Institute of Software Technology

Department of Programming Languages and Compilers

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master Thesis Nr. 3554

Language Independent Modelling of Parallelism

Nazmul Alam

Course of Study:	INFOTECH
Examiner:	Prof.Dr.rer.nat./Harvard Univ. Erhard Plödereder
Supervisor:	Dipl.-Inf. Mikhail Prokharau
Commenced:	August 23, 2013
Completed:	May 09, 2014
CR-Classification:	C(1.1,1.2,4b),D(1.3b,2.5f,3.4f,3.4g,4.1,4.2d,4.7e,4.8)

Abstract

To make programs work in parallel contexts without any hazards, programming languages require changes to their structures and compilers. One of the most complicated parts is memory models and how programming languages deal with memory interactions. Different processors provide a different level of safety guarantees (i.e. ARM provides relaxed whereas Intel provides strong guarantees). On the other hand, different programming languages provide different structures for parallel computation and have individual protocols for communicating with parallel processes. Unfortunately, no specific choice is best in all situations. This thesis focuses on memory models of various programming languages and processors highlighting some positive and negative features from the point of view of programmability, performance and portability. In order to give some evidence of problems and performance bottlenecks, some small programs have been developed. This thesis also concentrates on incorrect behaviors, especially on data race conditions in programs, providing suggestions on how to avoid them. Also, some litmus tests on systems featuring different vendors' processors were performed to observe data races on each system. Nowadays programming paradigms also became a big issue. Some of the programming styles support observable non-determinism which is the main reason for incorrect behavior in programs. In this thesis, different programming models are also discussed based on the current state of the available research. Also, the imperative and functional paradigms in different contexts are compared. Finally, a mathematical problem was solved using two different paradigms to provide some practical evidence of the theory.

Acknowledgments

I would like to express my gratitude to Prof. Dr. rer. nat./Harvard Univ. Erhard Plödereder for creating the opportunity and allowing me to do my master's thesis at the Department of Programming Languages and Compilers, Institute of Software Technology, University of Stuttgart.

Moreover, I would like to express the deepest appreciation to my supervisor Dipl.-Inf. Mikhail Prokharau who has continuously supervised, suggested and corrected different problems with his great enthusiasm and passion throughout my whole thesis work. He also provided guidance and productive discussions for structuring of my thesis work. Furthermore, he also supported me with vital suggestions of corrections all the way through the writing process of my thesis.

In addition, my sincere appreciation to (Dipl.-Ing. FH) Klemens Krause and Ms. Kornelia Kuhle for their continuous support. They ensured a flawless working environment for this thesis by instant support and provision of tools, information and required access permissions to the server and computer lab.

I would like to thank the Department of Programming Languages and Compilers, especially those members who provided some constructive suggestions after my intermediate presentation of this thesis, as well as my colleagues who were doing their master's thesis in this department at the same time with me.

Finally, and most importantly, I would like to thank Miss Mounomita Nasreen for her support, encouragement and always having a warm-hearted ear. Without her unconditional love, I could not have completed this study. I would like to give special thanks to my parents for their faith in me and allowing me to be as ambitious as I wanted. It was under their watchful eye that I gained so much drive and the ability to tackle challenges head on. I also thank all of my friends for their support in all the ways during my master's study.

Contents

List of Figures

List of Tables

1. Basic Concepts

1.1 Definitions

1.1.1 Memory Model

A memory model is a specification of the memory system for hardware or software system that will appear to programs. It eliminates the gap between the expected behavior and the actual behavior supported by the system [6].

1.1.2 Atomic Variable

A thread concurrently reading the object from shared memory will see the old value or the new value, never see any value in-between. In order to ensure this, an `atomic` type variable may have stricter alignment than a plain type variable[100].

1.1.3 Data Race

A **Data race** occurs when multiple threads want to access same shared memory location during the execution, and at least one of them tries to modify that location. It may introduce unpredictability, unexpected results that are often hard to detect[76].

1.1.4 Sequential Consistency (SC)

A multiprocessor based parallel system is called sequentially consistent if, the result of any execution is the same as if all processors executed their operations in some sequential order and the operation result for each processors appears in this sequence the way the program specified the order[67].

1.1.5 Sequential Consistency for Data Race Free Programs (SC-DRF)

If a program does not allow data races and synchronizes with only the following options-

- Sequentially consistent atomic, and
- Use acquire and release semantics for `Lock` and `Unlock` operations respectively.

-then the program follows sequential consistency order of execution, i.e. the result for any execution appears as if the program executed sequentially interleaving its actions letting all read operations see the preceding value stored to the location in this interleaving sequence[4].

1.2 Why Threads?

A thread executes as a small sequence of program instructions. An operating system scheduler can manage it independently.
Threads can change the timing of operations, but threads should not change the semantics of a program. For this reason, thread programming is always a smart solution for program performance issues[54]. Following are some examples where programmer might use threads:

- Lengthy processing: when an operating system is performing some long mathematical calculation, it cannot process any more messages. As a result, the display driver is unable to update data.
- Background processing: Some tasks may need to perform continuously but might not if time is critical. There could be some task which needs to perform continuously with a fixed time interval.
- IO work: DMA (Direct Memory Access) operation or I/O to a memory disc can have unpredictable delays. The programmer confirmed that I/O latency of thread operations does not delay unrelated parts of the application.

Some real-time operations sometimes have limited execution, however, some operations experience unpredictable delay or CPU hogging. In thread programming, tasks with unavailable resources or low priority wait in a waiting-state until resources become available and then are scheduled for execution. Threads also make use of multi-core processor systems in a multi-threaded program. It is not expected that a multi-core processing system will be used by only one application with one thread. Threads also do efficient time-sharing, programmers can make sure that all threads have fair allocation of CPU time by using threads and process priority[54].

1.3 Transformations & Optimizations

A processor does not execute the program that a programmer wrote. Because, there are some other factors that come in between program code and processor execution[9], i.e. Compiler Optimizations, Processor Out of Order *(OoO)* execution, Cache Coherency.

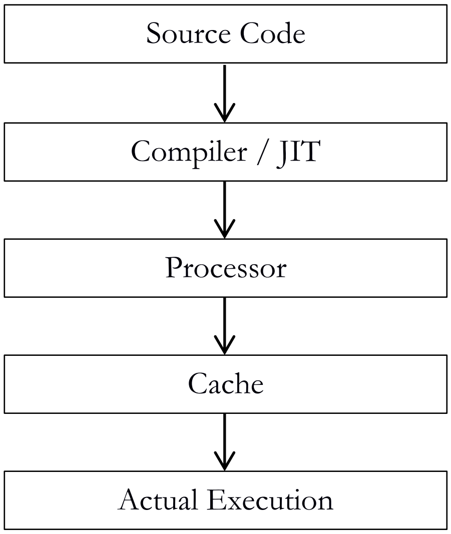


Fig. 1.1: Phase of execution [9]

Usually a programmer cannot tell on which level the optimization happens. The only thing programmers care about is that the program is synchronized correctly.

1.4 Compiler Optimizations

A compiler knows all memory operations in a thread and exactly what they will do, including data dependencies and how to be conservative enough in the face of possible aliasing[9]. On the other hand, a compiler does not know which memory locations are "mutable shared" variables and could change asynchronously due to memory operations in other threads and how to be conservative enough in the face of possible sharing.

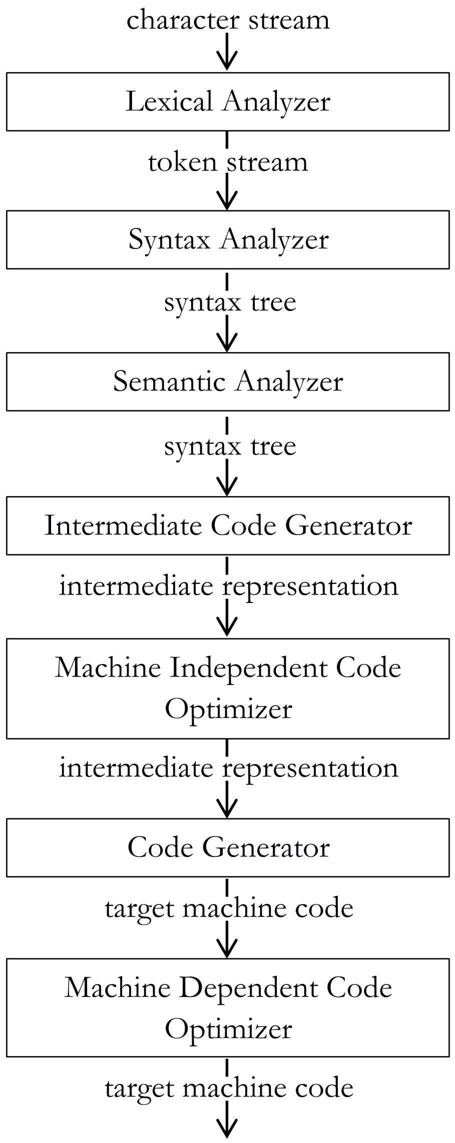


Fig. 1.2: Phases of compiler translation [9]

1.5 Processor Out-of-Order (OoO) execution

In pipelined programming, cycles are sometimes wasted for stalling[9]. In order to reduce the number of stalls in the execution, processor needs to execute the program instructions out of order.

In an out-of-order execution the system processor tries to find the instructions in the instruction stream that are independent of the current (stalled) instruction and can be executed in parallel with it, i.e. the x64 family of processors supports out-of-order execution.

An obstacle is that processors become more complex rapidly as the degree of ”Out-of-Orderness” is increased[9].

2. Various Parallelisation Concepts

The concept of parallel computing is to solve a large computation problem with multiple processing units simultaneously by dividing the problem into small tasks assigned to all processing units. There are several different types of parallel computer architectures and parallel computing styles. Although parallel computing was confined to scientific applications, nowadays it is coming into commercial and business applications to provide high performance computing capabilities for decision support, data mining and risk management applications.

There are several different types of parallel computers and parallel computing (programming) styles that exist from different points of view.

2.1 Types of parallel computers

Parallel computer architectures can be categorized from different points of view, some of them are as follows:

1. Flynn's Classical Taxonomy of computer architectures[45].

   i Single Instruction, Single Data stream (SISD)

   ii Single Instruction, Multiple Data stream (SIMD)

   iii Multiple Instruction, Single Data stream (MISD)

   iv Multiple Instruction, Multiple Data stream (MIMD)

2. Classification according to memory arrangement and communication among Processing Elements (PEs).

   i Shared Memory Multiprocessor

   - Uniform Memory Access (UMA)
   - Nonuniform Memory Access (NUMA)
   - Cache-only Memory Architecture (COMA)

   ii Message passing multiprocessors / Distributed memory multiprocessors.

   iii Hybrid distributed model.

3. According to Interconnection Network.

2.1.1 Flynn's Classical Taxonomy of computer architectures [45]

Since 1966, one of the most popular used classifications is Flynn's classical Taxonomy. It distinguishes multi-processor computer architectures with respect to two independent dimensions: **Instruction Stream** and **Data Streams** : **single** or **multiple**[45].

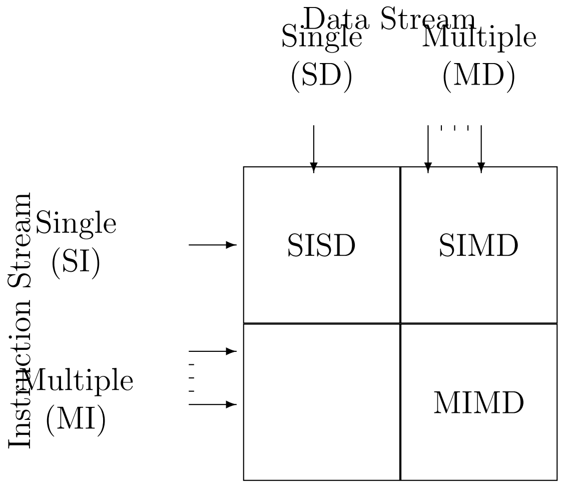


Fig. 2.1: Flynn's Classical Taxonomy of computer architectures

2.1.1.1 **Single Instruction, Single Data stream (SISD)**:

SISD is the standard for uniprocessor computers. CPU can execute only one instruction stream in one clock cycle. So, only one data stream is being used as input during any one clock cycle. The result of execution is deterministic[45].

Load X	timeA
Load Y	
Z = X + Y	
store Z	
X = Y * 2	
store X	

Tab. 2.1: A SISD instruction sequence

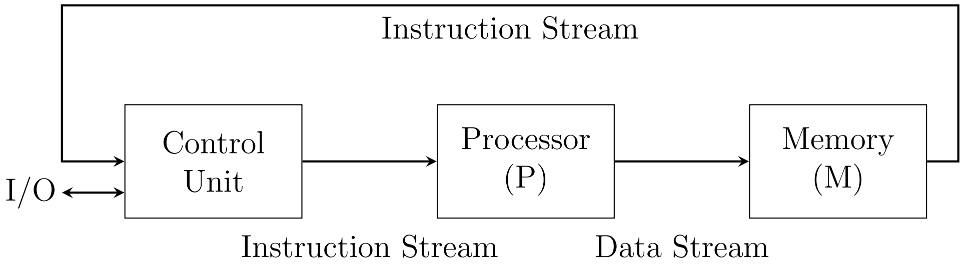


Fig. 2.2: SISD Architecture.

2.1.1.2 **Single Instruction, Multiple Data stream (SIMD)**:

SIMD is a parallel computer architecture. Processors execute the same instruction stream with different data, e.g., graphical machine (RAW to JPEG conversion). This taxonomy is best suitable for solving specially categorized problems characterized by a high degree of regularity, such as graphics/image format conversion. This execution is always synchronous and deterministic[45].

There are two varieties of SIMD:

i Processor arrays: connection machines ILLIAC IV, CM-2, MasPar MP-1 & MP-2.

ii Vector pipelines: Fujitsu VP, Cray X-MP, IBM 9000, Y-MP & C90, NEC SX-2, Hitachi S820, ETA 10

Prev. Inst.	Prev. Inst.	Prev. Inst.	timeA
Load X(1)	Load X(2)	Load X(n)	
Load Y(1)	Load Y(2)	Load Y(n)	
Z(1) = X(1) * Y(1)	Z(2) = X(2) * Y(2)	Z(n) = X(n) * Y(n)	
store Z(1)	store Z(2)	store Z(n)	

Tab. 2.2: A SIMD instruction sequence

The computer with a Graphics Processing Unit (GPU) and most modern computers employ SIMD instructions and execution units.

2.1.1.3 Multiple Instruction, Single Data streams (MISD):

MISD is another parallel computer architecture type. All processing units execute a single data stream independently with separate instruction streams. In real life, only few computers were built using this architecture. One of these is the experimental Carnegie-Mellon C.mmp computer (1971)[110][45].

Prev. Inst.	Prev. Inst.	Prev. Inst.
Load X(1)	Load X(1)	Load X(n)
Z(1) = X(1) * 1	Z(2) = X(1) * 2	Z(n) = X(1) * m
store Z(1)	store Z(2)	store Z(n)
Next Inst.	Next Inst.	Next Inst.

timeA

Tab. 2.3: A MISD instruction sequence

2.1.1.4 Multiple Instruction, Multiple Data stream (MIMD):

This architecture is used in modern types of parallel computers. Every processor can work with a different data stream and every processor can execute a different instruction stream. Synchronous and asynchronous execution, deterministic or non-deterministic execution can co-exist on this architecture. Today's high performance computer architectures are built using this concept. Most of the commonly used personal computers and supercomputers use this architecture[45].

Prev. Inst.	Prev. Inst.	Prev. Inst.
Load X(1)	Call funcD	do 10 i = 1,N
Load Y(1)	X = Y * Z	alpha = W * 3
C(1) = X(1) * Y(1)	sum = X * 2	Zeta = C(i)
store Z(1)	call sub1(i,j)	10 continue
Next. Inst.	Next. Inst.	Next Inst.

timeA

Tab. 2.4: A MIMD instruction sequence

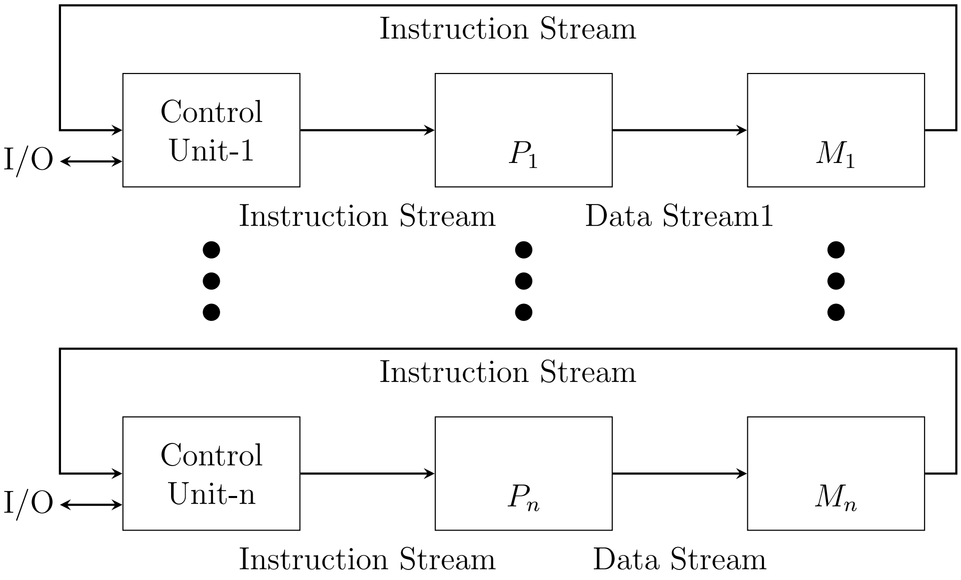


Fig. 2.3: MIMD Architecture.

Many MIMD architectures also include SIMD execution units as sub-components.

2.1.2 According to memory arrangement and communication among Processing Elements(PEs)

From the point of view of interconnection between processor and memory, parallel computer architectures can be classified into following categories.

2.1.2.1 **Shared Memory Multiprocessors**[97]

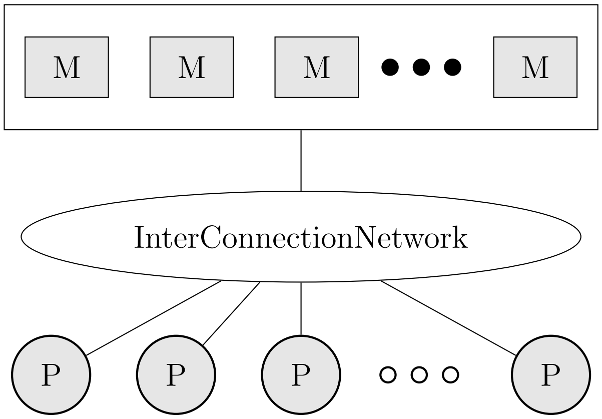


Fig. 2.4: Shared Memory Architecture

In a shared memory multiprocessor architecture, all processors have equal access to the memory module and these memory modules are seen as a single address space by all processors. Each memory module stores data as well as serving to establish communication among the processors via some bus arrangement. Programming in this architecture is quite straightforward and attractive. The executable programming code and data related to the program are stored in memory for each processor to execute. There is no direct processor-to-processor communication involved in the programming process; instead communication is handled mainly via shared memory modules. Access to these memory modules can easily be controlled through appropriate programming mechanisms such as multitasking. However, this architecture suffers from a bottleneck problem when a number of processors endeavors to access global memory at the same time[97]. This limits the scalability of the system.

There are different types of shared memory architectures in existence

- Uniform Memory Access (UMA)[97]
- Nonuniform Memory Access (NUMA) [97]
- Cache-only Memory Architecture (COMA) [52]

Uniform Memory Access (UMA)[97]:

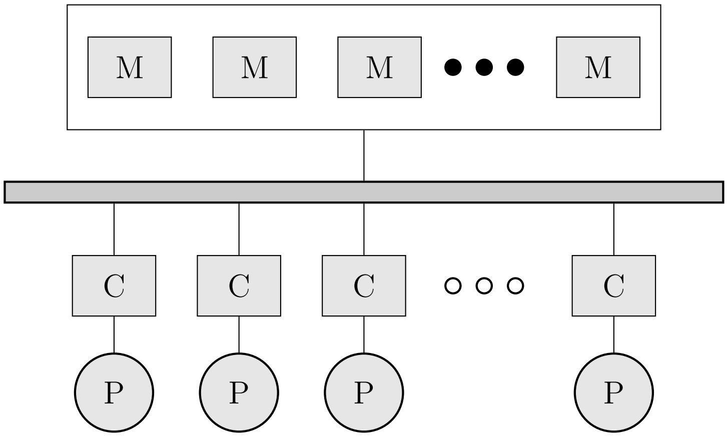


Fig. 2.5: UMA Shared Memory Architecture

UMA architecture is also called symmetric multiprocessor. A UMA architecture is composed of multiple processors with identical characteristics. The processors share the same main memory and IO facilities and are interconnected by some form of bus-based interconnection scheme such that the memory access time is approximately the same for all processors. The **Sun Starfire servers**[28] are an example of UMA architecture.

A subclass of UMA is called Cache Coherence UMA (CC-UMA). Cache coherence means that if one processor updated a shared global memory then all other processors should know about the newly updated values. At the hardware level, cache coherency is accomplished.

Non-Uniform Memory Access (NUMA)[97]:

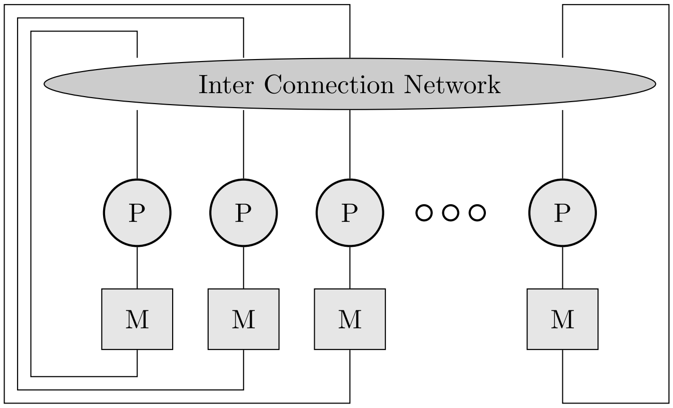


Fig. 2.6: NUMA Shared Memory Architecture

The NUMA architecture is often created by physically linking two or more UMA architectures. In NUMA architectures, the memory access time depends on the different regions of memory. Memory access across the links is slower. **Intel Nehalem**[98] and **Tukwila**[33] CPUs support NUMA. Both CPU families share a common chipset (Intel Quick Path Interconnect (QPI)) for interconnection between CPUs.

A subclass of NUMA systems is Cache Coherent NUMA (CC-NUMA) where cache coherence is maintained among the caches of various processors. The main advantage of the CC-NUMA is that it can deliver efficient performance at higher levels of parallelism than UMA architecture. The **Stanford DASH**[111] is based on the CC-NUMA architecture.

Cache-only memory architecture (COMA)[52]:

COMA is similar to NUMA, in this architecture[1] shared memory is divided into processor-related blocks and the memory is connected through an interconnected network. However, in this system shared memory is made of cache memory. In a COMA system data need to migrate from one processor block to another on a processor request. To migrate the data, the cache directory (D) is used for access from a remote processor. An example of COMA machine is "**The Kendall Square Research's KSR-1**"[111].

Advantages of the shared memory architecture:

- Global address space architecture provides an easy, and user-friendly programming perspective to memory.
- Due to the proximity of memory to CPUs, data sharing between tasks is uniform

Disadvantages of shared memory architecture:

- Primary disadvantage is the lack of scalability among CPUs and memory. Adding more CPUs can geometrically increase traffic on the shared memory-CPU path, and for a cache coherent system, geometrically increase traffic associated with cache/memory management.
- The programmer is responsible for synchronization constructs that ensure correct access to global memory.
- To design and produce large scale shared memory machines, difficulties and cost increase with the increasing number of processors.

2.1.2.2 Message passing multiprocessors / Distributed memory multiprocessors [94]

The distributed memory multiprocessor architecture is different from shared memory architecture in that each unit of the architecture is a complete computer building block including processor, memory and I/O system. A processor can access directly attached memory. A processor can communicate with another processor in the form of I/O operations through message signaling and bus networks. Changes a processor made in its local memory have no effect on the memory of other processors. Hence, the cache coherency does not apply.

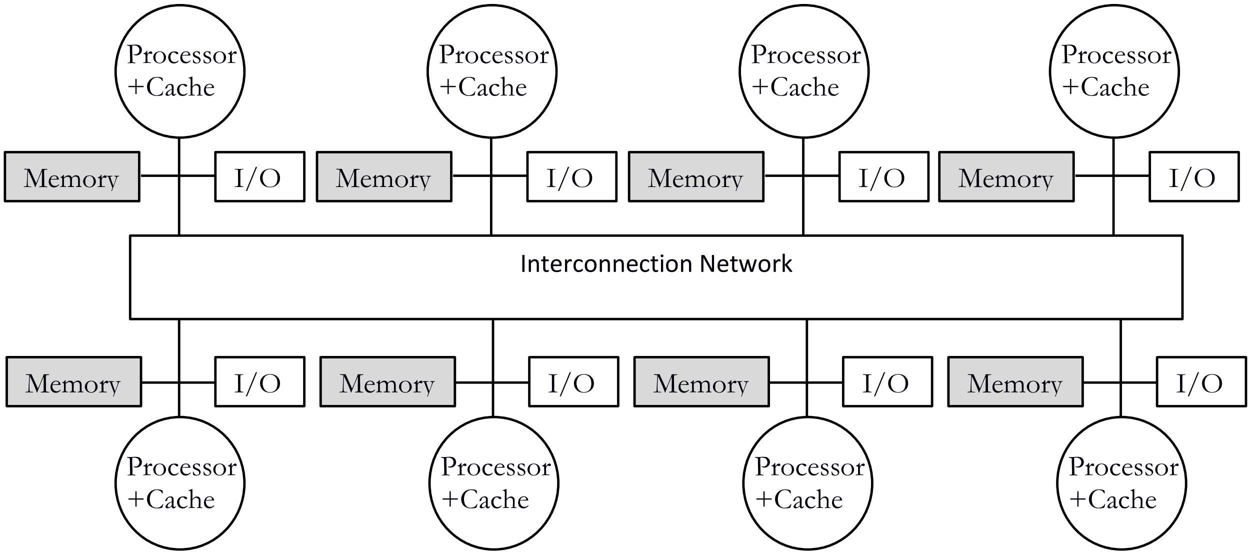


Fig. 2.7: Distributed Processor Computer Architecture

Advantages of the distributed memory architecture:

- Memory is scalable with the number of processors. Memory increases proportionally with increment of the number of processors.

- Each processor can access its own memory without interfering with other processors.
- Cost effective: can use commodity, off-the-shelf processors and networking.

Disadvantages of the distributed memory architecture:

- The programmer is responsible for many of the details associated with data communication among processors.
- Difficult to map data structures based on global memory confined to this memory organization.
- Non-uniform memory access time – data residing on a remote node take longer to access than node local data.

2.1.2.3 Hybrid distributed model[99]

The largest and fastest computer in the world employs both shared and distributed memory architectures. The shared memory components can be accessed by shared memory-CPUs and remote memory locations can be accessed through the interconnection network. Therefore, network communication is required to transfer data from one processor to a remote processor.

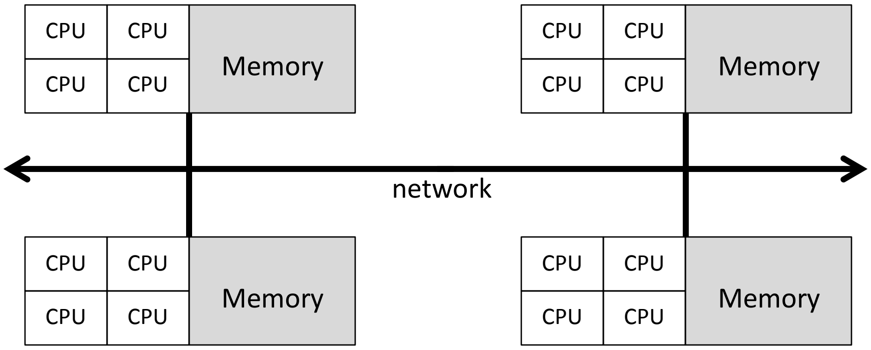


Fig. 2.8: Hybrid Processor Computer Architecture

Advantages and disadvantages of the hybrid memory architecture:

- Whatever is common to both shared and distributed memory architectures.
- The most significant advantage is it increases scalability.
- The most significant disadvantage is it increases programmer time.

2.1.3 According to Interconnection Networks

Parallel computers can be categorized according to the interconnection network between processors and memory.

i Linear network

ii A single shared bus network

iii Multiple shared bus network

iv Crossbar interconnection network

v Star interconnection network

vi Ring interconnection network

vii Tree interconnection network

viii Hypercube interconnection network

ix Mesh and torus interconnection network

x Complete graph interconnection network

xi Switching or dynamic interconnection network

2.1.4 Cache Coherency [37]

In a shared memory multiprocessor architecture with each processor containing a separate cache memory, it is possible to have several copies of any single instruction operand (main memory, each processor cache). All copies for each processor should be updated to their new values with an update of a single processing unit.

Cache coherency protocol is the discipline that ensures that changes in the shared operand values are propagated throughout the system in a timely fashion.

There are three distinct levels of cache coherence:

1. Every write operation appears to occur instantaneously.
2. All processes see exactly the same sequence of changes of values for each separate operand.
3. Different methods may see an operand assuming different sequences of values. (This is considered non-coherent behavior.)

Possible solution to cache coherence problems:

1. No data cache memory
   Eliminating the data cache memory from a shared memory architecture could be a solution of the cache coherence problems, e.g., **Cray MTA -2** uses no data cache [10]. This reduces CPU complexity and eliminates the cache coherence problem. However, no data caching introduces performance problems [13], e.g. memory reference takes 150-170 cycles which is a much higher latency than when using a slower cache.

2. By software: disallow caching of shared variables[72]
   By disallowing cache memory to have a local copy of shared operands could be another way to avoid cache coherence problems, e.g. **Cray T3D [81][61]**. Nowadays generic programming languages are using this concept to access shared variableS exclusively, e.g. **volatile in Java**[**72**].

3. Use a cache coherence protocol

2.1.4.1 **Cache coherence protocols**[14]

In shared memory systems of modern computers two types of cache coherence protocols are used.

1. Bus Snooping Protocol [101]:

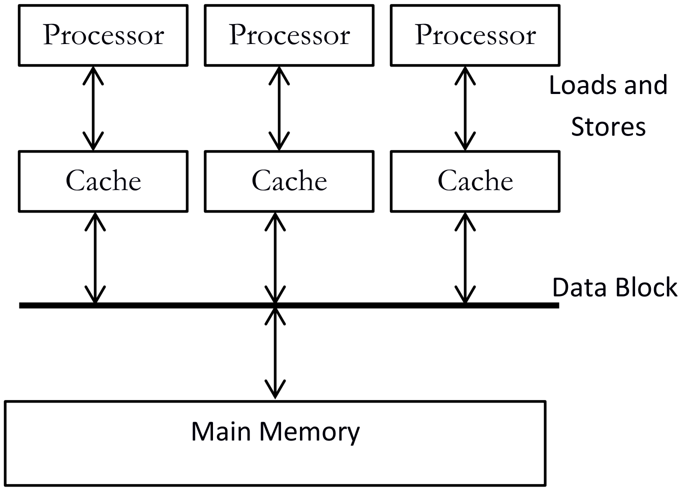


Fig. 2.9: Bus Snooping Protocol architectures

This protocol is used by bus-based and small scale interconnected shared memory systems. It relies on a common channel (or bus) connecting the processors to main memory. This protocol may be further classified into two schemes.

i Write Update Scheme
With this scheme processor immediately broadcasts the performed write operation on the bus; thus as other processors observe the new data being broadcast over the bus, they update their copy of the block data. This scheme creates much traffic on the bus.

ii Write Invalidates Scheme
In this scheme processor perform an invalidate bus transaction before writing the data, in order to ensure that it has the only valid copy of the data block.

Snooping protocols are extensively used in commercial multiprocessor systems such as **Pentium 4** and **PowerPC**.

2. Directory Based Protocol [69]

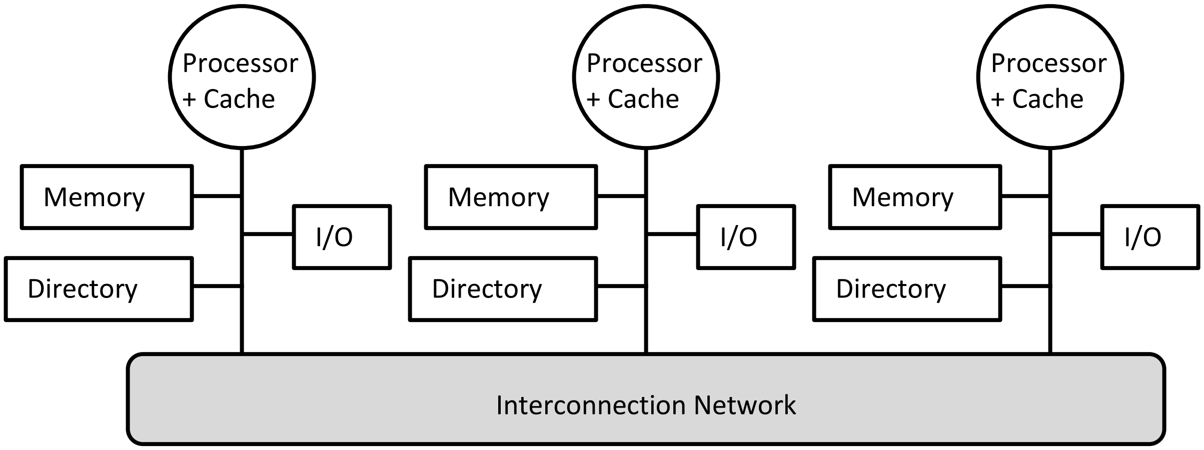


Fig. 2.10: Directory-Based Protocol architectures

In a directory-based system, a common directory is used for shared data to maintain the duplicate data from different caches. The directory works as a filter. To load data into its cache memory from main memory, the processor must ask for the permission. The directory either changes or updates whenever the entry is changed. The **DASH** multiprocessor system uses this protocol.

2.2 Efficiency analysis of multiprocessor architectures [40]

If n is a number of subtasks of a given task, t_s is the execution time of the whole task using a single processor and t_m is the time to execute the whole task on n processors, then $t_m = t_s/n$; the speedup factor of a parallel system is

$$\begin{aligned} S(n) &= Speedup\ Factors \\ &= \frac{t_s}{t_m} \\ &= \frac{t_s}{\frac{t_s}{n}} \\ &= n \end{aligned}$$

The communication overhead factor has been overlooked in the above derivation, which results in the time needed for processor to communicate and synchronize with each other. If tc is the time to communication then $t_m = (t_s/n) + t_c$.

$$\begin{aligned} S(n) &= Speedup\ factor\ with\ communication\ overhead \\ &= \frac{t_s}{t_m} \\ &= \frac{t_s}{\frac{t_s}{n} + t_c} \\ &= \frac{n}{1 + n \times \frac{t_c}{t_s}} \end{aligned}$$

The efficiency (η) measurement of a parallel system is-

$$\eta \quad = \frac{1}{1 + n \times \frac{t_c}{t_s}}$$

Let (f) be a fraction of the given task of a concurrent program that has to execute sequentially. The remaining part $(1 - f)$ is assumed to be divisible into concurrent subtasks executed concurrently. Now, time required to execute the task on n processors is $t_m = ft_s + (1 - f)(t_s/n)$. The speed-up factor is now -

$$\begin{aligned} S(n) &= \frac{t_s}{ft_s + (1-f)(t_s/n) + t_c} \\ &= \frac{nt_s}{nft_s + t_s - ft_s + nt_c} \\ &= \frac{n}{nf + 1 - f + n\frac{t_c}{t_s}} \\ &= \frac{n}{f(n-1) + 1 + n\frac{t_c}{t_s}} \end{aligned}$$

The maximum speed-up factor under such condition is given by -

$$\begin{aligned}\lim_{n\to\infty} S(n) &= \lim_{n\to\infty} \frac{n}{f(n-1)+1+n\dfrac{t_c}{t_s}} \\ &= \frac{1}{f+\dfrac{t_c}{t_s}}\end{aligned}$$

Again, the maximum speed-up factor without communication overhead is -

$$\lim_{n\to\infty} S(n) = \frac{1}{f}$$

Now, the new efficiency (η) is -

$$\begin{aligned}\eta(\text{with no communication overhead}) &= \frac{1}{1+n\times\frac{t_c}{t_s}} \\ \eta(\text{with communication overhead}) &= \frac{n}{f(n-1)+1+n\dfrac{t_c}{t_s}}\end{aligned}$$

2.3 Predictability analysis of multiprocessor architectures

Nowadays, multi-core processors are conquering the world to meet the demand for computational power. As computer processors are becoming more capable, new intelligent applications will emerge, such as, real-time image processing and object recognition, multiple-sensor information fusion, and online spectral analysis for state-based maintenance. Some of these applications will have hard real-time constraints, and these constraints will then have to be predictable on hardware level with respect to time. This requires a Worst Case Execution Time (WCET) analysis for the involved tasks to make the system predictable.

Estimating WCET is very hard. If the estimation is too tight for the system, it becomes possible to violate task deadlines for hard real-time systems, causes being buffer overflow or cache-misses. On the contrary, if estimation is relaxed enough the system efficiency will go down because processor idle time will increase. Moreover, the fundamental problem with the WCET analysis on multiprocessor systems is that the load on other processors is generally unknown. For a task the number of cache misses and their location in time depend on the program control flow path. This means that it is very hard to foresee where there will be bus access collisions, since this will differ from execution to execution. Furthermore, the worst-case control flow path of the task will change depending on the bus load originating from other concurrent tasks.

On the other hand, programming languages contain observable non-determinism properties, which make programs unpredictable with respect to the result of the program execution. Non-determinism is caused when a program execution is not completely determined by its specification; during execution, program can choose the next step by using the run-time scheduler. For instance, observe the following example with parallel thread $T1$ and $T2$, sharing the variable n:

$T1 : x;\ n = 1,000,000;$
$T2 : y;\ n = 10; for i = 1\ to\ n\ do\ r;$
Here, there is a race condition for the global variable n. If the processor executes T1 first, the loop will iterate 10 times, if the processor executes $T2$ first then switches to $T1$ by preemption before executing the loop then the loop will iterate 1,000,000 times. Moreover, if the programs use synchronization through locks, then an imprecise analysis might even falsely detect deadlocks. For instance, the following example with parallel threads $T1$ and $T2$ using a global variable n might create deadlock of a program.

$T1 : x;\ lock\ l;$
$T2 : y;\ lock\ l;\ n = 10; unlock\ l;$

Way to avoid non-determinism in a concurrent language: The easiest way to avoid non-determinism is to design a language that does not support any non-determinism. However, this is unrealistic in practice, because most programming languages must allow some internal optimization to make the program more efficient. So, the problem could be solved by distinguishing non-determinism into two types. The first one is inside non-determinism, which cannot be avoided, the second is observable non-determinism, which might be avoidable.[104]

The table 3.1 on chapter 3 shows a comparison of five programming paradigms with respect to their non-deterministic properties.

2.4 Types of Parallel Programming Models

2.4.1 Instruction Level Parallelism (ILP)

Instruction Level Parallelism (ILP) [56] is a process where several instructions can execute in parallel. In pipelined processes, only one instruction per cycle can execute, but in ILP multiple instructions per cycle can execute. In these processes instructions which are not dependant on previous instruction can execute simultaneously. For example, consider the following program optimization by ILP. In this example, ILP process required 3 instruction cycles instead of 5.

```
Cycle 1: LOAD r1,(r2);
Cycle 2: ADD r5,r6,r7;
Cycle 3: SUB r4,r1,r4;
Cycle 4: MUL r8,r9,r10;
Cycle 5: STORE (r11),r4;
```

```
Cycle 1: LOAD r1,(r2);    ADD r5,r6,r7;
Cycle 2: SUB r4,r1,r4;    MUL r8,r9,r10;
Cycle 3: STORE (r11),r4;
```

Fig. 2.11: Instruction level parallelism example

The available parallelism is limited by any sequence of instructions [106]. ILP is limited by following hazards/dependencies.

- Data Dependencies: RAW (Read After Write), WAR (Write After Read), WAW (Write After Write);

- Control Dependencies: If a program has a condition branch then until execution of this condition all later instructions must wait.

- Memory Dependencies: $100(r1)$ and $35(r3)$ may indicate the same memory location. Dependencies that flow through memory locations are difficult to detect.

2.4.2 Thread Level / Task Level Parallelism (TLP)

An alternative model of parallelism is Thread/Task Level Parallelism (TLP) [60] where multiple flows of executions run on a single processor. Sometimes processes wait for their resources (e.g. printer acknowledgement signal), and the processor has some idle time. In this idle time, the processor can start another independent process. To do this, a big process is divided into a few independent small chunks named threads/tasks. Each thread contains its own instruction and data. In a multiprocessor system, TLP is achieved by running different threads/tasks on a different processor with the same or different data. When a thread's resource is not available then the thread goes into a blocked state and waits until the resource is available. Once the resource becomes available afterwards, the thread goes into a waiting queue and finally executes. The thread may execute the same or different code. Different threads can communicate with each other through message passing or using shared global memory. With this method processes need an extra mechanism to synchronize with each other.

2.4.3 Data Level Parallelism (DLP)

In a multiprocessor system Data Level Parallelism (DLP) [70] is a method where data are distributed into different parallel computing nodes. For example in the image processing engine, different processors execute the same line of instructions with a different set of data, vector processing is also an example of DLP.

2.5 Tests and Analysis

To test and analyze the performance and efficiency of different parallelization techniques, it is necessary to have different hardware resources. The following section discusses the resulta of different parallelization tests taken from the currently available published research.

Case 1 : Efficient Parallelization using Combined Loop and Data Transformations [77].

Usually some parts of the program consist of a loop or an array access which requires particular transformation while other parts of that program require completely different transformations. For these various transformation techniques large cache memory is required, or else, cache misses occur. To solve this problem, parallelizing compilers need to minimize the degree of internal synchronization and inter-processor communication and also need to maximize temporal and spatial locality within a program. To improve spatial locality, Cierniak and Li [30] and Kandemir et al. [58],[57] have combined non-singular loop transformations with data transformations.

The research has been done to develop a compiler heuristic MARS [21] to minimize parallelization overheads, which explore how to resolve the requirement of a conflicting loop by data transformation and vice-versa[77]. This is achieved by treating data and loop transformation in a unified manner. They used seven theorems for their compiler to develop a communication and synchronization mechanism. Finally, they combined loop and data transformation for optimization.

For testing they developed a compiler to test alternative techniques in a number of experiments on an SGi Origin 2000. Three SPECfp92 kernels: vpetst, btrtst, chotst were selected, and four different approaches tested: combined, data, loop, PFA(a commercial loop-oriented parallelizing compiler). Their performance was plotted into a graph against the number of processors used[77]. With the VPETST kernel, the combined method shows a vast performance improvement on a larger number of processors (e.g. 32). The results were 50% faster over PFA and 20% over 30% faster over loop method, and with BTRTST kernel the combined approach is a factor of 2 faster than PFA for a large number of processors. Finally, the CHOTST kernel test shows that combined, data and loop give the same result whereas PFA process is 80% slower than the other method[77].

Case 2 : A detailed analysis of contemporary ARM and x86 Architectures[20]:

In a processor architecture design system the Instruction Set Architecture (ISA) plays a vital role in the performance, efficiency and consumption power of that processor. The development of RISC and CISC ISA began in the 1980 when the design complexity and cheap area were the primary concerns. Today, the low power ARM ISA enters into the high-performance market for desktop server PCs, while the traditional high-performance Intel x86

ISA is trying to enter into the low-power mobile device market. Thus, the question arises whether the energy efficiency is dominating the market over ISA or vice-versa.

Clark and Bhandarkar compared the VAX and MIPS ISA by comparing the Digital VAX 8700 to the M/2000 implementations [19] and summed up with "RISC as exemplified by MIPS provided a significant processor performance advantage." In 1995 another research by Bhandarkar compared the Alpha 21164 to Intel Pentium-Pro [19], this study focused on performance and concluded "the Pentium Pro processor achieves 80% to 90% of the performance of the Alpha 21164".

A detailed analysis has been done on Intel Atom and Sandybridge i7 and ARM cortex-A8 and cortex-A9 microprocessors [20]. The test used four different workloads spanning mobile, desktop INT and FP memory footprints and server computing. This test demonstrates the role of ISA on modern processors in performance and efficiency. For this test, they used Linux 2.6 LTS kernel with some minor board-specific patches and a gcc4.4 cross compiler [20]. Machine specific tuning and THUMB instructions were disabled and x86 32-bit processors were used. For the application in mobile client WebKit regression tests [50], desktop SPECCPU2006 (www.spec.org) and server Cloud Suite workload[43] were used [20].

The performance analysis from this test found a large performance gap across the four platforms, where Intel was showing high performance. Compared to A9, i7 perform 5 times to 102 times faster and compared to A8, Atom performs 2 times to 997 times faster. In cycle count comparison i7 took 2.5 times less cycle time than A9 and Atom took 1.5 times less time than the A8 processor[20]. So, finally, in Power and Energy test i7 consumed 17 to 21 times more power than the A9 processor and Atom consumed about 3 times more power than the A8 processor [20].

The result was as expected that Intel processors provide high performance but consume much power, on the other hand, ARM processors provide energy efficiency but perform comparatively slowly.

3. Incorrect Behaviors

In order to build compatible software for modern high performance multicore computers, programmers have to design complicated parallel software architectures to achieve performance improvement and hardware efficiency.

Several level optimizations are required for creating mainstream software applications that utilize the full power of parallel hardware. Mainstream generic programming languages like C++, Java, Ada [22]allow the non-determinism in a program to achieve maximum efficiency of the processors. The execution is called non-deterministic if, during its execution a program has to decide the next line of instruction to execute. This non-deterministic behavior may introduce bugs into the program that will cause its incorrect functionality.

3.1 Common incorrect behaviors in a concurrent program

There are several kinds of bug that may appear in a program [102] -

- Race condition
- Deadlock
- Livelock
- Starvation

Race Condition: A race condition affects program's correctness by changing the timing and ordering of program events [76]. More generally, some external efforts need to change the timing and ordering of the program, to produce a race condition. General examples are OS signals, hardware interrupts, context switching and memory operations on a multiprocessor system.

A **Data race** occurs when multiple threads want to access same shared memory location during the execution, and at least one of them tries to modify that location. It may introduce unpredictability and unexpected results that are often hard to detect.

Race condition usually depends on the execution order of a program that may vary on different hardware architectures. So, a program with a race condition may perform normally on particular hardware. Race conditions can only be avoided but not eliminated from the program in programming languages that allow the non-determinism.

Deadlock: If two or more threads wait on each other, forming a cycle that prevents all of them from making any further progress, this is called Deadlock[113]. It could be created by the programmer while trying to avoid race conditions. For example, incorrect use of synchronization condition primitives such as locks may introduce multiple threads waiting for each other. Deadlock is also possible without synchronization mechanisms; circular wait in a program can result in a deadlock.

Starvation: Starvation is a situation in a multi-threaded application, where single or multiple threads are delayed indefinitely or blocked permanently[3]. For example, a thread with a low priority is waiting for being scheduled, but high priority threads are executed although this lower priority thread is neither blocked nor waiting for any resources. Typically, scheduling rules and policies are the reasons for starvation in a multi-threaded program.

Livelock: Livelock is a situation where multiple threads depend on each other and changes of their own states result in a circular way in response to changes in the other threads. The result is none of them will complete[36].

3.2 Factors that are commonly responsible for incorrect behaviors

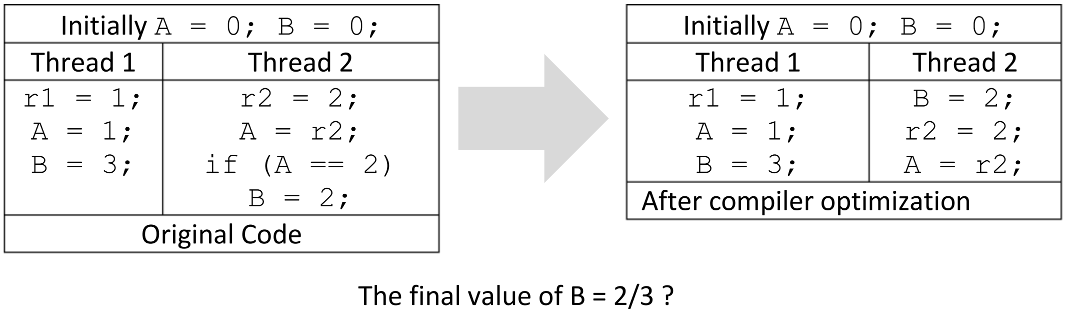
3.2.1 Race condition

The first and main source of race conditions in a program is `observable non-deterministic`[104] behavior of mainstream programming languages e.g. C++, Java. On the other hand, programming languages like Oz or Alice do not support any race conditions, because they are free of observable non-determinism.

Among the programming languages that support observable non-determinism, `shared memory / variables` are responsible for producing data races into programs. However, shared memory plays a vital role for communication among processors in a shared memory computer architecture. It is also possible to use message passing communication mechanisms in systems with shared memory architectures, but, performance and efficiency of those systems will fall dramatically. So, to achieve performance and efficiency at an optimum level, programmers have to use shared variables while maintaining specific rules and regulations.

To introduce a race condition into a program, `concurrent access` is required in a multiple threaded shared memory program. Multi-threaded sequential consistent program cannot produce race condition.

`Compiler optimizations` may change order of program events and may produce an unexpected result[9]. For example, the following optimization may introduce a race condition into a program.



Initially `A = 0; B = 0;`	
Thread 1	Thread 2

`r1 = 1;`
`A = 1;`
`B = 3;`
	`r2 = 2;`
	`A = r2;`
	`if (A == 2)`
	`B = 2;`

Original Code	

Initially `A = 0; B = 0;`	
Thread 1	Thread 2

`r1 = 1;`
`A = 1;`
`B = 3;`
	`B = 2;`
	`r2 = 2;`
	`A = r2;`

After compiler optimization	

The final value of B = 2/3 ?

Fig. 3.1: Compiler Optimization may introduce unexpected result

In some programming languages, `adjacent data field` may introduce data races. For example, the following program may create data races in the C++ programming language. According to the C++11 standard, adjacent bit fields are one "Object"[23].

a global **s** of type `struct {char m:9; char n:7;}`

```
//Thread 1
lock_guard<mutex>lock(mMutex)
{
 s.m = 1;
}
```

```
//Thread 2
lock_guard<mutex>lock(nMutex)
{
 s.n = 1;
}
```

Finally, `execution order` may also introduce data races into a program. As an example, the following program shows "Dekker's" example (all variables are initially zero):

```
Thread 1
x = 1;
r1 = y;
```

```
Thread 2
y = 1;
r2 = x;
```

Could be executed as:

x = 1; y = 1; r2 = x; r1 = y;
or,
x = 1; **y = 1; r1 = y;** r2 = x;

Whereas, second execution sequence has a **data race** with y variable e.g.- r1 variable trying to read y while y is updating its own values.

3.2.2 Deadlock

Deadlock can arise in any concurrent program, where processes can preempt each other and generate a waiting cycle chain with each program/thread. More generally, deadlock can be introduced in any system that satisfies Coffman's four conditions [31].

Mutual exclusion: Threads claim accesses of resources exclusively. (e.g. Threads grab a lock)

Hold-and-wait: Threads that already hold a resource claim new resources.

No preemption: Thread holding a resource cannot be forcibly removed, only a process/thread holding a resource can release it.

Circular wait: Multiple processes create a circular waiting chain where all processes request a resource that the next process in the chain holds.

Concurrent programs may satisfy these requirements, as they use synchronization mechanisms and may deadlock. Deadlocks can be avoided by rearranging the program so that one of the Coffman conditions does not hold.

3.2.3 Livelock

A livelock is almost the same as a deadlock, except that the involved process states constantly change one after another, but no one can progress[16].

3.2.4 Starvation

Typically following factors cause process/thread starvation [59]

1. Processes hand over resources to other processes without any control. If resource allocations for processes are decided locally without considering the overall resource requirements for the system, irregularities can occur and result in some processes suffering starvation.
2. Higher priority processes consume all CPU time from lower priority processes. For example, the programmer sets process priority so that longer execution time processes will have higher priority. So, the processes with lower execution time may never execute and suffer starvation.

3. Randomly selected processes using resources may also cause starvation. If a waiting queue is not maintained for processes waiting service, some processes can never use required resources and may suffer starvation.
4. If the number of processes is much higher than the number of resources, some process execution periods may be exceeded before the process can use required resources.

Starvation can occur at any organized scheduling level though it more often takes place in the automatic process allocation than in the higher-level manual process allocation parts.

3.3 Way to avoid/eliminate/prevent incorrect behaviors

3.3.1 Race Condition

3.3.1.1 Avoid observable non-determinism

The easiest way to eliminate a race condition from a program is by using a programming language that doesn't have non-determinism. However, nearly all main stream programming languages support non-determinism[104]. So, the programmer has to make a clear distinction between non-determinism *inside*the system, which cannot be avoided, and *observable* non-determinism, which may be avoidable. This can be done with two following steps:

First, separate and limit observable non-deterministic events from a program. The remaining part of the program should have no observable non-determinism.

Second, define the programming languages so that it would be possible to write concurrent programs without observable non-determinism.

Concurrent Paradigm	Example Languages	Race Possible?	Input can be nondeterministic?
Declaration concurrency	OZ[32], Alice[103]	No	No
Constraint programming	Gecode[89], Numerica[82]	No	No
Functional respective programming	FrTime[51], Yampa[53]	No	Yes
Discrete synchronous programming	Esterel[46], Lustre[75]	No	Yes
Message-passing concurrency	Erlang[15], E[74]	Yes	Yes

Tab. 3.1: Deterministic concurrent state in different programming paradigms [104]

Java, C++ and C# use shared state concurrency[27] and Erlang, E use message passing concurrency [15] [74]; both paradigms have observable non-determinism. Fortunately, there are at least four useful programming paradigms in existence that are concurrent but do not allow observable non-determinism[104]. Table 3.1 lists these paradigms together with message passing concurrency.

3.3.1.2 Use of Synchronization Mechanisms

Mutex Locks:

Mutex lock: Mutex lock is the basic form of synchronization between processes/threads[72][23], where Mutex stands for Mutual Exclusion. It guarantees that only one process/threads can access into Mutex block. If a code section is blocked by Mutex then it makes sure that only one thread can lock this code section. Other thread can lock this block only after the first thread unlocks this block.

```
//The mutex has been previously constructed
lock_the_mutex();
//This code will be executed only by one thread
//at a time.
unlock_the_mutex();
```

Condition Variable:

Condition variables allow threads to wait until the occurrence of a particular condition[38]. They cannot be shared across processes; they are user mode objects. They allow threads to release the lock automatically and enter the sleeping state. A condition variable can do two following things:

First, wait: the thread has to wait until some other thread notifies that it can continue because of fulfillment of the condition.

Second, notify: the thread sends a signal to a particular thread or to all threads to tell them that the condition that provoked their wait is fulfilled.

```
CRITICAL_SECTION CritSection;
CONDITION_VARIABLE ConditionVar;

void condition variable()
{
    EnterCriticalSection(&CritSection);

    // Wait until the predicate is TRUE

    while( TestPredicate() == FALSE )
    {
        SleepConditionVariableCS(&ConditionVar, &CritSection, INFINITE);
    }

    // The data can be changed safely because we own the critical
    // section and the predicate is TRUE

    ChangeSharedData();
```

```

  LeaveCriticalSection(&CritSection);

  // If necessary, signal the condition variable by calling
  // WakeConditionVariable or WakeAllConditionVariable so other
  // threads can wake
}
```

Semaphores:
Semaphore is a synchronization mechanism that uses an integer type variable or abstract data type that is used to control the access of shared resources by multiple processes[62]. It uses the following two types of signals for the process synchronization.

Wait: Processes will test the variable values and wait until the value become greater than zero. Otherwise, process will decrement the semaphore variable. If S is the variable for semaphore then `wait` can be defined as -

```
wait(S){
  while(S <= 0 );\\wait until the value is > 0;
  s--;
}
```

Signal: Increment the value of the semaphore and awake a process if any of them is blocked. If S is the variable for semaphore then `Signal` can be defined as -

```
signal(S){
  S++;
}
```

If the initial value of the semaphore variable is 1, a `wait` operation is similar to Mutex locking and `Signal` value is similar to Mutex unlocking. To synchronize multiple threads with a semaphore -

```
\\wait for access into critical section
wait(S);

\\enter critical section
GlobalVaribal = GlobalVariable + 2;

\\exit from critical section
signal(S);
```

Low level primitives:

Many popular generic programming languages include low-level primitives for synchronizing programs/threads at low level. However, different languages were implemented using different names for their own low level primitives. For example, C++ uses atomic type variables [18] whereas Java uses volatile and atomic as volatile with some extra features[84], Ada uses volatile and atomic pairs via pragmas[1]. Low level primitive read and write operations are not stored into the cash register, but are directly transferred to the memory location. Moreover, this prevents compiler optimizer to reorder memory access and automatically includes Read/Write operations as memory acquire/release respectively. Furthermore, its write operation happens-before all following reads of the same variable. Compared to lock/unlock or monitor exit/monitor enters operations, writes work as unlock or monitor exit and reads work as lock or monitor enter. In addition, it also guarantees the visibility and ordering of operations and these types of operations are cheaper than other synchronization operations. The main disadvantage of low level primitives is that even if there is only a limited set of operations using this variable in a program, it is very complicated and requires sophisticated analysis to verify, although it looks very easy to use. An example of low level primitives in Java could be as follows:

```java
import java.util.concurrent.atomic.AtomicInteger;

class AtomicVariableExample {
    private AtomicInteger num = new AtomicInteger(0);

    public void increment() {
        num.incrementAndGet();
    }

    public void decrement() {
        num.decrementAndGet();
    }

    public int value() {
        return num.get();
    }

}
```

3.3.1.3 Sequential Consistency for Data Race Free (SC-DRF)

If a program does not allow data races and synchronizes with only following options -

- Sequentially consistent atomic, and
- Uses acquire and release semantics for `Lock` and `Unlock` operations respectively.

- then the program follows sequential consistency rules, i.e. it behaves as though it had been executed by sequentially interleaving its actions, and letting each load instruction see the last preceding value stored to the same location in this interleaving.

Software memory model has converged on SC-DRF. Java required SC-DRF since 2005 [72] and in C/C++ since 2011[23] it is the default type memory ordering.

3.3.1.4 Use Race detection algorithm

Several algorithms have been developed to detect race conditions in a program. Further mentioned some of the well-known algorithms that are used to detect race conditions in programs.

- FastTrack:Efficient and Precise Dynamic Race Detection [44]
- HARD: Hardware Assisted Lockset Based Race Detection[112]
- Eraser: A Dynamic Data Race Detector for Multithreaded Programs[88]
- Hybrid Dynamic Data Race Detection[78]

3.3.2 Deadlock

3.3.2.1 Deadlock Prevention

To prevent deadlocks, it is necessary to make it logically impossible for one of the four Coffman deadlock situations to hold in a concurrent program[105]. To achieve this, several methods have been already developed:

Elimination of “Mutual Exclusion” condition: To eliminate mutual exclusion from a program means that no process will have exclusive access to a resource. On the other hand, avoiding data races in program tasks means exclusive use of resources is needed. Non-blocking synchronization algorithms [48] can be used to avoid mutual exclusion in a program. Low level primitives are also useful to avoid mutual exclusion in a concurrent program.

Elimination of “Hold-and-wait” condition: There are two techniques that can be used to eliminate hold-and-wait conditions[105]. The first method is achieved by acquiring all locks at once atomically. In practice, this could be achieved as follows:

```
lock(holdandwait);
lock(R1);
lock(R2);
lock(R3);
  ... ...
unlock(holdandwait);
```

The first lock guarantees that only one process can acquire all resources and no other process can acquire these resources until the first process releases it. However, this method is problematic for a number of reasons, i.e. this method decreases the concurrency as all shared resources are acquired early on (at once) instead of exactly when they are truly needed. Second method is by disallowing processes to request a resource whenever it was allocated previously. In this technique the system has to grant resources on one or none basis. If the full set of resources that is needed by a process cannot be acquired then it has to wait until the complete set is available. So while waiting, processes may not hold any resources to avoid a deadlock situation. However, this approach also can lead to waste of resources and processes may have to wait for a long time.

Elimination of "No-preemption" condition: The non-preemption condition can be eliminated by sending the process that is waiting for a resource that cannot be allocated immediately to a waiting state to hand over all of its currently held resources, so that other processes may use those resources. However, eliminating no-preemption situation from a concurrent program may introduce circular wait for processes and result in a deadlock.

Elimination of "Circular Wait" condition: by imposing a total ordering on all resource types circular wait condition can be eliminated. All processes request resources in a certain order. This method imposes a totally ordered use of all resource types with this rule; the resource allocation graph can never have a cycle. Lamport's happens before relation [66] could be used for a total order of resource access.

3.3.2.2 Avoiding Deadlock

In some scenarios, deadlock avoidance is preferable to preventing a deadlock. This technique to the deadlock problem predicts deadlock before it occurs. A process uses an algorithm to predict the possibility of a deadlock and to act accordingly. This method is not similar to the deadlock prevention that guarantees that deadlock cannot occur by denying one of the four necessary conditions for deadlocks. Moreover, with this approach if necessary conditions for a deadlock are in place, deadlock avoidance would be still possible. One famous example algorithm for this method is Dijkstra's Banker's algorithm[35]. In addition, resource allocation graph algorithm is also useful where only one instance per resource is present.

3.3.2.3 Deadlock Detection

The final strategy is deadlock detection. In this approach, the system may enter into a deadlock state. In that state, the system needs an algorithm that periodically examines if a deadlock has occurred in the system and offers a procedure to recover from the deadlock [92]. The deadlock detection algorithms maintain a wait-for graph and periodically invoke an algorithm that searches for cycles in the graph. A deadlock is detected in that system if any cycle is completed. If a deadlock is detected, the system needs to recover from that state. Recovery can be done using two approaches. One is terminate all processes in the cycle; it works fast but may lose process work. Second is terminate one process from the cycle and run deadlock detection algorithm again; it's better in terms of process work, but with an extra work to resolve a deadlock.

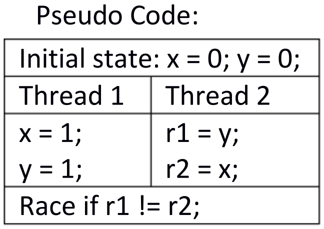
3.3.3 Starvation

Remedies for starvation are applied by ensuring the conditions for starvation cannot happen[3]. Here is a selection.

1. There should be an independent manager for each resource, which will manage all allocation for its resources; this will guarantee that processes do not just pass resources around between themselves without making them available for general allocation.
2. Fair scheduling method for all processes may avoid starvation by changing process priority at execution level.
3. There should be a waiting queue for processes that need to access the resources. Random selection technique, uncontrolled competition, should be avoided for resource allocation.
4. Provide fair number of resources compared with the number of processes, though this solution can cost money. However, it is better than having process starvation which may cause some serious issues in the system.

3.4 Tests and Analysis

In this thesis data race detection test has been done on two different machines with two different versions of Linux kernel and one Windows version in two different programming languages. In this test, a simple multi-threaded program with global and local variable runs 1,000,000 times using different hardware, operating system and programming languages. The pseudo code of this program is as follows:



Pseudo Code:

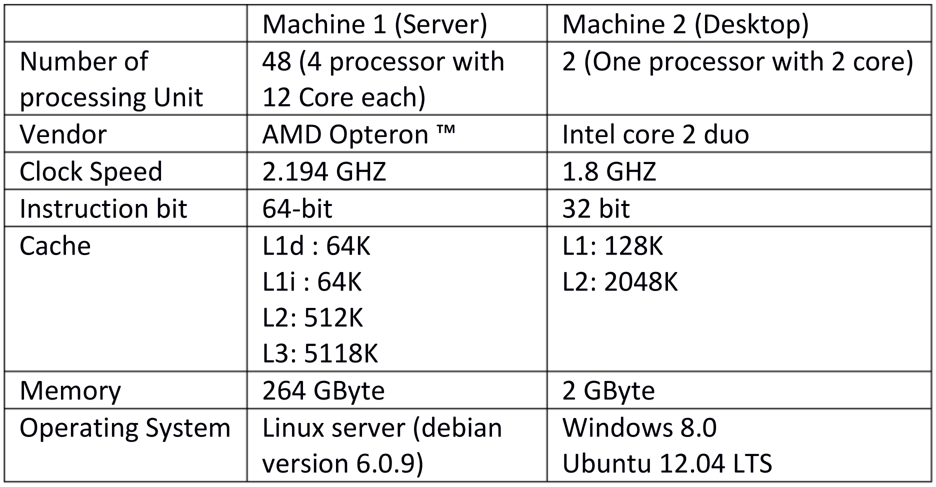
Initial state: x = 0; y = 0;	
Thread 1	Thread 2

x = 1;
y = 1;
	r1 = y;
	r2 = x;

Race if r1 != r2;	

In the above program, one thread updates global variables and another thread reads global variables and copies them into local variables. In a sequential consistent execution, either thread 1 or thread 2 will execute first and the values for local variables r1, r2 will be 0 or 1. But, two different values of r1 and r2 will cause a data race.

Implementation Platforms: Hardware Platform: In this thesis two different hardware platforms: one server and one desktop PC were used. The detailed information about both hardware systems is shown in table 3.2.



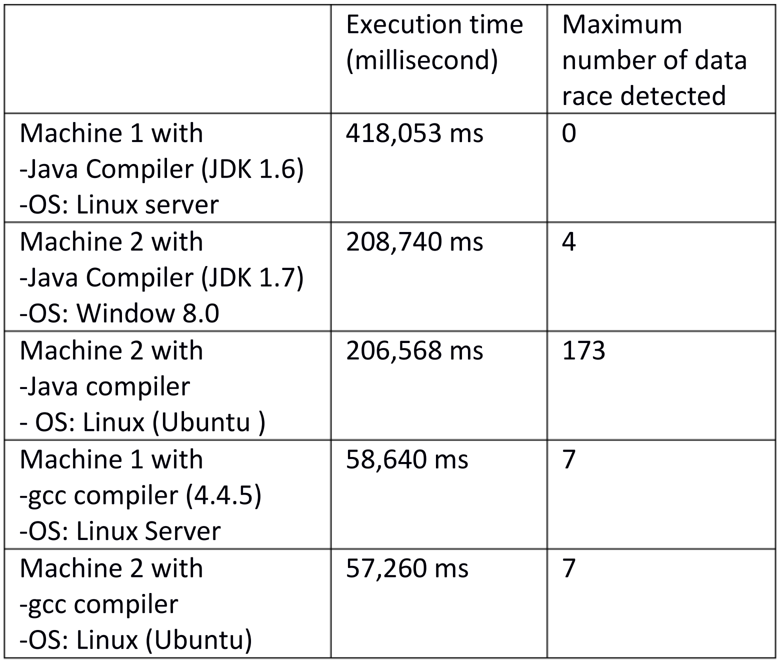
	Machine 1 (Server)	Machine 2 (Desktop)
Number of processing Unit	48 (4 processor with 12 Core each)	2 (One processor with 2 core)
Vendor	AMD Opteron ™	Intel core 2 duo
Clock Speed	2.194 GHZ	1.8 GHZ
Instruction bit	64-bit	32 bit
Cache	L1d : 64K L1i : 64K L2: 512K L3: 5118K	L1: 128K L2: 2048K
Memory	264 GByte	2 GByte
Operating System	Linux server (debian version 6.0.9)	Windows 8.0 Ubuntu 12.04 LTS

Tab. 3.2: Test Machine Specification

Operating System: Three different operating systems were used on two different types of hardware systems. Linux Server (Debian version 6.0.9) was used on the server machine and Desktop PC Windows 8.0 and Ubuntu 12.04 operating system were used on the desktop machine.

Compiler: On both machines, three different operating systems, eclipse IDE for Java and C++ data race check were used. On the server machine, the Java version JDK 1.6 and gcc version 4.4.5 were used. So on the desktop machine JDK 1.7 for the Java compiler in Windows, JDK 1.6 in Linux and gcc 4.4.7 for C++ in Linux were used. Compiler optimization level was manually turned off for this test process.

Test process: The above program ran on both machines on three different operating systems. For a single tested data set, the same program was run in the same environment several times then the maximum number of data races was counted. Each time the above threads ran 1,000,0000 times and the number of data race was counted in the program. The resulting data is shown in table 3.3.



	Execution time (millisecond)	Maximum number of data race detected
Machine 1 with -Java Compiler (JDK 1.6) -OS: Linux server	418,053 ms	0
Machine 2 with -Java Compiler (JDK 1.7) -OS: Window 8.0	208,740 ms	4
Machine 2 with -Java compiler - OS: Linux (Ubuntu)	206,568 ms	173
Machine 1 with -gcc compiler (4.4.5) -OS: Linux Server	58,640 ms	7
Machine 2 with -gcc compiler -OS: Linux (Ubuntu)	57,260 ms	7

Tab. 3.3: Number of data races detected on different machines

Analysis: Data races are very hard to detect. Their number depends on compiler and hardware optimization systems. In this test a huge change in result was observed on machine-two; for the Java compiler the number was four in Windows whereas 173 races were detected on the Linux (Ubuntu) operating system, although the hardware and the code were the same in both cases.

4. Interaction Between Software and Hardware

A computer is a state machine. If a processor is forced into its initial condition, then its next state is completely predictable. We can use this simple fact to see what happens when a computer system boots.

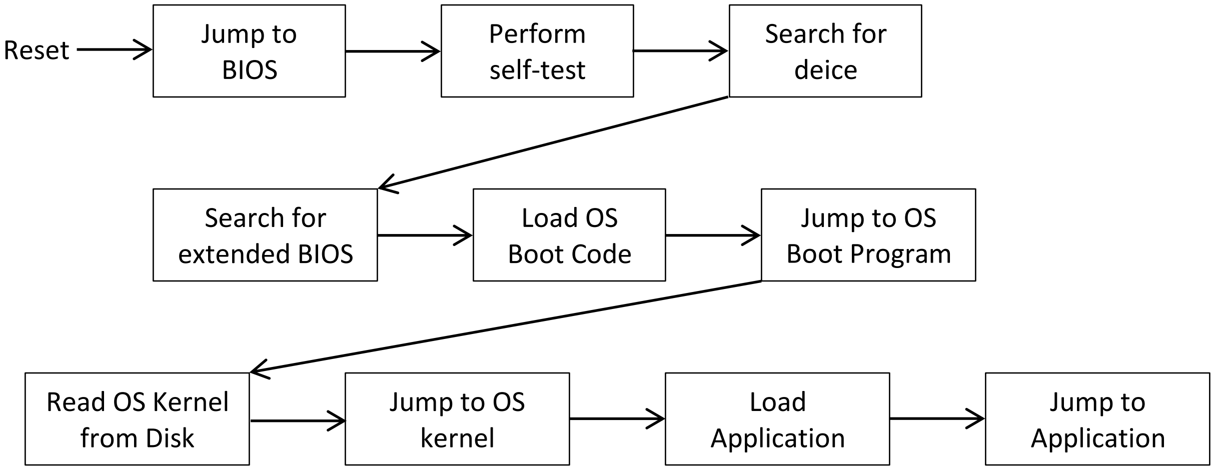


Fig. 4.1: A state diagram of a computer start-up process

Over the past few decades processors developed rapidly showing increases in performance, speed and power while coming from different vendors (e.i. Intel, ARM, POWER PC, etc.) using different architectures but none of them is best for all tasks. For example, Intel chips are good in performance but have had highest power consumption and price; ARM chips have lower power consumption and are significantly less expensive, but suffer from the performance problem. Moreover, each vendor has its own multicore processing architecture. As a result, software portability became limited to a particular vendor's design. So, software developers had to develop several versions of software with the same functionality.

The interface between a program and any software or hardware (for example, the virtual machine, operating system, the compiler, or any dynamic optimizer) is defined by a memory model. In a concurrent hardware or programming system, it is not possible to meaningfully write a program (written either in a machine language, assembly, byte code, or a high-level language) or any part of the program without an explicit memory model.

4.1 Memory Models

The memory consistency model or memory model defines the set of rules of a system or program stating what it is allowed to do. It is the heart of the concurrency semantics of a shared memory system or program, thereby defining the basic semantics of shared variables[5], for example, the return value of a memory read operation, thread synchronization mechanism, concurrent write operation rules for two adjacent data fields, eliminating out-of-thin air values.

It is difficult to write a program with a complex memory model. An extremely complex one may limit the compiler and hardware optimization, critically reducing performance, portability and maintainability of programs. Therefore, the memory model has long lasting effects. The hardware architecture with a strong memory model cannot change later without breaking binary compatibility for a weaker model, and a compiler with a weaker memory model may require rewritten source code. Finally, memory-model-related decisions must be considered for the rest of the system. If the memory system designer gives a weaker memory model, processor vendors cannot guarantee a strong hardware model, and a strong hardware model cannot provide full performance with a program that compiled using a weak memory model programming language. However, the importance of memory models has often been emphasized. The reason behind that could be the surprising complexity to specifying a model, which balances all desirable properties of portability, performance and programmability.

Another challenge had to be faced by hardware architects in that they confronted the limitation of memory models at the programming language level. Programmer expectations from hardware were not clear. Despite that, hardware researchers proposed different approaches to minimize this gap[7]. Since 2000, experts have started to specify cleanly memory models at the programming language level. As a result, in 2004/05 Java and in 2011 C++ provided a well-defined software memory model, for C and other languages efforts are now under way.

Nowadays, most hardware vendors and mainstream programming languages have published (or plan to publish) compatible memory model specifications. Despite a dramatic improvement achieved by this convergence, some basic shortcomings were exposed in parallel languages and their compatibility with hardware. After decades of research, it is still undefined what value a read operation can return while using any modern safety/synchronization mechanism.

4.1.1 Hardware Memory Models

The relaxed memory model is weaker than the sequential consistency. Currently, most hardware supports the relaxed memory model. It takes a performance-centric or implementation view, where model specification derives from the hardware optimization level.[8][6] Typical strong memory model guarantees that shared memory write operation values will become visible to all other thread read operations in the program order. Such models additionally insert fence instructions to confirm the total program order. Such a program ordering and fencing style of specification is easy to use. However, its excessive use results in an inefficient system.

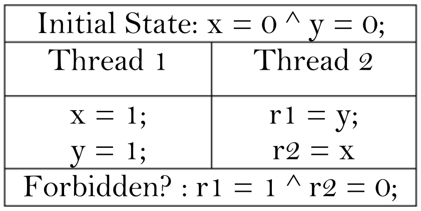
4.1.1.1 IBM Power and ARM processors

IBM Power and ARM multiprocessor products come with a low-power consumption architecture and an inexpensive RISC instruction set[73]. Nowadays many smart phones and tablet PCs have ARM processors inside them to achieve their low power consumption. Finally, ARM chips started to improve their performance. As a result, it is observable how much faster tablet computers and smart phones have become over the past few years. On the other hand, PowerPC was designed to use as desktop CPU but it has become popular in embedded systems. Many game consoles, embedded architecture, router and network switches use PowerPC chips.

PowerPC and ARM multiprocessors both have the relaxed memory model to achieve the maximum hardware optimization. However, in a multiprocessor architecture excessive optimizations required out-of-order execution which resulted in incorrect behavior (e.g. deadlock, data race) of a program. To avoid such incorrect/undefined behavior and to regain the program execution order, a strong memory barrier or synchronization mechanism is presented by both ARM and PowerPC. PowerPC uses *sync* instructions[11] and ARM uses *dmb* [29] instructions for a strong memory barrier. PowerPC introduced another *lightweight sync* instruction called *lwsync* that is potentially faster than the sync instruction.

ARM7 has Princeton memory architecture (one bus for both data and instructions, and never both at the same time) and the ARM9 processor has Harvard architecture (separate buses, each for data and instructions). To test the ARM and IBM Power processors, a simple classic Message Passing (MP) example was written with two threads and two global variables (x and y). This is a simple low-level concurrency programming idiom, the desired behavior is that if the Thread 2 reads y = 1 then read x != 0, In other words, the situation where of r1 = 1 and r2 = 0 should be forbidden.

MP Pseudo code



Initial State: x = 0 ^ y = 0;	
Thread 1	Thread 2

x = 1;
y = 1;
	r1 = y;
	r2 = x

Forbidden? : r1 = 1 ^ r2 = 0;	

In a sequentially consistent model, there are only the following possible ways to execute and none of them results in r1 = 1 and r2 = 0; to get this result the processor must have to execute instructions out-of-order.

Order of instruction	Registers final value
x=1; y=1; r1=y; r2=x	r1=1 ∧ r2=1
x=1; r1=y; y=1; r2=x	r1=0 ∧ r2=1
x=1; r1=y; r2=x; y=1	r1=0 ∧ r2=1
r1=y; r2=x; x=1; y=1	r1=0 ∧ r2=0
r1=y; x=1; r2=x; y=1	r1=0 ∧ r2=1
r1=y; x=1; y=1; r2=x	r1=0 ∧ r2=1

The only possible order of instructions to get r1 = 1 and r2 = 0 is x=1; r1=y; r2=x; y=1; To see whether it is observable or not, a test using the Litmus tool (a test harness [12] on particular processors was implemented [73]. The following table gives some sample experimental data, this test has been implemented for different versions of the IBM Power and ARM processors using the test harness produced by the Litmus tool [12]. Each entry in the table gives a ratio of n/m, where n is the number of times that the final result was r1 = 1 and r2 = 0; and m is the number of trials.

	POWER			ARM			
	PowerG5	Power6	Power7	Tegra2	Tegra3	APQ8060	A5X
MP	10M/4.9G	6.5/29G	1.7G/167G	40M/3.8G	138K/16M	61K/552M	437K/185M

From the above table it is clear that both POWER and ARM processors executed out-of-order instructions or contain relaxed memory operations. But, sometimes these types of behavior in a program cause incorrect/unpredictable results. To regain the program order, the programmers need to use a memory barrier explicitly between the two write operations of Thread 1 and two read operations of Thread 2, which is sufficient. On PowerPC this would be *sync/hwsync*, and on ARM it would be *dmb*, as follows:

MP+dmb/sync Pseudo code

Initial State: x = 0 ^ y = 0;	
Thread 1	Thread 2

x = 1;
dmb/sync;
y = 1;
	r1 = y;
	dmb/sync;
	r2 = x;

Forbidden : r1 = 1 ^ r2 = 0;	

The *dmb/sync* barrier causes an extra cycle in a program, but fulfills the ordering properties, if inserted between every pair of read/write operations accessing the memory. A brief explanation dealing with the four cases of pairs of read/write operations before and after a barrier is as follows:

RR: For two reads separated by *dmb/sync* barrier, ensure the memory is read in program order.

RW: For a read before a write operation separated by *dmb/sync*, the barrier will ensure that the read operation will execute before any consecutive write operation, and becomes visible to any other thread.

WR: For a write before a read operation separated by *dmb/sync*, the barrier will ensure that the write operation will execute and be visible to all other treads before the consecutive read operation.

WW: For a write before another write operation separated by *dmb/sync*, the barrier will ensure that the first write operation will execute and be visible to all other threads before any consecutive write operations.

The above properties of *dmb/sync* instructions make the program strictly sequentially consistent for memory read/write operations and add extra cost to a program. To reduce that cost IBM PowerPC introduced a 'light weight sync' instruction called *lwsync*[11], which is weaker and potentially faster than the 'heavy weight sync' instructions. The ARM does not have any instruction like *lwsync*. The properties of *lwsync* are as follows:

RR: For two read operations separated with the *lwsync* instruction it works just like sync instruction

RW: For a read before a write operation separated by the *lwsync* instruction, it also works like sync instruction

WR: For a write before a read operation separated by a *lwsync* barrier, it ensures that the write is committed before the read is satisfied, but lets the read be satisfied before the write has been propagated to any other thread

WW: For a write operation before another write operation separated by a *lwsync* barrier, it ensures that for any other thread it is observed that the first write has been propagated before the second write

Below is shown some experimental data for the Litmus test using Message Passing (MP) and *dmb/sync* instruction for ARM and PowerPC, and Message Passing (MP) and lwsync for PowerPC processors.

	POWER			ARM			
	PowerG5	Power6	Power7	Tegra2	Tegra3	APQ8060	A5X
MP	10M/4.9G	6.5/29G	1.7G/167G	40M/3.8G	138K/16M	61K/552M	437K/185M
MP+dmb/sync	0/6.9G	0/40G	0/252G	0/24G	0/39G	0/26G	0/2.2G
MP+lwsync	0/6.9G	0/40G	0/220G	-	-	-	-

The above table shows that the relaxed memory operation with message passing is observable on all platforms, while with barrier instructions the relaxed memory operation cannot be observed on any processor.

Moreover, a parallel program contains another hazard named dependency. The different dependencies are address dependency, control dependency, and data dependency. To avoid the control dependency problem, ARM introduced the isb and PowerPC introduced the isync instructions. All dependency problems can be avoided by proper use of sync/isb on ARM and sync/lwsync/isync on PowerPC processors. A more detailed Litmus test for all dependency problems has been done by [73].

4.1.1.2 Intel Processors

Intel started their processing products with guaranteed high performance but they were expensive and consumed more power using the CISC instruction set[19]. As a result, mobile devices where power consumption becomes a sophisticated issue avoid Intel processors, although most desktop and notebook computers use Intel processors. Finally, Intel realized that they had fallen behind to ARM on mobile devices and started improving the power consumption of their x86 and x64 processors. Intel began their improvements with the introduction of inexpensive low power consumption high performance products. So the end-user chips became more price competitive.

Intel multiprocessor uses the strong memory model to achieve maximum performance. They published several versions of their hardware memory model; some of them introduced a lot of changes compared to earlier versions. From August 2007, Intel published their more precise memory model with detailed analysis and examples named Intel White Paper (IWP). Prior to IWP, Intel Software Developer Manual (SDM) was published as an unofficial specification model called “processor ordering”. It was very hard to interpret based on its description, nor was it supported with any examples.

In August 2007, Intel published their first hardware memory model “Intel White Paper [49] (IWP)” which gave somewhat more clarity. In this IWP they informally proposed eight principles that were supported by 10 litmus test examples. This principle was unchanged in the later versions of IWP. These principles allow an independent reader to observe independent writes (by different processor to a different address) in a different order [23].

The most recent version of the Intel software developer’s manual is [49] which was published in February 2014. This IWP also proposed some unofficial specification style similar to the previous version and was supported by litmus tests. Some parts of the content are related to memory operations on multicore CPUs. Some other parts define strong guarantees on memory operations. These strong guarantees[49] are as follows:

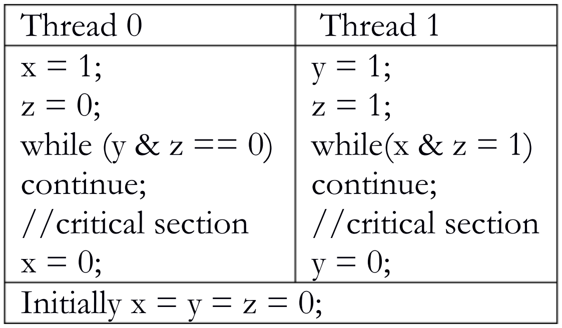
- ”Stores are not reordered with other stores.”
- ”Stores are not reordered with older loads.”
- ”Loads are not reordered with older stores to the same location.”

- ”Stores are transitively visible.”
- ”Stores are seen in a consistent order by other processors.”
- ”Stores across string operations are not reordered.”
- ”String operations are not reordered with later stores.”
- ”String operations are not reordered with earlier stores.”

Based on the above lines, it can be said that Intel processors have strong memory ordering protocols. There is no chance for out-of-order execution, so there is no need to add any extra fencing into programs to ensure memory ordering or to prevent hardware optimizations. However, according to their SDM, a few statements are relaxed enough to reorder their execution. These relaxed statements[49] are as follows:

- ”Loads may be reordered with older stores to a different location.”
- ”Intra-processor forwarding is allowed.”
- ”Stores within a string operation may be reordered.”

As mentioned above “loads may be reordered with the older stores to a different location”, which may present incorrect behavior in a program. To further clarify this statement, Peterson lock could serve as an example that can be illustrated as follows:

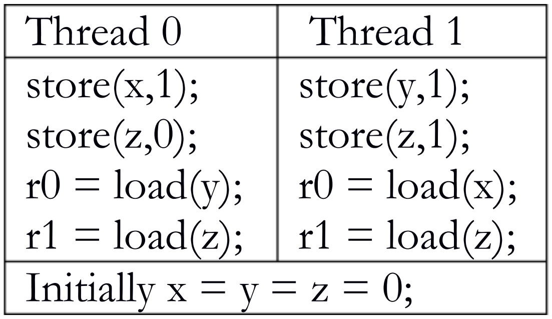


Thread 0	Thread 1

x = 1;
z = 0;
while (y & z == 0)
continue;
//critical section
x = 0;
	y = 1;
	z = 1;
	while(x & z = 1)
	continue;
	//critical section
	y = 0;

Initially x = y = z = 0;	

The above program rewritten in pseudo assembly:

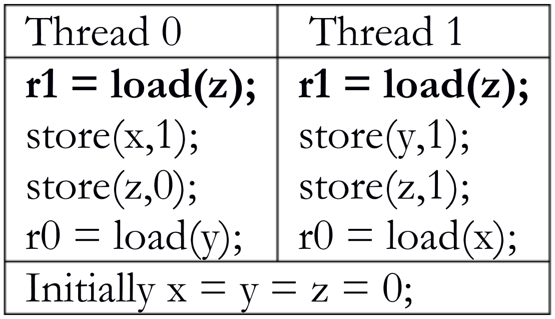


Thread 0	Thread 1

store(x,1);
store(z,0);
r0 = load(y);
r1 = load(z);
	store(y,1);
	store(z,1);
	r0 = load(x);
	r1 = load(z);

Initially x = y = z = 0;	

The interesting point in the above program is load and store operations of x and y. They follow the same pattern as the Intel reordering statement. The processor is free to move the read of y to before the write to x. Similarly, it can move the read of x to before the write to y. This may end up with the following execution scenario



Thread 0	Thread 1

r1 = load(z);
store(x,1);
store(z,0);
r0 = load(y);
	r1 = load(z);
	store(y,1);
	store(z,1);
	r0 = load(x);

Initially x = y = z = 0;	

As a result, r1 and r2 may end up with zero. In that case, the while loop never executes and both threads enter into the critical section. So, Peterson locks are broken on x86 systems. To avoid this type of incorrect performance and strengthen the memory operation explicitly, Intel introduced SFENCE instruction (presented in the Pentium III processor IA-32 architecture), as well as the LFENCE and MFENCE instructions (introduced in the Pentium IV processor). These instructions provide synchronization and memory-ordering capabilities for specific types of memory operation. With these fencing instructions, Intel also provided a few statements[49] of the model as follows.

- ”Locked instructions have a total order.”
- ”Loads are not reordered with locks.”
- ”Stores are not reordered with locks.”

Finally, although Intel started with the strong memory model, nowadays they began to develop relaxed memory designs for mobile devices. In 2012, Intel said that at least 70

4.1.1.3 Comparison of Different Hardware Memory Models

Intel and ARM chips have different processor architectures and instruction sets. ARM and PowerPC are RISC (Reduced Instruction Set Computers) based, while Intel x86 is a CISC (Complex Instruction Set Computer) architecture [64]. This means an application compiled on Intel architecture cannot run on ARM and vice-versa. For example, Windows RT and Windows 8 are both Microsoft operating systems, but Windows RT runs on ARM architectures and Windows 8 runs on Intel architectures. However, an application running on Windows RT is incompatible with Windows 8.

To compare different hardware memory models, the Instruction Set Architecture (ISA) is a big concern for any processor system. ARM and IBM PowerPC both implemented Reduced Instruction Set Computing (RISC) which contains a fixed length instruction set, relatively simple for writing code. On the other hand, Intel x86 implemented Complex Instruction Set Computing (CISC) which has a variable length instruction set, relatively complex for writing code. Moreover, CISC has decode latency which prevents pipelining and results in slow decoding and higher code density. To reduce this decode latency, Intel has to implement a decode optimizer for general instructions and I-cache to reduce the code density impact. Concerning the complexity, CISC instructions are complex, multi-cycle instructions requiring encryption and string manipulation. On the contrary, RISC systems have simple, single function, single cycle instructions. However, the size of the code written on RISC is much higher than on CISC systems. The most important observation is that the number of data accesses from the cache is similar in both ISAs, because CISC instructions split into RISC-like microprocessor operations. Currently available research and tests show that expressing more semantic information has led to improved performance [], better security, and better visualization. Moreover, current research shows that some extensions of hardware allow to balance accuracy with efficiency [42] [34] and unique hardware extensions for energy efficiency [47].

In summary, most of hardware memory model specifications that were published by processor vendors are excessively complex, incomplete and ambiguous enough that they may be misinterpreted even by experts. For instance, such is the Intel x64 and IA-32 Architecture Software Developers Manual [49]. Furthermore, since hardware models mostly focused on hardware optimizations, which have often been not well-matched to software design requirements, the result was loss in performance or incorrect code.

4.1.2 Software Memory Models

Among the high-level programming languages, Ada 83 [68] was possibly the first high-level programming language that provided first-class support for shared memory models, although Ada's initial thread synchronization approach was significantly different from the currently popular high-level language memory design. However, until mainstream programming languages (e.g. Java) introduced memory design principles, thread and shared memory programming mostly used libraries and APIs such as OpenMP and Posix threads. When parallel programming started to be used for general purposes without a clear memory model coming from programming languages defining the context of threads, it was unclear what the programmer was allowed to assume and what compiler transformations were legal.

4.1.2.1 Java Memory Model

History:

1991: Oak was developed under the name Green Project by James Gosling and another developer within 18 months[26]. The project was initiated by Sun Micro-systems to develop a kernel for executing object-oriented and multithreaded software on set-top Boxes featuring security and portability.

Oak programming was designed to target interactive television market, but for that period it was too advanced a technology for the digital cable television industry.

1996 (January 23): The first stable version for Java was released officially as JDK 1.0[26]. It was also known as Java 1.

The original Java memory model was developed in this version, and was widely perceived as broken, preventing many run-time optimizations and providing guarantees not strong enough for code safety. Moreover, final fields could be observed to change and finalization semantics were unclear.

2004: Original memory model was updated through Java Community Process, like Java Specification request 133 (JSR-133), for Java 5.0 (code name Tiger). [72]

Java language community updated their language specification as Java Specification Request (JSR). JSR-133 is related to memory operations and synchronization mechanisms which describe the semantics of locks, threads, volatile variables and data races. The goals of JSR 133 were as follows [84]:

- Clear and easy to understand.
- Provide out-of-thin air safety.
- Foster reliable multi-threaded code
- Allow for high performance Java Virtual Machine (JVM).
- New synchronization idioms for security guarantees.
- Minimal impact on existing code.

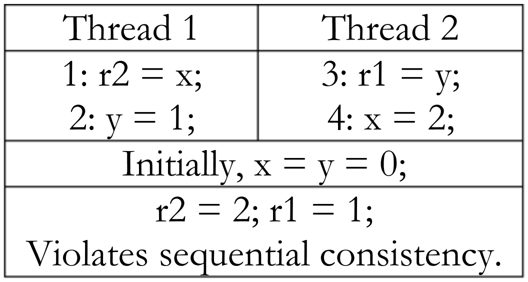
Programming languages had to face two implementation challenges that do not occur in hardware systems. First, many programming languages provide safety and security properties of their code that must be respected. Second, while performing optimization the ability of the compiler to perform subtle and global analysis of all variables constrains program execution order. Research [85] [86] shows that the previous version of the Java memory system had serious problems. To overcome these issues, the Java memory model was revised in 2005. The new design provides a clear notation how to write a correct program, gives greater flexibility to programmers and provides clear semantics defining both incorrect and correct programs.

Regardless of these fundamental changes, the programming style and program quality remain the same. The memory model needs to keep a balance between implementation flexibility for the system designer and the programmer's ease-of-use.

The major goals of the new Java memory model were to provide a balance between sufficient ease-of-use and transformations and optimizations in current compilers and hardware. However, current compilers and hardware transformations violate sequential consistency of programs. For this reason, currently it was not possible to give a strong sequentially consistent memory model for Java. Currently the Java memory model is relaxed (a weaker model). A research paper provided details about the requirements of the Java memory model and their evolution [71].

Program synchronization:

The first requirement of the Java memory model is to provide a sequentially consistent data-race-free programming model. Programmers need to worry about the impact of their program code transformation on their program result only if that program contains data races, as in the following example:



Thread 1	Thread 2

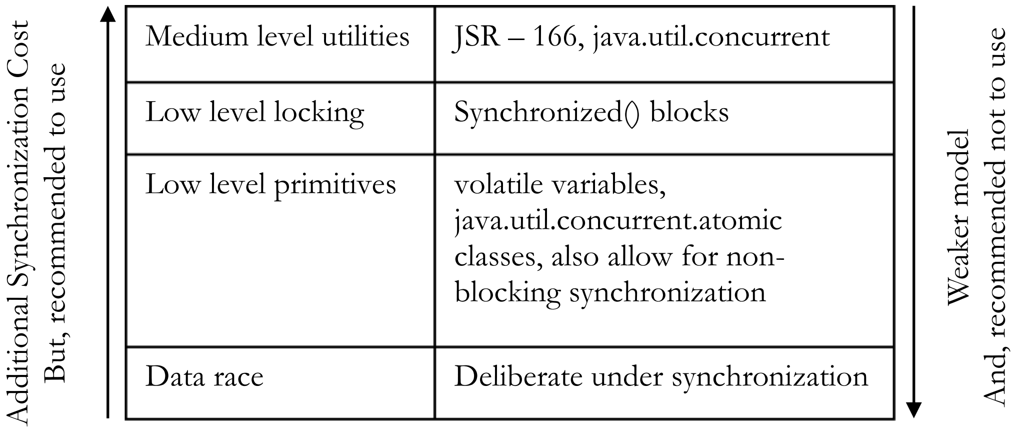
1: r2 = x;
2: y = 1;
	3: r1 = y;
	4: x = 2;

Initially, x = y = 0;	
r2 = 2; r1 = 1; Violates sequential consistency.	

Tab. 4.1: A Violation of Sequential Consistency

The above example is incorrectly synchronized. The global variables x and y have access conflicts for any sequentially consistent execution. So, these conflicting accesses are not ordered by the happens-before relation. A possible way to make this program correctly synchronized is by declaring the global variables volatile. Volatile variables ensure the sequential consistency in the program.

To make the program code data-race-free, the Java specification provides a different level of synchronization mechanisms [84]. A hierarchical table of those mechanisms is as follows:



Additional Synchronization Cost
But, recommended to use

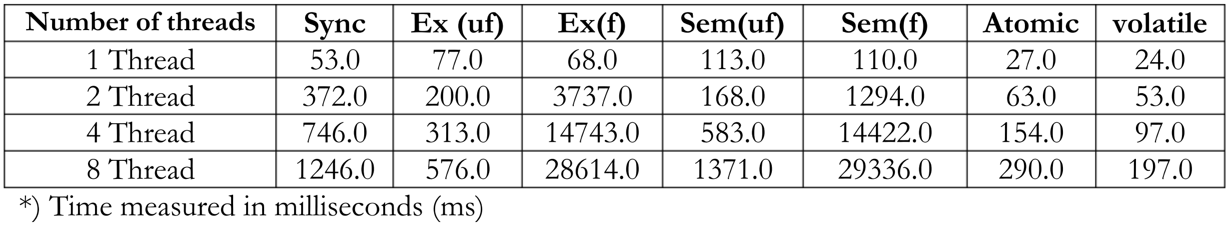
Medium level utilities	JSR – 166, java.util.concurrent
Low level locking	Synchronized() blocks
Low level primitives	volatile variables, java.util.concurrent.atomic classes, also allow for non-blocking synchronization
Data race	Deliberate under synchronization

Weaker model
And, recommended not to use

Tab. 4.2: Hierarchical table of Java synchronization mechanisms

From the above model the programmer can determine the required synchronization mechanisms for their programs. As shown in the table, the weaker synchronization mechanism provides higher efficiency but lower security guarantees; on the other hand, stronger (medium level) mechanisms give higher security guarantees but incur additional cost.

A performance check has been done using different Java synchronization mechanisms (synchronized block, semaphores fair and unfair, explicit locks fair and unfair, atomic variables). The results are as follows:



Number of threads	Sync	Ex (uf)	Ex(f)	Sem(uf)	Sem(f)	Atomic	volatile
1 Thread	53.0	77.0	68.0	113.0	110.0	27.0	24.0
2 Thread	372.0	200.0	3737.0	168.0	1294.0	63.0	53.0
4 Thread	746.0	313.0	14743.0	583.0	14422.0	154.0	97.0
8 Thread	1246.0	576.0	28614.0	1371.0	29336.0	290.0	197.0

*) Time measured in milliseconds (ms)

Tab. 4.3: Java synchronization mechanisms performance table

The test is made using a little piece of code written for this thesis. Each method executed 600,000 times exactly.

Hardware platform: The test was executed on a Windows 8.0 machine with the Oracle Java 7 virtual machine. The computer featured a 32 bit Core 2 Duo 1.8 GHz processor with 2GB of DDR2 memory.

Out-of-thin air safety:

The previous strategy of leaving semantics of the Java language was incorrect, as the program is inconsistent with Java safety and security guarantees. Sometimes the program's read operations may read unexpected values. The reason behind this could be an interruption, or cache memory value changes by another thread. The following illustrates this on an example:

Thread 1	Thread 2

1: r1 = x;
2: y = r1;
	3: r2 = y;
	4: x = r2;

Initially, x = y = 0;	
Incorrectly synchronized, java want to disallow r1 = r2 = 42;	

Tab. 4.4: An Out of Thin Air Result

Although the above program is incorrectly synchronized, getting the result 42 is not allowed by the Java semantics. Java provides out-of-thin air value safety by default from Java 2005[84] on.

Happens-before relation:

The happens-before memory model is a synchronization order over a synchronization mechanism[72].

Thread 1	Thread 2

1: x=1;
2: go = true;
	3: if(go)
	4: r1 = x;

Initially, x = 0, go = false;	
If r1 = x execute, it will read 1.	

Tab. 4.5: Happens-Before relations example

Happens-before consistency says that if a read operation r reads a variable v, it is allowed to observe a write w of v in the happens-before partial order of execution:

- if $w \xrightarrow{hb} r$, write operation w happens before read r, the write operation is not observable if it happens after read.
- there is no intervening write w' between these two operations that are ordered by happens-before relation. $w \xrightarrow{hb} w' \xrightarrow{hb} r$, the write was not overwritten by another write operation along happens-before path.

The Java Language Specification does not provide any guarantees of preemption for multi-threaded programming, nor any fairness. The reason behind this is such guarantees would make the specification complicated by issues such as thread priorities and real-time threads[84]. On the contrary, Java Virtual Machine provides some fairness guarantees, but it depends on the particular hardware specification.

4.1.2.2 The C++ Memory Model

The C/C++ programming language style hung on to single core CPUs or single threaded languages, without any reference to threads. The most recent version of the C++ programming language standard is C++11 (formerly known as C++0X) which supports multi-threaded programming.

History of the C++ memory model:[87]

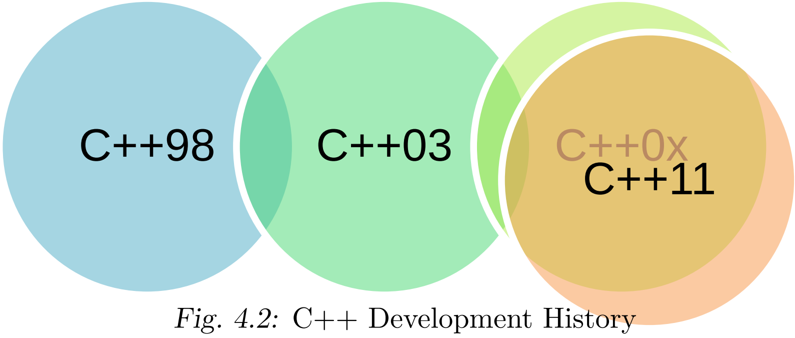


Fig. 4.2: C++ Development History

The C++ language was initiated as a development of C programming language by including classes, then multiple inheritance, operator overloading, virtual functions, exception handling and templates, among other features. After several years of the development process, in 1998, an official version of the C++ programming language was standardized and published as ISO/IEC 14882:1998.

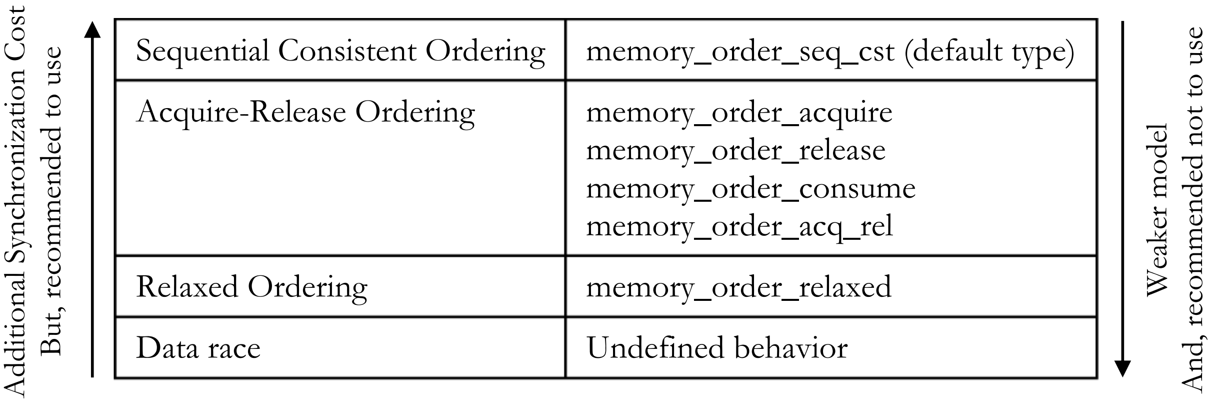
After five years, the standard was revised by the technical Corrigendum at 2003, ISO/IEC 14882:2003.

Finally, in September 2011 the current standard extended C++ with new features approved and published by ISO as ISO/IEC 14882:2011 [55] (informally known as C++11).

The C++ Memory Model:

Prior to C++11, C++ was specified as a single threaded language, multi-threading was supported only by libraries such as pThreads. C++11 integrated thread specification into its new standard and now defines the behavior of multithreaded applications. Moreover, for low level synchronization it introduces and simplifies atomic operations that allow writing lock-free algorithms. As a result, more performance conscious way of programming can be achieved.

C++11 provides distinct memory ordering semantics that can have varying cost on different CPU architecture. This ordering gives fine grain control over the visibility of the operations by processor. The following table shows a hierarchical diagram of memory ordering constrains in C++11[23].



Additional Synchronization Cost
But, recommended to use

Sequential Consistent Ordering	memory_order_seq_cst (default type)
Acquire-Release Ordering	memory_order_acquire memory_order_release memory_order_consume memory_order_acq_rel
Relaxed Ordering	memory_order_relaxed
Data race	Undefined behavior

Weaker model
And, recommended not to use

Fig. 4.3: Hierarchical Table of C++ Synchronization Mechanisms

Synchronization mechanisms: there might be additional synchronization need for sequentially consistent ordering over the relax ordering or acquire-release ordering and for acquire-release ordering over relaxed ordering. In a multiprocessor system, the synchronization instructions add extra costs to the program and decrease overall program efficiency [109]. Currently, C++11 supports following synchronization mechanisms.

Mutex locks $< mutex >$:

A *mutex* is a synchronization mechanism that is designed to be use in programs to exclusively use some part of the program code [23]. It is a standard low level locking primitive. It prevents other threads from using resources concurrently by providing protection. Pseudo code for *mutex* lock is as follows:

```
    mutex.lock(1)
     r1 = z;
     z = r1 + 1;
    mutex.unlock(1)
```

mutex objects do not support recursion (i.e., a thread shall not lock a mutex it already owns) and provide exclusive ownership. It is guaranteed to be a standard-layout class. [www.cplusplus.com/reference/mutex/mutex/]

Condition_variable < *condition_variable* >:

A *conditionvariable* is a low level synchronization primitive object that can block the calling thread until *notified* to resume[23].

```
    condition\_variable.wait(unique\_lock);
     r1 = z;
     z = r1 + 1;
    condition\_variable.notify\_all();
```

To lock a thread, it uses a *unique_lock* when one of its *wait_function*s is called. Until another thread wakes up this lock, the thread that calls a *notification_function* remains blocked on the same condition variable object.

Ordered Atomic:

Memory acquire and release operations done automatically with atomic variable reads and writes respectively[23].

```
    int x;
    std:: atomic<int> r
    while (r!=4)//acquire exclusively
     read/write of x
    r = x;      //release exclusively
```

Transactional memory:

This concept is still at research level now, but a rough idea is given by the following[91]:

```
    Atomic{     //acquire exclusively
     read/write of x
    }           //release exclusively
```

The C++11 standard provides a clear definition about the memory model and standard operations. However, no standard compiler yet fully implement the C++11 standard. Visual C++ 2013 has not yet implemented data dependency ordering, i.e. function annotation and atomic in signal handlers [http://msdn.microsoft.com/en-us/library/hh567368.aspx]. GCC 4.9 does not yet implement variable templates, features relaxing requirements on constexpr functions and member initializers, aggregates and sized deallocation [http://gcc.gnu.org/projects/cxx1y.html].

4.1.2.3 Ada Memory Model

Ada is a modern object-oriented, highly reliable and efficient high level programming language designed for long-lived, large applications and embedded systems in particular. Nowadays, the places where high security/high integrity/safety critical domains are required include medical devices, air traffic control, commercial and military aircraft avionics, railroad systems, all of which use Ada programming. Ada is a multi-faceted programming language. It has structured control statements, strong type checking, simple syntax, flexible data composition facilities, a mechanism for exception handling and code modularization features[2].

History of the Ada programming language:[17]

1983: First reference manual for military standard 1815A, ANSI standard published and also the Ada/Ed implementation of the language was validated the same year.

1995: The older version of Ada was revised and published, a new joint ANSI and ISO standard was validated. It described overall objectives and scope of Ada95. Its Reference Manual summarized the significant difference between Ada83 and Ada95.

2005: Ada language was revised using the basic structure of Ada83 and Ada95. ISO/IEC standardization process was approved and published as ISO/IEC 8652:1995/Amd 1:2007. The Rationale for Ada 2005 described the difference between Ada95 and Ada2005.

2012: Ada language was revised again introducing safe, reliable and secure features for modern processors. The current ISO/IEC standard version for Ada is ISO/IEC 8652:2012. The Rationale for Ada 2012 provides the differences between Ada2005 and Ada2012.

The Ada memory model:

Ada provides first class support for concurrent programming. Among all high level mainstream programming languages Ada83 was the first language that provided clear support for shared memory models. From the beginning of Ada, it was always concerned with parallel architectures. Ada provides the following three types of task synchronization mechanisms.

Protected Objects:

A protected object gives mutually exclusive access to shared data through calls on its visible protection operations. The access could protect entries or protected subprograms. Protected declarations define protected units, which contain a corresponding protected body. A protected declaration may be a *sing_type_declartion* or *protected_type_declaration*. The *protected_type_declarations* declares a named protected type while *single_protected_declaration* defines an anonymous protected type. The syntax of protected objects is as follows[39]:

```
protected_type_declaration ::=
      protected type defining_identifier [known_discriminant_part] is
      protected_definition;

   single_protected_declaration ::=
      protected defining_identifier is protected_definition;

   protected_definition ::=
      { protected_operation_declaration }
   [ private
      { protected_element_declaration } ]
   end [protected_identifier]

   protected_operation_declaration ::=
        subprogram_declaration
      | entry_declaration
      | aspect_clause

   protected_element_declaration ::=
        protected_operation_declaration
      | component_declaration
```

Protected type does not require predefined assignment or comparison operators because it is a “limited type”.

Suspension objects:

The *suspension_object* type is described in Real-Time annex; it is used for event-based synchronization between two tasks. The syntax for suspension object is as follows[39]:

```
package Ada.Synchronous_Task_Control is
  type Suspension_Object is limited private;
  procedure Set_True(S : in out Suspension_Object);
  procedure Set_False(S : in out Suspension_Object);
  function Current_State(S : Suspension_Object) return Boolean;
  procedure Suspend_Until_True(S : in out Suspension_Object);
private
     ... -- not specified by the language
end Ada.Synchronous_Task_Control;
```

The *suspension_object* type is a by-reference type object, it has two visible states: false and true. The default value is false.

Rendezvous:

The Rendezvous concept is Ada's universal mechanism of synchronization and data exchange between tasks[2]. Two tasks, an emitter and a receiver, synchronize by executing the code sequence together on the receiver's side. An accept instruction, a so-called call point, introduces this code sequence. The call points are also listed in the task's declarative part as shown in the following figure. Like procedures, they can feature input and output parameters that realize the data exchange and successful communication.

```
-- Declarative Part:
task myTask is
   entry CallPoint1;
   -- further entries possible
end myTask;

-- Body:
task body myTask is
   -- possibly variable decl.
begin
   -- possible statements...
   accept CallPoint1 do
     -- InstructionSequence1
   end CallPoint1;
   -- ...
end myTask;
```

```
-- Declarative Part:
task myTask is
   entry CallPoint1;
   -- further entries possible
end myTask;

-- Body:
task body myTask is
   -- possibly variable decl.
begin
   -- possible statements...
   accept CallPoint1;
   -- ...
end myTask;
```

Fig. 4.4: Scheme of a task with a call point

If there is no data transfer between the tasks and the call point serves for synchronization only, there is usually no need for jointly executed code sequence (that would be "Instruction Sequence 1" in Figure on the left). In this case, the accepted instruction can be abbreviated as shown in the figure on the right side.

Mutual exclusion mechanism:[2]

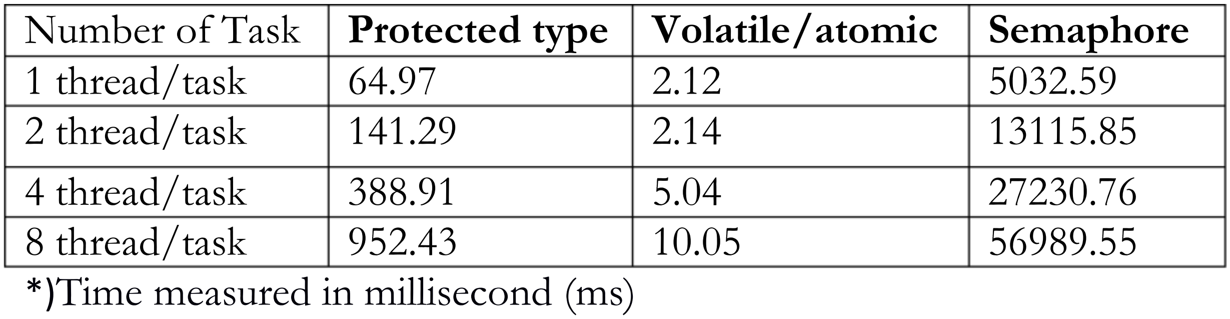
volatile: Specified via a pragma, which may not be optimized into a "temporary" location such as a register.

atomic: Specified via a pragma, whose accesses are indivisible (with respect to task context switching) because of the hardware, and which may not be optimized into a "temporary" location such as a register

protected object/type: With associated protected operations, based on the concept of "concurrent read, exclusive write" locks and designed for efficient implementation.

passive task: Expressed as a loop around an accept or $selective_accept$ statement.

A performance check has been done using different Ada mutual exclusion mechanisms (protective type, volatile/atomic/Semaphore). The results are as follows:



Number of Task	**Protected type**	**Volatile/atomic**	**Semaphore**
1 thread/task	64.97	2.12	5032.59
2 thread/task	141.29	2.14	13115.85
4 thread/task	388.91	5.04	27230.76
8 thread/task	952.43	10.05	56989.55

***)**Time measured in millisecond (ms)

Tab. 4.6: Ada mutual exclusion mechanism performance table

The test is made using a little code written for this thesis. Each method was executed 600,000 times exactly.

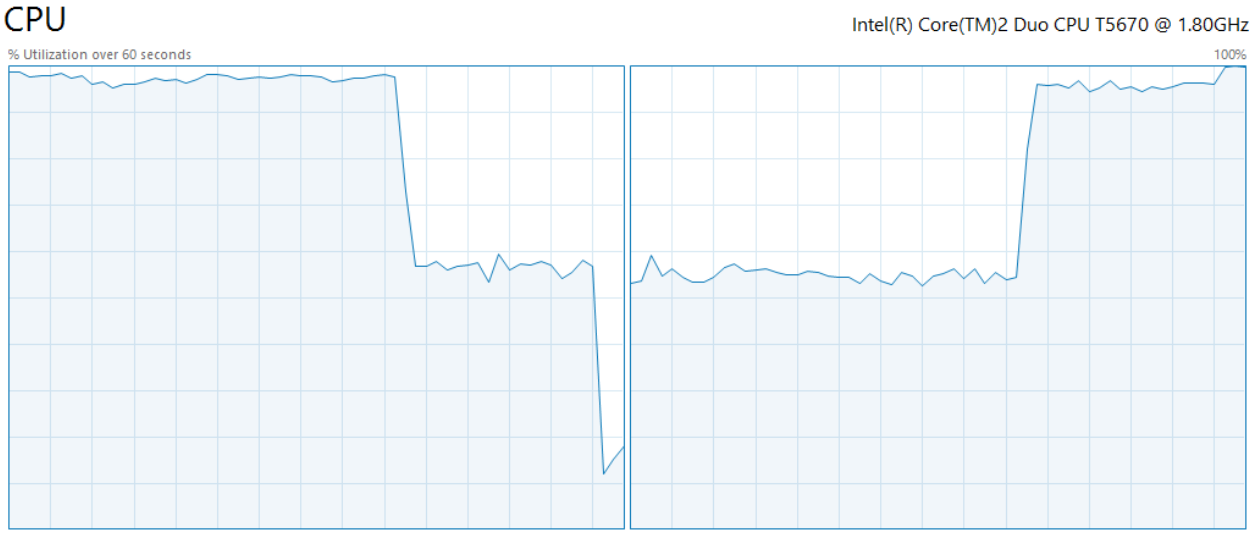
Hardware platform: The test was executed on a Windows 8.0 machine with the Oracle Java 7 virtual machine. The computer features a 32 bit Core 2 Duo 1.8 GHz processor with 2GB of DDR2 memory.

Ada2012 includes new features in the new standard called explicit processor allocation found in Real-Time System annex[2].

A new package named *System.Multiprocessor* was introduced as follows[2]:

```
"package System.Multiprocessors is
    typeCPU_Rangeis range0..implementation-defined;
    Not_A_Specific_CPU: constantCPU_Range := 0:
    subtypeCPU isCPU_Range range1 .. CPU_Range'Last;
    functionNumber_Of_CPUs returnCPU;
endSystem.Multiprocessors;"
```

But, in practice it is broken; sometimes it swaps CPU cores, a program with string out operations keeps other processors busy as shown in the following figure.


CPU
Intel(R) Core(TM)2 Duo CPU T5670 @ 1.80GHz
% Utilization over 60 seconds
100%

Tab. 4.7: Ada explicit processor allocation check

4.1.2.4 Comparison of different software memory models

As comparison of different aspects of different programming language memory models one could use practical approaches (e.g., efficiency measurement for different hardware, synchronization mechanism validation, performance check, program portability check, hardware compatibility check), or, abstract level ones (compare different research papers and different language protocols). Here only the abstract level comparison has been done because of resource limitation. For abstract level comparison, the faced difficulty was insufficient number of research papers for comparison of different memory models. The reason behind it is the fact that most definitions of memory models defined by the programming languages are recently published (e.g. C++ at 2011, Ada at 2012).

Portability :

For portability Java says that "compile once run anywhere [65]" is true because of Java's use of a virtual machine to run programs on different platforms. On the other hand, Ada and C++ say that "write once compile anywhere [90]" means that to run an Ada or C++ program on a platform like Linux requires a respective compiler for that particular language. So, when comparing these three different languages, Java is more portable than C++ and Ada.

Correctness :

In multiprocessor programming, especially in a shared memory environment, the main problem is incorrect behavior of programs. The main reason behind this is compiler optimizations. Java, C++ and Ada provide mechanisms for synchronization. Using these mechanisms in a program, one can regain the correct behavior but has to compromise performance. The reason behind this is that synchronization mechanisms add extra clock cycles into programs. To reduce the number of these extra clock cycles, all three programming languages provide non-blocking synchronization called mutual exclusion mechanism. Although it minimizes the extra clock frequency compared to the blocking synchronization mechanism, it makes it very hard to program even for experts.

Flexibility :

In multiprocessor programming flexibility rules are a significant issue of concern for programs, for instance, scheduling policy and explicit processor allocation. In a real-time environment poor thread/task scheduling policy can cause deadline violations, while the result could be catastrophically dangerous. Java and C++ provide implementation defined scheduling, whereas Ada uses specific scheduling [25]. Moreover, Ada 2012 included explicit processor allocation into its specification [39]. However, currently this specification is broken as shown in the Ada memory model section, it is expected to be fixed in the next update.

4.2 Real-time systems

Most embedded systems are bound to real-time constraints. “Real-time” system means that an IT system is no longer in control of its own time domain. Now it may progress with time of the physical world or its time may be artificially generated by some surrounding environment. “A real-time computer system is a computer system in which the correctness of the system behavior depends not only on the logical result of the computation, but also on the physical instant at which these results are produced.”[63] It can be concluded that in real-time systems the program logic of application tasks has to be designed in accordance with the timing mechanism of that particular hardware. Many tasks have to be executed concurrently on an embedded computing system. Such situations are usually handled by some kind of operating system, named Real Time Operating System (RTOS). In RTOS, the worst-case execution time (WCET) on a specific target architecture of any real-time task has to be available. WCET should be as small as possible. Sometimes, over-estimating of WCET could reduce efficiency of the implemented system.

4.2.1 Predictability in real-time systems

In a large complex real-time system, 100% guarantees of scheduling must be relaxed [96]. The following example will demonstrate a detailed idea about this:

Let a complex, large, real-time system be operating in a non-deterministic environment. It has both loose-time constraint and time constraint tasks; it has soft and hard real-time tasks; some of its tasks are critical; part of the system may be highly static, but many parts require a dynamic approach. In other words, all dimensions of a possible real-time system are included simultaneously into the system. Designers cannot focus on one predominant feature of the system such as tight time constraints, or only be bound to the interrupt latency, or assume a fixed set of periodic tasks. In this system, it is difficult to define and demonstrate predictability. The notion that the system can obtain a 100% guarantee must be relaxed.

Designers have to show that the requirements are met at two levels of detail and for each class of tasks. At the macroscopic level, it has to be shown that all critical tasks will always meet their deadlines (100% guarantee) and then non-critical tasks both soft and hard deadline tasks meet overall requirements. As an example, 98% of hard real-time and 94% of soft real-time tasks meet their deadlines. At a microscopic level (on the task group and individual task level), also have to achieve some level of predictability. Critical tasks are already defined and always meet their deadlines. For other real-time tasks, their performance depends on the state of the system. Scheduling algorithm can define the state of the system and at any point the system can identify exactly which tasks will meet their deadlines. This will give the ability to handle overloads, and the ability to make more intelligent decisions concerning the overall operation of the system [95].

4.2.1.1 Achieving predictability

There could be two different ways to achieve predictability in complex real-time systems [96]. The first is layer-by-layer approach and the second is top layer approach. Layer-bye-layer approach requires both macroscopic and microscopic predictability while top layer approach requires only macroscopic predictability. In addition, these approaches can be merged.

Layer-by-Layer approach:

In this approach, to obtain a predictable system, it is necessary to have a tight interaction among all aspects of the system starting from programming language to the compiler, to the operating system, to the hardware, to the design rules and constraints used. With a precise hardware and software design, it is possible to achieve both macroscopic and microscopic predictability. The worst case execution time of any task can be computed at the microscopic level. Furthermore, it also provides performance analysis of non-critical hard real-time tasks giving the expected normal and overload workloads.

In some circumstances, layer-by-layer approach may not satisfy the requirements, because everything is not 100% guaranteed. However, this calculation is necessary and unavoidable in a complex, non-deterministic real-time environment.

Top layer approach:

In the top layer approach, the calculation concentrates on application layer predictability requirements. The lower layer only provides services for predictable application layer, so lower level predictability is not required. In a critical real-time system, the error handlers support fail-safe and fail-stop behavior; even if predictability has been proven to work as required at each of the layers [96]. The job of the error handler is to detect predictability errors in an algorithm, it also provides the guarantee to return the system to a safe state. So, it is important to include an error handler into the scheduling algorithm with a 100

The top-layer approach is most suitable for complex activities. For example, sometimes it may not be possible to break a complex activity into different layers. Furthermore, in the situation where the bounds found at one layer are based on bounds at the lower layers, they may be so high as not to be of any practical value. For example, if two sub-problem activities execute on two different nodes and use a common communication channel, then both activity and communication channels are bound at the node that takes the maximum time to execute.

5. Programming Paradigms

5.1 Introduction

A fundamental style of computer programming is programming paradigm, a way of building elements and structures of a computer program. A programming paradigm is an approach based on a coherent set of principles or mathematical theory to program a computer. There exist several programming paradigms. All paradigms support a set of ideas and rules that make them the best for a particular problem. For example, object-oriented programming paradigm is most popular for problems with a large number of related data abstractions organized in a hierarchy. On the other hand, logic programming is well known for being able to transform or navigate complex symbolic structures according to rules of logic.

There are many programming languages on the market, and practical programming languages are usually quite complicated.

On the programming language side, the language should ideally support many ideas (paradigms) in a well factored way, so that the programmer can choose the best ideas whenever they are needed. Unfortunately, popular mainstream languages such as Java, C++ or Ada support just one or two separate paradigms[104], different programming concepts need to solve different programming problems cleanly, and the available paradigms often do not provide the best ideas. For example, if programmers need to model many independent activities, then the programmer will have to implement different execution stacks, a scheduler, and a mechanism for switching execution from one activity to another. All this complexity is unnecessary if programmer adds one concept to the language: concurrency.

The number of programming paradigms is much smaller than the number of programming languages. So, it would be easier to learn programming paradigms rather than learn a huge number of programming languages. From this point of view, such languages as C#, Java, Python, Javascript and Ruby are all virtually identical: they all use the object-oriented paradigm with only minor differences[104].

Although the number of programming paradigm is much smaller than the programming languages, there are still many paradigms. Figure5.1 [104] mentioned 27 different paradigms that are actually used, and all have reasonable implementations and practical applications. Fortunately, these paradigms have a lot of features in common. In the figure5.1, when a language is mentioned under a paradigm, it means that this particular language is intended to support that paradigm without interference with the other paradigms.

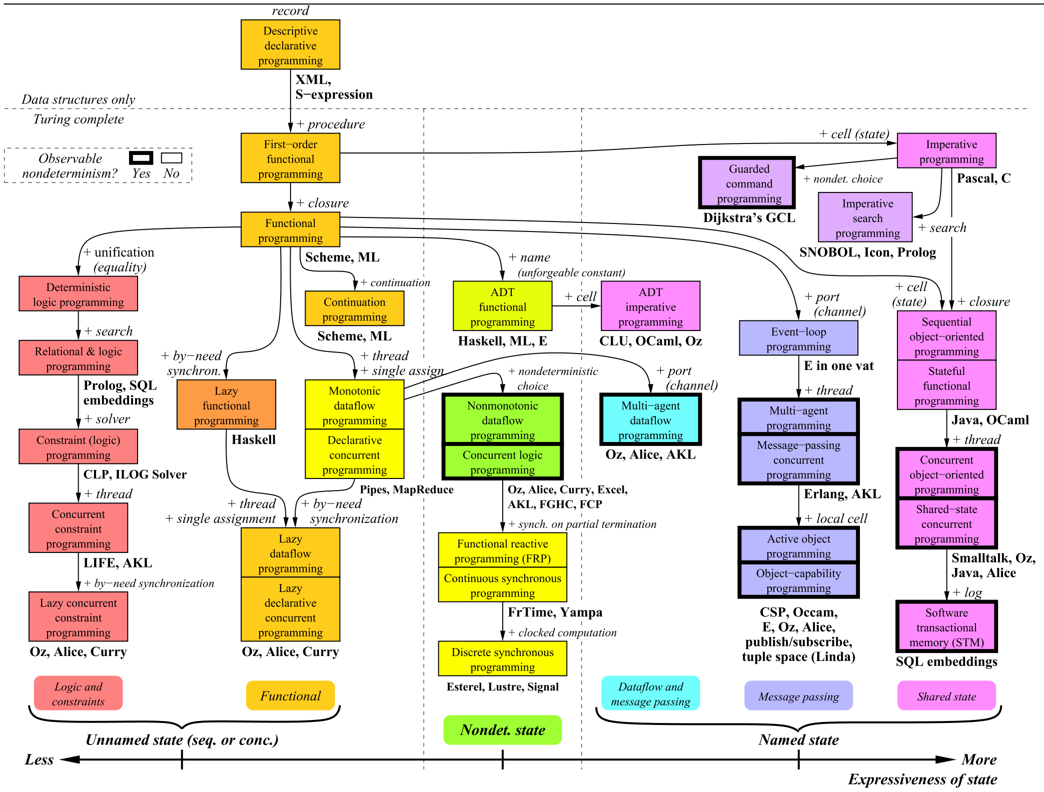

record
Descriptive declarative programming
XML, S−expression
Data structures only
Turing complete
+ procedure
Observable nondeterminism? Yes No
First−order functional programming
+ cell (state)
Imperative programming
Pascal, C
+ nondet. choice
Guarded command programming
Dijkstra's GCL
Imperative search programming
+ search
SNOBOL, Icon, Prolog
+ closure
Functional programming
+ unification (equality)
Scheme, ML
+ continuation
Continuation programming
Scheme, ML
+ name (unforgeable constant)
ADT functional programming
+ cell
ADT imperative programming
Haskell, ML, E
CLU, OCaml, Oz
+ port (channel)
Event−loop programming
E in one vat
+ thread
+ cell (state)
+ closure
Sequential object−oriented programming
Stateful functional programming
Java, OCaml
+ thread
Deterministic logic programming
+ search
Relational & logic programming
Prolog, SQL embeddings
+ solver
Constraint (logic) programming
CLP, ILOG Solver
+ thread
Concurrent constraint programming
LIFE, AKL
+ by−need synchronization
Lazy concurrent constraint programming
Oz, Alice, Curry
+ by−need synchron.
Lazy functional programming
Haskell
+ thread
+ single assign.
Monotonic dataflow programming
Declarative concurrent programming
Pipes, MapReduce
+ thread
+ single assignment
+ by−need synchronization
Lazy dataflow programming
Lazy declarative concurrent programming
Oz, Alice, Curry
+ nondeterministic choice
Nonmonotonic dataflow programming
Concurrent logic programming
Oz, Alice, Curry, Excel, AKL, FGHC, FCP
+ synch. on partial termination
Functional reactive programming (FRP)
Continuous synchronous programming
FrTime, Yampa
+ clocked computation
Discrete synchronous programming
Esterel, Lustre, Signal
+ port (channel)
Multi−agent dataflow programming
Oz, Alice, AKL
Multi−agent programming
Message−passing concurrent programming
Erlang, AKL
+ local cell
Active object programming
Object−capability programming
CSP, Occam, E, Oz, Alice, publish/subscribe, tuple space (Linda)
Concurrent object−oriented programming
Shared−state concurrent programming
Smalltalk, Oz, Java, Alice
+ log
Software transactional memory (STM)
SQL embeddings
Logic and constraints
Functional
Nondet. state
Dataflow and message passing
Message passing
Shared state
Unnamed state (seq. or conc.)
Named state
Less
More
Expressiveness of state

Fig. 5.1: Taxonomy of programming paradigms [104]

Observable non-determinism:
Non-determinism arises when program execution is not completely determined by its specification, i.e. during the execution of a program the language specification allows the compiler to choose the next step. Non-determinism can be highly undesirable, because programmers can see different results of a program execution on the same internal configuration. Typical effect is *a race condition*, *dead locks*[104].

Named State:
The strongest to support the state is the second key property of a paradigm. The ability to remember information is called State, or more precisely, the ability to store a sequence of values in time. Support of state by different programming paradigms differs. This level of support in paradigm can be separated into three axes of expressiveness[104] depending on whether the state is named or unnamed, deterministic or non-deterministic, sequential or concurrent. Figure5.2 arranges this into a lattice;

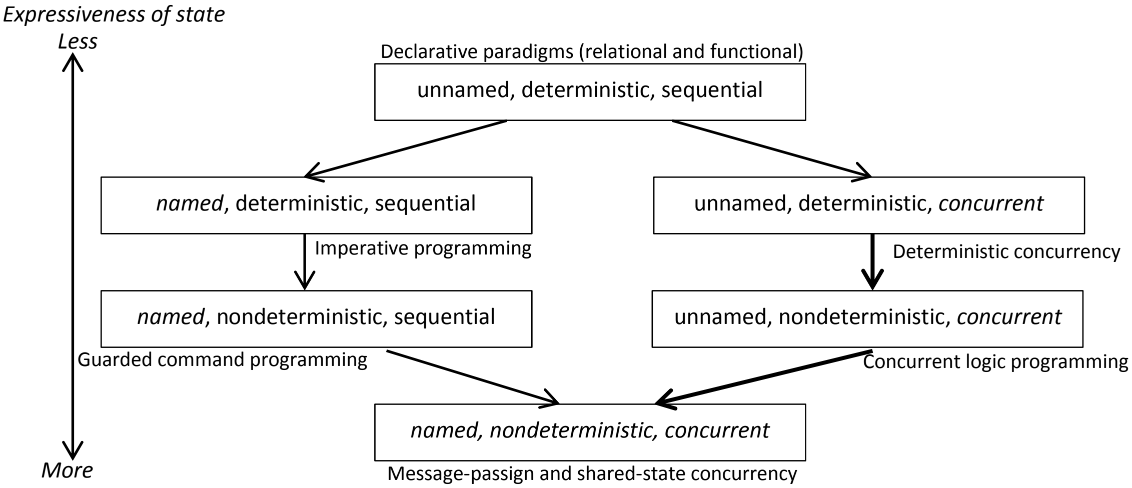

Expressiveness of state
Less
More
Declarative paradigms (relational and functional)
unnamed, deterministic, sequential
named, deterministic, sequential
unnamed, deterministic, concurrent
Imperative programming
Deterministic concurrency
named, nondeterministic, sequential
unnamed, nondeterministic, concurrent
Guarded command programming
Concurrent logic programming
named, nondeterministic, concurrent
Message-passign and shared-state concurrency

Fig. 5.2: Different levels of support for state[104]

5.2 Types of Programming Paradigms

Although there are a number of programming paradigms, mainstream paradigms can be of 4 types.

1. Imperative (Procedural) Paradigm.
2. Functional (Applicative) Paradigm
3. Logic (Rule Based) Paradigm.
4. Object Oriented Paradigm

5.2.1 Imperative Paradigm

Imperative programming (also known as procedural programming) works by modifying a memory state from a sequence of instructions[41]. The principle is almost close to actual computer structures. The current state of the program is defined by all of the variable values, the locations of the next execution statements and currently active internal data that need to return from the subprogram. The state of program changes with the change of variable values after execution of each statement. Thus, an imperative program execution can be viewed as the executing line of sequential instructions. Examples of imperative languages are $Ada, C, Pascal, Cobol$ and $Fortran$.

5.2.2 Functional Paradigm

Functional Paradigm (also known as Applicative) is a programming style which concerns itself with what has to be computed[41]. This style prevents one from having any global variables. Function parameters are used to pass the variable values and, therefore, variable values are stored locally to all processors. Implementing efficient code with this style is harder than with the other programming styles. $Haskell, Scheme, LISP$ and $Miranda$ languages use Functional Program style.

5.2.3 Logic Paradigm

Logic programming (also known as Rule-Based programming) is a programming paradigm intended to solve problems using predicate logic, in which the basic concept is a relation[108]. The main focus of logic programming is to achieve the goal rather than focusing on performance of the program. Examples of major logic programming languages include *Datalog* and $Prolog$.

5.2.4 Object Oriented Paradigm

Object Oriented programming (often written as O-O) is an approach to deal with designing modular, reusable software systems[107]. It integrates the code and data of the program as an ”object” which has state(data) and behavior(code). In addition, this style provides program security by encapsulating the data that restricts direct visibility of data. The mainstream programming languages like $Java, C++$ and *Delphi* use this style of programming.

5.3 A comparison between Functional and Imperative Programming

The primary difference between functional and imperative style is based on program execution control and memory management of the program data[41].

A functional program executes an expression and results in a value[41]. The execution order of a functional program does not matter, because the result is the same anyway. On the other hand, imperative programming modifies the memory state by executing a sequence of instructions. Each instruction execution controls the instruction to be executed. Hence, the order of execution is maintained strictly, or otherwise the result may differ. Moreover, imperative languages provide greater control over the execution of a program and the memory representation of data, and are closer to the actual machine but lose the execution safety. Contrary to this, functional programming provides a higher level of abstraction and a greater level of execution safety.

A purely functional program cannot be iterative because the value of the condition of a loop never varies. By contrast, an imperative program may be iterative. Moreover, compared with imperative programming, functional style is less error-prone and more productive. On the other hand, imperative programming gives more control over the program to achieve better performance compared to the functional style[93].

5.4 Test and Analysis

In this thesis, two mathematical problems were solved using both functional and imperative model in a Visual C# 2010 compiler as a code test. This language supports both functional and imperative style programming. This test program shows that the lines of code for both problems differ for each paradigm. The functional requires a smaller number of lines of codes compared with imperative. However, functional style programming is complex compared with imperative. Moreover, the execution time required for functional programming is higher than that for the imperative style.

A large experiment has been done to determine performance and efficiency of two different programming styles, imperative and functional [83]. Two different programming languages have been used to complete this experiment. One is Scala, which combines functional and imperative programming [79]. Second is Java, which focuses on imperative shared memory programming. For this experiment, 13 experienced programmers wrote 39 Scala and 39 Java parallel program in two phases. The first one is a training phase, the second one is an industry project. The test environment and the resulting outcome of this experiment [83] are as follows-

Test environment:

Hardware: All programs were evaluated on two different machines. The first one was Intel x5677 which is a single-chip architecture with 4 cores and 2 hardware threads per core, having 48GB of memory. The operating system was RedHat Linux 6.0 Enterprise Edition[83]. The second one was Sun SPARC T3-4 which has a 4 chip NUMA architecture with 8 cores per chip and 8 hardware threads per core and 256 GB of memory. Operating system was Solaris 10[83].

Compilers: Two different compilers were used for Scala and Java. Java is a well-known mainstream imperative programming language [24] which supports parallel programming through threads. On the other hand, "Scala's programs tend to short"[80, p. 59] and "a typical Scala program should have about half the number of lines of the same program written in Java"[80, p. 59]. "Scala's functional programming constructs make it easy to build interesting things quickly from simpler parts" [80, p. 49] and "Scala is easy to get into"[80, p. 49]. "Scala's unique features promise to make parallel software development easier"[83].

Parallelization technique: Scala parallel programming technique was by using actors and Java was shared memory parallelization[83].

Results:

Efforts: Scala required more effort to finish this project compare with Java. The median is 56 hours and 43 hours for Scala and Java respectively [83]. However, Scala parallel programming technique using actors is easier than the shared-memory programming in Java according to the test programmers and 69% of the programmes said that Scala's programming composition is much easier than Java's [83].

Programming Compactness: For this experimental project, Java required more Lines of Code compared with Scala. The median Lines of Code (LOC) is 533 (mean 536) and 546 (mean 632) for Scala and Java respectively [83]. However, Scala promised that the number of lines of code would be half the number compared with Java, in practice it was not the case.

Performance: On the Intel x5677 machine, the best Scala runtime was 7 seconds and the best Java run time was 4 seconds. The median runtime of all programs amounted to 83 seconds for Scala and 98 seconds for Java, which shows that average runtime for Scala programs is shorter than for Java programs. On the Sun SPARC T3-4 machine, the best runtime for Scala was 34 seconds and for Java it was 37 seconds. Moreover, the median run time for Scala was 466 seconds and for Java 576 seconds. Again, Scala was faster[83].

6. Conclusion and Future Work

6.1 Conclusion

In this thesis various parallelization techniques that are found on modern computation systems and their incorrect behavior has been studied in detail based on the available related literature in the context of predictability and efficiency. A simple test process has been developed to test the data races on different systems. Furthermore, the interaction between hardware and software affecting correctness, predictability and efficiency in the different contexts has also been studied. A practical test has been done to measure the performance of different programming memory models. Finally, different programming paradigms of different languages have been studied and a comparison has been made for the functional and imperative styles based on the currently available published literature. Also, some mathematical problems have been solved in both paradigms as proof of literature sources that were mentioned in this comparison.

There are a large number of parallelization techniques and system available. Some of them were developed for the purposes of special use and a few of them are for general use. To measure the predictability and efficiency of all systems and find the most efficient and predictable one, access to all categories of machines would be required, in practice that was not possible in this thesis because of resource limitations. From the available literature, two test results have been analyzed. First one was for software efficiency test, and the second one was for two types of hardware (ARM and Intel x86).

In order to detect, avoid and eliminate the incorrect behavior of parallel systems, four types of problems (races, deadlocks, livelocks, starvation) were discussed in this thesis. In this discussion, reasons behind the problems and ways to avoid this kind of behavior were also included. Some tests of data races show that the number of races detected in Linux environment was much higher than the Windows environment, in particular on an Intel system.

To observe the software performance on hardware systems, the interaction between software and hardware that affects correctness, predictability and efficiency was discussed. It was observed that memory models for both software and hardware play the most significant role for compatibility between the two. So, memory models in hardware and software have been discussed in depth. Some simple programs were also developed to test the software memory model performance using different synchronization mechanisms. These tests were compiled by Java, C++ and Ada compilers. For all of the compilers, non-blocking synchronization mechanisms result in significantly better performance and provide maximum efficiency, whereas blocking synchronization mechanism provides maximum predictability and reliability.

Finally, different types of programming paradigms were also discussed in this thesis based on the currently available research. In addition, a comparison has been made between Functional and Imperative styles. The results of the functional and imperative comparison test were also analyzed based on a recent test experiment. In this thesis some solutions to mathematical problems were also developed in both functional and imperative paradigms to give some practical proof of the analysis.

6.2 Future Work

The current study about various parallelization concepts included most of the theoretical knowledge and analyzed some test results from the currently available literature. Doing some practical tests and comparing most popular multi-core processors would be an interesting opportunity to add a new dimension to current research.

The current test of incorrect behavior of different hardware and software architectures was limited to two different hardware systems and three types of operating systems. An extension of this test could be possible by adding more hardware and operating system tests while using the same code. In addition, the test in this thesis deals only with data races. Adding deadlock, livelock and starvation tests of threads would add extra features to the testing process.

To check the compatibility of software and hardware, different language memory models and their synchronization mechanisms have been tested in this thesis. The I/O operation and interrupt handling tests would be a nice addition.

Finally, a comparison between Imperative and Function programming paradigm has been done. Other combinations of comparison of all four paradigms would be of interest.

Bibliography

[1] ISO/IEC 8652:1995(E). Ada reference manual - with technical corrigendum 1 and amendment 1. Technical report, 2012.

[2] ISO/IEC 8652:2012(E). Ada reference manual 2012. *Language and Standard Libraries*, 2012.

[3] Josh Aas. Understanding the linux 2.6. 8.1 cpu scheduler. *Retrieved Oct*, 16:1–38, 2005.

[4] Martın ABADI and Examinateur Santa Cruz. Operational semantics of relaxed memory models.

[5] Sarita V Adve and Hans-J Boehm. Memory models: a case for rethinking parallel languages and hardware. *Communications of the ACM*, 53(8):90–101, 2010.

[6] Sarita V Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *computer*, 29(12):66–76, 1996.

[7] Sarita V Adve and Mark D Hill. Weak ordering—a new definition. In *ACM SIGARCH Computer Architecture News*, volume 18, pages 2–14. ACM, 1990.

[8] Sarita Vikram Adve. *Designing memory consistency models for shared-memory multiprocessors*, volume 2. University of Wisconsin–Madison, 1993.

[9] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, & tools*, volume 1009. Pearson/Addison Wesley, 2007.

[10] Sadaf R Alam, Richard F Barrett Collin B McCurdy, Philip C Roth, and Jeffrey S Vetter. Characterizing applications on the cray mta-2 multithreading architecture. In *Proc. of CUG Conf*, 2006.

[11] Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. The semantics of power and arm multiprocessor machine code. In *Proceedings of the 4th workshop on Declarative aspects of multicore programming*, pages 13–24. ACM, 2009.

[12] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Litmus: Running tests against hardware. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 41–44. Springer, 2011.

[13] Wendell Anderson, Preston Briggs, C Stephen Hellberg, Daryl W Hess, Alexei Khokhlov, Marco Lanzagorta, and Robert Rosenberg. Early experience with scientific programs on the cray mta-2. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 46. ACM, 2003.

[14] James Archibald and Jean-Loup Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems (TOCS)*, 4(4):273–298, 1986.

[15] Wikström C. Armstrong J., Williams M. and Virding R. Concurrent programming in erlang, prentice-hall. 1996. See www.erlang.org.

[16] Alan F. Babich. Proving total correctness of parallel programs. *Software Engineering, IEEE Transactions on*, (6):558–574, 1979.

[17] John Barnes. Ada rationale 2012. *Ada Core*, 2012.

[18] Pete Becker et al. Working draft, standard for programming language c++. Technical report, 2011.

[19] Dileep Bhandarkar. Risc versus cisc: a tale of two chips. *ACM SIGARCH Computer Architecture News*, 25(1):1–12, 1997.

[20] Emily Blem, Jaikrishnan Menon, and Karthikeyan Sankaralingam. A detailed analysis of contemporary arm and x86 architectures. *Report, UW-Madison Technical*, 2013.

[21] François Bodin and Michael O'Boyle. A compiler strategy for shared virtual memories. In *Languages, Compilers and Run-Time Systems for Scalable Computers*, pages 57–69. Springer, 1996.

[22] Hans-J Boehm. Position paper: Nondeterminism is unavoidable, but data races are pure evil. In *Proceedings of the 2012 ACM workshop on Relaxing synchronization for multicore and manycore scalability*, pages 9–14. ACM, 2012.

[23] Hans-J Boehm and Sarita V Adve. Foundations of the c++ concurrency memory model. In *ACM SIGPLAN Notices*, volume 43, pages 68–78. ACM, 2008.

[24] Rafael H Bordini, Lars Braubach, Mehdi Dastani, Amal El Fallah Seghrouchni, Jorge J Gomez-Sanz, Joao Leite, Gregory O'Hare, Alexander Pokahr, and Alessandro Ricci. A survey of programming languages and platforms for multi-agent systems. *Informatica (03505596)*, 30(1), 2006.

[25] Benjamin M Brosgol. A comparison of the concurrency features of ada 95 and java. *ACM SIGAda Ada Letters*, 18(6):175–192, 1998.

[26] Jon Byous. Java technology: an early history. *URL: http://java. sun. com/features/1998/05/birthday. html, Artigo pesquisado em 07 de Junho de 2002*, 1998.

[27] Richard H Carver and Kuo-Chung Tai. *Modern multithreading: implementing, testing, and debugging multithreaded Java and C++/Pthreads/Win32 programs.* John Wiley & Sons, 2005.

[28] Alan Charlesworth, Nicholas Aneshansley, Mark Haakmeester, Dan Drogichen, Gary Gilbert, Ricki Williams, and Andrew Phelps. The starfire smp interconnect. In *Supercomputing, ACM/IEEE 1997 Conference*, pages 37–37. IEEE, 1997.

[29] Nathan Chong and Samin Ishtiaq. Reasoning about the arm weakly consistent memory model. In *Proceedings of the 2008 ACM SIGPLAN workshop on Memory systems performance and correctness: held in conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08)*, pages 16–19. ACM, 2008.

[30] Michał Cierniak and Wei Li. *Unifying data and control transformations for distributed shared-memory machines*, volume 30. ACM, 1995.

[31] Edward G Coffman, Melanie Elphick, and Arie Shoshani. System deadlocks. *ACM Computing Surveys (CSUR)*, 3(2):67–78, 1971.

[32] Mozart Consortium et al. The mozart programming system. 1.4.0, July 2008. see www.mozart-oz.org.

[33] Intel Corp. Intel quickpath architecture: A new system architecture for unleashing the performance of future generations of intel multi-core microprocessors. *white paper*, 2008. http://www.intel.com/content/dam/doc/white-paper/performance-quickpath-architecture-paper.pdf.

[34] Marc de Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. Relax: An architectural framework for software recovery of hardware faults. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 497–508. ACM, 2010.

[35] Edsger W Dijkstra. Cooperating sequential processes, the origin of concurrent programming: from semaphores to remote procedure calls, 2002.

[36] Greggory D Donley and Manoj Gujral. Livelock avoidance, June 2 1998. US Patent 5,761,446.

[37] Michel Dubois and Faye A Briggs. Effects of cache coherency in multiprocessors. *Computers, IEEE Transactions on*, 100(11):1083–1099, 1982.

[38] Polina Dudnik and Michael M Swift. Condition variables and transactional memory: Problem or opportunity. In *Proceedings of the 4th ACM SIGPLAN Workshop on Transactional Computing*. Citeseer, 2009.

[39] S Tucker Taft Robert A Duff, Randall L Brukardt Erhard Ploedereder, and Pascal Leroy Edmond Schonberg. Ada 2012 reference manual.

[40] Hesham El-Rewini and Mostafa Abd-El-Barr. *Advanced computer architecture and parallel processing*, volume 42. John Wiley & Sons, 2005.

[41] Pascal Manoury Emmanuel Chailloux and Bruno Pagano. *Developing Applications With objective caml.* O'Reilly France, 2000.

[42] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Architecture support for disciplined approximate programming. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 301–312. ACM, 2012.

[43] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 37–48. ACM, 2012.

[44] Cormac Flanagan and Stephen N Freund. Fasttrack: efficient and precise dynamic race detection. In *ACM Sigplan Notices*, volume 44, pages 121–133. ACM, 2009.

[45] Michael Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, 100(9):948–960, 1972.

[46] Berry G. The esterel v5 language primer. *Ècole des Mines and INRIA*, April 1999.

[47] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 503–514. IEEE, 2011.

[48] Michael Barry Greenwald. *Non-blocking synchronization and system design.* PhD thesis, Stanford University, 1999.

[49] Part Guide. Intel® 64 and ia-32 architectures software developer's manual. 2010.

[50] Anthony Gutierrez, Ronald G Dreslinski, Thomas F Wenisch, Trevor Mudge, Ali Saidi, Chris Emmons, and Nigel Paver. Full-system analysis and characterization of interactive smartphone applications. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 81–90. IEEE, 2011.

[51] Cooper G. H. Integrating dataflow evaluation into a practical higher-order callby-value language. *Ph.D. dissertation, Brown University, Providence, Rhode Island*, May 2008.

[52] Erik Hagersten, Anders Landin, and Seif Haridi. Ddm-a cache-only memory architecture. *Computer*, 25(9):44–54, 1992.

[53] Nilsson H. Hudak P., Courtney A. and Peterson J. Arrows, robots, and functional reactive programming. *In summer School on Advanced Functional programming*, pages 159–187, 2003. Springer LNCS 2638.

[54] Paul Hyde. *Java thread programming.* Sams Pub., 1999.

[55] ISO ISO. Iec 14882: 2011 information technology programming languages — c++. *International Organization for Standardization, Geneva, Switzerland*, 27:59, 2012.

[56] Norman P Jouppi and David W Wall. *Available instruction-level parallelism for superscalar and superpipelined machines*, volume 17. ACM, 1989.

[57] M Kandemir, Alok Choudhary, Jagannathan Ramanujam, and Prithviraj Banerjee. A matrix-based approach to the global locality optimization problem. In *Parallel Architectures and Compilation Techniques, 1998. Proceedings. 1998 International Conference on*, pages 306–313. IEEE, 1998.

[58] Mahmut Kandemir, J Ramanujam, and Alok Choudhary. A compiler algorithm for optimizing locality in loop nests. In *Proceedings of the 11th international conference on Supercomputing*, pages 269–276. ACM, 1997.

[59] Gerald M. Karam and Raymond J. A. Buhr. Starvation and critical race analyzers for ada. *Software Engineering, IEEE Transactions on*, 16(8):829–843, 1990.

[60] Stephen W Keckler, William J Dally, Daniel Maskit, Nicholas P Carter, Andrew Chang, and Whay S Lee. Exploiting fine-grain thread level parallelism on the mit multi-alu processor. In *ACM SIGARCH Computer Architecture News*, volume 26, pages 306–317. IEEE Computer Society, 1998.

[61] Richard E Kessler and James L Schwarzmeier. Cray t3d: A new dimension for cray research. In *Compcon Spring'93, Digest of Papers.*, pages 176–182. IEEE, 1993.

[62] Steve Kleiman, Devang Shah, and Bart Smaalders. *Programming with threads.* Sun Soft Press, 1996.

[63] Hermann Kopetz. *Real-time systems: design principles for distributed embedded applications.* Springer, 2011.

[64] Christoforos Kozyrakis and David Patterson. Vector vs. superscalar and vliw architectures for embedded multimedia benchmarks. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 283–293. IEEE Computer Society Press, 2002.

[65] Douglas Kramer. The java platform. *White Paper, Sun Microsystems, Mountain View, CA*, 1996.

[66] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[67] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, 100(9):690–691, 1979.

[68] Henry Ledgard. *Reference Manual for the ADA Programming Language.* Springer-Verlag New York, Inc., 1983.

[69] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. *The directory-based cache coherence protocol for the DASH multiprocessor*, volume 18. ACM, 1990.

[70] Thinking Machines. Introduction to data level parallelism. *TechnicM Report*, 86, 1986.

[71] Jeremy Manson and William Pugh. Requirements for programming language memory models. In *Workshop on Concurrency and Synchronization in Java Programs, in association with PODC*. Citeseer, 2004.

[72] Jeremy Manson, William Pugh, and Sarita V Adve. *The Java memory model*, volume 40. ACM, 2005.

[73] Luc Maranget, Susmit Sarkar, and Peter Sewell. A tutorial introduction to the arm and power relaxed memory models.

[74] Close T. Frantz B. Yee K-P. Morningstar C. Shapiro J. Hardy N. Tribble E .D. Barnes D. Bornstien D. Wilcox-O'Hearn B. Stanley T. Reid K. Miller M S., Stiegler M. and Bacon D. E: Open source distributed capabilities. 2001. See www.erights.org.

[75] Halbwachs N. and Pascal R. A tutorial of lustre. January 2002.

[76] Robert HB Netzer, Sanjoy Ghosh, et al. Efficient race condition detection for shared-memory programs with post/wait synchronization. In *ICPP (2)*, pages 242–246, 1992.

[77] Michael FP O'Boyle and Peter MW Knijnenburg. Efficient parallelisation using combined loop and data transformations. In *Parallel Architectures and Compilation Techniques, 1999. Proceedings. 1999 International Conference on*, pages 283–291. IEEE, 1999.

[78] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. *ACM SIGPLAN Notices*, 38(10):167–178, 2003.

[79] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in scala.* Artima Inc, 2008.

[80] Martin Odersky, Lex Spoon, and Bill Venners. Scala. *URL: http://blog. typesafe. com/why-scala (last accessed: 2012-08-28)*, 2011.

[81] Wilfried Oed. The cray research massively parallel processor system, cray t3d. *Cray Research, Munich*, 1993.

[82] Van Hentenryck P. A gentle introduction to numerica. *Artif. Intell 103 (1-2)*, Aug 1998. pp. 209-235.

[83] Victor Pankratius, Felix Schmidt, and Gilda Garretón. Combining functional and imperative programming for multicore software: an empirical study evaluating scala and java. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 123–133. IEEE Press, 2012.

[84] W Pugh. Java memory model and thread specification revision, 2004. jsr 133.

[85] William Pugh. Fixing the java memory model. In *Proceedings of the ACM 1999 conference on Java Grande*, pages 89–98. ACM, 1999.

[86] William Pugh. The java memory model is fatally flawed. *Concurrency - Practice and Experience*, 12(6):445–455, 2000.

[87] Amaya S. History of c++. *URL: http://www.cplusplus.com/user/Albatross/*, 2012.

[88] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

[89] Lagerkvist M. Schulte C. and Tack G. Gecode: Generic constraint development environment. 2006. see www.gecode.org.

[90] Sohrab P Shah, David YM He, Jessica N Sawkins, Jeffrey C Druce, Gerald Quon, Drew Lett, Grace XY Zheng, Tao Xu, and BF Francis Ouellette. Pegasys: software for executing and integrating analyses of biological sequences. *BMC bioinformatics*, 5(1):40, 2004.

[91] Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Robert Geva, Yang Ni, and Adam Welc. Towards transactional memory semantics for c++. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 49–58. ACM, 2009.

[92] Mukesh Singhal. Deadlock detection in distributed systems. *Computer*, 22(11):37–48, 1989.

[93] David B Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Computing Surveys (CSUR)*, 30(2):123–169, 1998.

[94] Marc Snir. Distributed-memory multiprocessor. In *Encyclopedia of Parallel Computing*, pages 574–578. Springer, 2011.

[95] John A. Stankovic and Krithi Ramamritham. The spring kernel: a new paradigm for real-time operating systems. *ACM SIGOPS Operating Systems Review*, 23(3):54–71, 1989.

[96] John A Stankovic and Krithi Ramamritham. What is predictability for real-time systems? *Real-Time Systems*, 2(4):247–254, 1990.

[97] Norihisa Suzuki. *Shared Memory Multiprocessing.* MIT Press, 1992.

[98] Michael E Thomadakis. The architecture of the nehalem processor and nehalem-ep smp platforms. *Resource*, 3, 2011.

[99] Milo Tomaševic, Jelica Protic, Milo Tomasevic, and Veljko Milutinović. *Distributed shared memory: Concepts and systems*, volume 21. John Wiley & Sons, 1998.

[100] John Tromp. *How to construct an atomic variable.* Springer, 1989.

[101] Rasmus Ulfsnes. Design of a snoop filter for snoop based cache coherency protocols. 2013.

[102] Stephen H Unger. Hazards, critical races, and metastability. *Computers, IEEE Transactions on*, 44(6):754–768, 1995.

[103] Saarland University. Programming systems lab. *Alice ML Version 1.4*, 2004. see www.ps.uni-sb.de/alice.

[104] Peter Van Roy. Programming paradigms for dummies: What every programmer should know. *New Computational Paradigms for Computer Music*, 2009.

[105] N Viswanadham, Y Narahari, and Timothy L Johnson. Deadlock prevention and deadlock avoidance in flexible manufacturing systems using petri net models. *IEEE Transactions on Robotics & Automation Magazine*, 6(6):713–723, 1990.

[106] David W Wall. *Limits of instruction-level parallelism*, volume 19. ACM, 1991.

[107] Peter Wegner. Concepts and paradigms of object-oriented programming. *ACM SIGPLAN OOPS Messenger*, 1(1):7–87, 1990.

[108] Peter Wegner. The logic programming paradigm and prolog. *ACM SIGPLAN OOPS Messenger*, 1(1):7–87, 1990.

[109] Anthony Williams. *C++ Concurrency in Action.* Manning; Pearson Education, 2012.

[110] William A Wulf and C Gordon Bell. C. mmp: a multi-mini-processor. In *Proceedings of the December 5-7, 1972, fall joint computer conference, part II*, pages 765–777. ACM, 1972.

[111] Xiaodong Zhang and Yong Yan. Comparative modeling and evaluation of cc-numa and coma on hierarchical ring architectures. *Parallel and Distributed Systems, IEEE Transactions on*, 6(12):1316–1331, 1995.

[112] Pin Zhou, Radu Teodorescu, and Yuanyuan Zhou. Hard: Hardware-assisted lockset-based race detection. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 121–132. IEEE, 2007.

[113] Dieter Zöbel. The deadlock problem: A classifying bibliography. *ACM SIGOPS Operating Systems Review*, 17(4):6–15, 1983.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature