How to Solve the Parallel Programming Crisis

By Louis Savain
Foreword

This e-book is a collection of 36 articles and essays about computer science that I have written over the years. This entire book centers around a new software model called COSA that promises to radically change the way we build and program our computers. I originally envisioned COSA as a solution to the software reliability and productivity crisis but, seeing that COSA programs and objects were inherently parallel from the start, I began to promote it as a solution to the parallel programming crisis. I decided to put the five articles that describe the core COSA concept at the end of the book starting with Why Software Is Bad and What We Can Do to Fix It. One reason is that it is still work in progress even though, in a sense, it will always be work in progress since every COSA application is an extension of the COSA operating system. Another reason is that I felt that the reader should be free to get acquainted with the COSA model at his/her own leisure. Hopefully by that time, the reader will be sufficiently persuaded that the Turing Computing Model of computing was a bad idea from the moment the computer industry embraced semiconductors. By now, it should be clear to everybody in the business that the Turing Model of computing contributes absolutely nothing toward solving the parallel programming crisis. I hope this book will convince a handful in the computer industry that it is time to abandon the flawed ideas of the last half-century and forge a bold new future.

Please note that the articles in this book, with the exception of the COSA articles mentioned above, are not organized in any particular order. Most of the articles end with a list of one or more related articles. As you read, it is important to keep in mind that all arguments in this book have a single purpose and that is to defend and support the COSA software model. My blog is Rebel Science News. Check it out if you are interested in alternative views on computing, physics and artificial intelligence.

Acknowledgment

I would like to thank all my readers, especially those of you who have followed my work and encouraged me over the years. I do appreciate your constructive criticism even when I make a fuss about it.
How to Solve the Parallel Programming Crisis

Abstract

Solving the parallel computing problem will require a universal computing model that is easy to program and is equally at home in all types of computing environments. In addition, the applications must be rock-solid. Such a model must implement fine-grained parallelism within a deterministic processing environment. This, in essence, is what I am proposing.

No Threads

The solution to the parallel programming problem is to do away with threads altogether. Threads are evil. There is a way to design and program a parallel computer that is 100% threadless. It is based on a method that has been around for decades. Programmers have been using it to simulate parallelism in such apps as neural networks, cellular automata, simulations, video games and even VHDL. Essentially, it requires two buffers and an endless loop. While the parallel objects in one buffer are being processed, the other buffer is filled with the objects to be processed in the next cycle. At the end of the cycle, the buffers are swapped and the cycle begins anew. Two buffers are used in order to prevent racing conditions. This method guarantees rock-solid deterministic behavior and is thus free of all the problems associated with multithreading. Determinism is essential to mission and safety-critical environments where unreliable software is not an option.

Speed, Transparency and Universality

The two-buffer/loop mechanism described above works great in software but only for coarse-grain objects such as neurons in a network or cells in a cellular automaton. For fine-grain parallelism, it must be applied at the instruction level. That is to say, the processor instructions themselves become the parallel objects. However, doing so in software would be much too slow. What is needed is to make the mechanism an inherent part of the processor itself by incorporating the two buffers on the chip and use internal circuitry for buffer swapping. Of course, this simple two-buffer system can be optimized for performance by adding one or more buffers for use with an instruction prefetch mechanism if so desired. Additionally, since the instructions in the buffer are independent, there is no need to process them sequentially with a traditional CPU. Ideally, the processor core should be a pure MIMD (multiple instructions, multiple data) vector core, which is not to be confused with a GPU core, which uses an SIMD (single instruction, multiple data) configuration.
The processor can be either single core or multicore. In a multicore processor, the cores would divide the instruction load in the buffers among themselves in a way that is completely transparent to the programmer. Adding more cores would simply increase processing power without having to modify the programs. Furthermore, since the model uses fine-grain parallelism and an MIMD configuration, the processor is universal, meaning that it can handle all types of applications. There is no need to have a separate processor for graphics and another for general purpose computing. A single homogeneous processor can do it all. This approach to parallelism will do wonders for productivity and make both GPUs and traditional CPUs obsolete.

**Easy to Program**

The main philosophy underlying this parallel processing model is that software should behave logically more like hardware. A program is thus a collection of elementary objects that use signals to communicate. This approach is ideal for graphical programming and the use of plug-compatible components. Just drag them and drop them, and they connect themselves automatically. This will open up programming to a huge number of people that were heretofore excluded.

**Conclusion**

Admittedly, the solution that I am proposing will require a reinvention of the computer and of software construction methodology as we know them. But there is no stopping it. The sooner we get our heads out of the threaded sand and do the right thing, the better off we will be.
Parallel Computing: Why the Future Is Non-Algorithmic

Single Threading Considered Harmful

There has been a lot of talk lately about how the use of multiple concurrent threads is considered harmful by a growing number of experts. I think the problem is much deeper than that. What many fail to realize is that multithreading is the direct evolutionary outcome of single threading. Whether running singly or concurrently with other threads, a thread is still a thread. In my writings on the software crisis, I argue that the thread concept is the root cause of every ill that ails computing, from the chronic problems of unreliability and low productivity to the current parallel programming crisis. Obviously, if a single thread is bad, multiple concurrent threads will make things worse. Fortunately, there is a way to design and program computers that does not involve the use of threads at all.

Algorithmic vs. Non-Algorithmic Computing Model

A thread is an algorithm, i.e., a one-dimensional sequence of operations to be executed one at a time. Even though the execution order of the operations is implicitly specified by their position in the sequence, it pays to view a program as a collection of communicating elements or objects. Immediately after performing its operation, an object sends a signal to its successor in the sequence saying, ‘I am done; now it’s your turn”. As seen in the figure below, an element in a thread can have only one predecessor and one successor. In other words, only one element can be executed at a time. The arrow represents the direction of signal flow.

![Algorithmic Diagram]

In a non-algorithmic program, by contrast, there is no limit to the number of predecessors or successors that an element can have. A non-algorithmic program is inherently parallel. As seen below, signal flow is multidimensional and any number of elements can be processed at the same time.

![Non-Algorithmic Diagram]
Note the similarity to a neural network. The interactive nature of a neural network is obviously non-algorithmic since sensory (i.e., non-algorithmically obtained) signals can be inserted into the program while it is running. In other words, a non-algorithmic program is a reactive system. Note also that all the elements (operations) in a stable non-algorithmic software system must have equal durations based on a virtual system-wide clock; otherwise signal timing would quickly get out of step and result in failure. Deterministic execution order, also known as synchronous processing, is absolutely essential to reliability. The figure below is a graphical example of a small parallel program composed using COSA objects. The fact that a non-algorithmic program looks like a logic circuit is no accident since logic circuits are essentially non-algorithmic behaving systems.

No Two Ways About It

The non-algorithmic model of computing that I propose is inherently parallel, synchronous and reactive. I have argued in the past and I continue to argue that it is the solution to all the major problems that currently afflict the computer industry. There is only one way to implement this model in a Von Neumann computer. As I have said repeatedly elsewhere, it is not rocket science. Essentially, it requires a collection of linked elements (or objects), two buffers and a loop mechanism. While the objects in one buffer are being processed, the other buffer is filled with objects to be processed during the next cycle. Two buffers are used in order to prevent signal racing conditions. Programmers have been using this technique to simulate parallelism for ages. They use it in such well-known applications as neural networks, cellular automata, simulations, video games, and VHDL. And it is all done without threads, mind you. What is needed in order to turn this technique into a parallel programming model is to apply it at the instruction level. However, doing so in software would be too slow. This is the reason that the two buffers and the loop mechanism should ideally reside within the processor and managed by on-chip circuitry. The underlying process should be transparent to the programmer and he or she should not have to care about whether the processor is single-core or multicore. Below is a block diagram for a single-core non-algorithmic processor.

Adding more cores to the processor does not affect existing non-algorithmic programs; they should automatically run faster, that is, depending on the number of objects to be processed in parallel. Indeed the application developer should not have to think about cores at all, other than as a way to increase performance. Using the non-algorithmic software model, it is possible to design an auto-scalable, self-balancing multicore processor that implements fine-grained deterministic parallelism and can handle anything you can throw at it. There is no reason to have one type of processor for graphics and another for general-purpose programs. One processor should do everything with equal ease. For a more detailed description of the non-algorithmic software model, take a look at Project COSA.

Don’t Trust Your Dog to Guard Your Lunch

The recent flurry of activity among the big players in the multicore processor industry underscores the general feeling that parallel computing has hit a major snag. Several parallel computing research labs are being privately funded at major universities. What the industry fails to understand is that it is the academic community that got them into this mess in the first place.
British mathematician Charles Babbage introduced algorithmic computing to the world with the design of the analytical engine more than 150 years ago. Sure, Babbage was a genius but parallel programming was the furthest thing from his mind. One would think that after all this time, computer academics would have realized that there is something fundamentally wrong with basing software construction on the algorithm. On the contrary, the algorithm became the backbone of a new religion with Alan Turing as the godhead and the Turing machine as the quintessential algorithmic computer. The problem is now firmly institutionalized and computer academics will not suffer an outsider, such as myself, to come on their turf to teach them the correct way to do things. That’s too bad. It remains that throwing money at academia in the hope of finding a solution to the parallel programming problem is like trusting your dog to guard your lunch. Bad idea. Sooner or later, something will have to give.

Conclusion

The computer industry is facing an acute crisis. In the past, revenue growth has always been tied to performance increases. Unless the industry finds a quick solution to the parallel programming problem, performance increases will slow down to a crawl and so will revenue. However, parallel programming is just one symptom of a deeper malady. The real cancer is the thread. Get rid of the thread by adopting a non-algorithmic, synchronous, reactive computing model and all the other symptoms (unreliability and low productivity) will disappear as well.

See Also:
How to Solve the Parallel Programming Crisis
Parallel Computing: The End of the Turing Madness
Parallel Computing: Why the Future Is Synchronous
Parallel Computing: Why the Future Is Reactive
Why Parallel Programming Is So Hard
Why I Hate All Computer Programming Languages
Parallel Programming, Math and the Curse of the Algorithm
The COSA Saga

Parallel Computing: Why the Future Is Synchronous

Synchronous Processing Is Deterministic

I have always maintained (see the COSA Software Model) that all elementary processes (operations) in a parallel program should be synchronized to a global virtual clock and that all elementary calculations should have equal durations, equal to one virtual cycle. The main reason for this is that synchronous processing (not to be confused with synchronous messaging) guarantees that the execution order of operations is deterministic. Temporal order determinism goes a long way toward making software stable and reliable. This is because the relative execution order (concurrent or sequential) of a huge number of events in a deterministic system can be easily predicted and the predictions can in turn be used to detect violations, i.e., bugs.
Expected events (or event correlations) are like constraints. They can be used to force all additions or modifications to an application under construction to be consistent with the code already in place. The end result is that, in addition to being robust, the application is easier and cheaper to maintain.

**Synchronous Processing Is Easy to Understand**

The second most important reason for having a synchronous system has to do with the temporal nature of the human brain. There is a direct causal correlation between the temporal nature of the brain and program comprehensibility. Most of us may not think of the world that we sense as being temporally deterministic and predictable but almost all of it is. If it weren’t, we would have a hard time making sense of it and adapting to it. Note that, here, I am speaking of the macroscopic world of our senses, not the microscopic quantum universe, which is known to be probabilistic. For example, as we scan a landscape with our eyes, the relative motion of the objects in our visual field occurs according to the laws of optics and perspective. Our visual cortex is genetically wired to learn these deterministic temporal correlations. Once the correlations are learned, the newly formed neural structures become fixed and they can then be used to instantly recognize previously learned patterns every time they occur.

The point I am driving at is that the brain is exquisitely programmed to recognize deterministic temporal patterns within an evolving sensory space. Pattern predictability is the key to comprehension and behavioral adaptation. This is the main reason that multithreaded programs are so hard to write and maintain: they are unpredictable. The brain finds it hard to learn and understand unpredictable patterns. It needs stable temporal relationships in order to build the corresponding neural correlations. It is partially for this reason that I claim that, given a synchronous execution environment, the productivity of future parallel programmers will be several orders of magnitude greater than that of their sequential programming predecessors.

**Synchronous Processing and Load Balancing**

An astute reader wrote to me a few days ago to point out a potential problem with parallel synchronous processing. During any given cycle, the cores will be processing a variety of operations (elementary actions). Not all the operations will last an equal number of real time clock cycles. An addition might take two or three cycles while a multiplication might take ten cycles. The reader asked, does this mean that a core that finishes first has to stay idle until all the others are finished? The answer is, not at all. And here is why. Until and unless technology advances to the point where every operator is its own processor (the ultimate parallel system), a multicore processor will almost always have to execute many more operations per parallel cycle than the number of available cores. In other words, most of the times, even in a thousand-core processor, a core will be given dozens if not hundreds of operations to execute within a given parallel cycle. The reason is that the number of cores will never be enough to satisfy our need for faster machines, as we will always find new processor-intensive applications that will push the limits of performance. The load balancing mechanism of a multicore processor must be able to mix the operations of different durations among the cores so as to achieve a near perfect balance overall. Still, even in cases when the load balance is imperfect, the performance penalty will be insignificant compared to the overall load. Good automatic load balancing must be a priority of
multicore research. This is the reason that I am so impressed with Plurality’s load-balancing claim for its Hypercore processor. However, as far as I can tell, Plurality does not use a synchronous software model. They are making a big mistake in this regard, in my opinion.

Conclusion

In conclusion, I will reiterate my conviction that the designers of future parallel systems will have to adopt a synchronous processing model. Synchronous processing is a must, not only for reliability, but for program comprehension and programmer productivity as well. Of course, the adoption of a pure, fine-grain, synchronous software model has direct consequences on the design of future multicore processors. In the next article, I will go over the reasons that the future of parallel computing is necessarily reactive.

See Also:
Nightmare on Core Street
Parallel Computing: The End of the Turing Madness
Parallel Computing: Why the Future Is Non-Algorithmic
Parallel Computing: Why the Future Is Reactive
Why Parallel Programming Is So Hard
Parallel Programming, Math and the Curse of the Algorithm
The COSA Saga

Parallel Computing: Why the Future is Reactive

Reactive vs. Non-Reactive Systems

A reactive system is one in which every stimulus (discrete change or event) triggers an immediate response within the next system cycle. That is to say, there is no latency between stimulus and response. Algorithmic software systems are only partially reactive. Even though an operation in an algorithmic sequence reacts immediately to the execution of the preceding operation, it often happens that a variable is changed in one part of the system but the change is not sensed (by calling a comparison operation) until later. In other words, in an algorithmic program, there is no consistent, deterministic causal link between a stimulus and its response.

The End of Blind Code

Algorithmic systems place a critical burden on the programmer because he or she has to remember to manually add code (usually a call to a subroutine) to deal with a changed variable. If an application is complex or if the programmer is not familiar with the code, the probability that a modification will introduce an unforeseen side effect (bug) is much higher. Sometimes, even if the programmer remembers to add code to handle the change, it may be too late. I call blind code any portion of an application that does not get automatically and immediately notified of a relevant change in a variable.
Potential problems due to the blind code problem are so hard to assess and can have such catastrophic effects that many system managers would rather find alternative ways around a deficiency than modify the code, if at all possible. The way to cure blind code is to adopt a reactive, non-algorithmic software model. In a reactive programming system, a change in a variable is sensed as it happens and, if necessary, a signal is broadcast to every part of the system that depends on the change. It turns out that the development tools can automatically link sensors and effectors at design time so as to eliminate blind code altogether. See Automatic Elimination of Blind Code in Project COSA for more info on the use of sensor/effector association for blind code elimination.

Conclusion

The synchronous reactive software model is the future of parallel computing. It enforces temporal determinism and eliminates blind code and all the reliability problems that plague conventional algorithmic software. In addition, it is ideally suited to the creation of highly stable and reusable plug-compatible software modules. Drag’m and drop’m. These easy to use, snap-together modules will encourage the use of a plug-and-play, trial-and-error approach to software construction and design. Rapid application development will never be the same. This is what Project COSA is all about. Unfortunately, a truly viable reactive system will have to await the development of single and multicore processors that are designed from the ground up to support the non-algorithmic software model. Hopefully, the current multicore programming crisis will force the processor industry to wake up and realize the folly of its ways.

See Also:
Nightmare on Core Street
Parallel Computing: The End of the Turing Madness
Parallel Programming: Why the Future Is Synchronous
Parallel Computing: Why the Future Is Non-Algorithmic
Why Parallel Programming Is So Hard
Parallel Programming, Math and the Curse of the Algorithm
The COSA Saga

Why Parallel Programming Is So Hard

The Parallel Brain

The human brain is a super parallel signal-processing machine and, as such, it is perfectly suited to the concurrent processing of huge numbers of parallel streams of sensory and proprioceptive signals. So why is it that we find parallel programming so hard? I will argue that it is not because the human brain finds it hard to think in parallel, but because what passes for parallel programming is not parallel programming in the first place. Switch to a true parallel programming environment and the problem will disappear.
**Fake Parallelism**

What is the difference between a sequential program and a parallel program? A sequential program is an algorithm or a list of instructions arranged in a specific order such that predecessors and successors are implicit. Is there such a thing as a parallel algorithm? In my opinion, the term ‘parallel algorithm’ is an oxymoron because an algorithm, at least as originally defined, is a sequence of steps. There is nothing parallel about algorithms whether or not they are running concurrently on a single processor or on multiple processors. A multithreaded application consists of multiple algorithms (threads) running concurrently. Other than the ability to share memory, this form of parallelism is really no different than multiple communicating programs running concurrently on a distributed network. I call it fake parallelism.

**True Parallelism**

In a truly parallel system, all events are synchronized to a global clock so that they can be unambiguously identified as being either concurrent or sequential. Synchronization is an absolute must in a deterministic parallel system, otherwise events quickly get out of step and inferring temporal correlations becomes near impossible. Note that ‘synchronous processing’ is not synonymous with ‘synchronous messaging’. A truly parallel system must use asynchronous messaging; otherwise the timing of events becomes chaotic and unpredictable. The human brain is a temporal signal processing network that needs consistent temporal markers to establish correlations. While single thread programs provide adequate temporal (sequential) cues, concurrent threads are non-deterministic and thus concurrent temporal cues are hard to establish, which leads to confusion. See also Parallel Programming: Why the Future Is Synchronous for more on this subject.

It is beneficial to view a computer program as a communication system in which elementary processes send and receive signals to one another. In this light, immediately after execution, an operation (predecessor) in an algorithm sends a signal to the next operation (successor) in the sequence meaning essentially, 'I'm done; now it's your turn'. Whereas in an algorithmic program, every element or operation is assumed to have only one predecessor and one successor, by contrast, in a parallel program, there is no limit to the number of predecessors or successors an element can have. This is the reason that sequential order must be explicitly specified in a parallel program. Conversely, concurrency is implicit, i.e., no special construct is needed to specify that two or more elements are to be executed simultaneously.

**Composition vs. Decomposition**

The common wisdom in the industry is that the best way to write a parallel program is to break an existing sequential program down into multiple threads that can be assigned to separate cores in a multicore processor. Decomposition, it seems, is what the experts are recommending as the correct method of parallelization. However, this begs a couple of questions. If composition is the proper method of constructing sequential programs, why should parallel programs be any different? In other words, if we use sequential elements or components to build a sequential program, why should we not use parallel elements or components to build parallel programs? If
the compositional approach to software construction is known to work in sequential programs, it follows that the same approach should be used in parallel software construction. It turns out that signal-based parallel software lends itself well to the use of plug-compatible components that can snap together automatically. Composition is natural and easy. Decomposition is unnatural and hard.

**Conclusion**

In conclusion, the reason that parallel programming is hard is that it is not what it is claimed to be. As soon as parallel applications become implicitly parallel, synchronous and compositional in nature, parallel programming will be at least an order of magnitude easier than sequential programming. Debugging is a breeze in a deterministic environment, cutting development time considerably.

**See Also:**
- How to Solve the Parallel Programming Crisis
- Why I Hate All Computer Programming Languages
- Nightmare on Core Street
- Parallel Programming: Why the Future Is Synchronous
- Parallel Programming: Why the Future Is Non-Algorithmic
- Parallel Programming, Math and the Curse of the Algorithm
- Parallel Computing: Why the Future Is Compositional

---

**Nightmare on Core Street, Part I**

**Part I, II, III, IV, V**

The Parallel Programming Crisis

**Panic in Multicore Land**

There is widespread disagreement among experts on how best to design and program multicore processors. Some, like senior AMD fellow, Chuck Moore, believe that the industry should move to a new model based on a multiplicity of cores optimized for various tasks. Others (e.g., Anant Agarwal, CTO of Tilera Corporation) disagree on the grounds that heterogeneous processors would be too hard to program. Some see multithreading as the best way for most applications to take advantage of parallel hardware. Others (Agarwal) consider threads to be evil. The only emerging consensus seems to be that multicore computing is facing a major crisis. Here’s a short excerpt from an interview conducted by Dr. Dobb’s Journal with Ryan Schneider, CTO and co-founder of Acceleware:

**DDJ:** It seems like a system running, say a NVIDIA GPU and a many-core CPU could get pretty complicated to program for. If so, what's a developer to do?
RS: Hide. Look for a different job. Day trade on the stock market... Personally, I find that the fetal position helps. :) In all seriousness though, this is a nasty problem. Your question really describes a heterogeneous system, and most ISVs etc. are probably having enough trouble squeezing decent performance/multiples out of a quad-core CPU without adding another, different beast into the mix.

Schneider is not very optimistic about the future of parallel programming. He goes on to say, “Ultimately there is no easy solution to developing parallel systems.” He’s not alone in his pessimism. In a recent EETIMES article titled “Multicore puts screws to parallel-programming models”, AMD’s Moore is reported to have said that “the industry is in a little bit of a panic about how to program multicore processors, especially heterogeneous ones.”

**Incompatible Beasts and Hideous Monsters**

The main problem with multicore processors is that they are hard to program. In addition, there is a huge legacy of software applications that cannot automatically take advantage of parallel processing. That being said, what is remarkable, in my view, is that there is currently no single architecture and/or programming model that can be used universally across all types of applications. Multicore processors come in essentially two incompatible flavors, MIMD (multiple instructions, multiple data) and SIMD (single instruction, multiple data). Neither flavor is optimal for every situation. Logic dictates that universality should be the primary objective of multicore research. Yet, amazingly, industry leaders like Intel and AMD are now actively pushing the field toward a hybrid (i.e., heterogeneous) type of parallel processor, a truly hideous monster that mixes both MIMD and SIMD cores on a single die. It is obvious, at least from my perspective, why the industry is in a crisis: they don’t seem to have a clue as to the real nature of the problem.

In Part II of this five-part article, I will go over the pros and cons of the MIMD parallel programming model as it is currently used in multicore CPUs. In the meantime, please read How to Solve the Parallel Programming Crisis to get an idea of where I am going with this.

**Nightmare on Core Street, Part II**

**Part I, II, III, IV, V**

MIMD

**Recap**

In Part I, I wrote that the computer industry is in a sort of panic, their planned transition from sequential computing to parallel processing having hit a brick wall. The existing multicore architectures are not only hard to program, most legacy applications cannot take advantage of the parallelism. Some experts, especially MIT Professor [Anant Agarwal](https://www.mit.edu/people/ananthag.html), CTO and co-founder of...
Tilera Corporation, are adamant that the industry needs to come up with a new software model because current thread-based operating systems that use cache coherency snooping will not scale up to the many-core processors of the future (source: EETimes). I agree with Professor Agarwal. In this installment, I will describe the difference between fine and coarse grain parallelism, the origin of threads and the pros and cons of thread-based MIMD multicore processors. MIMD simply means that the parallel cores can execute different instructions on multiple data simultaneously.

**Fine Grain Vs. Coarse Grain**

General-purpose multicore processors (the kind built in the newest laptops, desktops and servers) use an MIMD architecture. These processors implement coarse-grain parallelism. That is to say, applications are divided into multiple concurrent modules or threads of various sizes. Each core can be assigned one or more threads so as to share the load; this results in faster processing. By contrast, in fine-grain parallelism, applications are broken down to their smallest constituents, i.e., the individual instructions. Ideally, these instructions can be assigned to separate cores for parallel execution. Fine grain parallelism is much more desirable than coarse-grain parallelism because it makes it possible to parallelize well-known functions like those used in array sorting or tree searching.

**Threads**

Threads are a legacy of the early years of electronics digital computing. They stem from an idea called multitasking that was originally invented to eliminate a problem with sequential batch processing. In the old days, multiple programs (jobs) were stacked in a batch and a job controller was used to feed them into the computer to be executed one after the other. This was time consuming and very expensive. Multitasking made it possible for multiple jobs to run simultaneously on the same sequential processor. The rationale is that the processor is so fast that it can switch execution from one concurrent task to another many times a second. Each task would have its own memory space and would behave as if it had the processor entirely to itself. It did not take long for someone to figure out that a single application could be divided into multiple concurrent internal mini-tasks running in the same memory space. These are called threads.

**The Good**

Even though multitasking and multithreading were never intended to be a parallel programming model, this is nevertheless the model that most major multicore CPU vendors have embraced. Early on, everybody understood that threads and/or tasks could be divided among multiple parallel cores running concurrently and that it would result in faster processing. In addition, threads provided a direct evolutionary path from single-core to multicore computing without upsetting the cart too much, so to speak. Many existing applications that already used threads extensively could make the transition with little effort and programmers could continue to use the same old compilers and languages. Even non-threaded applications could take advantage of the new CPUs if several of them can be kept running concurrently on the same multicore computer.
The Bad

The need for programming continuity and for compatibility with the existing code base is the reason that companies like AMD and Intel are trying their best to encourage programmers to use as many threads as possible in their code. There are many problems with threads, however, too many to discuss without writing a long thesis. So I’ll just mention a few here. The biggest problem is programming difficulty. Even after decades of research and hundreds of millions of dollars spent on making multithreaded programming easier, threaded applications are still a pain in the ass to write. Threads are inherently non-deterministic and, as a result, they tend to be unreliable and hard to understand, debug and maintain. In addition, the coarse-grain parallelism used in threaded programs is not well suited to data-parallel applications such as graphics processing, simulations and a whole slew of scientific computations. These types of programs run much faster in a fine-grain parallel environment. Another problem is that a huge number of legacy applications were not written with threads in mind and cannot take advantage of the processing power of multicore CPUs. Converting them into threaded applications is a monumental undertaking that will prove to be prohibitively costly and time consuming. Rewriting them from scratch will be equally costly.

Obviously, thread-based MIMD parallelism is not the answer the industry is looking for, wishful thinking notwithstanding. In Part III, I will examine the pros and cons of SIMD and heterogeneous multicore processors.

Nightmare on Core Street, Part III

Part I, II, III, IV, V

SIMD

Recap

In Part II of this five-part article I went over the pros and cons of MIMD multicore CPU architectures that are designed to run coarse-grain, multithreaded applications. Current MIMD multicore architectures are an evolutionary step from single core architectures in that they make it easy for existing threaded applications to make the transition to multicore processing without much modification. The bad thing is that multithreaded applications are unreliable and too hard to program and maintain. In addition, coarse-grain parallelism is not well suited to many important types of computations such as graphics and scientific/engineering simulations. Here I describe the advantages and disadvantages of SIMD (single instruction, multiple data) parallelism, also known as data level or vector parallelism.
Most multicore processors can be configured to run in SIMD mode. In this mode, all the cores are forced to execute the same instruction on multiple data simultaneously. SIMD is normally used in high performance computers running scientific applications and simulations. This is great when there is a need to perform a given operation on a large data set and in situations when programs have low data dependencies, i.e., when the outcome of an operation rarely affect the execution of a succeeding operation.

Many graphics processors use SIMD because graphics processing is data intensive. If you have a computer with decent graphics capabilities, chances are that it has a special co-processor that uses SIMD to handle the graphics display. Companies like NVIDIA and ATI (now part of AMD) make and sell SIMD graphics processors. In the last few years, many people in the business have come to realize that these dedicated graphics processors can do more than just handle graphics. They can be equally well suited to non-graphical scientific and/or simulation applications that can benefit from a similar data-flow approach to parallel processing.

The Good

One of the advantages of SIMD processors is that, unlike general-purpose MIMD multicore processors, they handle fine-grain parallel processing, which can result in very high performance under certain conditions. Another advantage is that SIMD processing is temporally deterministic, that is to say, operations are guaranteed to always execute in the same temporal order. Temporal order determinism is icing on the parallel cake, so to speak. It is a very desirable property to have in a computer because it is one of the essential ingredients of stable and reliable software.

The Bad

The bad thing about SIMD is that it is lousy in situations that call for a mixture of operations to be performed in parallel. Under these conditions, performance degrades significantly. Applications that have high data dependencies will also perform poorly. I am talking about situations where a computation is performed based on the outcome of a previous computation. An SIMD processor will choke if you have too many of these. Unfortunately, many applications are like that.

Hybrid Processors

The latest trend in multicore CPU design is to mix MIMD and SIMD processing cores on the same die. AMD has been working hard on its Fusion processor, which they plan to release in 2009. Not to be outdone, Intel is quietly working on its own GPU/CPU multicore offering, the Larrabee (recently canceled). Indeed, Intel started the trend or mixing graphics and general purpose cores with its failed MMX Pentium processor a while back. Sony, Toshiba and IBM already have a multicore processor that mixes SIMD and MIMD processing cores on one chip. It is called the Cell processor and it is the processor being shipped with Sony’s PlayStation 3 video game console.

The idea behind these so-called heterogeneous processors is that their designers believe that
SIMD and MIMD complement each other’s capabilities, which is true. In addition, having both types of cores on the same chip increases performance because communication between cores is faster since it does not have to use a slow external bus. The problem with hybrid processors, however, is that programming them is extremely painful. In the past, I have compared it to pulling teeth with a crowbar. This is something that the industry is acutely aware of and hundreds of millions of dollars are currently being spent on finding a solution that will alleviate the pain.

**Fundamental Flaw**

In my opinion, all of the current approaches to multicore parallel processing will fail in the end and they will fail miserably. They will fail because they are fundamentally flawed. And they are flawed because, 150 years after Babbage designed the first general-purpose computer, neither academia nor the computer industry has come to understand the true purpose of a CPU. In Part IV of this series, I will explain why I think the computer industry is making a colossal error that will come back to haunt them. Stay tuned.

**Nightmare on Core Street, Part IV**

*Part I, II, III, IV, V*

Gambling on Threads

**Recap**

In Part I, II and III of this five-part article, I wrote that the computer industry is in a panic because there is no easy way to program multicore processors. I also went over the advantages and disadvantages of MIMD, SIMD and heterogeneous multicore architectures. In my opinion, what is needed is a new programming model/processor architecture that combines the strengths of both SIMD and MIMD while eliminating their weaknesses. I am proposing a universal parallel computing model that uses fine-grain, deterministic parallelism in an MIMD configuration to handle anything you can throw at it. I will describe what I have in mind in Part V. What follows is an explanation of why I think the computer industry’s current multicore strategy will turn out to be a colossal failure.

**High Stakes Gamble on the Future of Parallel Computing**

The Chief Software Evangelist for Intel, James Reinders, believes that the key to successful parallel programming centers around scaling, debugging and future proofing (source: Dr. Dobb’s Journal). To that list I would add automatic load balancing and ease of programming. Of course, being an Intel evangelist and the author of the new book “Intel Threading Building Blocks”, Reinders is necessarily biased. When he says, “think parallel”, what he means is, “think threads”. Reinders strongly advises programmers to use threaded libraries or, better yet, Intel’s own
threading building blocks. He is pushing the use of threads because Intel’s multicore processors are pretty much useless for non-threaded parallel applications. The same goes for AMD and other multicore vendors. These guys are so desperate they’re giving all their code away, free.

Let’s say you decided to listen to the industry pundits and you painstakingly rewrote your entire legacy code to use multiple threads and got it stable enough to be useful. That would make Intel and AMD very happy and your code would indeed run faster on their multicore systems. But what if they are wrong about the future of parallel programming? What if (horror of horrors) the future is not multithreaded? Would all your time and effort have been in vain? The answer is yes, of course. All right, I am not trying to be an alarmist for the hell of it. I am trying to drive home an important point, which is this: Intel and AMD have already made the fateful decision to go the multithreading route and they are now irreversibly committed to it. If their gamble does not win out, it will undoubtedly mean tens of billions of dollars in losses for them and their customers. That would be a disaster of enormous proportions and they know it. This is what Reinders really means by ‘future proofing’. He is more concerned about future-proofing Intel's multicore CPUs than anything else. So if you listen to evangelists like Reinders (or AMD's Chuck Moore), you do so at your own risk because the sad reality (see below) is that the industry does not have a clue as to the real future of parallel computing. There’s something sadly pathetic about the blind leading the blind and both falling into the same ditch.

**Persistent Problem**

The parallel programming problem is not a new one. It has been around for decades. In a [GCN article](https://www.gcn.com/article/77437) titled “The Multicore Challenge”, [Cray J. Henry](https://www.gcn.com/author/cray-j-henry), the director of the U.S. Defense Department's High Performance Computing Modernization Program, wrote:

> The challenge of writing parallel software has been the key issue for the computational science and supercomputing community for the last 20 years. There is no easy answer; creating parallel software applications is difficult and time consuming.

This is rather telling. For two decades, some of the smartest people in the computer research community have been using threads to program super high-performance parallel computers. Even after spending untold zillions of dollars and man-hours on the parallel programming problem, they still have no answer. It is still “difficult and time consuming.” What is wrong with this picture? The answer should be obvious to anybody who is not blinded by reality: the academic research community has no idea what the solution might be. They are not any closer to a solution than they were when they started. They have spent twenty long years (an eternity in this business) trying to fit a square peg into a round hole! And, incredibly, they are still at it.

So what makes either Intel or AMD so confident that threads are the way to go? What is the theoretical basis of their multicore strategy? Well, the truth is that they are not confident at all. They have no leg to stand on, really. If they had solved the problem, they would not continue to pour hundreds of millions of dollars into research labs around the globe to find a solution. They are not pushing threads because they think it is the right way to do things. They are pushing
threads because they have no alternative. And they have no alternative because they are following in the footsteps of academia, the same people who, in my opinion, got everybody into this mess in the first place. It is one more example of the blind leading the blind.

**The Hidden Nature of Computing**

The way I see it, if computer scientists had started out with the correct computing model (yes, there is such a thing, even with a single-core processor), there would be no crisis at all and you would not be reading this article. Adding more cores to a processor would have been a relatively painless evolution of computer technology, a mere engineering problem. Unfortunately, the entire computer industry still has the same conception of what a computer should be that Charles Babbage and Lady Ada Lovelace had one hundred and fifty years ago! Is it any wonder that we are in a crisis? To solve the parallel programming problem, the industry first needs to understand the true nature of computing and then reinvent the computer accordingly, both software and hardware. There are no two ways about it. And the longer they wait to wake up and realize their folly, the worse the problem is going to get.

The true nature of computing has nothing to do with universal Turing machines or the Turing computability model or any of the other stuff that academics have intoxicated themselves with. I have already written plenty about this subject elsewhere and it does not make sense to repeat myself here. Suffice it to say that, as soon as one comes to grips with the true nature of computing, it becomes immediately clear that the multithreaded approach to parallel computing is a complete joke, an abomination even. If you are wise, you would take heed not to listen to the likes of Mr. Reinders or to anybody who goes around selling what I call the “threaded snake oil”. As the native Americans used to say, they speak with a forked tongue. :-) Threads are not part of the future of computing. Using threads for so-called future proofing is a disaster in the making, wishful thinking notwithstanding. Reality can be cruel that way.

There is a way to build self-balancing, MIMD, multicore computers to implement fine-grain, reactive, deterministic parallelism that will not only solve the parallel programming problem but the reliability problem as well. I’ll go over this in Part V.

**Nightmare on Core Street, Part V**

[Part I, II, III, IV, V]

The COSA Software Model

**Recap**

In Part IV, I wrote that the reason that the computer industry’s multicore strategy will not work is that it is based on multithreading, a technique that was never intended to be the basis of a parallel software model, only as a mechanism for executing multiple sequential (not parallel) algorithms
concurrently. I am proposing an alternative model that is inherently non-algorithmic, deterministic and parallel. It is called the COSA software model and it incorporates the qualities of both MIMD and SIMD parallelism without their shortcomings. The initial reason behind COSA was to solve one of the most pressing problems in computer science today, software unreliability. As it turns out, COSA addresses the parallel programming problem as well.

**The COSA Model**

Any serious attempt to formulate a parallel software model would do well to emulate parallel systems in nature. One such system is a biological neural network. Imagine an interconnected spiking (pulsed) neural network. Each elementary cell (neuron) in the network is a parallel element or processor that waits for a discrete signal (a pulse or spike) from another cell or a change in the environment (event), performs an action (executes an operation) on its environment and sends a signal to one or more cells. There is no limit to the number of cells that can be executed simultaneously. What I have just described is a behaving system, i.e., a reactive network of cells that use signals to communicate with each other. This is essentially what a COSA program is. In COSA, the cells are the operators; and these can be either effectors (addition, subtraction, multiplication, division, etc…) or sensors (comparison or logic/temporal operators). The environment consists of the data variables and/or constants. Below is an example of a COSA low-level module that consists of five elementary cells.

![COSA Diagram](image_url)

"While Loop"

Alternatively, a COSA program can be viewed as a logic circuit with lines linking various gates (operators or cells) together. Indeed, a COSA program can potentially be turned into an actual electronics circuit. This aspect of COSA has applications in future exciting computing technologies like the one being investigated by the [Phoenix Project](#) at Carnegie Mellon University. The main difference between a COSA program and a logic circuit is that, in COSA, there is no signal racing. All gates are synchronized to a global virtual clock and signal travel times are equal to zero, i.e., they occur within one cycle. A global clock means that every operation is assumed to have equal duration, one virtual cycle. The advantage of this convention is that COSA programs are 100% deterministic, meaning that the execution order (concurrent or sequential) of the operations in a COSA program is guaranteed to remain the same. Temporal order determinism is essential for automated verification purposes, which, in turn, lead to rock-solid reliability and security.
The COSA Process Emulation

Ideally, every COSA cell should be its own processor, like a neuron in the brain or a logic gate. However, such a super-parallel system must await future advances. In the meantime we are forced to use one or more very fast processors to do the work of multiple parallel cells. In this light, a COSA processor (see below) should be seen as a cell emulator. The technique is simple and well known. It is used in neural networks, cellular automata and simulations. It is based on an endless loop and two cell buffers. Each pass through the loop represents one cycle. While the processor is executing the cells in one buffer, the downstream cells to be processed during the next cycle are appended to the other buffer. As soon as all the cells in the first buffer are processed, the buffers are swapped and the cycle begins anew. Two buffers are used in order to prevent the signal racing conditions that would otherwise occur.

The COSA Processor

As seen above, we already know how to emulate deterministic parallel processes in software and we can do it without the use of threads. It is not rocket science. However, using a software loop to emulate parallelism at the instruction level would be prohibitively slow. For performance purposes, the two buffers should be integrated into the COSA processor and the cell emulation performed by the processor. In a multicore processor, each core should have its own pair of buffers.

Comparison Between COSA and Other Parallel Software Models

<table>
<thead>
<tr>
<th></th>
<th>MIMD (threaded)</th>
<th>SIMD</th>
<th>COSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deterministic</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Fine-Grain Parallelism</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Robust</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Easy to Program</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Universal</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal-Based</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Self-Balancing</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Auto-Scalable</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Transparent</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Verifiable</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Multiple Instructions</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Asynchronous Messages</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Ease of programming is one of the better attributes of COSA. The reason is that programming in COSA is graphical and consists almost entirely of connecting objects together. Most of the time, all that is necessary is to drag the object into the application. High-level objects are plug-compatible and know how to connect themselves automatically.

The figure above is an example of a COSA high-level module under construction. Please take a look at the Project COSA web page for further information.

See Also:
- How to Solve the Parallel Programming Crisis
- Parallel Computing: The End of the Turing Madness
- Parallel Programming: Why the Future Is Non-Algorithmic
- Parallel Programming: Why the Future Is Synchronous
- Parallel Computing: Why the Future Is Reactive
- Why Parallel Programming Is So Hard
- Parallel Programming, Math and the Curse of the Algorithm
- The COSA Saga

The Death of Larrabee or Intel, I Told You So

I Had Foreseen It
Back in June of 2009, I wrote the following comment in response to a New York Times' Bits blog article by Ashlee Vance about Sun Microsystems' cancellation of its Rock chip project:

[...]The parallel programming crisis is an unprecedented opportunity for a real maverick to shift the computing paradigm and forge a new future. It’s obvious that neither Intel nor AMD have a solution. You can rest assured that Sun’s Rock chip will not be the last big chip failure in the industry. Get ready to witness Intel’s Larrabee and AMD’s Fusion projects come crashing down like the Hindenburg.

Anybody who thinks that last century’s multithreading CPU and GPU technologies will survive in the age of massive parallelism is delusional, in my opinion. After the industry has suffered enough (it’s all about money), it will suddenly dawn on everybody that it is time to force the baby boomers (the Turing Machine worshipers) to finally retire and boldly break away from 20th century’s failed computing models.

Sun Microsystems blew it but it’s never too late. Oracle should let bygones be bygones and immediately fund another big chip project, one designed to rock the industry and ruffle as many feathers as possible. That is, if they know what’s good for them.

Will Oracle do the right thing? I doubt it. Now that Intel has announced the de facto demise of Larrabee, my prediction is now partially vindicated. Soon, AMD will announce the cancellation of its Fusion chip and my prediction will then be fully vindicated. Fusion is another hideous heterogeneous beast that is also destined for oblivion. There is no escaping this, in my opinion, because the big chipmakers are going about it the wrong way, for reasons that I have written about in the last few years. I see other big failures on the horizon unless, of course, the industry finally sees the light. But I am not counting on that happening anytime soon.

Goodbye Larrabee

Sorry Intel. I am not one to say I told you so, but I did. Goodbye Larrabee and good riddance. Nice knowing ya even if it was for such a short time. Your only consolation is that you will have plenty of company in the growing heap of failed processors. Say hello to IBM's Cell Processor when you arrive.

See Also:
How to Solve the Parallel Programming Crisis
Nightmare on Core Street
Parallel Computing: The End of the Turing Madness
Jeff Han and the Future of Parallel Programming

Forget computer languages and keyboards. I have seen the future of computer programming and this is it. The computer industry is on the verge of a new revolution. The old algorithmic software model has reached the end of its usefulness and is about to be replaced; kicking and screaming, if need be. Programming is going parallel and Jeff Han’s multi-touch screen interface technology is going to help make it happen. The more I think about it, the more I am convinced that Han’s technology is the perfect interface for the COSA programming model. COSA is about plug-compatible objects connected to other plug-compatible objects. Just drag 'em and drop 'em. What better way is there to compose, manipulate and move objects around than Han’s touch screen interface?

This COSA component that I drew years ago looks painfully primitive compared to Han's images but I can imagine a bunch of COSA cells and components being morphed into really cool 3-D objects that can be easily rotated, opened or moved around on a multi-touch screen. A complex COSA program could be navigated through as one would navigate in a first-person 3-D video game. The COSA programmer could jump inside the program and look at the cells firing, not entirely unlike what it would look like moving through a maze of firing neurons inside the human brain. Add a speech interface and eliminate the keyboard altogether. I never liked keyboards anyway. The computer will not come of age until keyboards (a relic from the eighteenth century) go the way of the slide rule. This is exciting.

How to Make Computer Geeks Obsolete

Charles Simonyi

I just finished reading a very interesting article over at MIT Technology Review about former Microsoft programming guru and billionaire, Charles Simonyi. Essentially, Simonyi, much like everyone else in the computer business with a head on their shoulders, realized a long time ago
that there is something fundamentally wrong with the way we construct software. So, while working at Microsoft, he came up with a new approach called intentional programming to attack the problem. Seeing that his bosses at Microsoft were not entirely impressed, Simonyi quit his position and founded his own company, Intentional Software Corporation, to develop and market the idea. It’s been a while, though. I am not entirely sure what’s holding things up at Intentional but methinks they may have run into a brick wall and, knowing what I know about Simonyi’s style, he is probably doing some deconstruction and reconstruction.

**Sorry, Charlie, Geeks Love the Dark Ages**

There is a lot of secrecy surrounding the project but, in my opinion, Simonyi and the folks at Intentional will have to come around to the conclusion that the solution will involve the use of graphical tools. At a recent Emerging Technology Conference at MIT, Simonyi tried to convince programmers to leave the Dark Ages, as he put it. His idea is to bring the business people (i.e., the domain experts) into software development. I applaud Simonyi’s courage but my question to him is this; if your goal is to turn domain experts into developers, why give a talk at a techie conference? The last thing a computer geek wants to hear is that he or she may no longer be needed. In fact, based on my own personal experience, the geeks will fight Simonyi every step of the way on this issue. Ironically enough, geeks are the new Luddites of the automation age. Unfortunately for the geeks but fortunately for Simonyi, he is not exactly looking for venture capital. With about a billion dollars in his piggy bank, a mega-yacht in the bay and Martha Stewart at his side, the man can pretty much do as he pleases.

**The Future of Software Development**

In my opinion, Simonyi does not go far enough. In his picture of the future of software development, he sees the domain expert continuing to work side by side with the programmer. In my picture, by contrast, I see only the domain expert gesturing in front of one of Jeff Han’s multitouch screens and speaking into a microphone. The programmer is nowhere to be seen. How can this be? Well, the whole idea of automation is to make previous expertise obsolete so as to save time and money, right? Programmers will have joined blacksmiths and keypunch operators as the newest victims of the automation age. Sorry. I am just telling it like I see it. But don't feel bad if you're a programmer because, eventually, with the advent of true AI, even the domain expert will disappear from the picture.

**Intentional Design vs. Intentional Programming**

The way I see it, future software development will be strictly about design and composition. Forget programming. I see a software application as a collection of concurrent, elementary behaving entities organized into plug-compatible modules that communicate via message connectors. Modules are like pieces in a giant picture puzzle. The main difference is that modules are intelligent: they know how to connect to one another. For example, let’s say you are standing in front of your beautiful new multi-touch screen and you are composing a new business application. Suppose you get to a point where you have some floating-point data that you want the program to display as a bar graph. You simply say, “give me bar graph display module” into the microphone. Problem is, there are all sorts of bar graph display modules
available and the computer displays them all on the right side of the screen. No worry. You simply grab all of them with your right hand and throw them into your app space like confetti driven by the wind. And, lo and behold, the one that is compatible with your data magically and automatically connects itself to your app and voila! You smile and say “clean up!” and all the incompatible modules disappear, as if by magic. You suddenly remember Tom Cruise’s character in the movie, Minority Report, and you can barely keep from laughing. Creating software is so much fun! This tiny glimpse of the future of software development is brought to you by Project COSA.

In conclusion, my advice to Charles Simonyi is to start thinking in terms of reactive, plug-compatible parallel objects and to get somebody like Jeff Han on board. Also, stop trying to convince computer geeks.

See Also:
- Why I Hate All Computer Programming Languages
- COSA: A New Kind of Programming
- Why Parallel Programming Is So Hard
- How to Solve the Parallel Programming Crisis

**Why I Hate All Computer Programming Languages**

**That’s All I Want to Do!**

I hate computer languages because they force me to learn a bunch of shit that are completely irrelevant to what I want to use them for. When I design an application, I just want to build it. I don’t want to have to use a complex language to describe my intentions to a compiler. Here is what I want to do: I want to look into my bag of components, pick out the ones that I need and snap them together, and that’s it! That’s all I want to do.

I don’t want to know about how to implement loops, tree structures, search algorithms and all that other jazz. If I want my program to save an audio recording to a file, I don’t want to learn about frequency ranges, formats, fidelity, file library interface, audio library interface and so forth. This stuff really gets in the way. I just want to look into my bag of tricks, find what I need and drag them out. Sometimes, when I meditate about modern computer software development tools, I get so frustrated that I feel like screaming at the top of my lungs: **That is all I want to do!**

**Linguistic Straightjacket**

To me, one of the main reasons that the linguistic approach to programming totally sucks is that it is entirely descriptive by definition. This is a major drawback because it immediately forces you into a straightjacket. Unless you are ready to describe things in the prescribed, controlled format, you are not allowed to program a computer, sorry. The problem with this is that, we
humans are tinkerers by nature. We like to play with toys. We enjoy trying various combinations of things to see how they fit together. We like the element of discovery that comes from not knowing exactly how things will behave if they are joined together or taken apart. We like to say things like, “ooh”, “ah”, or “that’s cool” when we half-intentionally fumble our way into a surprising design that does exactly what we want it to do and more. Computer languages get in the way of this sort of pleasure because they were created by geeks for geeks. Geeks love to spoil your fun with a bunch of boring crap. For crying out loud, I don’t want to be a geek, even if I am one by necessity. I want to be happy. I want to do cool stuff. I want to build cool things. And, goddammit, that’s all I want to do!

**Conclusion**

Unless your application development tool feels like a toy and makes you want to play like a child, then it is crap. It is a primitive relic from a primitive age. It belongs in the Smithsonian right next to the slide rule and the buggy whip. If you, like me, just want to do fun stuff, you should check out Project COSA. COSA is about the future of programming, about making programming fast, rock solid and fun.

**See Also:**
- Parallel Computing: Why the Future Is Compositional
- COSA, A New Kind of Programming
- How to Solve the Parallel Programming Crisis
- The COSA Control Hierarchy

**Parallel Computing: The End of the Turing Madness, Part I**

**Part I, II**

**Hail, Turing**

*Alan Turing* is, without a doubt, the most acclaimed of all computer scientists. He is considered to be the father of modern computer science. Soon after his untimely death in 1954 at the age of 41, the computer science community elevated him to the status of a god. Books are written, movies are made and statues are erected in his memory. Turing is so revered that the most coveted prize in computer science, the *A. M. Turing Award*, was named after him. It is worth noting that Turing’s sexual inclinations and religious convictions did little to diminish his notoriety. Homosexual and atheist movements around the world have wasted not time in transforming the man into a martyr and the sad history of his life into a veritable cause célèbre. The end result is that nothing negative may be said about Turing. It is all right to talk about the *Von Neumann bottleneck* but mentioning that the bottleneck was already part and parcel of the Turing machine is taboo. The adulation of Turing is such that criticizing him or his ideas is guaranteed to deal a deathblow to the career of any scientist who would be so foolish.

**Unabashedly Bashing Turing and Loving It**
It is a good thing that I don’t lose any sleep over my reputation in either the scientific community or the computer industry, otherwise I would be in a world of hurt. I am free to bash Turing and his supporters to my heart’s content. I am free to point out that the emperor is buck-naked and ugly even while everyone around me is fawning at his non-existent clothes. As someone with a strong iconoclastic bent, I admit that I enjoy it. Free speech is a wonderful thing. I have argued forcefully and unabashedly in the past that academia’s strange and pathological obsession with Turing is the primary cause of the software reliability and productivity crisis. I now argue even more forcefully that the parallel programming crisis can be directly traced to our having swallowed Turing’s erroneous ideas on computing, hook, line and sinker. Had the computer science community adopted the correct computing model from the start, there would be no crisis and you would not be reading this article and getting offended by my irreverent treatment of Mr. Turing. In fairness to Turing, my criticism is directed mostly toward those who have turned the man into the infallible god that he never was and never claimed to be.

The Not So Universal UTM

Unfortunately for Turing’s admirers, for many years now, the laws of physics have been quietly conspiring against their hero. Sequential processors have reached the limits of their performance due to heat dissipation problems. The computer industry is thus forced to embrace parallel processing as the only way out of its predicament. Parallelism is a good thing but it turns out that programming parallel processors is much easier said than done. In fact, it is such a pain that the big players in the multicore industry are spending hundreds of millions of dollars in research labs around the world in the hope of finding a viable solution. They have been at it for decades without any hint of success in sight. Worse, the leaders of the industry, out of sheer desperation, are now turning to the very people who got everybody into this mess in the first place, none other than the Turing worshipers in academia. It would be funny if it weren’t so pathetic.

Consider that Turing’s ideas on computing, especially the Universal Turing machine (UTM), were strongly influenced by his love of mathematics. He was, first and foremost, a mathematician and, like Babbage and Lady Ada a century before him, he saw the computer primarily as a tool for solving serial math problems. It is highly unlikely that he ever thought of the concept of parallel processing or the idea that a computer might be used for anything other than problem solving and the execution of instruction sequences. The time has come for the computer industry to realize that the UTM is the quintessential sequential computer and, as such, it is ill suited as a model for parallel computing. It is time to devise a truly universal computing machine, an anti-UTM machine if you will, one that can handle both sequential and concurrent processing with equal ease.

Smashing the Turing Machine

The Turing machine cannot be said to be universal because it is a strictly sequential machine by definition whereas the universe is inherently parallel. It is certainly possible to use a UTM to emulate parallel processes. Programmers have been emulating parallelism in such applications as neural networks, video games and cellular automata for decades. However, the emulation is not part of the Turing computing model. It is an ad-hoc abstraction that a programmer can use to
pretend that he or she is performing parallel processing even though the underlying model is sequential. Programmers get away with the pretense because processors are extremely fast. Slow the processor down to a few hundred Hertz and the illusion of parallelism disappears.

One can always argue that having two or more Turing machines running side by side is an example of parallel computation. However, two Turing machines running independently cannot be construed as representing one Universal Turing machine, as defined by Turing. Parallelism is not synonymous with temporal independence. On the contrary, a truly parallel system is one in which all events can be unambiguously determined as being either concurrent or sequential. Even if one could extend the definition of the UTM to include multiple parallel machines, deciding which operations are performed concurrently requires even more ad-hoc additions such as a communication channel between the two machines. Alternatively, one can imagine a Turing like machine with an infinite number of read/write heads on a single infinitely long tape. Still, the Turing model does not provide a deterministic mechanism for the detection of either concurrency or sequentiality and the problem becomes worse with the addition of multiple heads. There is only one way to solve the parallel programming crisis. We must smash the non-universal Turing machine and invent a truly universal alternative. In Part II, I will describe what I mean by a true universal computing machine and do a direct comparison with the UTM.

See Also:
- How to Solve the Parallel Programming Crisis
- Parallel Computing: Why the Future Is Non-Algorithmic

Parallel Computing: The End of the Turing Madness, Part II

Part I, II

Turing Is the Problem, Not the Solution

In Part I, I wrote that Alan Turing is a naked emperor. Consider that the computer industry is struggling with not just one but three major crises [note: there is also a fourth crisis having to do with memory bandwidth]. The software reliability and productivity crises have been around since the sixties. The parallel programming crisis has just recently begun to wreak havoc. It has gotten to the point where the multicore vendors are starting to panic. Turing’s ideas on computation are obviously not helping; otherwise there would be no crises. My thesis, which I defend below, is that they are, in fact, the cause of the industry’s problems, not the solution. What is needed is a new computing model, one that is the very opposite of what Turing proposed, that is, one that models both parallel and sequential processes from the start.

UBM vs. UTM

I have touched on this before in my seminal work on software reliability but I would like to elaborate on it a little to make my point. The computing model that I am proposing is based on
an idealized machine that I call the Universal Behaving Machine or UBM for short. It assumes that a computer is a behaving machine that senses and reacts to changes in its environment.

<table>
<thead>
<tr>
<th>Universal Behaving Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Actors</strong></td>
</tr>
<tr>
<td>Sensor</td>
</tr>
<tr>
<td>Variable</td>
</tr>
</tbody>
</table>

Please read the paragraph on [The Hidden Nature of Computing](#) before continuing. Below, I contrast several characteristics of the UBM with those of the UTM. The Turing machine does not provide for it but I will be gracious and use multithreading as the Turing version of parallel processing.

<table>
<thead>
<tr>
<th><strong>UBM</strong></th>
<th><strong>UTM</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit sequential processing</td>
<td>Implicit sequential processing</td>
</tr>
<tr>
<td>Implicit parallel processing</td>
<td>Explicit parallel processing</td>
</tr>
<tr>
<td>Deterministic</td>
<td>Non-deterministic</td>
</tr>
<tr>
<td>Reactive (change based)</td>
<td>Procedural</td>
</tr>
<tr>
<td>Non-algorithmic</td>
<td>Algorithmic</td>
</tr>
</tbody>
</table>

Although multithreading is not part of the UTM, this is the mechanism that multicore processor vendors have adopted as their parallel processing model. Turing's supporters will argue that parallelism can be simulated in a UTM without threads and they are correct. However, as I explain below, a simulation does not change the sequential nature of the Turing computing model. For an explanation of “non-algorithmic”, please read [Parallel Computing: Why the Future is Non-Algorithimic](#).

**Simulation Does Not a Computing Model Make**

True universality requires that a computing model should handle both serial and parallel computations and events by definition. In other words, both types of computation should be inherent parts of the model. One of the arguments that I invariably get from Turing’s supporters is that the Turing machine is a universal computing model because you can use it to simulate anything, even a parallel computer. This is a rather lame argument because observing that a Turing machine can be used to simulate a parallel computer does not magically transform it into a parallel computing model. This would be like saying that, since a Turing machine can be used to simulate a video game or a chess computer, that it is therefore a video game or a chess-computing model. That is absurd. Simulation does not a model make. Whenever one uses one mechanism to simulate another, one climbs to a new level of abstraction, a new model, one that does not exist at the lower level.

**To Model or to Simulate, That Is the Question**
The Turing machine is a model for a mechanism that executes a sequence of instructions. It does not model a parallel computer, or a tic-tac-toe program or a spreadsheet or anything else, even if it can be used to simulate those applications. The simulation exists only in the mind of the modeler, not in the underlying mechanism. The fallacy of universality is even more transparent when one realizes that a true parallel machine like the UBM does not have to simulate a Turing machine the way the UTM has to simulate the UBM. The reason is that the UBM can duplicate any computation that a Turing machine can perform. In other words, the UTM is an inherent part of the UBM but the opposite is not true.

The Beginning of the End of the Turing Madness

Thomas Kuhn wrote in his book, “The Structure of Scientific Revolutions” that scientific progress occurs through revolutions or paradigm shifts. Max Planck, himself a scientist, said that "a new scientific truth does not triumph by convincing its opponents and making them see the light, but rather because its opponents eventually die, and a new generation grows up that is familiar with it." Last but not least, Paul Feyerabend wrote the following in Against Method: “… the most stupid procedures and the most laughable results in their domain are surrounded with an aura of excellence. It is time to cut them down to size and give them a more modest position in society.”

I think that all the major problems of the computer industry can be attributed to the elitism and intransigence that is rampant in the scientific community. The peer review system is partly to blame. It is a control mechanism that keeps outsiders at bay. As such, it limits the size of the meme pool in much the same way that incest limits the size of the gene pool in a closed community. Sooner or later, the system engenders monstrous absurdities but the community is blind to it. The Turing machine is a case in point. The point that I am getting at is that it is time to eradicate the Turing cult for the sake of progress in computer science. With the parallel programming crisis in full swing, the computer industry desperately needs a Kuhnian revolution. There is no stopping it. Many reactionaries will fight it teeth and nails but they will fall by the wayside. We are witnessing the beginning of the end of the Turing madness. I say, good riddance.

See Also:
How to Solve the Parallel Programming Crisis
Parallel Computing: Why the Future Is Non-Algorithmic

Half a Century of Crappy Computing

Decades of Deception and Disillusion

I remember being elated back in the early 80s when event-driven programming became popular. At the time, I took it as a hopeful sign that the computer industry was finally beginning to see the light and that it would not be long before pure event-driven, reactive programming was
embraced as the universal programming model. Boy, was I wrong! I totally underestimated the
capacity of computer geeks to deceive themselves and everyone else around them about their
business. Instead of asynchronous events and signals, we got more synchronous function calls;
and instead of elementary reactions, we got more functions and methods. The unified approach
to software construction that I was eagerly hoping for never materialized. In its place, we got
inundated with a flood of hopelessly flawed programming languages, operating systems and
CPU architectures, a sure sign of an immature discipline.

The Geek Pantheon

Not once did anybody in academia stop to consider that the 150-year-old algorithmic approach to
computing might be flawed. On the contrary, they loved it. Academics like Fred Brooks decreed
to the world that the reliability problem is unsolvable and everybody worshiped the ground he
walked on. Alan Turing was elevated to the status of a deity and the Turing machine became the
de facto computing model. As a result, the true nature of computing has remained hidden from
generations of programmers and CPU architects. Unreliable software was accepted as the norm.
Needless to say, with all this crap going on, I quickly became disillusioned with computer
science. I knew instinctively what had to be done but the industry was and still is under the firm
political control of a bunch of old computer geeks. And, as we all know, computer geeks believe
and have managed to convince everyone that they are the smartest human beings on earth. Their
wisdom and knowledge must not be questioned. The price [pdf], of course, has been staggering.

In Their Faces

What really bothers me about computer scientists is that the solution to the parallel programming
and reliability problems has been in their faces from the beginning. We have been using it to
emulate parallelism in such applications as neural networks, cellular automata, simulations,
VHDL, Verilog, video games, etc… It is a change-based or event-driven model. Essentially, you
have a global loop and two buffers (A and B) that are used to contain the objects to be processed
in parallel. While one buffer (A) is being processed, the other buffer (B) is filled with the objects
that will be processed in the next cycle. As soon as all the objects in buffer A are processed, the
two buffers are swapped and the cycle repeats. Two buffers are used in order to prevent the
signal racing conditions that would otherwise occur. Notice that there is no need for threads,
which means that all the problems normally associated with thread-based programming are non-
existent. What could be simpler? Unfortunately, all the brilliant computer savants in academia
and industry were and still are collectively blind to it. How could they not? They are all busy
studying the subtleties of Universal Turing Machines and comparing notes.

We Must Reinvent the Computer

I am what you would call a purist when it come to event-driven programming. In my opinion,
everything that happens in a computer program should be event-driven, down to the instruction
level. This is absolutely essential to reliability because it makes it possible to globally enforce
temporal determinism. As seen above, simulating parallelism with a single-core CPU is not
rocket science. What needs to be done is to apply this model down to the individual instruction
level. Unfortunately, programs would be too slow at that level because current CPUs are
designed for the algorithmic model. This means that we must reinvent the computer. We must design new single and multiple-core CPU architectures to directly emulate fine-grained parallelism. There is no getting around it.

**Easy to Program and Understand**

A pure event-driven software model lends itself well to fine-grain parallelism and graphical programming. The reason is that an event is really a signal that travels from one object to another. As every logic circuit designer knows, diagrams are ideally suited to the depiction of signal flow between objects. Diagrams are much easier to understand than textual code, especially when the code is spread across multiple pages. Here is a graphical example of a fine-grained parallel component:

![Diagram of a fine-grained parallel component](image)

“While Loop”

Computer geeks often write to argue that it is easier and faster to write keywords like ‘while’, ‘+’, ‘-‘, ‘=’, etc… than it is to click and drag an icon. To that I say, phooey! The real beauty of event-driven reactive programming is that it makes it easy to create and use plug-compatible components. Once you’ve build a comprehensive collection of low-level components, then there is no longer a need to create new ones. Programming will quickly become entirely high-level and all programs will be built entirely from existing components. Just drag’m and drop’m. This is the reason that I have been saying that Jeff Han’s multi-touch screen interface technology will play a major role in the future of parallel programming. Programming for the masses!

**See Also:**
- Parallel Programming, Math and the Curse of the Algorithm
- The Age of Crappy Concurrency: Erlang, Tilera, Intel, AMD, IBM, Freescale, etc.
- Parallel Computing: The End of the Turing Madness

**The COSA Saga**

*Rockwell Aim 65*
I had my initial idea for COSA back in 1980. I had just bought myself a 1 MHz Rockwell AIM 65 single board computer with 4k of RAM. At the time, I knew almost nothing about computers other than that they fascinated me to no end. On the same day my computer arrived, I dove into one of the assembly language programming books that came in the package. In the introductory chapter, the author (I can’t remember his name) covered the basics of how computers work and what a program consists of. He then explained that, unlike a biological brain, which has many slow processors (neurons) working in parallel, a computer has a single CPU that processes instructions one after the other. However, he continued, a CPU such as the 6502 is so fast that it can do the work of many slow parallel processors in a short interval.

Right away, even before I knew enough assembly language to write my first program, it was clear to me that computers were fundamentally flawed. I don’t know how but I immediately understood that a computer program should be more like a neural network. I knew that processors should be designed and optimized to emulate the reactive parallelism of the brain at the instruction level. I also realized that, for performance purposes, the emulation mechanism had to reside within the processor hardware itself.

**Good Ideas Don’t Get Stolen**

Soon afterward, I mastered assembly language and landed a job as a game programmer at a small company. I tried to explain my idea to other programmers but they either did not care or thought it was a dumb idea. I was stunned by their reaction. I just could not understand how seemingly intelligent people could be so blind to something that was so obvious. Surely, game programmers understood the concept of simulating parallelism with the help of two buffers and a loop. My idea essentially took the concept down to the lowest possible level. It is not rocket science. Someone even told me that, if my idea was any good, I should keep it a secret. I was then
reminded of the following quote by computer pioneer Howard Aiken: “Don't worry about people stealing your ideas. If your ideas are any good, you'll have to ram them down people's throats.”

**No Reason to Change**

Over the ensuing decades, I developed a love-hate relationship with computers: I loved their potential but I hated the way they worked and the way they were programmed. I hated programming languages. I still do. I longed to do something about it but, even though COSA is dear to me, I could never find much time to devote to it. Like everyone else, I was busy making a living. Besides, I was preoccupied with another passion of mine, artificial intelligence, which is the real reason that I became interested in computers in the first place. I watched in horror as computer scientists turned computer programming into a complete mess, a veritable tower of Babel. The Turing machine was accepted as the de facto computing model. Hopelessly flawed computer languages and operating systems began to proliferate like mad. Needless to say, unreliability and low productivity wreaked havoc everywhere. A few people in the business began to complain and the reliability industry sprouted like a mushroom. Years later, the problem is still with us, worse than ever. Realizing that the COSA model is deterministic and inherently reliable, I got the idea of using the software reliability crisis as a springboard to promote Project COSA. That worked to a certain extent. A handful of people started paying attention but that was not enough to motivate the leaders of the industry to change to a better model. After all, since they had a captive market, people were making money hand over fist and so there was no real reason to change. Besides, the computer science community (mostly a bunch of old computer geeks who don't know when it is time to retire) still has a firm grip on the business.

**Hooked on Speed**

Buggy software, high production costs and late product launches took their toll but the industry was resigned to live with the pain. Even if everybody became somehow convinced that a solution to their problems was available, there was already too much time and effort invested in legacy applications to turn the ship around. But more trouble was lurking around the corner. As the complexity of software applications increased, the demand for higher performance also went up. Moore’s law took care of that problem for decades. Eventually however, the laws of physics got the upper hand and it became impossible to increase speed while keeping the chips cool enough. Disaster loomed. The industry could see only one way out of its predicament: multicore parallel processing based on an old programming technique called multithreading. Problem is, programming with threads is hard as hell. The solution is obviously not working and, lately, there has been an odor of panic in the air.

**A New Opportunity**

The COSA software model is, of course, inherently parallel and has been from the start. In addition, it does not use threads and is thus free of all the nasty problems associated with multithreading. Furthermore, since COSA is signal-based just like hardware components, it is ideally suited to compositional software construction via plug-compatible modules. In other words, fast and easy parallel programming; precisely what the industry is searching for. In the
past, I never saw the need to emphasize the parallel nature of COSA. It did not occur to me until about a year ago that the only thing that would convince the computer industry to abandon its evil ways would be the performance issue. I saw it as an excellent opportunity to promote COSA in earnest. I made a lot of noise and a lot of people in the business are beginning to take notice. Still, I am a realist. The computer industry is like the Titanic and nobody can expect it to turn on a dime. There is a lot of inertia that must be overcome, not only in terms of engineering, but also in terms of leadership. And by this I am not talking about leaders in academia. I had long ago given up on the computer science community since I am convinced that computer scientists are the ones who got us all into this mess in the first place. They are not about to accept anything like COSA because COSA will make them look stupid. Only the captains of the industry have the power to decide its future course. I am talking about people like Bill Gates, Steve Ballmer, or Paul Otellini. Problem is, trying to get the attention of people like that is like trying to get an audience with the Pope. Oh well. I am doing the best I can and it’s not over until it’s over.

See Also:
How to Solve the Parallel Programming Crisis

COSA: A New Kind of Programming, Part I

Abstract

A few exciting ideas related to the on-going evolution of the COSA programming model have been percolating in my mind for quite some time. These ideas form the basis of a radically new way of looking at software construction that is so intuitive, it promises (or threatens, as the case may be) to reclassify computer programming as a mostly geek-only occupation into something that the average computer user can partake in and enjoy. This multi-part article is an account of the reasoning that led to my current thinking.

Something Is Missing

There is no doubt that the graphical approach to parallel programming can do wonders to productivity and program comprehension. It is true that the use of plug-compatible components in COSA will facilitate drag-and-drop software composition but the simple and intuitive feel that one gets from connecting a sensor to an effector is absent in the realm of high-level components.

![Diagram of a While Loop](image-url)
Masters and Slaves

I realized that the only way to solve the problem is to return to COSA’s roots. The COSA philosophy is that a program is a behaving machine that senses and effects changes in its environment. Of course, there is more to the design of a behaving machine than realizing that programs behave. We, as designers, want the program to behave in a certain way. That is to say, we want control. At the lowest level, we can control the behavior of a program by connecting specific sensors to specific effectors. The applicable metaphor, in this example, is that the sensor is the master or controller and the effector is the slave, i.e., the object that is under control. I rather like the master/slave symbolism because it perfectly illustrates the principle of complementarity. Those of you who have followed my work over the years know that I have always maintained that complementarity is the most important of all the COSA principles because it is the basic underlying principle of every complex organism.

In Part II, I will describe how behavior control in COSA works at the elementary level.

See Also:
How to Solve the Parallel Programming Crisis

COSA: A New Kind of Programming, Part II

Part I, II, III, IV, V, VI

Abstract
In Part I of this multi-part article, I wrote that the master/slave approach to elementary behavior control in the COSA programming model should be extended to high-level software construction. My thesis is that this control mechanism is so simple and intuitive that it will revolutionize computer programming by moving it out of the exclusive realm of computer nerds into that of the average computer user. Since COSA is inherently parallel, this will, in effect, solve the parallel programming problem. Below, I go over the elementary principles of control used in COSA.

Effector Control

Most of us are familiar with the controls that come with many of our electrical and electronic appliances. Some may be a little bit fancier than others but almost all come equipped with a minimum set of controls: the on/off (start/stop) buttons. To reuse a previous metaphor, we are the masters and the appliances are the slaves. It turns out that motor command neurons in the brain’s basal ganglia use a similar method to control their target muscles: excitatory (start) and inhibitory (stop) signals. The way it works is that the neuron begins firing as soon as it receives an excitatory signal and stops firing when it receives an inhibitory signal. This is what gave me the original idea for COSA effectors. The addition effector shown below will repeat its operation over and over until it receives a stop command.

A single motor command neuron may receive excitatory and inhibitory signals from hundreds or even thousands of afferent synaptic connections. It goes without saying that the basal ganglia are using some kind of error detection mechanism in order to keep all those incoming control signals from conflicting with one another. COSA effectors, too, use a special mechanism that automatically detects command conflicts. It is applied during application development for debugging purposes and it is based on what I call the principle of motor coordination:

No action can be started if it has already started, or stopped if it is already stopped.

In sum, low-level behavior control in COSA is a very simple thing that even children can grasp. In Part III, I will explain how to control the behavior of high-level COSA components by applying the same principles used with elementary objects.

See Also:
How to Solve the Parallel Programming Crisis
Parallel Computing: Why the Future Is Compositional
COSA: A New Kind of Programming, Part III

Part I, II, III, IV, V, VI

Abstract

In Part II, I showed that behavior control in the COSA programming model is based on a simple master/slave mechanism that uses complementary start/stop control commands. Below, I reveal how the same method can be used to control high-level components as well.

Component As Effector

Controlling a component simply means that the component can be seen, for all intents and purposes, as a low-level COSA effector. Every component will have a special control/effector cell connected to a multiconnector with three connections: two input connections for the start and stop signals and one output connection for the ‘done’ signal that is emitted when the component is finished with its task. The control effector can be used both internally and externally.

Connectors are great because they enforce plug-compatibility and make drag-and-drop software composition possible, but they don’t do much for program comprehension. The reason is that part of a connector’s purpose is to hide information so as not to overwhelm the user with too much complexity. My philosophy is that information should be delivered on an as-needed basis only. That being said, it would be nice if we could summons a special view of a group of a component in order to see exactly how they interact together.

An Example

Let’s say we have a water level component that is used to control a pump that maintains water in a tank at a certain level. Suppose we don’t want the pump to be turned on too often for maintenance or costs reasons. To do that, we can use a generic real-time timer cell to wait, say, 30 minutes between activations. We could incorporate the timer cell directly into the water level component but the latter would no longer be a generic component. A better alternative is to create a separate supervisor component that uses the timer cell to control the activation of the
water level component. The figure below shows the two components, as they would normally appear.

In a complex program, the supervisor component would normally be extended to control an indefinite number of slave components. Note that, while the figure shows the components involved, it does not tell us how the water level component is controlled. For that, we need a control view. As seen below, the control view is a simple diagram that depicts the manner in which the water level component is controlled by the supervisor. It displays only the control connections; all other connections, if any, are omitted.

Essentially, the water level component emits a signal when it's done. This signal is used to start the delay timer. At the end of the delay, the timer emits a Done signal, which is used to start the water level component and the cycle begins anew. In Part IV, I will describe the overall control architecture of a COSA component and explain what happens internally to a component when it receives a start or a stop signal.

See Also:
How to Solve the Parallel Programming Crisis
Parallel Computing: Why the Future Is Compositional

COSA: A New Kind of Programming, Part IV

Part I, II, III, IV, V, VI

Abstract
In Part III, I introduced the concept of the control effector, a high-level behavior control mechanism that makes it possible to control a COSA component as if it were a low-level effector. What follows is a description of the overall control architecture used in the COSA programming model and I explain what happens internally to a component under control.

**Control Architecture**

None of this is chiseled in stone but the way I envisioned it, every high-level component should contain a single master supervisor in charge of one or more slave components. The only components that do not contain a supervisor are low-level components, i.e., components that are composed of cells. It goes without saying that a supervisor is a low-level component. (See Two Types of Components). In the figure below, a component is shown with its supervisor and five slave components. Note that only the control connections are shown for clarity. Normally, the slave components will make data connections with one another and some of the connectors may be visible outside the component. Every one of the slave components may have internal supervisors of their own, if necessary.

![Control Architecture Diagram](image)

Only a supervisor can access the control connector (small red circle) of a slave. The latter cannot access the control connector of its supervisor or that of another slave. The control connector of a supervisor component can only be accessed by an external supervisor component. When a supervisor receives a start or stop signal, it may pass it to the slaves concurrently or in a given sequential order dictated by the design. In an actual development environment, the order in which the components are activated can be shown in slow motion by visually highlighting them in some fashion.

**Control Effector**
A control effector is a special COSA cell that is used to activate and/or deactivate a low-level component. In a previous illustration (reproduced below) I showed a control effector connected to its 3-input multiconnector. That is the default configuration. This is not a requirement, however.

It is up to the designer to decide what to do with the start and stop signals. For example, the component may need to initialize or reset certain variables after starting and before stopping. Or it may do nothing other than outputting a ‘Done’ signal when it receives a ‘Stop’ signal (by routing the ‘Stop’ signal to the ‘Done’ terminal). It also has the option of stopping itself for whatever reason by sending a 'Stop' signal to its control effector.

**Deactivation**

Stopping a component means to deactivate it so that it can no longer process signals. Deactivating a component is a simple matter of clearing a special activation flag in every cell of the component. This causes the processor to ignore them. A component is fully deactivated only if its control effector receives a 'Stop' signal.

**Activation**

Activating or starting a component is a little more complicated than just resetting the activation flags. Recall that, unlike conventional programming models, COSA is a change-based or reactive model that uses reactive sensing. That is to say, in a COSA program, a comparison operation (i.e., sensor) is not explicitly executed by the program but is invoked automatically whenever there is a change in a data variable that may potentially affect the comparison (See Effector-Sensor Associations). Being a change detector, a sensor must compare the current state of its assigned variable with its previous state. It fires only when there is a correct change. For example, a non-zero sensor fires only if its variable changes from zero to non-zero. The problem is that, when a component is activated, its sensors have no idea what the previous states were. The solution is to set all sensors to their default states upon activation and invoke them immediately afterward. This way, when a component is activated, all of its sensors perform their assigned tests or comparisons on their assigned data and fire if necessary. A component is fully activated when its control effector receives a 'Start' signal.

In Part V, I will describe reactive data connectors and explain why they are preferable to active message passing.

**See Also:**
Abstract

In Part IV, I described the overall control architecture used in the COSA programming model and what happens to a component when it receives a ‘Start’ or ‘Stop’ signal. In this installment, I explain why simple reactive data connectors are better than message connectors.

The Problem with Messages

Whenever two or more components cooperate on performing a common task, they must not only be attentive to changes in their own immediate environment, but also, to what the other components are doing. When I first devised the COSA model, I had assumed that sending messages was the best way for components to communicate with one another. I have since changed my mind.

The problem with sending messages is that the message sender has no way of knowing when the message should be sent and, as a result, the receiver has no choice but to query the sender for info. This complicates things because it requires the use of a two-way communication mechanism involving two message connectors or a two-line multiconnector. Alternatively, the sender is forced to send messages all the time whether or not the receiver needs it. I now think that messaging should be used strictly in conjunction with a messenger component dedicated to that purpose. I am also slowly coming to the conclusion that messaging should be used only in a distributed environment between components that reside on separate machines.

Reactive Data Sensing

The communication method that I prefer is reactive data sensing in which components simply watch one another’s actions within a shared data environment. This way, sharers can immediately sense any change in relevant data and react to it if desired. Since it is up to the receiving components to decide whether or not a change is relevant to them, it makes future extensions easier to implement. Sure, you can have restrictions such as which components have permission to effect changes but the component that owns the data should not impose any restriction on when or whether those changes are detected and used by others. Reactive data sensing is a simple matter of creating sensors (comparison operators) and associating them to relevant effectors. Effector-sensor association is done automatically in COSA.
In the figure above, the dotted line means that the + effector is associated with the != sensor. The way it works is that the != comparison operation is performed automatically every time the effector executes its operation. If the assigned change occurs, the sensor emits a signal.

**Reactive Data Sensing**

Reactive sensing makes sense within the context of behavior control. The primary reasons for stopping or deactivating a component are that a) its services are either not needed at certain times (deactivation saves cycles) or b) they would conflict with the work of another component (this prevents errors). The main job of a COSA Supervisor is to precisely time the activation and deactivation of various components under its charge so as to ensure smooth cooperation while eliminating conflicts and improving system performance.

**Data Connectors**

A data connector is just a mechanism for sharing data. The data must reside in only one component, the owner or server. The latter has read/write permission on its own data. A client component that is attached to a data owner via a data connector has only read permission by default. It is up to the component designer to specify whether client components can have write permission as well. The figure below illustrates the use of both data and signal connectors. To the right is the color convention that I currently use for the connectors. Note that control connectors are signal connectors.
The data assigned to a data connector can be a single variable, a list of variables, a C++ like data structure or an array. In a design environment, double-clicking on a data connector will open up a box showing the type of data that is assigned to it and the effector and sensors assigned to each data item, if any.

In Part VI, I will introduce a new COSA high-level behavior control mechanism with two complementary commands, ‘Pause’ and ‘Continue’.

See Also:
- How to Solve the Parallel Programming Crisis
- Parallel Computing: Why the Future Is Compositional

COSA: A New Kind of Programming, Part VI

Part I, II, III, IV, V, VI

Abstract

In Part V, I came out against the use of messaging for common communication between components and showed my preference for the flexibility of reactive data sensing and data connectors. Here, I describe a new approach to task prioritization based on a simple extension to the COSA behavior control mechanism.
Thread-Based Prioritization

One of the few advantages of using multithreading is that it makes it possible to prioritize threads. Even though this capability will not matter much with the advent of processors sporting hundreds or thousands of cores, the fact remains that, whenever processing power is at a premium, it makes sense to allocate more of the processor’s cycles to critical functions that must execute in real time. It goes without saying that thread-based priority scheduling wreaks havoc with deterministic timing, which is one of the reasons that the COSA software model does not use threads.

Message-Based Prioritization

Since COSA is not multithreaded, my initial approach was to use a message-based scheme for task prioritization. Lately, however, I have been having serious doubts about the suitability of messaging for inter-component communication. The approach that I originally had in mind would use a prioritized message queue within a client-server context. High priority messages would simply go to the head of the queue. It then occurred to me that a queued client-server approach makes no sense in an inherently parallel environment. Indeed, why have a serial server processing queued requests one at a time when using multiple parallel server clones could scale in performance as the number of processor cores is increased? Not a good idea.

Behavior-Based Prioritization

I have already explained how basic behavior control is done in COSA. While the Stop command can be used at times to save cycles, this is not its main intended function. Its main function is to prevent conflicts among cooperating components. Besides, an improperly timed Stop command will probably result in failure because intermediate results are discarded. I figure that the best way is to introduce a new control mechanism that can temporarily pause a component. I call it the Pause Effector. Its purpose is to give the system the ability to alleviate the processor’s load so that the more time-critical tasks can be processed in real time. The caveat is that any signal received by the component during its comatose state will be ignored.

The way it works is as follows. When a signal arrives at the ‘Pause’ terminal, the component goes into a coma and all cell activities in the component are suspended. Internally, the system sets a ‘stop’ flag in all the cells that causes the processor to stop processing them. All cells that were about to be processed in the next cycle are placed in a temporary hold buffer. On reception of a ‘Continue’ signal, the system clears the ‘stop’ flag in all the cells and the cells that were in
the temporary hold buffer are transferred into the processor’s input buffer for processing. A signal is emitted to indicate that the component is once again active.

**Processor Load Manager**

At this point, I believe that the Pause effector should not be directly accessible by user applications. Every COSA operating system will have a Load Manager whose job it is to manage processor load according to every component’s load requirement. My current thinking is that only the Load Manager should control the Pause Effector but this may change if a good enough reason is brought to my attention. Again, I will say that I don’t think that task prioritization will be needed in the future with the advent of processors with hundreds and even thousands of cores.

**See Also:**
- How to Solve the Parallel Programming Crisis
- Parallel Computing: Why the Future Is Compositional

**Why Timing Is the Most Important Thing in Computer Programming**

**An Analogy**

Architects, carpenters, mechanical and civil engineers expect things to have specific sizes. If, during construction, the sizes of parts are found to be different than specified or do not match properly with other parts, a search will be conducted to find the cause of the discrepancy and correct it. In this article, I will argue that size (distance) is to architecture what timing is to computing. In other words, deterministic timing is essential to software reliability.

**Deterministic Timing in Reactive Concurrent Systems**

In a Von Neumann computer, it is unrealistic to expect every operation of a program to occur at a specific relative time based on a real time clock. The reason is that operations must wait their turn to be processed. The problem is twofold. First, the processor load varies from moment to moment and second, algorithmic software is such that it is impossible to predict the duration (in numbers of system cycles) of every subroutine in an average program. However, it is possible to simulate a parallel, signal-based, reactive system based on a virtual system clock. In such a system, every operation is required to be purely reactive, that is to say, it must execute within one system cycle immediately upon receiving its signal. These two requirements (every elementary operation is reactive and is processed in one cycle) are sufficient to enforce deterministic timing in a program, based on the virtual system clock. Deterministic timing means that reaction times are predictable. It does not mean that all the events (such as the movements of
a mouse) that trigger the reactions are predictable. However, one event may trigger one or more chains of reactions and these, too, are deterministic, relative to the first event.

**Timing Watchdogs**

One nice thing about concurrent reactive systems is that interval detectors can be used to automatically find invariant intervals between any number of signals within a program. We can place timing watchdogs at various places in the program (this, too, can be done automatically) so that any discrepancy between an expected interval and the actual measured value will trigger an alarm. The temporal signature of a reactive system remains fixed for the life of the system and this makes for rock-solid reliability. So there are only two ways a timing watchdog can trigger an alarm; either the code was modified or there was a local physical system failure.

**Automatic Discovery and Resolution of Data and Event Dependencies**

Another nice aspect of concurrent reactive systems is that they are based on change. A change to a program variable is immediately communicated to every part of the program that may be affected by the change. The development environment can automatically link every entity or operator that changes a variable to every sensor that detects the change. This essentially eliminates blind code.

**Side Effects in Complex Systems**

We all know how hard it is to maintain complex legacy systems. A minor modification often triggers unforeseen side effects that may lay dormant for a long time after release. The right combination of events may cause a system failure that can be directly linked to the modification. For this reason, most system managers will look for alternative ways around a problem before committing to modifying the code. The side effects problem not only places an upper limit to the complexity of software systems, but the cost of continued development and maintenance soon becomes prohibitive. This is a problem that will never go away as long as we continue to use algorithmic systems. Luckily, the problem becomes nonexistent in the temporally deterministic reactive system that I described above. This is because blind code elimination and the use of timing watchdogs make it impossible to introduce undetected side effects. Indeed, timing is so deterministic and precise in a purely reactive system that the smallest modification is bound to violate a temporal expectation and trigger an alarm. It is up to the designer to either accept the new temporal signature, change it or revert back to the old code. As a result, we can build our software as complex as possible without having to worry about hidden bugs. In fact, and this is rather counterintuitive, more complex software will mean more timing constraints and thus more correct and robust systems, i.e., systems that work according to specs.

**The Entire Computer Industry Is Wrong**

All right. I know this sounds arrogant but it is true. We have been building and programming computers the wrong way from the beginning (since the days of Charles Babbage and Lady Ada Lovelace). To solve the unreliability and low productivity problems that have been plaguing the industry, we must change to a new way of doing things. We cannot continue with the same old
antiquated stuff. It is no good. We must abandon the algorithmic software model and adopt a concurrent reactive model. And what better time is there to change than now, seeing that the industry is just beginning to transition from sequential computing to massive parallelism? Reactive concurrent systems are right at home in a parallel, multicore universe. We must change now or continue to suffer the consequences of increasingly unreliable and hard to develop software.

See Also:
How to Solve the Parallel Programming Crisis

The Age of Crappy Concurrency: Erlang, Tilera, Intel, AMD, IBM, Freescale, etc…

I’ll get right to the point. If your multicore CPU or concurrent programming language or operating system does not support fine-grain, instruction-level parallelism in a MIMD (multiple instruction, multiple data) environment, it is crap. OK, I know that you are offended and you don’t think that your company’s CPU, language, or OS is crap but it is. And I mean Crap with a capital C. The computer market is looking for fine-grain parallel systems that are fast, secure, easy to program, auto-scalable and bug free. The crap that I see out there does not even come close to delivering what the market wants. Let me twist the knife in the wound a little deeper because I don’t think you people (I’m tempted to say, you idiots :-), but I’ll let it slide) are getting the message. Here is a list of highly desirable things that the market wants right now, things that your crappy products are not supporting but should be.

- **Fast, fine-grain, instruction-level parallelism using MIMD.** What is the point of parallelism otherwise? And please, don't tell your customers that it can only be done using SIMD. As I wrote elsewhere on this blog, using SIMD to develop software is like pulling teeth with a crowbar. You don't know how to do it with MIMD because you are a bunch of thread monkeys, that's all.
- **Easy software composition.** This means a graphical interface for non-algorithmic programming, among other things. It also means plug-compatible components. Just drag ’m and drop ’m. No more (text-based) computer languages, por favor! Computer languages are a primitive, awkward, unnecessarily complex and messy legacy from the 20th century. They are a stupid way to program computers. Only computer geeks love computer languages. The market is not about what geeks like. It's about profit, reliability, security and productivity.
- **Deterministic timing of events** at the elementary operation level. Deterministic temporal order (parallel or concurrent) is a must for reliability and security. This is possible only by adopting a convention whereby all instructions (elementary operations) are parallel reactive processes and have equal durations based on a virtual system-wide clock. That is to say, they all execute in exactly one virtual cycle. This is called ‘synchronous reactive computing’ (not to be confused with synchronous messaging).
• Implicit parallelism at the design level, and by this, I don’t mean compilers that extract parallelism from crappy sequential programs. I mean the programming environment should use objects that are inherently parallel. Otherwise, you're just fooling yourself that you're doing parallel programming. Implicit parallelism is the natural way to program and that's the way it should have been from the start, even with single core CPUs.
• Explicit sequential order. Either you’re doing parallel programming or you’re not. If you are, then sequences should not be implicit but explicitly specified by the developer. Otherwise, things become hard to organize and understand.
• Automatic resolution of data dependencies. This eliminates blind code and otherwise hidden side effects of code modification. It is an essential ingredient for reliability. It can only be done using a reactive synchronous software model.
• Fast asynchronous message passing using shared memory structures. Copying an entire message to a queue a la Erlang is a joke.
• Automatic transparent scalability. The developer should never have to think about cores. Programs should take advantage of all the cores automatically.
• Impregnable security. This is possible only in a system that enforces deterministic timing.

I could go on but that’s enough for now. What is my point? My point is that you people in the computer industry who insist on putting out one crappy product after another, have no excuse. And please, don’t say anything about customers wanting to preserve their legacy tools and software. It’s bullshit. If you gave your customers the things that I list above, they would switch to it as fast as they can. Why? Because it’s not crap, that’s why. It would save them a shitload of money and eliminate the headaches. It would allow them to develop systems of arbitrary complexity without having to worry about unreliability and insecurity. We would finally be able to implement truly automated transportation systems (no, you don’t need AI for this) and eliminate the need for human drivers on the road, thereby saving 40,000 lives a year in the US alone! And don’t tell me that it cannot be done because I know otherwise.

The public wants cars, trains and buses that drive themselves and planes that fly themselves. They want automated air traffic control that never fails. Your crappy products are killing people out there, by default and otherwise. You have no excuses! If you don’t get off your sorry asses :-) right now and investigate the soundness of the COSA software model, you should all be prosecuted for gross negligence. I mean it.

PS. There is one cool thing that happened in the industry recently and that's Jeff Han's multi-touch screen technology. Everybody in this business should take a good look at Jeff’s stuff and study that user interface closely because fast, touch 'n drag composition is part of the future of parallel programming.

See Also:
Erlang Is Not the Solution
Nightmare on Core Street
Parallel Programming, Math and the Curse of the Algorithm
Half a Century of Crappy Computing
Impregnable Security in Parallel Reactive Systems

The Security Problem

Government agencies, public and private corporations, and organizations of all sorts have legitimate reasons to protect their private information from prying malevolent eyes. This is a serious problem. A week does not go by without some report in the media about a new virus, worm, ID theft, break-in or other nefarious act perpetrated against computer users. In my article on the importance of timing in programming, I wrote that the temporal signature of a concurrent reactive system is guaranteed to remain fixed during the lifetime of the system. The reason is that the timing or temporal order of reactions in such a system is deterministic. This characteristic of concurrent reactive system can be used to great advantage in the battle against unreliability and malware.

Temporal Signature

How can reactive concurrent systems be used to secure a system against viruses, key loggers, spyware, intruders, worms and other malicious programs? Well, deterministic timing means that a program or operating system based on the COSA software model has a certain fixed temporal signature. A number of timing watchdogs can be inserted in a COSA system that will trigger an alarm as soon as the slightest change in the system’s temporal signature is detected. It is impossible for someone to surreptitiously wedge a spying program into a COSA system without changing its temporal signature and triggering an alarm.

See Also:
How to Solve the Parallel Programming Crisis

Erlang Is Not the Solution

Abstract

What follows is a comment that I posted at Intel Research Blogs in response to a recent article by Anwar Ghuloum titled Unwelcome Advice. I was actually replying to another commenter, Falde, who is promoting Erlang for parallel programming. Since Erlang is back in the news lately (see Slashdot article), I felt it would be appropriate to let my readers know what I feel about the suitability of Erlang as a solution to the parallel programming problem.

Erlang Is Not the Solution

The Erlang “solution” to the parallel programming problem is not the solution, otherwise we would not be here discussing the problem. The functional programming model has major
drawbacks, not the least of which is that most programmers are not familiar with it and have a hard time wrapping their minds around it.

Another problem is that the Erlang message-passing model forces the OS to copy entire arrays onto a message channel. This is absurd because performance takes a major hit. Shared memory messaging is the way to go. Certainly, using shared memory can be very problematic in a multithreaded environment but that is not because it is wrong. It is because multithreading is wrong. Threads are inherently non-deterministic and, as a result, hard to manage. Memory (data) is the environment that parallel agents use to accomplish a task. Agents sense and react to changes in their environment: reactive parallelism is the way it should be. Memory should be shared for the same reason that we humans share our environment. Switch to a non-algorithmic, deterministic model that does not use threads and all the problems with shared memory will disappear.

Furthermore, in spite of its proponents’ insistence on the use of words like “micro-threading” and "lightweight processes", Erlang enforces coarse-grain parallelism. In my opinion, if your OS, or programming language or multicore processor does not use fine-grain parallelism, it is crap. There are lots of situations that call for fine-grain processing. I have yet to see a fine-grained parallel quicksort implemented in Erlang.

Finally, a true parallel programming paradigm should be universal. There is no reason to have one type of processor for graphics and another for general purpose computing. If that’s what you are proposing, then it is obvious that your model is wrong and in serious need of being replaced. [Interestingly, Erlang proponents have nothing interesting to say on this issue.]

**Conclusion**

I will reiterate what I have said before. What is needed to solve the parallel programming crisis is a non-algorithmic, synchronous, reactive software model. By the way, I am not the only one ringing the non-algorithmic/reactive bell. Check out the work of Peter Wegner and Dina Goldin at Brown university. They’ve been ringing that bell for years but nobody is listening. You people are hard of hearing. And you are hard of hearing because you are all Turing Machine worshipers. It’s time for a Kuhnian revolution in computer science.

**See also:**
- Parallel Computing: Why the Future Is Non-Algorithmic
- Nightmare on Core Street
- How to Solve the Parallel Programming Crisis

**Parallel Programming, Math and the Curse of the Algorithm**

**The CPU, a Necessary Evil**
The universe can be seen as the ultimate parallel computer. I say ‘ultimate’ because, instead of having a single central processor that processes everything, every fundamental particle is its own little processor that operates on a small set of properties. The universe is a reactive computer as well because every action performed by a particle is a reaction to an action by another particle. Ideally, our own computers should work the same way. Every computer program should be a collection of small reactive processors that perform elementary actions (operations) on their assigned data in response to actions by other processors. In other words, an elementary program is a tiny behaving machine that can sense and effect changes in its environment. It consists of at least two actors (a sensor and an effector) and a changeable environment (data variable). In addition, the sensor must be able to communicate with the effector. I call this elementary parallel processor the Universal Behaving Machine (UBM).

More complex programs can have an indefinite number of UBMs and a sensor can send signals to more than one effector. Unfortunately, even though computer technology is moving in the general direction of our ideal parallel computer (one processor per elementary operator), we are not there yet. And we won’t be there for a while, I’m afraid. The reason is that computer memory can be accessed by only one processor at a time. Until someone finds a solution to this bottleneck, we have no choice but to use a monster known as the CPU, a necessary evil that can do the work of a huge number of small processors. We get away with it because the CPU is very fast. Keep in mind that understanding the true purpose of the CPU is the key to solving the parallel programming problem.

Multicore and The Need for Speed

Although the CPU is fast, it is never fast enough. The reason is that the number of operations we want it to execute in a given interval keeps growing all the time. This has been the main driving force behind CPU research. Over the last few decades, technological advances insured a steady stream of ever faster CPUs but the technology has gotten to a point where we can no longer make them work much faster. The solution, of course, is a no-brainer: just add more processors into the mix and let them share the load, and the more the better. Multicore processors have thus become all the rage. Unsurprisingly, we are witnessing an inexorable march toward our ideal computer in which every elementary operator in a program is its own processor. It’s exciting.

Mathematicians and the Birth of the Algorithmic Computer

Adding more CPU cores to a processor should have been a relatively painless evolution of computer technology but it turned out to be a real pain in the ass, programming wise. Why? To understand the problem, we must go back to the very beginning of the computer age, close to a
hundred and fifty years ago, when an Englishman named Charles Babbage designed the world’s first general purpose computer, the analytical engine. Babbage was a mathematician and like most mathematicians of his day, he longed for a time when he would be freed from the tedium of performing long calculation sequences. All he wanted was a reasonably fast calculator that could reliably execute mathematical sequences or algorithms. The idea of using a single fast central processor to emulate the behaviors of multiple small parallel processors was the furthest thing from his mind. Indeed, the very first program written for the analytical engine by Babbage’s friend and fellow mathematician, Lady Ada Lovelace, was a table of instructions meant to calculate the Bernoulli numbers, a sequence of rational numbers. Neither Babbage nor Lady Ada should be faulted for this but current modern computers are still based on Babbage’s sequential model. Is it any wonder that the computer industry is having such a hard time making the transition from sequential to parallel computing?

Square Peg vs. Round Hole

There is a big difference between our ideal parallel computer model in which every element is a parallel processor and the mathematicians’ model in which elements are steps in an algorithm to be executed sequentially. Even if we are forced to use a single fast CPU to emulate the parallel behavior of a huge number of parallel entities, the two models require different frames of mind. For example, in a true parallel programming model, parallelism is implicit but sequential order is explicit, that is to say, sequences must be explicitly specified by the programmer. In the algorithmic model, by contrast, sequential order is implicit and parallelism must be explicitly specified. But the difference is even more profound than this. Whereas an element in an algorithm can send a signal to only one other element (the successor in the sequence) at a time, an element in a parallel program can send a signal to as many successors as necessary. This is what is commonly referred to as fine-grain or instruction-level parallelism, which is highly desirable but impossible to obtain in an MIMD execution model using current multicore CPU technology.

The image above represents a small parallel program. A signal enters at the left and a ‘done’ signal is emitted at the right. We can observe various elementary parallel operators communicating with one another. Signals flow from the output of one element (small red circle) to the input of another (white or black circle). The splitting of signals into multiple parallel
streams has no analog in an algorithmic sequence or thread. Notice that parallelism is implicit but sequential order is explicit. But that’s not all. A true parallel system that uses signals to communicate must be synchronous, i.e., every operation must execute in exactly one system cycle. This insures that the system is temporally deterministic. Otherwise signal timing quickly gets out of step. Temporal determinism is icing on the parallel cake because it solves a whole slew of problems related to reliability and security.

It should be obvious that using Babbage’s and Lady Ada’s 150-year old computing model to program a parallel computer is like trying to fit a square peg into a round hole. One would think that, by now, the computer industry would have figured out that there is something fundamentally wrong with the way it builds and programs computers but, unfortunately, the mathematicians are at it again. The latest trend is to use functional languages like Erlang for thread-based parallel programming. Thread-based, coarse-grain parallelism is a joke, in my opinion. There is a way to design a fine-grain, self-balancing multicore CPU for an MIMD execution environment that does not use threads. Threaded programs are error-prone, hard to program and difficult to understand. Decidedly, the notion of a computer as a calculating machine will die hard. It is frustrating, to say the least. When are we going to learn?

Lifting the Curse of the Algorithm

To solve the parallel programming problem, we must lift the curse of the algorithm. We must abandon the old model and switch to a true parallel model. To do so, we must reinvent the computer. What I mean is that we must change, not only our software model, but our hardware model as well. Current CPUs were designed and optimized for the algorithmic model. We need a new processor architecture (both single core and multicore) that is designed from the ground up to emulate non-algorithmic, synchronous parallelism. It’s not rocket science. We already know how to emulate parallelism in our neural networks and our cellular automata. However, using current CPUs to do so at the instruction level would be too slow. The market wants super fast, fine-grain, self-balancing and auto-scalable multicore processors that use an MIMD execution model. It wants parallel software systems that are easy to program and do not fail. Right now there is nothing out there that fits the bill

The Next Computer Revolution

It remains to be seen who, among the various processor manufacturers, will be the first to see the light. Which nation will be the standard bearer of the new computing paradigm? When will the big switch happen? Who knows? But when it does, it will be the dawning of the next computer revolution, one which will make the first one pale in comparison. We will be able to build super fast computers and programs of arbitrary complexity that do not fail. It will be the true golden age of automation. I can’t wait.

See Also:
Nightmare on Core Street
Why Parallel Programming Is So Hard
The Age of Crappy Concurrency: Erlang, Tilera, Intel, AMD, IBM, Freescale, etc...
Half a Century of Crappy Computing
Tilera’s TILE64: The Good, the Bad and the Possible, Part I

Abstract

This is a three-part article in which I will examine what I think is good and bad about the TILE64 multicore processor and what I think Tilera can do in order to blow everybody out of the water. And by everybody, I mean Intel, AMD, Nvidia, MIPS, Sony, IBM, Sun Microsystems, Freescale Semiconductor, Texas Instruments, you name it. I mean everybody.

The Good

Tilera Corporation’s TILE64™ multicore processor is a marvel of engineering. It sports 64 general purpose, full-featured processor cores organized in an 8x8 grid linked together by Tilera’s super fast iMesh™ on-chip network. Each core has its own L1 and L2 caches with separate L1 partitions for instruction and data. The TILE64 cores can be individually switched into sleep mode to save energy when not in use.

One of the biggest problems in multicore processor design has to do with random access memory. Memory access has always been a problem because the processor needs data faster than the memory subsystem can deliver. The problem is even worse when there are multiple cores sharing memory because you get into bus contention problems. It is very likely that some future breakthrough in quantum tunneling or optical memory will eliminate the bottleneck but, in the meantime, the best that processor designers can do is to keep frequently accessed data in fast on-chip caches. This is all fine and dandy but a new problem arises when two or more caches contain overlapping areas of memory; if you modify the data in one, you must do so in the others. Maintaining cache coherence can quickly turn into a performance killing mess.
Tilera came up with an elegant way around this problem by arranging the cores in a grid and connecting them with a high-speed network or mesh. Tilera’s iMesh™ network makes it possible for a core to access the cache of an adjacent core or even that of a distant core, if necessary. This way, there is no problem of cache coherence. Apparently, the way it works is this; if a core needs a piece of data and the data is already in its own cache, then everything is hunky dory and there is no need to look elsewhere. If the data is not in the core’s local cache, the core uses the mesh to find it elsewhere. Obviously, in order to minimize latency as much as possible, it pays to optimize the system in such a way that cached data is as close as possible to the cores that are using it. I suspect that Tilera's approach is not a problem with scalability; that is to say, the performance hit is not exponential as you increase the number of cores. We can expect Tilera to come out with processors boasting hundreds of cores in the foreseeable future. In sum, Tilera’s TILE64™ is a beautiful thing. Next: Part II, The Bad

Tilera’s TILE64: The Good, the Bad and the Possible, Part II
Abstract

In the Part I, I wrote about what I think is good about Tilera Corporation’s TILE64™ multicore processor. Here, I will show where I think Tilera has gone wrong.

The Bad

The bad thing about the TILE64 multicore processor is that it is a pain in the ass to program. This is not something that is unique to the TILE64, however. All multicore processors currently on the market have the same problem. Right now, Tilera offers nothing new or revolutionary in the way of development tools that will make anybody stand up and take notice. C and C++ and multithreading are not going to cut it, I am sorry. I have said it before (and the industry is painfully aware of it by now), writing multithreaded parallel programs in C or C++ is like pulling teeth with a crowbar. Advertising that the TILE64 has a Linux SMP runtime environment is lame to the core. Linux is a freaking dinosaur, in my opinion, an ancient technology that somehow escaped from a computer museum of the last century. LOL. But then again, I think that all operating systems are dinosaurs.

I remember that when Tilera first announced the TILE64 about a year or so ago, the Erlang fanatics were jumping up and down, drooling at the prospect of using Tilera’s spanking new iMesh™ technology to send fat messages back and forth between their so-called lightweight processes. I remember thinking, my God, what a waste of technology. I am sure some of them contacted professor Anant Agarwal, CTO and co-founder of Tilera, and tried to convince him of the superiority of functional languages for parallel computing. I don’t know what happened but I am glad to see that Tilera does not mention Erlang or functional programming on its site. Erlang is a waste of time in my opinion. Its message-passing, copy-everything, anti-state approach to computing is not just counter-intuitive; it is a disaster as far as performance is concerned. The same goes for all functional programming models. As my readers already know, I don’t mince words when it comes to telling it like I see it.

In conclusion, I will say that Tilera, regardless of its engineering prowess, has committed the same unforgivable sin as the rest of the processor industry. Its leaders decided to design a processor before they could perfect a sensible programming model. It should be the other way around. Unless they take steps to correct that mistake, they will fail. Their only consolation is that everybody else is panicking and riding in the same sinking boat. Not much of a consolation if there are no lifeboats around, that’s for sure. Still, I don’t think that all is necessarily lost.

I will add that my main interest in parallel computing comes from my research in AI. I am convinced that artificial brains will be the most prevalent computer applications in the not too distant future. These smart programs will require massive parallelism in the form of super fast and efficient multicore processors. It does not really matter to me which organization or company unveils the correct solution to the parallel programming crisis and/or uses it to bring out a kick-ass product to market that blows everybody else out of the water. I just think that Tilera has a better chance than say, Intel, Sun, IBM or AMD. Those guys are way too married to last century’s technologies to notice the writings on the wall. Next: Part III, The Possible
Abstract

Previously in this series, I went over the good and the bad of Tilera Corporation’s TILE64™ multicore processor. I praised the TILE64’s engineering and its iMesh™ technology and I criticized its programming model and development tools. Below, I argue that Tilera can leapfrog over the rest of the industry by adopting a radically different parallel programming model and modifying the design of the TILE64 to support the model. Add a comprehensive suite of easy-to-use development tools and the rest of the multicore industry won’t know what hit it until it’s too late.

The Possible

Computer science professor, Kunle Olukotun, said recently, "If I were the computer industry, I would be panicked, because it's not obvious what the solution is going to look like and whether we will get there in time for these new machines" (source: CNN Money). The solution may not be obvious to Professor Olukotun and his team of thread-obsessed [pdf] researchers at Stanford’s Pervasive Parallelism Lab but it does not mean that there is no solution. I have argued that the solution has been staring us in the face for decades but the computer science community is blind to it. It is blind to it because of its infatuation with the Turing Machine, a soon-to-be obsolete computing model that is woefully inadequate to the task. I will not repeat my arguments here for the sake of brevity. Those of you who are interested can click on the links listed below for further information.

How to Solve the Parallel Programming Crisis
Parallel Computing: Why the Future Is Non-Algorithmic
Nightmare on Core Street
Why Parallel Programming Is So Hard
Half a Century of Crappy Computing
Parallel Computing: The End of the Turing Madness
The COSA Saga

Holy Grail

The holy grail of the computer industry is a multicore processor that has the following qualities:
• Easy to program using graphical tools for rapid application development. Drag and drop software composition. Programming for the masses.
• Implicit parallelism. Sequences must be explicitly specified.
• Automatic, transparent scalability. No need to rewrite the application when upgrading the processor. Just plug it in.
• Automatic load balancing. The programmer/designer should concentrate on the application and should never have to think about the processor.
• Fast, fine-grain (instruction level) parallelism using an MIMD processing model.
• Universality and homogeneity: one processor does it all. No need for a separate graphics processor.
• Deterministic timing for rock-solid applications that do not fail, regardless of complexity. The golden age of automation.
• Energy efficiency. Automated energy management.

Note that there is no mention of multithreading anywhere on the list. Any company that has the wisdom and the courage to design and build this processor, together with the required development tools, will have solved the parallel programming crisis and will dominate the computer industry for decades to come. There is only one caveat. This is a radical departure from the existing paradigm. Legacy applications will not run on this processor except maybe in simulation or interpreted mode. But who cares? The old machines can be slowly phased out and will eventually disappear altogether.

Tilera Can Do It

I believe that Tilera has a chance to become the next computer powerhouse by modifying its existing TILE64 processor and writing the necessary development tools on existing machines. In my next article, I will go over some of the specific changes that Tilera can make to its processor in order to turn it into a kick-ass product that will leave the competition in the dust.

See Also:
How to Solve the Parallel Programming Crisis
Transforming the TILE64 into a Kick-Ass Parallel Machine

Transforming the TILE64 into a Kick-Ass Parallel Machine, Part I

Part I, II, III, IV

Reading Material

Please read the following articles before continuing:

How to Solve the Parallel Programming Crisis
Tilera’s TILE64: The Good, the Bad and the Possible, Part I, II, III

Abstract

Tilera Corporation’s TILE64™ processor technology can serve as a springboard for a new type of multicore processor that can jettison Tilera to the forefront of the computer universe. Should it decide to act on this advice, the entire computer industry, the world over, would come to worship at its feet. It must act quickly, however, because many others have shown a marked interest in Project COSA and the parallel programming articles on this blog. In this multi-part article, I will describe several modifications that Tilera can make to the TILE64 that will turn it into a kick-ass multicore processor second to none.

Radical Core Transformation

Assuming that Tilera wants to be the industry leader and not just another me-too multicore vendor, the first thing it must do is to change the design of its processor core from a sequential or scalar core into a pure vector core. And by pure vector core, I don’t mean a GPU. I mean a pure MIMD vector processor (see section on vector processing below) in which every single instruction is a vector that can perform its operation in parallel! That’s right. You are not going to find this kind of processor core lying around anywhere. ARM doesn’t have one and neither do MIPS, Sun Microsystems, IBM, Nvidia, Intel or anybody else.

This is a major change and it is highly doubtful that Tilera could just modify the base MIPS core that it is currently using. The best thing to do is to redesign the core from scratch. Although this transformation is not absolutely necessary in order to support the COSA programming model, the performance increase would be so tremendous that it would be foolish not to do it. In my estimation, even a single-core, pure MIMD vector processor would see, on average, at least an order of magnitude increase in performance over a conventional scalar or sequential processor. Even a superscalar architecture would look like a snail in comparison. Imagine that! a one-core, general purpose, fine-grain, deterministic, parallel processor! This is the kind of benefits that can be obtained by adopting the COSA model. My thesis is that this is the way CPUs should have been designed from the beginning. (In a future article, I will examine the pros and cons of having multiple cores with a few vector units vs. having a single core with a huge number of vector units.)

So, if Tilera were to transform every core of its 64-core processor into a pure vector core, Intel’s much-ballyhooed 48-core Larrabee (with its ancient x86 technology and its 16 SIMD vector units per core) would look like a miserable slug in comparison. LOL.

MIMD Vector Processing

Why a pure MIMD vector processor and how is it even possible? The answer is that, in the COSA software model, instructions are not presented to the processor in a sequence but as an array of independent elements to be executed in parallel, if possible. This means that, ideally, a COSA core should be designed as an MIMD (multiple instructions, multiple data) vector processor as opposed to an SISD (single instruction, single data) scalar processor. This way,
every instruction is an elementary parallel vector with its own dedicated registers. Normally a vector processor operates in an SIMD (single instruction, multiple data) mode. A graphics processor (GPU) is an excellent example of an SIMD vector processor. The problem with the GPU, however, is that its performance takes a major hit when it is presented with multiple parallel instructions because it is forced to process them sequentially, which defeats the purpose of parallelism. Not so with an MIMD vector processor. Pure parallel bliss is what it is. In Part II, I will talk about vector optimization.

See Also:
How to Solve the Parallel Programming Crisis
Tilera’s TILE64: The Good, the Bad and the Possible, Part I, II, III

Transforming the TILE64 into a Kick-Ass Parallel Machine, Part II

Part I, II, III, IV

Abstract

In the Part I, I proposed that Tilera Corporation should abandon the sequential architecture for the cores used in its TILE64™ processor and adopt a pure MIMD vector architecture in which every instruction is a vector. The reason is that, in the COSA software model, instructions are not presented to the processor in a sequence but as an array of independent parallel elements. When you stop to think about it, what are sequential cores doing in a parallel processor in the first place? It does not make sense. In this chapter, I will explain how to optimize a pure MIMD vector processor for the best possible performance and energy saving.

Superscalar Architecture with a Twist

One CPU architecture that comes somewhat closer to what I mean by a pure MIMD vector processor is the superscalar architecture pioneered by the legendary Seymour Cray except that, in the COSA model, there is no need to check for dependencies between instructions because they are guaranteed to have none. In this context, note that Intel and Hewlett Packard have experimented with explicitly parallel instruction computing, a technique that uses a parallelizing compiler to pre-package instructions for parallel execution. This was the basis for the Itanium processor. I think that was a pretty cool idea because it is not only faster, it also simplifies the processor architecture. But why use an imperfect parallelizing compiler to uncover parallelism in sequential programs when COSA programs are inherently parallel to start with? Just a thought.

At any rate, the way I understand it (my readers will correct me if I’m wrong), parallel execution in a superscalar processor is achieved by placing multiple ALUs and/or FPUs on the same die. I feel that this is an unnecessary waste of both energy and silicon real estate because only one
functional unit (adder, multiplier, etc.) within any ALU or FPU can be active at any given time. This is true even if instruction pipelining is used.

**Vector Optimization**

What I am proposing is this; instead of using multiple ALUs and FPUs, it would be best to have a single ALU or FPU but use multiple independent functional units for every type of operation. In other words, you can have two 32-bit integer adders and two 32-bit integer multipliers within the same ALU. However, just doubling or tripling the functional units within an ALU would not serve any purpose that adding more ALUs could not serve. The reason for using multiple functional units is that some instructions are used more often than others. It would be best if the proportion of parallel functional units for a given instruction reflected the usage statistics for that instruction. For example, we might want to have more adders and multipliers than left or right shifters. The advantage of this approach is that the percentage of functional units that are active simultaneously will be higher on average. The effect is to increase overall performance while using much less energy and silicon than would be necessary otherwise. Based on this approach, I envision a processor core that can easily handle 16 or more independent instructions concurrently. In fact, the entire instruction set could, in theory, be processed concurrently since every instruction is an independent vector. (I could patent some of this stuff, I know.) In Part III, I will write about instruction caching and the COSA heartbeat.

**See Also:**
- How to Solve the Parallel Programming Crisis
- Tilera’s TILE64: The Good, the Bad and the Possible, Part I, II, III

**Transforming the TILE64 into a Kick-Ass Parallel Machine, Part III**

Part I, II, III, IV

**Abstract**

Previously in Part I and II of this series, I suggested that Tilera Corporation could blow the competition out of the water by adopting the COSA software model and dumping the processor core it is currently using for its TILE64™ multicore processor in favor of a pure MIMD vector core. The latter is somewhat similar to a superscalar processor except that every instruction is a vector and there is no need to test for instruction dependencies (the EPIC architecture of Intel's Itanium is probably a better comparison). The reason is that instructions in the core's input buffer are guaranteed to be independent in the COSA model. The new core should be capable of executing 16 or more different instructions simultaneously. This sort of hyper parallelism would transform the TILE64 into the fastest processor on the planet. Today, I will talk about how to optimize parallel instruction caching for improved performance. But there is more to computing
than performance. The COSA heartbeat is a global virtual clock that synchronizes all operations. Synchronization enforces deterministic processing, a must for rock-solid applications and security. Please read the previous chapters in the series and How to Solve the Parallel Programming Crisis before continuing.

Instruction Caching

The L1 instruction cache of every core would normally be divided into two parallel buffers A and B as seen below. L2 and data caches are not shown for clarity.

![Single Core Cell Processor Diagram](image)

While the instructions (COSA cells) in buffer A are being processed (hopefully, concurrently), buffer B is filled with the instructions that will be processed during the next cycle. As soon as all the instructions in buffer A are done, the buffers are swapped and the cycle begins anew. Of course, there are ways to optimize this process. Instead of just two buffers, the cache could conceivably be divided into three, four or more buffers and processed in a round robin fashion. An instruction prefetch mechanism can be used to fill the buffers ahead of time while the core is executing the current instructions. Even sensor-dependent branches (decision branches) can be fetched ahead of time. Branches that are not taken are simply discarded at the time of sensing (comparison). More detailed information on COSA sensors (comparison operators) can be found in the COSA Operating System article.

The COSA Heartbeat

The COSA software model is based on the idea that everything that happens in a computer should be synchronized to a global virtual clock and that every elementary operation lasts exactly one virtual cycle. This is extremely important because it is the only way to enforce deterministic processing, a must for reliability and security. It would make the TILE64 ideal for mission and safety-critical applications, especially in embedded systems. None of the current or projected multicore processors on the market supports deterministic processing.

Essentially, the heartbeat is a signal that tells a core to advance to the next parallel instruction buffer. This signal must be made available to all the running programs because it is used to update a global counter that is accessible by all programs. The counter is used by sequence detectors and other cells to calculate intervals.

In a two-buffer, single-core processor, the switch happens as soon as all the instructions in the current buffer are executed. In a multicore system, we need a mechanism that sends a heartbeat signal to all the cores on the chip so they can advance in unison. This mechanism can be as
simple as an AND gate that fires when all the current instruction buffers are done. It goes without saying that no core should have to wait long for a heartbeat after finishing its current buffer. This is why precise load balancing is important. In Part IV, I will go over automatic load balancing and data cache optimization.

See Also:
How to Solve the Parallel Programming Crisis
Tilera’s TILE64: The Good, the Bad and the Possible, Part I, II, III

Transforming the TILE64 into a Kick-Ass Parallel Machine, Part IV

Part I, II, III, IV

Abstract

Previously in this series, I suggested that Tilera should adopt the COSA software model and I argued that it should dump the sequential processor core it is currently using for its TILE64™ multicore processor in favor of a pure MIMD vector core. I argued that the number of functional units for every instruction should reflect the usage statistics for that instruction. This would increase performance while lowering energy consumption. I went over instruction cache optimization and the importance of the COSA heartbeat, a global synchronization mechanism that enforces deterministic processing, a must for software reliability and security. In today’s chapter (the last in this series), I talk about automatic load balancing and data cache optimization. As it turns out, Tilera’s iMesh™ technology is ideally suited for both tasks. Please read the previous chapters in the series before continuing.

Automatic Load Balancing

I have always maintained that an application developer should never have to worry about load balancing and scalability. Changing one’s processor to a more powerful model with more cores should automatically increase the performance of existing parallel applications without modification. An automatic self-balancing mechanism must be part of the processor’s hardware because there is a need to respond quickly to changing application loads. There is also a need for high precision in order to optimize bandwidth and minimize energy consumption.

Multiple Caches

Load balancing would not be so bad if all the cores could share a single instruction cache. Then it would be a matter of every core fetching and processing as many instructions as possible until the current instruction buffer is empty and then go on to the next buffer. The picture that I have in mind, as a metaphor, is that of several farm animals drinking water from a single trough at the
same time. Unfortunately, we can’t do it this way in the TILE64 because we would have 64 cores accessing the same cache, which would quickly run into a performance-killing bottleneck. It is better for every core to have its own cache. Besides, with the kind of pure vector core architecture that I am calling for, the core bandwidth would be at least an order of magnitude greater than say, that of an x86, a MIPS or an ARM core. In fact, it would be faster overall than existing GPU cores because it can handle both general purpose and graphics programs with equal ease and performance.

iMesh to the Rescue

The problem with multiple caches is that you want every input buffer to have more or less an equal number of instructions (see Part III for more on instruction buffers). If the buffers are completely independent, good luck on keeping them balanced. Fortunately for the TILE64, it comes with a fast on-chip network that connects all the cores together on an 8x8 grid. If we use the previous metaphor in which an instruction cache is seen as a water trough for farm animals, then we can use pipes linking the bottom of the troughs to represent the iMesh network.

Just as gravity and fluid dynamics would keep the water surface at the same height in every trough, our load-balancing mechanism should try to keep the number of instructions in the
caches about equal. I say ‘about’ because, unlike water with its huge numbers of super fine-grain molecules, processor instructions are comparatively coarse-grained.

**Fast and Tricky**

There are probably several solutions to the load-balancing problem. One that comes to mind calls for a central load balancer that is independent from the instruction fetch mechanism. The balancer would use the iMesh to keep track of how many instructions are in every cache and move them from one cache to another if necessary. The method for doing this might be a little tricky because we don’t want any instruction to stray too far from its core of origin, whose cache is where its data operands most likely reside.

Another solution is to place a mini-balancer in every core. It would have access only to the caches of its nearest neighbors and would use that information to determine whether or not to exchange instructions. This too could be tricky because there is the danger of entering into never-ending back and forth oscillations, which are undesirable, to say the least. Whichever method one decides to use, it must be fast because it must perform its work during instruction prefetch and must be done by the time the cores are ready to process the instructions.

**Conclusion**

The computer industry is delaying the inevitable: solving the parallel programming problem will require a radical paradigm shift in computing. The big players are all eying each other’s moves and are not willing to be the first to make the decisive move away from the inadequate computing models of the last century. It seems that the first steps that will trigger the revolution will have to come from an outsider, that is to say, a startup company. In my considered opinion, Tilera is rather well positioned, strategically, to make a winning move and take the computer world by storm. The writing is on the wall.

See Also:
- How to Solve the Parallel Programming Crisis
- Tilera’s TILE64: The Good, the Bad and the Possible, Part I, II, III
- Heralding the Impending Death of the CPU

**How to Design a Self-Balancing Multicore Processor, Part I**

Part I, II, III

**Reinventing the Computer**

Forget threads. Forget algorithms. Forget programming languages. That stuff is not only old, primitive, ugly and labor-intensive, it will soon be obsolete. The computer market of this century
wants super fast, fine-grain, parallel desktop supercomputers. The market wants rock-solid software that does not fail, software that automatically takes advantage of all the available cores of the processor it is running on. The new market also wants an order of magnitude improvement in productivity, at the very least.

There is only one way the market is going to get what it wants. We must abandon the flawed algorithmic/procedural computing model and adopt a non-algorithmic, implicitly parallel, reactive model. For that, we must rethink, not only our software model, but the computer itself. Yes, the computer must be reinvented! Current processors are optimized for the algorithm, the old paradigm. We must design new processors to support the new paradigm. And what better time is there to make the switch than now, now that the industry is taking its first painful steps away from sequential computing toward massive parallelism? This three-part article is aimed at anybody who is interested in the future of parallel computing.

The Real Purpose of the Processor

The most important question a processor designer must ask himself or herself is, what is the purpose of the processor? Most people in the business will immediately answer that the purpose of the processor is to execute sequences of coded instructions. Sorry, this is precisely what got us in the mess that we are in. In order to figure out the answer to this crucial question, we must first grok the true nature of computing. And the sooner we learn that a computer is really a collection of concurrently behaving and communicating entities, the sooner it will dawn on us that the processor is a necessary evil.

Necessary Evil

Why necessary evil? Because, ideally, each and every elementary entity in a computer should be its own self-processor, not unlike a neuron in a biological nervous system that sits around most of the time waiting for a signal to do something. In other words, ideally speaking, addition operators should add by themselves and multiplication operators should multiply by themselves. Unfortunately, we have not yet progressed to the point where we can build super-parallel systems like the brain with its tens of billions of tiny, interconnected, self-executing elementary processors. That will come one day, but not today.

Processor as Simulator

Fortunately, what we lack in numbers, we make up in speed. We can have a single fast processor core do the work of thousands or even millions of elementary processors (cells). Ideally, adding more processor cores into the mix should improve performance linearly. At least, that is the goal. This, then, is the true purpose of the processor, to simulate the behavior of a large number of inter-communicating cells. So, essentially, the processor is a cell simulator, or cell processor if you wish.

The Key to Proper Multicore Processor Design
The key to designing a multicore processor is to keep in mind that one is designing a cell simulator in hardware. Nothing more, nothing less. Come to think of it, regardless of the number of cores, this is the way it should have been designed in the first place. At any rate, simulating large numbers of small concurrent entities is neither new nor rocket science. Neural network and cellular automaton programmers have been doing it for years, in software. In our case, we are simulating two things:

**Behavior**
This is what the cells actually do, i.e., their operations (adding, subtracting, comparing, etc…).

**Signaling**
Most of the time, a cell will alert one or more other cells that they just performed an operation.

Essentially, a cell simulator consists of a cell processor, two buffers and a repeating loop. In Part II, I will explain how it works traditionally in software, and how it can be performed entirely by the processor. In addition, the processor can be easily designed to take advantage of all the available cores automatically and keep them busy. Scalable, automatic load balancing is what I’m talking about.

**See Also:**
- How to Solve the Parallel Programming Crisis
- Transforming the TILE64 into a Kick-Ass Parallel Machine
- Parallel Programming: Why the Future Is Non-Algorithmic
- Parallel Programming: Why the Future Is Synchronous

**How to Design a Self-Balancing Multicore Processor, Part II**

**Part I, II, III**

**Implicit Parallelism, Explicit Sequential Order**

In Part I, I wrote that a processor should be seen as a necessary evil. It is needed only because cells (elementary objects or instructions) in computer memory cannot process themselves. Thus a processor should be seen as a cell processor. From a programming point of view, the cells are the elementary operations that comprise a program residing in memory. In a truly parallel programming environment, parallelism is implicit and sequential order is explicit. In other words, unless specified otherwise, all cells are assumed to run concurrently with other cells. Every cell must explicitly specify its successor or successors, if any. A parallel program thus consists of multiple linked lists of cells (there are ways to use simple lists for optimization purposes but that’s another story). Immediately after executing its operation, a cell must send a
signal to its successor or successors, if any, alerting them that it is their turn to execute. In a reactive system like COSA, execution and signaling count as one system cycle, meaning that they happen simultaneously.

Two Instruction Buffers

In order to prevent the troublesome signal racing conditions that would otherwise occur, the cell processor must use two instruction or cell buffers, A and B. The way it works is rather straightforward. As the cells in buffer A are being processed, buffer B is populated with successor cells that will be processed during the next cycle. As soon as all the cells in buffer A are processed, the two buffers are swapped and the cycle begins anew. Simple. I use this exact method in my own neural network experiments. Of course, this method can be modified and optimized for performance in a processor. Instead of just two buffers one can have three, four or more buffers and an instruction prefetch mechanism can be used to fill the buffers ahead of time. This is important because load balancing must happen before the processor is ready to process a buffer.

The beauty of the two-buffer scheme is that the parallelism is fine grained and scalable and it can be performed entirely by the processor. It can be used in both single core and multicore architectures. It should be done using an MIMD configuration, if possible, an order to take full advantage of vectorization. Highly parallel programs will automatically scale to take advantage of all available cores. In fact, neither the program nor the programmer need be aware of the number of cores. The cores are presented with a list of cells to process at every system cycle and they divide the cell load among themselves. Program modules can send messages to each other via shared memory structures. Both buffers should reside on the chip for fast processing. The size of the buffers should be big enough to accommodate the most demanding situation so as to avoid using main memory. In a single-core processor, managing the buffers should be straightforward. In a multicore processor, however, we need a special controller, a task manager. In Part III, I will describe the task manager’s job, which is mostly load balancing.

See Also:
How to Solve the Parallel Programming Crisis
Transforming the TILE64 into a Kick-Ass Parallel Machine
Parallel Programming: Why the Future Is Non-Algorithmic
Parallel Programming: Why the Future Is Synchronous

How to Design a Self-Balancing Multicore Processor, Part III

Part I, II, III

Recap
In Part I and II of this three-part article, I argued that a processor should be seen as a device for simulating or processing vast numbers of concurrent, elementary processes or cells. A cell is a basic entity that performs an elementary operation. Unlike other concurrent languages, the programming model used for this processor design assumes that parallelism is implicit but sequential order must be explicitly specified. Simulating a parallel collection of cells requires the use of two processing buffers (A and B). Essentially, while the cells in buffer A are being processed, buffer B is being filled with the cells that will be processed during the next consecutive cycle. This is done in order to avoid signal racing conditions that would otherwise occur. As soon as all the cells in buffer A are processed, the two buffers are swapped (A becomes B and B becomes A) and the cycle begins anew.

Note: Both cells and data are shown sharing the same memory space. This is for convenience only. In a real processor, they would be separate and would use separate address and data buses.

The Task Manager

As already mentioned, both buffers should reside on the processor and buffer management can be handled by relatively simple on-chip circuitry. On a multicore processor, however, the load must be distributed among the cores for fine-grain parallel processing. Note that this is all MIMD processing. Using SIMD for development is like pulling teeth. Besides, SIMD is worthless when it comes to enforcing the deterministic timing of events. There are probably several ways to design a fast mechanism to manage load balancing among multiple cores. The one that I envision calls for every core to have its own pair of processing buffers, A and B. However, the job of populating the B buffers should be that of a special on-chip controller that I call the Task Manager (TM).
Essentially, the job of filling the buffers with instructions is that of the task manager. Whenever 
the TM encounters new cells to be processed, it should distribute them as equally as possible 
amongst the B buffers. The TM must have intimate knowledge of the status (the number of 
items) of the A and B buffers of every core. It should also know the execution duration (in terms 
of clock cycles) of every cell according to its type. This way, the TM can intelligently assign 
cells to each core and equalize their individual loads as much as possible. The TM has only one 
more job to do: as soon as all the A buffers are empty, it must signal the cores to swap the 
buffers and repeat the cycle. One nice thing about this approach is that the TM works in parallel 
with the cores and does not slow their performance. Another advantage has to do with power 
consumption. Since the TM has perfect knowledge of processor load at all times, it can 
automatically turn off a percentage of the cores, depending on the load, in order to save energy.

The Road Ahead

As far as I know, no other multicore architecture provides for fine-grain, self-balancing 
parallelism using an MIMD execution model. There is no doubt in my mind that it is the correct 
approach to designing the multicore architectures of the future. There are additional advantages 
that are inherent in the software model, such as fault tolerance, deterministic timing, the 
automatic discovery and enforcement of data dependencies. The result is rock-solid software 
reliability and high performance.

See Also:
How to Solve the Parallel Programming Crisis
Transforming the TILE64 into a Kick-Ass Parallel Machine
Parallel Programming: Why the Future Is Non-Algorithmic
Parallel Programming: Why the Future Is Synchronous

Parallel Computing: Both CPU and GPU Are Doomed

Tim Sweeny

A few weeks after I wrote the impending death of the CPU on my blog, Tim Sweeney, the 
renowned founder of Epics Games and pioneering game engine designer, predicts the impending 
fall of the GPU. In an interview titled Twilight of the GPU: an epic interview with Tim Sweeney, 
published by Ars Technica, the same day hurricane Ike ripped Texas a new one, Sweeny does the 
same to the future of graphics processors. Here is something that Sweeny said that caught my 
attention:

In the next console generation you could have consoles consist of a single non-
commodity chip. It could be a general processor, whether it evolved from a past 
CPU architecture or GPU architecture, and it could potentially run everything—
the graphics, the AI, sound, and all these systems in an entirely homogeneous
manner. That's a very interesting prospect, because it could dramatically simplify
the toolset and the processes for creating software.

This is exactly what I have been saying for a long time. Homogeneity and universality are the
names of the new game. I may not agree with Sweeny on what development tools we will use in
the future (he seems to be married to the old C, C++ linguistic approach), but he is absolutely
correct about the future of parallel processors.

**Nvidia**

This brings me to thinking about Nvidia. Unlike Intel and AMD, Nvidia’s financial future is not
tied to the CPU. The CPU will soon join the vacuum tube and the buggy whip in the heap of
obsolete technologies. The future of parallel computing is in vector processing and, as we all
know, Nvidia’s GPUs are vector processors. Sure, GPUs are not universal parallel processors
because they use an SIMD (single instruction, multiple data) configuration. However, this is a
problem that Nvidia will eventually correct by switching over to a pure MIMD (multiple
instruction, multiple data) vector architecture. In my opinion, Nvidia is ideally positioned to
dominate the processor industry in the decades to come. That is, assuming its leadership is
shrewd enough to see and heed the writings on the wall.

**See Also:**
*Radical Future of Computing, Part II*
*Heralding the Impending Death of the CPU*
*Transforming the TILE64 into a Kick-Ass Parallel Machine*
*How to Solve the Parallel Programming Crisis*

---

**Heralding the Impending Death of the CPU**

**Ancient Model vs. Modern Necessities**

The modern CPU may seem like a product of the space age but its roots are rather ancient.
British mathematician Charles Babbage first conceived of the principles behind the CPU more
than 150 years ago and he was building on ideas that were older still, such as Jacquard’s punch
card-driven loom and the algorithm, a mathematical concept that was invented a thousand years
earlier by Persian mathematician, al-Khwārizmī. Like most mathematicians of his day, Babbage
longed for a reliable machine that could automate the tedious task of calculating algorithms or
tables of instructions. Parallel computing was the furthest thing from his mind. Yet amazingly, a
century and a half later, the computer industry is still clinging to Babbage’s ancient computing
model in the age of multicore computers.

**The Good Way vs. the Bad Way**
There are two main competing approaches to parallelism. The thread-based approach, the one chosen by the computer industry for general purpose computing, calls for having multiple instruction sequences processed side by side. The other approach, vector-based parallelism, calls for having multiple parallel lists of instructions, with the lists processed one after the other.

In the figure above, the small circles represent instructions or operations to be executed by the processor. The down arrows show the direction of execution. In thread-based parallelism, each thread can potentially be processed by a separate sequential core (CPU). The problem is that the threads are not synchronized. In vector-based parallelism, the operations are fed to the processor as a collection (buffer or chunk) of elements to be processed in parallel. All the instructions in a chunk are processed in parallel and the time it takes to process a chunk is seen as a single process cycle. To properly implement chunk-based processing, one would need a pure MIMD (multiple instructions, multiple data) vector processor in which every instruction is an independent vector that can be processed in parallel with the others. Both approaches will increase performance but, as I have explained elsewhere, the vector-based approach is better because it is deterministic and fine-grained; and it reflects the way parallel objects normally behave in nature.

Our brains are temporal signal-processing machines. The ability to sense the temporal relationships between events is essential to our understanding of the world around us. We can make sense of the vector-based approach because we can easily determine which processes are concurrent and which are sequential, and we can make a clear distinction between predecessors and successors. This sort of predictability is the sine qua non of learning and understanding. This is part of the basis of the COSA computing model. Multithreaded applications, by contrast, are hard to understand and maintain precisely because they are temporally non-deterministic. The idea that we can solve the parallel programming crisis by holding on to a flawed and inadequate programming model is preposterous, to say the least.

**The World Does Not Need Another CPU**

I was happy to read the news that Nvidia has denied rumors that it was planning on developing
its own x86 compatible CPU in order to compete against Intel and AMD. Good for them. The last thing the world needs is another x86 CPU, or any CPU for that matter. Nvidia should stick to vector processors because that is where the future of computing is. However, it will have to do something in order to counteract the fierce competition from AMD’s ATI graphics products and Intel’s upcoming Larrabee. The most sensible thing for Nvidia to do, in my opinion, is to transform its base GPU from an SIMD (single-instruction, multiple data) vector core into a pure MIMD vector core. This would make it an ideal processor core for Tilera’s TILE64™ as I suggested in a previous article. Come to think of it, maybe Nvidia should just acquire Tilera. That would be a superb marriage of complementary technologies, in my opinion.

**Conclusion**

The CPU is on its deathbed. The doctors are trying their best to keep it alive but they can only postpone the inevitable for a little while longer. Soon it will die from old age but I, for one, will not be shedding any tears. Good riddance! It is not really hard to figure out what will replace it. It is not rocket science. It will take courage and fortitude more than brains. Who will be the standard bearer for the coming computer revolution? Which organization? Which company? Which country will see the writings on the wall and dominate computing for decades to come? Will it be Intel, AMD, Nvidia, or Tilera? Will it be the US, India, China, Germany, France, Spain, Sweden, Singapore, Japan or Taiwan? Who knows? Only time will tell but I sense that it won’t be long now.

**See Also:**
- Parallel Computing: Both CPU and GPU Are Doomed
- How to Solve the Parallel Programming Crisis
- Transforming the TILE64 into a Kick-Ass Parallel Machine
- Parallel Computing: Why the Future Is Non-Algorithmic
- Why Parallel Programming Is So Hard
- Parallel Computing, Math and the Curse of the Algorithm

**The Radical Future of Computing, Part I**

**Part I, II**

**Abstract**

A reader named Marc had an interesting comment in reply to my article, *Heralding the Impending Death of the CPU*. I hope Marc will forgive me for using his words as a vehicle on which to piggyback my message.

**Linguistic Origin of Programming**
"I think that algorithmic programming is popular because it is similar to the way many of us write in western natural language; people plan whether a thought should be after or before a previous one in academic essays, which is inherently sequential in nature."

I agree. I believe that the modern computer evolved from the sequential/algorithmic needs of mathematicians like Charles Babbage and Ada Lovelace. As you know, linguistic or textual symbols are perfect for expressing mathematical algorithms. I have often wondered what kind of computers we would have if clockmakers or locomotive engineers had had a more direct influence on the development of early computer technology. Those folks are more accustomed to having multiple interacting mechanisms performing their functions concurrently.

Note also that the typewriter predated modern computers and served as a model for the input device (keyboard) of the first mass market computers and has profoundly affected the way we perceive them. Although the mouse was a major influence in changing human-computer interactions, the event-driven approach to programming that it engendered somehow failed to convince computer scientists that every action in programming should be a reaction to some other action (event-driven), down to the instruction level. Hopefully, the new multi-touch screen technologies will drastically change our idea of what a computer is or should be.

**Petri Nets, Conditions, Metaphors and Simplicity**

"Native parallel programming requires that the programmer (or implementer if you'd rather call it that) decides what are the conditions that have to be met for each cell to trigger and what are the outputs that are produced based on those conditions so it requires skills that are part-user, part coder. Petri Nets are a great graphical symbolism for this. It actually requires that people focus on the problem instead of style."

I agree. Nobody should have to worry about syntax or have to learn the meaning of a token (in someone else’s native language and alphabet) in order to program a computer. Only a few graphical symbols (sensors, effectors, pathways, data and encapsulations) should be allowed. Labeling should be done in the designer’s preferred language. I believe that the main reason that graphical programming languages have not taken off is that their designers not only don’t seem to appreciate the importance of encapsulation (information hiding), but they have a tendency to multiply symbols beyond necessity. I am a fanatic when it comes to simplicity.

One of my goals is to turn computer programming into something that the lay public will find approachable and enjoyable. In this regard, I think that even Petri Nets, in spite of their simplicity compared to other programming models, are still too complicated and too abstract, making them unpalatable to the masses or the casual developer. I rather like PNs and I am sorry that the concept never really became mainstream. However, I have a bone to pick with the notion of conditions (places?). Don’t take me wrong; I don’t disagree that there is a need for conditions. I just don’t think the token concept is intuitive or concrete enough to appeal to the layperson. In my opinion everything should be driven by events (changes or transitions). What Petri calls a transition is what I call a sensor. A condition, to me, is just a past or current event and, as such, it
should be used in conjunction with sensors (logic sensors, sequence detectors, etc.). This makes it easy to extend the idea of conditions to include that of temporal expectations, a must for reliability in COSA.

That being said, the ideal programming metaphors, in my opinion, are those taken from the behavioral sciences such as psychology and neuroscience: stimulus/response, sensor/effect, sequence memory, action/reaction, environment (variables), etc… The reason is that a computer program is really a behaving machine that acts and reacts to changes in its environment. A layperson would have little trouble understanding these metaphors. Words like ‘transitions’, ‘tokens’ and ‘places’ don’t ring familiar bells. Let me add that, even though I applaud the clean graphical look of PNs, my main criticism is that they are not deterministic. In my view, this is an unpardonable sin. (I confess that I need to take another close look at PNs because it seems that they have evolved much over the years).

**New Leadership to Replace the Old**

"To me, starting with a software specification before implementing a solution seems obvious, but my son has mainly sold freelance projects to business types based on his suggested user interface first; when he tried to tell his potential customers what data sources he used and how he got to his results, the customers' eyes would often glaze over..."

Yes. People love to see pretty pictures, which is understandable because business types tend to see technology from a user’s perspective. They want to see the big picture and they don’t care about what makes it work. You make an interesting point because I have pretty much given up on selling my ideas directly to techies. I am slowly coming to the conclusion that the next computer revolution will have to emerge out of some government-funded initiative or some industry-wide consortium under the leadership of an independent, strategy-minded think tank. The reason is that the industry is in a deep malaise caused by the aging baby boomers who drove computer innovation in the last half of the 20th century, but lately have run out of ideas simply because they are old and set in their ways. I don't want to generalize too much but I think this is a major part of the problem. Their training has taught them to have a certain perspective on computing that is obviously not what is needed to solve the parallel programming and software reliability crises. Otherwise, they would have been solved decades ago. In fact, it is their perspective on computing that got the industry and the world into this mess in the first place.

As a case in point, consider this recent [article](HPCwire) by Michael Wolfe. It pretty much sums up what the pundits are thinking. Michael believes that “the ONLY reason to consider parallelism is for better performance.” I don’t know how old Michael is but it is obvious to me that his thinking is old and in serious need of an update. The problem is that the older computer nerds are still in charge at the various labs/universities around the world and they hold the purse strings that fund research and development. These folks have titles like CTO or Project Director or Chief Science Officer. That does not portend well for the future of computing.

As I wrote somewhere recently, the computer industry is in dire need of a seismic paradigm shift and there is only one way to do it. The old computer nerds must be forced into retirement and
new leadership must be brought in. The new mandate should be to reevaluate the computing paradigms and models of the last century and assess their continued adequacy to the pressing problems that the industry is currently facing, such as the parallel programming and software reliability crises. If they are found to be inadequate (no doubt about it from my perspective), then they should be replaced. These kinds of strategic decisions are not going to be made by the old techies but by the business leaders, both in the private sector and within the government. Sometimes, it pays not to be too married to the technology because you can’t see the forest for the trees.

**Software Should Be More Like Hardware and Vice Versa**

"There is plenty of parallel processing already going around in graphics processors, Field-programmable Gate Arrays and other Programmable Logic chips. It's just that people with software-experience who are used to a certain type of tool are afraid to make the effort to acquire what they see as hardware-type electrical engineer thought-habits; I know my programmer son would have an issue. The US has developed a dichotomy between electrical engineers and computer scientists."

Which is rather unfortunate, in my opinion. In principle, there should be no functional distinction between hardware and software, other than that software is flexible. I foresee a time when the distinction will be gone completely. The processor core as we know it will no longer exists. Instead, every operator will be a tiny, super-fast, parallel processor that can randomly access its data directly at any time without memory bus contention problems. We will have a kind of soft, super-parallel hardware that can instantly morph into any type of parallel computing program.

**Programming for the Masses**

"Talking heads" have a vested interest in promoting certain products that are only incremental improvements over the existing tools, because otherwise they would need to educate the clients about the details of the new paradigm, which would require extended marketing campaigns which would only pay back over the long term.

Yeah, legacy can be a big problem but it doesn’t have to be. But you bring out the important issue of client education, which is a major part of the paradigm shift that I am promoting. I think the time has come to move application design and development from the realm of computer geeks into that of the end user. The solution to the parallel programming problem gives us an unprecedented opportunity to transform computer programming from a tedious craft that only nerds can enjoy into something that almost anybody can play with, even children. Now that multi-touch screens are beginning to enter the consumer market, I envision people using trial-and-error methods together with their hands and fingers (and possibly spoken commands) to quickly manipulate parallel 3-D objects on the screen in order to create powerful and novel applications. I see this as the future of programming, kind of like putting Lego blocks together. In this regard, I don’t think we will need to reeducate traditional programmers to accept and use the new paradigm. They will have to get with the new paradigm or risk becoming obsolete.
The Radical Future of Computing, Part II

Part I, II

Abstract

Note: This is a continuation of my response to reader Marc’s comments in reply to my article, Heralding the Impending Death of the CPU.

The Market Wants Speed and Low Energy Consumption

The microprocessor market is also highly fragmented between cheap low-end processor makers like Microchip and Atmel, and desktop makers. The desktop players have their own mindset that has made them successful in the past. The obviously-easily parallelizable tasks (sound, graphics...) are so common that custom parallel processors were designed for them. You might be able to get Microchip to squeeze in 20 16f84 microcontrollers on one piece of silicon and could easily use a bunch of cheap PICs to emulate a bunch of 20 vector processors with current technology at a chip cost of maybe $100. But then, the optimum bus design would vary on the application. What application would be most compelling to investors? I don’t know... But I think an FPGA or multi-PIC proof of concept would help your idea become implemented at low cost, and a "suggestion software on how to parallelize applications" for sequentially-thinking programmers, combined with a parallel processor emulator for conventional chip architectures would help programmers see parallel programming as an approachable solution instead of a venture capitalist buzzword.

Well, I am not so sure that this would attract the people with the money. I sense that, when it comes to processors, people are more impressed with proven performance than anything else. And, nowadays, people also want low energy usage to go with the speed. Sure, it would be cool if I could demonstrate a powerful parallel programming tool, but it would be an expensive thing to develop and it would not prove the superiority of the target processor. What I would like to deliver, as an introduction, is a low wattage, general-purpose, single-core processor that is several times more powerful (measured in MIPS) than say, an Intel or AMD processor with four or more cores. I think I can do it using vector processing. This, too, is not something that can be built cheaply, in my estimation. It must be designed from scratch.

SIMD Vector Processor: Who Ordered That?
At this point in the game, there should be no doubt in anyone’s mind that vector processing is the way to go. As GPUs have already amply demonstrated, vector processing delivers both high performance and fine-grain deterministic parallelism. Nothing else can come close. That multicore vendors would want to use anything other than a vector core is an indication of the general malaise and wrongheadedness that have gripped the computer industry. As everyone knows, multithreading and vector processing are incompatible approaches to parallelism. For some unfathomable reason that will keep future psycho-historians busy, the computer intelligentsia cannot see past multithreading as a solution to general purpose parallel computing. That's too bad because, unless they change their perspective, they will fall by the wayside.

When I found out that Intel was slapping x86 cores laced together with SIMD vector units in their upcoming Larrabee GPU, I could not help cringing. What a waste of good silicon! The truth is that the only reason that current vector processors (GPUs) are not suitable for general-purpose parallel applications is that they use an SIMD (single instruction, multiple data) configuration. This is absurd to the extreme, in my opinion. Why SIMD? Who ordered that? Is it not obvious that what is needed is an MIMD (multiple instruction, multiple data) vector core? And it is not just because fine-grain MIMD would be ideal for general-purpose parallel applications, it would do wonders for graphics processing as well. Why? Because (correct me if I’m wrong) it happens that many times during processing, a bunch of SIMD vector units will sit idle because the program calls for only a few units (one instruction at a time) to be used on a single batch of data. The result is that the processor is underutilized. Wouldn't it be orders of magnitude better if other batches of data could be processed simultaneously using different instructions? Of course it would, if only because the parallel performance of a processor is directly dependent on the number of instructions that it can execute at any given time.

**Pure MIMD Vector Processing Is the Way to Go**

Most of my readers know that I absolutely abhor the multithreading approach to parallelism. I feel the same way about CPUs. A day will come soon when the CPU will be seen as the abomination that it always was (see Heralding the Impending Death of the CPU for more on this topic). However, SIMD vector processors are not the way to go either even if they have shown much higher performance than CPUs in limited domains. It is not just that they lack universality (an unforgivable sin, in my view) but the underutilization problem that is the bane of the SIMD model will only get worse when future vector processors are delivered with thousands or even millions of parallel vector units. The solution, obviously, is to design and build pure MIMD vector processors. As I explained in a previous article on Tilera’s TILE64, the best way to design an MIMD vector processor is to ensure that the proportion of vector units for every instruction reflects the overall usage statistics for that instruction. This would guarantee that a greater percentage of the units are used most of the time, which would, in turn, result in much lower power consumption and a greater utilization of the die’s real estate for a given parallel performance level. Of course, a pure MIMD vector core is useless unless you also have the correct parallel software model to use it with, which is what COSA is all about.

**See Also:**
- Transforming the TILE64 into a Kick-Ass Processor
- How to Solve the Parallel Programming Crisis
Heralding the Impending Death of the CPU

New Interfaces for Parallel Programming

It is no secret that I am a fan of Jeff Han’s multi-touch screen interface technology. I think it is a great interface for future parallel programming in a COSA development environment because it makes it easy to visually manipulate and experiment with a large pool of plug-compatible parallel components. Even though it is easy to represent objects in three dimensions on a 2-D surface, I always felt that it would be nice to be able to virtually immerse oneself into a program under construction. I envisioned a day when an application designer could visually jump into and navigate through a running program under construction. I think it would add a natural and highly productive feel to application design akin to moving around in a familiar environment such as one’s home. The reason is that the brain loves 3-D. It can quickly make sense of and adapt to its surroundings by getting a fix on its position and orientation relative to other nearby objects. Also, previously encountered 3-D configurations or patterns will add to overall scene comprehension in a way that makes it easy to spot out of place or wrongly connected objects that can potentially ruin a design.

This is the reason that I was pleasantly surprised to read about the newest 3-D user interfaces currently being shown at SIGGRAPH 2009, as reported by MIT’s Technology Review. I think this stuff is awesome. In my opinion, anybody who is interested in the future of parallel computing should get acquainted with these new technologies. The availability of a virtual touch interface is particularly interesting, not only because it adds another sensory modality that will reinforce the 3-D experience, but also because it may open the door for even the blind to enjoy computing in a powerful way that they are already familiar with. It might even provide them with an effective sense of 3-D vision through touch, a great help for navigation.

See Also:
How to Solve the Parallel Programming Crisis
Why I Hate All Computer Programming Languages

Parallel Computing: Why the Future Is Compositional, Part I

Part I, II

Abstract
In this two-part article, I will argue that there is a way to define, organize and use software objects that will transform computer programming from the complex unprincipled mess that it currently is into an orderly, easy to use and pristine compositional paradise. Please read Why I Hate All Computer Programming Languages before continuing.

The Promise

The computer revolution will not come of age until we are all computer programmers by default, whether we know it or not. This will not happen until software construction becomes strictly compositional in nature. Imagine being able to extend the functionality of an application simply by attaching a new component to it with the click of the mouse button. Computer programming for the masses, that is the promise of compositional software. The idea is to create such a huge and easily navigable repository of components that any application can be quickly and flawlessly assembled from existing objects using an intuitive and graphical user interface.

Past Failures

Computer scientists have known for decades that componentizing software is the best way to take advantage of the enormous potential of reuse. As it is, a huge part of computer programming consists of reinventing the wheel, leading to a waste of resources and low productivity. There have been many attempts at devising compositional software tools over the years but they have all fallen short of the goal of truly simplifying computer programming. In my opinion, all past and current approaches suffer from one or more of the following flaws.

1. Algorithmic (implicitly sequential) model.
2. Non-interactive, textual (descriptive) interface.
3. Non-synchronous (non-deterministic) processing.
4. Heterogeneous model.
5. No fundamental software building blocks.
6. No plug-compatible components.
7. No coherent component/object classification system.

The Software Tree

The COSA software model, in contrast to the above, is non-algorithmic (implicitly parallel), graphical, interactive, synchronous and homogeneous. It is based on fundamental building blocks called sensors and effectors, and organizes all objects into plug-compatible components that easily fit into a hierarchical compositional architecture.
The tree is the perfect metaphor for the new approach to software construction and classification that I am proposing. Every COSA application (component) is organized internally as a tree and is itself a node (or branch) in the universal tree of all COSA applications. The leaves of the tree are the fundamental building blocks.

In Part II, I will explain why the use of a tree classification system for all software components is essential to the success of component-based software development.

See Also:
- COSA, A New Kind of Programming
- New Interfaces for Parallel Programming
- How to Solve the Parallel Programming Crisis
- The COSA Control Hierarchy
- Why I Hate All Computer Programming Languages

Parallel Computing: Why the Future Is Compositional, Part II

Part I, II

Abstract

In Part I, I argued that software construction should be strictly compositional and modeled on a hierarchical framework. I wrote that every computer application (or sub-component) will be organized like a tree and will be a branch in the universal tree of all applications. Below, I argue against the linguistic approach to computer programming and I explain why the tree is the perfect organizational model for software composition.

No Fun For You
Many have argued that the linguistic approach to programming has adequately proved itself in the last half-century and that there is no reason to change to a new approach. My response is that the linguistic approach is what is keeping the computer revolution from reaching its true potential. The reason is that language is purely descriptive in nature and, as such, forces the application designer to learn a complex set of description rules that have nothing to do with application design. The designer’s freedom to create is limited by the non-interactive and tedious nature of the process. Worse, there is no quick and easy way to reuse previously written code in new programs. The end result is pages of code that are hard to decipher and debug because the relationships between objects are not obvious. Creating things is supposed to be fun but writing code is anything but; unless, of course, you’re a computer geek. Is it any wonder that certain cultures and women in general are under-represented in computer programming? The stuff that is out there is boring, ugly and painful. Some of us have no patience for it.

The Other Way Around

Computer programming, as it is now practiced, consists of communicating one’s intentions to a computer via a special language that is easy for the computer to interpret. From my perspective, this is primarily what is wrong with the linguistic approach. A tool should accommodate the user. That is to say, the programmer should not have to learn how computers think in order to communicate with them. I think it should be the other way around. I think it is the computer that should be trying its best to understand the programmer’s intentions and react instantly and intuitively. Consider that humans learn best through trial and error, through direct manipulation and interaction. This is the reason that video games and programs like Google Earth and graphic editors are such a pleasure to use. My thesis is that, since programming is a constructive activity and since humans enjoy using parts to build bigger parts, the programming environment should look and feel like an interactive video game for constructing things. The computer should make it as easy as possible for the programmer. While focusing on the task at hand, the programmer should be able to forget that there is a computer somewhere doing its magic to keep the interface working flawlessly. Programming should be fun.

Masters, Slaves and Hierarchies

Nature teaches us that the best architecture for composition and classification is the tree. The proper function of governments and other organizations is possible only because they are hierarchically structured. My thesis is that the main reason that compositional programming tools have not revolutionized software construction is that there are no master/slave relationships between components. The effective control of behavior (turning an object on or off) among cooperating entities is impossible without such relationships. Object-oriented class hierarchies, as a counter example, are not organized with behavior control in mind. Objects are really of the client/server type, which is not the same thing as master/slave since neither controls the other. Constructing a hierarchical tree of concurrent behaving objects is impossible unless there is a clear understanding as to what controls what. That is to say, every object must be designed in such a way as to form master/slave relationships with other compatible objects.

Automatic Classification
Another problem with current compositional tools has to do with locating the right components from the component repository. A keyword-driven component database can quickly turn into a horrible mess. The trick is to have a system that automatically organizes the components in the database. That is to say, the system should automatically position every component at its correct level and correct branch in the tree. This is possible only if master/slave complementarity is consistent from top to bottom. It suffices to count the levels in the hierarchy. For example, an object may interact (via a gender-enforced lateral connector) with another object inhabiting the same level in the hierarchy but can neither control it (turn it on or off) nor be controlled by it. Thus a designer can quickly locate a component by traversing the appropriate branch.

**Automatic Factoring**

Proper component factoring is essential to effective reuse and composition. Factoring consists of breaking down a component into two or more smaller components that can be used separately. This should not be the responsibility of the programmer but should be handled automatically by the classification mechanism. This can be done by comparing components to see if certain parts of a component are repeated in others. The system can instantly and autonomously break the components into smaller parts. This eliminates redundancy and atomizes parts into the finest possible granularity at a given level of abstraction. Fine granularity makes for easier composition by broadening the choice of components.

**The Universal Tree**

I foresee the need to maintain a single universal tree of all software components somewhere on the Internet, preferably with the branches distributed throughout the Cloud.

![Universal Tree Diagram](attachment:image.png)

The figure above depicts a hierarchy of components. This could be a single branch within a larger universal tree of applications. As the tree becomes populated, new applications become easier to compose using existing components. Organizations and individuals can still have secret components on their own private servers, if they so desire, but they should make sure that their component repository is properly synchronized with the free global tree. Additionally, it should be possible to insert a component into the public tree without publishing its internal composition. Having a public tree will encourage reuse and eliminate frequent reinvention of the wheel. It goes without saying that all software development tools should have access to the universal tree. More to come.

**See Also:**
The COSA Control Hierarchy

Abstract

Every COSA software application is organized like a tree. This is a fundamental aspect of COSA programming. In this article, I will argue that the use of a control or command hierarchy is the most effective and simplest way to design parallel applications and precisely control many objects operating in parallel. Please read the previous multi-part article, COSA: A New Kind of Programming, before continuing.

Why Use a Hierarchy?

The brain’s memory structure is organized as a hierarchy just like a COSA application. This is not surprising since they both consist of many parallel acting entities. There are excellent reasons for this arrangement in COSA. Examples are the grouping or classification of interrelated components, the reuse or sharing of components, easy program comprehension, the control of attention and the selection and coordination of tasks.
The figure above is a simplified representation of a COSA application shown in tree form. The leaf nodes are the low-level components that contain actual sensors and effectors. The other nodes (small round circles) are the supervisor components. The trunk of the tree is the main supervisor component, i.e., the application. Remember that node children are called slaves in COSA and that a supervisor can control an indefinite number of slaves. Here's another depiction of a supervisor and its slaves:

**Object Classification and Reuse**

One experiences an exponential rise in power and sophistication as one traverses toward the trunk of the program tree, away from the leaf nodes. A tree structure not only facilitates easy program comprehension, it also makes it easy to search the component repository for a particular component because the repository uses the exact same tree structure to store components. Related components are easily spotted because they lie on the same branch of the tree.

**Attention Control**

The brain has a finite number of motor effectors to choose from. This means that the effectors must be shared by a plurality of tasks (behaviors). Unless behaviors are carefully selected for activation and deactivation at the right time, motor conflicts will invariably crash the system. A tree hierarchy makes it possible for the brain’s action selection mechanism to easily pick non-conflicting branches of the tree for motor output. A similar method is used in a COSA program to solve motor conflicts. Even though effectors can be easily duplicated and executed in parallel, there are occasions when this is not possible. An example is a robotic system with a fixed set of motor effectors. Attention control allows the program to activate certain components while deactivating others. It forces the program to focus on a narrow set of tasks at a time, thus preventing failures. This is easier than it sounds because the COSA development environment will automatically alert the programmer of any real or potential motor conflicts (**Principle of Motor Coordination**).

**Conclusion**

The primary goal of the COSA visual design tools is to make it easy to compose complex, rock-solid applications as quickly as possible. I think the use of a tree architecture for program organization is part of the future of parallel programming. Hang in there.

**See Also:**
[COSSA, A New Kind of Programming](#)
[New Interfaces for Parallel Programming](#)
[How to Solve the Parallel Programming Crisis](#)
[The COSA Control Hierarchy](#)
[Why I Hate All Computer Programming Languages](#)
Why Timing Is the Most Important Thing in Computer Programming

An Analogy

Architects, carpenters, mechanical and civil engineers expect things to have specific sizes. If, during construction, the sizes of parts are found to be different than specified or do not match properly with other parts, a search will be conducted to find the cause of the discrepancy and correct it. In this article, I will argue that size (distance) is to architecture what timing is to computing. In other words, deterministic timing is essential to software reliability.

Deterministic Timing in Reactive Concurrent Systems

In a Von Neumann computer, it is unrealistic to expect every operation of a program to occur at a specific relative time based on a real time clock. The reason is that operations must wait their turn to be processed by the CPU. The problem is twofold. First, the CPU load varies from moment to moment and second, algorithmic software is such that it is impossible to predict the duration of every subroutine in an average program. However, it is possible to simulate a parallel, signal-based reactive system based on a virtual system clock. In such a system, every operation is required to be purely reactive, that is to say, it must execute within one system cycle immediately upon receiving its signal. These two requirements (every elementary operation is reactive and is processed in one cycle) are sufficient to enforce deterministic timing in a program, based on the virtual system clock. Deterministic timing means that reaction times are predictable. It does not mean that all the events (such as the movements of a mouse) that trigger the reactions are predictable. However, one event may trigger one or more chains of reactions and these, too, are deterministic, relative to the first event.

Timing Watchdogs

One nice thing about concurrent reactive systems is that interval detectors can be used to automatically find invariant intervals between any number of signals within a program. We can place timing watchdogs at various places in the program (this, too, can be done automatically) so that any discrepancy between an expected interval and the actual measured value will trigger an alarm. The temporal signature of a reactive system remains fixed for the life of the system and this makes for rock-solid reliability. So there are only two ways a timing watchdog can trigger an alarm; either the code was modified or there was a local physical system failure.

Automatic Discovery and Resolution of Data and Event Dependencies

Another nice aspect of concurrent reactive systems is that they are based on change. A change to a program variable is immediately communicated to every part of the program that may be affected by the change. The development environment can automatically link every entity or operator that changes a variable to every sensor that detects the change. This essentially eliminates blind code.

Side Effects in Complex Systems
We all know how hard it is to maintain complex legacy systems. A minor modification often triggers unforeseen side effects that may lay dormant for a long time after release. The right combination of events may cause a system failure that can be directly linked to the modification. For this reason, most system managers will look for alternative ways around a problem before committing to modifying the code. The side effects problem not only places an upper limit to the complexity of software systems, but the cost of continued development and maintenance soon becomes prohibitive. This is a problem that will never go away as long as we continue to use algorithmic systems. Luckily, the problem becomes nonexistent in the temporally deterministic reactive system that I described above. This is because blind code elimination and the use of timing watchdogs make it impossible to introduce undetected side effects. Indeed, timing is so deterministic and precise in a purely reactive system that the smallest modification is bound to violate a temporal expectation and trigger an alarm. It is up to the designer to either accept the new temporal signature, change it or revert back to the old code. As a result, we can build our software as complex as possible without having to worry about hidden bugs. In fact, and this is rather counterintuitive, more complex software will mean more timing constraints and thus more correct and robust systems, i.e., systems that work according to specs.

**The Entire Computer Industry Is Wrong**

I know this sounds arrogant but it is true. We have been building and programming computers the wrong way from the beginning, since the days of Charles Babbage and Lady Ada Lovelace, in fact. Babbage had an excuse because he clearly wanted a machine for solving long mathematical calculations. Modern computer designers should know better because the computer is no longer a mere calculator. To solve the unreliability and low productivity problems that have been plaguing the industry, we must change to a new way of doing things. We cannot continue with the same old stuff. It is no good. We must abandon the algorithmic software model and adopt a concurrent reactive model. And what better time is there to change than now, seeing that the industry is just beginning to transition from sequential computing to massive parallelism? Reactive concurrent systems are right at home in a parallel, multicore universe. We must change now or continue to suffer the consequences of increasingly unreliable and hard to develop software. This is what Project COSA is about.

**Parallel Computing: The Fourth Crisis, Part I**

[Part I, II, III]

**The Memory Bandwidth Problem**

The memory bandwidth problem is worse than the parallel programming crisis because it appears that the solution will require some sort of breakthrough in quantum tunneling or optical computing. This could happen now or ten years from now. The computer industry cannot wait.
Somehow, it must forge a solution as soon as possible, otherwise even big chip companies like Intel, IBM, and AMD will see their profit stream dwindle down to a trickle. These are companies that rely on constant improvements in performance to maintain their competitive edge. Even advances in new semiconductor materials such as graphene will not solve the problem because it does not eliminate the inherent physical limits.

The Four Crises

The computer world is battling four major crises, in my opinion. The first two, software reliability and programmer productivity, have been with us since the seventies and they show no sign of abating. Parallel programming, the third crisis, emerged only a few years ago with the commercialization of multicore processors. I have forcefully argued in the past that these three crises are really one and the same. What I mean is that it is a single problem that calls for a single solution. The fourth major crisis is the memory bandwidth problem. This is the worst one of them all, in my opinion, because, whether or not the other three are solved, slow memory threatens to repeal Moore’s law and bring progress in the field to a screeching halt. Nobody wants that to happen, at least not in the foreseeable future.

A New Kind of Computer

I think the world needs a new kind of computer. I have been thinking about a radically new way of achieving blazingly fast parallel computing without the use of a central processor. The idea has been percolating in my mind for quite some time. It is still partially baked but I think that it is worth pursuing. Essentially, I believe that the age of the central processor must come to an end. The communication bottleneck that results from segregating the processor from memory is simply intolerable. My idea is based primarily on certain characteristics of the COSA software model. I will describe what I have in mind in greater details in Part II of this article.

See Also:
How to Solve the Parallel Programming Crisis

Parallel Computing: The Fourth Crisis, Part II

Part I, II, III

Abstract

Previously, I wrote that in order to solve the memory bandwidth problem, the computer industry must design a new kind of computer, one that does away with the central processor. Alternatively, the industry must wait for some sort of breakthrough in quantum tunneling or optical memory. Here, I will argue that a decentralized design will result in extremely fast parallel processing. I further argue that the deterministic timing afforded by a synchronous
processing model can be used to implement a signaling mechanism that significantly reduces the requirement for a huge number of physical signal pathways.

**Applicability**

It should be noted that the multicore memory bandwidth problem is not universal. The use of on-chip caches together with an effective load balancing system will provide adequate scalability for many applications. If data and event dependencies are localized, inter-core communication is minimal and performance will not degrade as the number of cores increases. It is possible to minimize latency by automatically preventing data from drifting to distant caches far from the cores that service them. In this light, I should mention that Tilera’s Tile64 multicore processor with its iMesh™ technology is potentially well suited for this type of parallel computing (see links below). Unfortunately, there are many applications with numerous dependencies that will not scale regardless of the size of cache memory. Let me add here that a single breakthrough in quantum tunneling or optical memory that eliminates memory bus contention would solve the problem, in which case the ideas that I propose in this article would no longer be necessary.

**The Brain**

My favorite example of a non-centralized, high event-dependency parallel computing system is the brain. In a conventional computer, a central processor loads a stream of op-codes (instructions) from memory, deciphers their meaning and executes them one at a time. The brain, by contrast, uses as many tiny processors (neurons) as there are instructions in its program. Every elementary processor is directly attached to its data, so to speak. The most obvious advantage of this approach is that there is no instruction stream and therefore no need for either a central instruction decoder or an execution unit. Another advantage is the inherent parallelism, an extremely powerful feature that the brain uses to process vast quantities of signals simultaneously. Memory bandwidth is not an issue for the brain. I envision a new programmable, massively parallel computer technology where huge numbers of elementary processors are directly embedded into memory. The performance increase would be tremendous compared to a conventional computer.

**Event Dependencies and Signal Pathways**

Signaling is used for event timing purposes. It implies the ability to create a connection between a signal emitter and a receiver. We normally do not think of it as such but a computer program is a communication system. Every instruction in an algorithm sends a signal to the next instruction in the sequence meaning essentially “I’m done, now it’s your turn”. In a true parallel system (e.g., the brain), a predecessor instruction can send a signal to an indefinite number of successors: bus contention is not an issue because signal pathways are not shared. Signaling between program elements in a Von Neumann computer, by contrast, is accomplished via a memory addressing mechanism. This is a relatively slow process because it requires shared address decoder, memory controller, address and data buses that permit only one memory location to be accessed at a time. This is fine in most cases if the memory bandwidth is high enough. However, it becomes a major headache in future multicore environments when thousands or even tens of thousands of cores must have concurrent access to shared memory.
That’s when bus contention rears its ugly head and drives a nail in the coffin of scalability, so to speak.

**Self-Activation via Deterministic Timing**

The brain solves the contention problem by using dedicated axonic fibers to directly connect processing elements together. It would be nice if we could use the same approach in our computers but the ensuing proliferation of pathways would be prohibitive, given the current state of our field programmable technology. I’ve been thinking that one way to drastically reduce the number of signals and pathways is to take advantage of the deterministic timing that is inherent in a synchronous programming model such as the COSA Software Model. The idea is that temporally related elements (operators) would be programmed to know precisely when to activate themselves. All that is needed is a reset pulse and a heartbeat. In other words, all temporally related processors would be connected to only two lines, like pearls on a necklace. Each element would self-activate at their programmed time by counting heartbeat pulses. Some would activate simultaneously while others would do so sequentially according to the program. Multiple “necklaces” could be laid side by side on a single die to form an entire application. Programming the elements (or cells) could be done via conventional memory addressing. Of course, since some elements (such as an AND cell or an addition effector) will require multiple inputs, field-programmable interconnects would allow cross connections between elements residing in different necklaces. As I wrote in a previously, this is still a partially baked idea. I hope to have time to work on it some more in order to iron out the kinks and fill in the details.

**Data Dependencies**

It would be great if mimicking the brain’s decentralized architecture would solve all of our problems but, unfortunately, there are things that computers must do that the brain does not have to do. Directly attaching an elementary processor to its data would speed things up marvelously but it would only work up to a point because variables are not always local to the processors that use them. Our programs are expected to do things like moving or copying data from one place in memory to another. An example is matrix manipulation where a variable or constant may be copied to every cell in an array or one array may be used to transform another. In my Part III, I will describe a possible way to extend this model to maximize data bandwidth.

**See Also:**
- How to Solve the Parallel Programming Crisis
- Transforming the TILE64 into a Kick-Ass Parallel Machine

**Parallel Computing: The Fourth Crisis, Part III**

Part I, II, III

**Change of Heart**
Due to various considerations that I don't want to go into, I have decided not to write the third part of this article. Please don't fret. I may change my mind later as I often do.

Another Me-Too Chinese Project and Intel's Achilles' Heel

Chinese Supercomputer

The Chinese recently announced their intentions to build a supercomputer based on their Loongson processor, a general-purpose MIPS-based CPU developed at the Institute of Computing Technology, part of the Chinese Academy of Science. What a waste of good research talent! The hard reality is that any new processor that does not solve the parallel programming crisis is on a fast road to failure. No long march to victory in sight for the Loongson, sorry.

China should be trying to become a leader in this field, not just another me-too follower. There is an unprecedented opportunity to make a killing in the parallel processor industry in the years ahead. Intel may have cornered the market for now but they have an Achilles' heel: they are way too big and way too married to last century's flawed computing paradigms to change in time for the coming massively parallel computer revolution. Their x86 technology will be worthless when that happens. The trash bins of Silicon Valley will be filled with obsolete Intel chips.

Here's the problem. The computer industry is in a very serious crisis due to processor performance limitations and low programmer productivity. Going parallel is the right thing to do but the current multicore/multithreading approach to parallel computing is a disaster in the making. Using the erroneous Turing Machine-based paradigms of the last sixty years to solve this century's massive parallelism problem is pure folly. Intel knows this but they will never admit it because they've got too much invested in the old stuff. Too bad. They will lose the coming processor war. That's where China and Intel's competitors can excel if they play their cards right. The truth is that the thread concept (on which the Loongson and Intel's processors are based) is the cause of the crisis, not the solution. There is an infinitely better way to build and program computers that does not involve threads at all. Sooner or later, an unknown startup will pop out of nowhere and blow everybody out of the water.

My advice to China, Intel, AMD and the other big dogs in the computer business is this: first invest your resources into solving the parallel programming crisis. Only then will you know enough to properly tackle the embedded systems, supercomputing and cloud computing markets. Otherwise be prepared to lose a boatload of dough. When that happens, there shall be much weeping and gnashing of teeth but I'll be eating popcorn with a smirk on my face and saying "I told you so".

See Also:
How to Solve the Parallel Programming Crisis
Plurality’s HAL: Kicking Ass on Core Street

Misbehaving

I don’t do reviews and I don’t accept advertising. I just tell it like I see it and I try hard to be as brutally honest as possible when I do tell it. So if you contact me and ask me to take a look at your technology, know in advance that I never pull my punches. I will tear your pretty little baby into pieces if it so much as smells a little bit funny. Those of you who are familiar with my views on the multicore crisis already know that I am hard to please. I have strong ideas about how a multicore processor should behave and I will not compromise. And let me tell you, there is a lot of misbehaving going on out there. Right now, the multicore landscape is ugly as sin. Some people need an ass whipping, I swear. I especially dislike the crap being put out by industry leaders like Intel, AMD and the others. Lately I was beginning to despair of catching a glimpse of even a tiny hint of beauty in this dreadful landscape any time soon. That’s when someone brought Plurality Ltd to my attention.

Plurality’s Hypercore Processor

One cannot help being mesmerized by the unusual beauty of the colorful animation on Plurality’s welcome page. One is immediately struck with an irresistible urge to find out more about their technology. But Plurality, a small privately owned Israeli company founded in 2004 and headquartered in Netanya, is beautiful in more ways than one. In my opinion, their product, the Hypercore Processor, is a mixture of hyper beauty and a touch of ugliness. I’ll get to the ugly part in a minute. What caught my attention is that the company seems to have solved several crucial problems in multicore design that a behemoth like Intel, with all the gazillions that it lavishes on multicore research labs around the world, probably never even recognized as being crucial. Plurality’s engineers figured out a way to design a general purpose, fine-grained, self-balancing, auto-scalable, MIMD, 256-core processor! How cool is that?

The Good

The claims made by Plurality regarding the Hypercore’s so-called synchronizer/scheduler sound almost too good to be true. Take a look at the block diagram of their multicore architecture:
They’re claiming that their scheduler is able to perform near perfect load balancing. They’re also claiming that the cores can access shared memory without creating a bottleneck. This means that they are able to deliver fine-grain parallelism in a general purpose MIMD environment! This is incredible stuff. About a year ago I came up with a scheme to do fine-grain, real-time load balancing in a multicore computer. But then I realized that using a central controller to feed instructions one by one into a bunch of cores would not scale up as the number of cores increases. Here's a block diagram of my original design of the COSA multicore processor for comparison:

I also worried that having multiple memory caches would introduce unacceptable latencies if the cores had to frequently access memory in distant caches. In other words, my scheme was not scalable. I eventually thought of a decentralized mechanism (unpublished) that distributes the load-balancing task among the cores while keeping inter-core communication to a minimum. Apparently, Plurality seems to have solved, not only the scheduling problem, but the memory access bottleneck as well. How? I don’t know. It sounds almost like magic. If true, it means that at least one person at Plurality must be fully awake with a humongous thinking cap on.
The Bad

I am not hiding the fact that I am impressed with Plurality’s technology. Them Israeli boys are kicking major ass and taking names on Core Street, there is no doubt about it. They seem to have delivered some of the really nice things that I’ve been craving for, for a long time now. Intel and AMD should be ashamed of themselves. Does this mean that I believe that Plurality has solved the multicore problem? The answer is no. Nobody’s perfect and, as they say, Rome was not built in one day. Still, when I have a bone to pick I must pick it. I don’t like Plurality’s programming model. I think their Task Oriented Programming model is crap. In this respect, they’re making the same mistake that everybody else is making. I understand the motivation behind retaining some similarity and compatibility with the legacy technology. This situation may look somewhat analogous to the time when, more than a century ago, the first horseless carriages had wheels and stepping boards that looked like those of the buggies that they replaced. It’s not the same thing. Plurality's multicore processor may look and sound like a Ferrari or a Lamborghini but it is still attached to a mule because there is no fuel in the tank. Here is their pitch:

Programming the Hypercore, beyond what is required when programming a standard uncore processor, is based on Plurality's unique Task Oriented Programming. The only programmer requirement is to perform a simple partitioning of the algorithm into specific tasks, and compose a task map accordingly.

This is pure BS, of course. There is nothing simple about partitioning a sequential algorithm into parallel tasks and Plurality’s task oriented programming model is no different than breaking a program into threads. Plurality has simply given multithreading a new name in the hope that nobody would notice. Worse, the programmer has to compose a task dependency map on top of it! Come on, guys. You know better than that. I know you worked your butts off to get to where you are but your work is not done yet. Your programming model stinks. Your work will not be done until you come up with a processor that is designed from the ground up to support a truly parallel, synchronous reactive programming model like the COSA software model. Besides, what’s with the linguistic nonsense? Why would you want people to continue to program code like they did in the sequential days? This does not make sense. You either have a parallel processor or you don't. No way you’re going to tell me that you want to jump on board a shiny new 21st century spaceship and bring a pile of putrid 20th century garbage with you. Believe me, you will not be allowed into port until you jettison all that crap overboard. Take a look at Jeff Han's multitouch screen technology. That's the future interface of parallel programming. Just drag them and drop them.

Conclusion

Even though Plurality claims that “the Hypercore programmer needs not be familiar with the details of the machine, nor how its resources are interconnected and managed” (which is a very desirable and cool thing to have) I still think their programming model is crap. Having said that, I think Plurality is light years ahead of everybody else. I am not Jewish but I can tell you guys that, if you play your cards right, all the gentiles will come flocking like migratory birds to
worship at your feet. You guys are riding the crest of a revolutionary wave. You have demonstrated that you've got what it takes to design and build a kick-ass multicore processor. You aren't there yet but you are onto something big, something that can potentially catapult your nation, Israel, into an economic and technological powerhouse, the Lion of Judah and all that. And we all know what that means. The correct multicore architecture is the legendary pot of gold at the end of the rainbow. The Israeli government should declare Plurality a national treasure and immediately pump 100 million dollars into its coffers. OK, it’s kind of a joke, but I’m only half kidding.

See Also:
Transforming the TILE64 into a Kick-Ass Parallel Machine
Tilera's TILE64: The Good, the Bad and the Possible

Why Software Is Bad and What We Can Do to Fix It

Abstract

There is something fundamentally wrong with the way we create software. Contrary to conventional wisdom, unreliability is not an essential characteristic of complex software programs. In this article, I will propose a silver bullet solution to the software reliability and productivity crisis. The solution will require a radical change in the way we program our computers. I will argue that the main reason that software is so unreliable and so hard to develop has to do with a custom that is as old as the computer: the practice of using the algorithm as the basis of software construction (*). I will argue further that moving to a signal-based, synchronous (**), software model will not only result in an improvement of several orders of magnitude in productivity, but also in programs that are guaranteed free of defects, regardless of their complexity.

Software Is Bad and Getting Worse

The 'No Silver Bullet' Syndrome

Not long ago, in an otherwise superb article [pdf] on the software reliability crisis published by MIT Technology Review, the author blamed the problem on everything from bad planning and business decisions to bad programmers. The proposed solution: bring in the lawyers. Not once did the article mention that the computer industry's fundamental approach to software construction might be flawed. The reason for this omission has to do in part with a highly influential paper that was published in 1987 by a now famous computer scientist named Frederick P. Brooks. In the paper, titled "No Silver Bullet--Essence and Accidents of Software Engineering", Dr. Brooks writes:

But, as we look to the horizon of a decade hence, we see no silver bullet. There is no single development, in either technology or in management technique, that by itself promises even one order-of-magnitude improvement in productivity, in reliability, in simplicity.
Not only are there no silver bullets now in view, the very nature of software makes it unlikely that there will be any—not inventions that will do for software productivity, reliability, and simplicity what electronics, transistors, and large-scale integration did for computer hardware.

No other paper in the annals of software engineering has had a more detrimental effect on humanity’s efforts to find a solution to the software reliability crisis. Almost single-handedly, it succeeded in convincing the entire software development community that there is no hope in trying to find a solution. It is a rather unfortunate chapter in the history of programming. Untold billions of dollars and even human lives have been and will be wasted as a result.

When Brooks wrote his famous paper, he apparently did not realize that his arguments applied only to algorithmic complexity. Most people in the software engineering community wrongly assume that algorithmic software is the only possible type of software. Non-algorithmic or synchronous reactive software is similar to the signal-based model used in electronic circuits. It is, by its very nature, extremely stable and much easier to manage. This is evident in the amazing reliability of integrated circuits. See Targeting the Wrong Complexity below.

Calling in the lawyers and hiring more software experts schooled in an ancient paradigm will not solve the problem. It will only be costlier and, in the end, deadlier. The reason is threefold. First, the complexity and ubiquity of software continue to grow unabated. Second, the threat of lawsuits means that the cost of software development will skyrocket (lawyers, experts and trained engineers do not work for beans). Third, the incremental stop-gap measures offered by the experts are not designed to get to the heart of the problem. They are designed to provide short-term relief at the expense of keeping the experts employed. In the meantime, the crisis continues.

Ancient Paradigm

Why ancient paradigm? Because the root cause of the crisis is as old as Lady Ada Lovelace who invented the sequential stored program (or table of instructions) for Charles Babbage's analytical engine around 1842. Built out of gears and rotating shafts, the analytical engine was the first true general-purpose numerical computer, the ancestor of the modern electronic computer. But the idea of using a step by step procedure in a machine is at least as old as Jacquard's punched cards which were used to control the first automated loom in 1801. The Persian mathematician Muhammad ibn Mūsā al-Khwārizmī is credited for having invented the algorithm in 825 AD, as a problem solving method. The word algorithm derives from 'al-Khwārizmī.'

Why The Experts Are Wrong

Turing's Baby

Early computer scientists of the twentieth century were all trained mathematicians. They viewed the computer primarily as a tool with which to solve mathematical problems written in an algorithmic format. Indeed, the very name computer implies the ability to perform a calculation and return a result. Soon after the introduction of electronic computers in the 1950s, scientists fell in love with the ideas of famed British computer and artificial intelligence pioneer, Alan Turing. According to Turing, to be computable, a problem has to be executable on an abstract computer called the universal Turing machine (UTM). As everyone knows, a UTM (an infinitely long tape with a movable read/write head) is the quintessential algorithmic computer, a direct
descendent of Lovelace's sequential stored program. It did not take long for the Turing computability model (TCM) to become the de facto religion of the entire computer industry.

**A Fly in the Ointment**
The UTM is a very powerful abstraction because it is perfectly suited to the automation of all sorts of serial tasks for problem solving. Lovelace and Babbage would have been delighted, but Turing's critics could argue that the UTM, being a sequential computer, cannot be used to simulate real-world problems which require multiple simultaneous computations. Turing's advocates could counter that the UTM is an idealized computer and, as such, can be imagined as having infinite read/write speed. The critics could then point out that, idealized or not, an infinitely fast computer introduces all sorts of logical/temporal headaches since all computations are performed simultaneously, making it unsuitable to inherently sequential problems. As the saying goes, you cannot have your cake and eat it too. At the very least, the TCM should have been extended to include both sequential and concurrent processes. However, having an infinite number of tapes and an infinite number of heads that can move from one tape to another would destroy the purity of the UTM ideal. The point that I want to make here is that the TCM is inadequate as a modern computing model because 1) it does not address parallelism and 2) timing is not an inherent part of the model.

**The Hidden Nature of Computing**
The biggest problem with the UTM is not so much that it cannot be adapted to certain real-world parallel applications but that it hides the true nature of computing. Most students of computer science will recognize that a computer program is, in reality, a behaving machine (BM). That is to say, a program is an automaton that detects changes in its environment and effects changes in it. As such, it belongs in the same class of machines as biological nervous systems and integrated circuits. A basic universal behaving machine (UBM) consists, on the one hand, of a couple of elementary behaving entities (a sensor and an effector) or actors and, on the other, of an environment (a variable).

<table>
<thead>
<tr>
<th>Universal Behaving Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actors</td>
</tr>
<tr>
<td>Sensor</td>
</tr>
<tr>
<td>Environment</td>
</tr>
<tr>
<td>Variable</td>
</tr>
</tbody>
</table>

More complex UBM's consist of arbitrarily large numbers of actors and environmental variables. This computing model, which I have dubbed the behavioral computing model (BCM), is a radical departure from the TCM. Whereas a UTM is primarily a calculation tool for solving algorithmic problems, a UBM is simply an agent that reacts to one or more environmental stimuli. As seen in the figure below, in order for a UBM to act on and react to its environment, sensors and effectors must be able to communicate with each other.

UBM Communication Pathways
The main point of this argument is that, even though communication is an essential part of the nature of computing, this is not readily apparent from examining a UTM. Indeed, there are no signaling entities, no signals and no signal pathways on a Turing tape or in computer memory. The reason is that, unlike hardware objects which are directly observable, software entities are virtual and must be logically inferred.

**Fateful Choice**
Unfortunately for the world, it did not occur to early computer scientists that a program is, at its core, a tightly integrated collection of communicating entities interacting with each other and with their environment. As a result, the computer industry had no choice but to embrace a method of software construction that sees the computer simply as a tool for the execution of instruction sequences. The problem with this approach is that it forces the programmer to explicitly identify and resolve a number of critical communication-related issues that, ideally, should have been implicitly and automatically handled at the system level. The TCM is now so ingrained in the collective mind of the software engineering community that most programmers do not even recognize these issues as having anything to do with either communication or behavior. This would not be such a bad thing except that a programmer cannot possibly be relied upon to resolve all the dependencies of a complex software application during a normal development cycle. Worse, given the inherently messy nature of algorithmic software, there is no guarantee that they can be completely resolved. This is true even if one had an unlimited amount of time to work on it. The end result is that software applications become less predictable and less stable as their complexity increases.

**To Model or Not to Model**
It can be convincingly argued that the UBM described above should have been adopted as the proper basis of software engineering from the very beginning of the modern computer era. Note that, whereas a UBM can directly model a UTM, a UTM can only simulate a UBM (using an infinite loop). The reason is that a UBM is synchronous (**) by nature, that is to say, more than two of its constituent objects can communicate simultaneously.

![UBM Communication Diagram]

**UBM Communication**

In a UTM, by contrast, only two objects can communicate at a time: a predecessor and a successor.

![UTM Communication Diagram]

**UTM Communication**

The question is, is an emulation of a parallel synchronous system adequate for the purpose of resolving the communication issues mentioned in the previous paragraph? As explained below, the answer is a resounding yes. That is, as long as the processor is fast enough.
Turing's Monster

It is tempting to speculate that, had it not been for our early infatuation with the sanctity of the TCM, we might not be in the sorry mess that we are in today. Software engineers have had to deal with defective software from the very beginning. Computer time was expensive and, as was the practice in the early days, a programmer had to reserve access to a computer days and sometimes weeks in advance. So programmers found themselves spending countless hours meticulously scrutinizing program listings in search of bugs. By the mid 1970s, as software systems grew in complexity and applicability, people in the business began to talk of a reliability crisis. Innovations such as high-level languages, structured and/or object-oriented programming did little to solve the reliability problem. Turing's baby had quickly grown into a monster.

Vested Interest

Software reliability experts have a vested interest in seeing that the crisis lasts as long as possible. It is their raison d'être. Computer scientists and software engineers love Dr. Brooks' ideas because an insoluble software crisis affords them with a well-paying job and a lifetime career as reliability engineers. Not that these folks do not bring worthwhile advances to the table. They do. But looking for a breakthrough solution that will produce Brooks' order-of-magnitude improvement in reliability and productivity is not on their agenda. They adamantly deny that such a breakthrough is even possible. Brooks' paper is their new testament and 'no silver bullet' their mantra. Worst of all, most of them are sincere in their convictions.

This attitude (pathological denial) has the unfortunate effect of prolonging the crisis. Most of the burden of ensuring the reliability of software is now resting squarely on the programmer's shoulders. An entire reliability industry has sprouted with countless experts and tool vendors touting various labor-intensive engineering recipes, theories and practices. But more than thirty years after people began to refer to the problem as a crisis, it is worse than ever. As the Technology Review article points out, the cost has been staggering.

There Is a Silver Bullet After All

Reliability is best understood in terms of complexity vs. defects. A program consisting of one thousand lines of code is generally more complex and less reliable than a one with a hundred lines of code. Due to its sheer astronomical complexity, the human brain is the most reliable behaving system in the world. Its reliability is many orders of magnitude greater than that of any complex program in existence (see devil's advocate). Any software application with the complexity of the brain would be so riddled with bugs as to be unusable. Conversely, given their low relative complexity, any software application with the reliability of the brain would almost never fail. Imagine how complex it is to be able to recognize someone's face under all sorts of lighting conditions, velocities and orientations. Just driving a car around town (taxi drivers do it all day long, everyday) without getting lost or into an accident is incredibly more complex than anything any software program in existence can accomplish. Sure brains make mistakes, but the things that they do are so complex, especially the myriads of little things that we are oblivious to, that the mistakes pale in comparison to the successes. And when they do make mistakes, it is usually due to physical reasons (e.g., sickness, intoxication, injuries, genetic defects, etc.) or to external circumstances beyond their control (e.g., they did not know). Mistakes are rarely the result of defects in the brain's existing software.
The brain is proof that the reliability of a behaving system (which is what a computer program is) does not have to be inversely proportional to its complexity, as is the case with current software systems. In fact, the more complex the brain gets (as it learns), the more reliable it becomes. But the brain is not the only proof that we have of the existence of a silver bullet. We all know of the amazing reliability of integrated circuits. No one can seriously deny that a modern CPU is a very complex device, what with some of the high-end chips from Intel, AMD and others sporting hundreds of millions of transistors. Yet, in all the years that I have owned and used computers, only once did a CPU fail on me and it was because its cooling fan stopped working. This seems to be the norm with integrated circuits in general: when they fail, it is almost always due to a physical fault and almost never to a defect in the logic. Moore's law does not seem to have had a deleterious effect on hardware reliability since, to my knowledge, the reliability of CPUs and other large scale integrated circuits did not degrade over the years as they increased in speed and complexity.

Deconstructing Brooks' Complexity Arguments
Frederick Brooks' arguments fall apart in one important area. Although Brooks' conclusion is correct as far as the unreliability of complex algorithmic software is concerned, it is correct for the wrong reason. I argue that software programs are unreliable not because they are complex (Brooks' conclusion), but because they are algorithmic in nature. In his paper, Brooks defines two types of complexity, essential and accidental. He writes:

> The complexity of software is an essential property, not an accidental one.

According to Brooks, one can control the accidental complexity of software engineering (with the help of compilers, syntax and buffer overflow checkers, data typing, etc.), but one can do nothing about its essential complexity. Brooks then explains why he thinks this essential complexity leads to unreliability:

> From the complexity comes the difficulty of enumerating, much less understanding, all the possible states of the program, and from that comes the unreliability.

This immediately begs several questions: Why must the essential complexity of software automatically lead to unreliability? Why is this not also true of the essential complexity of other types of behaving systems? In other words, is the complexity of a brain or an integrated circuit any less essential than that of a software program? Brooks is mum on these questions even though he acknowledges in the same paper that the reliability and productivity problem has already been solved in hardware through large-scale integration.

More importantly, notice the specific claim that Brooks is making. He asserts that the unreliability of a program comes from the difficulty of enumerating and/or understanding all the possible states of the program. This is an often repeated claim in the software engineering community but it is fallacious nonetheless. It overlooks the fact that it is equally difficult to enumerate all the possible states of a complex hardware system. This is especially true if one considers that most such systems consist of many integrated circuits that interact with one another in very complex ways. Yet, in spite of this difficulty, hardware systems are orders of
magnitude more robust than software systems (see the [COSA Reliability Principle](#) for more on this subject).

Brooks backs up his assertion with neither logic nor evidence. But even more disturbing, nobody in the ensuing years has bothered to challenge the validity of the claim. Rather, Brooks has been elevated to the status of a demigod in the software engineering community and his ideas on the causes of software unreliability are now bandied about as infallible dogma.

**Targeting the Wrong Complexity**

Obviously, whether essential or accidental, complexity is not, in and of itself, conducive to unreliability. There is something inherent in the nature of our software that makes it prone to failure, something that has nothing to do with complexity per se. Note that, when Brooks speaks of software, he has a particular type of software in mind:

> The essence of a software entity is a construct of interlocking concepts: data sets, relationships among data items, algorithms, and invocations of functions.

By software, Brooks specifically means *algorithmic* software, the type of software which is coded in every computer in existence. Just like Alan Turing before him, Brooks fails to see past the algorithmic model. He fails to realize that the unreliability of software comes from not understanding the [true nature of computing](#). It has nothing to do with the difficulty of enumerating all the states of a program. In the remainder of this article, I will argue that all the effort in time and money being spent on making software more reliable is being targeted at the wrong complexity, that of algorithmic software. And it is a particularly insidious and intractable form of complexity, one which humanity, fortunately, does not have to live with. Switch to the right complexity and the problem will disappear.

**The Billion Dollar Question**

The billion (trillion?) dollar question is: What is it about the brain and integrated circuits that makes them so much more reliable in spite of their essential complexity? But even more important, can we emulate it in our software? If the answer is yes, then we have found the silver bullet.

**The Silver Bullet**

**Why Software Is Bad**

Algorithmic software is unreliable because of the following reasons:

- **Britleness**
  
  An algorithm is not unlike a chain. Break a link and the entire chain is broken. As a result, algorithmic programs tend to suffer from catastrophic failures even in situations where the actual defect is minor and globally insignificant.

- **Temporal Inconsistency**
  
  With algorithmic software it is virtually impossible to guarantee the timing of various processes because the execution times of subroutines vary unpredictably.
They vary mainly because of a construct called ‘conditional branching’, a necessary decision mechanism used in instruction sequences. But that is not all. While a subroutine is being executed, the calling program goes into a coma. The use of threads and message passing between threads does somewhat alleviate the problem but the multithreading solution is way too coarse and unwieldy to make a difference in highly complex applications. And besides, a thread is just another algorithm. The inherent temporal uncertainty (from the point of view of the programmer) of algorithmic systems leads to program decisions happening at the wrong time, under the wrong conditions.

Unresolved Dependencies
The biggest contributing factor to unreliability in software has to do with unresolved dependencies. In an algorithmic system, the enforcement of relationships among data items (part of what Brooks defines as the essence of software) is solely the responsibility of the programmer. That is to say, every time a property is changed by a statement or a subroutine, it is up to the programmer to remember to update every other part of the program that is potentially affected by the change. The problem is that relationships can be so numerous and complex that programmers often fail to resolve them all.

Why Hardware is Good
Brains and integrated circuits are, by contrast, parallel signal-based systems. Their reliability is due primarily to three reasons:

Strict Enforcement of Signal Timing through Synchronization
Neurons fire at the right time, under the right temporal conditions. Timing is consistent because of the brain's synchronous architecture (**). A similar argument can be made with regard to integrated circuits.

Distributed Concurrent Architecture
Since every element runs independently and synchronously, the localized malfunctions of a few (or even many) elements will not cause the catastrophic failure of the entire system.

Automatic Resolution of Event Dependencies
A signal-based synchronous system makes it possible to automatically resolve event dependencies. That is to say, every change in a system's variable is immediately and automatically communicated to every object that depends on it.

Programs as Communication Systems
Although we are not accustomed to think of it as such, a computer program is, in reality, a communication system. During execution, every statement or instruction in an algorithmic procedure essentially sends a signal to the next statement, saying: 'I'm done, now it's your turn.' A statement should be seen as an elementary object having a single input and a single output. It waits for an input signal, does something, and then sends an output signal to the next object. Multiple objects are linked together to form a one-dimensional (single path) sequential chain.
The problem is that, in an algorithm, communication is limited to only two objects at a time, a sender and a receiver. Consequently, even though there may be forks (conditional branches) along the way, a signal may only take one path at a time.

My thesis is that this mechanism is too restrictive and leads to unreliable software. Why? Because there are occasions when a particular event or action must be communicated to several objects simultaneously. This is known as an event dependency. Algorithmic development environments make it hard to attach orthogonal signaling branches to a sequential thread and therein lies the problem. The burden is on the programmer to remember to add code to handle delayed reaction cases: something that occurred previously in the procedure needs to be addressed at the earliest opportunity by another part of the program. Every so often we either forget to add the necessary code (usually, a call to a subroutine) or we fail to spot the dependency.

Event Dependencies and the Blind Code Problem
The state of a system at any given time is defined by the collection of properties (variables) that comprise the system's data, including the data contained in input/output registers. The relationships or dependencies between properties determine the system's behavior. A dependency simply means that a change in one property (also known as an event) must be followed by a change in one or more related properties. In order to ensure flawless and consistent behavior, it is imperative that all dependencies are resolved during development and are processed in a timely manner during execution. It takes intimate knowledge of an algorithmic program to identify and remember all the dependencies. Due to the large turnover in the software industry, programmers often inherit strange legacy code which aggravates the problem. Still, even good familiarity is not a guarantee that all dependencies will be spotted and correctly handled. Oftentimes, a program is so big and complex that its original authors completely lose sight of old dependencies. Blind code leads to wrong assumptions which often result in unexpected and catastrophic failures. The problem is so pervasive and so hard to fix that most managers in charge of maintaining complex mission-critical software systems will try to find alternative ways around a bug that do not involve modifying the existing code.

The Cure For Blind Code
To cure code blindness, all objects in a program must, in a sense, have eyes in the back of their heads. What this means is that every event (a change in a data variable) occurring anywhere in the program must be detected and promptly communicated to every object that depends on it. The cure consists of three remedies, as follows:

**Automatic Resolution of Event Dependencies**
The problem of unresolved dependencies can be easily solved in a change-driven system through the use of a technique called *dynamic pairing* whereby change detectors (comparison sensors) are associated with related operators (effectors). This way, the development environment can automatically identify and resolve every dependency between sensors and effectors, leaving nothing to chance.

**One-to-many Connectivity**
One of the factors contributing to blind code in algorithmic systems is the
inability to attach *one-to-many* orthogonal branches to a thread. This problem is non-existent in a synchronous system because every signal can be channeled through as many pathways as necessary. As a result, every change to a property is immediately broadcast to every object that is affected by the change.

**Immediacy**
During the processing of any element in an algorithmic sequence, all the other elements in the sequence are disabled. Thus, any change or event that may require the immediate attention of either preceding or succeeding elements in the chain is ignored. Latency is a major problem in conventional programs. By contrast, immediacy is an inherent characteristic of synchronous systems.

**Software Design vs. Hardware Design**
All the good things that are implicit and taken for granted in hardware logic design become explicit and a constant headache for the algorithmic software designer. The blindness problem that afflicts conventional software simply does not exist in electronic circuits. The reason is that hardware is inherently synchronous. This makes it easy to add orthogonal branches to a circuit. Signals are thus promptly dispatched to every element or object that depends on them. Furthermore, whereas *sensors* (comparison operators) in software must be explicitly associated with relevant effectors and invoked at the right time, hardware sensors are self-processing. That is, a hardware sensor works independently of the causes of the phenomenon (change) it is designed to detect. As a result, barring a physical failure, it is impossible for a hardware system to fail to notice an event.

By contrast, in software, sensors must be explicitly processed in order for a change to be detected. The result of a comparison operation is likely to be useless unless the operator is called at the right time, i.e., immediately after or concurrent with the change. As mentioned previously, in a complex software system, programmers often fail to update all relevant sensors after a change in a property. Is it any wonder that logic circuits are so much more reliable than software programs?

As Jiantao Pan points out in his excellent *paper* on software reliability, "hardware faults are mostly *physical faults*, while software faults are *design faults*, which are harder to visualize, classify, detect, and correct." This begs the question. Why can't software engineers do what hardware designers do? In other words, why can't software designers design software the same way hardware designers design hardware? (Note that, by hardware design, I mean the design of the hardware's logic). When hardware fails, it is almost always due to some physical malfunction, and almost never to a problem in the underlying logic. Since software has no physical faults and only design faults, by adopting the synchronous reactive model of hardware logic design, we can bring software reliability to at least a level on a par with that of hardware. Fortunately for software engineering, all the advantages of hardware can also be made intrinsic to software. And it can be done in a manner that is completely transparent to the programmer.

**Thinking of Everything**
When it comes to safety-critical applications such as air traffic control or avionics software systems, even a single defect is not an option since it is potentially catastrophic. Unless we can
guarantee that our programs are logically consistent and completely free of defects, the reliability problem will not go away. In other words, extremely reliable software is just not good enough. What we need is 100% reliable software. There is no getting around this fact.

Jeff Voas, a leading proponent of the 'there is no silver bullet' movement, a software-reliability consulting firm in Dulles, VA, once said that "it's the things that you never thought of that get you every time." It is true that nobody can think of everything, especially when working with algorithmic systems. However, it is also true that a signal-based, synchronous program can be put together in such a way that all internal dependencies and incompatibilities are spotted and resolved automatically, thus relieving the programmer of the responsibility to think of them all. In addition, since all conditions to which the program is designed to react are explicit, they can all be tested automatically before deployment. But even more important is that, in a signal-based synchronous system, all possible signal pathways can be discovered and tested before deployment. Guaranteed bug-free software is an essential aspect of the COSA Project and the COSA operating system. Refer to the COSA Reliability Principle for more on this topic.

The COSA software model makes it possible to automatically find design inconsistencies in a complex program based on temporal constraints. There is a simple method that will ensure that a complex software system is free of internal logical contradictions. With this method, it is possible to increase design correctness simply by increasing complexity. The consistency mechanism can find all temporal constraints in a complex program automatically, while the program is running. The application designer is given the final say as to whether or not any discovered constraint is retained.

Normally, logical consistency is inversely proportional to complexity. The COSA software model introduces the rather counterintuitive notion that higher complexity is conducive to greater consistency. The reason is that both complexity and consistency increase with the number of constraints without necessarily adding to the system's functionality. Any new functionality will be forced to be compatible with the existing constraints while adding new constraints of its own, thereby increasing design correctness and application robustness. Consequently, there is no limit to how complex our future software systems will be.

Plug-Compatible Components
Many have suggested that we should componentize computer programs in the hope of doing for software what integrated circuits did for hardware. Indeed, componentization is a giant step in the right direction but, even though the use of software components (e.g., Microsoft's ActiveX® controls, Java beans, C++ objects, etc.) in the last decade has automated much of the pain out of programming, the reliability problem is still with us. The reason should be obvious: software components are constructed with things that are utterly alien to a hardware IC designer: algorithms. Also a thoroughly tested algorithmic component may work fine in one application but fail in another. The reason is that its temporal behavior is not consistent. It varies from one environment to another. This problem does not exist in a synchronous model making it ideal as a platform for components.

Another known reason for bad software has to do with compatibility. In the brain, signal pathways are not connected willy-nilly. Connections are made according to their types. Refer, for
example, to the retinotopic mapping of the visual cortex: signals from a retinal ganglion cell ultimately reach a specific neuron in the visual cortex, all the way in the back of the brain. This is accomplished via a biochemical identification mechanism during the brain's early development. It is a way of enforcing compatibility between connected parts of the brain. We should follow nature's example and use a strict typing mechanism in our software in order to ensure compatibility between communicating objects. All message connectors should have unique message types, and all connectors should be unidirectional, i.e., they should be either male (sender) or female (receiver). This will eliminate mix-ups and ensure robust connectivity. The use of libraries of pre-built components will automate over 90% of the software development process and turn everyday users into software developers. These plug-compatible components should snap together automatically: just click, drag and drop. Thus the burden of assuring compatibility is the responsibility of the development system, not the programmer.

Some may say that typed connectors are not new and they are correct. Objects that communicate via connectors have indeed been tried before, and with very good results. However, as mentioned earlier, in a pure signal-based system, objects do not contain algorithms. Calling a function in a C++ object is not the same as sending a typed signal to a synchronous component. The only native (directly executable) algorithmic code that should exist in the entire system is a small microkernel. No new algorithmic code should be allowed since the microkernel runs everything. Furthermore, the underlying parallelism and the signaling mechanism should be implemented and enforced at the operating system level in such a way as to be completely transparent to the software designer. (Again, see the COSA Operating System for more details on this topic).

**Event Ordering Is Critical**
Consistent timing is vital to reliability but the use of algorithms plays havoc with event ordering. To ensure consistency, the prescribed scheduling of every operation or action in a software application must be maintained throughout the life of the application, regardless of the host environment. Nothing should be allowed to happen before or after its time. In a signal-based, synchronous software development environment, the enforcement of order must be deterministic in the sense that every reaction must be triggered by precise, predetermined and explicit conditions. Luckily, this is not something that developers need to be concerned with because it is a natural consequence of the system's parallelism. Note that the term 'consistent timing' does not mean that operations must be synchronized to a real time clock (although they may). It means that the prescribed logical or relative order of operations must be enforced automatically and maintained throughout the life of the system.

**Von Neumann Architecture**
The astute reader may point out that the synchronous nature of hardware cannot be truly duplicated in software because the latter is inherently sequential due to the Von Neumann architecture of our computers. This is true but, thanks to the high speed of modern processors, we can easily simulate the parallelism of integrated circuits in software. This is not new. We already simulate nature's parallelism in our artificial neural networks, cellular automata, computer spreadsheets, video games and other types of applications consisting of large numbers of entities running concurrently. The technique is simple: Use two processing buffers. While one buffer is being processed, the other buffer is filled with the instructions to be processed during the next cycle. When the first buffer is finished processing, switch them and the cycle begins.
The main idea is that, within any given processing cycle or frame interval, a single fast central processor does the work of many small parallel virtual processors residing in memory.

One may further argue that in an emulated parallel system, the algorithms are still there even if they are not visible to the developer, and that therefore, the unreliability of algorithmic software cannot be avoided. This would be true if unreliability were due to the use of a single algorithm or even a handful of them. This is neither what is observed in practice nor what is being claimed in this article. It is certainly possible to create one or more flawless algorithmic procedures. We do it all the time. The unreliability comes from the unbridled proliferation of procedures, the unpredictability of their interactions, and the lack of a surefire method with which to manage and enforce dependencies (see the blind code discussion above).

As mentioned previously, in a synchronous software system, no new algorithmic code is ever allowed. The only pure algorithm in the entire system is a small, highly optimized and thoroughly tested execution kernel which is responsible for emulating the system's parallelism. The strict prohibition against the deployment of new algorithmic code effectively guarantees that the system will remain stable.

Software ICs with a Twist
In a 1995 article titled "What if there's a Silver Bullet..." Dr. Brad Cox wrote the following:

Building applications (rack-level modules) solely with tightly-coupled technologies like subroutine libraries (block-level modules) is logically equivalent to wafer-scale integration, something that hardware engineering can barely accomplish to this day. So seven years ago, Stepstone began to play a role analogous to the silicon chip vendors, providing chip-level software components, or Software-ICs[TM], to the system-building community.

While I agree with the use of modules for software composition, I take issue with Dr. Cox's analogy, primarily because subroutine libraries have no analog in integrated circuit design. The biggest difference between hardware and conventional software is that the former operates in a synchronous, signal-based universe where timing is systematic and consistent, whereas the latter uses algorithmic procedures which result in haphazard timing.

Achieving true logical equivalence between software and hardware necessitates a signal-based, synchronous software model. In other words, software should not be radically different than hardware. Rather, it should serve as an extension to it. It should emulate the functionality of hardware by adding only what is lacking: flexibility and ease of modification. In the future, when we develop technologies for non-von Neumann computers that can sprout new physical signal pathways and new self-processing objects on the fly, the operational distinction between software and hardware will no longer be valid.

As an aside, it is my hope that the major IC manufacturers (Intel, AMD, Motorola, Texas Instruments, Sun Microsystems, etc.) will soon recognize the importance of synchronous software objects and produce highly optimized CPUs designed specifically for this sort of parallelism. This way, the entire execution kernel could be made to reside on the processor chip.
This would not only completely eliminate the need for algorithmic code in program memory but would result in unparalleled performance. See the description of the COSA Operating System Kernel for more on this.

**Failure Localization**

An algorithmic program is more like a chain, and like a chain, it is as strong as its weakest link. Break any link and the entire chain is broken. This brittleness can be somewhat alleviated by the use of multiple parallel threads. A malfunctioning thread usually does not affect the proper functioning of the other threads. Failure localization is a very effective way to increase a system's fault tolerance. But the sad reality is that, even though threaded operating systems are the norm in the software industry, our systems are still susceptible to catastrophic failures. Why? The answer is that threads do not entirely eliminate algorithmic coding. They encapsulate algorithms into concurrent programs running on the same computer. Another even more serious problem with threads is that they are, by necessity, asynchronous. Synchronous processing (in which all elementary operations have equal durations and are synchronized to a common clock) is a must for reliability.

Threads can carry a heavy price because of the performance overhead associated with context switching. Increasing the number of threads in a system so as to encapsulate and parallelize elementary operations quickly becomes unworkable. The performance hit would be tremendous. Fortunately, there is a simple parallelization technique that does away with threads altogether. As mentioned, earlier, it is commonly used in such applications as cellular automata, neural networks, and other simulation-type programs.

**Boosting Productivity**

The notion that the computer is merely a machine for the execution of instruction sequences is a conceptual disaster. The computer should be seen as a behaving system, i.e., a collection of synchronously interacting objects. The adoption of a synchronous model will improve productivity by several orders of magnitude for the following reasons:

**Visual Software Composition**

The synchronous model lends itself superbly to a graphical development environment for software composition. It is much easier to grasp the meaning of a few well-defined icons than it is to decipher dozens of keywords in a language which may not even be one's own. It takes less mental effort to follow signal activation pathways on a diagram than it is to unravel someone's obscure algorithmic code spread over multiple files. The application designer can get a better feel for the flow of things because every signal propagates from one object to another using a unidirectional pathway. A drag-and-drop visual composition environment not only automates a large part of software development, it also eliminates the usual chaos of textual environments by effectively hiding away any information that lies below the current level of abstraction. For more information, see Software Composition in COSA.

**Complementarity**

One of the greatest impediments to software productivity is the intrinsic
messiness of algorithmic software. Although the adoption of structured code and object-oriented programming in the last century was a significant improvement, one could never quite achieve a true sense of order and completeness. There is a secure satisfaction one gets from a finished puzzle in which every element fits perfectly. This sort of order is a natural consequence of what I call the principle of complementarity. Nothing brings order into chaos like complementarity. Fortunately, the synchronous model is an ideal environment for an organizational approach which is strictly based on complementarity. Indeed, complementarity is the most important of the basic principles underlying Project COSA.

**Fewer Bugs**
The above gains will be due to a marked increase in clarity and comprehensibility. But what will drastically boost productivity will be the fewer number of bugs to fix. It is common knowledge that the average programmer's development time is spent mostly in testing and debugging. The use of snap-together components (click, drag and drop) will automate a huge part of the development process while preventing and eliminating all sorts of problems associated with incompatible components. In addition, development environments will contain debugging tools that will find, correct and prevent all the internal design bugs automatically. A signal-based, synchronous environment will facilitate safe, automated software development and will open up computer programming to the lay public.

**Conclusion**

**Slaying the Werewolf**
Unreliable software is the most urgent issue facing the computer industry. Reliable software is critical to the safety, security and prosperity of the modern computerized world. Software has become too much a part of our everyday lives to be entrusted to the vagaries of an archaic and hopelessly flawed paradigm. We need a new approach based on a rock-solid foundation, an approach worthy of the twenty-first century. And we need it desperately! We simply cannot afford to continue doing business as usual. Frederick Brooks is right about one thing: there is indeed no silver bullet that can solve the reliability problem of complex algorithmic systems. But what Brooks and others fail to consider is that his arguments apply only to the complexity of algorithmic software, not to that of behaving systems in general. In other words, the werewolf is not complexity per se but algorithmic software. The bullet should be used to slay the beast once and for all, not to alleviate the symptoms of its incurable illness.

**Rotten at the Core**
In conclusion, we can solve the software reliability and productivity crisis. To do so, we must acknowledge that there is something rotten at the core of software engineering. We must understand that using the algorithm as the basis of computer programming is the last of the stumbling blocks that are preventing us from achieving an effective and safe componentization of software comparable to what has been done in hardware. It is the reason that current quality control
measures will always fail in the end. To solve the crisis, we must adopt a synchronous, signal-based software model. Only then will our software programs be guaranteed free of defects, irrespective of their complexity.

* This is not to say that algorithmic solutions are bad or that they should not be used, but that the algorithm should not be the basis of software construction. A purely algorithmic procedure is one in which communication is restricted to only two elements or statements at a time. In a non-algorithmic system, the number of elements that can communicate simultaneously is only limited by physical factors.

* A synchronous system is one in which all objects are active at the same time. This does not mean that all signals must be generated simultaneously. It means that every object reacts to its related events immediately, i.e., without delay. The end result is that the timing of reactions is deterministic.

### The Devil's Advocate

**Abstract**

I regularly get criticism from detractors who object to my arguments in favor of adopting a non-algorithmic, signal-based, synchronous software model. Essentially the "devil" is saying repeatedly, "there is no silver bullet", not because the he is right but because the he is afraid to be wrong. The following is a compiled list of objections followed by my rebuttals. I will add new items to the list as they come to my attention.

**The Devil's Objections**

**Hardware is more reliable than software because correcting flaws in hardware is very difficult and expensive, so they get it right the first time.**

Correcting flaws in mission-critical software is equally expensive. Just ask any manufacturer who has had to recall thousands of products due to a defect in the software. Ask NASA or the FAA how expensive and dangerous malfunctioning software can be. Mission and safety critical software goes through the same stringent testing as hardware. The fact remains that algorithmic software is still more prone to failure than hardware regardless of how careful the designers and testers are.

**Hardware has just as many bugs as software. Just look at the errata sheets for a new chip.**

Nobody is claiming that there are no bugs in hardware. The claim is that almost all bugs in hardware are found, corrected and documented during the testing process. Once released, an integrated circuit will almost never fail except for
physical reasons. By contrast, there are almost always hidden bugs in released software that the quality control process invariably fails to catch during testing. In addition, most hardware bugs are due to physical defects introduced during manufacturing or the result of bad physical layout.

**Hardware is more reliable than software because it is less complex.**

Not true for two reasons. First, if one compares hardware and software of roughly equal complexity, the hardware is invariably orders of magnitude more stable than the software. Second, when most people talk about hardware, they usually think of a single IC chip or function. They overlook the fact that a chip is more comparable to one or more subroutines or objects in a software application. A hardware system, such as a computer, usually consists of multiple chips working together in very complex ways. Combining any number of chips to form large systems is not known to increase their logical failure rate after release. Likewise, combining many functions on a single chip does not degrade the quality of the finished product. By contrast, combining subroutines to create larger programs is known to increase the likelihood of failure in deployed software systems.

**The brain is asynchronous, not synchronous as you claim.**

This is not supported by research in neurobiology. One of the most amazing aspects of the brain that has come to light in the last half century is the existence of synchronizing oscillations mostly in the 10 to 120 Hertz range.

**Contrary to your claims, the human brain is a very unreliable system. It continually makes mistakes, creates memories of events that never happened and often makes irrational decisions.**

Unlike our current computer systems, the brain is self-correcting. That is to say, it uses a trial and error process to modify itself. Making and correcting mistakes is what it is programmed to do. To expect a child to ride a bicycle without falling or running into obstacles is like faulting a chess program for not playing tic-tac-toe. Barring a physical failure, the brain always does what it is programmed to do, flawlessly, even if it turns out to be a mistake.

In order to survive in an uncertain and chaotic environment, the brain uses a technique known as pattern completion to fill in missing or unknown information. This mechanism makes it possible for us to understand a garbled conversation in a noisy room or recognize a partially occluded face. It also makes it possible for animals to recognize danger in the wild even when the predator is hidden from view. Certainly, the mechanism often leads to false assumptions but this must not be equated with failure on the part of the brain. This is the way it is supposed to work. Anything else would lead to extinction. As an aside, our future intelligent robots will behave in a very similar manner. The super rational, logical and unerring Mr. Spock is a modern myth.
What about the famous Pentium FDIV bug? Isn't that a case of hardware failing after release?

No. The Pentium floating-point processor did exactly what it was supposed to do, which was to fetch a value from a location in a table in on-chip memory. It just so happened that the table was wrong. This Pentium division bug is a perfect example of blaming hardware for a fault in the embedded software. For whatever reason, the quality control department had failed to test a portion of the design. The promise of the synchronous model is not to eliminate design mistakes, although it can go a long way toward that goal. The promise is this: once a design is tested to behave a certain way, it will continue to behave in the same way barring a physical failure. One must not fault a chess program for not playing tic-tac-toe.

See Also:
Why Software Is Bad And What We Can Do to Fix It

Project COSA

To Drastically Improve Software Reliability and Productivity

Abstract:

COSA is a reactive, signal-based software construction and execution environment. A COSA program is inherently concurrent and ideally suited for fine-grained parallel processing. The two main goals of Project COSA is to enable the creation of bug-free software applications and to improve software productivity by several orders of magnitude. COSA is based on the premise that unreliability is not an essential characteristic of complex software systems. The primary reason that computer programs are unreliable is the age-old practice of using the algorithm as the basis of software construction. Switch to a synchronous, signal-based model and the problem will disappear (see The Silver Bullet article for more information on this topic).

The Problem

The software industry is in trouble. Several decades of process improvements and innovations in testing procedures have failed to tame the twin beasts of unreliability and low productivity. The widely perceived severity of the problem is justified by a legitimate concern over human safety and the risk of economic loss, security breaches, liability lawsuits and other misfortunes. As the world becomes more and more dependent on computers and as the complexity of software systems continues to rise, the crisis can only get worse. It is particularly severe in vital segments of the economy such as the health and financial services, and the transportation, manufacturing, communication, power generation, and defense industries. The expensive custom systems used
by these sectors account for more than 85 percent of the software market. Their failure rate, i.e., the percentage of projects that are started but are never deployed, is reported to be over 50 percent, a staggering figure considering the high cost of software development.

**The Cause**

The main reason for the crisis is the lack of a sound software construction methodology with which to manage the high complexity of modern applications. There are those who blame engineering incompetence and lack of discipline for the problem. They are not seeing the whole picture. The truth is that software engineering is bad mainly because the development tools and operating systems are bad. The tools are bad because there is something wrong with the way we construct and run our software, something that no amount of quality assurance measures can ever cure. Software is unreliable because of the practice of using the algorithm as the basis of software construction. This is not to say that there is anything wrong with the algorithm. It just should not be the basis of software. See [The Silver Bullet](#) article for more on this.

**The Solution**

We must abandon the algorithmic model and embrace a new paradigm. Objects in the real world behave synchronously and causally. Why should software objects be any different?

<table>
<thead>
<tr>
<th>Algorithmic Model</th>
<th>COSA Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implicitly sequential</td>
<td>Explicitly sequential</td>
</tr>
<tr>
<td>Explicitly Parallel</td>
<td>Implicitly Parallel</td>
</tr>
<tr>
<td>Procedural</td>
<td>Reactive (signal-based)</td>
</tr>
<tr>
<td>Latency</td>
<td>Immediacy</td>
</tr>
<tr>
<td>Function calling</td>
<td>Message passing</td>
</tr>
<tr>
<td>Asynchronous processes</td>
<td>Synchronous processes</td>
</tr>
<tr>
<td>Monolithic</td>
<td>Compositional</td>
</tr>
<tr>
<td>Chaotic</td>
<td>Systematic</td>
</tr>
<tr>
<td>Textual</td>
<td>Graphical</td>
</tr>
<tr>
<td>Hard to predict</td>
<td>Causally-ordered/deterministic</td>
</tr>
<tr>
<td>Hand-coded</td>
<td>Automated</td>
</tr>
</tbody>
</table>

The list above shows characteristics of both models for comparison. The solution will require a strong commitment to deterministic causal principles. It can only be done by adopting a software model based on reactive concurrency. However, the required discipline and the commitment to these principles should not be the responsibility of the application developer but that of the operating system and tool vendor. Unless we can create tools and systems that automatically enforce the new model painlessly and transparently, the world will continue to pay a heavy price for unreliable software.
Note: Synchronous processing is not the same as synchronous messaging. Synchronous processing means that all elementary operations in a system have equal durations and are synchronized to a master clock. Deterministic timing is a must for reliability. The COSA model supports asynchronous messaging and signaling. It also supports fine-grain parallelism.

What Is COSA?

Currently there exist hundreds of programming languages, operating systems and development tools competing against one another, not counting custom proprietary technologies. A veritable tower of Babel. Worse, the complexity of many of the tools is often greater than that of the applications. Becoming proficient in their use often requires years of training and experience. This is a sign of the chronic immaturity of the software industry. Software engineering will not come of age until a single software construction and execution model is universally adopted. Even computer processors must eventually be redesigned to support the new software model. This is part of the motivation behind Project COSA.

COSA stands for Complementary Objects for Software Applications. It is a reactive, signal-based software construction and execution environment. The word 'Cosa' means 'thing' in several languages. It is a fitting label for a software model in which the computer is viewed, not as a machine for the execution of instruction sequences, but as a collection of synchronously interacting objects or things. An object in COSA is a simple virtual software entity that resides in computer memory, waits for a signal to perform an elementary operation, and then emits an output signal. Software creation consists of connecting elementary objects (cells) together using a graphical software composition environment. Cells can be combined into high-level, plug-compatible components and/or applications. There are no procedures, no subroutines and no compile/run cycles. Best of all, there is no programming language to learn.

Basic Principles

One of the ideas behind COSA is that software should not be radically different than hardware. Software should be an extension of hardware. It should emulate the functionality of hardware and add only what is lacking: flexibility, random data access and ease of modification. COSA is guided by the following principles:

Complementarity

First and foremost, every object or concept in COSA has a complement or opposite. Examples of complementary pairs are: sensor/effector, cause/effect, departure/arrival, start/stop, input/output, male/female, source/destination, mouse-up/mouse-down, component/constituent, etc. Complementarity is the single most powerful organizing principle in existence. Any system, regardless of complexity, can be effectively and efficiently managed by applying this principle at every level of abstraction. This can be argued on the basis of natural evidence ranging from the elementary particles and antiparticles of physics to the complementary base pairs of DNA.
Change-Driven
All COSA objects are concurrent and synchronous. That is to say, they are always active, in the sense that a neuron is always active and ready to fire upon receiving a signal. A signal-driven system is a change-driven system. This means that no action or operation can take place unless something has changed. A change (aka event) can be anything from a key press to a change in a data variable. The primary advantage of this approach is that causes and effects are clearly identified.

Temporal Consistency
Strict temporal determinacy is crucial to long-term reliability. The relative temporal order of every action must be guaranteed to always be consistent. Nothing must be allowed to happen before or after its time. COSA guarantees that a software component will have a deterministic temporal behavior during its entire life cycle regardless of the environment in which it is used. This is a natural consequence of the synchronous nature of COSA objects.

Passive and Active Objects
The old object-oriented philosophy of looking at every software object as an encapsulation of methods and data is flawed, in my opinion. Every operation (method) should be viewed as an object in its own right: it waits for a signal to perform an action and emits an output signal immediately afterward. The principle of complementarity forces us to classify software objects into active (e.g., sensors and effectors) and passive (properties, data variables) categories. Even data should be classified into complementary types (e.g., positive and negative numbers, true and false values, etc.). The right metaphor draws a clear distinction between behaving entities (active objects) and their environment or domain (passive objects).

Cells and Components
In COSA, an active object is called a cell and a cell can be either a sensor or an effector. A passive object is just a data operand, i.e., a variable property or a constant. A group of passive objects is called a domain or territory. A component is analogous to a country. It encapsulates the active objects and their territory. Passive objects can be shared among several active objects belonging to the same territory but are never shared across territories, except when passing messages. This is described in greater detail in Software Composition in COSA.

Extreme Simplicity
If it is not simple, it is wrong. Simplicity is a natural consequence of applying the principle of complementarity. Every object or concept in COSA is stripped of unnecessary complexity. No algorithmic construct (e.g., IF/THEN/ELSE, WHILE, GOSUB, GOTO, DO WHILE, DO UNTIL, etc…) is allowed.
As seen above, a COSA program or component is just a collection of active objects (sensors and effectors) on the one hand, and the environment (data or passive objects) on the other. Nothing more and nothing less. As will be seen later in the article on the COSA operating system, these objects are represented graphically with a minimum number of icons. An active object's type (sensor or effector) is immediately recognizable. This, in effect, does away with a major part of the "essential complexity" of software that Dr. Brooks alluded to in his "No Silver Bullet" paper.

**Causality**
One of the strengths of hardware logic is the clear distinction it makes between cause and effect, i.e., between events and reactions. The same distinction must exist in software. The essence of software is this: a computer program is a reactive entity that detects changes in its environment and acts on it. This makes it easy for the developer to determine which actions are taken and why. The upshot is high program comprehensibility.

**Total Vision**
Unresolved dependencies are the leading cause of software failures in complex systems: A critical property or variable is changed by one part of a program, unbeknownst to the other parts. Blind code is the bane of programmers and managers who are given the task of maintaining complex legacy algorithmic systems. Even a minor modification is likely to introduce one or more unforeseen side effects. By contrast, a COSA system has what I call 'total vision.' It is a mechanism based on complementarity that automatically resolves all data dependencies, leaving nothing to chance. The identification and resolution of dependencies must be effected both at the cell level (in-memory data) and at the component level (mass storage data). Total vision makes it possible to eliminate unwanted side effects and opens the way to the creation of extremely complex yet robust systems.

**Visual Software Construction**
COSA supports the notion that software should be composed from a few simple, visually-depicted objects as opposed to being written in a syntactic language. Developers should concentrate on application design and requirements and should not have to worry about syntax and spelling. They certainly should not have to learn to decipher keywords based on a language (usually English) that may not be their native language. Software construction should consist almost entirely of using a small number of simple graphical icons to connect objects together to form higher level components. Pre-built, plug-compatible components can be effortlessly selected from a repository and joined together to form larger applications. Just click, drag and drop. Automating a large part of programming is guaranteed to do wonders for both productivity and reliability.
Programming by Design
Software development traditionally goes through several phases such as requirement analysis, design, programming (including compiling) and testing. Project COSA proposes to combine all phases into one. Good software design under COSA means bug-free code that fulfills requirements from the start. When the design is finished, i.e., when all requirements are met, the application is ready for deployment. There is no distinction between the application development tool and the application design tool. In addition, a COSA application under development is always "running." Most additions and modifications can be tested instantly without the need to recompile and re-launch the application.

The COSA Reliability Principle
The promise of the COSA software model is as simple as it is revolutionary. A COSA program is guaranteed to be completely free of internal defects regardless of its complexity. I call it the COSA Reliability Principle or CRP. The principle is based primarily on the knowledge that a COSA program works essentially like a logic circuit. More details on the CRP can be found in the COSA operating article below.

Disadvantages

Slower Performance
The main disadvantage of COSA will be slower performance on current processors than pure assembly or compiled code. This is because the objects are interpreted. But this is not such a big problem nowadays, as the performance of processors continues to obey Moore's law. For now, the COSA model must use an interpreter but only because current processors are optimized for conventional algorithmic software. Eventually, when the COSA model becomes widely adopted by the computer software industry (hopefully the software reliability crisis will make sure of that), chip manufacturers will have to follow suit by designing their processors to embrace the new paradigm. As an added bonus, only the COSA model makes it possible to design a fine-grain multicore processors in an MIMD execution environment. See Transforming the TILE64 into a Kick-Ass Parallel Machine and How to Design a Self-Balancing Multicore Processor.

It is possible to create a COSA compiler that generates fast optimized compiled code to be run as a standalone embedded program or as an application to be run in an existing OS. Performance would be at least on a par with if not better than hand-written code. This is fine for embedded systems and as a bridge solution for legacy systems but it is inadequate for a full-blown operating system which will have to run multiple synchronous applications concurrently.

Note that the COSA execution kernel described below should be no slower than other code interpreters. FORTH is a case in point. FORTH interpreters have certainly proven their performance adequacy for all sorts of real time applications, especially in embedded and robotic systems. Dedicated FORTH processors also
exist that can execute FORTH code directly, resulting in unparalleled increases in performance. My hope is that the same thing will happen with COSA.

**Unfamiliarity**
Another disadvantage of COSA is unfamiliarity. COSA is a radically different environment than what most people (with the exception of hardware designers) are used to. Most programmers are trained to use traditional programming languages and may have a hard time switching to a development system that discards all the familiar algorithmic constructs. There is also the problem of programmer prejudice. Many wrongly feel that visual development systems can never be powerful enough to suit their needs. The other side of the coin is that non-programmers will have no problem learning COSA.

**Legacy Systems**
The ideal environment is one in which everything, from the operating system components to the applications, works according to COSA principles. Unfortunately, the world has already invested trillions of dollars and countless man-hours in legacy systems. There is a huge algorithmic infrastructure already in place, one which will not go away overnight. Regardless of how great the advantages of the new paradigm are, it will take much time and effort to displace the old systems. Fortunately, standalone COSA applications can easily be made to work within an existing OS. These applications can migrate to pure COSA environments at a later date, when these become common place.

**Advantages**

**High Dependability**
The primary advantage of COSA is the COSA Reliability Principle. Every COSA program is guaranteed to be 100% free of internal defects. Application areas that come to mind are simulations, video games, virtual reality, neural networks, etc. But safety and mission-critical applications are where COSA will really shine. Examples are: air traffic control, avionics, transportation automation, collision avoidance systems, power generation, high-traffic communication systems, real-time embedded software, discrete signal processing, medical and financial systems, multi-sensor network management software, instrumentation, robotics, factory automation, etc.

**Parallel and Distributed Systems**
Since a COSA system consists exclusively of synchronous (parallel-activated) objects, it is ideally suited for parallel/distributed computing and clustering. COSA is an ideal environment for the simulation of physical phenomena that require huge numbers of synchronous entities. If an application needs higher real-time performance than can be obtained with a single CPU, processing power can be increased simply by adding new CPUs to form a cluster and redistributing some of the components so as to lighten the overall CPU load. This can be done automatically with the help of clustering and load balancing tools. In COSA, the
parallelism of objects is implicit. Thus the software designer is not forced to use special language constructs to access or create parallel (concurrent) objects. Furthermore, the mechanism that maintains and manages object concurrency is completely transparent to the user/designer.

**High Productivity**
As everyone knows, the average programmer's time is spent mostly in debugging. Software that is bug-free from the start is COSA's main forte. This goes a long way toward improving productivity. First off, COSA eliminates all problems resulting from incompatible software components via the use of plug-compatible connectors. Second, the COSA software construction environment automatically resolves all data dependencies automatically. See the discussion on blind code for more on this topic.

There is more to rapid application development than the creation of reliable programs. One of the best aids to productivity is program comprehensibility. The complexity of an application must not get in the way of comprehension. Managing complexity is one COSA's strengths. COSA adds clarity to software development in several ways:

**Complementarity**
COSA uses a simple but powerful organizing principle, which stipulates that every object or concept must have a complement.

**Extreme Simplicity**
This is a direct result of the rigorous enforcement of the principle of complementarity.

**Extreme Information Hiding**
Components are treated as black boxes with well-defined interfaces and behaviors. Implementation details (data operands, arithmetic operations) are kept separate and hidden from organizational and behavioral structure such as cell connectivity and activation logic.

**Active and Passive**
There is a clear demarcation between active objects (cells) and passive objects (data). This is analogous to organisms and the environment.

**Sensors and Effectors**
Active objects are divided into two complementary subcategories: sensors and effectors. What an object does and when and why it does it is easy to understand.
**Sensor/Effector Associations**
Data comparison sensors are instantly and automatically associated with pertinent effectors upon creation. This way, a developer can immediately identify every effector object that can potentially affect a comparison. This is a must for program comprehension.

**Simple Graphical Interface**
Objects and components are displayed graphically through the use of a minimum number of simple graphical icons.

**One-way Signal Pathways**
Activation order (cause and effect) is always clear and explicit.

**Male and Female Connectors**
No confusion or mix-ups.

**Plug-compatible Components**
Automated and flawless software composition using connection or message IDs.

**Visual Software Composition**
Click, drag and drop. Instant depiction of control pathways. No syntax or spelling errors to worry about.

Last but not least, there are no compile cycles in a COSA development environment. As mentioned earlier, a COSA program is always running even while under development. This may not be such a big thing when developing small applications but very large and complex programs may sometimes take hours and even days to compile. Although new tools can distribute the make process among many machines running simultaneously, even an order of magnitude improvement can still mean a 1-hour compile or more in some situations. Eliminating compile cycles improves productivity not only because it saves computer time, but also because it enables an interactive environment free of distractions and interruptions.

**Synchronous Reactive Systems**

COSA belongs to a class of software systems called synchronous reactive systems. Here is a quote from an article by Axel Poigné of the German National Research Center in Information Technology:

"Synchronous programming reinterprets the basic idea of hardware design, and of discrete engineering formalisms, namely that processes run on a clock; a system reacts to each stimulus of the environment instantaneously meaning that the reaction consumes no (observable) time, or in hardware terms: every reaction terminates before the next 'clock tick'. Software design is less accustomed to this idea which, however, should be the basis of systems with hard real time"
This means that all reactions to simultaneous events must occur before the next clock tick. In COSA, the clock is virtual and a clock tick represents an execution cycle. There are a few important differences between COSA and other reactive systems, as seen below:

**Universal Model**

There is a general consensus in the reactive programming community that so-called classical computations (such as reading and writing to memory, looping and performing arithmetic and comparison operations on data) are to be handled separately from input and output processing. In a reactive language such as Esterel, the environment is defined as being external to the computer. By contrast, in COSA, all sensory events, whether external (I/O interrupts) or internal (changes in data), are treated equally. In other words, there is no process distinction between internal and external sensory phenomena in a COSA program. The same principle is applied to output processing: there is no process distinction between writing a byte to an output register and performing an arithmetic operation on a data variable. Both operations are classified as effects. The COSA model views the environment simply as a collection of data variables. Whether or not these variables are linked to conditions external to the computer is irrelevant.

This realization has far-reaching implications: it brings the reliability and verifiability benefits of the reactive model to all areas of computation, not just to input and output processing. For example, comparisons in COSA are not seen as operations to be contained in an algorithmic procedure but rather, as sensory (input) processes. Likewise, arithmetic operations are viewed as effects, i.e., a type of output or motor processing. This way, classical (i.e., algorithmic) procedures are transformed and organized into a concurrent, reactive network of interconnected sensors and effectors. This approach has three principal benefits. a) It insures that all steps in any given computation are executed in a deterministic order (concurrent or sequential), a must for verifiability and reliability; c) It provides a simple and effective mechanism to identify and resolve data dependencies; and b) It classifies objects in a simple way that enhances comprehensibility. For more information on this new reactive model, please see the description of the COSA system.

**Not Just for Mission-Critical Applications**

Most researchers in the reactive programming community (one notable exception that I am aware of, is this music-minded team at Berkeley) seem to be targeting real-time, mission-critical and/or embedded software applications exclusively. The COSA philosophy is that all software development should be based on reactive principles. In order to solve the software crisis, it is imperative that the entire software development industry switches to the synchronous reactive model. The reason is simple. The use of any software model that improves reliability and
productivity benefits all areas of software engineering, not just mission-critical systems.

**Visual Composition**
The reactive community seems to have embraced textual development tools as their favorite approach to software construction. Several reactive programming languages already exist: Esterel, LUSTRE, SIGNAL, etc. Even VHDL can be said to fall in the category of reactive programming languages, although it is meant to be a description language for hardware design. The idea seems to be that a conventional algorithmic language is necessary to describe classical computations. This is an unfortunate legacy of algorithmic computing. In order to inherit the full promise of the reactive model, all computations, including so-called classical computations, can and should be reactive in nature.

The COSA philosophy is that the use of visual composition tools is the most intuitive, efficient and safe way to describe and construct reactive programs. An application designer should concentrate on the design at hand and should not have to learn or worry about such things as syntax and spelling. In addition to facilitating program comprehension, a visual environment automatically enforces all necessary constraints in a way that is beyond the capability of text-based development tools.

These and other COSA original ideas will not only go a long way toward solving the software reliability and productivity crisis, but should make synchronous reactive programming much more appealing to the software engineering community at large, as a universal approach to parallel software construction. For more information on current reactive systems and languages, click on the links below:

- The Esterel Language
- Tick, Synchronous Reactive Systems
- Syrf Project

**See Also:**
[Why Software Is Bad And What We Can Do to Fix It](#)

---

**The COSA Operating System**

*Synchronous Reactive Parallel Software Objects*

**Abstract**
This page contains a description of the COSA operating system. COSA is an alternative software construction and execution environment designed to improve software productivity by several orders of magnitude. In addition, COSA will enable the creation of bug-free software applications of arbitrary complexity. It is based on the premise that the primary reason that computer programs are unreliable is the age-old practice of using the algorithm as the basis of software construction. Switch to a synchronous, signal-based model and the problem will disappear. Please refer to the previous articles (Silver Bullet, Project COSA) in the series for important background information.

**Introduction**

COSA is a complete operating system. It consists of two major subsystems, the execution kernel and the support system. The latter consists of common service components that are needed for a modern operating system. Note that there is no clear distinction between a system service component and a user application other than the convenience of labeling them as such. Every application is seen as an extension of the operating system. Thus, every COSA system can be custom-tailored for a given purpose. That is to say, if, for examples, the application requirements do not call for file I/O and graphics capabilities, the file and graphics components are simply left out.

**Execution Kernel**

The execution kernel is the part of COSA that runs everything. It consists of the cell processor, the cells proper, and the data objects. The term "cell" is borrowed directly from neurobiology because biological neurons are similar to the elementary processors used in COSA, as explained below. Components are high-level objects which are made of cells and data objects or other components. Components are not considered part of the execution kernel. They are explained in detail in the Software Composition article below. The following is a short list of the items that comprise the execution kernel.

**Cell Processor**

The cell processor is an elementary object processor. It runs everything in COSA including applications and services.

**Data Objects**

Data objects are also called passive objects or properties because they do not do anything of themselves. They are operated on by cells (active objects). A data object can be either a variable or a constant, or a collection of properties.

**Cells**

Cells are active synchronous objects that reside in computer memory. A cell communicates with other cells via signals. There are two types of cells: sensors and effectors. Sensors detect changes or patterns of changes and effectors execute changes. Together with data (passive objects) cells comprise the basic building blocks of every COSA application.
Synapses
A synapse is part of a cell. It is a small data structure which is used to connect one cell to another.

Sensors
A sensor is a type of cell that detects change. Sensors detect changes either in data or in the external environment.

Logic Detectors
A logic detector is a special sensor that detects a logical combination of events.

Sequence Detectors
A sequence detector is a special sensor that detects a pattern of events (changes) occurring over time.

Effectors
An effector is a type of cell that operates on data variables and/or the external environment.

The Cell Processor
The cell processor is a highly optimized interpreter. It is the part of the COSA operating system that handles the reactive logic for the entire system. The processor’s function is to emulate parallelism by performing an operation for every cell which is in need of processing. It should be viewed as a necessary evil. It is needed only because software objects in a Von Neumann machine cannot process themselves. The cell processor is designed to be completely transparent to the application developer.

In COSA, every computation is considered to be a form of either input or output processing. For example, detecting a change in a data variable (a comparison operation) is just as much a sensory event as a key-press or a button-down signal. Likewise, incrementing a counter variable is no less a motor action than sending a character to the printer. The following elaborates on various aspects of the cell processor.

Single Execution Thread
An ideal COSA operating system would have no algorithmic processes whatsoever as all objects would be self-processing and would communicate via direct connection pathways, not unlike biological neurons. Unfortunately, the architecture of Von Neumann computers is such that a software system must have at least one algorithmic thread (a set of CPU instructions). As a result, communication pathways between objects are virtual.

In COSA, there is a single loop thread called the execution thread which runs the entire system, including applications and services. The cell processor is essentially a virtual machine similar to a FORTH interpreter or the Java™ virtual machine. It is designed to be completely transparent to the software designer. It is
the only directly executable code in the entire system. No new executable code is allowed, not even for traditional system services such as file I/O, memory management, device drivers, etc.

Software construction is achieved by connecting elementary cells together using simple signal connectors called synapses. Cells can be combined into larger modules or plug-compatible components. This is mostly for the benefit of the application designer because the cell processor sees only the cells, their synapses and assigned data operands, if any.

**Exceptions to the Single Thread Rule**

There are a few exceptions to the single thread rule. Every operating system must be able to handle hardware interrupts in real time. Interrupts are, in essence, sensory events indicating that something important has happened which needs immediate handling. Examples are mouse, keyboard, network card or hard drive events. COSA uses small interrupt service routines to modify relevant data/message structures and mark appropriate sensors for updating by the cell processor at the earliest opportunity. Neither applications nor services have permission to access interrupts directly. Indeed they cannot do so since no new native (directly executable) code is allowed in COSA. All possible interrupt eventualities must be handled by the operating system and, if the need arises, corresponding sensors are provided for use by applications and services. This eliminates a lot of headaches, especially in safety-critical domains.

Another exception has to do with copying memory buffers. Sometimes, it is imperative that such tasks be executed as rapidly as possible. It would be best to delegate them to a dedicated DMA or graphics hardware chip. However, this may not always be possible. The alternative is to use a super-optimized assembly language thread. To the software designer, a buffer-copy component would be no different than any other COSA service component. It would have an input connector for service request messages and an output connector for acknowledging when a task is completed. (See the Software Composition in COSA for more information on components and connectors).

**Two Update Lists (Buffers)**

The job of the cell processor is to update all cells that need updating at every tick of the master clock. This is done with the help of two update lists (cell or instruction buffers), one for input processing and the other for output processing. The lists contain pointers to all objects that need updating at any given time. As one list is processed, the other is filled. The reason for using two lists instead of one is to prevent signal racing conditions that would otherwise arise.

During every cycle, the lists are processed one after the other starting with the input list. During input processing, the cell processor performs a primitive operation on behalf of every cell in the input list according to the cell’s type. The cell is stamped with an integer value representing the time of activation (I shall
explain later why this is important). After each operation is performed, the cell is placed in the output list for subsequent output processing. The input list is emptied after processing is completed. Output processing is much simpler. It consists of placing the destination targets of every cell currently in the output list into the input list. After completion, the output list is emptied, the master clock is incremented and the cycle begins anew. Of course, in order for any processing to occur, cells must first be placed in the input list. This normally happens when an application or component is loaded into memory and started or when a sensor detects a change in the hardware.

**COSA-Optimized Processors**

Unfortunately for the COSA software model, current microprocessors are optimized for conventional algorithmic software. As I mentioned elsewhere on this site, this is an old tradition that has its roots in Charles Babbage's and Lady Lovelace's ideas for the analytical engine which was built more than one hundred and sixty-two years ago out of metal gears and rotating shafts! The use of the algorithm as the basis of computing may have been a necessity in the early days of the modern computer era when clock speeds were measured in kilohertz and computer memory was at a premium, but with the advent of megahertz and gigahertz CPUs, there is no longer any excuse for processor manufacturers to continue doing business as usual.

A specially designed COSA-optimized processor could maintain both update lists in the processor's on-chip cache, obviating the need to access main memory to read and write to them, thus saving processing time. And since COSA cells are concurrent objects, they can be dispatched to available execution cores within a single multicore processor for simultaneous processing. Vector processing techniques can be used extensively to increase performance. It should even be possible to keep copies of the most often used cells in cache memory in order to further improve performance. In addition, the processor should have intimate knowledge of every cell type. As with FORTH processors, this would eliminate the need for a software interpreter or microkernel. The CPU itself would be the cell processor. These optimizations, among others, could bring performance to a level on a par with or better than that of RISC processors. See also Transforming the Tile64 into a Kick-Ass Parallel Machine.

One of the advantages of the COSA model is that it is a change-based computing environment. As a result, a comparison test is made only once, i.e., when a related change occurs. Contrast this with algorithmic software where unnecessary comparison operations are often performed every time a subroutine is called. Sooner or later, the computer industry will come to its senses and recognize the soundness of the synchronous software model. The traditional algorithmic microprocessor will then join the slide rule and vacuum-tube computer as mere historical curiosities.
Signals
In a COSA program, there are obviously no tangible signals--comparable to the changes of potential in a digital circuit--that travel from one cell to another. There are no physical pathways between cells. Signals are virtual and signal "travel" time is always one cycle. The mere placement of a cell in an update list means that it has either received or is on the verge of transmitting a signal. This is a plus for software because the signal racing conditions that commonly occur in hardware are non-existent. It should be noted that signals do not carry any information in COSA. Particularly, a signal does not have a Boolean value. A signal is a simple temporal marker that indicates that some phenomenon just occurred. The nature or origin of the phenomenon cannot be determined by examining the signal.

There is a difference between a signal and a message in COSA. As explained above, a signal is an abstract temporal marker. With regard to effector cells (see below), it always marks either the end of an elementary operation or the beginning of another or both. With regard to sensor cells it marks either a positive or negative change in the environment. Signals are always processed synchronously (i.e., immediately) by the cell processor.

A message, on the other hand, is a data structure that can be placed in a queue (FIFO) or a stack (LIFO) and assigned a priority level. Messages are shared data structures between two or more components and are associated with a set of male and female connectors. Message queuing and stacking are not handled directly by the execution kernel. That is the job of the message server, one of the many service components in COSA (this will be explained further in a future article).

Virtual Master Clock
One of the most important aspects of using a master clock is that every elementary operation lasts exactly one execution cycle. An uncompromising consistency in the relative order of signals and operations must be maintained at all costs. Logically speaking, it does not matter whether the durations of the clock cycles are not equal. As long as the master clock (really an integer count) is incremented once every cycle and as long as it serves as the reference clock for timing purposes, logical order consistency is maintained. This is because coincidence and sequence detection are based on the master clock, not on a real-time clock. Thus all concurrent operations execute synchronously between heartbeats or clock ticks. Because of this deterministic concurrency, COSA can be said to belong to a class of software systems known as synchronous reactive systems (more).

The temporal deterministic nature of cell activation is maintained even though the heart rate of a COSA system is non-deterministic relative to real time: each cycle lasts only as long as necessary. This is not entirely unlike the digital processor design approach known as 'clockless computing.' There is no need to waste time waiting for an external clock tick if one does not have to. This does not mean that
a COSA system cannot react in real-time. It can. In fact, the software designer will have ample flexibility to tweak message priorities to handle the most demanding environments. The important thing is that the temporal integrity of cell activations is maintained relative to the virtual clock.

**Real Time Processing**
Sometimes (e.g., simulation systems, parallel computing, etc.) it is necessary to synchronize a COSA program with an external clock. The best way to achieve this is to use the clock ticks emitted by one of the system's interrupt-driven sensors as reference signals. In such cases, all operations must execute within the fixed interval allotted between real-time ticks. In the event that the execution time exceeds the clock interval, the application designer has several options: a) use a slower real-time clock; b) use a faster CPU; c) split long operations into shorter ones; d) lower the priority of some message servers; or e) lighten CPU load by adding more CPUs to the system and redistributing the load.

**Time Slicing**
There is no need for multithreaded applications in COSA. The performance overhead and latency problems associated with context switching is nonexistent. In COSA, every cell is its own tiny task or process, so to speak. This does not mean that the cells are always running. Since the system is driven by change, only the cells that are directly affected by events need to be processed during any given cycle. This makes for a very efficient approach to concurrency because only a relatively small percentage of the cells in a system are active at any one time. As a result, a single-CPU COSA operating system can easily support tens of thousands and even millions of concurrent cells.

Priority processing is controlled at the message server level, not at the kernel level. The kernel never processes messages directly. High-priority messages are immediately placed at the head of their queues. In addition, all message servers automatically stop serving low-priority messages while any high-priority message is still in a queue. This technique ensures that time-critical tasks always get a disproportionate share of processor runtime. Please refer to the Software Composition article for more details on this topic.

**Verification**
As mentioned above, one of the things that distinguishes concurrent COSA objects from concurrent algorithms is that all COSA objects perform their assigned elementary operations within one execution cycle. This uniformity of execution lends itself to the creation of verification tools and techniques for both static and dynamic analysis. For example, it is easy to determine at run time whether or not a given data operand is being accessed concurrently for modification and/or reading by multiple effectors and/or sensors. It is also possible to determine statically--by counting and analyzing convergent decision pathways--whether the potential exists for conflicting data access. In addition, since COSA program is a condition-driven behaving system and since all
conditions are explicit in the design, it is easy to create tools that automatically
test every condition before deployment. This is part of the mechanism that
guarantees the reliability of COSA programs.

COSA Cells

A COSA cell is a primitive concurrent object. Cells communicate with one another via simple
connectors called synapses, to borrow a term from neurobiology. The function of cells is to
provide a general purpose set of basic behavioral objects which can be combined in various ways
to produce high-level components and/or applications. There is no computational problem that
can be solved with a general purpose algorithmic language that cannot also be solved with a
combination of COSA cells.

The number of input synapses a cell may have depends on its type. The number of output
synapses is unlimited. Comparison sensors, for example, do not have input synapses. Internally,
a cell is just a data structure that contains the cell's type, a list of synapses and pointers to data
operands, if any. When a cell is processed by the execution thread during input processing, an
elementary operation is performed on its behalf according to the cell's type. After processing, the
cell immediately emits an output signal to any destination cell it may be connected to.

COSA cells are divided into two complementary categories: sensors and effectors. The latter
operate on data while the former detect changes or patterns of changes. What follows is a
discussion of various topics related to cells and their operations.

Synapses

A synapse is a simple data structure that serves as a connector between two cells.
It contains two addresses, one for the source cell and another for the destination
cell. A synapse is an integral part of its source cell. It is used by the cell processor
to effect communication between cells. When two cells are connected, they share
a synapse together. Note that, even though cells are graphically depicted as being
connected via signal pathways, internally however, there are no pathways. There
are only the cells and their synapses. Still, it is beneficial to think of a source and
a destination cell as communicating over a unidirectional pathway. The pathway
is considered an essential part of the source cell, similar to the axon of a neuron.

All synapses maintain a strength variable at least during the development phase.
The variable is incremented every time a signal arrives at the synapse. Synaptic
strength is an indication of usage and/or experience. It can serve as a powerful
debugging aid. In case of failure, the culprit is almost invariably a young or rarely
exercised synapse. From the point of view of reliability, it is important to verify
that every signal pathway has seen activity at least once. Since all signals must
pass through synapses, the use of synaptic strengths can be used to ensure total
coverage of all decision pathways during testing. This is a tremendous asset in
failure diagnosis (blame assignment).
Sensors
Sensors have no input synapses (except Logic and Sequence detectors) but they can have as many output synapses as necessary. Their function is to detect specific changes or state transitions, either in the computer's hardware or in data variables in memory. When a change is detected, a sensor immediately sends a signal to all its destination cells. A sensor should be thought as being always active in the sense that it is always ready to fire upon detection of a change. Every sensor in COSA has a complement or opposite. The following table lists several examples of sensors and their complements:

<table>
<thead>
<tr>
<th>mouse-button-up</th>
<th>mouse-button-down</th>
</tr>
</thead>
<tbody>
<tr>
<td>key-up</td>
<td>key-down</td>
</tr>
<tr>
<td>greater than</td>
<td>not greater than</td>
</tr>
<tr>
<td>less-than</td>
<td>not-less-than</td>
</tr>
<tr>
<td>equal</td>
<td>not-equal</td>
</tr>
<tr>
<td>bit-set</td>
<td>bit-clear</td>
</tr>
</tbody>
</table>

Comparison Sensors
Comparison sensors--also called data sensors--must be associated with one or more effectors (see the discussion on sensor/effector associations below). Their function is to detect specific types of changes in their assigned data. As mentioned earlier, a signal-based system is a change-driven system.

As an example of how this works, let us say a given comparison sensor's job is to detect equality between two variables A and B. If A and B are equal both before and after an operation on either A or B, the sensor will do nothing. There has to be a change from not-equal to equal in order for the sensor to fire. The same change requirement applies to all sensors. COSA does not draw a process distinction between external (interrupts) and internal (changes in data) sensory phenomena.

Note: A cell's output synapses are always depicted as small red circles in order to distinguish them from input synapses that can be either white or black.

The Principle of Sensor Coordination
There is an implied logic to sensory signals, one that is assumed a priori. The logic is so obvious and simple as to be easily overlooked. Sensors react to specific phenomena or changes in the environment. The offset of a given phenomenon is assumed to follow the onset of the same phenomenon. This is what I call the principle of sensor coordination or PSC. It can be stated thus:
No sensed phenomenon can start if it has already started or stop if it is already stopped.

As an example, if the intensity of a light source increases to a given level, it must first go back below that level in order to increase to that level again. The logical order of events is implicit in all sensory systems. That is to say, it is imposed externally (in the environment) and no special sensory mechanism is needed to enforce it. This may seem trivial but its significance will become clear when I discuss motor coordination and effectors below. As the principle of complementarity would have it, effector coordination is the exact mirror opposite of sensor coordination. That is, effector logic is the complementary opposite of sensory logic. I like to say that effecting is sensing in reverse.

It goes without saying that a complementary pair of sensors (e.g., button-up and button-down) should never fire simultaneously. If they do, something is obviously malfunctioning. An important corollary of the PSC is that positive and negative sensors always alternate. For example, a mouse-button-up signal cannot be followed by another mouse-button-up signal without an intervening mouse-button-down signal. The PSC can be used as a debugging aid during development and/or as part of a malfunction alert mechanism during normal operation.

Effectors
An effector is roughly analogous to a simple code statement in a conventional program. Usually, it performs a single operation on data such as \((A = B + 1)\) or \((A = B \times C)\). An operation is any effect that modifies a passive object (a variable). Effectors operate on assigned data items using either direct, indirect or indexed addressing. Data types (integer, floating point, etc.) are specified at creation time.

![Effectors Diagram](image-url)

**Arithmetic Effectors**
Effectors are self-activating cells, not unlike the tonically active neurons found in the brain's motor system. What this means is that, once triggered, they will repeatedly execute their operation for a prescribed number of cycles. The left effector in the figure is preset to repeat an addition 10 times while the right effector is preset to perform 40 multiplications. The number of operations may range anywhere from 1 (for one-shot effectors) up to an indefinitely high value. For this reason, all effectors, except one-shot effectors, have two input synapses, one for starting and the other for stopping the activation.

**Hybrid Effectors**
In keeping with the Principle of Complementarity, an effector is defined as the opposite of a sensor. The complementarity can be seen in the following table:
It often turns out that one or more objects need to be notified whenever a given operation is performed. Normally, this calls for the use of a complementary activation sensor, one for every effector. However, rather than having two separate cells, it seems much more practical to combine the function of effector and sensor into a single hybrid cell, as seen in the figure below.

On receipt of a start signal, a hybrid effector performs its assigned operation and then emits an outgoing signal immediately afterward. When creating a new effector, it is up to the software developer to choose between either the normal or the hybrid type.

**The Principle of Motor Coordination**

An effector may have any number of start and stop synapses. If an effector does not receive a stop command signal after initial activation, it will repeat its operation until it has run its course, at which time it will emit an output signal to indicate that it has finished its task. There is a strict rule that governs the manner in which start and stop command signals may arrive at an effector. I call it the Principle of Motor Coordination or PMC. The PMC is used to detect command timing conflicts as they happen.

No action can be started if it has already started, or stopped if it is already stopped.

In other words, a start signal must not follow another start signal and a stop signal must not follow another stop signal. The PMC further stipulates that an effector must not receive more than one signal at a time, regardless of type. The reason is that an effector must not be invoked for two different purposes simultaneously. In other words, one cannot serve more than one master at the same time. Bad timing invariably results in one or more signal arriving out of turn, which can lead to failures. During development or even during normal operation in mission-critical environments, the environment can be set to trigger an alert whenever the PMC is violated. In addition, it is possible to create analysis tools to determine whether or not the PMC can potentially be violated as a result of one or more combination of
events. This can be done by analyzing all relevant signal pathways, event conditions and cells leading to a particular effector.

**Effector-Sensor Association (ESA)**

In a truly parallel system (such as an electronic circuit), every cell is its own processor. Since a hardware sensor is always active, it continually keeps watch on whatever change event it is designed to detect and fires upon detection. In a software system, however, sensors must be explicitly updated. Updating every sensor at every tick of the clock would be prohibitively time-consuming in anything but the simplest of applications. The reason is that a single main processor must do the work of many small processors.

But all is not lost. There is an easy and efficient technique for getting around the bottleneck. I call it dynamic pairing or coupling. It suffices to associate one or more comparison sensors with one's chosen effector (see figure above) and the cell processor does the rest. The dotted line shown in the figure indicates that the 100+ effector is associated with the != comparison sensor. In other words, the sensor does a comparison every time the effector performs an addition. This is taken care of automatically by the cell processor. The sensor shown in the example sends a stop signal to the very effector with which it is associated. This way, as soon as the comparison is satisfied, the sensor terminates the iteration. As will be seen in the [Software Composition](#) article below, this arrangement can be used to implement simple traditional loops.

Note that, even though, underneath, the sensor does a comparison test immediately after the operation, both the operation and the comparison should be thought of as happening simultaneously, i.e., within one execution cycle. The idea is that the sensor immediately reacts to the change, as it happens. Note also that an indefinite number of comparison sensors may be associated with a given effector. This way, multiple related comparisons can be done every time an action is taken. Conversely, a given comparison sensor may be associated with more than one effector, that is, with any effector whose actions may potentially affect the sensor.

I originally conceived of the concept of dynamic pairing as a way to get around the problem of updating every sensor in a program at every tick of the master
clock. I soon realized that the development system can be designed so that the programmer can be relieved of the burden of finding and creating associations by hand. The system can take care of it automatically. The result is that all data dependencies are resolved, thereby completely eliminating blind code!

**Message Effectors (ME)**
A message effector is a special hybrid effector that is used to effect message communication between components. An ME is an integral part of every message connector and is the only cell that is shared by two components: a sender and a receiver.

![Message Effector Diagram]

The sole restriction is that only the sender can start a ME and only the receiver can stop it. Remember that a hybrid cell emits a signal immediately at the end of its operation. The signal is interpreted by the sender component to mean that the message has been received and that the receiver is ready to receive another. Note that the effector pictured above stops automatically after 10 cycles. This way, the sender component can send a new message automatically every 10 cycles. It is up to the component designer to choose an appropriate interval.

**Granularity**
It is important to pick the right granularity level for basic effector operations. It must be neither too high nor too low. For example, a typical addition operation like \( A = B + C \) is performed in several steps by the CPU. First B is fetched from memory and then C is fetched. A binary addition is then performed and the result placed in a temporary register. Finally, the result is copied into A. Should each step be considered a separate action to be performed by a separate cell? The answer is no.

As mentioned earlier, a signal-based program is driven by change. The only relevant changes are changes in the data operands that reside in memory. Whichever method is used by the processor to effect a change is irrelevant to the program because it belongs to a level of abstraction that is below that of the program's reactive logic. In the addition example, the only operand that may have changed after the operation is the variable A. The changing of A is the only event that has the potential of affecting the behavior of the program. So the entire operation should be considered a single effect that takes place during a single execution cycle.

**Atomicity**
When an application designer creates an effector in COSA, he or she is presented
with a choice of operations. For example one can choose between simple assignment (e.g., \( A = B \)), and complex assignment (\( A = B + C \)). Once an operation is chosen, one is given a choice of addressing modes for each variable, direct, indirect, indexed, etc. Here are some examples of possible effector operations:

1. \( A = B; \)
2. \( A = B + C; \)
3. \( A[x] + B[y]; \)
4. \( A = A / 3.1416; \)
5. \( A = B[x][y][z]; \)

Note that every operation assigns a value to a target variable. This is the general form of all primitive COSA operations: one assignment and one arithmetic operation, if any. This is what is meant by an atomic operation: it cannot be divided any further while remaining within the application's level of abstraction.

**The Automatic Elimination of Blind Code**

A further advantage of ESA is that it allows the creation of tools that automatically identify and correct weak or missing data dependencies in an application. For example, let us say we create a comparison sensor to detect when variable \( A \) changes to a value greater than 0. As explained earlier, sensor \( A \) must be associated with one or more existing effectors. This could be left to the discretion of the developer but why leave anything to chance if it can be handled by the development system? At the moment of creation, the system should immediately find every effector that can potentially change variable \( A \) to a positive non-zero value and associate it with our sensor. This can be done automatically without any intervention from the programmer.

In sum, the system should enforce associations in such a way that every comparison sensor is associated with every effector that may potentially affect the comparison. This way, new additions (sensors and/or effectors) to an existing program will not introduce hidden side effects which are potentially catastrophic. The end result is extremely robust applications.

**The COSA Reliability Principle (CRP)**

Perhaps the most revolutionary consequence of the automatic elimination of blind code, as seen above, is what I call the COSA Reliability Principle or CRP. It can be stated thus:

**All COSA programs are guaranteed to be free of internal defects regardless of their complexity.**

Here, complexity is defined simply as the number of connections or synapses in the program. Why is the CRP true? The reason is simple. A COSA program obeys the same reactive principles as a logic circuit. If a logic circuit can be guaranteed to be free of defects, so can a COSA program. As an example, let us say a programmer creates a simple thermostat-based program that sends a signal \( A \) whenever it detects that integer variable \( T \) changes to a value greater than 70 and a signal \( B \) when \( T \) changes to a value less than 60. Does this mean that the programmer
has to test every possible value of \( T \) in order to prove that the program is correct? Of course not. The programmer needs to test only the prescribed conditions. This is true regardless of the complexity of the program.

Note that a condition is not a state but a state transition. Since all signal activation pathways and all conditions in a COSA program are explicit, circuit analyzers can be created to exhaustively test every possible eventuality (i.e., every condition) before it happens. Thus, all potential conflicts can be resolved long before deployment. As I wrote in the Silver Bullet article, the claim by Frederick P. Brooks and others that unreliability comes from the difficulty of enumerating and/or understanding all the possible states of a program is not a valid claim. The truth is that only the conditions (state changes) for which the program is designed to react to must be tested. All other conditions are simply ignored because they are irrelevant.

**Logic Detectors**

A logic detector is a special sensor that detects a logical combination of events. Logic detectors are not to be confused with logic gates. Whereas a logic gate operates on Boolean states, a logic sensor operates on transient signals. Like all sensors, logic detectors come in complementary pairs, positive or negative. A positive logic detector fires if it receives a combination of input signals that satisfy its function. Conversely, a negative detector fires if it receives a combination of signals that do not satisfy its function. There are three types of logic detectors in COSA: AND, OR, and XOR.

Note: The labels "AND", "OR", and "XOR" are used below only for their clear correspondence with the language (English) of the accompanying text. In an actual COSA development environment, they would be replaced with the more traditional and universal Boolean logic symbols.

**AND**

An AND detector is used to detect concurrency. It fires if all its input synapses fire at the same time.

The AND detector depicted above has three positive synapses (white) and one negative synapse (black). The cell is updated only when a signal arrives at a positive synapse. If the three positive signals arrive simultaneously, the cell fires. But if the negative signal arrives at the same time as the others, the cell will not fire. The principle of sensory coordination (PSC) is strictly enforced during development or even during normal operation (mission-critical environments). This means that the system will not allow the developer to create an AND cell.
that receive signals from complementary sensors unless, of course, the synapses are opposites.

**OR**
An OR sensor fires whenever a signal arrives at one or more of its positive inputs.

The OR cell shown above has three positive synapses and one negative synapse. It will not fire if a negative signal arrives alone or at the same time as a positive signal.

**XOR**
An XOR detector fires if it receives a single signal. It will not fire if it receives two or more signals simultaneously. Unlike Boolean XOR operators, a COSA XOR detector can have an unlimited number of input synapses.

Note that the XOR detector shown above has a negative synapse. Negative synapses are used only as a way to exclude other signals. They cannot cause the cell to fire.

**Sequence Detectors**

There are times when the temporal order of signals is important for decision making. This is the function of the sequence detector. All sequence detectors have a special input synapse that is called the master or reference synapse. The other synapses are called slaves. A sequence detector is updated only when a signal arrives at the master input. In order for a sequence detector to fire, its slave signals must arrive at their prescribed times relative to the master signal. The designer must specify whether the prescribed setting for each input connection represents a precise temporal interval relative to the master signal or just a relative order of arrival or rank. That is to say, there are two types of sequence detectors, time-based and rank-based.

**Time-Based Sequence Detectors**
In the figure below, the negative integer next to a slave is the precise timing value for the connection. It indicates the temporal position of the slave relative to the arrival time of the master. As with AND sensors, the system will enforce the principle of sensory coordination (PSC). This means that, if two slaves receive signals from complementary sensors, they cannot have equal temporal settings.

Rank-Based Sequence Detectors
If a sequence detector is rank-based, a slave setting represents the order of arrival relative to the other connections. In this case, if two slaves have equal settings, it does not mean that they must arrive concurrently. It only means that they must precede a connection with a higher setting or come after one with a lesser setting. Rank-based detectors are perfect for the detection of key activation sequences. A good example is the Ctrl-Alt-Delete pattern used in the MS Windows™ operating system:

As seen in the above figure, for the detector to fire, the Ctrl-Down and the Alt-Down signals must arrive before the Del-Down signal. But the Alt key and the Ctrl key must not be released in between. The order in which the Alt and Ctrl keys are pressed is unimportant.

Temporal Order Invariance
All sequence detectors come in complementary pairs, positive and negative. A negative detector is just like its positive sibling, except that if fires if its prescribed temporal conditions are not met when the master signal arrives. It is
sometimes a good idea to test for both eventualities. A good COSA-compliant design tool must be able to automatically create negative and positive detectors at the click of a mouse button.

In this light, it is easy to see how a pattern detector can be used as an alarm mechanism to enforce timing constraints. For example, a software designer may have reasons to expect certain events to always occur in a given temporal order called an invariant order. Conversely, there may be reasons to expect certain events to never occur in a particular temporal order. In such cases, a temporal cell can be used to sound an alarm or take an appropriate action whenever the expectation is violated.

Many catastrophic failures in software have to do with one or more violated assumptions about event timing. Adding as many error detectors as possible is a good way to detect problems that would otherwise go unnoticed until too late.

**Timer Cells**
A timer cell emits a signal a specified number of cycles after receiving a start signal. Like an effector, a timer-cell can have any number of start and stop input synapses. The arrival of a stop signal inhibits the cell's internal counter.

![Timer Cell](image)

By default, a timer stops after it times out. However, the designer has the option of specifying that the timer resets itself automatically so as to emit a signal at regular intervals.

**Real Time Watchdogs**
The timer cell described above is called a virtual timer because its internal counter is synchronized to the virtual master clock. There is another type of timer cell called a 'real-time timer' which is synchronized to an external clock. It can be used, as part of a watchdog to enforce real time constraints. Sometimes, especially in embedded applications, it is imperative that a computation finishes within a predetermined interval. A watchdog cell can easily determine whether or not a given signal A arrived before a timer signal B and alert the user if necessary.

**COSA Innovations**
Reactive software is not a new idea but many of the concepts and principles explained here are new to reactive computing. False modesty aside, I consider several of them to be genuine
advances because of their value to software reliability and productivity. I list some of the innovations below, not necessarily in order of importance:

- The principle of complementarity (PC).
- The principle of motor coordination (PMC).
- The principle of sensor coordination (PSC).
- The concept of effector-sensor association (ESA).
- The automatic resolution of data dependencies at the instruction level.
- The separation of implementation details from organizational structure.
- Self-activated effectors.
- Comparisons as sensory processes.
- No process distinction between internal and external sensory events.
- No process distinction between internal and external effects.
- Logical consistency via deterministic timing.

See Also:
How to Solve the Parallel Programming Crisis

Software Composition in COSA

Programming for the Masses

Abstract

This page describes the COSA software composition environment. COSA is an alternative software construction and execution system designed to improve reliability and productivity by several orders of magnitude. COSA is based on the premise that the primary reason that computer programs are unreliable is the age-old practice of using the algorithm as the basis of software construction. Switch to a synchronous, signal-based model and the problem will disappear. Please refer to the previous articles (The Silver Bullet, Project COSA and The COSA Operating System) in the series for important background information.

Note: This page is still under construction. The information that follows is not complete and may not accurately reflect the COSA model. Please read COSA: A New Kind of Programming and The COSA Control Hierarchy for the latest in COSA software composition.

Visual Software Construction

The COSA software construction environment is designed to make software development as painless and easy to understand as possible. There is no cryptic language to learn and no syntax to worry about. A developer can quickly put an application together by dragging in a few objects into the work area and connecting them together. Once the structural design is in place, the
designer can then go in and specify the underlying details. Visual software tools have been around for some time but they usually come with some sort of algorithmic scripting language and that is their main flaw. Another major flaw is the proliferation of icons. In COSA, there are only seven icons: cells (effectors and sensors), synapses, components, connectors, data and signal pathways.

One of the nicer things about a component-based visual development environment is that it affords the software composer with a quick and easy way to identify and navigate through dependencies. If two objects are connected, they obviously share a dependency. Double click on an object and instantly access its specifications. Double click on a connector to modify its message or type. Pre-built components can be stored in a keyword-searchable and browsable repository or can be downloaded from the web. Drop a component into an application under development and, whenever possible, it will automatically and flawlessly connect itself to other compatible components.

Note that COSA does not prohibit the use of sequential steps to solve any problem for which one would normally use an algorithm. The main difference is that COSA is 100% change-driven. Comparison operators thus become change sensors. That is, they are self-activating as opposed to being explicitly invoked, as is the case in conventional software. A COSA development system automatically couples every sensor (comparison operators) with any effector (operation) that may potentially affect the sensor. The result is that blind code is completely eliminated, thus enabling the creation of extremely robust software. The whole thing is, of course, completely transparent to the developer.

**Components**

A component is a container for cells, connectors, data, or other components. It is important to understand that components exist primarily for the benefit of the software designer and the end user. Their main function is to serve as an organizing mechanism for saving and loading, and for grouping cells together into plug-compatible ensembles. The COSA execution kernel does not handle components and connectors directly. Indeed, the kernel only “sees” cells, their synapses and their data. Nothing else. Internally, a component is a collection of pointers to cells, connectors, synapses, data and/or other components.
The left figure above shows a component with three female and three male connectors. Double clicking on a component opens it up to reveal its contents. The picture on the right shows a component that has been clicked open to reveal several other components. In this case, the designer chose eight internal connectors to be externally accessible. Note that three of the externally accessible connectors are grouped into a single male multiconnector on the left side of the component. The actual layout of objects inside a component enclosure is arbitrary and is left to the discretion of the designer. The naming labels are omitted in the figures due to the reduced size. User-created labels (such as ‘Save Document’, ‘count+++’, ‘A = B + C’ or ‘width != 60’) make it easy to figure out what is actually going on. The developer has the option of showing or hiding the labels. In the case of elementary cells, labels can be automatically created by the development tool.

**Two Types of Components**

There are two types of components, low-level and high-level. A low-level component contains one or more elementary cells, at least one connector (for input) and associated data. A high-level component consists of one or more other components. Cells are not allowed to exist at the same containment level as components. A component can be thought of as an inheritance mechanism. Drop a component into another and the target component automatically inherits the functionality of the other.

The figures below show examples of what a designer may see after double clicking on a component icon. The left figure is a high-level component and the right figure is low-level component. The color scheme is as follows: all components are painted dark green; effectors are light green and sensors are yellow. The small red circles are male connectors or synapses. Female synapses can be either white (positive) or black (negative).

**The "While Loop" Component**
The "while loop" component (above right) is a low-level component because it contains only cells. The one-shot assignment effector (1=) initializes the loop upon receipt of a start signal and then activates the two “loop” effectors (100+ and 100=). The effectors repeatedly increments an internal counter variable while adding (the 100+ cell) and storing data (the 100= cell) in memory until either the not-equal (!=) comparison sensor is satisfied or the effectors have run their course. The dashed-line bracket joining the 100+ and the != cells indicates a sensor-effector association. When the comparison is satisfied, the comparison sensor (!=) outputs a signal that stops both effectors simultaneously. It also sends a signal to the OR detector to indicate that the loop has terminated. The cell fires if it receives a signal from either the comparison sensor or the 100= effector. This means that there are two ways in which the loop can terminate: either the comparison is satisfied or the effector has run its course. The equivalent algorithmic code might look something like this in C++:

```c++
A = 0; // the 1= cell
x = 0; // do loop
do {
    A = A + 1; // the 100+ cell
    Buffer1[x] = Buffer2[x]; // the 100= cell
    x = x + 1; // increment counter
} while (x != 100 && Buffer1[x] != 0) // comparisons
```

**Note.** Even though I use the word 'loop' in this discussion, there really is no looping going on in the given example. No signal is fed from the output side of any of the cells back to the input side. This is because every effector is self-activating, that is to say, upon receiving a start signal, an effector will repeat its operation up to a prescribed number of cycles unless it receives a stop signal beforehand. Additionally, if a sensor is associated with an effector, effecting and sensing take place concurrently during the same cycle. This is not to say that actual signal looping cannot be done in COSA. Indeed, at times, it cannot be avoided. For example, it may be necessary to send a message to another component at every iteration of the loop and wait for a return message.

Constructing the loop component consists of using the mouse to drag the desired cells and connectors into a new component enclosure, and then connecting them together. Each connector is automatically given a unique ID. It is up to the designer to accept the default or use an existing ID. Click on the data structure to define the operands and constants for the component. Click on the effectors to open up a property box. This will allow you to specify the operator and operand types for each effector. Use the mouse to connect the cells to each other as desired. Finally connect the cells to the connectors. Of course, if you already have a loop component in the repository (which is very likely), it is easier to just copy the existing component and modify it to suit your requirements. Most low-level development in COSA will consist of reusing existing components.

**Reentrance**
The reentrancy problems that sometimes plague conventional algorithmic software are nonexistent in COSA. The loop component provides a perfect example of how potential timing conflicts are resolved. What would happen if the component receives a second start signal while the loop is still running? During development, COSA enforces the Principle of Motor Coordination or PMC which states that a start signal must not follow another start signal and a stop signal must not follow another stop signal. The PMC enforcer alerts the designer whenever there is a timing conflict. In the loop component, a conflict would be flagged if either of the two loop effectors receive a start signal before they are finished.

If two or more components require the services of another component simultaneously, they must use a message server. The latter places all request messages in a queue to be processed one at a time. If necessary, especially in time-critical environments, multiple instances of a loop component may be used simultaneously.

Note: The above does not mean that a recurrent process cannot be implemented within a component or with the use of two or more components. Reentrance is not the same as recurrence.

**Data Sharing**

High-level components do not have data at their level. They are just containers for other components. Most low-level components will have a data structure, the properties of which can be accessed by its cells. Components cannot directly access each other's data. That is to say, a component's internal data is always private. The only way a component may access another component's data is through an explicit request message. For example, a quick sort component may receive a request from another component to sort an array of integers. After completing the request, the sorter sends a 'service completed' signal back to the service requester.

The only data that may be shared directly between two components is message data. Even then, the destination component only has read permission. This is because two communicating components share a single message structure. The primary reason for the strict rule against data sharing is to prevent misuse. It would be too easy for one component to modify a property in another component without the knowledge of the other. This can potentially lead to incorrect assumptions and ultimately to catastrophic failure. In multithreaded architectures, it is customary to use semaphores and other techniques to prevent two threads from simultaneously accessing a data structure. These techniques are not used in COSA because data sharing is forbidden.

**Connectors**

A connector is a mechanism used by the designer to link two components together. Internally, however, only cells can be connected to one another, as explained below. Connectors exist mainly for the benefit of the application developer. The same can be said of components. Connectors do nothing, in and of themselves. Their purpose is to enforce compatibility between connected components. There are two types of connectors: uniconnectors and multiconnectors.

**Uniconnectors**

A uniconnector is a mechanism that serves as a one-way signal conduit between
two components. Every uniconnector must have a unique ID number and a gender (male or female). Males can only be connected to females and vice versa. Signal direction is always from male (plug) to female (socket). A male uniconnector and its female counterpart share the same unique ID.

Although uniconnectors are said to connect components together, internally however, only cells are connected to each other. A pair of uniconnectors is used to connect a sender cell from a source component to a receiving cell in a destination component.

Uniconnectors are normally used for one-to-one or one-to-many connections. Many-to-one communication is possible in COSA but it is managed with the help of a special component called the message server. Multiple messages arriving asynchronously must be placed in a queue and wait their turn to be processed. It follows that, while a male uniconnector may be connected to multiple female uniconnectors, the reverse is not true. That is to say, a female uniconnector cannot be connected to more than one male connector. However, a female connector may be attached to more than one cell residing in the component it is attached to.

Multiconnectors
A multiconnector is a set of uniconnectors. It too has a unique ID, a visibility level and can be either male or female. A multiconnector can have an unlimited number of male and/or female uniconnectors. A male multiconnector is the complement of a female multiconnector. For every uniconnector in a male multiconnector there must be a counterpart in the female complement. The gender of a multiconnector is used only as a way to ensure compatibility between complementary connectors. It is not as an indicator of signal direction. Indeed, a multiconnector can be either unidirectional or bi-directional depending on its constituents.

Multiconnectors are convenient composition aids because, more often than not, a component must make multiple connections with another, usually in the form of bi-directional two-pin connectors. The component designer may want to constrain a connection so as to require that all the connectors in an MC be connected simultaneously. If any of the uniconnectors fails to connect, the connection manager will not complete the connection. Multiconnectors add compatibility constraints and thus eliminate opportunities for errors. Unlike uniconnectors which are one-to-many, a multiconnector is a one-to-one connector.

Visibility
All connectors are created with an initial visibility level. A connector is visible only to other connectors residing at the same containment level. In other words, if component A is placed inside component B, A's connectors are visible to all components inside B but invisible to all components outside of B. It is up to the software designer to decide whether or not to increase a connector's visibility.
level so as to make it visible to components above its current level. Visibility levels are somewhat analogous to the public and protected keywords in C++.

Message Connectors
A message connector is a special unicommunicator used for passing messages from one source component to one or more destination components. It is up to the receiver to act on the message as quickly as possible and acknowledge receipt so as to free the structure for future messages. This is normally done with a two-way connection between sender and receiver, one for sending and one for acknowledgment. If a receiving component expects to receive multiple messages of a given type from several senders or if the sender cannot afford to wait for delayed acknowledgment, then it is best to use the services of a message server. Normally it is best to use messages for asynchronous communication. Since components can share data, they can communicate synchronously using signals.

However, effector-sensor associations are forbidden across components. That is to say, an effector in one component cannot be associated with a sensor in another. The COSA development environment will not allow it because, for organizational and security purposes, no component is allowed to have direct sensory access to what is happening in another.

Messages
A message is a structure with various fields of data, the content and format of which is only accessible to compatible sender and receiver components. A message structure is an integral part of its parent unicommunicator. In fact, they share a unique ID together. All messages must contain at least two fields, one to hold the length of the message and another to hold the ID proper. Some messages, such as the ones used with the message router component, must have two additional fields: one to hold the address of the sender cell and the other for the destination cell.

Sending a message is no different than sending a signal. Of course, the sending component should fill the message structure with pertinent data before signaling the destination component. Note that the data structure that represents the message never moves in memory. This makes for rapid message communication. A component must have either read or write permission to access or modify a message. This is enforced by the development tool.

Unique IDs
Both messages and connectors are assigned unique 16-byte ID numbers on creation. Each ID number represents a unique message or connector type. IDs are universally unique, meaning that no two signal or message types can share a given ID. It does not mean that there is only one connector per ID. An ID registry has to be maintained as part of the component repository for easy access to all developers within an organization. Public IDs and their definitions may have to be kept in a registry maintained by a non-profit organization. There are other issues
related to IDs that will have to be resolved. For example, a component designer may decide, for security reasons, to keep some of his or her IDs private by not publishing their meanings or the specifications of their associated messages. Or even the ID itself. This way, the ID essentially serves as an unbreakable 16-byte password.

**Construction and Destruction**
Every component that requires initialization must use a special female uniconnector for construction and a special male uniconnector to signal when it is ready. The component manager sends a special "construct" signal after loading a component into memory. The component designer should use construction connectors to string components together so as to achieve the desired order of initialization.

Destruction also requires special male and female uniconnectors. Again, it is up to the component designer to make sure that components are destructed in the desired order. After destruction, the component manager deletes the component from memory.

**Security**
I am toying with the idea of adding password protection to the connectors themselves. That would be in addition to type ID. Of course, if a component publisher declines to publish the unique IDs of a component's connectors, no one can connect to them. However, sometimes a software designer may want to use a known public component while finding it necessary to restrict access to a few chosen clients. Having the ability to add password protection to individual connectors would solve this problem.

One of the nice things about not allowing new executable code into the system is that it eliminates a lot of security problems. This is especially true in the age of the internet with thousands of machines connected to one another.

**Data Types and Structures**
Data typing in COSA follows the same principles as data typing in algorithmic languages such as C and C++. In other words, types will include integer, long integer, floating point, character, etc. In addition, the software developer will have the option of creating custom data types from any combination of the basic types. Typing enforcement is maintained automatically at the connector level. It is impossible to create two matched (male/female) connectors with incompatible message types.

**Applications, Components and Component Repository**
Every application is a COSA component and, as such, is an extension of the operating system. However, COSA will come with a suite of customizable service components that form the core of a minimum operating system. Examples are message servers (stack-based and queue-based),
file I/O server, memory manager, connection manager, component constructors and destructors, component and cell containers (lists, arrays, etc.), print manager, graphics manager, RAD development tools, etc. Of special importance is the connection manager (CM) which can be used by an application/component for self-modification. The CM can be directed to create new connections on the fly or to sever one or more existing connections if necessary. This can happen either at the component level or the cell/synapse level.

As explained here and here, every COSA component is a small part of the universal tree of all COSA components. Each branch of the tree represents an application domain. For example, an accounting branch would contain every component that is related to accounting. The classification of components will be automatic for the most part. This is to prevent misclassification. The component repository is also a browsable and keyword searchable database. Its organization and operation reflect the hierarchical structure of COSA applications. Its job is to facilitate and encourage reuse during application development. After a component has been tested and documented, it should be dropped in the repository for future reuse. The repository will come with tools that automatically search a stored component's documentation for pertinent keywords. Software designers should make it a habit of searching the repository before creating a new component.

See Also:
Parallel Computing: Why the Future Is Compositional, Part I
COSA: A New Kind of Programming, Part I

Copyright © 2010 Louis Savain