of the popped variable, the value of the variable (5) and the operation of the relation (-). (In the example given, the augmented variable would be (v, i, 5, -).) Depending on the operation, the augmented variable is then pushed onto the evaluation queue, the tuple queue or the index queue to be dealt with by one of the units described below.

The *Evaluation unit* deals with operations on variable values. It pops an augmented variable off the evaluation queue and applies the operation to the value. If the result is defined, a new variable is created with the identifier and index of the augmented variable, and the value resulting from the operation. (In the example, (v, i, -5)). This variable is pushed onto the variable queue for processing by the mapping unit.

The *Index unit* deals with operations on indices. It pops an augmented variable (such as (v, 3, 5, [+1])) off the index queue and applies the (index) operation to the index. If the result is defined, a new variable is created with the identifier and value of the augmented variable, and the index resulting from the (index) operation. (In the example, (v, 4, 5).) The variable is pushed onto the variable queue for processing by the mapping unit.

The *Tuple unit* deals with the creation and matching of tuples. It pops an augmented variable off the tuple queue (for example (v, i, 5, $\rtimes$)). If the matching tuple does not exist (within the Tuple unit's hash table), it creates a variable with a partial tuple (in this case (v, i, [;5]) which remains in the Tuple unit (stored in a hash table) to be matched in the future. If the matching tuple exists (for example v_i=[3;]), it creates a variable with identifier and index of the augmented variable and a tuple made up of the values of the two matching tuples. (In the example, (v, i, [3,5]).) The variable is pushed onto the variable queue for processing by the mapping unit.

The design deliberately enables each of these units to operate independently of each other, thus achieving the desired decoupling of the execution steps. Fig. 1 gives a schematic view of how such a model can be implemented.

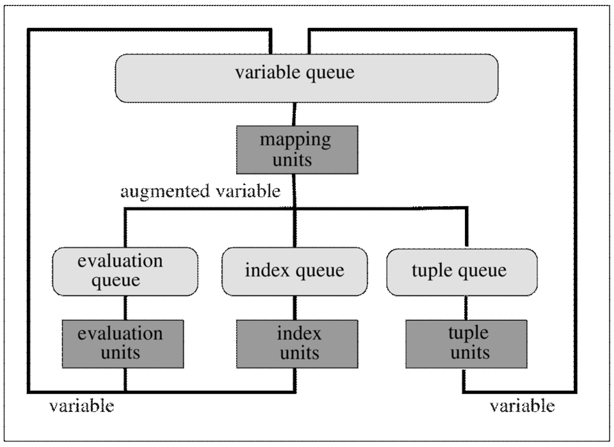

Fig. 1. Schematic of the proposed architecture

Indices can be used to handle abstraction and complex flexible data structures. Input can be handled by another unit that interfaces with the input device and translates the input into variables that are pushed onto the variable queue. Output variables are picked up by an output unit and translated into the correct format for the output device. A more complete description of the implementation and simulations of its functioning can be found elsewhere [10, 11]. The model has shown promise with experimentation on an emulation which has limited operations. High degrees of parallelism were achieved and no unexpected pitfalls emerged [12].

In this model a program is expressed as an unordered list of relations between variables. This simplifies programming as the programmer does not need to be concerned with the timing of execution and states. The use of queues means that processing takes place as soon as the variables become available to process, giving us the desired “just-in-time” processing.

The purpose of this paper is to argue for exploring such new models of computation rather than giving the full details of the implementation. The following section discusses how the suggested model avoids the weaknesses identified in the instruction and address-based model, the objective being to show that these weaknesses are not present in all computational models and architectures.

THE PERFORMANCE POTENTIAL OF ALTERNATIVE MODELS

The argument is that the instruction and address-based model has inherent weaknesses which make it unsuitable for high performance computing. Six properties were identified that are desirable for high performance computing and the previous section presented a model that has these properties. In this section an assessment is made as to the potential of the proposed model to meet the needs of high performance computing. As identified in the beginning of Section 4, the considerations are: (1) memory management, (2) parallelism, (3) use of the processor, (4) running costs, (5) cost of design and (6) software costs and reliability.

F. Memory management

First, the problems related to memory management include (a) inefficient use of memory, (b) the cost of bottlenecks in moving data around, and (c) an additional workload for the central processing unit to manage data.

The proposed variable-based model uses memory more efficiently because only active variables are stored. No storage is allocated to undefined variables or to variables that have been processed and are no longer required.

A limited number of active variables need to be kept in high speed memory. As the gap between processing times and memory access times widens, it is necessary to use high speed caches which can match processor speeds. This model only requires buffers for the front and back of the variable queue to handle variables about to be processed, and the newly created variables about to be stored. The sizes of these buffers are minimal as they only have to be large enough to handle the transfer. This reduces the cache and memory sizes. Data bottlenecks are avoided by ensuring that only data about to be processed is transferred to high speed memory.

The model avoids the need for interrupts. In addition, the queue of variables can contain variables from different tasks with the task information stored as part of the semantic information in each variable, thus avoiding the need for time-slicing.

The instruction and address-based model requires a complex memory hierarchy to handle the virtual memory model, involving different levels of caches, memory and secondary storage. Blocks of data are transferred in and out of these layers with additional costs if the block has to be written back. Large amounts of data that are not required are unnecessarily moved around because access to a variable in a block requires the whole block to be transferred into cache. An interrupt or a time slice inevitably results in transferring blocks of data across the memory hierarchy.

In the proposed variable-based model, the atomic elements of a data structure are separate variables and their relationship to the data structure is contained in the semantic information that forms part of the variable identifier. Each of the atomic components that make up the structure is processed independently. Thus there is no need for data structures such as an array to be stored as a continuous block of memory. Nor does the structure need to remain in memory from when one element of the structure is defined until there are no remaining elements that may be referenced.

The atomic variables in the proposed model contain structural information of more complex data structures in the semantic component of the variable. Unlike the instruction and address-based model, where structure is represented either using pointers or linear positional information. The variable-based model thus avoids the central processing unit overhead of managing dynamic space, as well as calculating the address of components in the structure.

G. Parallelism and use of processors

There is broad agreement that any architecture that is to meet the demands of large data and intensive computation will need high degrees of parallelism and highly efficient use of processors. Optimal levels of parallelism will be attained if processing can be assigned at a low level of granularity to available processors during execution.

The instruction- and address-based model is an inherently sequential process. Parallelism is achieved by breaking up the computation into a number of concurrent sequential processes. These processes require intervention by the operating system to ensure each process is given access to a processor and to handle communication between processes, be it message passing or shared memory. Processes need to be allocated to a specific processor. The code determines the order computations take place, the memory allocation and the use of memory. All of this adds complexity to programming and the execution of parallel programs and parallel processing cannot be optimised during execution.

The proposed variable-based model is inherently parallel because it operates non-sequentially and processing is at the level of each variable. Variables can be evaluated, as they become available, by any available processor, making the parallelism fine-grained. The mechanism allocates resources, rather than a program, and no intervention is required by the operating system.

In this model, a program is a specification of relationships between variables expressed in simple mathematical notation. Programming consists of describing relations between variables. The programmer does not have to define the sequence of execution or specify the allocation of resources. There is no need to break a program into parallel tasks with complex synchronisation mechanisms between them, and the programmer does not need to concern themselves with the architecture or configuration of processors.

A major focus in improving the performance of current architectures has been on improving the execution cycle. The steps of getting the instruction, getting the data, performing the operation, saving the result and updating the program counter need to done in order and synchronised by the clock cycle. These steps are required in order to combine the instruction with the data before the instruction can be performed. The instruction is determined by the program counter, just before the instruction is performed, and the necessary data identified once the instruction is loaded. These phases have to be synchronised to maintain the instruction pipeline, hence the clock is critical.

The variable based model decouples the different functions involved in the processing of a variable. The functions of mapping, evaluating, indexing, forming tuples and input and output can proceed independently and in parallel with each other; synchronisation is handled by the system. So the model is inherently parallel in terms of its design as well as in terms of the execution of programs.

H. Costs and other considerations

Increasing the speed of the clock increases power consumption, cost of manufacture, and running costs, while reducing reliability. Not having to synchronise phases of the execution cycle with a clock avoids delays when one part of the cycle takes longer than the other. It also allows the architecture to have multiple units for each of the phases executing in parallel. Both of these aspects increase the performance on the execution cycle without having to increase an overall clock speed. This has impacts on the reliability of the hardware and energy consumption as well the cost of the hardware and running costs.

Because processing is handled by four dedicated processing units, each unit can designed independently of the rest. This simplifies the design and allows for performance optimisation within each unit (including shorter connection lengths). There is no need for complex synchronisation between units, which is handled by the variable queue. Only limited use is made of high speed cache, allocated to store the front of the queue of variables about to be processed.

The proposed model simplifies programming (and thus reduces the cost of programming) by reducing it to the specification of relationships between variables. This makes the programming task much simpler because the programmer does not have to specify the order of execution or how resources are allocated. Reasoning about static relationships is simpler; the programmer no longer has to be concerned with state and temporal reasoning. There is no need to explicitly optimise code for parallel execution.

Current architectures have a major weakness in that they are vulnerable to accidental and malicious corruption of memory. Perhaps even worse are the unknown consequences of undetected pointer and index errors. The proposed model avoids such problems because it is not instruction driven. Variable-driven processing avoids these vulnerabilities because it is not possible to incorrectly alter an instruction or value, either accidentally or maliciously. The identifier of a variable determines how the variable is processed. An operation can only result in a new variable being created, so no code can be generated and no other variable can be modified as each variable is processed. The operating system functions are not performed by the units that process variables. Thus the proposed model saves on the cost of identifying such errors as well as the cost of the consequences. It is a safer model of computing.

The variable based model illustrates that the weaknesses of the instruction- address-based model are not inevitable. Overcoming these weaknesses would have considerable performance benefits for high performance computing. This section discusses one possible model that has the potential to address the identified weaknesses.

POTENTIAL PITFALLS AND SIGNS OF SUCCESS

There are two aspects of the variable-based model presented here that have performance implications: the matching of tuples and managing indices.

Forming tuples is a critical component in being able to move away from addressing, providing the critical intermediate step to form variables for binary operations. The matching of tuples is likely to be of the same order as that of the algorithm used in the computation. The problem occurs when having to pair a newly created unmatched tuple against an excessive number of unmatched tuples. Work has been done to address this using grouped tuples and distributing the matching task.

The model also requires that there be no limit on the number of indices that can be attached to the semantic component of variables. This enables complex data structures and high levels of abstraction. Without going into too much detail, it is possible to manage multiple indicies using chains.

The model has shown promise when emulated with limited operations [12]. High degrees of parallelism were achieved and no unexpected pitfalls emerged. In comparing a matrix multiplication program written in C to run in parallel on multiple processors with an equivalent variable-based program on the emulation, the former has to perform more instructions than the latter performs operations.

CONCLUSION

Some, such as the Berkeley group, advocate major change in computing paradigms as follows:

> *Since real world applications are naturally parallel and hardware is naturally parallel, what we need is a programming model, system software, and a supporting architecture that are naturally parallel. Researchers have the rare opportunity to re-invent these cornerstones of computing, provided they simplify the efficient programming of highly parallel systems. [1]*

Yet we see little evidence of this challenge being taken up. Instead considerable effort and resources are going into improving the instruction and address-based model. This suggests that either there is a belief that significant improvements can still be made or that the predominant belief is that there is no alternative. However researchers continue to identify and analyse weaknesses of the dominant model and to argue that it has hard limitations and alternative models need to be explored.

Proposing a radical departure from current models of computing is not an easy task. Investments in the current architecture in terms of current research and development, as well as the installed base of hardware and software are overwhelming and will be slow to change [1]. However this ought not to deter research into alternatives that may be needed to make progress in the future.

The instruction and address-based model is so entrenched in our thinking that it is difficult to conceive of alternatives. This paper attempts to think more freely about computing by first identifying the desirable features of a computational model and architecture for high performance computing, and then exploring the design of a model that exhibits these properties.

The proposed model is non-sequential, allowing immediate computation of any operation whose operands are defined, putting the focus on processing variables rather than instructions. The paper gives a brief outline of how such a model might work. More detailed information about the model, as well as the results of emulation experiments are described elsewhere [10–12]. The proposed model does not exhibit the weaknesses of the von Neumann model. The two models are compared in terms of memory management, parallelism and the use of processors, and issues of cost, to argue that the proposed model has the potential to be better suited for high-performance computing.

Being able to come up with an alternative model that has the desirable attributes shows that there are alternatives to the instruction and address-based model that are worth exploring. At this stage it is not possible to argue that the model can provide the basis for alternative architectures that will out-perform current architectures. It is possible that weaknesses in this model will emerge that may outweigh the benefits. Considerably more work will be needed to develop the model more fully, but early attempts at emulating its functioning have been promising.

The conclusion is that the von Neumann model is not the most appropriate model to address the needs of large data and high performance computing because of its inherent weaknesses. However it is possible to design computational models and computing architectures that do not have the same inherent weaknesses, with the potential to better meet the processing needs that science is expecting from computing. Such alternatives are worth exploring further.

ACKNOWLEDGMENT

The author thanks Judy Backhouse for significant improvements to the paper.

REFERENCES

[1] M. Oskin, J. Torrellas, C. Das, J. Davis, S. Dwarkadas, L. Eeckhout, B. Feiereisen, D. Jimenez, M. Hill, M. Kim,

J. Larus, M. Martonosi, O. Mutlu, K. Olukotun, A. Putnam, T. Sherwood, J. Smith, D. Wood, C. Zilles, "Workshop on Advancing Computer Architecture Research (ACAR-II) Laying a New Foundation for IT: Computer Architecture for 2025 and Beyond", Computing Research Association, Seattle, Washington, 2010.

[2] A. V. Aho, M. S. Sethi, J. D. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986. ISBN 0-201-10088-6

[3] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. P. Patterson, W. L. Plishker, J. Shalf, S. W. Wiliams and K. A. Yelick. "The Landscape of Parallel Computing Research: A View from Berkeley". Tech. rep., Electical Engineering and Computer Sciences, University of California at Berkeley, 2006. URL www://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html.

[4] A Community White Paper. "21st Century Computer Architecture", http://cra.org/ccc/docs/init/21stcenturyarchitecturewhitepaper.pdf, 2012. URL http://cra.org/ccc/docs/init/21stcenturyarchitecturewhitepaper.pdf.

[5] T.P.I.T.A.C. (PITAC). "Computational Science: Ensuring America's Competitiveness". May 2005.

[6] 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, pp. 613–641, 1977.

[7] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kauffman, 4th edition edn., 2007.

[8] M. D. Hill. "Research Directions for 21st Century Computer Systems: Asplos 2013 Panel". *SIGARCH Comput. Archit. News*, vol. 41, no. 1, pp. 459–460, Mar. 2013. ISSN 0163-5964. 10.1145/2490301.2451165. URL http://doi.acm.org/10.1145/2490301.2451165.

[9] S. A. McKee. "Reflections on the Memory Wall". In *Proceedings of the 1st Conference on Computing Frontiers*, CF '04, pp. 162–. ACM, New York, NY, USA, 2004. ISBN 1-58113-741-9. 10.1145/977091.977115. URL http://doi.acm.org/10.1145/977091.977115.

[10] C. Mueller. "Element-Based Computational Model". *International Journal of Modern Education and Computer Science (IJMECS)*, vol. 4, no. 1, pp. 1–11, February 2012.

[11] C. Mueller. "Axiom based architecture". *SIGARCH Comput. Archit. News*, vol. 40, no. 2, pp. 10–17, May 2012. ISSN 0163-5964. 10.1145/2234336.2234339. URL http://doi.acm.org/10.1145/2234336.2234339.

[12] P. Mukala, J. Kinyua and C. Muller. A Theoretical Evaluation of AriDeM using Matrix Multiplication, 2011 International Conference on Communication Engineering and Networks, ISSN: 2010-460X

Conrad S. M. Mueller has been retired since 2016 but holds a position on the council of the University of the Witwatersrand (Wits) South Africa. He obtained his B.Sc. (1974), B.Sc.Hons. (1975) and Ph. D. (1989) in computer science from Wits and a M. Sc. (1977) in computer science from Rand AfrikaansUniversity South Africa.

His earlier research started in the South African Council for Scientific Research (CSIR) in 1976. After this in 1979 he took a post as project engineer at Anglo American to develop a control system based on the first micro processors. Wanting to study further and contribute to teaching resulted in him taking on an academic post at Wits in 1981. His experience with control systems, teaching of programming and the beauty of mathematics resulted in him questioning the imperative computational model. He has developed a new paradigm based on manual arithmetic to express computation and an architectural model to do the computation. Towards the end of his career, he took on more managerial roles of head of department and head of school as well as academic consultant in setting up two new universities. He held the position of professor at Wits and extraordinary professor at the University of South Africa. His is a member of the ACM.