UNIVERSITY OF JOENSUU
DEPARTMENT OF COMPUTER SCIENCE
Report Series A

# Architectural differences of efficient sequential and parallel computers

Martti Forsell

Report A-2000-4

ACM C.1.0; C.4
UDK 681.3. 02
ISSN 0789-7316
ISBN 951-708-949-X

# Architectural differences of efficient sequential and parallel computers

Martti J. Forsell

*Department of Computer Science, University of Joensuu, PB 111, FIN-80101 Joensuu, Finland*
*Email : mforsell@cs.joensuu.fi*

November 15, 2000

*Abstract:*

In this article we try to conclude what kind of a computer architecture is efficient for executing sequential problems, and what kind of an architecture is efficient for executing parallel problems from the processor architect's point of view. For that purpose we analytically evaluate the performance of five sequential architectures and seven parallel architectures representing widely both commercial and scientific processor designs attached to efficient memory systems. The results are interesting: Sequential problems are executed most efficiently in a computer using a minimally pipelined VLIW processor and a memory system, which is as fast as possible. Parallel problems are executed most efficiently in a computer using deeply superpipelined, chained, and multithreaded processors and a high throughput pipelined memory system. Thus, designing a computer for efficient sequential computation leads to very different architecture than designing one for efficient parallel computation.

*Keywords:* Computer architecture, sequential computation, parallel computation

---

## 1. Introduction

Until now computer designers and architects have mostly used architectures and technical solutions originally designed for sequential computation assuming some degree of instruction level parallelism for both sequential and parallel computation. As a result of this, current computers aren't optimal in solving sequential problems nor in solving parallel problems.

In this article we try to conclude what kind of a computer architecture is efficient for executing sequential problems, and what kind of an architecture is efficient for executing parallel problems from the processor architect's point of view. For that purpose we analytically evaluate the performance of five sequential architectures and seven parallel architectures representing widely both commercial and scientific processor designs attached to efficient memory systems.

The results are interesting: Sequential problems are executed most efficiently in a computer using a minimally pipelined

VLIW processor and a memory system, which is as fast as possible. Parallel problems are executed most efficiently in a computer using deeply superpipelined, chained, and multithreaded processors and a high throughput pipelined memory system. Thus, designing a computer for efficient sequential computation leads to very different architecture than designing one for efficient parallel computation.

### 1.1 Comparable studies

There exists already some research efforts in this area of modeling performance of different type of computer systems:

- Optimal length of pipelining in supercomputers [Kunkel86]

- Characterization of Branch and Data Dependencies in Programs for Evaluating Pipeline Performance [Emma87]

- Evaluating Performance Tradeoffs Between Fine-Grained and Coarse-Grained Alternatives [Dubey95]

There is, however, still room for this kind of research, because earlier studies e.g. omit certain advanced processor architectures, like those described in subsections 2.2.5 and 3.2.7, use difficult-to-program message passing paradigm, and assume limited capacity communication systems.

### 1.2 Organization of this article

The rest of this article is organized so that in Section 2 we describe the model of sequential computation, analytically evaluate five most interesting processor architectures attached to an efficient memory system for executing sequential problems, and consider signal propagation and silicon lay-out issues of sequential hardware. In Section 3 we describe the model of parallel computation, analytically evaluate seven interesting processor architectures attached to an efficient memory system for executing parallel problems, and consider signal propagation and silicon lay-out issues of parallel hardware. In Section 4 we compare the performance of the described architectures by calculating the execution times of both a sequential problem and a parallel problem as a function of several parameters that have effect on performance. Finally, in Section 5 we give our conclusions.

## 2. Sequential computation

### 2.1 The model

A computational problem is *strictly sequential*, if it consists of a chain of $O$ operations, in which each operation is dependent on the result of the preceding operation.

It is very rare that a problem is strictly sequential, because all operations, except the first one, must be arithmetic and logical operations. Therefore we extend the definition to include situations where the problem is otherwise strictly sequential, but it is allowed to contain load, store, and branch operations. In that case, we say that the problem is *sequential.*

By the term *sequential computation*, we refer to a situation where a computer capable of executing single thread executes a problem that is sequential. Note that this

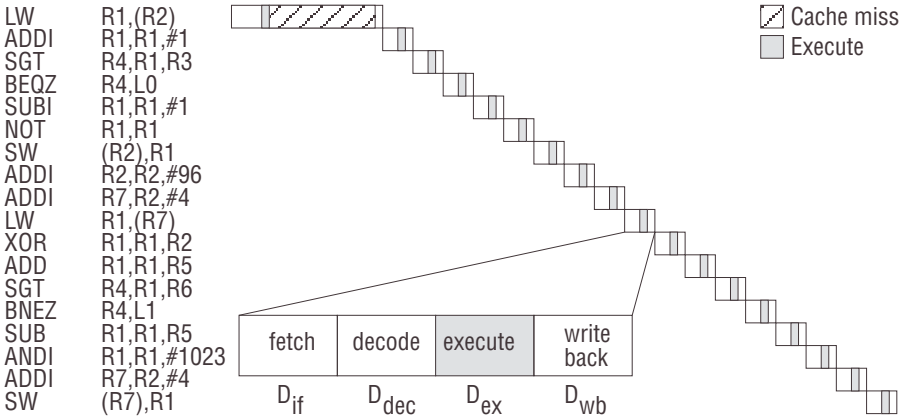


**Figure 1.** Non-pipelined processor S1 executing simple sequential program.

definition allows for any type of instruction-level parallelism as long as only single thread of control is used.

### 2.2 Sequential hardware

In this subsection we consider five sequential computer architectures, which represent both commercial and scientific processor architecture alternatives aimed for sequential computation. For each architecture we determine the minimum clock cycle time and the execution time of a sequential problem.

For simplicity we consider synchronous execution of integer operations only, because using floating point operations would generate much instruction level parallelism, almost all processors are currently synchronous, and asynchronous processors are still in developmental state [Werner97].

We assume also that all sequential architectures use efficient hierarchized memory system, i.e., at least one level of cache memories, because almost all remarkable sequential computers do so, main memory access times are currently much longer than cycle times of processors, and it is trivial to see that using efficient memory hierarchy increases performance. The effects of possible instruction cache misses are omitted in this study, because they affect the performance of all sequential architectures almost equally.

In our sequential architectures, the number of additional clock cycles required due to a data cache miss is

$$C_{cm} = \lceil D_{mem}/D_{cycle} \rceil - 1,$$

where $D_{mem}$ is the delay caused by accessing main memory and $D_{cycle}$ is the minimum delay of a processor clock cycle.

#### 2.2.1 Non-pipelined

A non-pipelined processor (S1) executes instructions sequentially one after another (see Figure 1). It is assumed that the instruction set of the processor is designed such that each instruction corresponds to precisely one operation. The phases of execution are instruction fetch, instruction decode, operation execute, and write results to registers. Each phase cause a constant delay ($D_{if}$, $D_{de}$, $D_{op}$, and $D_{wb}$

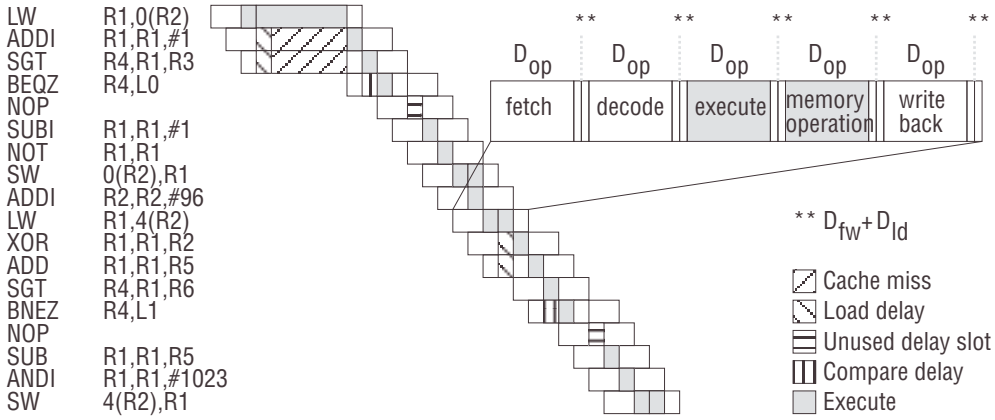


**Figure 2.** Basic pipelined processor S2 executing simple sequential program.

respectively) to signals passing through these phases. Thus, the minimum cycle time for a non-pipelined processor is

$$D_{cycle} = D_{if} + D_{dec} + D_{op} + D_{wb} .$$

The execution time of an instruction is constant with the exception of memory references that miss the data cache causing $C_{cm}$ cycle penalty. The probability that an operation is a memory reference is denoted by $P_m$ and the probability that a memory reference misses cache is denoted by $P_{cm}$. With these assumptions, the execution time of a sequential problem consisting of $O$ operations can be obtained from the following equation:

$$T_o = O\, D_{cycle}(1 + P_m P_{cm} C_{cm})$$

### 2.2.2 Basic pipelined

In a basic pipelined processor (S2) execution of instructions is divided into five parts (instruction fetch, instruction decode, operation execute, memory reference, and write back to registers), such that different parts of subsequent instructions can be executed in parallel (see Figure 2). Note that functional units performing address calculations and memory references are organized as a chain. It is assumed that the basic pipelined processor uses forwarding to decrease the number of pipeline hazards caused by data dependencies of the program, and delayed branches to decrease penalties caused by branching. In that sense basic pipelined processor resembles closely first MIPS, SPARC and DLX [Hennessy90] processors.

Due to overlapped execution of instructions, the phases of execution can be seen as execute (i.e. one of the instruction fetch, instruction decode, operation execute, memory reference and write back to registers), forward the results of the operation, and latch the forwarded results into interstage latches. Each phase cause a constant delay ($D_{op}$, $D_{fw}$, and $D_{ld}$ respectively) to a signal passing through these phases. The constant $D_{op}$ can be obtained from the equation

$$D_{op} = \max(D_{if}, D_{dec}, D_{op}, D_{cr}, D_{wb}) ,$$

where $D_{cr}$ is the delay caused by data cache reference. Thus, the minimum cycle time of a basic pipelined processor is

$$D_{cycle} = D_{op} + D_{fw} + D_{ld} .$$

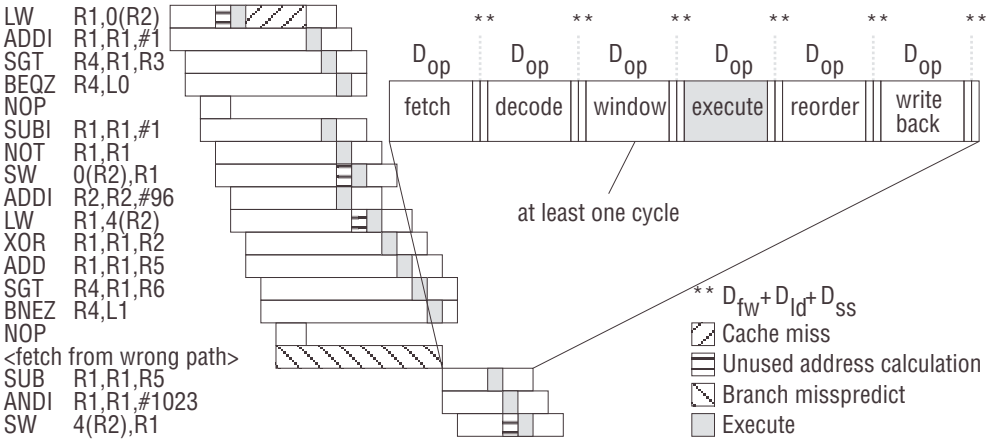


**Figure 3.** Out of order superscalar processor S3 executing simple sequential program.

The reasons for causing delays to the rate of one instruction per one cycle are cache misses, branches (by both a compare instruction preceding a branch and an empty delay slot after the branch), and read operations in general. The probability that an operation is a branch is denoted by $P_b$, the probability that an operation is an independent branch (e.g. jump) is denoted by $P_{bnd}$, and the probability that an operation is a memory read is denoted by $P_{rd}$. The execution time of a sequential problem consisting of $O$ operations is then

$$T_o = O\,D_{cycle}(1 + P_m P_{cm} C_{cm} + 2(P_b - P_{bnd}) + P_{rd}) .$$

### 2.2.3 Out-of-order superscalar

An out-of-order superscalar processor (S3) is simultaneously able to execute at most $F$ instructions in $F$ functional units, i.e., in parallel pipelines (see Figure 3). The processor has advanced execution logic, that places fetched and decoded instructions into an instruction window, and selects in the beginning of each clock cycle which instructions in the window can be executed during that cycle. This causes an additional delay in cycle time denoted by $D_{ss}$. Thus, we obtain

$$D_{cycle} = D_{op} + D_{fw} + D_{ld} + D_{ss}$$

for the out-of-order superscalar architecture in hands. The effect original execution order is emulated after the execution phase by using reorder buffer before writing results to registers. Due to long pipeline and high number of functional units, a complex forwarding network is required to avoid delays caused by data dependencies. Penalties caused by control dependencies between instructions are reduced by using hardware branch prediction. This kind of solutions are used e.g. in PA-RISC and superDLX [Hennessy90, Moura93].

The reasons for causing delays to the rate of one instruction per one cycle are cache misses, branches and useless address calculations. The number of operations to be executed sequentially reduces to $O(1 - P_{mnd} - P_{bnd})$, where $P_{mnd}$ is the probability that an operation is an independent memory reference. That is because out-of-order superscalar processor is able to execute at most $F$ instructions in parallel if they are independent. The execution time

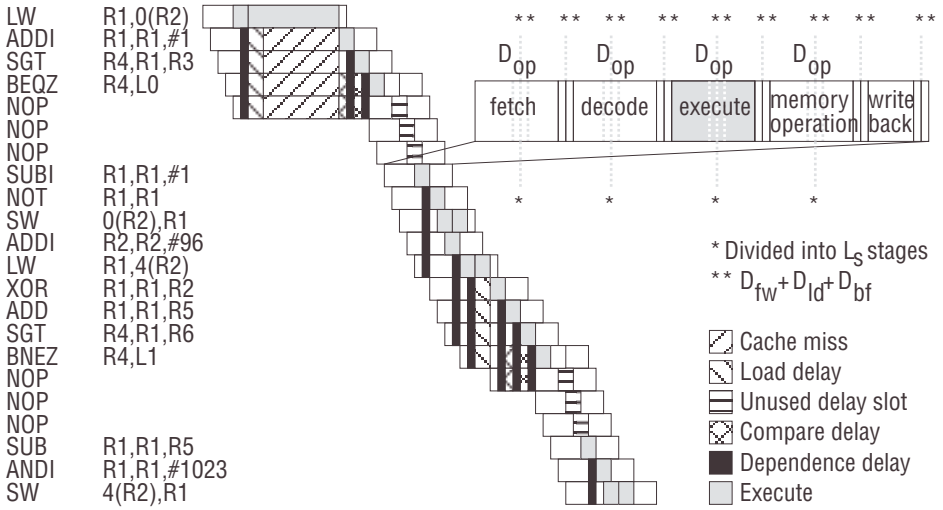


**Figure 4.** $L_S$-level superpipelined processor S4 executing simple sequential program.

of a sequential problem consisting of $O$ operations is

$$T_o = O(1 - P_{mnd} - P_{bnd}) D_{cycle} (1 + P_m P_{cm} C_{cm} + P_b P_{mp} C_{mp} + P_{uea}),$$

where $P_{uea}$ is the probability that an instruction causes an useless empty address calculation and $P_{mp}$ is the probability that a branch is predicted incorrectly.

### 2.2.4 $L_s$-level superpipelined

In an $L_s$-level superpipelined processor (S4) execution of instructions is divided into parts just like in the basic pipelined processor except that the division is deeper: The standard phases (instruction fetch, instruction decode, operation execute, memory reference, and write back to registers) are further divided into $L_s$ parts (see Figure 4).

The processor uses similar techniques as a basic pipelined processor, only the forwarding network gets quadratically more complex and the number of delay slots associated to branches increases as $L_s$ increases. Superpipelining is used e.g. in new Alpha, MIPS, and Pentium processors.

The additional delay included in cycle time is a balancing factor denoted by $D_{bf}$, because the standard phases of execution may not evenly divide into $L_s$ parts:

$$D_{cycle} = D_{op}/L_s + D_{bf} + D_{fw} + D_{ld}$$

The reasons causing delays to the rate of one instruction per one cycle are cache misses, branches (by both a compare instruction preceding a branch and empty delay slots after the branch), and read operations in general. In addition to that, all $O(1 - P_{mnd} - P_{bnd})$ fully dependent operations cause pipeline hazards of length of $L_s$ cycles. The rest $O(P_{mnd} + P_{bnd})$ independent operations can be executed in the rate of one cycle per instruction, because they do not cause pipeline hazards. The execution time of a sequential problem consisting of $O$ operations is

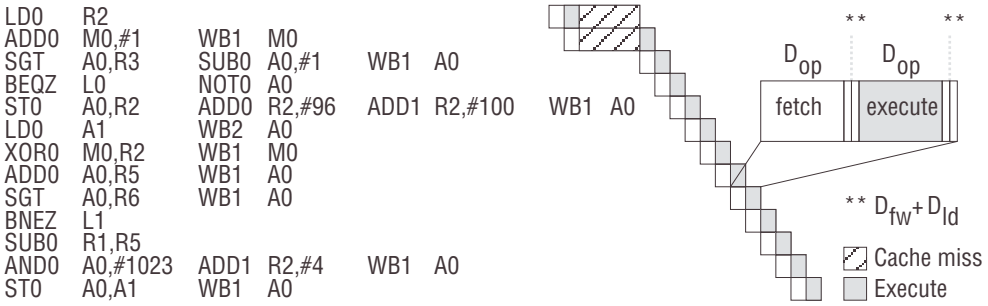


**Figure 5.** Two-level pipelined VLIW processor S5 executing simple sequential program.

$$T_o = O(L_s(1 - P_{mnd} - P_{bnd}) + (P_{mnd} + P_{bnd}))\, D_{cycle}(1 + P_m P_{cm} C_{cm} + 2(L_s - 1)(P_b - P_{bnd}) + L_s P_{rd}) .$$

### 2.2.5 Two-level pipelined VLIW

A two-level pipelined VLIW processor (S5) is a novel alternative to out-of-order superscalar processor [Forsell96b]. Like out-of-order superscalar processor it is able to execute at most $F$ instructions in parallel in $F$ functional units (see Figure 5). It features special two-level pipeline and uses techniques like general forwarding, simple instruction set, uncoded instructions, and fast branching to minimize delays caused by data and control dependencies between instructions [Forsell96b]. The operation of the processor requires a complex forwarding network and the minimum cycle time for a two-level pipelined processor is

$$D_{cycle} = D_{op} + D_{fw} + D_{ld}$$

just like in the case of the basic pipelined processor.

The only reasons causing delays to rate of one instruction per one cycle are cache misses. The number of operations to be executed sequentially reduces to $O(1 - P_{mnd} - P_{bnd})$, because as a VLIW processor it is able to execute at most $F$ instructions in parallel if they are independent. The execution time of a sequential problem consisting of $O$ operations is

$$T_o = O(1 - P_{mnd} - P_{bnd})\, D_{cycle}(1 + P_m P_{cm} C_{cm}) .$$

## 2.3 Signal propagation

Consider signal propagation through the operations in a sequential problem. Signal propagation is speed-optimal, if signal propagates through an operation and then immediately proceeds to the next operation. In an execution architecture this means that after executing an operation in a functional unit, the result of the operation must be sent to another functional unit where the next operation will be executed. Due to synchronous nature of execution this functional unit must latch incoming signal to an intermediate register. The fastest way to send signal between multiple functional units is to use crossbar-style connection network, which requires sending the output of all functional units to inputs of all functional units, and selecting the right input from the outputs of all functional units.

In other words, in optimal case signal must go from a functional unit to another through a crossbar-style forwarding network, such that the receiving unit selects the correct input and latches it for execution in next clock cycle. This is just what happens in the two-level pipelined VLIW architecture S5. The other sequential architectures have different types of additional propagation delays which decrease their performance:

- In the non-pipelined architecture S1 the output of an operation must proceed through write back, instruction fetch and decode phases before it reaches the input of a functional unit where the next operation will be executed.

- The basic pipelined architecture S2 is optimal except the fact that executing load and branch instructions generate false dependency delays due to inefficient organization of the pipeline.

- In the out-of-order superscalar architecture S3 the output of an operation is first sent to the instruction window. After that the instruction selection logic selects which at most $F$ instructions will be executed in the next clock cycle. Finally, these instructions are forwarded to functional units allocated for their execution.

- The $L_S$-level superpipelined architecture S4 has similar false dependency delays as basic pipelined architecture, but they occur more frequently. In addition to that, signal propagating through operation execution goes through $L_S$ interstage latches and dividing execution into $L_S$ parts may require delays for balancing the length of the stages for pipelining.

### 2.4 Silicon lay-out issues

The properties that have influence on the efficiency and feasibility of silicon lay-out of a circuit are the number of transistors, the number and length of interconnection lines, the regularity of overall structure, and the fan-in and fan-out of single gates.

Unfortunately the number of lines required by the efficient forwarding network between the functional units increases quadratically as the number of functional unit increases. Also the length of lines required by the forwarding network increases to quite high, because functional units take lot of silicon space. This limits the feasible number of functional units to only few dozens. Out-of-order superscalar architecture requires also lot of silicon space for fetch buffer, instruction window and reorder buffer. Furthermore the overall structure of architectures S3 to S5 is not regular due to forwarding network and instruction buffers used in S3.

## 3. Parallel computation

### 3.1 The model

A computational problem is *theoretically parallel,* if it is not strictly sequential. This definition is, however, not practical for our purposes, because we want to refer to the problems that can be at least partially divided into a number of threads. Therefore we define that a computational problem is *parallel*, if it consists of $T$ sequential threads, each of which consisting of $O/T$ operations. These threads are able to communicate with each others through an interprocessor communication system which can be implemented as a shared

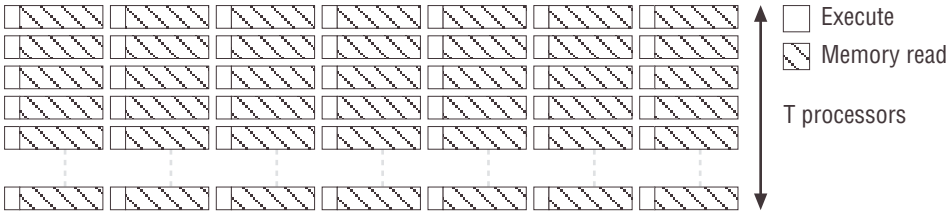


**Figure 6.** T singlethreaded basic pipelined processors P1 executing simple parallel program.

memory or as a message passing network. By the term *parallel computation*, we refer to a situation where a computer capable of executing multiple threads in parallel executes a problem that is parallel.

The scope of this paper is limited to the shared memory model, in which threads are executed stepwise synchronously, because it provides large number of analyzed algorithms, easy programmability, high performance and feasibility [Leighton91, Ranade91, Abolhassan93, Forsell97a]. Note that this model restricts the hardware to deterministic in-order execution without any memory hierarchy in front of the memory system. Out-of-order execution would lose step-wise synchronity immediately, because processors are executing different programs. Similarly, maintenance of the cache coherency between all caches would require very complex hardware with quadratic cost compared to number of processors [Tomasevic93, Forsell94]. Instead, it is possible to place caches in front of each memory module, but the advantages of doing that are not obvious as figured out in [Forsell00].

We assume that the processors are connected to each others with a coated mesh topology, which is efficient enough for handling all the traffic generated by the shared memory simulation on a distributed memory machine [Leppänen95]. The average number of communication system cycles taken by a main memory read is then

$$C_{\mathrm{mem}} = 6\sqrt{\frac{P}{6}} + \lceil D_{\mathrm{mem}}/D_{\mathrm{cycle}} \rceil .$$

### 3.2 Parallel hardware

In this subsection we consider seven computer architectures, which represent both commercial and scientific processor architecture alternatives aimed for parallel computation. For each architecture we determine the minimum cycle time and the execution time of a parallel problem. Like in subsection 2.2 we consider here execution of integer operations only.

#### 3.2.1 Singlethreaded, basic pipelined, single thread per processor

A singlethreaded basic pipelined, single thread per processor parallel computer (P1) has $T$ basic pipelined processors, which are executing one thread each (see Figure 6).

The minimum cycle time for a basic pipelined processor is

$$D_{\mathrm{cycle}} = D_{\mathrm{op}} + D_{\mathrm{fw}} + D_{\mathrm{ld}}$$

as described in subsection 2.2.2.

A processor has to assume that some of the other processors possibly make memory

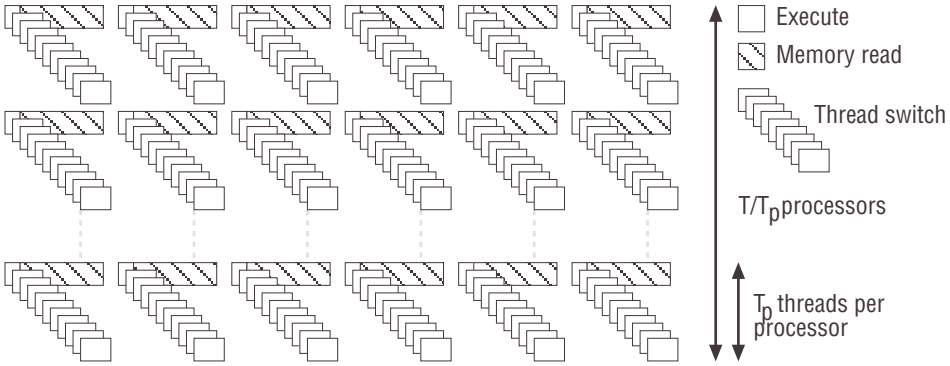


**Figure 7.** T/Tp singlethreaded basic pipelined processors P2 executing simple parallel program.

references, because it can not know what kind of a program they execute. Therefore every operation must be treated like a memory read causing $C_{mem}$ - 1 additional cycles per each operation. In the worst case, the result of this memory read is used by a branch operation which cause additional two cycle dependency delay. Thus the execution time of a parallel problem consisting of $O$ operations is

$$T_o = O\,D_{cycle}(2 + C_{mem})/P\,.$$

### 3.2.2 Singlethreaded, basic pipelined, $T_p$ threads per processor

A singlethreaded basic pipelined $T_p$ threads per processor parallel computer (P2) has $T/T_p$ basic pipelined processors, which are executing $T_p$ threads each (see Figure 7). The reason for executing multiple threads in one processor is an attempt to hide the long latency of the memory system. The minimum cycle time for a basic pipelined processor is

$$D_{cycle} = D_{op} + D_{fw} + D_{ld}$$

as in described in subsection 2.2.2.

To maintain stepwise synchronity processors need to switch thread after each instruction. This causes a delay, which is the maximum of the number of clock cycles required for a thread (or context) switch $C_{csw}$ and the number of cycles required for a memory read reference $C_{mem}$. The execution time of a parallel problem consisting of $O$ operations can be then obtained from the following equation:

$$T_o = O\,D_{cycle}(1 + \max(C_{csw}, C_{mem}))/P\,.$$

### 3.2.3 Multithreaded, non-pipelined

A multithreaded non-pipelined parallel computer (P3) has $T/T_p$ basic pipelined processors, which are executing $T_p$ threads each (see Figure 8). The processors are able to switch thread in every clock cycle without an overhead, which is possible e.g. by using an instruction switch delay to switch also thread [Moore96, Forsell97b].

Like in all parallel computers a processor has to assume that some of the other processors possibly make memory read references, because it can not know what kind of a program they execute. As a result of this, the delay caused by an operation is

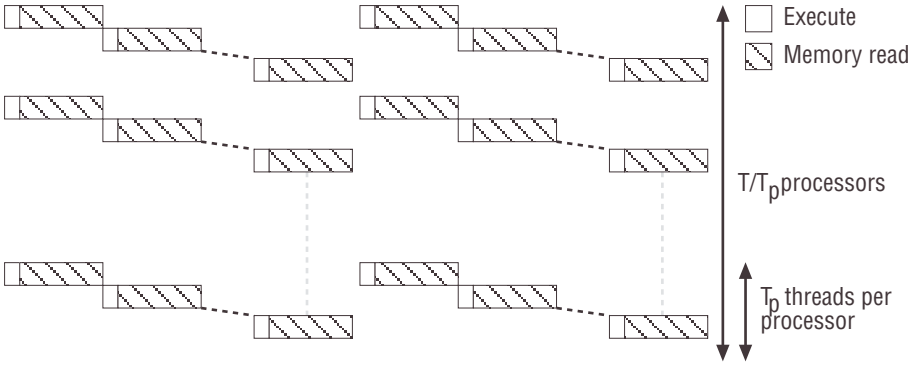


**Figure 8.** T/Tp multithreaded non-pipelined processors P3 executing simple parallel program.

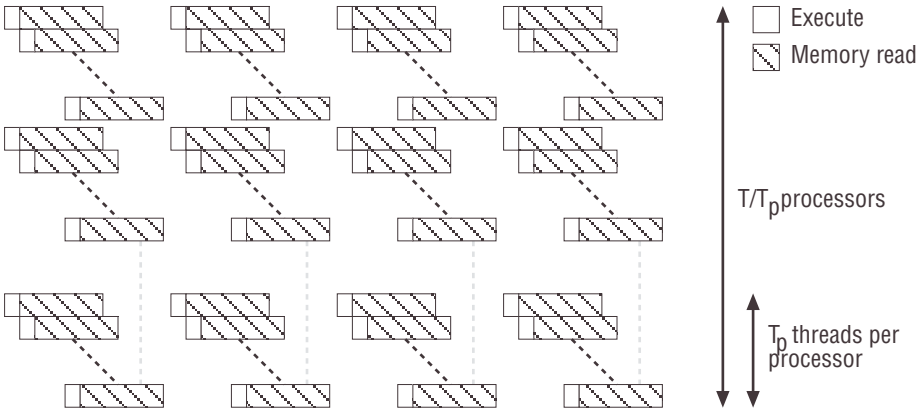


**Figure 9.** T/Tp multithreaded Tp-level interthread pipelined processors P4 executing simple parallel program.

be the maximum of $D_{op}$ and $D_{mem}$. Thus, the minimum cycle time for a multithreaded non-pipelined processor is

$$D_{cycle} = D_{if} + D_{dec} + \max(D_{op}, D_{mem}) + D_{wb}\,.$$

There are no reasons causing delays to the rate of one instruction per clock cycle, because processors are non-pipelined and there are no memory hierarchy. Thus the execution time of a parallel problem consisting of $O$ operations is

$$T_o = O\,D_{cycle}/P\,.$$

### 3.2.4 Multithreaded, $T_p$-level interthread pipelined

A multithreaded $T_p$-level interthread pipelined parallel computer (P4) has $T/T_p$ $T_p$-level interthread pipelined processors, which are executing $T_p$ threads each (see Figure 9). The processors are able to switch thread in every clock cycle without an overhead, but a thread can be re-executed after no less than $T_p$ cycles of time due to $T_p$-level interthread pipelining. There are no pipeline delays, because threads are independent within a cycle, which makes interthread pipelining very efficient technique. The minimum cycle time for a multithreaded $T_p$-level interthread pipelined processor is

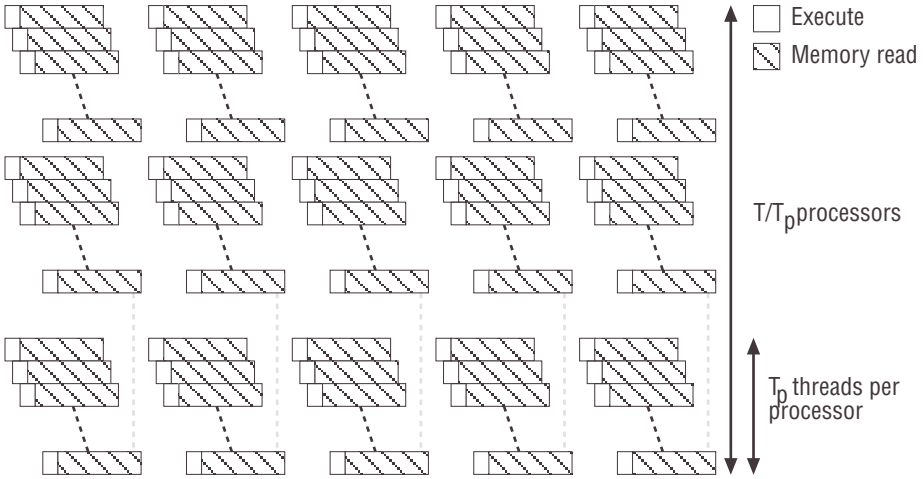


**Figure 10.** T/Tp multithreaded Ls-level interthread superpipelined processors P5 executing simple parallel program.

$$D_{cycle} = D_{op} + D_{fw} + D_{ld} .$$

The property that a multithreaded $T_p$-level interthread pipelined processor can re-execute a thread after no less than $T_p$ cycles causes the utilization of threading $U_t$ to drop below one, if the number of threads in a problem is less than $T_p$ per processor. The utilization can be obtained from the following equation:

$$U_t = \begin{cases} 1, & \text{if } T_p \geq \dfrac{D_{mem}}{D_{cycle}} \\ \dfrac{T_p}{\left(\dfrac{D_{mem}}{D_{cycle}}\right)}, & \text{if } T_p < \dfrac{D_{mem}}{D_{cycle}} \end{cases} .$$

Taking the utilization into account, the execution time of a parallel problem consisting of $O$ operations is:

$$T_o = O\, D_{cycle}/(P\, U_t) .$$

### 3.2.5 Multithreaded, $L_s$-level interthread superpipelined

A multithreaded $L_s$-level interthread superpipelined parallel computer (P5) has $T/T_p$ $L_s$-level interthread superpipelined processors, which are executing $T_p$ threads each (see Figure 10). The processors are able to switch thread in every clock cycle without an overhead, but a thread can be re-executed after no less than $T_p$ cycles of time due to $L_s$-level interthread superpipelining. The minimum cycle time for a multithreaded $L_s$-level interthread superpipelined processor is

$$D_{cycle} = D_{op}/L_s + D_{bf} + D_{fw} + D_{ld}$$

as described in subsection 2.2.4. The execution time of a parallel problem consisting of $O$ operations is

$$T_o = O\, D_{cycle}/(P\, U_t)$$

like in the case of multithreaded Tp-level interthread pipelined architecture.

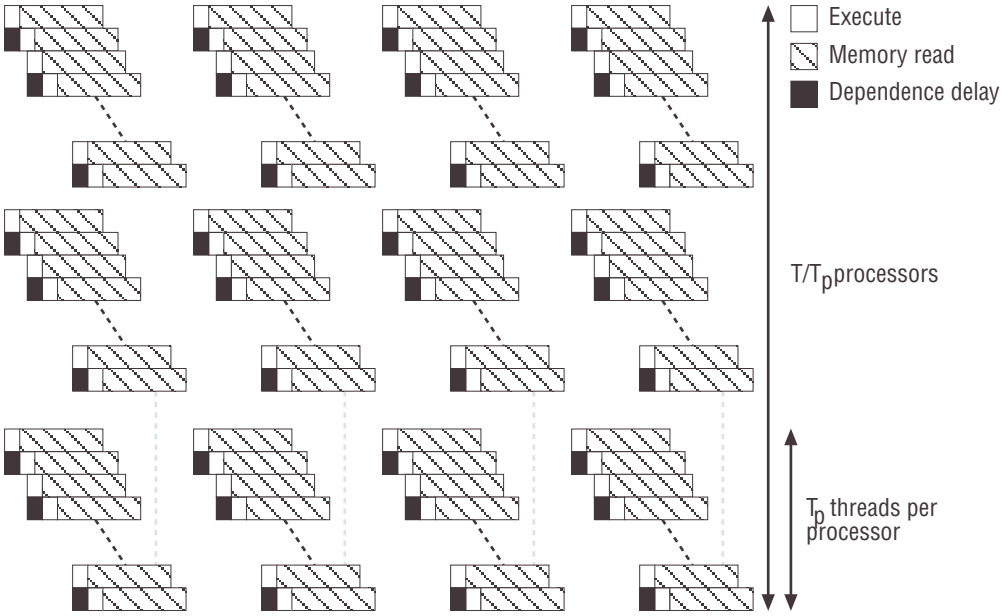


**Figure 11.** T/Tp multithreaded Ls-level interthread superpipelined F functional unit processors P6 executing simple parallel program.

### 3.2.6 Multithreaded, $L_s$-level interthread superpipelined, $F$ parallel functional units

A multithreaded $L_s$-level interthread superpipelined parallel computer (P6) has $T/T_p$ $L_s$-level interthread superpipelined processors with VLIW-style statically scheduled $F$ parallel functional units, which are executing $T_p$ threads each (see Figure 11). The processors are able to switch thread in every clock cycle without an overhead, but a thread can be re-executed after no less than $T_p$ cycles of time due to $L_s$-level interthread superpipelining. The functional units of each processor are organized to operate in parallel.

The minimum cycle time for a multithreaded $L_s$-level interthread superpipelined with $F$ parallel functional units processor is

$$D_{cycle} = D_{op}/L_s + D_{bf} + D_{fw} + D_{ld}$$

as described in subsection 2.2.4.

The number of operations to be executed sequentially reduces to $O(1 - P_{mnd} - P_{bnd}) / P$, because processors are able to execute at most $F$ instructions in parallel if they are independent. Taking the reduced number of sequential operations and the utilization of threading into account, the execution time of a parallel problem consisting of $O$ operations is

$$T_o = O(1 - P_{mnd} - P_{bnd}) D_{cycle}/(P U_t) .$$

### 3.2.7 Multithreaded, $L_s$-level interthread superpipelined, $L_c$-level chained

A multithreaded $L_s$-level interthread superpipelined, $L_c$-level chained parallel computer (P7) is a novel architecture designed specially for shared memory style parallel computation [Forsell97a, Forsell97b] (see Figure 12). It has $T/T_p$ $L_s$-level

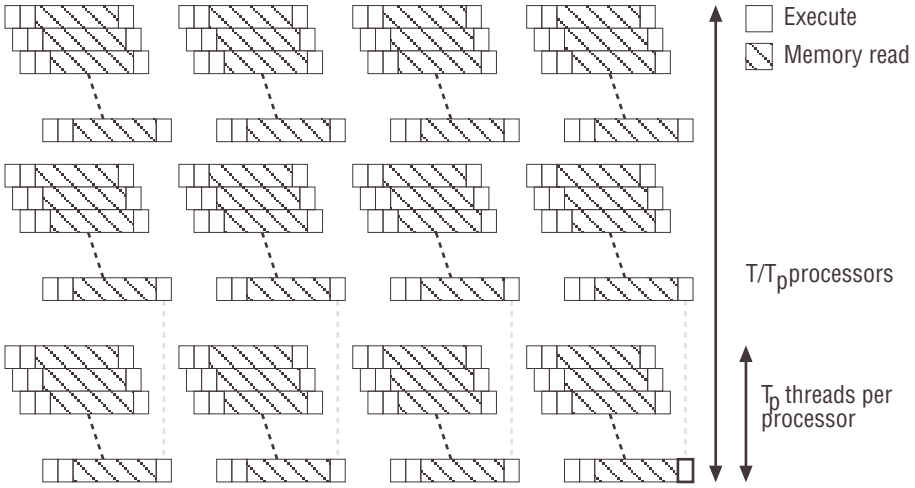


**Figure 12.** T/Tp multithreaded Ls-level interthread superpipelined Lc-level chained processors P7 executing simple parallel program.

interthread superpipelined $L_c$-level chained processors, which are executing $T_p$ threads each.

The processors are able to switch thread in every clock cycle without an overhead, but a thread can be re-executed after no less than $T_p$ cycles of time due to $L_s$-level interthread superpipelining. The functional units of each processor are organized to operate in chain such that a functional unit can use the results of its predecessors.

The minimum cycle time for a multithreaded Ls-level interthread superpipelined $L_c$ -level chained processor is

$$D_{cycle} = D_{op}/L_s + D_{bf} + D_{fw} + D_{ld}$$

as described in subsection 2.2.4.

The number of operations to be executed sequentially is divided by term $L_c\ U_c$, where $U_c$ is the utilization of chaining. That is because processors are able to execute at most $L_c$ instructions in chain, if functional units are suitable for operations to be executed in them. Note that $U_c$ is a function of $L_c$ [Forsell97b]. Taking the utilization of chaining and the utilization of threading into account, the execution time of a parallel problem consisting of $O$ operations can be obtained from the following equation:

$$T_o = O\ D_{cycle}/(P\ U_t\ L_c\ U_c)\ .$$

### 3.3 Signal propagation

In the context of parallel computation signal propagation does not have the same importance as it has with sequential computation, because parallel slackness can be used to balance the time taken by execution and communication. Efficient use of parallel slackness requires that the number of threads per processor $T_p$ is at least the number of cycles required by a memory read reference $C_{mem}$. This together with zero overhead multithreading make it possible to pipeline memory accesses such that a processor can execute other threads while it waits for the completion of a memory reference for this thread.

| | | |
|---|---|---|
| $O = 10^{12}$ | $D_{op} = 800$ | $P_c = 0.4$ |
| $T = 10^6$ | $D_{wb} = 300$ | $P_{cm} = 0.25$ |
| $C_{csw} = 1000$ | $D_{ss} = 100$ | $P_m = 0.3$ |
| $C_{mp} = 3$ | $F = 8$ | $P_{mnd} = 0.2$ |
| $D_{mem} = 64\,000$ | $L_c = 2$ | $P_{mp} = 0.2$ |
| $D_{dec} = 700$ | $L_s = 2$ | $P_{rd} = 0.2$ |
| $D_{fw} = 100$ | $P_a = 0.5$ | $P_{uea} = 0.15$ |
| $D_{if} = 800$ | $P_b = 0.2$ | $U_c = 0.6$ |
| $D_{ld} = 100$ | $P_{bnd} = 0.1$ | |

**Table 1.** Parameters and their values used in evaluation.

Architectures P4 to P7 make use of this property. The performance of rest of the architectures is affected by the unbalance of execution and communication.

### 3.4 Silicon lay-out issues

The number of transistors required by processors of multithreaded architectures P4 to P7 is high, but there is no need for on-chip nor of-chip caches saving very much space, the structure operational elements is regular, and the length of lines can be limited by superpipelining in architectures P5 to P7.

## 4. Evaluation

We evaluated the architectures described in sections 2 and 3 analytically by calculating the execution time as the function of memory reference delay $D_{mem}$, the number of functional units $F$, the level of superpipelining $L_s$, the probability that a memory reference causes a cache miss $P_{cm}$, and the probability that a branch is predicted incorrectly $P_{mp}$ using parameters shown in Table 1. These parameters were chosen to represent the properties of current components and technology.

The sequential architectures S1 to S5 were assumed executing a sequential problem consisting of $10^{12}$ operations and the parallel architectures P1 to P7 were assumed executing a parallel problem consisting of $10^{12}$ operations. The problem divides evenly into $10^6$ sequential threads. The results of the evaluation are shown in Figures 13-17.

We can make following observations from these results and equations shown in sections 2 and 3:

(1) Increasing the memory reference delay increases the execution time in sequential architectures respectively. The execution time of the superpipelined architecture S4 seems to increase faster than the execution time of the others due to high number of dependency delays making it the most inefficient one with the reference delays longer than 32 ns and given assumptions. The execution time in parallel architectures P4 to P7 remains the same until the reference delay grows to longer than 550 ns, where utilization of threading drops below one.

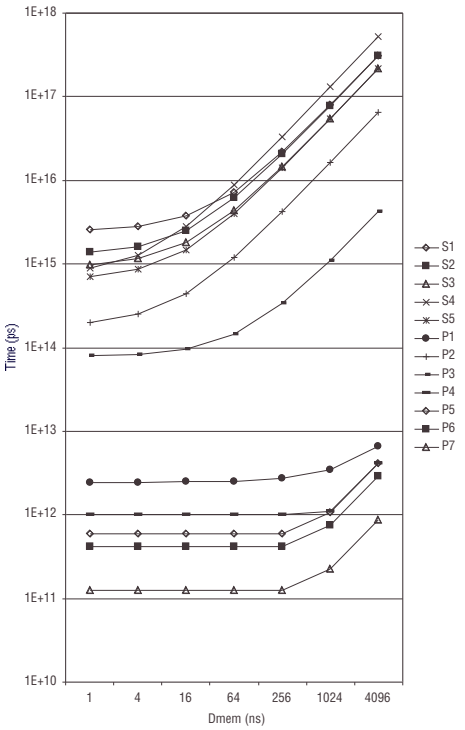


**Figure 13.** Execution time as the function of memory reference delay.

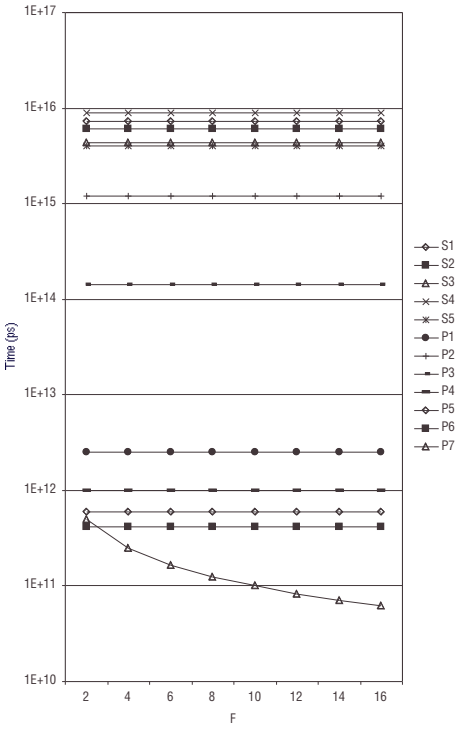


**Figure 14.** Execution time as the function of the number of functional units F.

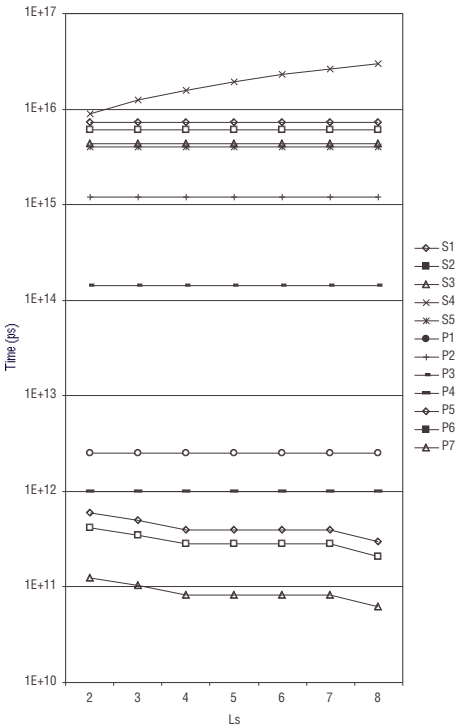


**Figure 15.** Execution time as the function of the level of superpipelining.

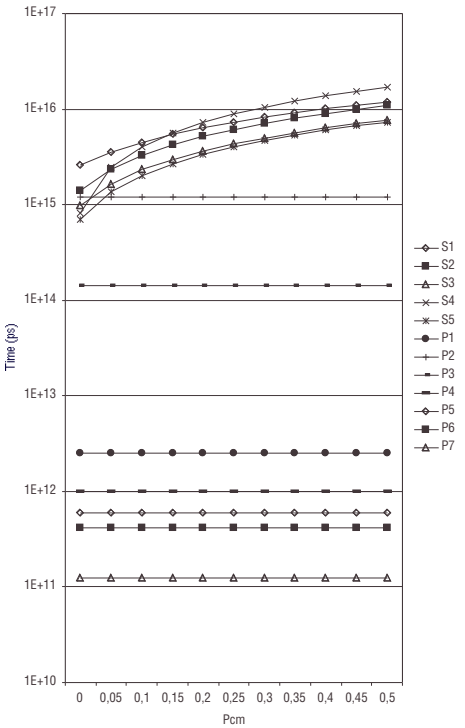


**Figure 16.** Execution time as the function of the probability of a cache miss.

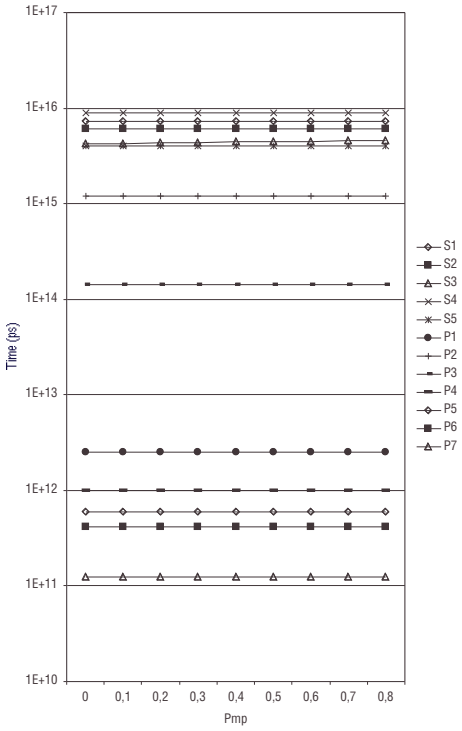


**Figure 17.** Execution time as the function of the probability of a branch misspredict.

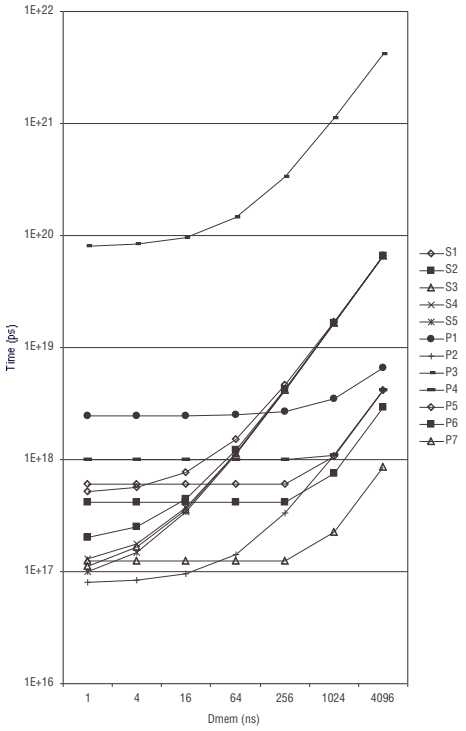


**Figure 18.** Execution time for sequential machines executing parallel problem and vice verse.

(2) Changing the number of functional units has effect only on the execution time of multithreaded, interthread superpipelined, chained architecture P7, because only minimal degree of instruction level parallelism is available. Increasing the number of functional units decreases the execution time in P7, because it increases the degree of interpipeline overlapping.

(3) Increasing the degree of superpipelining increases the execution time of sequential superpipelined architecture S4, but decreases the execution time of superpipelined parallel architectures P5, P6 and P7. The reason for this contrary behaviour is that increasing the degree of superpipelining causes the degree of interpipeline overlapping to increase in parallel architectures, but not in sequential architectures, because they are executing just one thread.

(4) As the probability of a cache miss increases, the execution time of sequential architectures increases respectively. Again, the execution time of the superpipelined architecture seems to increase faster than the execution time of the others. The execution times of parallel architectures are not changed, because they do not have caches.

(5) As the probability of a branch misspredict increases, the execution time of sequential out-of-order superscalar architecture S3 increases. The probability of a branch misspredict does not have effect on the other architectures, because they do not use branch prediction.

| Property/Feature | Sequential computation | Parallel computation |
|---|---|---|
| Optimal depth of pipeline | Minimal (two stages) | Maximal (several hundreds) |
| Increasing the depth of pipeline | Decreases performance | Increases performance |
| Type of pipeline | Interinstruction pipeline | Interthread pipeline |
| Memory hierarchy | Needed | Not needed |
| Speed of memory | As fast as possible | As cost-effective as possible |
| Optimal number of threads in problem | 1 | ≥ number of hardware threads times memory latency |
| Optimal length of functional unit chain | 1 | ≥ number of operations in a basic block of code |
| Increasing the length of functional unit chain | Decreases performance | Increases performance |
| Optimal organization of functional units | Parallel | Sequential chain |
| Internal wiring from silicon engineer's point of view | Irregular, lot of long lines (due to need for fast internal communication) | Regular, all lines can be short (due to pipelining) |
| Most of the silicon area | On-chip caches | Buffers for holding threads |

**Table 2.** Properties of computer architecture execution hardware shown both for sequential computation and parallel computation.

For comparison purposes we evaluated also the execution time as a function of memory system reference delay $D_{mem}$, assuming sequential architectures execute a parallel problem and vice verse (see Figure 18).

(6) The performance of the evaluated architectures decreases dramatically when sequential architectures execute a parallel problem and vice verse: Best parallel architectures are two magnitudes slower in executing a sequential problem than fastest sequential architectures and best sequential architectures are even six to seven magnitudes slower in executing a parallel problem than fastest parallel architectures. Surprisingly, the singlethreaded, basic pipelined $T_p$ threads per processor P2 is the fastest of all parallel architectures, if the memory system delay is shorter than 45 ns.

## 5. Conclusions

We have tried to conclude what kind of a computer architecture is efficient for executing sequential problems, and what kind of an architecture is efficient for executing parallel problems from the processor architect's point of view. For that purpose we have analytically evaluated five sequential architectures and seven parallel architectures representing widely both commercial and scientific processor designs attached to efficient memory systems. The results are interesting: Sequential problems are executed most efficiently in a computer

using a minimally pipelined VLIW processor and a memory system, which is as fast as possible. Parallel problems are executed most efficiently in a computer using deeply superpipelined, chained, and multithreaded processors and a high throughput pipelined memory system. Thus, designing a computer for efficient sequential computation leads to very different architecture than designing one for efficient parallel computation.

More detailed effect of the examined properties on the evaluated architectures is summarized in Table 2. Interestingly, a sequential problem is executed efficiently in a computer applying high-level sequentialism and low-level parallelism and a parallel problem is executed efficiently in a computer applying high-level parallelism and low-level sequentialism.

**References**

[Abolhassan93]
F. Abolhassan, R. Drefenstedt, J. Keller, W. Paul, D. Scheerer, On the Physical Design of PRAMs, Computer Journal 36, 8 (1993), 756-762.

[Cragon92]
H. Cragon, Branch Strategy Taxonomy and Performance Models, IEEE Computer Society Press, Washington, 1992.

[Cragon96]
H. Cragon, Memory Systems and Pipelined Processors, Jones and Bartlett Publishers, Boston, 1996.

[Dubey95]
P. Dubey, G. Adams and M. Flynn, Evaluating Performance Tradeoffs Between Fine-Grained and Coarse-Grained Alternatives, IEEE Transactions on Parallel and Distributed Systems 6, 1 (January 1995), 17-27.

[Emma87]
P. Emma and E. Davidson, Characterization of Branch and Data Dependencies in Programs for Evaluating Pipeline Performance, IEEE Transactions on Computers C-36, 7 (July 1987), 859-875.

[Forsell94b]
M. Forsell, Are Multiport Memories Physically Feasible?, Computer Architecture News 22, 4 (September 1994), 47-54.

[Forsell96a]
M. Forsell, V. Leppänen and M.Penttonen, Efficient Two-Level Mesh based Simulation of PRAMs, Proceesings of the International Symposium on Parallel Architectures, Algorithms, and Networks, June 12-14, 1996, Beijing, 29-35.

[Forsell96b]
M. Forsell, Minimal Pipeline Architecture-an Alternative to Superscalar Architecture, Microprocessors and Microsystems 20, 5 (1996), 277-284.

[Forsell97a]
M. Forsell, Implementation of Instruction-Level and Thread-Level Parallelism in Computers, Dissertations 2, Department of Computer Science, University of Joensuu, Joensuu, 1997.

[Forsell97b]
M. Forsell, MTAC—A Multithreaded VLIW Architecture for PRAM Simulation, Journal of Universal Computer Science 3, 9 (1997), 1037-1055.

[Forsell98]

M. Forsell, M. Penttonen and V. Leppänen, Primitives of Sequential and Parallel Computation, Report 1998/A/3, Department of Computer Science and Applied Mathematics, University of Kuopio, Kuopio, 1998.

[Forsell00]

M. Forsell and V. Leppänen, Memory module structures for shared memory simulation, Report A-2000-5, Department of Computer Science, University of Joensuu, Joensuu, 2000.

[Hennessy90]

J. Hennessy and D. Patterson, Computer Architecture: A Quantitative Approach, Morgan Kaufmann Publishers Inc., Palo Alto, 1990.

[Kunkel86]

S. Kunkel, J. Smith, Optimal Pipelining in Supercomputers, Proceedings of the 13th Symposium on Computer Architecture, June 1986, Tokyo, 404-414.

[Leighton91]

T. F. Leighton, Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes, Morgan Kaufmann, San Mateo, 1992.

[Leppänen95]

Performance of work-optimal PRAM simulation algorithms on coated meshes, the Computer Journal 38, 10 (1995), 801-810.

[Moore96]

S. Moore, Multithreaded Processor Design, Kluwer Academic Publishers, Boston, 1996.

[Moura93]

C. Moura, SuperDLX - A Generic Superscalar Simulator, ACAPS Technical Memo 64, McGill University, Montreal, 1993.

[Ranade91]

A. Ranade, How to Emulate Shared Memory, Journal of Computer and System Sciences 42, (1991), 307--326.

[Tomasevic93]

M. Tomasevic and V. Milutinovic, A Survey of Maintenance of Cache Coherence in Shared Memory Multiprocessors, Proceedings of the Hawaii International Conference on System Sciences, Koloa, January 5-8, 1993.

[Werner97]

T. Werner and V. Akella, Asynchronous Processor Survey, Computer 30, 11 (November 1997), 67-76.