Chapter 17
On Performance: From Hardware up to Distributed Systems

Igor Schagaev, Hao Cai and Simon Monkman

Abstract Nothing is easy nowadays: frequency of processors increased thousand times, system performance as a whole sometimes tripled. Complexity of the system became uncontrollable with zillions of processes and elements to juggle increased unconsciously, leaving for us some comfort but at an astronomical cost. What it means? We are doing something seriously wrong and doing it consistently and persistently. Thus, authors of this work have decided to put together our own discussions and estimations we did since 2002 up to now. We show that system performance depends on user, hardware, and software, structure or architecture of a system and its topology. We propose to see performance analysis a bit wider, thinking systematically what various zones of computer or distributed system can bring or contribute, including the role of processor, structure of system software and overvalued parallelization (try to eat and dance at the same time—it might be fun). We have introduced a kind of virtual architecture through which see instruction execution considering what is in there for us and what system requires for itself. The observation is rather pessimistic. We have briefly demonstrated what simplest architecture if carefully designed can give regarding performance, reliability and energy efficiency AT THE SAME TIME! Regarding distributed systems, we show that Amdahl Law is also very overoptimistic mostly serves to promote parallel architectures and distributed systems. Simple model that we have explained for kids from British primary school and even did field study with them so-called "fence model" made clear that the limit of performance or simply overall reasonably good design is unachievable until we start rethinking the whole architecture and its main element interaction—human, hardware and system software together, pursuing three nonfunctional requirements, performance, reliability, and energy efficiency in concert.

As it was presented and argued in [1–6] any system should be considered from the first sketch down to maintenance using the following nonfunctional requirements:

Performance;
Reliability;
Efficiency (cost, energy).

I. Schagaev et al., *Software Design for Resilient Computer Systems*,
https://doi.org/10.1007/978-3-030-21244-5_17

We call it PRE-requirements. When a system can trade P for R or E and vice versa we call this system PRE-smart system. This chapter is about "P"—performance.

Any system is evaluated in terms of performance, considering performance of elements and system as a whole. Good systems exceed performance of their components, or equal production of component performance; badly design system in terms of overall performance is much less than production or sum of performances of its components. Regretfully, computer systems are poorly designed if we accept this classification. This chapter is about performance and ways to analyze it. We also will model computer system from the position of performance and seeking the ways to improve it, considering use and system performance aspect.

17.1 System Level

Suppose one element has performance Pi; then system of n elements if we can add performance will have maximum performance as n*Pi, i.e., linear growth is assumed. Unfortunately, one has to take into account that external interaction zone (Fig. 17.1) and task structure reduce our expectations about unlimited performance growth.

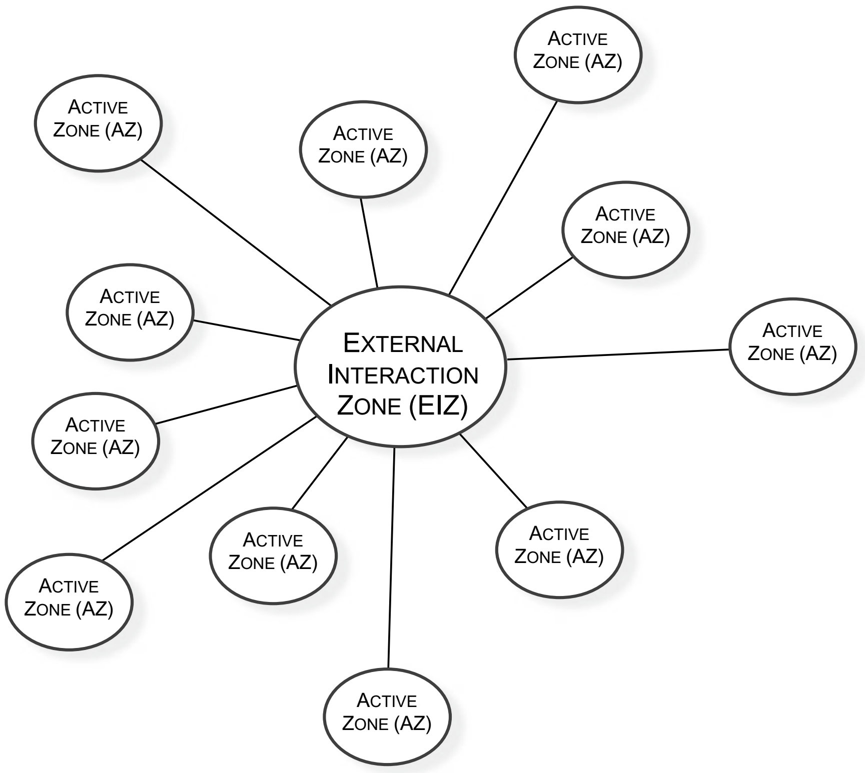

Fig. 17.1 System level of distributed computing

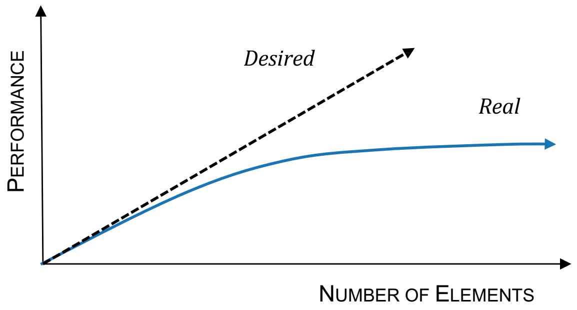

Fig. 17.2 performance versus system structure

Then performance growth is defined as a function of number of elements and EIZ:

$$\mathrm{Ps} = \mathrm{f}(\mathrm{EIZ}, \mathrm{n}) \tag{17.1}$$

where EIZ stands for external interacting zone, n—number of "performers" considered only by performance, not the organization.

Thus, structure of EIZ and its dynamic features (ability to connect transparently arbitrary number of elements with heavy information exchange requests) will impact on system performance of both: system level of performance and element level of performance (Fig. 17.2).

One has to address EIZ features as well as properties of program structure to achieve reasonable gain in performance. System performance-wise program structure itself impact is crucial, as well as ability program to split into independent elements. This ability, in turn, is limited; this causes substantial amount of traffic through EIZ and, therefore, kills performance gain.

17.2 Information Processing Aspect

On the information processing level, consider a system as a black box with input x and output y, with arbitrary function F, Fig. 17.3, top box.

A function or a task of this box in Fig. 17.3 is to get a result y from an input x within allocated time and, if necessary, alter appearance of x in the time slot expected.

Contrasting with mathematical assumptions, for information processing, in principle, input and output timing is loosely dependent, input x might have its own duration while readiness of output y has its own duration, both might overlap, see Fig. 17.4.

A special interest might be in resolving the following cases: how form of x defines (is connected) with function and form of y. Comparative study of durations x and y might be interesting research for embedded systems especially.

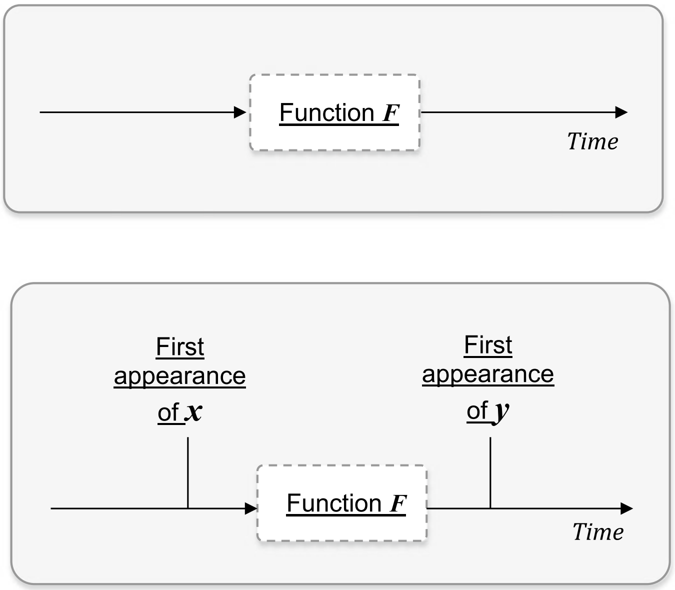

Fig. 17.3 System as a black box, with arbitrary function F

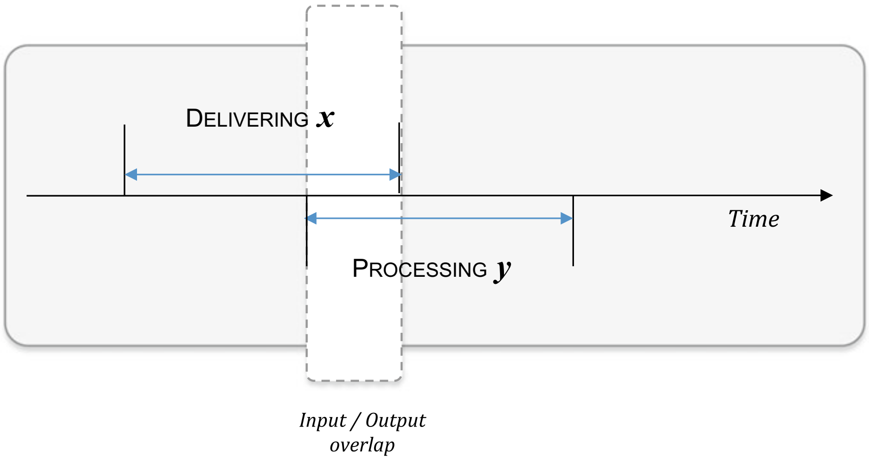

Fig. 17.4 Input appearance might be overlapped with outcome

17.3 Information Systems Task-Wise Hardware Involved Performance

Information system is a combination of three wares: userware, software, and hardware, UW, SW, and HW, respectively (Fig. 17.5).

Thus information processing system might be described in terms of performance of all three: UW, SW, and HW, as they all are involved in information processing. It means that in the long-run performance and efficiency of the system depends on userware, software, and hardware performance.

Their combination might have very peculiar form and mix. In terms of Fig. 17.5, userware, software, and hardware might be connected to perform task as shown in Fig. 17.6.

For real-time systems (and embedded systems especially), or general applications, computer systems over 30 years user features were ignored in terms of overall system performance; it is great regret, we now press much more buttons and click more clicks than 20 years ago achieving very disputable advantage, if any.

This is a subject of special study in UW-SW-HW systems. Here, we spend some more time and analyze details of HW only. However, SW, especially system software (further SSW) has serious impact on performance of hardware, it will be also shown further.

Performance is about task completion in time allocated. The same principle of task allocation and analysis might be applied further down, to the whole system

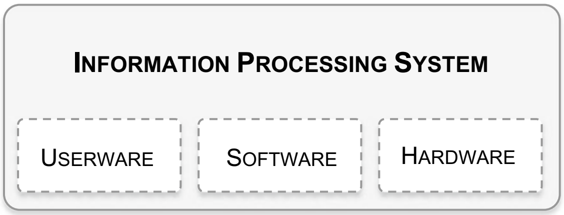

Fig. 17.5 Information system components

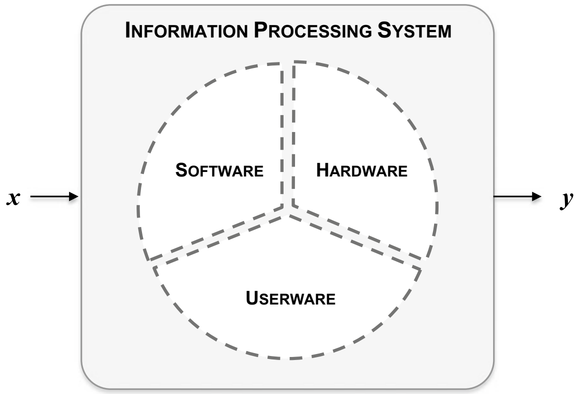

Fig. 17.6 Information system box: x + time = y as three components, S, H, U

hierarchy, for any component—UW, SSW, HW, and business management. Leaving efficiency of management discussion to business schools, we will consider mostly SSW-HW model.

17.4 SSW-HW Performance Model

Task of user (userware tasks, further (UWT)) splits down at the level of HW as a series of micro-tasks (instructions, microinstructions). Program tasks are surrounded by another group of micro-tasks that define the system software (SSW) involvement on execution of the user program, secure a completion, and system resource monitoring.

Thus, userware tasks UWT are accompanied by another group of micro-tasks defined by the system software (system software tasks, further SSWT) (Fig. 17.7).

This total amount of hardware workload in number of instructions W_{ux} to perform user task *x* can be expressed as in Eq. 17.2 where i, j indexes stand for number of hardware instructions required to complete supportive actions (system software need) and user ones (1):

$$W_{ux} = \sum_{j=1}^{m} h_j(sswt) + \sum_{i=1}^{n} h_i(uwt) \tag{17.2}$$

Indexes m and n stand for a system software and user software instructions execution time. Assuming that all hardware instructions have similar execution time (for RISC systems it is essential design condition), one might introduce an efficiency of measure as it is shown below, Eq. 17.3:

$$E_{ux} = \frac{\sum_{i=1}^{n} h_i(uwt)}{\sum_{j=1}^{m} h_j(sswt) + \sum_{i=1}^{n} h_i(uwt)} \tag{17.3}$$

Further, we will dig deeper on performance and efficiency in terms of hardware impact on performance of the system but here note existing relation of efficiency and performance.

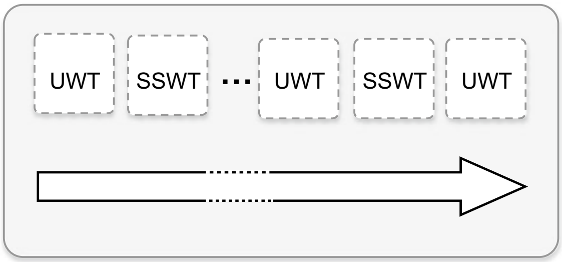

Fig. 17.7 User and system task sequences

Definition 1 Efficiency Eux of computer system is measured by number of instructions required to perform to the total number of instructions performed by computer system.

Naturally, efficiency Eux → 1, while m → 0, and, no matter what frequency a processor is if m → n, Eux → 0.

Regretfully, it is a case for current state of the art in computer systems and especially embedded computer systems. What it means for embedded systems especially?

For various systems, it means that

- Application of Java, or use of modified standard operating system, unavoidably reduces efficiency and, above all, runs out our computer batteries for nothing;
- For military systems, an availability and reactiveness is substantially lower than it could be;
- For office systems, nowadays, the employees are sitting and waiting for Windows or Cisco service more than they actually work.

Leaving further comments about efficiency for system software research and PhD projects, let's concentrate on implementation of hardware instructions in terms of time.

17.5 Hardware Performance

One of the simplest information processing systems is a simple turing machine—(http://ideonexus.com/2009/02/05/javascript-turing-machine/); it demonstrates in principle almost minimum of hardware required to perform information processing. Turing machine usually is drawn similar to mine (Fig. 17.8).

Head is moving left and right accordingly instructions stored on the tape and modifies content of the tape. Surprisingly, if an algorithm for problem exists, it is very possible to calculate it and complete.

Turning machine was invented as a model well before Von Neumann's architecture. The latter one assumes that programs and data are placed on the same tape and head Q might perform instructions. More details about this machine might be

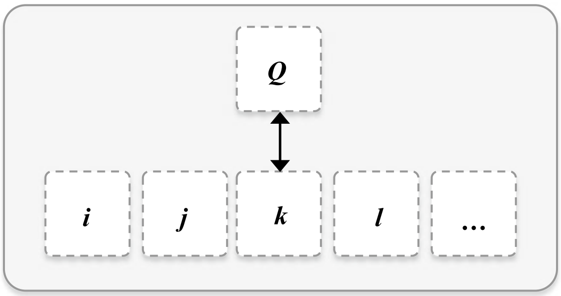

Fig. 17.8 A-la turing for performance evaluation

found in http://plato.stanford.edu/entries/turing-machine/. What is missing in turing or Von Neumann architectures is an answer *how* and *when* algorithm and data were placed into the memory (tape in Turing machine). Besides, arrow on the figure is assumed to connect but not explained how this communication is executed.

17.5.1 Hardware Zones

Thus, from information processing point of view, we have even from this simple picture involvement of three zones (Fig. 17.9):

- active zone,
- interface zone, and
- passive zone.

Active zone (AZ) includes schemes that change information (processors, converters). Interface zone (IZ) includes schemes that transfer information from external source or between AZ and PZ (internal source). Finally, passive zone (PZ) includes schemes that save or store information.

What is missing in the previous picture? To make any information processing model useful, we have to introduce input and output options for information. Each AZ, PZ, and IZ might be active in terms of data flow control (so far not processing) flags—demand of service, interruption request, message writing/sending requests, etc. Let us have a look at Fig. 17.10.

All arrows might be different in speed, and use various frequencies and bandwidths, bit size (8, 12, 24, 36, 48, 64, 128) procedures of control, and data delivery (parallel, sequential). This is still not self-explanatory how we deal with information exchange. More realistic figure is presented in Fig. 17.11.

Level of information interaction AZ, PZ, or IZ actually defines structure of the system similar to Flynn diagram (find what is it, dear reader, it might be useful).

In terms of input and output examples of implementation of each zone, one might suggest the following:

AZ inputs and outputs iAZ and oAZ:

- register content exchange between processors,
- signal write or read to interrupt processor execution,
- mutual exclusion efficiency of multiprocessor communication, and

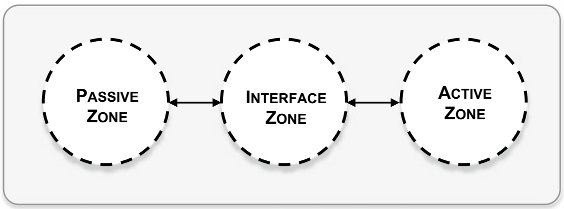

Fig. 17.9 Hardware segments of CA of information processing

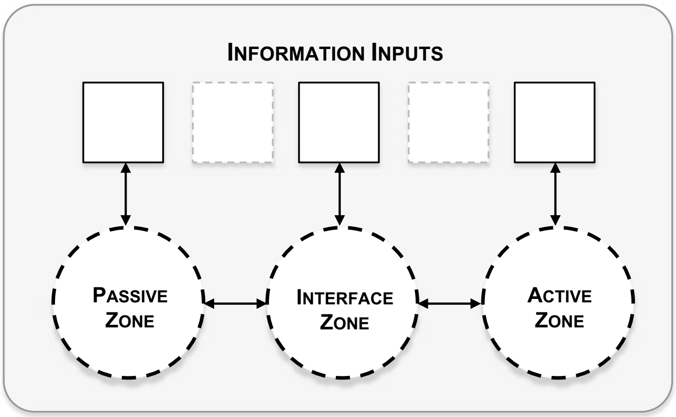

Fig. 17.10 Hardware segments of CA including information exchange options

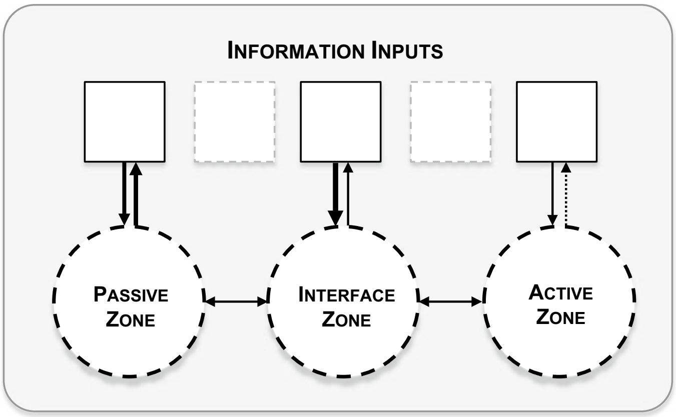

Fig. 17.11 Separation of input and output options for each zone

- Direct writing of processor status word down to syndrome register when syndrome register is external for AZ—flags N, Z, V, and C all might be sent out.

PZ inputs and outputs iPZ oPZ:

- data or control lines from internal bus,
- data path for memory to upload in AZ,
- writing of the result of instruction to output buffers using bus,
- direct access to memory, and
- dual-port memory.

As a small challenge, we invite a reader to present own examples for IZ input and outputs.

17.5.2 Hardware Performance—Instruction Execution

Hardware instruction is implemented using different blocks (zones) that in combination enable to processing.

An example of information tracing through instruction execution is presented in Fig. 17.13 and serves for further understanding of performance evaluation.

Data might be loaded:

- from external source straight to AZ (instruction type A),
- unloaded from AZ to external source (instruction type B),
- loaded directly to the passive zone (instruction type D), and
- unloaded from PZ to the external source (instruction type C).

Information might be

- processed and loaded to passive zone (instruction type H),
- retrieved from PZ into AZ to be processed (instruction type K), or
- processed in AZ and unloaded to PZ.

In fact, interface zone is involved in almost any information transfer internal or external, and in both directions inbound and outbound. It means that the performance of IZ contributes to overall performance of information processing. Special and most promising architecture will be discussed further that limit the role of IZ on overall performance of the computer system.

Other types of instruction not shown might combine the proposed types of instructions.

Let us elaborate a bit what Fig. 17.12 illustrates. Information can come externally through interfacing zone IZ and stored in active zone for further processing—A-type instruction.

From active zone (Arithmetic unit or logic unit or several of them), information through interfacing zone can be transferred out—B-type of instruction.

Well-known good old days direct memory access (DMA)—C-type instructions—are about disabled active zone and read data directly from passive zone (including registers and all types of memory).

The same way information might be loaded into passive zone—D-type instructions—and this is very useful for embedded systems.

In turn, self-isolated instructions, E-type, are about using only active zone and keeping results inside it—sometimes not even addressable at the level of registers—accumulators, signs, or working register.

When instruction processes data, the result can be transferred down to passive zone through interfacing zone—F-type.

Opposite data move: G-instruction are required to deliver data for further processing.

Well away from RISC architectures, instructions can have several cycles of data processing and include result transfer down to passive zone: H-type of instructions.

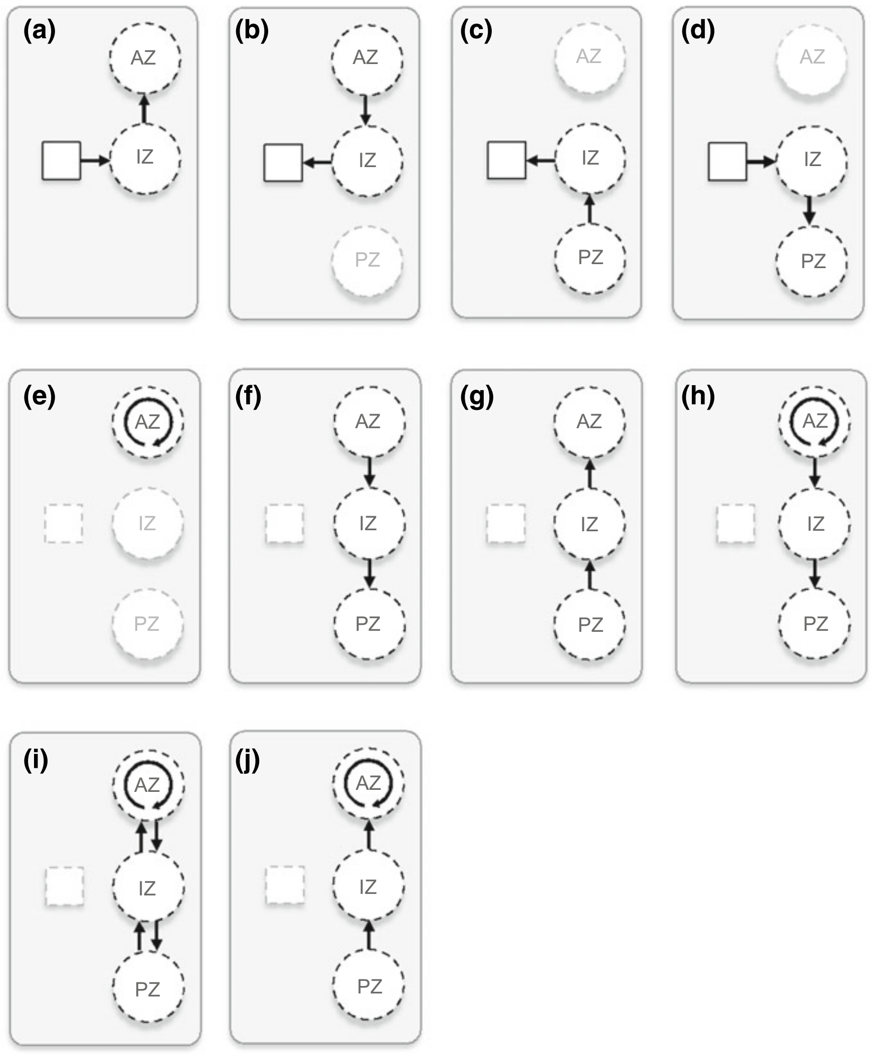

Fig. 17.12 Types of instructions

Last century architectures which are still procrastinated electronic technology by orthodox instruction sets include two types of instructions, I-type and K-type. These two assume that data is taken from passive zone and delivered to active zone, where sometimes after several cycles of processing go back to passive zone (for I-type).

K-type serves the purpose to grab data from passive zone and process, saving results inside active zone for immediate further use.

Please note that passive zone is not starting any instruction; instruction is either initiated by external zone or by active zone. There are some natural questions we might ask here:

Why we need to have a look on these types? why we need to know instruction formats at all?

The answer is simple:

Assuming sequential execution of the instructions, one can estimate an overall performance of hardware for particular tasks and task mix. Each zone has its own specific features in terms of performance and interaction with other zones.

Thus passive zone includes several sources of memory (Flash, SRAM, DRAM) that have different control schemes and timing diagrams, and while involved in instruction execution cause a lot of uncertainties. Interfacing zone might be different in control protocols, width, performance, and reliability.

Two examples of instruction execution using all three zones are shown in Fig. 17.13.

Hiy is loaded from external source, saved in passive zone PZ, and retrieved into AZ to be executed; when results are ready, they are delivered back to IZ_e, where index e stands for external.

Another type of instruction shown describes the load of HIy to AZ, executed in several iterations (or just one) and saved back in PZ.

Data might be processed by instruction when data come from external source directly to the active zone and after processing goes directly to PZ, or when data follow reversed order, from passive zone to active zone for processing and then to external destination.

Data go directly to passive zone—well-known direct access to memory, or when data are dread by external source—direct reading from memory.

Finally, there is a group of instructions in modern hardware that is assigned for reading or writing from or to external source and when data retrieved and processed in active zone.

Passive zone is not starting any instructions either initiated by external zone or by active zone.

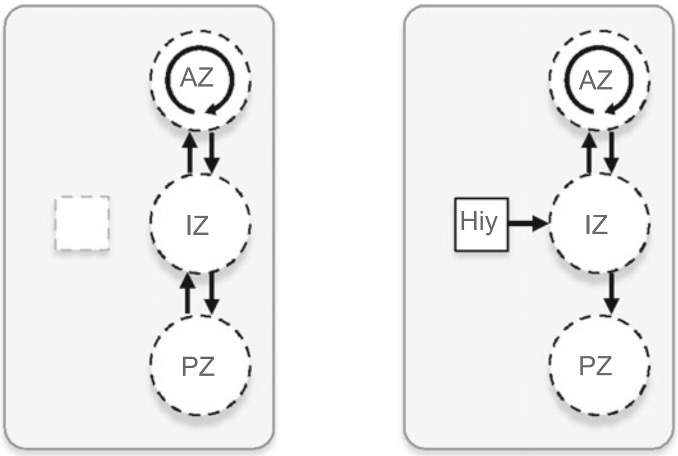

Fig. 17.13 Complex processor instruction from the previous century

In turn, active zone might consist of (even within a mobile telephone) several information processing units for parallel execution of tasks or specific sequence of instructions such as data coding and signal processing. Detailed description of the zones, and their involvement in instruction execution, therefore, might help to estimate the performance of embedded device.

17.5.3 Performance Estimation—Instruction Timing

To do performance estimation of embedded and any other computer system, we have to assume following points:

- Mix of instructions with weight of all types of instructions defining the system performance.
- Time required to perform any instruction—it is worth to separate AZ, IZ, and PZ when instruction is executed.
- Note that instructions that use IZ (external) part are not defined in completion time.

Thus, a mixture of presented types of instructions with taken into account an amount of each of them is a good reflection of overall hardware performance.

In other words, hardware performance might be calculated by summarized length of the instructions required to process user task. Let us consider what is AZ and others in terms of overheads:

$$T_{\text{instruction execution}} = \text{Tz} + \text{Tiz} + \text{Tpz} \tag{17.4}$$

Obviously, the shorter the cycle of instruction, the better the instructions of A, B, C, D, E, F, and G types that are preferable. It is possible to calculate what difference one might face if use H, I, and K types of instructions, but it is clear that 10^{-9}, 10^{-5}, and 10^{-6} do not fit together well.

It is worth to analyze active zone performance in terms of instruction execution phases:

- Loading,
- Decoding an instruction,
- Preparation of operands,
- Execution of the instruction, and
- Loading back modified operand

Loading means that instruction from memory goes to the load register in parallel with increment of program counter (inside processor) and goes through decoding scheme that activates requires activation operands and execution device ALU.

Decoding means that instruction (loaded from memory) is placed in the internal register (not always addressable) inside processor and by special timing diagram section all hardware links activated: for ALU, operand's addresses and signs (such as N, Z, V, C).

Preparation of operands means that input ports are activated and buffers for instruction data are enabled. For big instructions, it means that instruction execution will be postponed until all memory cycles required to upload operands in processor are complete.

Execution of instruction assumes that ALU (or combination of them) are ready and perform instruction from operation field (ADD, SUB, etc.) in assumption that required operands are ready and delivered from memory (or registers) to internal buffers.

Thus, performance of active zone might be estimated as

$$Perf_{AZ} = \frac{1}{T_{AZ}} \tag{17.5}$$

while

$\mathrm{Taz} = \mathrm{T}_{\mathrm{loading}} + \mathrm{T}_{\mathrm{decoding}} + \mathrm{T}_{\mathrm{preparation\ of\ operands}} + \mathrm{T}_{\mathrm{execution}} + \mathrm{T}_{\mathrm{loading\ back}}$

Pretty much the same one might do with passive and interfacing zones.

Let us have a look for illustrative purposes how instruction is executed by recoverable processor (ReP) made by ITACS Ltd. and prototyped with partial support by grant of FP6 www.onbass.org (Fig. 17.14).

At first from instruction register through control unit via control bus microinstruction goes to either AU or LU. At the same time, microinstruction about

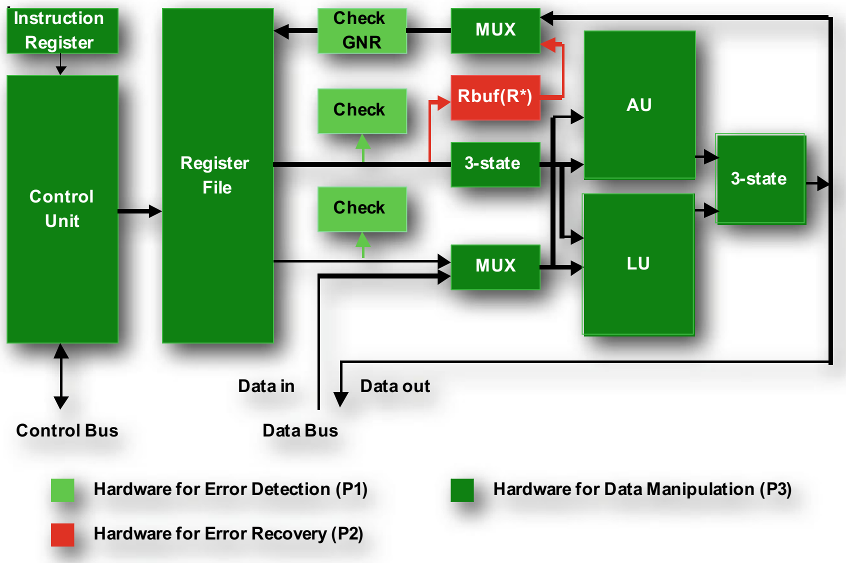

Fig. 17.14 REP structure

operands of instruction goes to register file and chosen operands go via three-state buffers down to either AU or LU. When all signals are arrived and timing diagram cycle is competed, information processed and arrived at the last right three-state buffer.

When confirmation from checking schemes arrives, i.e., no errors are detected, the right three-state buffer will release information to deliver either back to register file or through interfacing zone out of processor to the memory or bus.

The picture of prototype of ReP (Fig. 17.15) might be useful to analyze from the performance point of view and organization of embedded system.

Clear visible active zone and passive zone as well as board traces define performance of ReP. Interesting bit of this design is limitation of instructions complexity; only simple instruction to get or put data from and to outside world, store and read data from active zone, and process data inside active zone are allowed.

Thus, when we need fast calculation, processor is performing it with maximum frequency, without waiting of memory of IZ. In turn, clear read/write instructions limit slow down from interaction with external world.

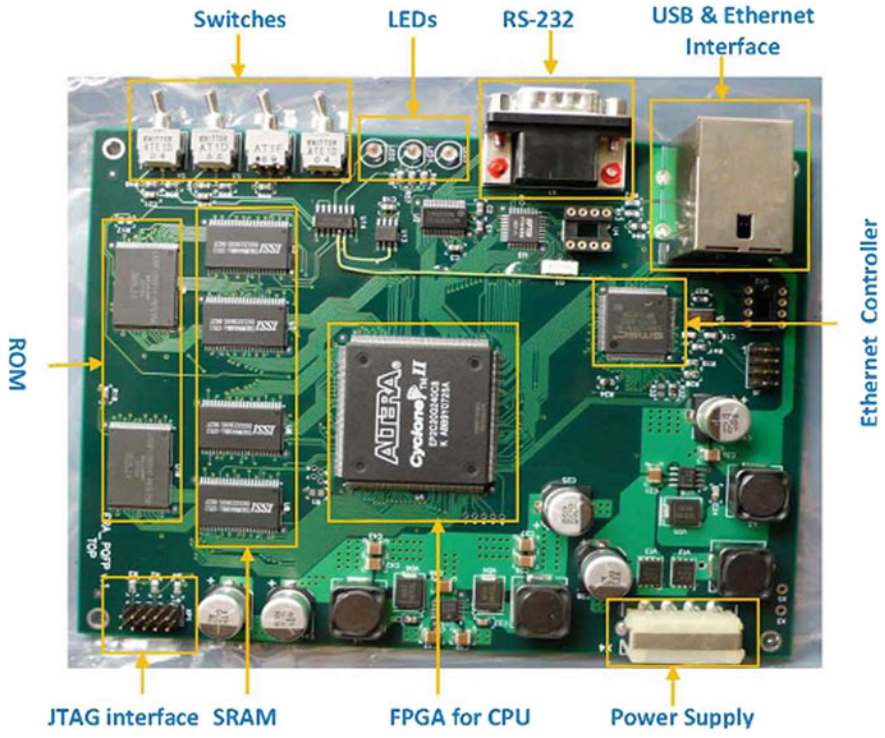

Fig. 17.15 Prototype of ReP

17.5.4 Standard Performance Tests

When we need to have a standard performance estimation, one might use SPECint.

SPECint is designed to measure processor speed and compare various hardware. There is a special company to look after this:

SPEC = Standard Performance Eval. Corp., their website is www.spec.org

What is widely known at the moment is SPECint2006.

SPECint2006 includes 12 tests (all in C/C++)—perlbench, gcc, bzip2, …, xalancbmk.

One can create performance comparison of embedded devices of processor overall 12 tests. Then, designed system might be 12-dimensional evaluation of the known system. Therefore, application segment relative performance might be known. For more details about standard performance, see

http://www.spec.org/cpu2006/results/cint2006.html.

17.5.5 Real Processor Hardware Comparison

The performance of processor is reduced if instruction uses intensively data traffics between memory and active zone. Today, a top-of-the-line x86 microprocessor runs at 3–4 GHz, while the memory runs at 100–500 MHz.

The time to execute an instruction inside a CPU is almost negligible in contrast with access memory time. In other words, the performance of architecture depends on memory traffics.

Proposed recently by ITACS performance of ReP is evaluated by the comparisons with 80 × 88 architecture and MIPs' instruction architecture.

Simple instructions dominate this list and are responsible for 96% of the instructions executed. These percentages are the average of the five SPECint92 programs [7].

Various mixes of instruction cover different applications. However, we can evaluate the CPU performance based on the average case as illustrated in Table 17.1. Within this evaluation, we assume loading data and storing data into memory takes equal time.

The execution time comparison between 80 × 86 and ReP is shown in Table 17.2.

MIPs dynamic instruction mix for five SPECint2000 programs (gap, gcc, gzip, mcf, and perl) is presented in [7]. The execution time comparison between MIPs and ReP is shown in Table 17.3.

Table 17.4 compares the performance of MIPs and ReP.

Modern embedded systems need efficient cost–performance ratio and very often minimize power consumption. One of the design options is as it was mentioned above the design as much as possible and therefore reduce the hardware overheads.

Table 17.1 Comparative ratio of memory use for ReP and 80 × 86

No.	80 × 86 instruction		ReP instructions		Integer average (% total executed) (%)
	Instructions	Memory traffic	Instructions	Memory traffic	
1	Load	2L	LD Ri, Rj	2L/2	22
2	Conditional branch	1L	XOR Mask, Rj	1L/2	20
			CBR Ri, Rj	1L/2	
3	Compare	1L	CND, Ri, Rj	1L/2	16
4	Store	1L, 1S	ST Ri, Rj	(1L + 1S)/2	12
5	Add	1L	ADD Ri, Rj	1L/2	8
6	And	1L	AND Ri, Rj	1L/2	6
7	Sub	1L	SUB Ri, Rj	1L/2	5
8	Move	1L	MV Ri, Rj	1L/2	4
9	Call	1L	MV Ri, Rj	1L/2	1
			CBR Ri, Rj	1L/2	
10	Return	1L	MV Ri, Rj	1L/2	1
			CBR Ri, Rj	1L/2	
Total					96

Table 17.2 Comparison of execution time for 80 × 86 and ReP

Processor and instruction mix	Ratio
80 × 86 22%* 2L+20%*1L+16*1L +12*2L +8*1L +6*1L +5*1L +4*1L +2*1L	=1.66
ReP 22%* L+20%*1L+(16*1L +12*2L +8*1L +6*1L +5*1L +4*1L)/2 +2*1L	

Table 17.3 Comparative ratio of memory use for MIPs and ReP

MIPs instruction		ReP instructions		Integer average (%)
Instruction	Memory traffic	Instruction	Memory traffic	
Load	2L	LOAD	2L/2	26
Add	1L	ADD	1L/2	19
Cond branch	1L	Compare	1L/2	12
		XOR	1L/2	
		CBR	1L/2	
Store	1L, 1S	STORE	(1L+1S)/2	10
Or	1L	OR	1L/2	9
Compare	1L	Compare	1L/2	5
And	1L	AND	1L/2	4
Sub	1L	SUB	1L/2	3
Xor	1L	XOR	1L/2	3

(continued)

Table 17.3 (continued)

MIPs instruction		ReP instructions		Integer average (%)
Instruction	Memory traffic	Instruction	Memory traffic	
Load imm	1L	LOAD	2L/2	2
Shift	1L	SHIFT	1L/2	2
Cond move	1L	COMPARE	1L/2	1
		XOR	1L/2	
		CBR	1L/2	
		MOVE	1L/2	
Jump	1L	MV	1L/2	1
		CBR	1L/2	
Call	1L	MV	1L/2	1
		CBR	1L/2	
Return	1L	MV	1L/2	1
		CBR	1L/2	
Total				99

Table 17.4 Comparative performance of MIPs and ReP

Processor and instruction mix	Ratio
MIPs 26%* 2L+19%*1L+12*1L +10*2L +9*1L +5*1L +4*1L +6*1L +8*1L	=1.67
ReP 26%* L +(19*1L +36L +10*2L +18*1L +6*1L)/2 +5*1L	

Thus, complex instructions of most processors for embedded systems are implemented by a sequence of simple instructions. For embedded systems, there is therefore developed set of performance evaluation, so-called embedded benchmarks.

17.5.6 Embedded Benchmarks

Benchmarks for embedded computing systems vary accordingly applications due to RT, HRT, PW, and CW performance requirements where RT stands for real time, HRT means hard real time, PW is power-wise and CW is cost-wise performance requirements, respectively.

That is why material presented above is important. There is one a bit more known benchmark developed by Embedded Microprocessor Benchmark Consortium (EEMBC). It has five categories: automotive/industrial, consumer, networking, office automation, and telecommunications.

Selecting Table 17.5 what this benchmark is (taken from Hennessy book recommended earlier), full site consists of 34 kernels in five classes.

Table 17.5 EEMBC benchmark suite

Benchmark type	This type	Example benchmark
Automotive/ industrial	16	Six microbenchmarks (arithmetic operations, pointer chasing, memory performance, matrix arithmetic, table lookup, bit manipulation), five automobile control benchmarks, and five filters or FFT benchmarks
Consumer	5	Five multimedia benchmarks (JPEG compress/decompress, filtering, and RGB conversions)
Networking	3	Shortest-path calculation, IP routing, and packet flow operations
Office automation	4	Graphics and text benchmarks (Bezier curve calculation, dithering, image rotation, text processing)
Telecommunication	6	Filtering and DSP benchmarks (autocorrelation, FFT, decoder, and encoder)

17.6 Relative Performance Gain—Amdahl's "Law"

Relative gain in performance usually called "Amdahl's law". Well, law in terms of science, not society, is "a regularity in the material world" (Shorter Oxford English Dictionary, 6e, Vol 1). Thus, naming simple proportion of performance after improvement P_{ai} with performance before improvement P_{bi} is, to put it politely, too ambitious.

$$Speedup = \frac{P_{ai}}{P_{bi}} \tag{17.6}$$

But this proportion is useful to evaluate success of the modification of processor structure after re-iterative design. What is interesting here is that arithmetic expectation of linear growth of performance by improving element performance (Figs. 17.1 and 17.2) has nothing near to the real situation.

17.6.1 Distributed Computing

In the late 1960s, an idea for the parallelization of computer program using distributed computing paradigm instead of single-processor scheme was proposed [8].

It was declared that parallelization of tasks and programs and use of available distributed hardware for support of parallel execution is the most feasible way to boost system performance.

Later, Sun [9] introduced "system fallacies" of distributed computing (Table 17.6). Omitting topologic factors and paying attention to Fallacy 2, 3, and 7, we discover that these fallacies fit into the area of parallel, closely connected computers with multiprocessors—in fact, all modern computers.

Table 17.6 Sun fallacies of distributed computing

1	The network (distributed system) is reliable
2	Latency is zero
3	Bandwidth is infinite
4	The network is secure
5	Topology doesn't change
6	There is one administrator
7	Transport cost is zero
8	The network is homogeneous

If we look harder, these fallacies might be not strong enough and some of the declared features described became obsolete.

Besides, again, when definition includes eight other elements that are not connected or have vague relation to each other, it seems odd or at least inconsistent.

If we follow Sun definition, we are not including Internet into the distributed computing even as a supportive hardware infrastructure. Anyway, we've proposed our own définition of distributed computing:

Definition 2 Distributed computing is a paradigm that assumes an execution of functionally connected tasks as a single process over distributed media and resources.

Clearly, a joint collaborative work of thousands of processors at once might bring substantial profit for both loosely connected tasks (when they share HW resources, but not logically connected, such as Google cluster), or closely tight models that include of several thousands of DE.

But in the second case, it is much harder to get the gain from distributed computing, and it is not a surprise.

Amdahl described drawbacks of distributed computing in the late 1960s [8], highlighting that even small parts of a program must be parallelized to reach their full potential. This way linear growth of speedup is not possible at all.

In other words, if 1 is a length of a sequential program and we have managed to parallelize p fraction of it, then sequential part is shrinking down to 1 − p, while parallel part requires p/n time where n stands for number of processors, (3) and Fig. 17.16.

$$S = \frac{1}{1 - p + p/n} \tag{17.7}$$

17.6.2 *Real Performance and Amdahl "Law"*

The proportion Eq. 17.7 is useful to evaluate a success of the modification of processor structure in re-iterative design. What is interesting here is that the expectation of linear growth of performance by improving element performance (Figs. 17.1 and 17.2) has nothing near to the real situation.

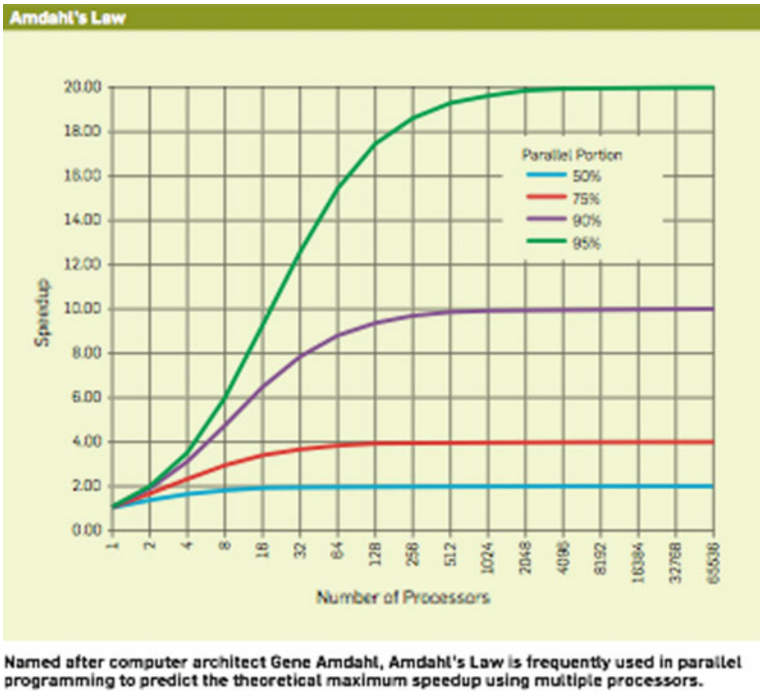

Fig. 17.16 System speedup by Amdahl [8]

It means that if we make super parallel execution of 80% of a program, we still have to complete another 20% sequentially. The number of speedups versus number of processors as a family of functions is presented in Fig. 17.16 taken from [8]

17.6.3 A Fine-Tuning of Parallel Speedup Model

The theory behind computational work in parallel has some limitations that reduce the advantages of parallelization. Usually, the goal in large-scale computation is to get as much work done as possible in the shortest time within the budget.

Furthermore, the system can be considered good and well-designed when it is able to get a big job done in less time, or a bigger job done in the same amount of time without any problem; in other words, a system should be a scalable.

Therefore, the power of a computational system can be represented as the amount of computational work done, divided by the total time it takes to do it. It is important to emphasize that usually the aim is to increase power per unit cost, or more importantly nowadays, cost–benefit, and in this regard physics and economics conspire to limit the raw power of individual single-processor systems available to perform any particular piece.

It is agreed within the research community that the cost–benefit scaling of increasingly power single-processor systems is usually nonlinear and very poor. For instance, one processor that is twice as fast might cost four times as much, yielding only half the cost–benefit per pound.

Physics sets its own limit as well—a so-called "thermal barrier" [5]—an amount of heat that material is capable to dissipate is limited making endless increase of frequency of operation impossible.

These two arguments are usually applied to justify alternative solutions and development of parallel designs. There are some drawbacks though, as Amdahl pointed out, and they are serious.

Let us rewrite Amdahl ratio in terms of time: T(N) will be the time necessary to finish the task on N processors. The speedup S(N) is expressed by the ratio (Eq. 17.8):

$$S(N) = \frac{T(1)}{T(N)} = \frac{Ts + Tp}{Ts + Tp/N} \tag{17.8}$$

In many cases, the time T(1) possesses, as represented above, both the serial part Ts and the parallelable part Tp.

Unfortunately, Amdahl ratio ignores a role of runtime system tasks (see first section of this chapter) that must be considered when a parallel execution is assumed.

A more detailed analysis of parallel speedup would include two more parameters of interest, namely,

- Ts—the original single-processor serial time;
- Tis—the average additional serial time spent performing, for example, inter-processor communication (IPCs), see Fig. 17.1, where it is introduced as EIZ, setup, and so forth in parallelized tasks. It is important to note that this time can depend on N in a variety of ways; nonetheless, the simplest assumption is that each system has to spend this much time one after the other, so that the additional serial time is, for example, N*Tis;
- Tp—the original single-processor parallelable time;
- Tip—the average additional time spent by each processor performing just the setup and work that it does in parallel; this may as well include idle times, which is also very important and should be accounted for separately.

The most important element that contributes to Tis is the time required for communication between the parallel subtasks. This communication time is always there—even in the simplest parallel models where identical jobs are farmed out and run in parallel on a cluster of networked computers, the remote jobs must begin and be controlled with message passing over the system.

In systems with more complex jobs, partial results developed on each CPU may have to be sent to all other CPUs in the distributed computing system for the calculation to proceed, which can be very costly in scaled time. The (average)

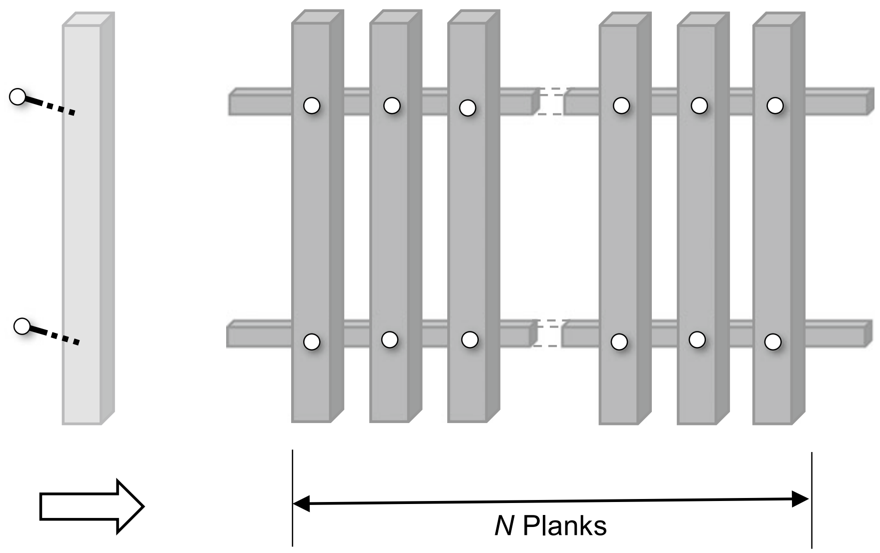

Fig. 17.17 Fence model of processing

additional serial time (Tis) plays an extremely important role in defining the speedup scaling of a given calculation.

Most computer systems process information sequentially. Lines of code in a computer program get translated into assembly language by the compiler, and the latter gets decoded into microcode in the processor. Everything and every step along the way is done sequentially. For example, a flowchart processing usually includes multiplication or comparison of two digits; it starts with the first digit, and then the second digit is introduced and the working register is set to 0.

To explain what is real and what is not and why Amdahl rule is mostly misleading, we have developed a simple model—so-called "fence making model", illustrated in Fig. 17.17 and following expert recommendations [10].

17.6.4 Parallel Versus Sequential: A Fence Model

Our task is to make a fence with *N* planks and two horizontal rails; each plank needs two nails and has to be "preprocessed". Two rails have to be placed at the assembling site. Each plank needs to be placed at site and finally nailed. We also need hammers and nails and sequence and instruction to operate.

Task requirements: number of planks *N*; number of rows—2. Each plank needs to be nailed half-way through before placement for final processing and assembling a fence.

There are two principally different options to make this fence:

(A) by distributing tasks;
(B) by making all tasks on site sequentially.

In case (A), distributing task scheme assumes the existence of agents–workers and distributers and their abilities to act:

- *N* workers for plank processing are available and ready;
- a distributor of the nails is in place;
- a distributer of the hammers is in place;
- a distributer of the planks is in place;
- a distributer of rails is in place;
- a collector of the fence segments initially is and placing the planks;
- nailing the planks at two rows are performed by workers;
- collecting the hammers is performed; and
- garbage collector is in place and completes the task execution.

Case (B), in turn, assumes that the same worker is doing all actions, like "a jack for all trade", has one hammer, bucket of nails, and does the following:

- takes nails;
- planks where they are;
- half-nail planks;
- places them on the rails;
- nails them all;
- place fence where necessary, collect garbage.

Let us consider the process of making the fence from *N* planks in more details for both cases, assuming that nails, hammers, planks, and rails are ready and placed in the local warehouse (storage and executed by "a system officer", while workers execute user task). Sequences are presented in Table 17.7.

Our task now is about giving elementary time slot t_e and constant coefficients equal for both variants of fence processing to prepare two variants of the fence completion as a sequence of steps for A and B cases. This will illustrate a gain from distribution of works.

We need to compare these cases as well as explain what is possible to prepare in preprocessing and what is possible only during operation. One might find useful to make a table of all works mentioned and using own experience and case estimate a concrete gain for concrete case.

Now we have to answer the following questions:

When distributed computing is efficient in comparison with sequential;

What impact system software makes on parallelization of task and efficiency of a system.

It is clear that *planks* are data, *nails* and *hammers* are programs to process data on site, *and distributer* is runtime system.

Table 17.7 Parallel versus sequential execution in more details

Parallel operation	Sequential operation
Distributor	*Distributor*
Gets pack of planks	Activate worker
Distribute planks	Check garbage left
Distribute rails	
Distribute nails	
Distribute hammers	
Distribute planks along rails	
Activate *N* workers start	
Collect hammers and left garbage	
Place two rails in assembling area	
Clean garbage	
Worker	*Worker*
Receive planks	Gets packs of planks
Receive nails	Gets buckers of nails
Receive hammer	Gets a hammer
Preprocess plank (two nails nailed half-way through)	Places (distribute) planks to the assembling area
Spread planks along rails (fine-tuning)	Places rails in assembling area
Nail plank (two nails) to the rails at the final assembling	Preprocess *N* planks (two nails per each)
Prepare to final assembling	Places (distribute) planks along the rails
	Nails *N* planks Assemble fence Clean garbage

Let us leave an arithmetic exercise with various values of parameters from job descriptions above to good master students.

Our estimation indicates that overheads of runtime system for distributed execution might achieve almost 60% of user task cost (time). We add in denominator of (5) a coefficient *k*, a relative value of system software overheads per user task (Eq. 17.9):

$$y = \frac{1}{(1-p)+k+\frac{p}{x}}, x = \{1, 2, \ldots, 10\}, p = \{0.85\}, k = \{0, 0.1, 0.4\} \qquad (17.9)$$

Following Eq. 17.9, the graph of Fig. 17.16 presents three curves in three colors: green, blue, and red k = 0, 0.1, 04, respectively. The top one stands for known "pure" Amdahl ratio (k = 0).

Figure 17.18 shows that for extremely good runtime system, one can double performance with 4 cores. It is still too optimistic statement, especially recalling Multics 85% and Window 65% of total workload time.

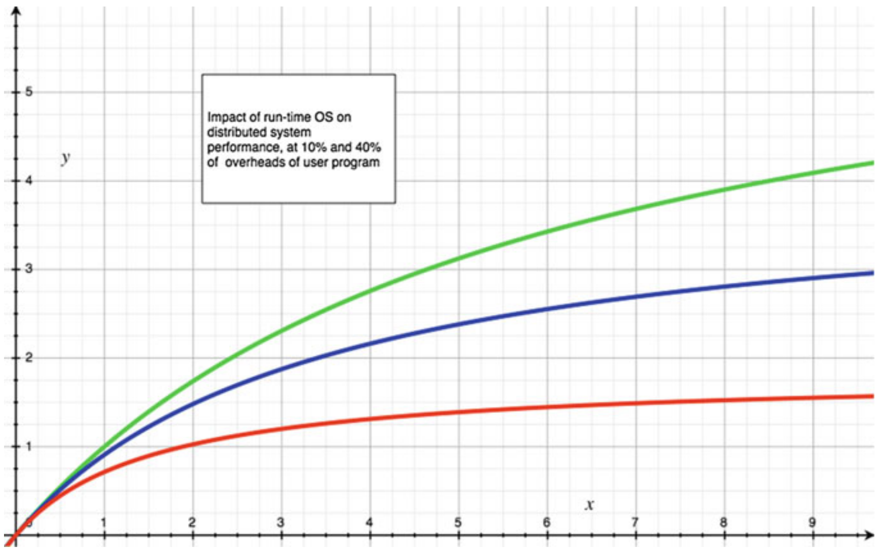

Fig. 17.18 System software role in distributed computing

17.7 Conclusion

- Hierarchy of performance models is proposed from the point of view of information processing.
- Shown that calculation of efficiency of computer system should include a role of user and system software as well as hardware.
- Model of hardware from the point of view of information processing is proposed.
- Calculation of performance of embedded system hardware is presented.
- Comparative study of proposed embedded architecture with most known architectures is presented.
- Role of complexity of instruction set on performance is briefly discussed.
- Brief description of what kind of benchmarks is used for performance evaluation of new and existing systems is presented.
- In details, using fence manufacture model shown that Amdahl Law is overoptimistic at an order of magnitude. It requires to consider the role of system software and algorithms ability of parallel computing.

References

1. Schagaev I (1990) Yet another classification of redundancy. In: IMEKO 7th symposium technical diagnostics, 17–19 Sept 1990, Helsinki, pp 485–491
2. Schagaev I (1990) Instruction sets and their role for computer architectures (in Russian). Electronics Publication
3. Sogomonyan ES, Schagaev IV (1988) Hardware and software of fail-safe computing systems. Automat I Telemech **2**:3–39
4. Schagaev I (2001) CASSA—concept of active system safety for aviation. In: IFAC automatic control in aerospace 2001 a proceedings of the 15th IFACS symposium Bologna/Forli, Italy, 2–7 September 2001
5. Blaeser L, Monkman S, Schagaev I (2014) Evolving systems. In: Resilient computer system design. Springer. ISBN 978-3-319-15069-7
6. Schagaev I. Active system control design of system resilience. https://doi.org/10.1007/978-3-319-46813-6. ISBN 978-3-319-46812-9
7. Hennesy J, Patterson D (2003) Computer architecture: a quantitative approach. Morgan Kaufmann Publishers Inc., San Francisco. ©2003 ISBN:1558607242
8. Amdahl GM (1967) Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the April 18–20, 1967, spring joint computer conference, AFIPS ‘67 (Spring), pp 483–485. https://doi.org/10.1145/1465482.1465560
9. https://blogs.oracle.com/jag/resource/Fallacies.html
10. www.doityourself.com/stry/buildwoodfences