Programmable Smart Machines

Technical Report BUCS-TR-2012-007

Jonathan Appavoo
Boston University
jappavoo@bu.edu

Amos Waterland
Harvard University
apw@seas.harvard.edu

Dan Schatzberg
Boston University
dschatz@bu.edu

Dramatic challenges face us in advancing traditional computer systems. With the demise of linear increases in clock speeds, we have been driven to entertain parallelism even in general purpose systems. Parallelism leads to complexity and works against the simple abstractions that isolate programmers from the details of the system architecture, giving rise to the "great software challenge". We believe that we are at a turning point in computing and that more aggressive and novel systems need to be explored. Perhaps there are alternative solutions to the challenge lurking outside of the traditional approaches.

Machine learning and brain inspired mechanisms that exploit parallelism in a very different way, while nowhere close to the maturity of traditional logic based systems, are slowly improving. We believe that systems research has an important role to play in exploring the future of how these mechanisms can be made relevant in a general purpose programmable way. We believe that they may provide avenues for addressing current challenges and enabling completely new function. In this paper we discuss one possible avenue for concretely exploring a futuristic machine model that exploits the general ability to learn, through the counting, correlating, and memorizing of occurrences of events, to automatically fast-forward a programmable computer. This is long term and high risk systems research. We hope that you find it thought provoking and assumption challenging.

There seems to be two basic ways of computing: 1) by logic, and 2) by pattern recognition. This work seeks to explore a hybrid system model that combines the two.

A key attribute of logic based computers is programmability. While it may require care, effort and creativity, prescribing what a computer should compute, via the construction of a program, is a doable task attested to by the amount of software constructed and in use. Programmability and the ability to treat programs as artifacts, in the form of software, that can be replicated, exchanged, extended and composed has proven incredibly powerful. The ability to fabricate the basic logic operations required for a programmable computer from generic manufacturable electronic circuits to produce a system whose function can be easily specified, modified and updated via software installation, post physical construction, has enabled the mass production and adoption of computers.

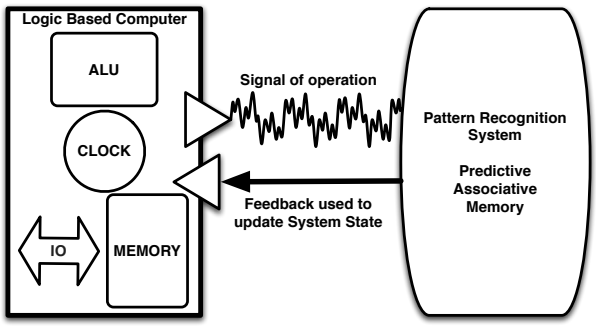


Figure 1: Abstract Model

On the other hand while the computational ability of pattern recognition is not obviously programmable, it possesses automatic abilities that we tend to categorize as intelligent or smart despite also considering it to be approximate and imprecise. In particular, pattern recognition has the ability to automatically adapt and improve computational performance based on experience – learn; the ability to integrate and exploit knowledge learnt across a broad temporal span – from inception to immediate context; the ability to generalize and automatically speculate to aid in computation. Additionally, the biological realization implies that pattern recognition based computation can be implemented robustly, efficiently and in a scalable way.

Is there a way of relating these two mechanisms and combining them into a single system that integrates and leverages strengths of both? In particular, can we conceive of a machine model that is both programmable and yet inherently possesses automatic abilities that we associate with smart computation? In this paper we frame this question in the form of a specific hypothesized hybrid execution and associated system model.

The execution model attempts to convert the idea of hot-path and hot-data optimization into an automated system feature where the observed statistics of the systems operation lead to the memorization of side effects, associated with hot computation, in the form of system state changes, that can be recalled and predicted.

1 Conjecture

A system can be constructed that exploits the general ability to learn through the counting, correlating, and memorizing of occurrences of events, to fast-forward a programmable computer.

Specifically, we conjecture that statistically significant patterns exist in the low-level operation of a logic based computer system and that these patterns carry sufficient information that can permit the system's future state (results) to be automatically generated via the recall and predictions of a pattern recognition system. This conjecture suggests a hybrid programmable smart machine (PSM) that transparently mixes both kinds of computation into a single system.

Figure 1 illustrates our basic abstract hybrid model. A *signal* is generated from the operation of a logic based computer. The signal should capture all low-level aspects of execution that affect the systems state. An example would be a time-stamped trace of all opcodes executed along with operand values, interrupt occurrence, and io register values. This signal is interfaced to a signal based pattern recognition system that acts as a predictive associative memory system – extracting and learning patterns or features in the signal. The pattern recognition system continually produces feedback about the future behavior of the signal. The feedback is used to synthesize updates to the logic based computer's state. Thus, some equivalent portion of the operations of the logic based computer is performed by the pattern recognition system. In this way the state and associated computation is progressed though a mixture of both systems. Given the exposure to more and more computation, the feedback from the associative memory system progressively improves and adapts such that the fraction of computation that it can precisely recall increases.

Abstractly, one can think of the pattern recognition system like a global hash table or database in which the immediate state associated with the logical computer is a lookup or query. A result from the query 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 pattern recognition system has no useful knowledge yet of the current state. If the set has one state then it knows that the current state will result in exactly one future state and the logical system can be updated directly to its state. If the set is composed of multiple states then the lookup result is a prediction of a possible set of future states of the logical system.

In order to proceed in a more concrete manner we detail our hypothesized execution model.

2 Signal Driven System Memoization

Our goal is a system that automatically eliminates increasingly larger fractions of 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.

We hypothesize that a functional trace of a CPU, containing opcodes along with operands and interrupts, statistically 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.

A key to the approach is that the execution model can lead to a system model in which a 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*[8] that could use *neuromorphic*[18] 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 memory and statistical processing.

Michie introduced memoization in his 1968 paper[8] 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 or recalled from memory but in either case the result is the same. He states that evaluation on a computer should transparently blend the two. He describes how a programmer can interpose a "memo" operator infront of certain kinds of functions. The memo operator wraps all calls to the function, caching input parameters to function results, and uses the cache to avoid function calls when pos-

sible. 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[16]. 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 for system-level automation. The use of a restricted functional language or restrictions on the types of computations does not seem feasible. Also, the use of a complex language runtime would impose base performance penalties and rule out low-level expert programmer optimization. It is also not obvious how to integrate or manage a language level cache in a scalable way across all applications.

Combining the idea of a memoization execution model, that mixes recall with computation, and a system-centric, low-level, statistically driven approach, we are lead to suggest an alternative model. Specifically, a signal processing generalization that operates transparently on the low-level signal of a system's operation and state. 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 cached information.

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 there is sufficient regular structure in general execution, such that caching portions of computation will be possible and prove useful, the associative memory and its functionality have to be efficient and viable at large scales. A 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"[14], 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[6], MIT[17], Johns Hopkins[3], and the DARPA sponsored SyNAPSE Project[2]. 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 [7, 11, 9, 4].

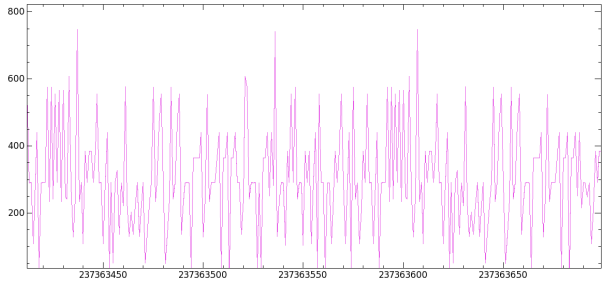
As a concrete example, consider the neuromorphic associative memory chip implemented by Pouliquen et al.[15]. The authors demonstrate that their chip is capable of performing an associative recall using approximately $100nJ$, or $1/6000$ of the $600\mu J$ consumed by a traditional processor performing the same function. While challenges still exist for commercially viable, general-purpose neuromophic associative memory chips, announced products, such as the Lyric GP5[1] Probability Processor and memresistor chips[10, 13], 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.

We have and are continuing to construct a prototype system level simulator to help us explore these ideas. We provide a brief intuition for the ideas using data from the simulator and then provide some context for the development of the ideas in the remainder of this section.

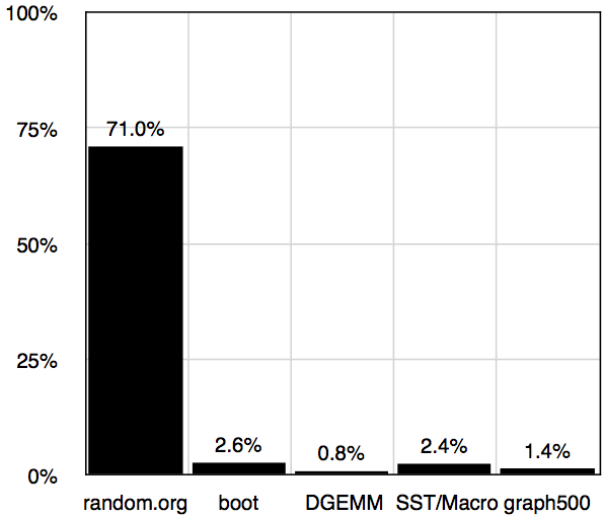
Our execution model aims to exploit patterns, or more formally redundancy, in computer execution. A system will update its state by mixing traditional program execution and associative memory recall. From this perspective, the effectiveness of such a hybrid system depends on the amount of redundancy 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 computer running Linux for some workloads. A small portion of such a signal is depicted in Figure 2a. 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, to compress the y value recordings and compare the size to the original.

Figure 2b, 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 computer and let it boot into a



(2a) portion of computer operation signal

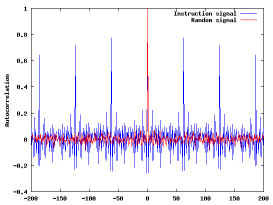


(2b) gzip compression

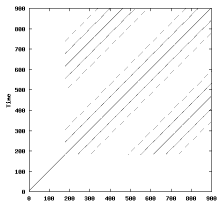
Figure 2

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 a core routine of a standard linear algebra library(with three 200×200 matrices of random values). In the third experiment, we run a test application of a large network discrete event simulation. In the final experiment we run a graph algorithm benchmark. 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 $100,000$ random event data file generated from *random.org* 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.

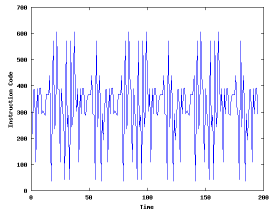
Figure 3 are an example of how automatic signal level analysis can reveal structure in execution in a semantic free fashion. The details of the mechanisms are beyond the scope of this paper. Figure 3a shows the autocorrelation of the first 1000 events in the boot execution trace. The regular spikes in the plot imply that there is some repeating structure in windows of 100 events.



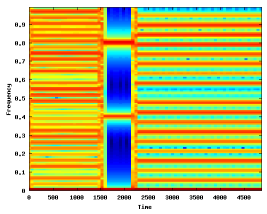
(3a) autocorrelation



(3b) reoccurrence



(3c) signal



(3d) spectrogram

Figure 3: Plots from Boot Experiment

Figure 3b we show a recurrence plot in which we analyze subsequences of length 100. The regular diagonal lines confirm that there are reoccurring subsequences of greater than length 100. Moreover the plot can be used to guide us to look for such a subsequence beginning at about time tick 179. Figure 3c shows a raw plot of 200 ticks of the execution trace starting at time tick 179 and in deed it is visually clear that a subsequence is repeating. In Figure 3d we show the spectrogram of the first 10,000 events. The banded structure suggests that we may someday accomplish phase detection, predictor training reinitialization, and loop characteristics via frequency based analysis. Such analysis suggests that statistical mechanisms on execution traces may be able to identify structure at various scales using simple signal based counting mechanisms.

Many areas of computer science try to consider and exploit the statistical structure in runtime behavior. Given space constraints we do not discuss all of them. Influenced by the trace cache methods of Dynamo[5], related work and our own experience working with SIMOS, we started considering such forms of instruction tracing and caching for system wide optimization. However, we quickly found that the main focus of the dynamic compilation 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. Such that 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.

The renewed interest in virtualization has brought with it interest in exploiting tracing and logging for various functions from debugging and intrusion detection to deterministic computation 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[12]. 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 in optimizing system behavior. In particular, system metrics are used 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.

When exploring the trace based approaches in the context of entire, cross-layer, systems optimization, a key conjecture was formed. Namely, that a computer's entire operation could be treated as a low-level unified temporal execution signal (ES). An ES can be 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 process. 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.

Execution Signals provide us with a concrete signal processing interpretation of execution. This gives us a context for recasting memoization and integrating neuromorphic devices into a hybrid system model. The details of the model and our associated simulator are beyond the scope of this paper. We are using the model and simulator to start quantitatively evaluating our conjecture.

3 Summary

In this paper we conjecture that a system can be constructed that exploits the general ability to learn through the counting, correlating, and memorizing of occurrences of events to fast-forward a programmable computer. In particular, we propose a signal based interpretation of a computer's execution that can be used to implement a form of system state memoization using a predictive associative memory. Such an approach may some day lead to a system that can utilize both traditional logic and neuromorphic or other biologically inspired mechanisms to be both programmable and smart.

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.

References

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

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

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

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

[5] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*.

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

[7] Lawrence Chisvin and R. James Duckworth. Content-addressable and associative memory: Alternatives to the ubiquitous ram. *Computer*.

[8] Michie Donald. "memo" functions and machine learning. *Nature*.

[9] Mohamad H. Hassoun, editor. *Associative neural memories*. 1993.

[10] 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.

[11] T. Kohonen. *Self-organization and associative memory: 3rd edition*. 1989.

[12] Geoffrey Lefebvre, Brendan Cully, Michael J. Feeley, Norman C. Hutchinson, and Andrew Warfield. Tralfamadore: unifying source code and execution experience. In *EuroSys*, 2009.

[13] 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.

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

[15] Philippe O. Pouliquen, Andreas G. Andreou, and Kim Strohbehn. Winner-takes-all associative memory: A hamming distance vector quantizer. *Analog Integr. Circuits Signal Process.*

[16] Gunda Pradeep, Ravindranath Lenin, Thekkath Chandramohan, Yu Yuan, and Zhuang Li. Nectar: Automatic management of data and computation in datacenters. In *OSDI*, 2010.

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

[18] M A Sivilotti, M R Emerling, and C A Mead. VLSI architectures for implementation of neural networks. In *Neural Networks for Computing*, 1987.