2012-04-15

Transistor scaled HPC application performance

Transistor Scaled HPC Application Performance

Technical Report BUCS-TR-2012-009

Jonathan Appavoo & Dan Schatzberg, Boston University
{jappavoo,dschatz}@bu.edu

This material is based upon work supported in part by the Department of Energy Office of Science under its agreement number DE-SC0005365 and upon work supported in part by National Science Foundation award #1012798.

Transistor Scaled HPC Application Performance

Jonathan Appavoo & Dan Schatzberg, Boston University
{jappavoo,dschatz}@bu.edu

We propose a radically new, biologically inspired, model of extreme scale computer on which application performance automatically scales with the transistor count even in the face of component failures. Today high performance computers are massively parallel systems composed of potentially hundreds of thousands of traditional processor cores, formed from trillions of transistors, consuming megawatts of power. Unfortunately, increasing the number of cores in a system, unlike increasing clock frequencies, does not automatically translate to application level improvements. No general auto-parallelization techniques or tools exist for HPC systems. To obtain application improvements, HPC application programmers must manually cope with the challenge of multicore programming and the significant drop in reliability associated with the sheer number of transistors.

Drawing on biological inspiration, the basic premise behind this work is that computation can be dramatically accelerated by integrating a very large-scale, system-wide, predictive associative memory into the operation of the computer. The memory effectively turns computation into a form of pattern recognition and prediction whose result can be used to avoid significant fractions of computation. To be effective the expectation is that the memory will require billions of concurrent devices akin to biological cortical systems, where each device implements a small amount of storage, computation and localized communication.

As typified by the recent announcement of the Lyric GP5 Probability Processor, very efficient scalable hardware for pattern recognition and prediction are on the horizon. One class of such devices, called neuromorphic, was pioneered by Carver Mead in the 80's to provide a path for breaking the power, scaling, and reliability barriers associated with standard digital VLSI technology. Recent neuromorphic research examples include work at Stanford, MIT, and the DARPA Sponsored SyNAPSE Project. These devices operate transistors as unclocked analog devices organized to implement pattern recognition and prediction several orders of magnitude more efficiently than functionally equivalent digital counterparts. Abstractly, the devices can be used to implement modern machine learning or statistical inference. When exposed to data as a time-varying signal, the devices learn and store patterns in the data at multiple time scales and constantly provide predictions about what the signal will do in the future. This kind of function can be seen as a form of predictive associative memory.

In this paper we describe our model and initial plans for exploring it.

High Performance Computing (HPC) application demand continues to surpass HPC system capacity. Serious challenges in power, reliability, and programability face us in continuing to scale HPC capacity[33, 72, 71].

> *Our goal is the exploration of a new class of extreme scale system that enables application performance to grow automatically with the size of the system.*

We want to move the burden of obtaining high performance from the application programmer to the system for traditionally challenging HPC applications such as those represented by Graph500[2] and even for applications that are difficult to express as a parallel program such as SST/macro[44]. This paper suggests a radically different approach to constructing an HPC system that: (i) is programmed easily but on which application performance scales with the number of transistors, (ii) is resilient to device failures, and (iii) consumes pico-watts per transistor. Our aim is a system that efficiently remembers and utilizes past execution to accelerate applications.

> *We suggest the use of a predictive associative memory, composed of billions of neuron like devices, to efficiently and transparently accelerate high performance computation, through memory recall from a vast base of computation it learns.*

This research is high risk, since the anticipated results are not guaranteed, but the potential benefits of a positive result to high performance computing are tremendous:

> *An HPC system that does not require multi-core programming, application performance that scales simply by making the machine larger, seamless robustness to individual transistor failures, and efficiency in power consumption.*

We are motivated by our own experience working on high performance scalable systems software[13, 11, 12] where: low-level runtime statistics of execution were used to identify "hot" computation and significant scalable improvements were achieved by using this knowledge to cache results in specialized scalable data structures that carefully controlled communication costs. Based on this experience, we observe that:

> *A functional trace of a cpu, containing opcodes along with operands and interrupts, reveals the underlying state changes that correspond to the computation being performed. We can leverage this observation to construct an alternative execution model in which we constantly: (i) monitor the cpu's actions and learn the state transition associated with statistically significant sequences of operations, and (ii) use any existing learnt sequences to predict the future state change associated with the currently observed operations. When sufficient constraints are met, we can synthesize a single update to the system's state based on the prediction without further execution, effectively performing the computation by associative memory recall. If enough structure exists and the mechanisms for learning and recall are efficient, it may be possible to realize dramatic automated gains in performance that are proportional to the size of the associative memory.*

The payoff we are seeking is:

> *A system that automatically eliminates increasingly larger fractions of application computation based on information gleaned over seconds, minutes, hours, and years — allowing programmers to focus on expressing their computational problem and not how to program the machine efficiently. Time to solution is significantly and automatically optimized by the system.*

This research seeks to quantify the benefit in power efficiency and productivity to be gained by the execution model, and will further attempt to quantify the overhead in area, power and cost of the proposed learning mechanisms. A key to the approach is that the execution model can lead to a system model in which the predictive associative memory function can be implemented with devices that have biological-scale efficiency and robustness. Specifically, the execution model can be seen as a signal processing generalization of *memoization*[27] that could use *neuromorphic*[66] devices for efficient implementation. Where memoization is the caching and recall of results from prior calculations to yield speedups and neuromorphic refers to devices that use transistors as analog components to implement neural processing.

Michie introduced memoization in his 1968 paper[27] with the following statement:

> "It would be useful if computers could learn from experience and thus automatically improve the efficiency of their own programs during execution... When I write a clumsy program for a contemporary computer a thousand runs on the machine do not re-educate my handiwork. On every execution, each time-wasting blemish and crudity, each needless test and redundant evaluation, is meticulously reproduced."

He goes on to observe that finding a function's value for a given input can either be done by calculation (rule) or recalled from memory (rote) but in either case the result is the same. He states that evaluation on a computer should,

> "on each given occasion proceed either by rule, or by rote, or by a blend of the two, solely as dictated by the expediency of the moment... and that the rule versus rote decisions shall be handled by the machine behind the scenes"

Researchers working on large-scale commercial data-center applications, written in a specific data parallel language, are exploring memoization as a way of improving performance[61]. Their results indicate that in some cases execution time can be reduced from 23 hours down to 10-20 minutes.

While the execution model implied by memoization is appealing, the standard realization as a language-level, programmer-driven, technique does not seem obviously applicable to HPC. The use of a restricted functional language or restrictions on the types of computations does not seem feasible for HPC. Also, the use of a complex language runtime would impose base performance penalties and rule out low-level expert programmer optimization. Our signal processing generalization, in contrast, operates transparently on low-level instruction stream signals. Our approach identifies and extracts hot paths composed of instructions and data from all layers of software, and caches their entire side effects as changes in system state. Hot path execution can then be eliminated by directly synthesizing a single update to the system state based on the cache information. While this approach may be viable as a system-level model, several key questions arise with its actual utility to applications:

1. Is there sufficient regular structure in HPC execution and data, such that caching portions of computation will be possible and prove useful?
2. In particular, can this approach serve as a building block for actual HPC application performance gains?
3. Does a real HPC application and its associated real data present sufficient opportunities for a system to automatically identify and replace portions of its execution with learned state changes?
4. Can a large long-term, persistent, associative memory of state changes yield application performance improvements that scales with the size of the memory?

These are the fundamental scientific questions that our work explores.

At the heart of the above approach is the predictive associative memory and its related statistical processes such as learning, pattern recognition and recall. Even if the answers to the above questions are positive, the associative memory and its functionality have to be efficient and viable at large scales. An extreme scale system might require a memory composed of billions of devices. Not only must the associative memory and its operations perform well and scale in terms of capacity, but also it must scale in terms of power consumption and reliability. It is likely that the devices for the associative memory will have to surpass current technologies in density, robustness, and power efficiency.

In the 80's, Carver Mead pioneered a class of device, called "neuromorphic"[57], to provide a path for breaking the power, scaling and reliability barriers associated with standard digital VLSI technologies. Recent neuromorphic research examples include work at Stanford[19], MIT[63], Johns Hopkins[7], and the DARPA sponsored SyNAPSE Project[6]. These devices operate transistors as unclocked analog devices organized to mimic neurobiological circuits several orders of magnitude more efficiently than equivalent digital counterparts. They are also by design robust to failures through the use of distributed and replicated structure similar to the neural circuits they model. One specific neuromorphic computational paradigm is cortically inspired associative memory and recall[24, 50, 39, 9, 8].

As a concrete example consider the neuromorphic associative memory chip implemented by Pouliquen et al.[60]. The authors demonstrate that their chip is capable of performing an associative recall using approximately $100nJ$, or $1/6000$ of the $600\mu J$ consumed by a traditional processor performing the same function. While challenges still exist for commercially viable, general-purpose neuromophic associative memory chips, announced products, such as the Lyric GP5[4] Probability Processor and memresistor chips[43, 56], are poised to revolutionize pattern matching and memory function. In addition, machine learning researchers are exploring scalable predictive associative memories capable of implementation with neuromorphic devices and providing the kind of predictive associative memory our model would need[35, 42, 41, 65].

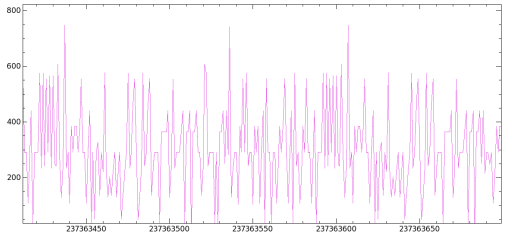
Our execution model leads to a system model that we describe in Section 2. The main focus is to establish that viable HPC application improvements would be enabled by these models. Our approach is to investigate the core premise that a challenging HPC application can benefit from the execution model of our system. Specifically, building upon our prototype compute node simulator, we plan to experimentally quantify the key questions stated on page 2. We will work up from simple calibration workloads to HPC kernels and finally explore Graph500 applications and the discrete event simulator SST/macro. Section 3 describes our method along with the project timeline and resources. As a prerequisite, we start with a more detailed development of our execution model.

1 Motivation and Background

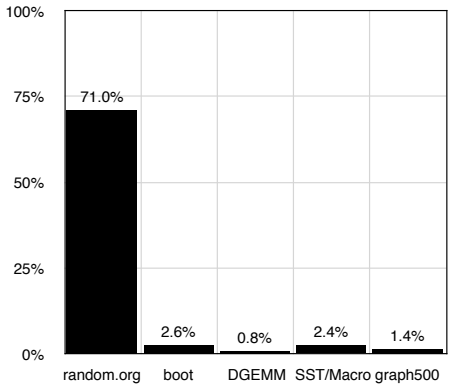
Our signal interpretation of memoization and the associated idea of using neuromorphic devices for its implementation have been developed over the last eight years. Recently, at Boston University we have been constructing a prototype system level simulator to help us explore these ideas. This prototype and the manner in which it will be extended and used for this work will be discussed in more detail in Section 3. In this section we will provide a brief intuition for the ideas using data from the simulator and then provide some context for the development of the ideas. We conclude the section with a formal description of the optimization that our model provides; this is presented as the impetus for our system model, as well as indicating how we will explore the key questions of this research.

Our execution model aims to exploit patterns, or more formally redundancy, in computer execution. A system will update its state by mixing traditional program execution and associative memory recall. From this perspective, the effectiveness of such a hybrid system depends on the amount of redundancy present in execution and then of course the ability to recognize and exploit that redundancy.

Consider a simple set of experiments in which we obtain a signal that captures the entire execution of a uniprocessor node running Linux for some workloads. A small portion of such a signal is depicted in Figure 1a. Each operation that the cpu can perform, including all instructions executed from all layers of software and all system events such as clock and I/O interrupts, are assigned a unique y value. Our signal recording mechanism precisely records the time at which each operation is started. Given a serialized cpu model, we get a simple one dimensional signal. For simplicity, the figure plots the data as a sequence, where the order in which the operations occurred is shown on the x-axis. To get a rough feel for how much redundancy is potentially present, we can use a standard compression program, gzip[3], to compress the y value recordings and compare the size to the original.



(a) portion of node operation signal



(b) gzip compression

Figure 1

Figure 1b, plots the compression results gathered with our current prototype simulator for four scenarios; boot, DGEMM, SST/macro and Graph500. In the boot scenario, we power on the node and let it boot into a full general purpose Linux installation and then have it immediately shut itself down, tracing the entire process. In the second scenario, we trace the execution of a simple Fortran program that invokes the standard BLAS[1] DGEMM routine (with three 200×200 matrices of random values and alpha of -1 and beta of 1). In the third experiment, we run a test application of the discrete event simulation SST/macro[44]. In the final experiment we run a small-scale Graph500[2] code. Compressing the traces yields output sizes that are, respectively, 2.63%, 0.8%, 2.4% and 1.4% of the orignal traces. The figure includes a 100000 random event data file generated from *random.org*[38] that includes redundancy due to our data format (two bytes per event with only 492 unique event types recorded during the boot experiment). We see that a random source of events compresses to approximately 70% of the original size. While one must be careful not to draw too many conclusions from this data, it does at least imply to us that a great deal of redundancy potentially exists in execution and it may be detectable via raw execution traces.

1.1 Background

This subsection presents a brief summary of the development of our key observations, as well as the questions that these observations have raised. Over the last several years, at the University of Toronto and IBM, we have been working on scalable systems. While constructing and optimizing general purpose scalable systems software[11, 13, 12, 15, 31, 51] for large scale multiprocessors[37, 47], three observations stood out:

1. The larger the scale of the system, the more one is driven to architectures that utilize larger and larger numbers of simplified computational nodes that incorporate a small number of processors and some fixed amount of memory. The hardware and software of such systems must focus on locality of communications. In the hardware, this results in interconnect topologies that scale by limiting or avoiding global buses and connecting each node to a limited number of physically proximate nodes, often introducing hierarchical structures for global communications. This drives one to use distributed systems software techniques that limit communication and focus on using communication between small physically local sets of nodes. Unfortunately, as the scale increases, it becomes more complex to develop high-performance software without significantly restricting its features. More and more of the programmer's effort is spent on either explicitly or implicitly managing communication rather than expressing a simple description of steps to solve the desired problem. Ultimately, the system may increase in computational power but only for a smaller and smaller set of limited computational tasks that can be programmed to that scale. Additionally, as the scale of the system crosses a certain threshold, the attendant decrease in mean time to interrupt (MTTI) further reduces programmability and drives one to even more specialized software.
2. Given the complexity of software and dynamics of I/O, optimizing for scalable end-to-end performance is driven by understanding what forms the "hot paths" so that communications can be moved from them to colder, less critical paths. This requires one to consider what is executing, how often, where the hot paths start and end, their life-times, and how they interact. These questions ultimately become a question of the temporal statistics of the system behavior and its external interactions that ultimately are the origin of all execution paths.
3. Finally, that the most effective common optimization is to exploit hot path information to introduce some form of scalable caching that avoids the hot computation altogether, essentially doing a dynamic runtime space-time tradeoff. As such, focusing on scalable hash-tables and their related search algorithms and mechanisms proved very useful[12, 13]. We also observed that this kind of optimization was general and independent of parallelism, improving performance whether the applications were parallel or not.

These observations led to the following preliminary questions. Can system wide, transparent, end-to-end performance optimization be itself the parallel application such that the real application's performance improves with the scale of the system without parallel programming? Is there a computational optimization task that continues to scale as the hardware parameters are pushed that could be used to speed up general purpose computation rather than directly implementing it? Could identification of hot paths and associated caching be such a task and could it be integrated into the basic operations of the system? These questions ultimately led to our execution and system models and the key questions on Page 2.

Many areas of computer science, including HPC, try to consider and exploit the statistical structure in runtime behavior[26, 28, 73, 25, 29, 10, 23]. From the hardware perspective, branch prediction, data and instruction caching, trace caches, and more advanced techniques[18, 48, 58, 67, 68, 69, 70] all attempt to exploit repeated patterns. However, these approaches are very limited in their scale and applicability given the limited temporal windows and type of events they consider. The dynamic compilation community also attempts to exploit runtime statistics. Influenced by the trace cache methods of Dynamo[17], related work[21, 22, 40] and our own experience[14] with SIMOS[62], we started considering such forms of instruction tracing and caching for system wide

optimization. However, we quickly found that the main focus of these techniques is to improve code quality and not ultimately determine why code was executing with respect to interactions across all software layers, data and external sources. Further, the techniques focused so heavily on inferring information from addresses and program semantics that they did not pay close enough attention to what the computers were doing as a statistical phenomena. We were much more interested in an approach that would lead to a holistic system model and permit a physics or information theory-like treatment of the system's behavior. A scalable system could be constructed that would exploit redundancy in a system's execution in the form of a transparent memory, or cache, of execution.

In the systems research area, the renewed interest in virtualization has brought with it interest in exploiting tracing and logging for various functions from debugging[49] and intrusion detection[30] to deterministic computation[16]. However, the only work that we know of that starts to consider the trace data as a complete artifact to infer structure in execution is Tralfamadore[54]. Tralfamadore, however, does not attempt to rigorously quantify the structure or exploit it for automatic runtime improvements. Trace analysis in Tralfamadore, at this point, is used only as a means of assisting software developers in understanding how their programs execute.

Other systems researchers have been turning to machine learning techniques to provide guidance on how to optimize system behavior. In particular they treat logs[76] and profile data[32, 73, 20] as inputs to machine learning algorithms. These algorithms are then used to identify cases in which a system is performing unexpectedly or to tune system parameters, such as scheduling parameters or compilation flags. We do not know of any researchers that have sought a systematic way of integrating machine learning mechanisms into the computational model itself.

Recent work by Shen's group[74, 46] attempts to exploit statistics in both instruction and data to guide dynamic compiler based program optimization. While we strongly support the use of statistics, we contend that taking a systems level approach is more likely to lead to a fully automated solution that can exploit structure across all layers, is language agnostic, and can be HPC viable. Based on their results, Tain et al.[74] states that a statistical based approach enables "Holism". They define holism as the ability to predict entire program, and potentially system, behavior and thus enable whole program and system optimization. However, the validation of this claim is left as future work. Furthermore, no intuition is given as to how a system wide dynamic compilation layer can be constructed that is both language agnostic, transparent and could meet the performance challenges of an HPC workload.

Our work has independently developed with this "Holism" view point in mind. When exploring the trace based approaches in the context of the entire systems optimization a key insight was made. Namely that a computer's entire operation could be treated as a low-level unified temporal signal that we refer to as an *Execution Signal (ES)*. An ES can be recorded and analyzed to quantify and identify statistical structure in the entire system behavior. An ES could reveal how the programs, data, external events and the machine model interact as a unified statistical process in time. Furthermore, feature analysis of this signal may be synonymous with hot path identification and their associated state changes. An ES may also capture the statistics of the input data values and external events influencing and causing the hot paths, without knowing their actual sources. Waterland[75] observed that an ES interpretation was analogous to a physics based dynamical systems treatment of the computer's operation. We later found that Giunti[36] had also suggested a dynamical systems analogy and that Bradley et al.[59] are presently pursuing a dynamical systems approaches to processor analysis.

Three critical aspects to the ES interpretation are stated below:

1. An Execution Signal captures all operation as a temporally unified signal (for practical reasons

it may be decomposed into many signals but all related through a common time base). A common clock should be used to record all operations that are interesting with respect to the programmed behavior (rather than idiosyncrasies of the hardware implementation[59]). The two obvious equivalent execution signals would be an operation trace and a state trace. Where the operation trace is of every instruction (not instruction addresses) and associated data operands (actual data values, again, not addresses) along with all external interrupts (external events that the computer is constructed to respond too) recorded with respect to a common unified clock. A state trace records the entire state of the machine against a common clock. For example, at time 0 the entire state that is recorded would be the initial value of all registers and memory. On every state change the time and state of the machine would be recorded. The resulting path traced by either signal captures all structure in the system's behavior.

2. The operation of any modern von Neumann machine can be viewed as an Execution Signal and analyzed for structure without understanding the details or semantics of the instruction set.
3. Redundancy in an Execution Signal can be used to identify state transitions that may be worth caching. Execution Signals from different programs that compute the same result (lead to common values of some identified portion of the machine state) can be compared with respect to redundancy.

Execution Signals provide us with a concrete signal processing interpretation of execution. This gives us a context for recasting memoization and a basis for neuromorphic device realization.

1.2 Application Performance, State Transition and Execution Compression

Now that we have a better feel for how these ideas developed and relate to other work we can more precisely define our execution model in terms of a formalism that we can use to guide our system model and experimental quantification.

We shall define an application as all components of the system, both hardware and software, that performs a desired computation C. Assume a sequential machine M that defines a state vector S and a set of possible actions or operations $O = (o_0, ..., o_k)$ on S. S is a vector that is a constituent of the values of every device that composes M. For example, the state of all registers, memory, I/O registers and I/O devices are contained in S. At any given time only one $o \in O$ can be active or executing. The duration of time that an operation takes to execute (complete) is a function $d(o, S)$. It should be noted d is a nontrivial function as it depends not only on implementation attributes of the machine but also on current state. We observe that an application, once started, defines a sequence $A = (a_0, a_1, \ldots, a_n)$ where $a_i \in O$. As such, a_0 is the first operation executed by A and a_n is the last. The time to solution for A, T_A, is defined to be

$$T_A = \sum_{i=0}^{n} d(a_i, S_i) \tag{1}$$

Where S_i is that value of S prior to a_i being executed. For the sake of this discussion we restrict ourselves to applications that produce a "result" to C in a finite length sequence either due to a termination test passing or by imposing a restriction on either the length of the sequence or on T_A.

One of the key goals of a high performance system is to yield solutions to computations in a time frame that would be intractable otherwise. There are three typical approaches to shortening an application's time to solution:

1. Reduce the duration $d(a_i, S_i)$. The common approach here is to improve features of M, by increasing clock frequency, reducing memory latency or exploiting instruction level parallelism through pipelining. The advantage of this approach is that it requires no changes to the programming model or software.
2. Decompose A into subsequences that do not depend on each other and hence can be executed in parallel. This of course requires extending M by replicating its components and adding interconnection facilities for the additional required communication. The decomposition may not be straight forward, or even possible. The approach typically introduces significant complexity to both the software and hardware and can impose considerable cost on application programmers.
3. Construct a new application, A', that computes C such that $T_{A'} < T_A$. In general we think of this as a transform on A to produce a sequence A' of operations that has shorter duration, or a construction of A' that results from a different algorithm that requires fundamentally fewer operations such that $length(A') << length(A)$, or a combination of both approaches. We typically think of the former as code quality improvements implemented by compilation tools and expert machine programmers while the latter we see as algorithm changes that require fundamental application and design knowledge.

This work seeks an orthogonal approach that can be transparently applied independent of the others. The goal is to automatically eliminate redundant portions of a sequence based both on information gathered from the execution of prior applications as well as all prior executions of the current application.

More specifically one can observe that each a_i of A represents a change in state, S, from the value S_i prior to a_i being executed to S_{i+1} after it was executed. With this in mind, repetitions of a subsequence $\alpha = a_p...a_q \in A$ may imply that the change from S_p to S_{q+1} is a difference in S that can be cached and applied to S more efficiently than re-executing α. From this perspective we are looking to shorten T_A by *deleting* portions of A while synthesizing a direct update to S based on cached values. We call this kind of optimization a form of *Execution Compression*. While the gzip compression results of Figure 1b cannot be interpreted as Execution Compression, they do suggest that real execution can have dramatic redundancy (97-99%) even within a single run. If we can translate even some fraction of this redundancy into deletions we may be able to realize equally dramatic performance gains.

2 Model : Programmable Computation Cortex Machine

In this section we describe our system model for a new HPC system that attempts to exploit Execution Compression, introduced in the prior section. As stated in the introduction, our model realizes our signal processing generalization of memoization using neuromorphic devices. To do this, the model uses a large, shared associative memory to accelerate the operation of its computational nodes using Execution Compression. Our model, which we shall refer to as a *Programmable Computation Cortex Machine (PCCM)* is composed of four layers as illustrated in Figure 2. The bottom layer, Software Stack, is the entire software that an application executes, including all operating systems, library and application software. This can be thought of as the binary image of what one would load onto a computational node of a system. Above Software Stack is the Virtual Node Instances (VNI) layer and is composed of one or more virtual node instances that form a standard programmable interface to the system. The software composing the Software Stack is constructed to conform to the VNI specification. As an example, the VNI of our prototype system is a uniprocessor x86 compute

node composed of memory (including a ROM bios), a local storage device, and an interconnect interface. The SS of our prototype is an arbitrary x86 software stack specified as a harddisk image. For most experiments, this stack is composed of a Linux distribution plus application binaries. As stated, in our prototype system we have chosen to utilize an x86 based VNI interface in order to be able to construct application Software Stack images from the rich existing body of x86 software.

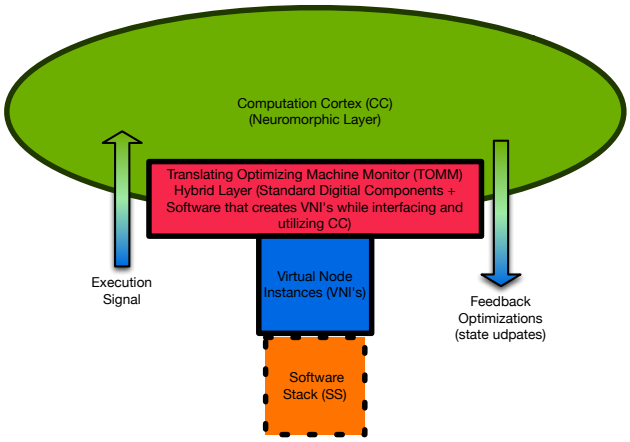


Figure 2: Programmable Computation Cortex Machine (PCCM)

The next two upper layers form the core of what is new in the PCCM model. At the top is a large scalable predictive associative memory called the *Computation Cortex (CC)*. Our design goal is that the CC and its interfaces be realized with neuromorphic devices. In particular we design our system model around the CC performing signal processing-based associative memory. Its input is in the form of Execution Signals and its automatic and continuous feedback is in the form of state predictions. The CC is a system wide shared structure. Ultimately, we expect it to constitute the majority of the system. Its function is to learn, store, and recall redundant computation based on Execution Signals (ES's) fed to it. Abstractly, one can think of the CC like a global hash table or database in which the immediate state associated with the source of an ES is a lookup or query. A result from the query is a feedback signal that is a set of future states (potentially expressed as a difference or function of the query state). If the set is composed of zero states, then the CC has no useful knowledge yet of the current state. If the set has one state then the CC knows that the current state will result in exactly one future state and the source of the ES can be updated directly to its state. If the set is composed of multiple states then lookup result is a prediction of a possible set of future states of the VNI associated with the ES. The CC is intended to be a long term structure for which at least some part of it will persist at an installation for the life time of the system. As such, at least some of its state will be accumulated not only over multiple nodes, but also over multiple application executions and multiple different applications. The main experimental goal is to establish that the ES of at least one node for at least some subcomputation of a real world HPC application has sufficient structure that the PCCM model would be viable and effective.

Below the CC is a machine monitor layer, the *Translating Optimizing Machine Monitor (TOMM)*, that is responsible for creating and managing the VNIs, translating their operation into a form that can be fed to the CC, interpreting the feedback from the CC, and implementing optimizations to the VNI's state. As illustrated by the arrows on the left and right of the diagram TOMM is responsible for implementing the two information flows. Each VNI represents a state vector that TOMM creates and manages. Unlike standard virtual machine monitors, TOMM's goal is not the over-commitment or multiplexing of physical resources – instead, its goal is to create VNIs such that the operation of each VNI can be translated into an Execution Signal for pre-processing and forwarding to the CC. The operation of the VNI can then be optimized based on feedback from the CC. In the long run, the TOMM layer, may be implemented entirely in hardware. For the moment, we expect TOMM to be implemented as a combination of hardware and software much like current

virtual machine monitors.[1].

Figure 3 illustrates our initial view of how the TOMM layer will be decomposed. Our goal is that the main shared global layer of the system is the CC, as its function and implementation are intended to be directly scalable with the size of the system. As such, the TOMM is designed to be a distributed layer composed of TOMM components that are associated with each VNI of the system. Each component will be designed to operate independently, without direct communication or coordination with each other. Unlike a Virtual Machine Monitor, a TOMM component manages a single VNI using a set of dedicated physical resources. One can think of a TOMM component as an extended standard HPC compute node that transparently interconnects to the common CC. As part of the VNI construction, a TOMM component integrates *probes* into the VNI that produce streams of values for actions that are taken by the VNI. A TOMM component has a *translator* subcomponent that translates the values from the probes into an a set of signals, with a common time base (bottom of Figure 3). These signals constitute the Execution Signal for the VNI. The signals are fed to initial signal pre-processing stages that operate on a specific component of the ES to aid in the processing by the CC. We refer to these stages as *nerves* as we see them akin to the biological processing stages that do modal-specific signal processing, potentially filtering and extracting specific features to aid in recognition of key points in the signal by the associative memory function of the CC. Conversely, a TOMM component also processes the feedback stream from the CC. The feedback stream represents future possible values of some or all of the VNI's state. The feedback is processed by a subcomponent called the *optimizer*. The optimizer is responsible for determining if and how to update the VNI's state based on the feedback. While we would like the optimizer in the common case to be able to directly apply the feedback as an update to the VNI state, we recognize that it may in general have more complex optimization mechanisms that use the feedback. For example, the optimizer may utilize the feedback to implement a form of speculative execution, where VNI instances are created and seeded to explore each predicted state. The optimizer uses a set of *actuators* that allow it to change the various portions of the VNI state in a safe, efficient and predictable way.

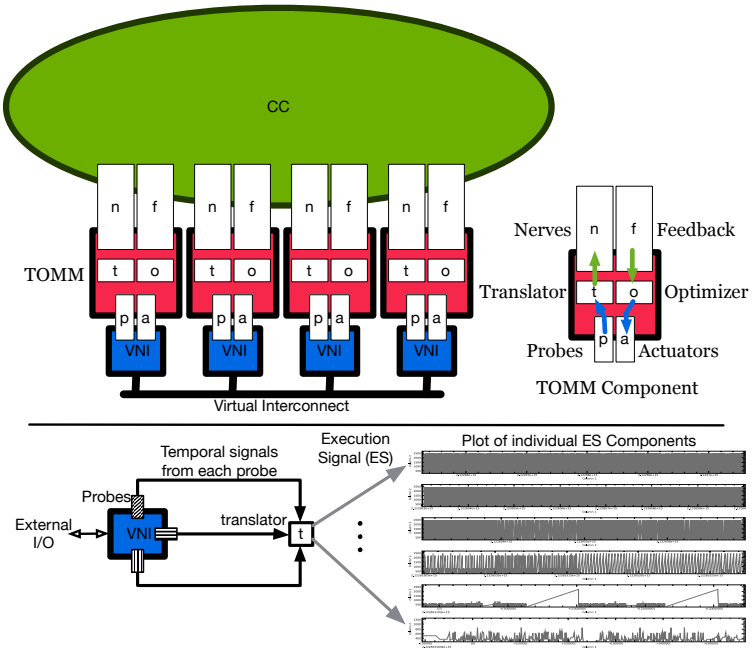


Figure 3: TOMM

3 Method

Practically we seek to quantify the value of our execution model on application performance and secondly the implementation viability given our associated system model. To do this, we build on the descriptions of Execution Compression from Section 1.2 and the PCCM system model from

[1]Such as Intel's VT support and hardware tracing and monitoring facilities

Section 2. Our overall method is a quantitative experimental approach that employs simulation. We break the work down into three objectives and related phases:

1. Simulation Infrastructure Development
2. Execution Signal Prototyping, Validation and Analytics Development
3. Experimental Exploration

While the above are discussed below in isolation, the expectation is that the steps will overlap and also be executed iteratively.

The goal of our simulation infrastructure is to create accurate data and enable a flexible workflow. We split our workflow into two phases:

1. Execution Signal generation and recording, and
2. Execution Signal analysis and Execution Compression simulation

The first phase uses a TOMM simulator on which we can prototype various probes and translators to construct and record a complete ES from a fully functional x86 VNI that is capable of running complete x86 software stacks. To this end, we have constructed an initial TOMM simulator illustrated in the bottom portion of Figure 4. The second phase of our workflow takes ES recordings and explores them with a set of analytics and potential neuromorphically viable CC algorithms in order to quantify the potential for PCCM Execution Compression. Given the sheer sizes of realistic ES recordings and the compute intensive nature of the analytics this second phase of work will be conducted on Blue Gene systems.

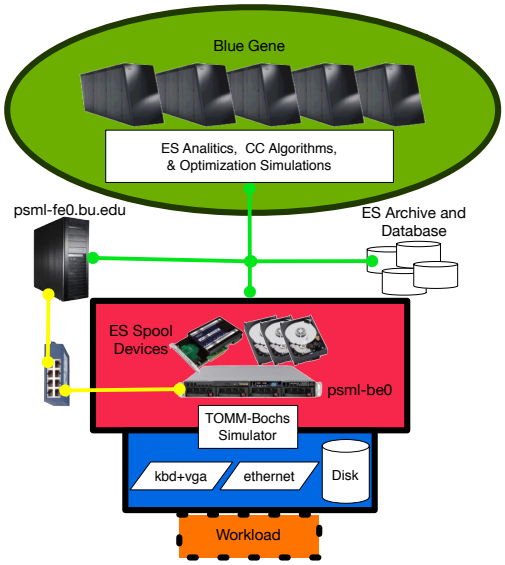


Figure 4: Simulation and Workflow

The first goal of the infrastructure phase will be to complete our TOMM simulator and replicate it so that multiple ES's from interacting VNIs can be studied. Our current simulator uses a combination of hardware and software. We have added a low overhead probe and recording infrastructure to Bochs[53], an open x86 full system simulator. Bochs is a configurable simulator capable of simulating not only the processor but also all I/O devices such that it can boot and run arbitrary x86 software including Linux, Windows and potentially research based operating systems such as Kitten[52], [2] along with their entire application software stacks. We run our modified version, which we call TOMM-Bochs on a dedicated hardware system (psml-be0) that has been carefully configured to ensure that the probe data can be recorded for about 3.5 hours of execution time with no pauses and a small perturbation to the simulation. To do this we have constructed a high-performance multi-core in-memory probe log and a raw multi-disk striping mechanism.

Our current version of Tomm-Bochs implements a high precision hardware based timestamp source and two probes: one for processor interrupts and the other for processor instructions. The current goal we are working on is to extend our TOMM simulation infrastructure to get a complete set of probe data that will allow us to construct a complete and accurate Execution Signal. To

[2] We can configure Bochs as an HPC compute node stripping it of any unnecessary io devices such as mice, display and keyboard.

do this, we will include probe data that accounts for all operand data by adding two more probes, one for the operands for each instruction and the other for memory and io port writes. In order to achieve this we are currently upgrading psml-be0 with a 1TB solid-state-disk and are developing new probe implementations. Our goal is to be in the position that once funding is secured we will finalize our single node TOMM simulator configuration and be able to construct a set of 4 TOMM simulators each capable of producing complete and accurate probe data streams.

For the second workflow phase, Execution Signal analysis and Execution Compression simulation we will construct a set of analytics to compute metrics discussed later. In addition, we will explore one or more of the following learning mechanisms, depending on time and resources: Deep Belief Nets[42], Causal-State Splitting Reconstruction[64, 65], Hierarchical Temporal Memory[35, 34, 45] and Cogent Confabulation[41]. All methods explored will be evaluated for their effectiveness in automating Execution Compression and implication to neuromorphic realization.

Prior to our experimental exploration, we shall use a set of controlled synthetic applications to develop a prototype Execution Signal specification from our probe data, with which we will validate accuracy and correctness of the Execution Signal, and develop our set of analytics. To date we have constructed two simple synthetic applications that we can use to gather Execution Signal recordings that have a well understood structure.

The first is a simple server and client that exchange request and response packets directly using the node's interconnect. The client takes a request stream as input along with timing information as to when to send the requests to the server. It then produces, as output, the responses from the server along with when they were received. The server reads requests from the client and computes a simple function such as factorial on the request value and returns the result to the client. The server is run on our TOMM simulator and probe data is gathered for the entire operation. The server can be configured to compute the function in one of several ways including a table lookup. The server is simple enough that it can be run either on top of Linux or directly from the boot loader with no OS. The second workload is a simple program that invokes DGEMM with specific matrix values and specific values of alpha and beta.

Our plan is to run both workloads with controlled inputs that force varying degrees of complexity and redundancy in the Execution Signal. We will then evaluate the recordings for the four factors described in the next subsection, and validate that they are effective for quantifying the opportunities for Execution Compression. Using the first workload, we can precisely test our infrastructure and experimental method given the known structure we have engineered through the control of the input sets and node software environment. Using the second workload, effectiveness with a less controlled environment will be explored. We will also simulate the potential for automating deletion identification, state recall and update, using one or more of the learning methods identified above. We intend to study the redundancy in an ES from not only a single application run, but also across multiple runs and across differing applications.

The goal of the experimental phase is to explore the potential of the PCCM model by experimentally investigating the conjectures about computation and associated optimizations that the model might enable. We first briefly outline the general method we intend to follow and then discuss the specific HPC application driven evaluation. This section utilizes the terminology defined and discussed in Sections 1.2 and 2.

While the PCCM model might use the CC to enable several forms of optimization we focus on Execution Compression via the deletion of portions of an application's execution as was introduced in section 1.2. We now define a deletion, in the context of the PCCM model, to be the situation where

portions of a VNI's future operation can be deterministically known and thus avoided or deleted and the VNI state simply accelerated to a future synthesized state. As such this optimization process requires utilizing the CC to recognize that a deletion is possible, generating the future state and updating the VNI. The goal is to establish that a PCCM system could conceivably stitch deletions into the operation of the VNI for a net improvement in performance.

Specifically we focus on exploring the potential for automatically improving the time to solution for an application without the requirement for reprogramming.

There are three main experimental goals we intend to pursue:

1. Validate the underlying conjecture for deletions; identify that deterministic portions of A exist and establish that they manifest themselves in the Execution Signal generated by a realistic VNI, at least for some restricted class of computation.
2. Show that for at least some applications, deletions can be performed, even if by hand, and lead to significant saving in time to solution.
3. Show that it is realistic to expect a PCCM system to learn at least some subset of the potential deletions for a workload, through the utilization of appropriate probes, nervers and CC analytics and storage, and that it is also realistic for the TOMM to apply them.

$$PCCM_{T_A} = VNI((1-\lambda)A) + CC(\lambda A) \tag{2}$$

Given a simulated PCCM structure, the experimental framework we propose employs Equation 2 as a simple time to solution model in which we explore deletion optimizations that permit a fraction, λ, of A, an application, to be deleted based on CC driven TOMM updates. The functions $VNI()$ and $CC()$, respectively, are the time that it takes the VNI to execute portions of A and the time it takes the CC to recall the portions of A that it was able to identify as redundant and for the TOMM to update the state of the VNI based on the feedback from CC. Given Equation 2, there are four main factors we will evaluate for a PCCM prototype for a given application:

1. What is the set of subsequences of A that can be deleted? To determine this, we will need to define, for each experiment, criteria for identifying repetitive changes in the VNI state. Identifying candidate sets of deletions for a given experimental application is of interest independent of the PCCM model. The set represents hot paths that would be useful to anyone working on optimizing the application.
2. What fraction of $VNI(A)$ can be deleted? Given that we have a functioning VNI, albeit a simulator, it gives us a consistent baseline that we can use to experimentally measure a relative cost for these values. This ratio defines an upper bound for the improvement that our experimental PCCM could achieve. However, independent of the PCCM model, this data will be useful to any research on these applications.
3. What is the set of actual deletions that can be identified *and realized* and how does it compare to the set of all subsequences that can be deleted? To answer this question we will construct ES analytics that simulate potential methods for processing the ES recordings and identifying a potential set of deletions. We expect to include at least one or more of the learning mechanisms identified earlier in the analytics methods.
4. For each realized deletion, d, what is the ratio $CC(d)/VNI(d)$? This ratio defines the effectiveness of execution by recall versus computation. This set of ratios will be used to evaluate the ability of the PCCM model to actually achieve time to solution improvements for a given ES recording. We will be able to explore how various performance parameters associated with the PCCM model would affect the final runtime of the applications. In particular it lets us define an envelope that a PCCM model would have to meet in order to realize a net gain in performance.

Once we have validated our infrastructure and experimental process using our synthetic control applications, we will explore standard HPC primitives and kernels using both controlled and application data. Given that the VNIs of our TOMM simulators are standard x86 instances we can use standard Linux implementations of BLAS routines. We will begin by building on our DGEMM control application to look at other isolated BLAS routines. We will then move on to looking at one or two benchmark HPC application kernels drawn from the HPC Challenge[55] and or LAPACK suites. We are seeking a complementary approach to optimization efforts such as PLASMA[5]. We hope to establish that our system can look for and exploit redundancy no matter what the base implementation. Our aim will be to quantify the four experimental factors described above as well as their sensitivity to both the data and implementations and then look at the effectiveness of the learning methods. Eventually, we hope that our approach will alleviate the human optimization burden and developers will be able to rely on the system's ability to constantly improve the performance. Our model, however, in no way precludes human optimization.

The larger application goal is to evaluate the potential for PCCM based execution compression to improve the performance of a complex larger HPC application such as a Discrete Event Simulation (DES) system. For exploration we have have been experimenting with the SST/macro simulator. SST/macro has been carefully designed and developed to provide a flexible and accurate base simulation infrastructure that can be used for several important DOE ASCR efforts (CERF, CECC, FOX) . It enables accurate macro-scale simulations of both skeletonized MPI and non-MPI programs with ranks ranging from $10^5 - 10^6$ nodes. However, SST/macro itself is a sequential application. Our goal is to evaluate if the PCCM model can essentially turn SST/macro into an HPC application whose performance would be transparently improved, thus effectively allowing it to simulate larger node ranks for some application.

As part of our involvement in the Fault Oblivious eXaScale (FOX) project, our group at BU is working with developers of SST/macro to simulate both a target application and system software stack. Using this knowledge, we have experimented with running SST/macro on our current TOMM simulator and gathering probe data for one of the simple test simulations that comes with SST/macro. Figure1b, includes compression results for probe data gathered from a default ten iteration test SST/macro simulation of an MPI ping-pong application run on a fat-tree network of 1000 nodes. This gives us the confidence to know that the TOMM simulator we are developing will be sufficient to gather the necessary data for our Execution Compression analysis.

It should be noted that, given the potential complexity that a real SST/macro simulation, or other suitably complex DES simulation, may represent in terms of an Execution Signal, we will likely need to decompose the analysis into an incremental and iterative set of experiments. Our expectation is that we may need to vary and study different components of a DES in isolation, building up to complete evaluation.

References

[1] *BLAS*. http://www.netlib.org/blas/.

[2] *Graph 500*. http://www.graph500.org/.

[3] *Gzip - GNU Project - Free Software Foundation*. http://www.gnu.org/software/gzip/.

[4] *Lyric Semiconductor | Technology: Probability Processor*. http://www.lyricsemiconductor.com/technology-processor.htm.

[5] *The Parallel Linear Algebra for Scalable Multi-core Architectures (PLASMA) Project*. http://icl.cs.utk.edu/plasma/index.html.

[6] *The SyNAPSE Project | Outreach And Impacts | CELEST | NSF Science of Learning Center*. http://celest.bu.edu/outreach-and-impacts/the-synapse-project.

[7] Andreas Andreou. *Adreas G. Andreou*. http://www.ece.jhu.edu/faculty/andreou/AGA/.

[8] Andreas G. Andreou. *Neuromorphic architectures: challenges and opportunites in the years to come*. http://mind.nd.edu/news/WorkshopSlides/11_Andreou.pdf.

[9] Andreas G. Andreou. *Stochastic Computational Associative Memories: Neuromorphic Architectures Beyond Moore's Law*. http://mind.nd.edu/news/WorkshopAbstracts/Abstract-Andreou.pdf.

[10] Clint Whaley Antoine, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the atlas project. *Parallel Computing*, 27:2001, 2000.

[11] J. Appavoo, K. Hui, C. A. N. Soules, R. W. Wisniewski, D. M. Da Silva, O. Krieger, M. A. Auslander, D. J. Edelsohn, B. Gamsa, G. R. Ganger, P. McKenney, M. Ostrowski, B. Rosenburg, M. Stumm, and J. Xenidis. Enabling autonomic behavior in systems software with hot swapping. *IBM Syst. J.*, 42(1):60–76, 2003.

[12] Jonathan Appavoo. *Clustered objects*. PhD thesis, University of Toronto, Toronto, Ont., Canada, Canada, 2005.

[13] Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Marc Auslander, Michal Ostrowski, Bryan Rosenburg, Amos Waterland, Robert W. Wisniewski, Jimi Xenidis, Michael Stumm, and Livio Soares. Experience distributing objects in an smmp os. *ACM Trans. Comput. Syst.*, 25(3):6, 2007.

[14] Jonathan Appavoo and Supervisor Michael Stumm. Clustered objects: Initial design, implementation and evaluation, 1998.

[15] Jonathan Appavoo, Volkmar Uhlig, and Amos Waterland. Project kittyhawk: building a global-scale computer: Blue Gene/P as a generic computing platform. *SIGOPS Oper. Syst. Rev.*, 42(1):77–84, 2008.

[16] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. In *In Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI*, pages 193–206, 2010.

[17] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.

[18] Saisanthosh Balakrishnan and Gurindar S. Sohi. Exploiting value locality in physical register files. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 265–, Washington, DC, USA, 2003. IEEE Computer Society.

[19] Kwabena Boahen. *Brains in Silicon*. http://www.stanford.edu/group/brainsinsilicon/index.html.

[20] Eric A. Brewer. High-level optimization via automated statistical modeling. *SIGPLAN Not.*, 30:80–91, August 1995.

[21] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '03, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society.

[22] Prashanth P. Bungale and Chi-Keung Luk. Pinos: a programmable framework for whole-system dynamic instrumentation. In *Proceedings of the 3rd international conference on Virtual execution environments*, VEE '07, pages 137–147, New York, NY, USA, 2007. ACM.

[23] Zizhong Chen, Jack Dongarra, Piotr Luszczek, and Kenneth Roche. Self-adapting software for numerical linear algebra and lapack for clusters. *Parallel Comput.*, 29:1723–1743, November 2003.

[24] Lawrence Chisvin and R. James Duckworth. Content-addressable and associative memory: Alternatives to the ubiquitous ram. *Computer*, 22:51–64, July 1989.

[25] Javier Cuenca, Domingo Giménez, and José González. Architecture of an automatically tuned linear algebra library. *Parallel Comput.*, 30:187–210, February 2004.

[26] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press.

[27] Michie Donald. "memo" functions and machine learning. *Nature*, 218:19–22, April 1968.

[28] J. Dongarra, G. Bosilca, Z. Chen, V. Eijkhout, G. E. Fagg, E. Fuentes, J. Langou, P. Luszczek, J. Pjesivac-Grbovic, K. Seymour, H. You, and S. S. Vadhiyar. Self-adapting numerical software (sans) effort. *IBM J. Res. Dev.*, 50:223–238, March 2006.

[29] Jack Dongarra and Victor Eijkhout. Self-adapting numerical software and automatic tuning of heuristics. In *Proceedings of the 2003 international conference on Computational science*, ICCS'03, pages 759–767, Berlin, Heidelberg, 2003. Springer-Verlag.

[30] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *In Proceedings of the 2002 Symposium on Operating Systems Design and Implementation*, pages 211–224, 2002.

[31] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *In Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI*, pages 87–100, 1999.

[32] Archana Ganapathi, Kaushik Datta, Armando Fox, and David Patterson. A case for machine learning to optimize multicore performance. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, HotPar'09, pages 1–1, Berkeley, CA, USA, 2009. USENIX Association.

[33] Al Geist and Robert Lucas. Major computer science challenges at exascale. *Int. J. High Perform. Comput. Appl.*, 23:427–436, November 2009.

[34] Dileep George. How the brain might work: A hierarchical and temporal model for learning and recognition [ph.d]. In *Stanford University*, 2008.

[35] Dileep George and Jeff Hawkins. Towards a mathematical theory of cortical micro-circuits. *PLoS Comput Biol*, 5(10):e1000532, 10 2009.

[36] Marco Giunti. *Computation, dynamics, and cognition*. Oxford University Press, Inc., New York, NY, USA, 1997.

[37] A. Grbic, S. Brown, S. Caranci, R. Grindley, M. Gusat, G. Lemieux, K. Loveless, N. Manjikian, S. Srbljic, M. Stumm, Z. Vranesic, and Z. Zilic. Design and implementation of the numachine multiprocessor. In *DAC '98: Proceedings of the 35th annual Design Automation Conference*, pages 66–69, New York, NY, USA, 1998. ACM.

[38] Mads Haahr. *RANDOM.ORG - True Random Number Service*. http://www.random.org.

[39] Mohamad H. Hassoun, editor. *Associative neural memories*. Oxford University Press, Inc., New York, NY, USA, 1993.

[40] Kim Hazelwood. *Code Cache Management in Dynamic Optimization Systems*. PhD thesis, Harvard University, 2004.

[41] Robert Hecht-Nielsen. Neural networks letter: Cogent confabulation. *Neural Netw.*, 18(2):111–115, 2005.

[42] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Comput.*, 18(7):1527–1554, 2006.

[43] HP. *Press Release: HP Collaborates with Hynix to Bring the Memristor to Market in Next-generation Memory*. http://www.hp.com/hpinfo/newsroom/press/2010/100831c.html, Aug 2010.

[44] Curtis L. Janssen, Helgi Adalsteinsson, Scott Cranford, Joseph P. Kenny, Ali Pinar, David A. Evensky, and Jackson Mayo. A simulator for large-scale parallel architectures. *International Journal of Parallel and Distributed Systems*, 1(2):57–73, 2010.

[45] Hawkins Jeffery. *On Intelligence*. Times Books, 2004.

[46] Yunlian Jiang, Eddy Z. Zhang, Kai Tian, Feng Mao, Malcom Gethers, Xipeng Shen, and Yaoqing Gao. Exploiting statistical correlations for proactive prediction of program behaviors. In *CGO '10: Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 248–256, New York, NY, USA, 2010. ACM.

[47] IBM journal of Research and Development staff. Overview of the IBM Blue Gene/P project. *IBM J. Res. Dev.*, 52(1/2):199–220, 2008.

[48] Jinpyo Kim, Wei-Chung Hsu, Pen-Chung Yew, Sreekumar R. Nair, and Robert Y. Geva. Entropy-based profile characterization and classification for automatic profile management. In *Asia-Pacific Computer Systems Architecture Conference*, pages 40–51, 2007.

[49] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 1–1, Berkeley, CA, USA, 2005. USENIX Association.

[50] T. Kohonen. *Self-organization and associative memory: 3rd edition*. Springer-Verlag New York, Inc., New York, NY, USA, 1989.

[51] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, Amos Waterland, and Volkmar Uhlig. K42: Building a complete operating system. In *Proc. of the First European Systems Conference*, Leuven, Belgium, April 2006.

[52] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, and R. Brightwell. Palacios and kitten: New high performance operating systems for scalable virtualized and native supercomputing. In *IPDPS '10: Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium*, 2010.

[53] Kevin P. Lawton. Bochs: A portable pc emulator for unix/x. *Linux J.*, page 7.

[54] Geoffrey Lefebvre, Brendan Cully, Michael J. Feeley, Norman C. Hutchinson, and Andrew Warfield. Tralfamadore: unifying source code and execution experience. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, pages 199–204, New York, NY, USA, 2009. ACM.

[55] Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. The hpc challenge (hpcc) benchmark suite. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 213, New York, NY, USA, 2006. ACM.

[56] LaPedus Mark. *HP and Hynix to commercialize the memristor*. http://www.eetimes.com/electronics-news/4207222/HP--Hynix-move-to-commercialize-the-memristor-semiconductor, Aug 2010.

[57] Carver Mead. Neuromorphic electronic systems. In *Proc. IEEE, 78:16291636*, 1990.

[58] Andreas Moshovos, Dionisios N. Pnevmatikatos, and Amirali Baniasadi. Slice-processors: an implementation of operation-based prediction. In *Proceedings of the 15th international conference on Supercomputing*, ICS '01, pages 321–334, New York, NY, USA, 2001. ACM.

[59] Todd Mytkowicz, Amer Diwan, and Elizabeth Bradley. Computer systems are dynamical systems. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 19(3):033124, 2009.

[60] Philippe O. Pouliquen, Andreas G. Andreou, and Kim Strohbehn. Winner-takes-all associative memory: A hamming distance vector quantizer. *Analog Integr. Circuits Signal Process.*, 13:211–222, May 1997.

[61] Gunda Pradeep, Ravindranath Lenin, Thekkath Chandramohan, Yu Yuan, and Zhuang Li. Nectar: Automatic management of data and computation in datacenters. In *In Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI*, pages 75–88, 2010.

[62] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Stephen A. Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM Trans. Model. Comput. Simul.*, 7:78–103, January 1997.

[63] Rahul Sarpeshkar. *RLE - Analog VLSI and Biological Systems Group - ULTRA LOW POWER BIOELECTRONICS*. http://www.rle.mit.edu/avbs/.

[64] Cosma Rohilla Shalizi and Kristina Lisa Klinkner. Blind construction of optimal nonlinear recursive predictors for discrete sequences. In Max Chickering and Joseph Y. Halpern, editors, *Uncertainty in Artificial Intelligence: Proceedings of the Twentieth Conference (UAI 2004)*, pages 504–511, Arlington, Virginia, 2004. AUAI Press.

[65] Cosma Rohilla Shalizi, Kristina Lisa Shalizi, and James P. Crutchfield. An algorithm for pattern discovery in time series. *CoRR*, cs.LG/0210025, 2002.

[66] M A Sivilotti, M R Emerling, and C A Mead. VLSI architectures for implementation of neural networks. In *AIP Conference Proceedings 151 on Neural Networks for Computing*, pages 408–413, Woodbury, NY, USA, 1987. American Institute of Physics Inc.

[67] Avinash Sodani and Gurindar S. Sohi. Dynamic instruction reuse. In *Proceedings of the 24th annual international symposium on Computer architecture*, ISCA '97, pages 194–205, New York, NY, USA, 1997. ACM.

[68] Gurindar S. Sohi and Amir Roth. Speculative multithreaded processors. *Computer*, 34(4):66–73, 2001.

[69] Ram Srinivasan, Jeanine Cook, and Olaf Lubeck. Ultra-fast cpu performance prediction: Extending the monte carlo approach. *Computer Architecture and High Performance Computing, Symposium on*, 0:107–116, 2006.

[70] Ram Srinivasan, Eitan Frachtenberg, Olaf Lubeck, Scott Pakin, and Jeanine Cook. An Idealistic Neuro-PPM Branch Predictor. *The Journal of Instruction-Level Parallelism*, 9, May 2007.

[71] DARPA/IPTO Study. *ExaScale Computing Software Study: Software Challenges in Extreme Scale Systems*. http://users.ece.gatech.edu/mrichard/ExascaleComputingStudyReports/ECSS%20report%20101909.pdf, Sept 2009.

[72] DARPA/IPTO Study. *ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems*. http://users.ece.gatech.edu/mrichard/ExascaleComputingStudyReports/exascale_final_report_100208.pdf, Sept 2009.

[73] Nathan Thomas, Gabriel Tanase, Olga Tkachyshyn, Jack Perdue, Nancy M. Amato, and Lawrence Rauchwerger. A framework for adaptive algorithm selection in STAPL. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pages 277–288, New York, NY, USA, 2005. ACM.

[74] Kai Tian, Yunlian Jiang, Eddy Z. Zhang, and Xipeng Shen. An input-centric paradigm for program dynamic optimizations. In *OOPSLA '10: Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 125–139, New York, NY, USA, 2010. ACM.

[75] Amos Waterland. Personal communication.

[76] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. Mining console logs for large-scale system problem detection. In *SysML'08: Proceedings of the Third conference on Tackling computer systems problems with machine learning techniques*, pages 4–4, Berkeley, CA, USA, 2008. USENIX Association.