

# Main Principles of E2K Architecture

Boris A. Babayan  
Elbrus International  
`boris.babayan@elbrus.ru`

July 31, 2001

The high level goal of E2K architecture is to build a **FAST COMPATIBLE** and **RELIABLE** Computer.

resources to execute operations in parallel. That was an era of a single issue, pre-superscalar RISC architecture.

## 1 FAST & COMPATIBLE Computer

As it is well known, the computer architecture is technology driven in a great degree.

### 1.1 CISC

In the early days of computing, when computer architects could not afford to use even a moderate volume of hardware, all operations were executed strongly sequentially, furthermore, each instruction and operation were split in smaller pieces, which also were executed sequentially, under the micro-program control. That was a typical CISC era of computing.

### 1.2 RISC

Exponential technology growth gave more hardware available for computer design and the microprogramming approach became obsolete, all operations became hardware implemented, but still the operations were executed sequentially. There were no enough

### 1.3 Single Issue Scheduling Problem

Before we will proceed to the next stage of computer architecture, I would like to discuss a little the process of creating executable binaries and the way of their usage on different computer models.

An executable binary includes a full description of a program algorithm, all operations and all data dependencies. This means that for each operation it is clear which operation results are used as arguments for this specific operation. A little bit less evident is, what specific resources of the given computer model should be used for execution of resources of a specific model for execution of specific operations from the program code binary is done.

For the described above presuperscalar CISC and RISC single-issue engines the situation was quite evident. The instruction sets of these computers are sequential by their nature — each instruction consists of a single operation and the whole program implies sequential execution of these instructions. A single-issue engine looks for a compiler, like a computer with a single execution unit. So, all operations from the program code should be appointed or scheduled to this single unit and ought to be executed in a sequence predetermined by the binary code.

For LOAD/STORE RISC architecture parallel execution of LOAD operation was introduced, creating a special optimization problem, which was and still is being solved by optimizing compilers. But nevertheless, the instruction issue process was still sequential.

Up to this point in the computer industry, the compilers of software vendors could do resources scheduling (strictly sequential) at the time of compilation and, what is extremely important, this schedule was (and is) valid for all different models (which different technical characteristics of resources) of the same platform.

For Single-issue computers the distributed binaries represent, besides the program algorithm, a detailed schedule of computational resources quite efficient for all models of a specific platform.

## 1.4 Multiple Issue

With the progress of technology multiple issue computers have become feasible. This means that now with this hardware the same distributed sequential binaries cannot include a detailed resource schedule, but only a correct program algorithm. The reason is very simple. Computer vendors usually deliver a new model maybe each half a year, in average, but the program binaries are updated much rarely. The same distributed binaries must be executed on many different computer models of the same platform, but for a multiple issue engine capable of parallel execution of different operations, a specific resource schedule should be substantially different for a different computer model with a different resource structure, unlike a single-issue engine.

## 1.5 Superscalar

To cope successfully with this problem the computer architects have developed dynamic scheduling hardware and a new stage of computer architecture history has been started. That was (and actually, still

is) a superscalar era. The very first commercial superscalar has been delivered by the Elbrus team long before. But for in the West, the superscalar approach became popular early 1990s.

Each individual superscalar computer, in its hardware, using the same distributed binary of a program code, during the program execution in real time dynamically appoints specific resources of this computer (execution units, register file locations, etc.) to each algorithm entity (operations, register locations, buses, etc.)

In presuperscalar computers each code instruction represents a real physical time step of an executable program; a reference to register location implies a real physical register location; a reference to operations (op code) implies a real physical execution unit (though a single one in a presuperscalar case).

Superscalar hardware considers all references in an instruction as references to virtual resources and dynamically appoints to them the real ones.

The instruction sequence becomes virtual (out-of-order execution), register reference — virtual (register renaming), execution unit — virtual (selection of one of the parallel physical units).

This dynamic scheduler tries to load, when it is possible, many available parallel hardware resources of this computer model. In many cases with this approach it becomes possible to issue many instructions at a time, which results in a substantial speed increase. Only a coarse schedule is used for the distributed binaries, detailed scheduling is done locally.

Many today's superscalars can issue up to 6 instructions at a time, and the average number of instructions executed in each clock is about 2 for integer jobs.

A positive point here is that this dynamic scheduling control adapts and schedules a single binary code to a specific resources structure of different computer models of the same platform. Wide use of the super-

scalar architecture shows clearly that it is impossible to include into the distributable binaries an efficient schedule of resources for different computer models with parallel hardware.

## 1.6 Superscalar Drawbacks

While so far this approach has been rather successful, it, nevertheless, has substantial drawbacks. It works well for certain level of hardware parallelism, when the hardware volume is not so big. When it exceeds this level, the job of resource scheduling becomes so complicated that its execution in a real time limits the speed of the computer.

The problem is even more complicated, because to be able to do the job of scheduling a superscalar engine must do an appropriate analysis to find out whether a specific optimization of the resources is applicable at this point. This analysis typically tries to discover certain data dependencies, which can prevent some optimizations.

The time slot favorable for superscalar microprocessors spans early 1990s till present time. But today this architecture has reached its limit.

As an example, the execution engine of ALPHA-464, according to the published papers, can be built with an issue rate equal to 8, but the control unit can not do analysis and scheduling in time. Thus, ALPHA-464 designer introduced 4 independent program counters with independent analysis units (no need of the data dependences analysis between 4 streams) to feed 8-way execution engine (SMT) — each program counter will use only a part of the available hardware parallelism. This approach should not be regarded as a proper solution of the problem, because it does not speed up a single integer stream, but on the contrary, it even slightly slows it down.

Due to the exponential growth of technology, we can predict a very rapid growth of the available hardware parallelism, which will make the superscalar approach

obsolete, and which makes the problem of resources scheduling even more challenging.

## 1.7 Elbrus Approach

To help use hardware parallelism in a greater degrees and to speed up future computers in accordance with the growing technology, the Elbrus team developed a new approach, which should signify the next stage in computer architecture progress.

The main innovation of this approach is that the job of analysis, optimization and scheduling is moved in a great degree from hardware to software.

The computer takes the distributed binaries, say, of x86 platform and, before their execution transparently for the user, statically in software, schedules the available resources of this particular computer for execution of this specific program.

Each model of Elbrus, like a superscalar, takes a single for all models binary and tailors, adopts it, makes scheduling to the specific resources of a given model. But unlike the superscalar, it makes this mainly in software, rather than in hardware. This helps to improve efficiency substantially.

In today's superscalar engines software is unable to do this, because the real resources are not under control of the executable binaries.

For our new computer to be able to directly schedule the resources, the program code should address these real resources directly, like in a presuperscalar computer. But unlike it, the today's computer includes a big number of executable resources working in parallel, so now each instruction should be wide enough to include possible specification of many resources working in parallel. This means that this new approach, as a direct consequence, implies a wide instruction approach.

It does not mean that all scheduling job is done nec-

essarily in the software. This scheme allows enough flexibility. We can implement a part of the scheduling work in hardware, and a part in software with the goal to reach the highest efficiency. We will discuss the problem of dynamic-static scheduling tradeoffs later.

This looks like a well-known process of binary compilation from one instruction set (x86) to another (Elbrus). But actually, our process differs substantially.

- Traditional binary translation (BT) compiles one known ISA to another, also known, both of which have been designed well before and independently of the BT itself. This leads to inefficiency and low reliability (not all programs can be translated correctly. A special HW support can solve (and, actually, solves in the Elbrus case) this problem and makes BT reliable and efficient.
- For efficiency reason, all primitive operations of a new computer are designed to be 100% compatible with the target architecture (x86), except for the rarely used ones, which can be emulated.
- The main goal of this binary code transformation is the resources scheduling. The main goal of the traditional BT is code porting, without precise resource scheduling.

The general way in which this computer works can be presented as follows.

When it tries to execute new x86 binaries, the system (special HW and system SW) detects automatically that there is no stored binary compiled schedule and starts dynamic compilation and execution. It creates an optimized version of the compiled code and store it for future execution. In case of the next call of the same program, the already optimized code will be used for execution.

A very important point is that the while process is transparent for the user. It looks for him like he is using a traditional x86 computer.

Another important point is that with this special H-W and SW system support, the execution is highly efficient and 100% reliable — all codes executable on a regular x86 computer, work in the same way on the new computer.

*The essence of our approach is that we introduce two interfaces:*

- *the first one is a common distributive, open for everybody, say, x86 or a specially designed Portable Object Code;*
- *the second one is ISA of each model, for internal use only, showing all resources to the compiler;*

*With local translation from one interface into another and local resource scheduling during translation.*

Some concluding reasoning.

The main superscalar problem to be solved is the following. On the one side, we should be able to do more complex scheduling for more complicated and parallel hardware of today and future. On the other side, we should decrease the complexity of hardware. This looks a little bit contradictory. So, a natural solution is to move this complicated scheduling work into software. Anyway, this scheduling work should be done either in hardware or in software. Moving it into software we are not increasing the amount of the work, instead, we are even decreasing it substantially, because we do it only once during compilation, while a superscalar is doing it repeatedly.

We can assume that a possible solution should be as follows. To include in the distributive more information which helps to do the hardware scheduling simpler. Unfortunately, it does not work. First, it contradicts to the problem of binary compatibility. We can change nothing in x86 binaries. But even if we are bold enough to introduce the new binaries, it is hardly possible to have enough information useful for any possible future models with different sizes and structures of resources. Each new resource structure

needs its specific scheduling optimization and, like in an optimizing compiler, each optimization needs its own specific analysis. Thus, to have a distributive suitable for all future unknown modes is just a science fiction.

So, it looks like the only possible solution of the problem is the Elbrus approach.

It can be named as “ExpLicit Basic Resources Utilization Scheduling”, or ELBRUS for short. And this approach should signify new technology period in computer industry.

## 1.8 Dynamic versus Static Scheduling Tradeoff

Now we would like to discuss a proper tradeoff between dynamic and static scheduling.

As has been stated already earlier, for single issue engines common static scheduling is not a problem.

For small number of units able to work in parallel, a superscalar dynamic resource scheduler works well and does not limit the execution speed.

But for a bigger number of parallel units, the job of algorithm analysis and resource scheduling grows quadratically. This work must be done each clock, so very soon it becomes a speed-limiting factor and can lead to the speed degradation.

On the other side, a static scheduler has other drawbacks.

- ◀ P.1 Computer spends some time for compilation.
- ◀ P.2 Some disk space should allocated for the optimized compiled code of the frequently used programs.
- ◀ P.3 While it can ensure a top most efficient usage of statically behaving resources, it cannot take into consideration real dynamic situations, cache misses and so on.

### Point 1.

Even for today’s program, time spent for compilation is a small part of the whole time spent for execution. This has been confirmed by Transmeta experience. Crusoe uses dynamic compilation only — it compiles each program from a scratch before each program execution and even with this approach they have quite affordable speed.

In the Elbrus case we are using static compilation, that means that in most cases we are executing statically compiled well-optimized version of code without these small losses at all.

Moreover, over the time, with the progress of technology, while computer became faster, computer executes much more instructions in a time unit, so the ratio between number of dynamically executed operations in a single program run and number of operations (static) in the same program code grows rapidly. This moves tradeoff point in favour of software scheduling approach with progress of technology.

### Point 2.

The disk space for a compiled program is rather small as compared with the today’s disk unit’s capacity, which will be growing even in greater degree with time.

Moreover, the size of this space is under system control, because in accordance with some strategy, rarely used optimized codes can be removed and recompiled again if needed.

### Point 3.

In case all hardware resources have precise statically predictable behavior, the approach of software scheduling ensures a close to the best resources usage and code efficiency. Unlike the hardware scheduler in superscalars, in this case the compiler can analyze a big portion of the program and optimize it nearly up to a possible limit. It does it only once for each specific piece of the program, while the superscalar repeats

it each time this code is being executed, burning extra heat and slowing down the execution.

There is only one parameter of a computational resource, which behavior is difficult, if possible at all, to predict precisely during compilation. This is a memory system READ latency.

Besides the traditional means of improving this parameter, like memory hierarchy (caches), the Elbrus project has implemented a number of innovations resulting in the losses decrease to a negligible value. These are the following improvements.

- LOAD hoisting across basic blocks and above ambiguous STORs (Using speculative LOADs and disambiguation memory);
- A big register file to keep the preLOADed data;
- Branch preparation for preLOADing target instructions;
- Array prefetch buffer, (FIFO buffer for asynchronous preLOADing array elements)(An element of dynamic scheduling);
- Explicit handling of cache misses by the program code, supplying two separate schedules: for hits and for misses;

These arrangements by means of preLOADing and explicit scheduling reduce the job memory latencies prediction to the worst cases and make static scheduling very efficient.

In rare cases (due to introduction of the above measures) of miss prediction, the computer uses a traditional scoreboarding technique, which is also a piece of the dynamic scheduler.

So, as a result, today a real tradeoff between the dynamic and static scheduling is in a great degree in favour of the static one and due to the technology progress will be further moving to static in the future.

## 1.9 E2K Architecture Advantages

Besides the high speed this approach has many outstanding advantages.

### 1. Simplicity.

As a result of removing from the hardware such complicated mechanisms as out-of-order execution, register renaming, speculative execution and branch prediction, a computer becomes conceptually as simple as presuperscalar RISCs.

### 2. Small die size. The same reason as above results in a substantially smaller die size.

### 3. Good cost/performance.

Result of high speed and small die size.

### 4. High clock frequency.

Conceptual simplicity leads to the ability of high clock frequency implementation.

### 5. Better use of high technology.

Proper use of a big number of transistors is straightforward.

### 6. More efficient compilation.

While the compilation process is not simple one, it is easier to reach high efficiency, because, unlike a superscalar, precise resources behavior is known at a compile time.

### 7. Regular design approach.

Control unit simplicity and freedom to change the instruction set for different models, leads to quite a regular design approach. We are designing the data paths of a computer first, without any limitations from the control unit. These data paths can be designed with a top performance according to its internal logic. Then we design the instruction set, which fully uses these data paths. This means that:

- The design process is easier: data paths and control logic are designed independently;

- The data paths (and as a result the computer itself) can be designed with a top most efficiently.

8. Testing (and correctness proof) of hardware is simpler.

In a superscalar the same instruction works in different environments depending on the different dynamic prehistory of different runs. Rather a long prehistory can influence the execution of this instruction. For good testing we should try a big number of these combinations. In our case all dependences are within the length of our short pipeline.

9. Scalability.

This system delivers the best scalability, because the job of tailoring to a specific model is moved into the software.

10. Multiplatform implementation.

Due to the usage of binary compilation it is possible to support many architecture platforms in the same computer.

## 1.10 Alternative Approaches

Now we are going to discuss other attempts to overcome the limitations of the traditional superscalar approach. There are only two such cases in the industry: IA-64 of Intel and Crusoe of Transmeta.

### 1.10.1 Intel IA-64

The basic principle of operation of IA-64 is very close to earlier superscalars — no out-of-order execution, no register renaming and no implicit speculative execution. Computer executes all instructions in order, but using a grouping logic tries to execute as much consecutive instructions at a time as possible without violation of the data dependences. It also inherited from the superscalclar branch prediction mechanism.

The main difference from the superscalar is that IA-64 made public (introduced in distributed instruction set) all internal techniques used by th out-of-order superscalar for dynamic local resources scheduling:

- Speculative execution of instructions including LOAD hoisting;
- Grouping logic;
- Disambiguation memory, etc.

Using these techniques, the compiler, which generates a distributive, can do nearly all-scheduling work, which out-of-order superscalar engine is doing dynamically in each mode in the run time.

In IA-64 the distributive instruction sequence is chopped into groups, consisting of the independent instructions, which can be issued concurrently. As we will show later, it can be done efficiently only with some assumption about the real available resources of a specific model.

For proper use of the available resources the compiler introduces enough speculative and predicated operations. In correspondence with the available resources and real memory system latencies the compiler moves up some LOADs, using somtims a speculative mode or disambiguation memory. All these optimizations have real sense, when we know real available resources.

Each model can execute not the while group necessarily, but its part only, according to the resources available in the model at this point. But the computer cannot issue instructions from different groups in the same clock. This is the only possible way of adaptation to a specific model. We will show later that it is less than enough.

The wrong point is that the distributive is a code optimized for one specific model only. IA-64 removed from the superscalar approach the mechanisms of adaptation to a specific model without any substitution (we cannot regard a partial group execution

as something adequate). It is dangerous, especially in terms of the rapidly growing chip fabrication technology posing a great challenge to the architecture designers.

The authors of IA-64 either decided that all future models will be based of the same resource scheme (which is unrealistic), or they decided that it will be OK for everybody to heavily lose the efficiency running a program compiles for one model on another one. Actually, this means that IA-64 computers will run (typically) a code optimized for some previous model with a corresponding loss of efficiency. So, for a regular user we should deduct at least 25% of speed from th officially declared values.

As it can be clearly understood from th above discussion, the scalability problem in IA-64 has been solved on a very poor level. But the efficient compatibility has not been addressed at all (in IA-64 proper). These two problems are quite close to each other.

IA-64 suggests for compatibility the implementation for both (x86 and IA-64) ISets in one chip (“two in one”). Having in mind poor scalability of IA-64, the natural generalization of IA-64 compatibility solution for future models is to include all previous models in the current chip.

IA-64 looks like a superficial, shallow E2K copy, implemented without even understanding of the deep roots leading to this solution.

Now a few examples, showing bad scalability of IA-64:

1. The following two pictures below show th examples, which prove clearly that it is impossible to do efficient grouping of instructions without precise knowledge of the computer width. The picture presents a sequence of instructions with the width dependencies. Stops show the end of the groups. The first example shows that for efficient scheduling in cases of a three-way and four-way computers we need to select different

orders of instructions, otherwise losses will make up to 25%.

In the second example to have an efficient scheduling for four-way and two-way computers we need to change the STOP positions, though the sequence of operations is optimal for both cases, Failure to change the STOP positions will result in 25% losses again.

2. With IA-64 approach it is difficult to introduce register clusters like in Elbrus-3, E2K, and Alpha. The system with clusters needs a special way of scheduling of instructions issued in each clock. In this case it is not enough to split the instructions into groups issuing in the same clock. It is necessary to appoint each instruction in a group to a specific cluster in addition. This job needs some additional analysis of dependencies. To avoid extra bubbles we should find out all separate instruction dependencies between the adjacent groups (this is a good piece of analysis) and schedule the dependent instructions to the same cluster.

This information is absent in IA-64 code, so, to be able to efficiently use the cluster structure we should either change the instruction set or return to complex analysis and scheduling to hardware (like in a superscalar). Even if we change the instruction set and include the information about clusters, it will be O’K only for a specific model.

The resume is either we should return the analysis and scheduling to hardware, and in this case IA-64 will have the same drawbacks like a superscalar, or we will lose all scalability.

3. The same situation exists with a two-storeys execution unit like in the E2K. We should appoint a linked instruction to a two-storeys unit, which needs special analysis and scheduling hardware. The same resume as above.
4. For each specific model width a compiler should choose an appropriate level of speculation. Excessive speculation for narrow models means a



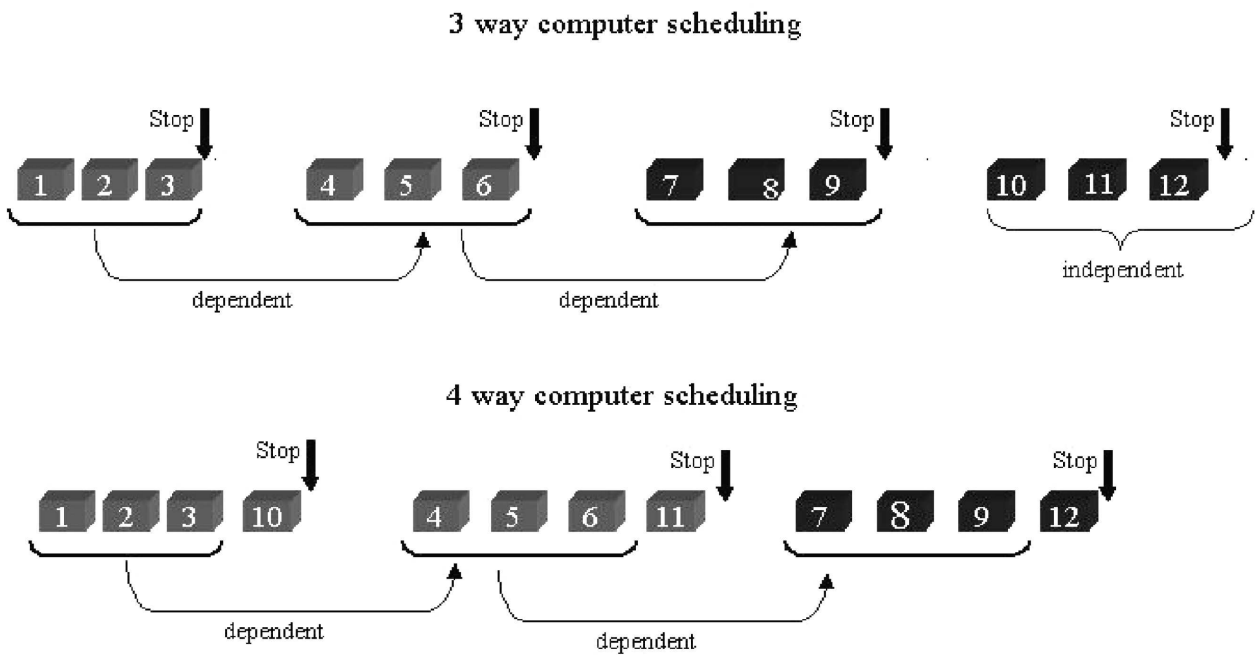


Figure 1. 3-way and 4-way computer scheduling

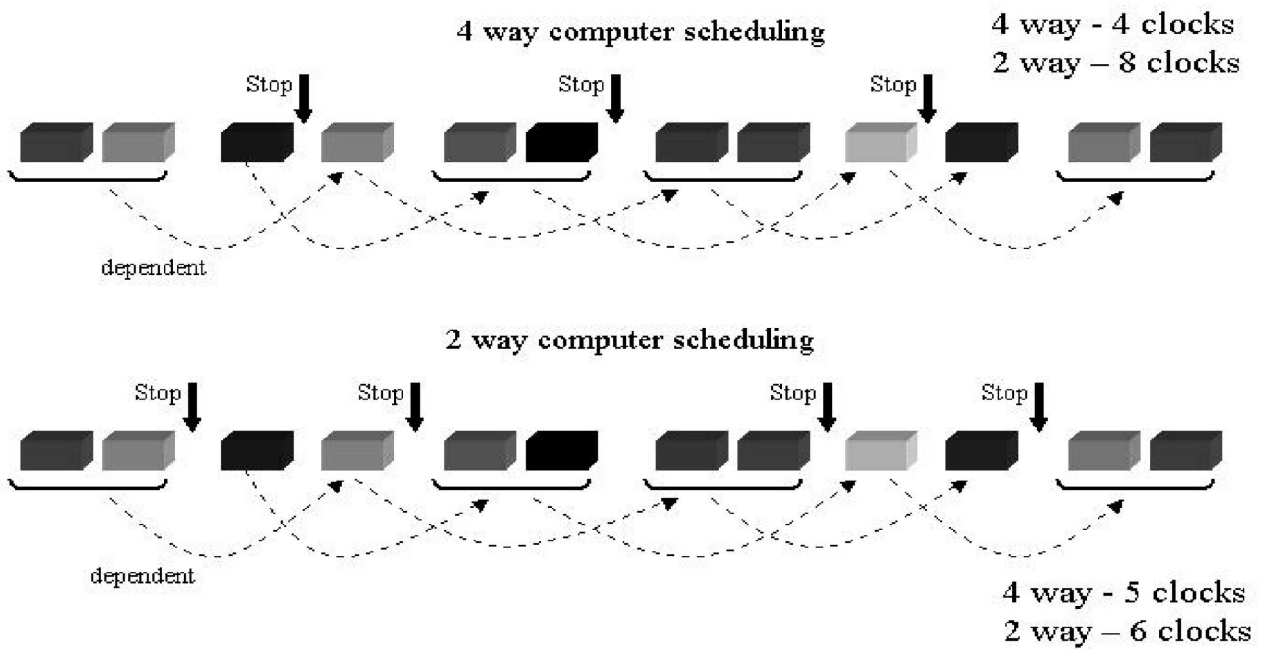


Figure 2. 4-way and 2-way computer scheduling

speed decrease due to the superfluous operation execution. Low speculation for wide models means a speed decrease due to the incomplete resource use.

Speculative operations are included in IA-64 code. This means that the code is already oriented to a specific computer width. This means again the lack of the scalability.

5. With the advent of  $0.1\mu$  technology tightly multicore chips will be designed. To use this advantage IA-64 distributive should be recompiled. Again, the lack of scalability.

It may seem that at a later stage Intel may try to fix the drawbacks of the IA-64 approach by introducing binary compilation approach like in the E2K. It is possible in principle, though the priority in this area belongs to the E2K authors.

The E2K scheme assumes 2 interfaces (as has been stated above): a distributive binary and an instruction set of a specific model, with the binary compilation in between.

But the IA-64 instruction set is poorly suited to both roles.

It is bad as a distributive, because it consists of many details dependent of the resource scheme of specific models, like speculative execution, disambiguation memory, etc., which should not be presented in a common distributive.

It is also bad as the instruction set of a specific model, because it gives no access to physical resources.

### 1.10.2 Transmeta

Transmeta has taken and implemented this technology more consistently. But they use this technology having in mind different goal.

They are using this technology to deliver a comput-

er, which dissipates low power. This example shows great power and flexibility of the technology.

Crusoe instruction has four syllables (Elbrus has 16 and up to 23 executable operations at a time), and it uses dynamic binary compilation only (Elbrus uses static compilation as well). So, Crusoe speed has enough room to grow.

## 2 RELIABLE COMPUTER

The general reliability of computation can be split to hardware and software reliability.

### 2.1 Hardware Reliability

The Elbrus team has big (3 generations of computer systems and more than 25 years) experience in designing highly reliable hardware logic, but, for some reason, it is not included into the E2K project now, though it can be included easily.

We would like now to present basic principles of Elbrus hardware reliability.

All units have extra hardware detecting all single hardware faults, either intermittent or solid, and signaling about them to the rest of the system.

Each unit has another special hardware, which, in case of such a signal, automatically without the software help, disconnects all signal lines from this faulty unit.

Then the rest of the system starts the software recovery process. Elbrus line computers are multiprocessor systems, so, in most cases there are enough units to continue the work successfully.

This kind of arrangement gives not only a high level of reliability (the system as a whole has no down time even in the case of low reliability of "raw" hardware),

but a high level of trustworthiness as well.

The system can have intermitter faults — most probably, it will recover successfully and continue the computation.

If the system cannot recover or if the fault is solid, most probably, the faulty unit will be excluded from the system and the rest of the system will continue the program execution and, eventually, will deliver a correct result.

In extremely rare cases the system can signal its inability to recover or to continue the execution.

Virtually, there is a zero probability that the system will deliver some wrong result without any signal of alarm.

## 2.2 Software Reliability

Software reliability means that the influence of a buggy and rogue program on the rest of the system must be excluded.

Software reliability means secure programming. It ensures:

- Easier program debugging;
- Substantial decrease of time to market for software products;
- Substantial decrease of bugs probability in the already delivered software systems and , as a result, decrease of rather dangerous consequences;
- And what is the most important, maybe, the exclusion of the viruses' danger.

The world society is losing annually many billions of dollars due to a poor implementation of these features in computer systems.

And it is very funny that a strong remedy has been well known to evrybody maybe from the very beginning of the computer industry. Its implementation is quite easy and straightforward. Many high-level language systems have very convincing practical experience. The Elbrus team has more than 25 years and three generations of computers experience of creating the whole system — high-level language and operating system.

To introduce this system it is necessary to change compatibility, this is difficult, but compared to the tremendous losses from viruses this slight incompatibility looks more than affordable. It is maybe less than the incompatibility between two versions of the same program. And we can suggest a very graceful and painless transition to the new system.

*The wisdom is simple — data types (for security pointers only) should be handled correctly.*

Nobody, except for the virus writer with their malicious intentions, wants to violate this simple rule. But the system must check for proper pointer handling to help debug the programs and stop the virus writers.

To discuss the subject in a more detail we should do it separately for memory and for file system.

### 2.2.1 Memory

Today in popular languages pointers are represented explicitly by an integer. A user can assign an integer to a pointer data type. This violates memory protection.

As a result, two different procedures of programs in the same virtual space are not protected against each other. This means:

- Even inside the same program — a bad debugging facility.

- No memory protection in the same virtual space.

Different programs usually use separate virtual spaces with a loss of efficiency and difficulty of communication and use of common data.

To cope with the problem some languages, like Java, exclude pointers from the language at all. But it makes the language non-universal and the programming less efficient.

Our approach is quite different. It can be regarded as an extension of the Java approach. We are supporting data and procedure pointers in hardware, which makes its use very efficient.

A procedure pointer consists of a reference to the context and a code and is used for procedure calls. The procedure mechanism (call/return instructions and parameters passing mechanism) is supported in hardware as memory protection domain. The procedure context is supported by means of the data pointers.

In the E2K all pointers are marked by a special bit, which helps hardware to check proper pointer handling. The E2K needs no special memory system — some extra combinations in ECC code are used instead of keeping this bit.

The main cost of this security hardware is two extra bit per each 32 bits in CPU and cache chips. It almost does not slow down the execution speed.

The Elbrus experience shows that this scheme:

- decrease debugging time substantially (by up to 10 times);
- decrease the probability of missing an undetected bug in the already delivered software.

*We have found more than 30 bugs in SPEC benchmarks, which should be regarded as well-debugged software.*

- ensure perfect protection between separate procedures in the same virtual space.

*You can run safely in the same virtual space any downloaded program, communicate with it through the parameters, without any danger to the rest of the program and data located in the same virtual space.*

It gives a good support for implementing a perfect protection in the file system.

### 2.2.2 File System

#### Traditional Systems

Current situation in the file system of traditional operating systems is even worse. The pointer data type is not introduced there at all. One should use a regular character string as a file name or a file pointer. This string describes a path to the named file from the root, common for the whole computer.

This arrangement creates a situation very favorable for the virus writers trying to introduce viruses.

Suppose we have downloaded a program from Internet and we are going to pass a parameter file to this program. Moreover, it is possible that inside the parameter file there are some strings representing other file names in our file space. This is the reason, which to the downloaded program the root of *our* file space must be given — to be able to run this kind of programs and to have an access to the parameter files and their derivatives in our file space. We cannot even restrict the access to the files with the access right control mechanism, just because the downloaded program is run under our name.

It is difficult to imagine a situation more favorable for the hackers and viruses.

Apart from the virus problem, it is a wrong arrangement from the point of view of regular programmers practice.

Suppose a programmer (not a virus writer) has cre-

ated a problem in, say, a computer A in the file space of the computer A, and then this program has been moved to and executed in a computer B. What happens now? This program will be executed in a different environment, which is wrong from the regular programmers practice point of view. Only the parameters should be passed from the user, who called it, both the rest of the environment should remain the same.

So, this is inconvenient for the programmers and favorable for the virus writers.

#### Elbrus Approach

As has already been told, the Elbrus solution is based on the introduction into the file system of two data types: a pointer to the file and a pointer to the program.

No Additional hardware support is needed to implement this approach. All pointers are located in dictionaries only. But unlike the traditional root centric system, this file system is network based. Pointer to the problem consist of a reference to the program code file and reference to the dictionary, which represents the file context for this specific program code. In a particular case, the context dictionary of all program files can coincide with the old root. So, the new system can be regarded as a strong extension of the traditional one.

In this system the access rights control becomes optional. The pointer control is more than enough for a strong and very convenient for the users file system.

We can easily find out that viruses cannot exist in this system at all. Assume that cross computer file pointers are implemented in the system.

Suppose a computer and user A downloaded a program (created by a user B) from a computer B. Now this program has an access only to the parameters passed explicitly by the user A and, probably, to its own context, if any, from the computer B (through an inter-computer reference). There is no possibility

to violate the file protection or infect any files from the computer A.

Though this system substantially differs from the traditional one from the implementation point of view, actually, it can be implemented with a very small incompatibility from the users' point of view. Moreover, we can suggest quite a graceful way of its introduction.

By the way, it is impossible to solve a virus problem without introduction of some incompatibilities, just because today any virus is quite a legal problem, and to the contrary, a restriction to execute a virus program is in a direct contradiction with all language and system manuals.

Our approach absolutely excluded the virus danger, unlike the today's endless practice, when people are waiting for a next virus to appear and create a treatment for this specific virus only still waiting for the appearance of the next ones.

There are two ways to infect the system by viruses: through bugs in the system software and through the described above design flaw in the file system.

Elbrus has addressed both, bugs in the system can be decreased substantially by our memory protection system and we have a sound suggestion how to improve the file system as well.

Elbrus has big experience in implementation and usage of this system (over 25 years and 3 generations of computers).

We know that introduction of this system into the real life is a big challenge, but avoiding the virus danger is even a more challenging problem.

Copyright ©2002 *Boris A. Babayan*

Verbatim copying and distribution of this entire article are permitted without royalty provided the copyright notice and this notice are preserved.

---

**Dear Readers!**

The Elbrus team has a sound and convincing experience in the practical design of commercial universal computing systems with high security level.

The security level of this technology ensures full protection against viruses' danger. It also substantially improves debugging facility, especially for big software systems.

This advantage is vitally important for the whole computer community today.

Large-scale implementation of this technology needs no research, but big implementation efforts.

These efforts fit well the free-domain community activity.

We will be happy to take part in the discussion that helps this idea come true.

— Boris Babayan

**About the Author** Prof. Boris A. Babayan is the developer of supercomputers in the Soviet Union and Russia. He leads the E2K development, oversees all design teams.

Boris Babayan was born on December 20, 1933. In 1958, he graduated from the Institute of Physics and Technology (the Chair of Computer Technology headed by Academician S.A. Lebedev). During 1958-1996, he worked in the Institute of Precise Mechanics and Computer Technology as the Chief Technological Officer, he is Technical leader of M-40, 5E92b and Elbrus I, II and III computers. He is the head architecture and logic designer for these systems. He obtained Ph. D. in 1964.

In 1974, he won the State Prize for development and implementation of complex equipment for CAD, manufacturing and control of complex electronics.

And in 1972, he was elected the Doctor of Science.

In 1977, he got the Professor title, and offered lectures to MPTI students.

In 1984, he became the corresponding member of the Russian Academy of Science.

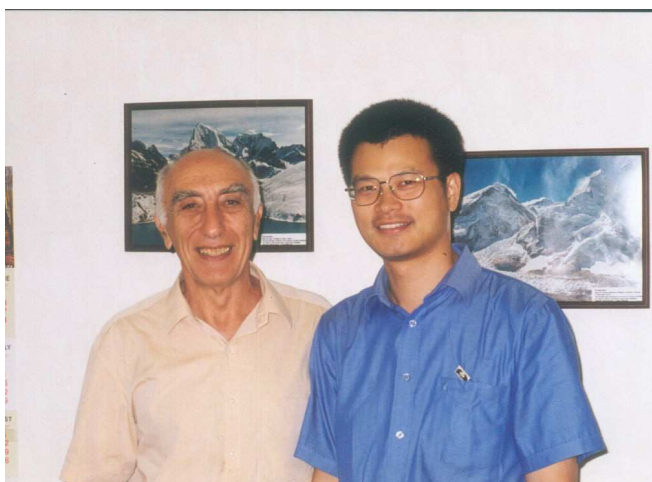
In 1987, he won the Lenin Prize for development and implementation of multiprocessor computing system Elbrus-2.

Since 1992, he is the Chief Technologist at the MCST.

Since 1997, he is the Chairman at Elbrus International.

Since 1998, he is the Director of the Institute of Microprocessors and Computers of the Russian Academy of Science.

He created over 90 patents, innovations and he is the author of publications in processor architecture. he's married and has three children: a son and two daughters.



Prof. Boris Babayan interviewed by Hong Feng, the Chief Editor of FREE SOFTWARE magazine at Elbrus Office, Moscow, July 30, 2001.

