An Immediate Concurrent Execution (ICE) Abstraction Proposal for Many-Cores

U. Vishkin
The University of Maryland Institute for Advanced Computer Studies (UMIACS) and Electrical and Computer Engineering Department

Abstract

Settling on a simple abstraction that programmers aim at, and hardware and software systems people enable and support, is an important step towards convergence to a robust many-core platform.

The current paper: (i) advocates incorporating a quest for the simplest possible abstraction in the debate on the future of many-core computers, (ii) suggests "immediate concurrent execution (ICE)" as a new abstraction, and (iii) argues that an XMT architecture is one possible demonstration of ICE providing an easy-to-program general-purpose many-core platform.

1. Case for Abstraction

In 2004, standard (desktop) computers comprised one processor core. In 2008, some have 8 cores. By 2012, 64-core computers (another factor of 8) are expected. Transition from serial computing to parallel computing mandates the reinvention of the very heart of computer science (CS). These highly parallel computers need to be built and programmed in a new way. Current solutions by leading vendors do not scale to tens of cores. Given that clock speeds have not been improving for quite a few years, the use of parallel processing for improving single-program completion time is a critical target for future designs. We need to figure out how to build scalable many-core computers, how to program them effectively so that programmers can get strong performance with minimal programming effort, how to train the workforce, and how to teach this new environment at all levels, including introductory programming courses to college freshmen and K-12 students.

Foremost among current challenges is timely ***convergence*** *to a robust many-core platform that will serve the world for many years to come*. Critical to the economy and workforce, the basic motivation behind the current position paper is bringing about the reinvention of CS for meeting this challenge: 1) Andy Grove (Intel) noted that the *software spiral* (hardware improvements lead to software improvements that lead back to hardware improvements) had been an engine of sustained growth for IT; but (as explained in [6] and since convergence is yet to happen), it is now broken! 2) Both under-trained and mis-trained for a future certain to be dominated by parallelism, most CS students only study the old serial paradigm, acquiring serial habits that complicate later transition to parallelism. But, how should we approach the convergence challenge, and, in particular, what the first step should be.

One of the dictionary definitions of *abstract* is *difficult to understand*, or *abstruse*. In CS, however, abstraction has become synonym with the quest for simplicity. Interestingly, the word *abstraction* in Hebrew shares the same root with *simple* (as well as *undress* and *expand*).

The final posting in a special series on why research advances are needed to overcome the problems posed by mulitcore processors on the Computing Community Consortium blog [5] perhaps implies a perception of despair in the community. The problem is not new. Many parallel computer architectures have been proposed and built over the last 40 years, but with limited success. Exploiting the parallelism present in them has often eluded their users. The main source of encouragement in [5] is a call on all involved communities to *collaboratively* start with a clean slate, rather than have language researchers locked into mechanisms supported by commodity hardware and hardware researchers locked into fully supporting any current software.

This is not the first time that CS is facing a complex system problem requiring a solution that involves many different players and should be robust over time in the face of system upgrades. It has become a signature intellectual success story of CS to address such problems by figuring out a simple abstraction that acts as "a single nail holding everything together". In fact, abstractions that present the user with a *virtual machine* that is easier to understand and program than the underlying hardware, but still allows effective use of the hardware, facilitated significant Computer Science accomplishments. Broad consensus built around these simple abstractions was the key. Some formative abstractions were: (i) *that any single instruction available for execution in a serial program executes immediately*, henceforth called *immediate serial execution (ISE)*; note that since an instruction may apply to any location in memory, ISE extended another formative abstraction that we call *"immediate memory access (IMA)"*: *that any particular word of an indefinitely large memory is immediately available*, and (ii) *that a computer is serving the task that the user is currently working on* ***exclusively,*** henceforth *exclusive computer availability (ECA)*.

The IMA abstraction abstracts away a hierarchy of memories, each with greater capacity, but slower access time, than the preceding one, and the ISE abstraction extends it to immediate execution of any operation. The ECA abstraction abstracts away virtual file systems that can be implemented in local storage or a local or global network, access to the Internet, and other tasks that may be concurrently using the same computer system. These abstractions have improved the productivity of programmers and other users, and contributed towards broadening participation in computing.

Some simple and robust abstraction can be the first writing on the clean slate sought in [5]. We will then need to build a consensus around such an abstraction as a way to reproduce past CS success stories for the many-core era. Finding the best many-core platform requires a battle of ideas whose outcome will affect a rather broad community. The need for acceptance by all relevant segments of the community suggests the necessity of benchmarks for predicting the success of a many-core platform. Development of such benchmarks is, in fact, long overdue. Abstractions provide an effective way for lowering the bar towards broadening participation in the debate to all relevant participants. While the utility of abstraction will become much clearer once such benchmarks are available, there is no reason not to focus on abstractions immediately.

The desired abstraction will: (i) be simple, hiding the details of the underlying hardware, (ii) be accessible to the broadest possible groups of users, (iii) allow strong speedups for applications, (iv) be scalable; a user of a 16-core computer should rely on the same abstraction as a user of a future generation 1024-core computer, or else performance code will have to be continuously rewritten; this will also help put the above noted software spiral back on track; (v) extend, rather than replace, existing (successful) abstractions; in particular, when code provides no parallelism, the user will need to be able to fall back on the serial abstraction ISE; and last, but definitely not least, (vi) be buildable; we must be able to build an actual computer system that provides good performance for users that rely on the abstraction. Note also that the ECA abstraction does not require change.

Ideally one would desire an indefinitely large memory capacity such that any particular ... word would be immediately available ... We are ... forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible. For historical context consider the above quote from [1], one of the most formative efforts in the history of the field. The quote reflects a tension between a desired abstraction and physical realization. Six decades later, the verdict on how this tension was resolved is clear. As imperfect as this abstraction is, mainstream CS holds that the abstraction won. A prevailing working assumption for nearly every computer scientist is the IMA abstraction (as well as the more general ISE abstraction). Consider those computer system and compiler professionals whose important work requires accounting for the memory hierarchy in order to mitigate its gaps with the IMA abstraction. These people actually labor to support this abstraction so that programmers can incorporate it in their programming model and improve their productivity. When it comes to the IMA abstraction, the only exception to all those who either abide by the abstraction or serve it is the relatively few who are seek to get the most out of the memory hierarchy for their application, by avoiding the IMA abstraction.

2. Our ICE abstraction candidate

The candidate abstraction proposed is*: That an indefinitely large number of instructions available for concurrent execution executes immediately*, and dub it *immediate concurrent execution (ICE)*. A step-by-step explication of the instructions that are available next for concurrent execution requires the lowest level of cognition relative to all current parallel programming models, is independent of the number of processors, and falls back on ISE, in case of one instruction per step. The embodiment below reinforces relative simplicity and ease-of-programming, while addressing speedups and implementation.

3. An embodiment of the ICE abstraction

We finally argue that we have already made significant progress towards an “XMT/PRAM” embodiment of the ICE abstraction. Addressing all the six properties above, different parts of the embodiment are at different maturity stages. The main components in this embodiment are: (i) The well-known PRAM parallel algorithmic approach that has never been seriously challenged on ease of thinking, or wealth of knowledge-base; PRAM algorithms are essentially prescribed as (a) a sequence of rounds, and (b) for each round, up to *p* processors can execute concurrently; where *p* is the number of processors assumed. The objective is minimizing the number of rounds. (ii) The Work-Depth methodology (due to [4]) suggests that the objective could be simpler. The parallel algorithm can be prescribed as (a) a sequence of rounds, and (b) for each round, any number of operations can be executed concurrently assuming unlimited hardware. The total number of operations is called "work" and the number of rounds is called "depth". The objective is reducing work and depth. The methodology of restricting attention only to work and depth has, in fact, been used as the main framework for the presentation of PRAM algorithms in texts such as [2,3]; see also the class notes available through [9]. By way of the Work-Depth methodology, *the PRAM provides a direct embodiment of the ICE abstraction*, and is a simple natural extension to serial algorithms as the ICE parallel abstraction generalizes the ISE serial abstraction. (iii) A very fine-grained, irregular general-purpose on-chip parallel computing platform, optimizing single-program completion time; developed as well as hardware and software prototyped at UMD, the platform comprises a so-called eXplicit Multi-Threaded (XMT) architecture that scales to 1000 processors on-chip and can be programmed using a PRAM-like (actually work-depth like) programming language XMTC [9]. XMTC is a modest extension of the language C augmented with a *spawn* command, and only one other command. And (iv) a “back-end” performance model [7] that is closer to hardware constraints, both as a compiler target and for coders seeking performance beyond ICE.

Advantages of the embodiment include: 1) a methodology for PRAM-like parallel programming that, as explained above, reflects the ICE abstraction, freeing the programmer from the need to first decompose the problem as typically required by other parallel programming approaches; recall that parallel programming difficulties have failed all general-purpose parallel systems to date by limiting their use. 2) XMT is, in fact, a practical implementation of the ICE abstraction. 3) XMT is comprehensive and coherent. It accounts for application programming (VHDL/Verilog, OpenGL, MATLAB, etc), parallel algorithms, parallel programming, compiling, architecture, power, deep-submicron implementation, and backward compatibility on serial code. 4) The approach goes after any type of application parallelism regardless of its amount, regularity, or grain size and is amenable to multiprogramming. 5) We have demonstrated its feasibility through hardware and software prototyping. We also demonstrated good performance, programmability and teachability. Highlights include: evidence of 100X speedups on general-purpose applications (on a simulator of 1000 on-chip processors), a 64-processor, 75MHz XMT FPGA-based computer [8], 90nm ASIC tape-outs, basic yet stable compiler, and a class tested programming methodology where college freshmen and even high-school students are taught only parallel algorithms and then self-study XMT programming. 6) Just released XMTC compiler and cycle-accurate simulator of XMT that can be downloaded to any standard desktop computing platform. This software release is available through the XMT home page, or sourceforge.net [9].

4. Conclusion

The memory hierarchy had been a challenge for serial computing, and the IMA abstraction addressed that. The fact that hardware needs more time to execute some operations than others was yet another challenge and the ISE abstraction extended IMA to address that, but ISE is not good enough to address the world of parallel hardware. Abstractions have played an important role in parallel computing, e.g., in parallel programming models. They should play a similar key role for many-cores, as well. Our main point is that the ICE abstraction, coupled with an XMT/PRAM platform (or perhaps some other embodiment in the future) provide a viable option for the many-core era

The community should engage in assessing candidate abstractions, and establish a merit-based process for a healthy competition among them. Consensus built around the abstraction that will be selected will go a long way towards convergence to a many-core platform that will put back on track the software spiral and reconnect the training of CS students with the future needs of the field.

Acknowledgement

Helpful comments by G. Caragea, M. Olano, A. Schulman, A. Tzannes and A. Varshney are gratefully acknowledged.

References

[1] A.W. Burks, H.H. Goldstine, and J. Von Neumann, Preliminary Discussion of the Logical Design of an Electronic Computer Instrument (1946).

[2] J. JaJa, An Introduction to Parallel Algorithms, Addison-Wesley, 1992.

[3] J. Keller, C.W. Kessler and J.L. Traeff. Practical PRAM Programming. Wiley-Interscience, 2001.

[4] Y. Shiloach and U. Vishkin. An O((n**2)log n) parallel max-flow algorithm. J. of Algorithms 3 (1982), 128--146.

[5] M. Snir, Multi-core and parallel programming: Is the sky falling? The Computing Community Consortium Blog, http://www.cccblog.org/2008/11/17/multi-core-and-parallel-programming-is-the-sky-falling/

[6] H. Sutter, The free lunch is over – A fundamental shift towards concurrency in software, Dr. Dobbs Journal 30 (3), March 2005.

[7] U. Vishkin, G. Caragea and B. Lee. Models for Advancing PRAM and Other Algorithms into Parallel Programs for a PRAM-On-Chip Platform. In Handbook on Parallel Computing: Models, Algorithms, and Applications (Eds S. Rajasekaran and J. Reif), Chapman and Hall/CRC Press, 2008.

[8] X. Wen and U. Vishkin. FPGA-based prototype of a PRAM-on-chip processor, ACM Computing Frontiers, Ischia, Italy, May 5-7, 2008.

[9] Explicit Multi-Threading (XMT): home page http://www.umiacs.umd.edu/users/vishkin/XMT/ and software release http://sourceforge.net/projects/xmtc/