A Multicore-aware Von Neumann Programming Model

János Végh[1], Zsolt Bagoly[2], Ádám Kicsák[2] and Péter Molnár[2]
[1]*Faculty of Informatics, University of Debrecen, Kassai Str 26, Debrecen, Hungary*
[2]*PhD School of Informatics, University of Debrecen, Kassai Str 26, Debrecen, Hungary*

Keywords: Modeling, Computer Architecture, Computing Paradigm, Multicore, Von Neumann, Performance, Parallel.

Abstract: An extension to the classic von Neumann paradigms is suggested, which –from the point of view of chip designers– considers modern many-core processors, and –from the point of view of programmers– still remains the classic von Neumann programming model. The work is based on the ideas that 1) the order in which the instructions (and/or code blocks) are executed does not matter, if some constraints do not force a special order of execution 2) a High Level Parallelism for code blocks (similar to Instruction Level Parallelism for instructions) can be introduced, allowing high-level out of order execution 3) discovering the possibilities for out of order execution can be done during compile time rather than runtime 4) the optimization possibilities discovered by the compile toolchain can be communicated to the processor in form of meta-information 5) the many computing resources (cores) can be assigned dynamically to machine instructions. It is shown that the multicore architectures could be transformed to a strongly enhanced single core processor. The key blocks of the proposal are a toolchain preparing the program code to run on many cores, a dispatch unit within the processor making effective use of the parallelized code, and also a much smarter communication method between the two key blocks is needed.

1 INTRODUCTION

In the forecasts given for even the farther future, sustained improvements in computer performance are always included, either implicitly or explicitly. A decade ago, however, it became clear that the computing performance cannot be increased any more through increasing clock frequency (Agarwal et al., 2000). Rather, the number of computing units started to rise. Recently, even this technology solution run into walls of "Dark Silicon" (Esmaeilzadeh et al., 2012).

The single-processor performance seems to be not able to follow the expectations, extrapolated from its historical trend a decade ago. The ratio of the missing computing performance will reach 100 around 2020 (Fuller and Millett, 2011), see Fig. 1. The warning signs reached the level of government and scientific advisory boards, both in the technology leader USA (Fuller and Millett, 2011) and Europe (S(o)OS project, 2010).

In case of attempting to use the cores in parallel, one faces the problems of ineffectivity of parallelization. At the beginning, at least recompiling is needed (even if you have auto-parallelizing compiler), and more typically manual parallelization must take place. It is especially hard to parallelize codes including foreign parts, like libraries. In addition, "*parallel programs . . . are notoriously difficult to write, test, analyze, debug, and verify, much more so than the sequential versions*" (Yang et al., 2014).

Although there exist for a while "look ahead" and "out-of-order" solutions in processors (with rather poor performance with respect to using the cores effectively, for example (Intel, 1998)), the present efforts concentrate mainly on exa-scale computing, believing that the "brute force" method, although in a very ineffective way, will solve the problem. The time and efforts needed to let the many processors cooperate, result in most cases in insignificant gain only. The efforts resulted in decreasing electric power consumption (also in absolute measure), improving computational efficiency and providing closer ways for cooperating between the cores, and even partly reconfigurable processors appeared allowing for open source hardware design for the end-user (Xilinx, 2012; Altera, 2013; Adapteva, 2014). Also, the possible high-level support is intensively searched (for examples see (HLPGPU workshop, 2012; World Scientific, 2014)), apparently with no breakthrough ideas.

Végh J., Bagoly Z., Kicsák Á. and Molnár P..
A Multicore-aware Von Neumann Programming Model.
DOI: 10.5220/0005097001500155
In *Proceedings of the 9th International Conference on Software Engineering and Applications* (ICSOFT-EA-2014), pages 150-155
ISBN: 978-989-758-036-9

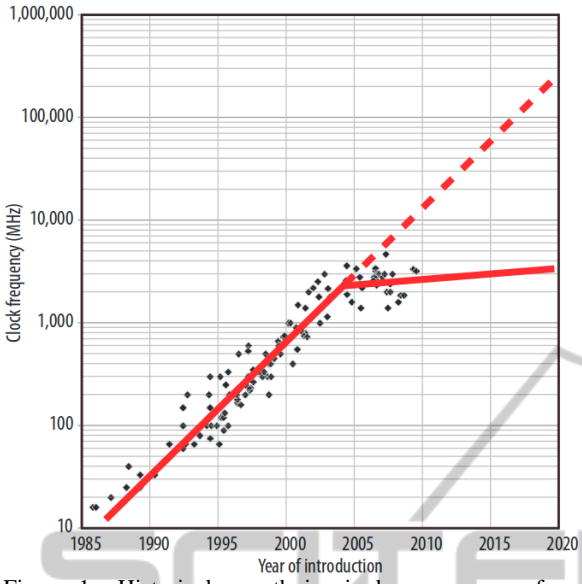

Figure 1: Historical growth in single-processor performance and a forecast of processor performance to 2020, based on the ITRS roadmap. A dashed line represents expectations if single-processor performance had continued its historical trend (Fuller and Millett, 2011).

The technology today allows to implement literally hundreds of cores in a single chip (for example, (Adapteva, 2014) is delivered recently with 64 cores, but its architecture is prepared for 4096 cores, Intel also developed chips with dozens of cores, etc.). However, as summarized on the project homepage (S(o)OS project, 2010): "Processor and network architectures are making rapid progress with more and more cores being integrated into single processors and more and more machines getting connected with increasing bandwidth. Processors become heterogeneous and reconfigurable ...***No current programming model is able to cope with this development, though, as they essentially still follow the classical van Neumann model***."

2 THE CLASSICAL VON NEUMANN MODEL – IN LESS CLASSICAL TERMS

The primary reason of the missing performance is, that although the computing resources are available, the programming methodology is not able to use them properly. Really, the abstractions used presently (and are known for the masses of programmers and engineers, including chip designers, educated till recently) are based on easy to understand, but outdated and non-technical view of computing.

The overwhelming majority of the plethora of the today's code is written for sequential processors, by programmers having sequential programming in mind. Even the concurrent (sometimes erroneously called "parallel") programming deals with pieces of sequential code. So, *it would be of great importance to find a way which allows some automatic parallelization of single-thread programs*, in this way converting the existing single-thread programs to massively parallelized ones.

On the other hand, one absolutely needs some level of abstraction and also the new model must be compatible to some measure with the old one. In order to *preserve compatibility with the existing programming paradigms*, the best way would be to modify the interpretation of the existing abstractions in a way which allows taking into account the present-day achievements of technology. In the sections below, first some of the vN abstractions are re-formulated using a somewhat unusual terminology, which makes the changes to be introduced easier to understand. Following that, a new meaning will be introduced for those old abstractions, having in mind some of the technical developments. Finally, the principles of a possible implementation will be presented.

2.1 The Old Picture

As (Godfrey and Hendry, 1993) mentions, although "The von Neumann report contains a wealth of insight and analysis still not available elsewhere", the computer described by von Neumann (vN) "*was never built, and its architecture and design seem now to be forgotten*". Well, the todays processors are far from those ones in the ancient ages. But, *from programming point of view, we still can describe their operation using vN abstractions*. The programming model was constructed early and – because of compatibility – kept some features of the early primitive implementations. Unfortunately, both hardware and software developers use that old model.

2.2 Computing Resource

In the vN world, the *sole computing resource* was the processor. Since only the processor had such processing ability, every single bit had to pass through the processor, and all parts of the processor were busy with executing the current instruction. This feature excluded any chance for executing instructions in parallel, and *concluded the serial execution of instructions*. Combined with the *stored instructions* principle, the *fetch-decode-execute cycle* mode processing was needed. The *computing flaw* was built up from

elementary *machine instructions*, which were stored in some *addressable stores* (memory cells).

In order to reduce the length of the instruction code, a *simplified addressing* was introduced. In normal cases, the control unit only advanced that special register to the next address by an external hardware. In exceptional cases the address of the next instruction was concluded from the stored instruction itself (jumps, calls, returns). The same control unit can even direct the instruction pointer to an address, defined by some external hardware. No contradiction with the vN principles: it only required "proper sequencing of operations".

This feature lead to the abstraction *process*, in which the running program has the exclusive use of the processor and the memory (and actually, all of its resources). Notice, that *nothing changes if we take another physical processor or core* (even if we use different cores for the different machine instructions): we are using an abstract one. So, the abstraction "process" can be trivially extended to multi-core case.

2.3 Instruction Set

At abstraction level, the *instruction set* tells what the processor is able to do and *the processor* is its actual implemention, with all of its technical details. The abstraction *instruction* specifies only the input and output, and leaves the implementation for the engineers. (That was the intention of Neumann: "*we will base our considerations on a hypothetical element, which functions essentially like a vacuum tube . . . but which can be discussed as an isolated entity. . . .* ***After the conclusions of the preliminary discussion the elements will have to be reconsidered***." (Aspray, 1990)) So, the instruction execution could be accelerated, using any trick, as long as it does not interfere with the established mathematical model, constructed on the basis of the von Neumann architecture.

If there exists only one executing engine (one CPU), it is naturally implied that *that processor* will execute all instructions. In this simplified picture it is trivial, that the processor will be available for executing the next instruction only after finishing the current instruction. So, *the computing resource that can be assigned for executing an instruction, is always the same and is always busy.*

Let us consider the same situation differently. The control unit points to an address where it finds some machine instruction. *Before executing the instruction, the control unit allocates a computing resource from the pool and assigns it to the instruction.* Then instructs the allocated computing resource to execute the instruction, waits until the instruction gets terminated and releases the resource back to the pool. From this change nothing appears for the abstraction "process", it is only an internal business of the control unit, being not part of the abstraction. If only one such execution unit exists, it will be continuously allocated and then released, with no gain in performance. *Until the current instruction provides a result, the next instruction is waiting for either the input data or the resource, which is the result of the current instruction or the processor itself.* Notice too, that in this simple picture the computing unit will be ready for executing another instruction exactly at the time when the result is also provided, i.e. waiting for a result or for availability of a resource is the same action.

2.4 Timeliness of Execution

Generally it is believed that in the vN model the instructions must be executed one after the other, *in the order as the programmer wrote them.* However, *the vN model requires only that the control unit must assure a proper sequencing for executing the instructions* (Aspray, 1990). The control unit executes the instructions actually pointed out by its instruction pointer, one at a time, really provides the illusion that the instructions are executed according to some strict time sequence. However, *the instructions are executed by the processor in a simple order-of-appearance*. So, *one may consider the possibility to reorder the instructions in the object file without affecting the final result*.

In modern processors, the pipelining simply reuses parts of the processor logic through cutting instruction executing into stages, and so introducing overlapped execution of instructions, which, in strict sense, does not preserve the one after another method of execution. This method however preserves the original execution sequence. So does also the *out of order execution* in modern processors, because a lot of efforts are done to reorder the states after executing the instructions in out of their order (including conditional and forecasted execution). *Do we really need to preserve the original ordering of the instructions, if we are allowed to execute them in a different order?*

3 THE MULTICORE VON NEUMANN COMPUTING MODEL

Modern processors might have many computing resources, in close vicinity to each other, so it is time to re-consider, *whether the vN operating model really*

requires separating the instruction execution in time and keeping the semantic order (as they were written in the program), or those requirements were just concluded from the technical possibilities of the time of the conclusion. The new model is the result of rethinking the complete process, comprising code generation, transmission and execution.

3.1 Instruction Level Parallelism (ILP)

Since the beginning of the computer era, the need for computing performance rose much quicker than the hardware and especially the software technology could provide it. At the very beginning, it was a technical necessity to use a sequential execution of instructions (and so: to introduce an *appearance-ordered execution*) : the CPU was very expensive, and it was only available in one single instance, so it had to do every single operation. Just a bit later, the question rose: *could we make some instructions in parallel, at the cost of building some extra hardware*?

The theory of the problem has been scrutinized around 1990 (Wall, 1993). Since this classic work, just minor improvements have been made, but this serves as a base for pipelining, as well as for the increasingly popular out-of-order evaluation in the modern processors. The author analysed many of the factors affecting the different dependencies, and concluded, that "*parallelism within a basic block rarely exceeds 3 or 4 on the average. This is unsurprising: basic blocks are typically around 10 instructions long, leaving little scope for a lot of parallelism. At the other extreme is a study (Nicolau and Fisher, 1984) that finds average parallelism as high as 1000, by considering highly parallel numeric programs and simulating a machine with unlimited hardware parallelism and an omniscient scheduler* (Wall, 1993)."

Not to surprise, many of the results of the conclusions are present in the modern processors. A kind of omniscient scheduler is implemented in hyper-threading processors, and the basic blocks are used when attempting to parallelize execution of a single thread using out-of-order execution within a multi-core processor. It looks like the question about the appearance-ordered execution is not yet decided: the out of order execution is allowed, but after that, a considerable amount of time is spent with reordering the result, in order to preserve the illusion that they were executed in order. *But, do we need to do that? Can be really a higher (at least around 100) parallelization reached, if the proper number of computing resources are available, and a wide enough instruction window is used?*

3.2 Changing Execution Order of Instructions

The order in which the instructions (and/or some series of instructions) are executed in most (maybe: in majority of) cases does not matter, if some constraints do not force a special order of execution. Examples include ILP (see above); the processes running under multitasking operating systems, where the code fragments are interleaved to each other; the out of order and speculative evaluation in modern processors; different hardware accelerators; synchronized multi-thread execution, GPUs, distributed processing, etc. Although those technical implementations work, and are widely used, the so called "semantic order of execution", dictated by the early primitive computer architectures, is still required.

3.3 High Level Parallelism for Code Blocks

As outlined above, the ILP provides an obvious method for parallelizing the execution of a single thread. One can, however, consider code blocks (like subroutines, functions, even loops or programmer-defined blocks) as a kind of super-complex instructions, and so an analogous technology can be used by the compiler to find out the data-dependence of this code flow comprising "instructions" from this "Super Complex Instruction Set". As a result, *the method High Level Parallelism (HLP) can be developed, allowing high-level out of order execution*. Just note, that the high level language possibilities introduce several new points to consider, so a lot of new HLP (in addition to ILP) algorithms must be developed. Since one of the important bottlenecks is the low number of registers in the processors, this step can increase by an order of magnitude the number of cores that can be used for parallelizing a single-thread process.

As (Nicolau and Fisher, 1984) pointed out, if the hardware resources are not limited (and, this situation is approached if the parallelization analysis takes place at compile time, rather than at runtime), parallelization of the order of several hundreds can be reached. Although the analysis of the possibilities of instruction-level parallelization takes place at compile time, one has to consider that *the larger the size of the instruction window considered, the longer will be the analysis time and the temporary storage needed.* Fortunately, the high level language compiler can provide information on a reasonable segmentation.

3.4 Discover Parallelization Possibilities at Compile Time

As shown in section 3.1, the simple ILP can enhance the single processor performance by a factor 3-4, even if an order of magnitude more cores are available. This situation is experienced in the modern many-core processors: only 3-5 cores (out of say 64) can be used simultanously for executing a single thread. The reason is that those modern processors discover the ILP possibilities at execution time, so due to the need for real-time operation size of the basic blocks cannot exceed just about a dozen of instructions. However, at compiler time we do have (nearly) unlimited time to discover and (nearly) unlimited hardware resources to find out those dependencies, so the conditions assumed by (Nicolau and Fisher, 1984) are (nearly) fulfilled.

Since the possible data dependencies must be scrutinized, and the time to discover those dependencies grows in a factorial way with the number of instructions considered, this task cannot be solved with a good performance at execution time. In the practice the high level programming units provide hints for selecting natural chunks for parallelizing. Also, the experiences with profilers can help a lot. On lower level, the known methods of ILP can be used, both at source code level and at machine code level.

In this way, this toolchain can generate object code, which provides the possibility (and the meta-information) for a smart processor to highly parallelize the code. The processor will ”see” the instructions in the order as they appear in the memory (which is mostly the same as in the object file), so the instructions which can be executed independently (parallel) should come first. The primary point of view should be to put the instructions in the order as they can be executed maximally independently. A secondary point of view can be to put mini-threads (actually: fragments, a piece of strongly sequential code) in consecutive locations, thus using out the pipelines structure of the cores. The expected maximum number of cores can be a parameter for the compilation process, and in such a case optimization for the actual case can be carried out.

3.5 Smarter Communication Between Compile Toolchain and Processor

So, a lot of parallelization information can be collected at compile time. However, only the part of the information, collected by the compile tools, containing the ILP optimized machine instructions, can be made known for the processor: namely, the object code the processor can read from the memory and execute it instruction by instruction. So, the processor needs to re-discover the possibilities for parallelization, with rather bad efficiency. When transferring the full information in form of meta-data, the many-core processors could do a much better job. The natural way to do so would be to extend the object code with those meta-data.

Some compile-time switch in the toolchain could decide whether the traditional or this multicore format object code should be generated, and also the processor could have a mode to decide whether to operate in single-core or multi-core mode. In the object code seen by the computer the instructions are ordered (as before) as they are expected to be executed. This order, however, can be mostly independent from the order of appearance in the source code, since the compiler can rearrange the instructions, in order to reach a high level of instruction-level parallelism. Just note, that this kind of smarter communication would be highly desirable in many other aspects, say for operating cache memories with enhanced performance.

3.6 Assigning Computing Resource Dynamically to the Machine Instructions

In the classic model *the only computing resource, the lone processor, is assigned statically to the process*, and the assignment happens at the beginning of the computation. The individual instructions simply inherit the computing resource, assigned to the process they belong to. Since only one computing unit exists, a default assignment does the job.

In the multicore model the individual cores are considered as a computing resource, much similar to as multiple copies of arithmetic units are present in some modern processors. The control unit fetches the instructions to execute one at a time, as usual in the vN model. However, in order *to execute an instruction, one has to assign a computing resource to it*, since there is no default assigned resource. This assignment of one of the available allocated computing resources to the instruction occurs dynamically, at the beginning of instruction execution. The allocation of the resources from the pool happens at the beginning of the process (although it might be dynamically modified during the flow of the process).

In this way *every single instruction has a computing resource, exactly the same way as in the classic model*, although here the computing resources, unlike in the classical model, can be different entities. *Also, provided that the control hardware forces considering the possible constraints, there will be no differ-*

ence in the instruction execution. The instructions are executed slightly overlapped, as also in the case of pipelined execution. This means, that – depending on the conditions – *the control unit can make any number of quasi-parallel assignments*: until no more computing resource is available or the constraints do not allow doing so.

To take the real advantage of the reordered code, a hardware dispatcher placed in front of the cores is necessary. Its operation is much similar to the dispatcher presently used in Intel's P6 architectures, but with important functional differences. First of all, the present dispatcher uses a very narrow ILP instruction window, because that parallelization information is assembled at runtime. In the proposed solution, this parallelization information is already assembled at compile time, and is ready to be used immediately by the processor. The other difference, that there is no need to reorder the instructions following an out-of-order execution. In the suggested solution, the dispatcher – in addition to the object code – takes also the compile-time metadata, which was assembled using very wide instruction window, and comprising global information on the parallelization.

For the classic processors, using reordered instruction sequences makes no confusion: since the strongly sequential codes are located in consecutive locations, no problem manifests because of the reordered code. No loss of information, but no gain in execution speed.

However, since some processors use a kind of ILP (with strongly limited instruction window width), feeding code optimized by the compiler over a wide instruction window for a narrow runtime window might result in performance gain even in case of presently existing processors, without any hardware change.

Using the pipeline available in the modern processors, means no theoretical advantage: a pipelined core simply acts like a higher speed core, which is ready to accept further instructions while still processing another ones. When using many cores, the *latency time* is clearly replaced by the much shorter *item time*.

4 CONCLUSIONS

The present model provides chances to considerably increase the single-processor computing performance. *In the suggested model the many-core architectures are included in such a way, that the abstract paradigms like process, processor and machine instruction, remain essentially unchanged from the programmers point of view.*

The model allows the programmers to continue writing single-thread programs and yet taking advantage of the computing performance due to the many cores. Using the new toolchain, the old source codes can be compiled to those new processors with really impressive enhanced raw computing power.

REFERENCES

Adapteva (2014). Parallella Board. http://www.parallella.org/.

Agarwal, V., Hrishikesh, M., Keckler, S., and Burger, D. (2000). Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*.

Altera (2013). DE1 SoC. http://www.altera.com/education/univ/materials/boards/de1-soc/unv-de1-soc-board.html/.

Aspray, W. (1990). *John von Neumann and the Origins of Modern Computing*, pages 34–48. MIT Press, Cambridge.

Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., and et al. (2012). Dark Silicon and the End of Multicore Scaling. *IEEE Micro*, 32(3):122–134.

Fuller, S. H. and Millett, L. I. (2011). Computing Performance: Game Over or Next Level? *Computer*, 44:31–38. http://download.nap.edu/cart/download.cgi?&record_id=12980.

Godfrey, M. D. and Hendry, D. F. (1993). The Computer as von Neumann Planned It. *IEEE Annals of the History of Computing*, 15(1):11–21.

HLPGPU workshop (2012). High-level programming for heterogeneous and hierarchical parallel systems. In *HiPEAC Conference*.

Intel (1998). P6 Family of Processors Hardware Developers Manual. ftp://download.intel.com/design/pentiumii/manuals/24400101.pdf.

Nicolau, A. and Fisher, J. A. (1984). Measuring the parallelism available for very long instruction word architectures. *IEEE Transactions on Computers*, C-33(11):968–976.

S(o)OS project (2010). Resource-independent execution support on exa-scale systems. http://www.soos-project.eu/index.php/related-initiatives.

Wall, D. W. (1993). Limits of Instruction-Level Parallelism. http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-93-6.pdf.

World Scientific (2014). Parallel Processing Letters. http://www.worldscientific.com/worldscinet/ppl.

Xilinx (2012). Zedboard. http://zedboard.org/.

Yang, J., Cui, H., Wu, J., Tang, Y., and Hu, G. (2014). Making Parallel Programs Reliable with Stable Multithreading. *Communications of the ACM*, 57(3):58–69.