SURVIVING THE DESIGN OF
MICROPROCESSOR
AND
MULTIMICROPROCESSOR
SYSTEMS
LESSONS LEARNED

Veljko Milutinović

Foreword by Michael Flynn
# Table of Contents

PROLOGUE ........................................................................................................................................... 8  
Foreword ............................................................................................................................................... 10  
Preface .................................................................................................................................................. 11  
Acknowledgments ............................................................................................................................... 15  

FACTS OF IMPORTANCE .................................................................................................................. 17  
Microprocessor Systems ...................................................................................................................... 1  
1. Basic Issues ....................................................................................................................................... 1  
   1.1. Pentium......................................................................................................................................... 6  
   1.1.1. Cache and Cache Hierarchy ................................................................................................. 9  
   1.1.2. Instruction-Level Parallelism .............................................................................................. 10  
   1.1.3. Branch Prediction ................................................................................................................. 10  
   1.1.4. Input/Output ......................................................................................................................... 11  
   1.1.5. Multithreading ..................................................................................................................... 12  
   1.1.6. Support for Shared Memory Multiprocessing .................................................................... 12  
   1.1.7. Support for Distributed Shared Memory ........................................................................... 15  
   1.2. Pentium MMX ............................................................................................................................ 16  
   1.3. Pentium Pro ............................................................................................................................... 16  
   1.4. Pentium II .................................................................................................................................. 18  
2. Advanced Issues .................................................................................................................................. 19  
3. About the Research of the Author and His Associates ..................................................................... 23  

ISSUES OF IMPORTANCE .................................................................................................................... 25  
Cache and Cache Hierarchy .................................................................................................................... 27  
1. Basic Issues ....................................................................................................................................... 27  
   1.1. Fully-associative cache .......................................................................................................... 28  
   1.2. Set-associative cache ............................................................................................................. 28  
   1.3. Direct-mapped cache .............................................................................................................. 29  
2. Advanced Issues .................................................................................................................................. 29  
3. About the Research of the Author and His Associates ..................................................................... 32  
Instruction-Level Parallelism .................................................................................................................. 34  
1. Basic Issues ....................................................................................................................................... 34  
   1.1. Example: MIPS R10000 ....................................................................................................... 40  
   1.2. Example: DEC Alpha 21164 ............................................................................................... 42  
   1.3. Example: DEC Alpha 21264 ............................................................................................... 43  
2. Advanced Issues .................................................................................................................................. 43  
3. About the Research of the Author and His Associates ..................................................................... 48  
Branch Prediction Strategies ................................................................................................................... 50  
1. Basic Issues ....................................................................................................................................... 50  
   1.1. Hardware BPS ....................................................................................................................... 51  
   1.2. Software BPS ......................................................................................................................... 60  
   1.3. Hybrid BPS .............................................................................................................................. 61  
   1.3.1. Predicated Instructions .................................................................................................... 61  
   1.3.2. Speculative Instructions .................................................................................................. 62
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1. Types of I/O Devices</td>
<td>71</td>
</tr>
<tr>
<td>1.2. Types of I/O Organization</td>
<td>73</td>
</tr>
<tr>
<td>1.3. Storage System Design for Uniprocessors</td>
<td>73</td>
</tr>
<tr>
<td>1.4. Storage System Design for Multiprocessor and Multicomputer Systems</td>
<td>76</td>
</tr>
<tr>
<td>2. Advanced Issues</td>
<td>78</td>
</tr>
<tr>
<td>2.1. The Disk Cache Disk</td>
<td>78</td>
</tr>
<tr>
<td>2.2. The Polling Watchdog Mechanism</td>
<td>79</td>
</tr>
<tr>
<td>3. About the Research of the Author and His Associates</td>
<td>79</td>
</tr>
<tr>
<td>1. Basic Issues</td>
<td>80</td>
</tr>
<tr>
<td>1.1. Coarse Grained Multithreading</td>
<td>80</td>
</tr>
<tr>
<td>1.2. Fine Grained Multithreading</td>
<td>82</td>
</tr>
<tr>
<td>2. Advanced Issues</td>
<td>84</td>
</tr>
<tr>
<td>3. About the Research of the Author and His Associates</td>
<td>85</td>
</tr>
<tr>
<td>1.1. Snoopy Protocols</td>
<td>88</td>
</tr>
<tr>
<td>1.1.1. Write-Invalidate Protocols</td>
<td>89</td>
</tr>
<tr>
<td>1.1.2. Write-Update Protocols</td>
<td>90</td>
</tr>
<tr>
<td>1.1.3. MOESI Protocol</td>
<td>90</td>
</tr>
<tr>
<td>1.1.4. MESI Protocol</td>
<td>91</td>
</tr>
<tr>
<td>1.2. Directory protocols</td>
<td>91</td>
</tr>
<tr>
<td>1.2.1. Full-Map Directory Protocols</td>
<td>93</td>
</tr>
<tr>
<td>1.2.2. Limited Directory Protocols</td>
<td>94</td>
</tr>
<tr>
<td>1.2.2.1. The Dir(i)NB Protocol</td>
<td>95</td>
</tr>
<tr>
<td>1.2.2.2. The Dir(i)B Protocol</td>
<td>95</td>
</tr>
<tr>
<td>1.2.3. Chained Directory Protocols</td>
<td>96</td>
</tr>
<tr>
<td>2. Advanced Issues</td>
<td>97</td>
</tr>
<tr>
<td>2.1. Extended Pointer Schemes</td>
<td>97</td>
</tr>
<tr>
<td>2.2. The University of Pisa Protocols</td>
<td>99</td>
</tr>
<tr>
<td>3. About the Research of the Author and His Associates</td>
<td>100</td>
</tr>
<tr>
<td>Distributed Shared Memory</td>
<td>101</td>
</tr>
<tr>
<td>1. Basic Issues</td>
<td>101</td>
</tr>
<tr>
<td>1.1. The Mechanisms of a DSM System and Their Implementation</td>
<td>102</td>
</tr>
<tr>
<td>1.2. The Internal Organization of Shared Data</td>
<td>103</td>
</tr>
<tr>
<td>1.3. The Granularity of Consistency Maintenance</td>
<td>103</td>
</tr>
<tr>
<td>1.4. The Access Algorithms of a DSM System</td>
<td>104</td>
</tr>
<tr>
<td>1.5. The Property Management of a DSM System</td>
<td>105</td>
</tr>
<tr>
<td>1.6. The Cache Consistency Protocols of a DSM System</td>
<td>105</td>
</tr>
<tr>
<td>1.7. The Memory Consistency Protocols of a DSM System</td>
<td>106</td>
</tr>
<tr>
<td>1.7.1. Release Consistency</td>
<td>108</td>
</tr>
<tr>
<td>1.7.2. Lazy Release Consistency</td>
<td>109</td>
</tr>
<tr>
<td>1.7.3. Entry Consistency</td>
<td>110</td>
</tr>
<tr>
<td>1.7.4. Automatic Update Release Consistency</td>
<td>111</td>
</tr>
<tr>
<td>1.7.5. Scope Consistency</td>
<td>113</td>
</tr>
<tr>
<td>1.8. A Special Case: Barriers and Their Treatment</td>
<td>114</td>
</tr>
</tbody>
</table>
PROLOGUE
Elements of this prologue are:

(a) Foreword,
(b) Preface, and
(c) Acknowledgments.
Foreword

There are several different styles in technical texts and monographs. The most familiar is the review style of the basic textbook. This style simply considers the technical literature and re-presents the data in a more orderly or useful way. Another style appears most commonly in monographs. This reviews either a particular aspect of a technology or all technical aspects of a single complex engineering system. A third style, represented by this book, is an integration of the first two styles, coupled with a personal reconciliation of important trends and movements in technology.

The author, Professor Milutinovic, has been one of the most productive leaders in the computer architecture field. Few readers will not have encountered his name on an array of publications involving the important issues of the day. His publications and books span almost all areas of computer architecture and computer engineering. It would be easy, then, but inaccurate, to imagine this work as a restatement of his earlier ideas. This book is different, as it uniquely synthesizes Professor Milutinovic's thinking on the important issues in computer architecture.

The issues themselves are presented quite concisely: cache, instruction level parallelism, the I/O bottleneck, multithreading, and multiprocessors. These are currently the principal research areas in computer architecture. Each one of these topics is presented in a crisp way, highlighting the important issues in the field together with Professor Milutinovic's special viewpoints on these issues, closing each section with a statement about his own group's research in this area. This statement of issues is coupled with three important case studies of fundamentally different computer systems implementations. The case studies use details of actual engineering implementations to help synthesize the issues presented. The result is a necessarily somewhat eclectic, personal statement by one of the leaders of the field about the important issues that face us at this time.

This work should prove invaluable to the serious student.

Michael J. Flynn
Preface

Design of microprocessor and/or multimicroprocessor systems represents a continuous struggle; success (if achieved) lasts infinitesimally long and disappears forever, unless a new struggle (with unpredictable results) starts immediately. In other words, it is a continuous survival process, which is the main motto of this book.

This book is about survival of those who have contributed to the state of the art in the rapidly changing field of microprocessing and multimicroprocessing on a single chip, and about the concepts that have to find their way into the next generation microprocessors and multimicroprocessors on a chip, in order to enable these products to stay on the competitive edge.

This book is based on the assumption that the ultimate goal of the single chip design is to have an entire distributed shared memory system on a single silicon die, together with numerous specialized accelerators, including the complex ones of SIMD and/or MISD type. Consequently, the book concentrates on the major problems to be solved on the way to this ultimate goal (distributed shared memory on a single chip), and summarizes the author’s experiences which led to such a conclusion (in other words, the problem is how to “invest one billion transistors” on a single chip).

This book is also about the microprocessor and multimicroprocessor based designs of the author himself, and about the lessons that he has learned through his own professional survival process which lasts for about two decades now; concepts from microprocessor and multimicroprocessor boards of the past represent potential solutions for the microprocessor and multimicroprocessor chips of the future, and (which is more important) represent the ground for the author’s belief that the ultimate goal is to have an entire distributed shared memory on a single chip, together with numerous specialized accelerators.

At first, distributed shared memory on a single chip may sound as a contradiction; however, it is not. As the dimensions of chips become larger, their full utilization can be obtained only with multimicroprocessor architectures. After the number of microprocessors reaches 16, the SMP architecture is no longer a viable solution since bus becomes a bottleneck; consequently, designers will be forced to move to the distributed shared memory paradigm (implemented in hardware, or partially in hardware and partially in software).

In this book, the issues of importance for current on-board microprocessor and multimicroprocessor based designs, as well as for future on-chip microprocessor and multimicroprocessor designs, have been divided into eight different topics. The first one is about the general microprocessor architecture, and the remaining seven are about seven different problem areas
of importance for the “ultimate goal:” distributed shared memory on a single chip, together with numerous specialized accelerators. Each of the topics is further subdivided into three different sections:

a) the first one on the basics (traditional body of knowledge),

b) the second one on the advances (state of the art information), and

c) the third one on the efforts of the author and his associates (a brief research report).

After long discussions with the more experienced colleagues (see the list in the acknowledgment section), and the more enthusiastic students (they always have excellent comments), the major topics have been selected, as follows:

a) Microprocessor systems on a chip,
b) Cache and cache hierarchy,
c) Instruction level parallelism,
d) Branch prediction strategies,
e) Input/output bottleneck,
f) Multithreaded processing,
g) Shared memory multiprocessing systems, and
h) Distributed shared memory systems.

Topics related to uniprocessing are of importance for microprocessor based designs of today and the microprocessor on-chip designs of immediate future. Topics related to multiprocessing are of importance for multimicroprocessor based designs of today and the multimicroprocessor on-chip designs of the not so immediate future.

As already indicated, the author is one of the believers in the prediction that future on-chip machines, even if not of the multimicroprocessor or multicomputer type, will include strong support for multiprocessing (single logical address space) and multicomputing (multiple logical address spaces). Consequently, as far as multiprocessing and multicomputing are concerned, only the issues of importance for future on-chip machines have been selected.

This book also includes a prologue section, which explains the roots of the idea behind it: combining synergistically the general body of knowledge and the particular experiences of an individual who has survived several pioneering design efforts of which some were relatively successful commercially.

Finally, this book also includes an epilogue section, with three case studies, on three multimicroprocessor based designs. The author was deeply engaged in all three designs. Each project, in the field which is the subject of this book, includes three major types of activities:

a) envisioning of the strategy (project directions and milestones),
b) consulting on the tactics (product architecture and organization), and
c) engaging in the battle (design and almost exhaustive testing at all logical levels, until the product is ready for production).

The first case study is on a multimicroprocessor implementation of a data modem receiver for high frequency (HF) radio. This design has often been quoted as the world’s first multimicroprocessor based high frequency data modem. The work was done in 70s; however, the interest in the results reincarnated both in 80s (due to technology impacts which enabled miniaturization) and in 90s (due to application impacts of wireless communications). The au-
author, absolutely alone, took all three activities (roles) defined above (one technician only helped with wire-wrapping, using the list prepared by the author), and brought the prototype to a performance success (the HF modem receiver provided better performance on a real HF medium, compared to the chosen competitor product), and to a market success (after the preparation for production was done by others: wire-wrap boards and older-date components were turned, by others, into the printed-circuit boards and newer-date components) in less than two years (possible only with the enthusiasm of a novice). See the references in the epilogue section, as a pointer to details (these references are not the earliest ones, but the ones which convey most information of interest for this book).

The second case study is on a multimicroprocessor implementation of a GaAs systolic array for Gram-Schmidt orthogonalization (GSO). This design has been often quoted as the world’s first GaAs systolic array. The work was done in 80s; the interest in the results did not reincarnate in 90s. The author took only the first two roles; the third one was taken by the others (see the acknowledgment section), but never really completed, since the project was canceled before its full completion, due to enormous cost (total of 8192 microprocessor nodes, each one running at the speed of 200 MHz). See the reference in the epilogue section, as a pointer to details (these references are not the earliest ones, but the ones which convey most information of interest for this book).

The third case study is on the implementation of a board (and the preceding research) which enables a personal computer (PC) to become a node in distributed shared memory (DSM) multiprocessor of the reflective memory system (RMS) type. This design has been often quoted as the world’s first DSM plug-in board for PC technology (some efforts with larger visibility came later; one of them, with probably the highest visibility [Gillett96], as an indirect consequence of this one). The work was done in 90s. The author took only the first role and was responsible for the project (details were taken care of by graduate students); fortunately, the project was completed successfully (and what is more important for a professor, papers were published with timestamps prior to those of the competition). See the references in the epilogue section, as a pointer to details (these references are not the earliest ones, but the ones which convey most information of interest for this book).

All three case studies have been specified with enough details, so the interested readers (typically undergraduate students) can redesign the same product using a state of the art technology. Throughout the book, the concepts/ideas and lessons/experiences are in the foreground; the technology characteristics and implementation details are in the background, and can be modified (updated) by the reader, if so desired. This book:

Milutinovic, V.,
“Surviving the Design of Microprocessor and Multimicroprocessor Systems: Lessons Learned.”
IEEE Computer Society Press, Los Alamitos, California, USA, 1998,
is nicely complemented with other books of the same author, by the same publisher. One of them is:

Milutinovic, V.,
“Surviving the Design of a 200 MHz RISC Microprocessor: Lessons Learned,”

The above two books together (in various forms) have been used for about a decade now, by the author himself, as the coursework material for two undergraduate courses that he has
taught at numerous universities worldwide. Other books are on the more advanced topics, and have been used in graduate teaching on the follow up subjects:

Ekmeć, I., Tartalja, I., Milutinovic, V.,
“Tutorial on Heterogeneous Processing: Concepts and Systems,”
IEEE Computer Society Press, Los Alamitos, California, USA, 1998
(currently in final stages of preparation; expected to be out by the time this book is published).

Protic, J., Tomasevic, M., Milutinovic, V.,
“Tutorial on Distributed Shared Memory: Concepts and Systems,”
IEEE Computer Society Press, Los Alamitos, California, USA, 1997
(currently in final stages of production; will be out definitely before this book).

Tartalja, I., Milutinovic, V.,
“Tutorial on Cache Consistency Problem in Shared Memory Multiprocessors: Software Solutions,”
IEEE Computer Society Press, Los Alamitos, California, USA, 1996.

Tomasevic, M., Milutinovic, V.,
“Tutorial on Cache Coherence Problem in Shared Memory Multiprocessors: Hardware Solutions,”

In conclusion, this book covers only the issues which are, in the opinion of the author, of strong interest for future design of microprocessors and multimicroprocessors on the chip, or the issues which have impacted his opinion about future trends in microprocessor and multimicroprocessor design. These issues have been treated selectively, with more attention paid to topics which are believed to be of more importance. This explains the difference in the breadth and depth of coverage throughout the book.

Also, the selected issues have been treated at the various levels of detail. This was done intentionally, in order to create room for creativeness of the students. Typical homework requires that the missing details be completed, and the inventiveness with which the students fulfill the requirement is sometimes unbelievable (the best student projects can be found on the author’s coursework web page). Consequently, one of the major educational goals of this book, if not the major one, is to help create the inventiveness among the students. Suggestions on how to achieve this goal more efficiently are more than welcome.

Finally, a few words on the educational approach used in this book. It is well known that “one picture is worth of one thousand words.” Consequently, the stress in this book has been placed on presentation methodologies in general, as well as figures and figure captions, in particular. All necessary explanations have been put into the figures and figure captions. The main body of the text has been kept to its minimum—only the issues of interest for the global understanding of the topic and/or the thoughts on experiences gained and lessons learned. Consequently, students claim that this book is fast to read and easy to comprehend.

Veljko Milutinović
vm@etf.bg.ac.yu
http://ubbg.etf.bg.ac.yu/~vm/
Acknowledgments

This book would not be possible without the help of numerous individuals; some of them helped the author to master the knowledge and to gather the experiences necessary to write this book; others have helped to create the structure or to improve the details. Since the book of this sort would not be possible if the author did not take place in the three large projects defined in the preface, the acknowledgment will start from those involved in the same projects, directly or indirectly.

In relation to the first project (MISD for DFT), the author is thankful to professor Georgije Lukatela from whom he has learned a lot, and also to his colleagues who worked on the similar problems in the same or other companies (Radenko Paunovic, Slobodan Nedic, Milenko Ostojic, David Monsen, Philip Leifer, and John Harris).

In relation to the second project (SIMD for GSO), the author is thankful to professor Jose Fortes who had an important role in the project, and also to his colleagues who were involved with the project in the same team or within the sponsor team (David Fura, Gih Jung Jung, Salim Lakhani, Ronald Andrews, Wayne Moyers, and Walter Helbig).

In relation to the third project (MIMD for RMS), the author is thankful to professor Milo Tomasevic who has contributed significantly, and also to colleagues who were involved in the same project, within his own team or within the sponsor team (Savo Savic, Milan Jovanovic, Aleksandra Grujic, Ziya Aral, Ilya Gartner, and Mark Natale).

The list of colleagues/professors who have helped about the overall structure and contents of the book, through formal or informal discussions, and direct or indirect advice, on one or more elements of the book, during the seminars presented at their universities or during the friendly chatting between conference sessions, or have influenced the author in other ways, includes but is not limited to the following individuals: Tihomir Aleksic, Vidojko Ciric, Jack Dennis, Hank Dietz, Jovan Djordjevic, Jozo Dujmovic, Milos Ercegovac, Michael Flynn, Borko Furht, Jean-Luc Gaudiot, Anoop Gupta, John Hennessy, Kai Hwang, Liviu Iftode, Emil Jovanov, Zoran Jovanovic, Borivoj Lasic, Bozidar Levi, Kai Li, Oskar Mencer, Srdjan Mitrovic, Trevor Mudge, Vojin Oklobdzija, Milutin Ostojic, Yale Patt, Branislava Perunicic, Antonio Prete, Bozidar Radenkovic, Jasna Ristic, Eduardo Sanchez, Richard Schwartz, H.J. Siegel, Alan Smith, Ljubisa Stankovic, Dusan Starcevic, Per Stenstrom, Daniel Tabak, Igor Tartalja, Jacques Tiberghien, Mateo Valero, Dusan Velasevic, and Dejan Zivkovic.

The list also includes numerous individuals from industry worldwide who have provided support or have helped clarify details on a number of issues of importance: Tom Brumett,

Students have helped a lot to maximize the overall educational quality of the book. Several generations of students have used the book before it went to press. Their comments and suggestions were of extreme value. Those who deserve special credit are: Goran Davidovic, Zoran Dimitrijevic, Vladan Dugaric, Milan Jovanovic, Petar Lazarevic, Davor Magdic, Darko Marinov, Aleksandar Milenkovic, Milena Petrovic, and Milos Prvulovic, and Dejan Raskovic. Also, Jovanka Ciric, Boris Markovic, Zvezdan Petkovic, Jelica Protic, Milo Tomasevic, and Slavisa Zigic.

Finally, the role of the family was crucial. Wife Dragana took on the management of teenagers (Dusan, Milan, Goran) so the father could write the book; she also has read carefully the most critical parts of the book, and has helped improve the wording. Father Milan, mother Simonida, and uncle Ratoljub have helped with their life experiences.

Veljko Milutinović
vm@etf.bg.ac.yu
http://ubbg.etf.bg.ac.yu/~vm/
FACTS OF IMPORTANCE
As already indicated, this author believes that the solution for a “one billion transistor” chip of the future is a complete distributed shared memory machine on a single chip, together with a number of specialized on-chip accelerators.

The eight sections to follow cover (a) essential facts about the current microprocessor architectures and (b) the seven major problem areas, to be resolved on the way to the final goal stated above.
Microprocessor Systems

This chapter includes three sections. The section on basic issues covers the past trends in microprocessor technology and characteristics of some contemporary microprocessors machines from the workstation market, namely Intel Pentium, Pentium MMX, Pentium Pro, and Pentium II, as the main driving forces of the today’s personal computing market. The section on advanced issues covers future trends in state of the art microprocessors. The section on the research of the author and his associates concentrates on design efforts using hardware description languages.

1. Basic Issues

It is interesting to compare current Intel CISC type products (which drive the personal computer market today) with the RISC products of Intel and of the other companies. At the time being, DEC Alpha family includes three representatives: 21064, 21164, and 21264. The PowerPC family was initially devised by IBM, Motorola, and Apple, and includes a series of microprocessors starting at PPC 601 (IBM name) or MPC 601 (Motorola name); the follow-up projects have been referred to as 602, 603, 604, and 620. The SUN Sparc family follows two lines: V.8 (32-bit machines) and V.9 (64-bit machines). The MIPS Rx000 series started with R2000/3000, followed by R4000, R6000, R8000, and R10000. Intel has introduced two different RISC machines: i960 and i860 (Pentium II has a number of RISC features included at the microarchitecture level). The “traditional” Motorola RISC line includes MC88100 and MC88110. The Hewlett-Packard series of RISC machines is referred to as PA (Precision Architecture).

All comparative data for microprocessors that are sitting on our desks for years now have been presented in the form of tables (manufacturer names and Internet URLs are given in Figures MPSU1a and MPSU1b). One has to be aware of the past, before starting to look into the future.

<table>
<thead>
<tr>
<th>Microprocessor</th>
<th>Company</th>
</tr>
</thead>
<tbody>
<tr>
<td>PowerPC 601</td>
<td>IBM, Motorola</td>
</tr>
<tr>
<td>PowerPC 604e</td>
<td>IBM, Motorola</td>
</tr>
<tr>
<td>PowerPC 620*</td>
<td>IBM, Motorola</td>
</tr>
<tr>
<td>Alpha 21064*</td>
<td>Digital Equipment Corporation (DEC)</td>
</tr>
<tr>
<td>Alpha 21164*</td>
<td>Digital Equipment Corporation (DEC)</td>
</tr>
<tr>
<td>Alpha 21264*</td>
<td>Digital Equipment Corporation (DEC)</td>
</tr>
<tr>
<td>SuperSPARC</td>
<td>Sun Microelectronics</td>
</tr>
</tbody>
</table>
UltraSPARC-I* Sun Microelectronics
UltraSPARC-II* Sun Microelectronics
R4400* MIPS Technologies
R10000* MIPS Technologies
PA7100 Hewlett-Packard
PA8000* Hewlett-Packard
PA8500* Hewlett-Packard
MC88110 Motorola
AMD K6 Advanced Micro Devices (AMD)
i860 XP Intel
Pentium II Intel

**Figure MPSU1a:** Microprocessors and their primary manufacturers (source: [Prvulovic97])

**Legend:**
* 64-bit microprocessors, all others are 32-bit microprocessors.

**Comment:**
Note that the number of companies manufacturing general purpose microprocessors is relatively small.

<table>
<thead>
<tr>
<th>Company</th>
<th>Internet URL of microprocessor family home page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Motorola</td>
<td><a href="http://www.mot.com/SPS/PowerPC">http://www.mot.com/SPS/PowerPC</a></td>
</tr>
<tr>
<td>DEC</td>
<td><a href="http://www.europe.digital.com/semiconductor/alpha.htm">http://www.europe.digital.com/semiconductor/alpha.htm</a></td>
</tr>
<tr>
<td>Sun</td>
<td><a href="http://www.sun.com/sparc">http://www.sun.com/sparc</a></td>
</tr>
<tr>
<td>Hewlett-Packard</td>
<td><a href="http://hpce920.external.hp.com/computing/framed/technology/micropro">http://hpce920.external.hp.com/computing/framed/technology/micropro</a></td>
</tr>
<tr>
<td>AMD</td>
<td><a href="http://www.amd.com/K6">http://www.amd.com/K6</a></td>
</tr>
<tr>
<td>Intel</td>
<td><a href="http://www.intel.com/english/PentiumII/zdn.htm">http://www.intel.com/english/PentiumII/zdn.htm</a></td>
</tr>
</tbody>
</table>

**Figure MPSU1b:** Microprocessor family home pages (source: [Prvulovic97])

**Comment:**
Listed URL addresses are unlikely to change; however, their contents do change over time.

Table 1 (in Figure MPSU2) compares the chip technology. Table 2 (in Figure MPSU3) compares selected architectural issues. Table 3 (in Figure MPSU4) compares the instruction level parallelism and the count of the related processing units. Table 4 (in Figure MPSU5) is related to cache memory, and Table 5 (in Figure MPSU6) includes miscellaneous issues, like TLB* structures and branch prediction solutions.

<table>
<thead>
<tr>
<th>Microprocessor</th>
<th>Technology</th>
<th>Transistors</th>
<th>Frequency [MHz]</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>PowerPC 601</td>
<td>0.6 μm, 4 L, CMOS</td>
<td>2,800,000</td>
<td>80</td>
<td>304 PGA</td>
</tr>
<tr>
<td>PowerPC 604e</td>
<td>0.35 μm, 5 L, CMOS</td>
<td>5,100,000</td>
<td>225</td>
<td>255 BGA</td>
</tr>
<tr>
<td>PowerPC 620</td>
<td>0.35 μm, 4 L, CMOS</td>
<td>7,000,000</td>
<td>200</td>
<td>625 BGA</td>
</tr>
<tr>
<td>Alpha 21064</td>
<td>0.7 μm, 3 L, CMOS</td>
<td>1,680,000</td>
<td>300</td>
<td>431 PGA</td>
</tr>
<tr>
<td>Alpha 21164</td>
<td>0.35 μm, 4 L, CMOS</td>
<td>9,300,000</td>
<td>500</td>
<td>499 PGA</td>
</tr>
<tr>
<td>Alpha 21264</td>
<td>0.35 μm, 6 L, CMOS</td>
<td>15,200,000</td>
<td>500</td>
<td>588 PGA</td>
</tr>
<tr>
<td>SuperSPARC</td>
<td>0.8 μm, 3 L, CMOS</td>
<td>3,100,000</td>
<td>60</td>
<td>293 PGA</td>
</tr>
<tr>
<td>UltraSPARC-I</td>
<td>0.4 μm, 4 L, CMOS</td>
<td>5,200,000</td>
<td>200</td>
<td>521 BGA</td>
</tr>
<tr>
<td>UltraSPARC-II</td>
<td>0.35 μm, 5 L, CMOS</td>
<td>5,400,000</td>
<td>250</td>
<td>521 BGA</td>
</tr>
<tr>
<td>R4400</td>
<td>0.6 μm, 2 L, CMOS</td>
<td>2,200,000</td>
<td>150</td>
<td>447 PGA</td>
</tr>
<tr>
<td>R10000</td>
<td>0.35 μm, 4 L, CMOS</td>
<td>6,700,000</td>
<td>200</td>
<td>599 LGA</td>
</tr>
<tr>
<td>PA7100</td>
<td>0.8 μm, 3 L, CMOS</td>
<td>850,000</td>
<td>100</td>
<td>504 PGA</td>
</tr>
<tr>
<td>PA8000</td>
<td>0.35 μm, 5 L, CMOS</td>
<td>3,800,000</td>
<td>180</td>
<td>1085 LGA</td>
</tr>
</tbody>
</table>

* TLB = Translation Lookaside Buffer
Figure MPSU2: Microprocessor technology (sources: [Prvulovic97], [Stojanovic95])

Legend:

- x L—x-layer metal (x = 2, 3, 4, 5);
- PGA—pin grid array;
- BGA—ball grid array;
- LGA—land grid array;
- SEC—single edge contact.

Comment:

Actually, this figure shows the strong and the not so strong sides of different manufacturers, as well as their basic development strategies. Some manufacturers generate large transistor count chips which are not very fast, and vice versa. Also, the pin count of chip packages differs, as well as the number of on-chip levels of metal, or the minimal feature size.

### Table: Microprocessor Technology

<table>
<thead>
<tr>
<th>Microprocessor</th>
<th>IU registers</th>
<th>FPU registers</th>
<th>VA</th>
<th>PA</th>
<th>EC Dbus</th>
<th>SYS Dbus</th>
</tr>
</thead>
<tbody>
<tr>
<td>PowerPC 601</td>
<td>32×32</td>
<td>32×64</td>
<td>52</td>
<td>32</td>
<td>none</td>
<td>64</td>
</tr>
<tr>
<td>PowerPC 604e</td>
<td>32×32+RB(12)</td>
<td>32×64+RB(8)</td>
<td>52</td>
<td>32</td>
<td>none</td>
<td>64</td>
</tr>
<tr>
<td>PowerPC 620</td>
<td>32×64+RB(8)</td>
<td>32×64+RB(8)</td>
<td>80</td>
<td>128</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alpha 21064</td>
<td>32×64</td>
<td>32×64</td>
<td>43</td>
<td>128</td>
<td></td>
<td>64</td>
</tr>
<tr>
<td>Alpha 21164</td>
<td>32×64+RB(8)</td>
<td>32×64</td>
<td>40</td>
<td>128</td>
<td></td>
<td>64</td>
</tr>
<tr>
<td>Alpha 21264</td>
<td>32×64+RB(48)</td>
<td>32×64+RB(40)</td>
<td>?</td>
<td>128</td>
<td></td>
<td>64</td>
</tr>
<tr>
<td>SuperSPARC</td>
<td>136×32</td>
<td>32×32*</td>
<td>32</td>
<td>64</td>
<td></td>
<td>64</td>
</tr>
<tr>
<td>UltraSPARC-I</td>
<td>136×64</td>
<td>32×64</td>
<td>44</td>
<td>128</td>
<td></td>
<td>64</td>
</tr>
<tr>
<td>UltraSPARC-II</td>
<td>136×64</td>
<td>32×64</td>
<td>44</td>
<td>128</td>
<td></td>
<td>64</td>
</tr>
<tr>
<td>R4400</td>
<td>32×64</td>
<td>32×64</td>
<td>40</td>
<td>128</td>
<td></td>
<td>64</td>
</tr>
<tr>
<td>R10000</td>
<td>32×64+RB(32)</td>
<td>32×64+RB(32)</td>
<td>44</td>
<td>128</td>
<td></td>
<td>64</td>
</tr>
<tr>
<td>PA7100</td>
<td>32×32</td>
<td>32×64</td>
<td>64</td>
<td>64</td>
<td></td>
<td>?</td>
</tr>
<tr>
<td>PA8000</td>
<td>32×64+RB(56)</td>
<td>32×64</td>
<td>48</td>
<td>64</td>
<td></td>
<td>64</td>
</tr>
<tr>
<td>PA8500</td>
<td>32×64+RB(56)</td>
<td>32×64</td>
<td>48</td>
<td>64</td>
<td></td>
<td>64</td>
</tr>
<tr>
<td>MC88110</td>
<td>32×32</td>
<td>32×80</td>
<td>32</td>
<td>64</td>
<td>none</td>
<td>?</td>
</tr>
<tr>
<td>AMD K6</td>
<td>8×32+RB(40)</td>
<td>8×80</td>
<td>48</td>
<td>64</td>
<td></td>
<td>64</td>
</tr>
<tr>
<td>i860 XP</td>
<td>32×32</td>
<td>32×32*</td>
<td>32</td>
<td>64</td>
<td>none</td>
<td>?</td>
</tr>
<tr>
<td>Pentium II</td>
<td>?</td>
<td>8×80</td>
<td>48</td>
<td>64</td>
<td></td>
<td>64</td>
</tr>
</tbody>
</table>

Figure MPSU3: Microprocessor architecture (sources: [Prvulovic97], [Stojanovic95])

Legend:

- IU—integer unit;
- FPU—floating point unit;
- VA—virtual address [bits];
- PA—physical address [bits];
- EC Dbus—external cache data bus width [bits];
- SYS Dbus—system bus width [bits];
- RB—rename buffer [size expressed in the number of registers];

Comment:

The number of integer unit registers shows the impact of initial RISC research, on the designers of a specific microprocessor. Only SUN Microsystems have opted for the extremely large register file, which is a sign of a direct or indirect impact of Berkeley RISC research. In the
other cases, smaller register files indicate the preferences corresponding directly or indirectly to the Stanford MIPS research.

<table>
<thead>
<tr>
<th>Microprocessor</th>
<th>ILP issue</th>
<th>LSU units</th>
<th>IU units</th>
<th>FPU units</th>
<th>GU units</th>
</tr>
</thead>
<tbody>
<tr>
<td>PowerPC 601</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>PowerPC 604e</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>PowerPC 620</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Alpha 21064</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Alpha 21164</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Alpha 21264</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>SuperSPARC</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>UltraSPARC-I</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>UltraSPARC-II</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>R4400</td>
<td>1*</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>R10000</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>PA7100</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>PA8000</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>PA8500</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>MC88110</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>AMD K6</td>
<td>6**</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1***</td>
</tr>
<tr>
<td>i860 XP</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Pentium II</td>
<td>5**</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

Figure MPSU4: Microprocessor ILP features (sources: [Prvulovic97], [Stojanovic95])

Legend:
ILP = instruction level parallelism;
LSU = load/store or address calculation unit;
IU = integer unit;
FPU = floating point unit;
GU = graphics unit;
* Superpipelined;
** RISC instructions, one or more of them are needed to emulate an 80x86 instruction;
*** MMX (multimedia extensions) unit.

Comment:
One can see that the total number of units (integer, floating point, and graphics) is always larger than or equal to the issue width. Intel and Motorola had a head start on the hardware acceleration of graphics function, which is the strategy adopted later by the follow up machines of most other manufacturers. Zero in the LSU column indicates no independent load/store units.

<table>
<thead>
<tr>
<th>Microprocessor</th>
<th>L1 Icache, KB</th>
<th>L1 Dcache, KB</th>
<th>L2 cache, KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>PowerPC 601</td>
<td>32, 8WSA, UNI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PowerPC 604e</td>
<td>32, 4WSA</td>
<td>32, 4WSA</td>
<td></td>
</tr>
<tr>
<td>PowerPC 620</td>
<td>32, 8WSA</td>
<td>32, 8WSA</td>
<td></td>
</tr>
<tr>
<td>Alpha 21064</td>
<td>8, DIR</td>
<td>8, DIR</td>
<td></td>
</tr>
<tr>
<td>Alpha 21164</td>
<td>8, DIR</td>
<td>8, DIR</td>
<td>96, 3WSA*</td>
</tr>
<tr>
<td>Alpha 21264</td>
<td>64, 2WSA</td>
<td>64, DIR</td>
<td></td>
</tr>
<tr>
<td>SuperSPARC</td>
<td>20, 5WSA</td>
<td>16, 4WSA</td>
<td></td>
</tr>
<tr>
<td>UltraSPARC—I</td>
<td>16, 2WSA</td>
<td>16, DIR</td>
<td></td>
</tr>
<tr>
<td>UltraSPARC—I—II</td>
<td>16, 2WSA</td>
<td>16, DIR</td>
<td></td>
</tr>
<tr>
<td>R4400</td>
<td>16, DIR</td>
<td>16, DIR</td>
<td></td>
</tr>
<tr>
<td>R10000</td>
<td>32, 2WSA</td>
<td>32, 2WSA</td>
<td></td>
</tr>
<tr>
<td>PA7100</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PA8000</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PA8500</td>
<td>512, 4WSA</td>
<td>1024, 4WSA</td>
<td></td>
</tr>
</tbody>
</table>
### Figure MPSU5: Microprocessor cache memory (sources: [Prvulovic97], [Stojanovic95])

**Legend:**
- Icache—on-chip instruction cache;
- Dcache—on-chip data cache;
- L2 cache—on-chip L2 cache;
- DIR—direct mapped;
- $x$WSA—$x$-way set associative ($x = 2, 3, 4, 5, 8$);
- UNI—unified L1 instruction and data cache;
- * on-chip cache controller for external L2 cache;
- ** on-chip cache controller for external L1 cache;
- *** L2 cache is in the same package, but on a different silicon die.

**Comment:**
It is only an illusion that early HP microprocessors are lagging behind the others, as far as the on-chip cache memory support; they are using the so-called on-chip assist cache, which can be treated as a zero-level cache memory, that works on slightly different principles, compared to traditional cache (as it will be explained later on in this book). On the other hand, DEC was the first one to place both level-1 and level-2 caches on the same chip with the CPU.

<table>
<thead>
<tr>
<th>Microprocessor</th>
<th>ITLB</th>
<th>DTLB</th>
<th>BPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>PowerPC 601</td>
<td>256, 2WSA, UNI</td>
<td>___</td>
<td>___</td>
</tr>
<tr>
<td>PowerPC 604e</td>
<td>128, 2WSA</td>
<td>128, 2WSA</td>
<td>512×2BC</td>
</tr>
<tr>
<td>PowerPC 620</td>
<td>128, 2WSA</td>
<td>128, 2WSA</td>
<td>2048×2BC</td>
</tr>
<tr>
<td>Alpha 21064</td>
<td>12</td>
<td>32</td>
<td>4096×2BC</td>
</tr>
<tr>
<td>Alpha 21164</td>
<td>48 ASSOC</td>
<td>64 ASSOC</td>
<td>1CS×2BC</td>
</tr>
<tr>
<td>Alpha 21264</td>
<td>128 ASSOC</td>
<td>128 ASSOC</td>
<td>2LMH, 32×RAS</td>
</tr>
<tr>
<td>SuperSPARC</td>
<td>64 ASSOC, UNI</td>
<td>___</td>
<td>___</td>
</tr>
<tr>
<td>UltraSPARC-I</td>
<td>64 ASSOC</td>
<td>64 ASSOC</td>
<td>1CS×2BC</td>
</tr>
<tr>
<td>UltraSPARC-II</td>
<td>64 ASSOC</td>
<td>64 ASSOC</td>
<td>1CS×2BC</td>
</tr>
<tr>
<td>R4400</td>
<td>48 ASSOC</td>
<td>48 ASSOC</td>
<td>___</td>
</tr>
<tr>
<td>R10000</td>
<td>64 ASSOC</td>
<td>64 ASSOC</td>
<td>512×2BC</td>
</tr>
<tr>
<td>PA7100</td>
<td>16</td>
<td>120</td>
<td>___</td>
</tr>
<tr>
<td>PA8000</td>
<td>4</td>
<td>96</td>
<td>256×3BSR</td>
</tr>
<tr>
<td>PA8500</td>
<td>160, UNI</td>
<td>___</td>
<td>___</td>
</tr>
<tr>
<td>MC88110</td>
<td>40</td>
<td>40</td>
<td>___</td>
</tr>
<tr>
<td>AMD K6</td>
<td>64</td>
<td>64</td>
<td>8192×2BC, 16×RAS</td>
</tr>
<tr>
<td>i860 XP</td>
<td>64, UNI</td>
<td>___</td>
<td>___</td>
</tr>
<tr>
<td>Pentium II</td>
<td>___</td>
<td>___</td>
<td>___</td>
</tr>
</tbody>
</table>

### Figure MPSU6: Miscellaneous microprocessor features (source: [Prvulovic97])

**Legend:**
- ITLB—translation lookaside buffer for code [entries];
- DTLB—translation lookaside buffer for data [entries];
- 2WSA—two-way set associative; ASSOC = fully associative;
- UNI—unified TLB for code and data;
- BPS—branch prediction strategy;
- 2BC—2-bit counter;
- 3BSR—three bit shift register;
- RAS—return address stack;
- 2LMH—two-level multi-hybrid
(gshare for the last 12 branch outcomes and pshare for the last 10 branch outcomes);
ICS—instruction cache size (2BC for every instruction in the instruction cache);
* hinted instructions available for static branch prediction.

Comment:
The great variety in TLB design numerics is a consequence of the fact that different manufacturers see differently the real benefits of having a TLB of a given size. Grouping of pages, in order to use one TLB entry for a number of pages, has been used by DEC and viewed as a viable price/performance trade-off. Variable page size has been first used by MIPS Technologies machines.

The following sections give a closer look into the Intel Pentium, Pentium MMX, Pentium Pro, and Pentium II machines. The presentation includes a number of facts which could be difficult to comprehend without enough knowledge on advanced concepts in microprocessing and multimicroprocessing. However, all relevant concepts will be explained through the rest of the book, so everything should be more clear during the second reading of the book. Such presentation strategy has been selected intentionally. During the first reading, it is the most important to obtain the bird’s view of the entire forest. The squirrel’s view of each tree in the forest should be obtained during the second reading.

1.1. Pentium
The major highlights of Pentium include the features which make it different in comparison with the i486. The processor is built out of 3.1 MTr (Million Transistors) using the Intel’s 0.8 μm BiCMOS silicon technology. It is packed into a 273-pin PGA (Pin Grid Array) package, as indicated in Figure MPSU7. Pentium pin functions are shown in Figure MPSU8.

Pentium is fully binary compatible with previous Intel machines in the x86 family. Some of the above mentioned enhancements are supported with new instructions. The MMU (Memory Management Unit) is fully compatible with i486, while the FPU (Floating-Point Unit) has been redesigned for better performance.

Block diagram of the Pentium processor is shown in Figure MPSU9. The core of the processor is the pipeline structure, which is shown in Figure MPSU20, comparatively with the pipeline structure of the i486. A precise description of activities in each pipeline stage can be found in [Intel93].
Figure MPSU7: Pentium pin layout (source: [Intel93])

Comment:
The total pin count is 273. As it is typical for packages with a relatively high pin count, a large percentage of pins is dedicated to power and ground (in the case of Pentium, 50 and 49, respectively). However, note that much larger packages are in use today, with DEC and HP among the companies which are the leaders in the packaging technology.

<table>
<thead>
<tr>
<th>Function</th>
<th>Pins</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock</td>
<td>CLK</td>
</tr>
<tr>
<td>Initialization</td>
<td>RESET, INIT</td>
</tr>
<tr>
<td>Address Bus</td>
<td>A31–A3, BE7#–BE0#</td>
</tr>
<tr>
<td>Address Mask</td>
<td>A20M#</td>
</tr>
<tr>
<td>Data Bus</td>
<td>D63–D0</td>
</tr>
<tr>
<td>Address Parity</td>
<td>AP, APCHK#</td>
</tr>
<tr>
<td>Data Parity</td>
<td>DP7–DP0, PCHK#, PEN#</td>
</tr>
<tr>
<td>Internal Parity Error</td>
<td>IERR#</td>
</tr>
<tr>
<td>System Error</td>
<td>BUSCHK#</td>
</tr>
<tr>
<td>Bus Cycle Definition</td>
<td>M/IO#, D/C#, W/R#, CACHE#, SCYC, LOCK#</td>
</tr>
<tr>
<td>Bus Control</td>
<td>ADS#, BRDY, NA#</td>
</tr>
<tr>
<td>Page Cacheability</td>
<td>PCD, PWT</td>
</tr>
<tr>
<td>Cache Control</td>
<td>KEN#, WB/WT#</td>
</tr>
<tr>
<td>Cache Snooping/Consistency</td>
<td>A_HOLD, EADS#, HIT#, HITM#, INV</td>
</tr>
</tbody>
</table>
Cache Flush: FLUSH#
Write Ordering: EWBE#
Bus Arbitration: BOFF#, BREQ, HOLD, HLDA
Interrupts: INTR, NMI
Floating Point Error Reporting: FERR#, IGNNE#
System Management Mode: SMI#, SMIACT#
Functional Redundancy Checking: FRCMC# (IERR#)
TAP Port: TCK, TMS, TDI, TDO, TRST#
Breakpoint/Performance Monitoring: PM0/BP0, PM1/BP1, BP3–2
Execution Tracing: BT3–BT0, IU, IV, IBT
Probe Mode: R/S#, PRDY

**Figure MPSU8:** Pentium pin functions (source: [Intel93])

**Legend:**
TAP—Processor boundary scan.

**Comment:**
Traditional pin functions are clock, initialization, addressing, data, bus control, bus arbitration, and interrupts. These or similar functions can be found all the way back to the earliest x86 machines. Pin functions like parity, error control, cache control, tracing, breakpoint, and performance monitoring can be found in immediate predecessors, in a somewhat reduced form.

**Figure MPSU9:** Pentium block diagram (source: [Intel93])

**Legend:**
TLB—Translation Lookaside Buffer.

**Comment:**
This block diagram sheds light on the superscaling of Pentium processor. Two pipelines (U and V) are responsible for the execution efficiency.

**Intel486™ Pipeline**
Figure MPSU20: Intel 486 pipeline versus Pentium pipeline (source: [Intel93])

Legend:
Pf—Prefetch;
D1/2—Decoding 1/2;
EX—Execution;
WB—Writeback.

Comment:
This block diagram sheds light on the superscaling of Pentium processor, in comparison with the i486 processor. The depth of the pipeline has not changed; only the width. This is a consequence of the fact that technology has changed drastically in the sense of on-chip transistor count, and minimally in the sense of off-chip-to-on-chip delay ratio.

Internal error detection is based on FRC (Functional Redundancy Checking), BIST (Built-In Self Test), and PTC (Parity Testing and Checking). Constructs for performance monitoring count occurrences of selected internal events and trace execution through the internal pipelines.

1.1.1. Cache and Cache Hierarchy

Internal cache organization follows the 2-way set associative approach. Each of the two caches (data cache and instruction cache) includes 128 sets. Each set includes two entries of 32 bytes each (one set includes 64 bytes). This means that each of the two caches is 8 KB large. Both caches use the LRU replacement policy. One can add second-level caches off the chip, if needed by the application.

One bit has been assigned to control the cacheability on the page by page basis: PCD (1=CacheingDisabled; 0=CacheingEnabled). One bit has been assigned to control the write policy for the second level caches: PWT (1=WriteThrough; 0=WriteBack). The circuitry used to generate signals PCD and PWT is shown in Figure MPSU21. The states of these two bits appear at the pins PWT and PCD during the memory access cycle. Signals PCD and KEN# are internally ANDed in order to control the cacheability on the cycle by cycle basis (see Figure MPSU21).
DIR PTRS | DIRECTORY | TABLE (Optional) | OFFSET
---|---|---|---

**Figure MPSU21:** Generation of PCD and PWT (source: [Intel93])

**Legend:**
- PCD—a bit which controls cacheability on a page by page basis;
- PWT—a bit which controls write policy for the second level caches;
- PTRS—pointers;
- CR\textsubscript{i}—control register bit \(i; i = 0, 1, 2, 3, \ldots\)

**Comment:**
Pentium processor enables the cacheability to be controlled on the page by page basis, as well as the write policy of the second level cache, which is useful for a number of applications, including DSM. This figure also sheds light on the method used to transform linear address elements into the relevant control signals.

### 1.1.2. Instruction-Level Parallelism

Pentium is a 32-bit microprocessor based on 32-bit addressing; however, it includes a 64-bit data bus. Internal architecture is superscalar with two pipelined integer units; consequently, it can execute, on average, more than one instruction per clock cycle.

### 1.1.3. Branch Prediction
Branch prediction is based on a BTB (Branch Target Buffer). Address of the instruction which is currently in the D1 stage is applied to BTB. If a BTB hit occurs, the assumption is made that the branch will be taken, and the execution continues with the target instruction without any pipeline stalls and/or flushes (because the decision has been made early enough, i.e. during the D1 stage). If a BTB miss occurs, the assumption is made that the branch will not be taken, and the execution continues with the next instruction, also without pipeline stalls and/or flushes. The flushing of the pipeline occurs if the branch is mispredicted (one way or the other), or if it was predicted correctly, but the target address did not match. The number of wasted cycles on misprediction depends on the branch type.

The enclosed code in Figure MPSU22 shows an example in which branch prediction reduces the Pentium execution time by three times, compared to i486.

- Loop for computing prime numbers:
  for(k = i + prime; k <= SIZE; k += prime)
  flags[k] = FALSE;

- Allocation:
  prime - ecx
  k - edx
  FALSE - al

- Assembly code:
  inner_loop:
    mov byte ptr flags[edx], al
    add edx, ecx
    cmp edx, SIZE
    jle inner_loop

- Pairing: mov + add and cmp + jle

- One loop iteration execution time:
  \( T_{exe}[\text{Pentium (with branch prediction)}] = 2 \)
  \( T_{exe}[\text{i486}] = 6 \)

**Figure MPSU22**: Benefits of branch prediction (source: [Intel93])

**Comment**: Pairing refers to the two-issue superscaling capability of the Pentium processor. Speed comparison is expressed in the number of clock cycles per loop iteration, so that the technological differences between i486 and Pentium are eliminated.

### 1.1.4. Input/Output

Organization of the interrupt mechanism is a feature which is of importance for incorporation of an off-the-shelf microprocessor into microprocessor and multimicroprocessor systems. Interrupts inform the processor or the multiprocessor of the occurrence of external asynchronous events. External interrupt related details are specified in Figure MPSU23.

- Pentium recognizes 7 external interrupts with the following priority:
  - BUSCHK#
  - R/S#
  - FLUSH#
  - SMI#
  - INIT
  - NMI
  - INTR

- Interrupts are recognized at instruction boundaries.
In Pentium, the instruction boundary is at the first clock in the execution stage of the instruction pipeline. Before an instruction is executed, Pentium checks if any interrupts are pending. If yes, it flushes the instruction pipeline, and services the interrupt.

**Figure MPSU23:** External interrupt (source: [Intel93])

**Legend:**
- BUSCHK#—pin T03;
- R/S#—pin R18;
- FLUSH#—pin U02;
- SMI#—pin P18;
- INIT—pin T20;
- NMI—pin N19;
- INTR—pin N18.

**Comment:**
The bus check input (BUSCHK#) allows the system to signal an unsuccessful completion of a bus cycle; if this input is active, address bus signals and selected control bus signals are latched into the machine check registers, and the Pentium processor vectors to the machine check exception. The R/S# interrupt input is asynchronous and edge sensitive; it is used to force the processor into the idle state, at the next instruction boundary, which means that it is well suited for the debugging related work. The cache flush (FLUSH#) interrupt input forces the Pentium processor to write back to memory all modified lines of the data cache; after that, it invalidates both internal caches (data cache and instruction cache). The system management interrupt (SMI#) forces the Pentium processor to enter the system management mode at the next instruction boundary. The initialization (INIT) interrupt input pin forces the Pentium processor to restart the execution, in the same way as after the RESET signal, except that internal caches, write buffers, and floating point registers preserve their initial values (those existing prior to INIT). The earliest x86 machines include only NMI and INTR, i.e. the two lowest-priority interrupts of the Pentium processor (non-maskable interrupt typically used for power failure and maskable interrupt typically used in conjunction with priority logic).

### 1.1.5. Multithreading
Multithreading on the fine-grain level is not supported in the Pentium processor. Of course, Pentium processor can be used in the coarse-grain multiprocessor systems, in which case the multithreading paradigm is achieved through appropriate software and additional hardware constructs off the processor chip.

### 1.1.6. Support for Shared Memory Multiprocessing
Multiprocessor support exists in the form of special instructions, constructs for easy incorporation of the second level cache, and the implementation of the MESI (Modified/Exclusive/Shared/Invalid) and the SI (Shared/Invalid) protocols. The MESI and the SI cache consistency maintenance protocols are supported in a way which is completely transparent to the software.

The 8-KB data cache is reconfigurable on the line-by-line basis, as write-back or write-through cache. In the write-back mode, it fully supports the MESI cache consistency protocol. Parts of data memory can be made non-cacheable, either by software action or by external hardware. The 8-KB instruction cache is inherently write-protected, and supports the SI (Shared/Invalid) protocol.

Data cache includes two state bits to support the MESI protocol. Instruction cache includes one state bit to support the SI protocol. Operating modes of the two caches are controlled with
two bits in the register CR0: CD (Code Disable) and NW (Not Write through). System reset makes CD = NW = 1. The best performance is potentially obtained with CD = NW = 0. Organization of code and data caches is shown in Figure MPSU24.

![Diagram](MPSU24.png)

**Figure MPSU24:** Organization of instruction and data caches (source: [Intel93])

**Legend:**
MESI—Modified/Exclusive/Shared/Invalid;
LRU—Least Recently Used.

**Comment:**
This figure stresses the fact that Pentium processor uses a two-way set associative cache memory. In addition to tag address bits, two more bits are needed to specify the MESI state (in the data cache), while one more bit is needed to specify the SI state (in the instruction cache).

A special snoop (inquire) cycle is used to determine if a line (with a specific address) is present in the code or data cache. If the line is present and is in the M (modified) state, processor (rather than memory) has the most recent information and must supply it.

The on chip caches can be flushed by external hardware (input pin FLUSH# low) or by internal software (instructions INVD and WBINVD). The WBINVD causes the modified lines in the internal data cache to be written back, and all lines in both caches are to be marked invalid. The INVD causes all lines in both data and code caches to be invalidated, without any writeback of modified lines in the data cache.

As already indicated, each line in the Pentium processor data cache is assigned a state, according to a set of rules defined by the MESI protocol. These states tell whether a line is valid or not (I = Invalid), if it is available to other caches or not (E = Exclusive or S = Shared), and if it has been modified in comparison to memory or not (M = Modified). An explanation of the MESI protocol is given in Figure MPSU25. The data cache state transitions on read, write, and snoop (inquire) cycles are defined in Figures MPSU26, MPSU27, and MPSU28, respectively.

<table>
<thead>
<tr>
<th>M—Modified:</th>
<th>E—Exclusive:</th>
<th>S—Shared:</th>
</tr>
</thead>
<tbody>
<tr>
<td>An M-state line is available in ONLY one cache, and it is also MODIFIED (different from main memory). An M-state line can be accessed (read/written to) without sending a cycle out on the bus.</td>
<td>An E-state line is also available in only one cache in the system, but the line is not MODIFIED (i.e., it is the same as main memory). An E-state line can be accessed (read/written to) without generating a bus cycle. A write to an E-state line will cause the line to become MODIFIED.</td>
<td>This state indicates that the line is potentially shared with other caches (i.e., the same line may exist in more than one cache). A read to an S-state line will not generate bus activity, but a write to a SHARED line will generate a write-through cycle on the bus.</td>
</tr>
</tbody>
</table>
The write-through cycle may invalidate this line in other caches.
A write to an S-state line will update the cache.

I—Invalid: This state indicates that the line is not available in the cache.
A read to this line will be a MISS,
and may cause the Pentium processor to execute LINE FILL.
A write to an INVALID line causes the Pentium processor
to execute a write-through cycle on the bus.

Figure MPSU25: Definition of states for the MESI and the SI protocols (source: [Intel93])
Legend:
LINE FILL—Fetching the whole line into the cache from main memory.
Comment:
This figure gives only the precise description of the MESI and the SI protocols. Detailed ex-
planations of the rationales behind, and more, are given later on in this book (section on cach-
ing in shared memory multiprocessors).

<table>
<thead>
<tr>
<th>Present State</th>
<th>Pin Activity</th>
<th>Next State</th>
<th>Description</th>
</tr>
</thead>
</table>
| M             | N/A          | M          | Read hit;
data is provided to processor core
by cache.
No bus cycle is generated. |
| E             | N/A          | E          | Read hit;
data is provided to processor core
by cache.
No bus cycle is generated. |
| S             | N/A          | S          | Read hit;
data is provided to processor core
by cache.
No bus cycle is generated. |
| I             | CACHE# low
AND KEN# low
AND WB/WT# high
AND PWT low | E          | Data item does not exist in cache (MISS).
A bus cycle (read) will be generated
by the Pentium™ processor.
This state transition will happen
if WB/WT# is sampled high
with first BRDY# or NA#. |
| I             | CACHE# low
AND KEN# low
AND (WB/WT# low
OR PWT high) | S          | Same as previous read miss case
except that WB/WT# is sampled low
with first BRDY# or NA#. |
| I             | CACHE# high
AND KEN# high | I          | KEN# pin inactive;
the line is not intended to be cached
in the Pentium processor. |

Figure MPSU26: Data cache state transitions for UNLOCKED Pentium™ processor initiated read cycles* (source: [Intel93])
Legend:
* Locked accesses to data cache will cause the accessed line to transition to Invalid state.
Comment:
For more details see the section on caching in shared memory multiprocessors and the refer-
ence [Tomasevic93]. In comparison with a “theoretical” case, this “practical” case includes a
number of additional state transition conditions, related to pin signals.
**Activity State Description**

<table>
<thead>
<tr>
<th>Activity</th>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>N/A</td>
<td>M write hit; update data cache. No bus cycle generated to update memory.</td>
</tr>
<tr>
<td>E</td>
<td>N/A</td>
<td>E write hit; update cache only. No bus cycle generated; line is now MODIFIED.</td>
</tr>
<tr>
<td>S</td>
<td>PWT low AND WB/WT# high</td>
<td>E Write hit; data cache updated with write data item. A write-through cycle is generated on bus to update memory and/or invalidate contents of other caches. The state transition occurs after the writethrough cycle completes on the bus (with the last BRDY#).</td>
</tr>
<tr>
<td>S</td>
<td>PWT low AND WB/WT# low</td>
<td>S Same as above case of write to S-state line except that WB/WT# is sampled low.</td>
</tr>
<tr>
<td>S</td>
<td>PWT high</td>
<td>S Same as above cases of writes to S state lines except that this is a write hit to a line in a write through page; status of WB/WT# pin is ignored.</td>
</tr>
<tr>
<td>I</td>
<td>N/A</td>
<td>I Write MISS; a write through cycle is generated on the bus to update external memory. No allocation done.</td>
</tr>
</tbody>
</table>

**Legend:**
- WB/WT—Writeback/Writethrough.

**Comment:**
For more details see the section on caching in shared memory multiprocessors and the reference [Tomasevic93]. In comparison with a “theoretical” case, this “practical” case includes a number of additional state transition conditions, related to pin signals.

<table>
<thead>
<tr>
<th>Present State</th>
<th>Next State INV=1</th>
<th>Next State INV=0</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>I</td>
<td>S</td>
<td>Snoop hit to a MODIFIED line indicated by HIT# and HITM# pins low. Pentium™ processor schedules the writing back of the modified line to memory.</td>
</tr>
<tr>
<td>E</td>
<td>I</td>
<td>S</td>
<td>Snoop hit indicated by HIT# pin low; no bus cycle generated.</td>
</tr>
<tr>
<td>S</td>
<td>I</td>
<td>S</td>
<td>Snoop hit indicated by HIT# pin low; no bus cycle generated.</td>
</tr>
<tr>
<td>I</td>
<td>I</td>
<td>I</td>
<td>Address not in cache; HIT# pin high.</td>
</tr>
</tbody>
</table>

**Legend:**
- INV—Invalid bit.

**Comment:**
For more details see the section on caching in shared memory multiprocessors and the reference [Tomasevic93]. A good exercise for the reader is to create a state diagram, using the data from this table, and the previous two tables.

### 1.1.7. Support for Distributed Shared Memory

There is no special support for distributed shared memory. However, Pentium architecture includes two write buffers (one per pipe) which enhances the performance of consecutive writes to memory. Writes into these two buffers are driven out onto the external bus to memory, using the strong ordering approach. This means that the writes can not bypass each other.
Consequently, system supports sequential memory consistency in hardware. More sophisticated memory consistency models can be achieved in software.

1.2. Pentium MMX

Pentium MMX (MultiMedia eXtensions) microprocessor is a regular Pentium with 57 additional instructions for fast execution of typical primitives in multimedia processing, like: (a) vector manipulations, (b) matrix manipulations, (c) bit-block moves, etc....

For typical multimedia applications Pentium MMX is about 25% faster compared to Pentium, which makes it better suited for Internet server and workstation applications. See Figure MPSU29 for the list of MMX instructions.

Appearance of the MMX can be treated as the proof of the validity of the opinion that accelerators will play an important role in future machines. The MMX subsystem can be treated as an accelerator on the chip.

<table>
<thead>
<tr>
<th>EMMS—Empty MMX state</th>
<th>MOVD—Move doubleword</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVQ—Move quadword</td>
<td>PACKSSDW—Pack doubleword to word data (signed with saturation)</td>
</tr>
<tr>
<td>PACKSSWB—Pack word to byte data (signed with saturation)</td>
<td></td>
</tr>
<tr>
<td>PACKUSWB—Pack word to byte data (unsigned with saturation)</td>
<td></td>
</tr>
<tr>
<td>PADD—Add with wrap-around</td>
<td></td>
</tr>
<tr>
<td>PADDUS—Add unsigned with saturation</td>
<td></td>
</tr>
<tr>
<td>PAND—Bitwise And</td>
<td>PANDN—Bitwise AndNot</td>
</tr>
<tr>
<td>PCMPEQ—Packed compare for equality</td>
<td></td>
</tr>
<tr>
<td>PCMPGT—Packed compare greater (signed)</td>
<td></td>
</tr>
<tr>
<td>PMADD—Packed multiply add</td>
<td></td>
</tr>
<tr>
<td>PMULH—Packed multiplication</td>
<td></td>
</tr>
<tr>
<td>PMULL—Packed multiplication</td>
<td></td>
</tr>
<tr>
<td>POR—Bitwise Or</td>
<td></td>
</tr>
<tr>
<td>PSLL—Packed shift left logical</td>
<td></td>
</tr>
<tr>
<td>PSRA—Packed shift right arithmetic</td>
<td></td>
</tr>
<tr>
<td>PSRL—Packed shift right logical</td>
<td></td>
</tr>
<tr>
<td>PSUB—Subtract with wrap-around</td>
<td></td>
</tr>
<tr>
<td>PSUBS—Subtract signed with saturation</td>
<td></td>
</tr>
<tr>
<td>PSUBUS—Subtract unsigned with saturation</td>
<td></td>
</tr>
<tr>
<td>PUNPCKH—Unpack high data to next larger type</td>
<td></td>
</tr>
<tr>
<td>PUNPCKL—Unpack low data to next larger type</td>
<td></td>
</tr>
<tr>
<td>PXOR—Bitwise Xor</td>
<td></td>
</tr>
</tbody>
</table>

Figure MPSU29: New instructions of the Pentium MMX processor (source: [Intel97])

Legend:
MMX—MultiMedia eXtension.

Comment:
These 26 code types expand to 57 instructions, once different data types are taken into consideration (one quad, two doublewords, four words, or eight bytes). Some multimedia applications have reported a 20% performance increase with Pentium MMX, compared to Pentium. Note that real application performance increase is always smaller than the benchmark performance increase.

1.3. Pentium Pro

The major highlights of Pentium Pro include the features which make it different in comparison with the Pentium processor. It was first announced in the year 1995. It is based on a
5.5. MTr design, the Intel’s 0.6 μm BiCMOS silicon technology, and runs at 150 MHz. A newer 0.3 μm version runs at 200 MHz. The slower version achieves 6.1 SPECint95 and 5.5 SPECfp95. The faster version achieves 8.1 SPECint95 and 6.8 SPECfp95.

Both the internal and the external buses are 64-bits wide and run at 50 MHz (slower version) and 66 MHz (faster version). Processor supports the split transactions approach, which means that address and data cycles are decoupled (another independent activity can happen between the address cycle and the data cycle). Processor includes 40 registers, each one 32 bits wide. Branch prediction is based on a BTB (Branch Target Buffer).

Pentium Pro is superpipelined and includes 14 stages. Pentium Pro is also superscalar and the IFU (Instruction Fetch Unit) fetches 16 bytes per clock cycle, while the IDU (Instruction Decoding Unit) decodes 3 instructions per clock cycle. Processor supports the speculative and the out-of-order execution.

The first level caches are on the processor chip; both of them are 2-way set-associative 8-KB caches with buffers which handle 4 outstanding misses. The second level cache includes both data and code. It is 4-way set-associative, 256 KB large, and includes 8 ECC (Error Correction Code) bits per 64 data bits.

Support for SMP (Shared Memory multiprocessing) and DSM (Distributed Shared Memory) is in the form of the MESI protocol support. Some primitive support for MCS (Multi-Computer Systems) and DCS (Distributed Computer Systems) is in the form of two dedicated ports, one for input and one for output.

Block diagram of the Pentium Pro machine is given in Figure MPSU30, together with brief explanations. Detailed explanations of underlying issues in ILP (Instruction Level Parallelism) and BPS (Branch Prediction Strategies) are given in the later chapters of this book.

**Figure MPSU30:** Pentium Pro block diagram (source: [Papworth96])

**Legend:**
AGU—Address Generation Unit
BIU—Bus Interface Unit
BTB—Branch Target Buffer
DCU—Data Cache Unit
FEU—Floating-point Execution Unit
ID—Instruction Decoder
IEU—Integer Execution Unit
IFU—Instruction Fetch Unit (with I-cache)
L2—Level-2 cache
MIS—MicroInstruction Sequencer
MIU—Memory Interface Unit
MOB—Memory reOrder Buffer
RAT—Register Alias Table
ROB—ReOrder Buffer
RRF—Retirement Register File
RS—Reservation Station

Comment:
The L2 cache is on the same package as the CPU, but on another chip (the Pentium Pro package includes two chips). Major additions, compared to Pentium processor, are those related to the exploitation of the instruction level parallelism (RAT, ROB, RRF, and RS).

1.4. Pentium II

In the first approximation, Pentium II can be treated as a regular Pentium Pro with additional MMX instructions to accelerate the multimedia applications. However, it includes other innovations aimed at decreasing the speed gap between the CPU and the rest of the system.

An important element of Pentium II is DIB (Dual Independent Bus) structure. As indicated in Figure MPSU31, one of the two Pentium II buses goes towards L2 cache; the other one goes towards DRAM (Dynamic RAM) main memory, and I/O, and other parts of the system. Consequently, the CPU bus is less of a system bottleneck now.

Also, a new packaging technology is used in Pentium II for the first time by Intel. It is the SEC (Single Edge Contact) connector, which enables the CPU to operate at larger speeds. For the most recent speed comparison data, interested reader is referred to Intel’s WWW presentation (http://www.intel.com/).

![Figure MPSU31: Conventional and Pentium II bus structures (source: [Intel97])](image)

Legend:
SB—single independent bus;
DIB—dual independent bus;
CLC—control logic chipset;
L2—second level cache;
SEC—Single Edge Contact connector.

Comment:
The dual bus structure of Pentium II (DIB) eliminates a critical system bottleneck of Pentium Pro—the single bus structure (SB). All L2 cache control logic is now integrated on the Pentium II chip. The CPU, L2 cache, and a cooler are on the same SEC. There are two cache options: 256 KB or 512 KB.
2. Advanced Issues

One of the basic issues of importance in microprocessing is correct prediction of future development trends. The facts presented here have been adopted from [Sheaffer96], and represent a possible way to be paved by Intel in future developments until the year 2000, and after that.

Soon after the year 2000, the on-chip transistor count is predicted to reach about 1GTr (one gigatransistors) using the minimum lithographic dimension of less than 0.1 μm (micrometers) and the gate oxide thickness of less than 40 Å (angstroms). Consequently, microprocessors on a single chip will be faster. Their clock rate is expected to be about 4 GHz (gigahertz), to enable the average speed of about 100 BIPS (billion instructions per second). This means that the microprocessor speed, measured using standard benchmarks, may reach to about 100 SPECint95 (which is about 3500 SPECint92).

In such conditions, the basic issue is how to design future microprocessors, in order to maintain the existing trend of approximately doubling the performance in every 18 months. Hopefully, some of the answers will be clear, once the reading of this book is completed.

As far as the process technology, the delay trends are given in Figure MPSS1, and the area trends in Figure MPSS2, for the case of Intel products. Position of the ongoing project, referred to as P7 or Merced, can be estimated using extrapolation. Frequency of operation and transistor count represent alternative ways to express performance and complexity. Related data for the line of Intel products are given in Figure MPSS3 and Figure MPSS4.

![Figure MPSS1: Microprocessor chip delay trends (source: [Sheaffer96])](image)

Legend:
Metal 2 (2 mm)—Two level metal.

Comment:
As the basic geometry becomes smaller, gates become faster (due to smaller distances that carriers have to cross), but wires become slower (due to increased parasitic capacitances and resistances). Architectures which are suitable for basic geometries larger than 0.25 μm have to be reconsidered before an attempt is made to reimplement them in basic geometries smaller than 0.25 μm.

<table>
<thead>
<tr>
<th>Silicon process technology</th>
<th>1.5 μm</th>
<th>1.0 μm</th>
<th>0.8 μm</th>
<th>0.6 μm</th>
<th>0.35 μm</th>
<th>0.25 μm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delay ps</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Figure MPSS2: Microprocessor chip area trends (source: [Sheaffer96])

Legend:
est—estimated.

Comment:
Once a new architecture is created, it is typically reimplemented once or twice in newer technologies, which results in smaller chips and better speeds. In some companies, the decision to go into the next generation architecture comes after the technology has advanced enough so that the next generation architecture can fit into the chip of approximately the same area as the first attempt of the previous generation. Consequently, the first implementations of i386 and i486 are of approximately the same chip area; also, the first implementations of Pentium and Pentium Pro are of approximately the same chip area. However, in some cases, the move to the next generation is possible only after a major technological advancement is achieved; consequently, the chip area of the initial implementation of Pentium is considerably larger than the chip area of the initial implementation of i486. In conclusion, development of microprocessor technology includes both evolutionary phases, as well as revolutionary steps.

Figure MPSS3: Microprocessor chip operation frequency (source: [Sheaffer96])

Legend:
PP—Pentium Processor;
PPro—Pentium Pro Processor.

Comment:
The raw speed of microprocessors increases at the steady speed of about one order of magnitude per decade. Note, however, that what is important is not the speed of the clock (raw speed), but the time to execute a useful application (semantic speed).

**Figure MPSS4**: Microprocessor and memory complexity (source: [Sheaffer96])

**Comment:**
The fact that the memory curve is above the processor curve tells that highly repetitive structures pack more transistors onto the same area die. The distance between the two curves has a slightly increasing trend which means that the higher the complexity, the more difficult it is to fit a given irregular structure onto the same area die.

Generally, modern microprocessors have been classified in two basic groups: (a) Brainiacs (lots of work during a single clock cycle, but slow clock), and (b) Speedemons (fast clock, but not much work accomplished during a single clock cycle). Figure MPSS5 can be treated as a proof of the statement that Intel microprocessors belong in between the two extreme groups. Looks like, Intel processors have started as brainiacs; however, over the time they have taken a course which is getting closer to speedemons. Figure MPSS6 compares the time budget of brainiacs and speedemons.
Figure MPSS5: Microprocessor sophistication (source: [Sheaffer96])

Comment:
Brainiacs tend to have a slower clock; however, on average, they accomplish more during a single clock period. On the other hand, speedemons are characterized by an extremely fast clock and little work done during each clock period. Of course, what counts is the speed of the compiled high-level language code (semantic speed). Consequently, an architecture is superior if it has been designed wisely, not if it has been designed as a brainiac or a speedemon. Typically, the design path in between the two extremes has the highest success expectations. Maybe that explains the decision of Intel to select the medium approach in between the two extremes.

"Brainiacs" Time Budget
3. About the Research of the Author and His Associates

During the past decade, in the specific area of microprocessor design and architecture, the research and development activities of the author and his associates have concentrated on: design of models of modern 32-bit and 64-bit microprocessors using HDLs (Hardware Description Languages) which are able to run machine/assembly level programs, for the major purpose of: (a) architecture experimentation and (b) silicon compilation.

Under the first term (architecture experimentation), an experimentation environment is assumed which enables one to test new ideas about future products based on the particular microprocessor, or to test new ideas about the architecture of that specific microprocessor.

Under the second term (silicon compilation), a design environment is assumed which enables one to do logic synthesis and VLSI implementation of improved versions of existing microprocessors, or completely new architectures of new microprocessors.

When it comes to models for architecture experimentation, so far 10 different model sets have been developed. Their list includes: University of California at Berkeley RISC, Stanford University MIPS, selected Intel machines (i8086, i486, Pentium Pro, i860, and i960), selected Motorola machines (mc68000, mc88100), SGI R4000, SUN Sparc V.8, DEC Alpha 21064, IBM Power601, HP PA7, and AMD K5. Those that have been tested exhaustively can be found on one of the university’s web pages [Milicev96]. For more information on the design strategies, the reader is referred to [Milicev97].
The major problem in all these projects was to come up with coding strategies which enable fast execution of compiled models written initially in one of the HDL languages. A more detailed description is given in [Milicev97].

When it comes to models for silicon compilation, so far one model has been designed, with the major goal to test the capabilities of the state-of-the-art silicon compilation. This model is an equivalent of Intel i860XP which can be treated as a jumbo cell (core) that enables on-chip accelerators to be added. The Intel i860XP is based on over 2.5 MTr and is treated as the first microprocessor with elements of 64-bit processing. Its HDL code (without details protected by the contract) can be found on one of the university’s web pages [Milicev96]. For more information, on the design experiences, the reader is referred to [Milutinovic95d].

The major problems in this particular project were: (a) to come up with a design strategy which enables a design with more than 1 MTr to be handled efficiently using silicon compilation with relatively modest capabilities, and also (b) to come up with a zero-bug design in minimal time (more than 10 MB of test programs have been prepared for this particular purpose, by a company in Japan which specializes in microprocessor test vector generation). For more information, the reader is referred to [Milutinovic95d].
ISSUES OF IMPORTANCE
The seven sections to follow cover the seven problem areas of importance for future microprocessors on a single VLSI chip, having in mind the ultimate goal, which is an entire DSM on a single chip, together with a number of simple or complex accelerators.
Cache and Cache Hierarchy

This chapter includes three sections. The section on basic issues covers three simple examples (associative, direct mapped, and set-associative cache), and gives the basic definitions of cache hierarchy. The section on advanced issues covers selected state of the art contributions. The section on the research of the author and his associates concentrates on the author’s idea to split the cache hierarchy into two parts—one for data which exhibit a predominantly temporal locality and one for data which exhibit a predominantly spatial locality—the concept which is of interest for future microprocessor designs.

1. Basic Issues

This section gives only the very basic definitions, and directs the reader to another source for details—the well known and educationally excellent book of Andrew Tanenbaum. The text to follow assumes that the basic caching issues are well understood, which is one of the prerequisites for reading this book.

The major issue in cache design is how to design a close-to-the-CPU memory which is both fast enough and large enough, but cheap enough. In other words, the major problem here is not technology, but techno-economics, in conditions of constantly changing technology costs and characteristics (which is what some researchers forget, sometimes).

The solution to the above defined problem which includes the elements of both technology and the economics is cache memory. This solution was made possible after it was recognized that programs do not access their data in a way which is completely random—the principle of locality is inherent to data and code access patterns. The discussion to follow implies the processor-memory organization from Figure CACU1 and the related access patterns.

![Figure CACU1: On-chip cache (source: [Tanenbaum90])](image)

Legend:
CAC—Cache and cache hierarchy.

Comment:
This figure presents a simple processor/memory model used to explain the caching approaches in the figures to follow. Note that some new microprocessor system architectures make a departure from this simple processor/memory model (like the dual independent bus structure of Intel Pentium II). However, the difference does not affect the essence of caching.

Three cache organizations are possible: (a) associative, as one extreme (b) set-associative, as a combined solution, and (c) direct-mapped, as another extreme. In all these cases, the memory capacity is assumed to be equal to $2^m$ and divided into consecutive blocks of $b$ bytes ($b$ is also a power of two).

1.1. Fully-associative cache
An example using the associative approach is shown in Figure CACU2. Each cache line contains one block, together with the block number and a validity bit (showing if the corresponding block contains a valid data item).

![Figure CACU2: An associative cache with 1024 lines and 4-byte blocks](source: [Tanenbaum90])

Comment:
In the case of a fully-associative cache it can not happen that a block is purged out of the cache only because a subset of its address bits matches the same subset of address bits of the newly referenced block. Consequently, if the replacement algorithm is properly designed, the block to be purged is always the block which is the least likely to be referenced again.

1.2. Set-associative cache
An example using the set-associative approach is shown in Figure CACU3. Each cache line contains several blocks, together with the block tag and a validity bit.

<table>
<thead>
<tr>
<th>Line</th>
<th>Valid</th>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1.3. Direct-mapped cache

An example using the direct-mapped approach is shown in Figure CACU4. Each cache line contains one block, together with the tag and a validity bit. Development of a detailed step-by-step example is left to the reader as an exercise.

```
<table>
<thead>
<tr>
<th>Line</th>
<th>Valid</th>
<th>Tag</th>
<th>Value</th>
<th>Addresses that use this line:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>12130</td>
<td>0, 4096, 8192, 12288, ...</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>170</td>
<td>4, 4100, 8196, 12292, ...</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2142</td>
<td>8, 4104, 8200, 12296, ...</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td>12, 4108, 8204, 12300, ...</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td>16, 4112, 8208, 12304, ...</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td></td>
<td></td>
<td>20, 4116, 8212, 12308, ...</td>
</tr>
<tr>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>4092, 8188, 12284, ...</td>
</tr>
<tr>
<td>1023</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Figure CACU4: A direct-mapped cache with 1024 4-byte lines and a 24-bit address (source: [Tanenbaum90])

Comment:
In the case of direct-mapped cache, the benefits of associativity are non-existent. Consequently, a match of a subset of address bits of a block will force block replacement of a block which is likely to be reused. Some compiled vector processing code may result in patterns which periodically destroy the item to be needed next. Interested reader is encouraged to develop such a code, to demonstrate the issue.

2. Advanced Issues

This section gives a short overview of selected research efforts in the field of uniprocessor cache memory. An idea which deserves attention is to use pointers to page numbers rather than page numbers, in cache architectures of modern microprocessors and multimicroprocessors. This is another European contribution in the field in which the US science dominates [Seznec96]. Figures CACS1, CACS2, CACS3, and CACS4 give the basic explanation.

<table>
<thead>
<tr>
<th>Address</th>
<th>Tag</th>
<th>V</th>
<th>Data</th>
</tr>
</thead>
</table>

Figure CACS1: Structure of a cache line (source: [Seznec96])

Legend:
V—Valid bit.
Comment:
This structure includes only the fields which are absolutely necessary to explain the concept. Realistic microprocessor system caches include a number of additional tags.

**Figure CACS2:** Indirect-tagged VV cache; TLB serves as a PN-cache (source: [Seznec96])

**Legend:**
VV—Virtually indexed, virtually tagged;  
PN—Page number.

**Comment:**
Unlike the next two structures, this one does not include a separate page number cache. The physical page number is obtained directly from the TLB.

**Figure CACS3:** Indirect-tagged VP cache with physical page numbers stored in the PN-cache (source: [Seznec96])

**Legend:**
VP—Virtually indexed, physically tagged;  
PN—Page number.

**Comment:**
Note that the address fields A0 and A1 are applied at the same time to both tag array and cache array.
This idea is of special interest for object-oriented environments where the pointers to page numbers may exhibit a higher level of locality, which is of crucial importance for cache efficiency. Variations of this idea are expected to be seen in future machines for object-oriented environments.

Still another European contribution which deserves attention is the difference-bit cache [Juan96]. The difference-bit cache is a two-way set-associative cache with a smaller access time, which is important, since the cycle time of a pipelined processor is usually determined by the cache access time.

The smaller access time was achieved by noticing that the two tags for a set have to differ at least by one bit, and this bit can be used to select the way in a two-way set associative cache. Consequently, all hits of a difference-bit cache are one cycle long.

Recent research at Hewlett-Packard [Chan96] introduces the concept of assist cache, which can be treated as a zero-level cache, in systems in which only the zero-level cache is on the microprocessor chip, and level-1/level-2 caches are off the processor chip.

The major idea behind the assist cache is to treat spatial data (elements of complex data structures) in a different way compared to other data, like temporal data (single variables), etc. Spatial data are likely to be used only once, and consequently after being used, on the next replacement, they are not placed back into the off-chip cache; this cache is bypassed and spatial data go directly to memory.

An important research track is related to victim caches. First mention of the underlying concept dates back to [DeJuan87+DeJuan88]. The concept was introduced under the name victim cache in [Jouppi90]. An improvement referred to as selective victim caching is described in [Stiliadis97].

This research starts from the design tradeoff between direct-mapped and set-associative or fully-associative caches. Direct-mapped caches are faster (which means a shorter cycle time); however, their miss ratio is higher (which means more cycles for execution of a program) because semantically related data can map into the same cache block. Fully-associative caches
represent the opposite extreme (longer access time, but higher hit ratio). Set-associative caches are in between.

One question is what is better: (a) fewer cycles which are longer (fully-associative cache), or (b) more cycles which are shorter (direct-mapped). Paper [Hill88] compares direct-mapped cache and set-associative cache (fully-associative cache has no practical value in a number of applications), and concludes that direct-mapped cache gives shorter overall execution time for a set of important system and application programs. Paper [Milutinovic86] brings the same conclusion for GaAs technology, through a side study leading to an early GaAs RISC [Milutinovic97].

Another question is what to do in order to obtain access time of the direct-mapped cache and, at the same time, hit ratio of the set-associative approach. Victim cache is a solution. It is a small fully-associative cache. Its place in the cache/memory hierarchy is one level above the first-level cache (L1), which is direct-mapped. Blocks replaced from the direct-mapped L1 cache are placed into the fully-associative victim (V) cache (kind of level 1½). If this block is referenced again, before being replaced from the V cache, the L1 miss penalty will be relatively small (because the victim cache is small and physically close). Since the V cache is fully-associative, many blocks which conflict with each other in the direct-mapped L1 cache, coexist without problems in the V cache.

In the case of the selective victim cache, incoming blocks to L1 are placed selectively either into L1 or into V cache, based on their past behavior. This implies the existence of a prediction mechanism, to determine the access likelihood of the incoming block. Blocks which are more likely to be accessed are placed into L1 cache; blocks which are less likely to be accessed are placed into V cache. Similarly, when a miss is being serviced from the V cache, the prediction mechanism is applied again, to determine if an interchange of the conflicting blocks is required (which is mandatory in the case of the classical V cache). For 10 different Spec92 benchmarks, selective victim cache provides the cache miss rate improvement of up to about 20%, and interchange traffic between L1 and V for up to about 70%. The selective victim cache approach can be based on a number of different prediction mechanisms.

Several research efforts which try to improve the cache subsystem by treating spatial and temporal data in different ways are in progress at the Polytechnic University of Barcelona, Colorado State University, and University of Belgrade. Details can be found in the IEEE TCCA Newsletter, March 1997 [Sanchez97, Tomasko97] and at the SCIzzL workshops [Milutinovic96a, Milutinovic96b].

Again, development of step-by-step procedures is left to the reader. Remember, the major goal of this book is to serve as a textbook for advanced undergraduate courses, and exercises with lots of freedom are treated as a major mechanism to induce creativity among students/readers.

An important new trend in cache design implies the techniques which optimize the cache design and performance in conditions typical of superscalar microprocessors [Vajapeyam97, Nair97].

3. About the Research of the Author and His Associates

During the past decade, in the specific area of caching for single processor environments, this author was involved in two major projects, and this section tries to summarize some of his experiences.
One was the simulation study of cache design parameters for the RCA’s early 32-bit 40 MHz SOS Silicon microprocessor, which is mentioned in the previous book [Milutinovic95b], and will not be further elaborated here. The essence of that research was to encompass absolutely all cache design parameters that one could imagine and to use (abuse) a set of VAX machines (state-of-the-art in those days) as a single user, for weeks. All that in the environment corresponding to specific RCA’s technology and end user’s application.

The other project was the development of an idea to split the data cache into two subsystems, one for data which expresses temporal locality, and another one for data which expresses spatial locality. Author created this idea while listening to a conference presentation in poor English, back in January 1995 at a conference in Hawaii (HICSS-28). For a while, due to the poor English of the presenter, the author was of the opinion that splitting the cache along the lines of locality type was the major contribution of the paper. Later, it turned out that the paper had absolutely nothing to do with locality and/or splitting. Simply, the first transparency included a bullet saying that caches are based on the locality principle, and (by accident) term “temporal” and term “spatial” were placed in two different text lines (which is what “switched on” the “bulb” in the authors mind). Technical report with the basic idea and the relevant mechanisms (developed in the plane on the way back) was published immediately after returning from the conference [Milutinovic95a].

The analysis of this idea was based on the concrete technology data from the SUN UltraSparc V.9. environment (one of the co-authors in the papers on performance and complexity analysis was the architect of SUN UltraSparc II microprocessor). Results of performance analysis have been given in [Milutinovic96a]. Results of the complexity analysis have been given in [Milutinovic96b]. Presentation in this book has been adopted (with permission of the co-authors) from a merged version of references [Milutinovic96a] and [Milutinovic96b].
Instruction-Level Parallelism

This chapter includes three sections. The first one is oriented to basic issues (background). The second one is oriented to advanced issues (state-of-the-art). The third one is oriented to the research of the author and his associates (basic ideas and pointers to references).

1. Basic Issues

Instruction level parallelism (ILP) and its efficient exploitation are crucial for maximization of the speed of compiled high-level language (HLL) code. Methods to exploit ILP include superscalar (SS), superpipelined (SP), and very large instruction word (VLIW) processing. In principle, ILP can be viewed as a measure of the average number of instructions that an appropriate SS, SP, or VLIW processor might be able to execute at the same time.

Instruction timing of a RISC processor and an SS processor are given in Figure ILPU1 and Figure ILPU2, respectively. The ILP is a function of the number of dependencies in relation to other instructions. Unfortunately, an architecture is not always able to support all available ILP. Consequently, another measure—machine level parallelism (MLP)—is often used, as a measure of the ability of an SS processor to take advantage of the available ILP. For the best performance/complexity ratio, the MLP and the ILP have to be balanced.

![Figure ILPU1: Instruction timing in a RISC processor without memory limitations (source: [Johnson91])](image)

34
Legend:
i—instruction;
f—fetch;
d—decode;
e—execute.

Comment:
This is a typical RISC-style instruction execution pipeline with three stages. It represents the baseline for explanation of other cases to follow.

![Diagram of instruction timing in a superscalar processor](image)

**Figure ILPU2:** Instruction timing in a superscalar processor without memory limitations (source: [Johnson91])

**Comment:**
This is a typical two-issue superscalar instruction execution pipeline with three stages. Two instructions are fetched at a time, and executed concurrently, on condition that no data dependencies are involved.

Fundamental performance limitations of an SS processor are illustrated in Figures ILPU3, ILPU4, and ILPU5, respectively. Figure ILPU3 explains the case of true data dependencies in a superscalar environment (one cycle is lost). Figure ILPU4 explains the case of procedural dependencies in the same superscalar environment (the number of lost cycles is much larger). Figure ILPU5 explains the case of resource conflicts in the same superscalar case (effects are the same as in Figure ILPU3, but the cause is different). Finally, Figure ILPU6 shows the impact of resource conflicts for the case of a long operation, in two different scenarios, when the execution unit is internally not pipelined (worse case, from the performance point of view) and internally fully pipelined (better case, from the performance point of view).

![Diagram of true data dependencies](image)

**Figure ILPU3:** True data dependencies—effect of true data dependencies on instruction timing (source: [Johnson91])

**Comment:**
The same instruction pipeline from the Figure ILPU2 is presented, in the case when the second of the two concurrently fetched instructions is data dependent on the first one.
Figure ILPU4: Procedural dependencies—effect of procedural dependencies on instruction timing (source: [Johnson91])

**Comment:**
The same instruction pipeline from Figure ILPU2 is presented again, now for the case when the second instruction is a branch.

<table>
<thead>
<tr>
<th>Time</th>
<th>Without Conflicts</th>
<th>With Conflict</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i_0 )</td>
<td>f d e</td>
<td>f d e</td>
</tr>
<tr>
<td>( i_1 )</td>
<td>f d e</td>
<td>f d e</td>
</tr>
<tr>
<td>( i_2 )</td>
<td>f d e</td>
<td></td>
</tr>
<tr>
<td>( i_3 )</td>
<td>f d e</td>
<td></td>
</tr>
<tr>
<td>( i_4 )</td>
<td>f d e</td>
<td></td>
</tr>
<tr>
<td>( i_5 )</td>
<td>f d e</td>
<td></td>
</tr>
</tbody>
</table>

Figure ILPU5: Resource conflicts—effect of resource conflicts on instruction timing (source: [Johnson91])

**Comment:**
This figure which is related to resource conflict looks the same as a previous figure which is related to data dependency, except that causes of the one cycle loss are different in the two different cases.

Case 1

\[
\begin{align*}
\hline
& i_0 & f & d & e_1 & e_2 & e_3 \\
& i_1 & f & d & e_1 & e_2 & e_3 \\
\hline
\end{align*}
\]

Case 2

\[
\begin{align*}
\hline
& i_0 & f & d & e_1 & e_2 & e_3 & e_4 \\
& i_1 & f & d & e_1 & e_2 & e_3 & e_4 \\
\hline
\end{align*}
\]

Figure ILPU6: Resource conflicts—effect of resource conflicts on instruction timing (source: [Johnson91])

**Comment:**
The same instruction pipeline from Figure ILPU5 is presented, under the condition when the execution takes several pipeline stages to complete. Under such a condition, two alternative
solutions represent a price/performance tradeoff. The one which wastes less cycles is more expensive to implement.

Issue (fetch and decode), execution (in the more general sense), and completion (write back which changes the state of the finite state machine called microprocessor) are the major elements of an instruction. In the least sophisticated microprocessors these elements are in-order. In the most sophisticated microprocessors these elements are out-of-order, which means better performance for more VLSI complexity. The essence will be explained using an example from [Johnson91].

In this example, a microprocessor which can issue two instructions at a time, execute three at a time, and complete two at a time, is assumed. Furthermore, the following program related characteristics are assumed: (a) Instruction I1 requires two cycles to execute, (b) Instructions I3 and I4 conflict for a functional unit, (c) Instruction I5 depends on the datum generated by I4, and (d) Instructions I5 and I6 conflict for a functional unit. The case of the in-order issue and in-order completion is given in Figure ILPU7. The case of in-order issue and out-of-order completion is given in Figure ILPU8. The case of out-of-order issue and out-of-order execution is given in Figure ILPU9.

### Figure ILPU7: Example II (source: [Johnson91])

**Legend:** II—In-order issue, In-order completion.

**Comment:** Instructions I1 and I2 are issued together to the execution unit and have to be written back (i.e., completed) together. The same applies for the pair I3 and I4, as well as the pair I5 and I6. Total execution time is eight cycles.

### Figure ILPU8: Example IO (source: [Johnson91])

**Legend:** IO—In-order issue, Out-of-order completion.

**Comment:** Instructions I1 and I2 are issued together to the execution unit; however, they do not have to be written back together. The same applies to I3 and I4, as well as I5 and I6, which saves one clock cycle. Total execution time is seven cycles.
Instruction pairs do not have to be issued together and can be executed out-of-order. Because of that, I6 is issued and executed before I5 (which is data dependent on I4), and consequently the execution time takes six clock cycles.

In this particular example, the simplest case (II, in-order issue and in-order completion) takes 8 cycles to execute. The medium case (IO, in-order issue and out-of-order completion) takes 7 cycles to execute. The most sophisticated case (OO, out-of-order issue and out-of-order completion) takes 6 cycles to execute. These values have been obtained after the following machine related characteristics are assumed: (a) Instruction is present in the decoding unit until its execution starts, and (b) Each instruction is executed in the appropriate execution unit.

In the simplest case (II), the completion can be done only after both paired instructions get fully executed; both get completed together. This approach (II) is typical of scalar microprocessors and rarely used in superscalar microprocessors.

In the medium case (IO), execution of an instruction can start as soon as the related resource is available; also, the completion is done as soon as the execution gets finished (I2 completes out-of-order). There are three basic cases when the issue of an instruction has to be stalled: (a) when such issue could generate a functional unit conflict, (b) when the instruction to be issued depends on the instruction(s) not yet completed, and (c) when the result of the issued instruction could be overwritten by an older instruction which takes longer to execute, or by a following instruction not yet executed. Special purpose control hardware is responsible for stalling, in all three cases. This approach (IO) was first used in scalar microprocessors; however, its major use is in superscalar microprocessors.

In the most sophisticated approach (OO), the processor is able to look ahead beyond the instruction which has been stalled (due to one of the reasons listed in the previous paragraph on the IO approach), which is not possible with the IO approach. Fetching and decoding beyond the stalled instruction is made possible by inserting a resource called *instruction window*, between the decode stage and the execute stage. Decoded instructions are placed into the instruction window (if enough space there), and examined for resource conflicts and possible dependencies. Term instruction window (or window of execution) refers to the full set of instructions that may be simultaneously considered for parallel execution, subject to data dependencies and data conflicts. As soon as an executable instruction is detected (like I6 in Figure ILPU9), it is scheduled for execution, regardless of the program order, i.e. out-of-order (the only condition is that program semantics are preserved). This approach (OO) introduces one additional type of hazard, when instruction \(N + 1\) destroys the input of instruction \(N\) (this case has to be watched for by the control hardware). So far, this approach has been used only in superscalar microprocessors.

In this context, important roles are played by BHT (branch history table) and BTB (branch target buffer). The BHT helps determine the branch outcome. The BTB helps compute the branch target address. These issues will be discussed in more detail later on.

A related approach—VLIW (Very Long Instruction Word)—is shown in Figure ILPU10. Single instruction specifies a larger number of operations to be executed concurrently. Conse-
quently, the number of run-time instructions is smaller, but the amount of compile-time activities is larger (and the code size may increase if the compiler is not adequate for the given architecture/application). The approach is well suited for special purpose architectures/applications and not well suited for making new VLIW machines which are binary compatible with existing machines (binary compatibility is the ability to execute a machine program written for an architecture of an earlier generation).

Figure ILPU10: Instruction timing of a VLIW processor (source: [Johnson91])

Comment:
The major difference between VLIW and superscalar processing is that VLIW fetches one instruction at the time; however, this instruction includes specifiers for several operations to be executed. On the contrary, superscalar fetches several instructions at the time; however, each one specifies only one operation to execute. Note that superscalar approach makes it easier to create a microprocessor which is code compatible with previous generation machines of the same manufacturer.

The related superpipelined approach is shown in Figure ILPU11. Stages are divided into substages; the more substages per stage—the deeper the superpipelining. Figure ILPU11 refers to the case with the depth of pipelining equal to two. Superpipelining (SP) takes longer to generate a result (1.5 clock cycles for two instructions), compared with superscaling (SS) which takes shorter (1 clock cycle for two instructions). On the other hand, SP takes less for simpler operations (e.g., 0.5 clock cycles), compared with SS which takes longer (e.g., 1 clock cycle, if and when no clock with finer resolution is available). Latency is shorter with SP (shorter basic clock) and consequently the outcome of a simple branch test is known sooner, but the clock period is shorter with SS (no extra latches which are necessary for the SP approach). Resources are less duplicated with the SP approach, but its major problem is clock skew. What is better—SS or SP—depends on the technology and the application.
Figure ILPU11: Instruction timing in a superpipelined processor (source: [Johnson91])

Comment:

This figure shows the case when the basic clock is subdivided into two sub-clocks. In theory, the sub-division can go beyond the limit of two. In practice, it rarely goes beyond two, due to the increased complexity of hardware design and increased problems in software utilization.

Of course, the best performance/complexity ratio can be achieved using hybrid techniques which combine SS, SP, and VLIW. Theoretical limits on the ILP exploitation are determined by conditional branches which affects clock count and the extra complexity which grows quadratically and affects clock speed. According to several authors, practical limits of ILP exploitation are 8-way issue (if only hardware employed) and \( N \)-operation VLIW (if both hardware and compiler employed); \( N > 8 \).

Issues discussed so far will be briefly revisited through two different examples of two different superscalar microprocessors: MIPS R10000 and DEC Alpha 21164.

1.1. Example: MIPS R10000

Microprocessor MIPS R10000 is a 4-way superscalar machine. It fetches four instructions at a time from the I cache. Its block diagram is given in Figure ILPU12.
FP—Floating-point.

Comment:
Major units which distinguish this microprocessor from its competitors are the predecode unit and the reorder-and-commit unit. The first one helps about the compactness of the code, which is important for the hit ratio of the L1 cache. The second one helps about the out-of-order execution and about the maintaining of precise states at the times of exceptions.

Before being put into the I cache, the instructions get precoded, to help the decoding process. While in the cache, the code compactness is the issue. While in the decoder, the code decodability is the issue. In this context, compactness means less bits per instruction; decodability means less gate delays per decoding. In the case of R10000, the difference between the two forms is equal to four bits. Consequently, the extra logic is minimal (since the number of instructions in the decoding unit is much smaller than the number of instructions in the typical size I cache), and the decoding process can be done immediately upon the instruction fetch (from the I cache).

The branch prediction unit is placed next to the I cache and shares some features of the cache in order to speed up the branch prediction. The I cache has 512 lines and the branch prediction table (BPT) has 512 entries. Branch prediction table is based on 2-bit counters, as explained later on in this book. If a branch is predicted taken, instruction fetching is redirected after a cycle. During that cycle, instructions in the not-taken path continue to be fetched; they get placed into the cache called resume cache, to be ready if the prediction happens not to be correct. The resume cache is large enough for instructions of the not-taken paths related to four consecutive branches.

Major execution phases are instruction fetch, instruction decoding, operand renaming (based on 64 physical and 32 logical registers), and dispatching to appropriate instruction queues (up to four instructions concurrently dispatched into three queues, for later execution in five different functional units).

During the register renaming process, the destination register is assigned a physical register which is listed as unused in the so called free list. At the same time, another list is updated to reflect the new logical-to-physical mapping. If needed, an operand is accessed through the list of logical-to-physical mappings.

During the instruction dispatching process, each of the queues can accept up to four instructions. Also, reservation bit for each physical result register is set busy. Queues act like reservation stations holding instructions and physical registers designators acting as pointers to data. Global register reservation bits for source operands are being constantly tested for availability. As soon as all source operands become available, the instruction is considered free, and can be issued.

The five functional units are: address adder, two integer ALUs (one with a shifter and the other with an integer multiplier, in addition to basic adder and logic unit), floating point adder, and floating point multioperation unit (multiplication, division, and square root).

The on-chip primary cache is 32 KB large, 2-way set associative, and includes 32-byte lines. The off-chip secondary cache is typically based on the inclusion principle (if something is in the primary cache, it will also be in the secondary cache).

A reorder buffer is used to maintain precise state at the time of exception. Exception conditions for noncommitted instructions are held at the reorder buffer. Interrupt occurs whenever an instruction with an exception is ready to be committed.
1.2. Example: DEC Alpha 21164

Microprocessor DEC Alpha 21164 is also a 4-way superscalar machine. It fetches four instructions at a time from the I cache. Its block diagram is given in Figure ILPU13.

**Legend:**
FP—Floating-point.

**Comment:**
This microprocessor belongs to the group of speedemons; consequently, the execution unit includes only the blocks which are absolutely necessary; however, these blocks are designed aggressively (i.e., transistor count is often traded for faster execution), in order to achieve maximal clock speed.

Instructions are fetched and placed into one of two instruction buffers (each one is four instructions deep). Instructions are issued (from an instruction buffer) in the program order (not bypassing each other). One instruction buffer is used until emptied, and then the issuing from the next instruction buffer starts (a solution which makes the control logic much less complex).

Again, the branch prediction table is associated with the instruction cache. A 2-bit branch history counter is associated with each cache entry. Only one branch can be in the state when it is predicted but not yet resolved. Therefore, the issue is stalled on the second branch if the first one is not yet resolved.

After decoding, instructions are arranged according to the type of functional unit that they are to use. After the operand data are ready, instructions are issued to units that they are to be executed at. Instructions are not allowed to bypass each other.

This microprocessor includes four functional units: two integer ALUs (one for basic ALU operations plus shift and multiplication; the other for basic ALU operations and evaluation of branches), one floating point adder, and one floating point multiplier.

Two levels of cache memory reside on the CPU chip. The first level cache, direct-mapped for fast one clock access, is split into two parts: instruction cache and data cache. Both instruction cache and data cache are 8 KB large. The second level cache (three-way set-associative) is shared (joint instruction and data cache). Its capacity is 96 KB.

Primary cache handles up to six outstanding misses. For that purpose, a six entry MAF (miss address file) is included. Each entry includes the missed memory address and the target register address of the instruction which exhibits a miss. If MAF contains two entries with the same memory address, then the two entries will merge into one.

In order to make the handling of precise interrupts easier, the instruction issue is in order. The final pipeline stage in the integer units updates the destination registers also in order. By-
pass registers are included into the data path structure, so that data can be used before they are being written into their destination registers. The final bypass stages of the floating point units update registers out-of-order. Consequently, not all floating point exceptions result in precise interrupts.

1.3. Example: DEC Alpha 21264

The more recent microprocessor from the same family [Gwennap97], the DEC Alpha 21264, has the following characteristics: (a) 4-way issue superscalar, (b) out-of-order execution architecture, (c) speculative execution, and (d) multi-hybrid branch prediction.

The cycle time is 600 MHz, which makes it deliver an estimated 40 SPECint95 and 60 SPECfp95 performance. This was made possible because both L1 and L2 caches are on the processor chip. Also, the path to memory enables the data transfer rate of over 3 GBs/s.

2. Advanced Issues

This part contains the author’s selection of research activities which, in his opinion, have made an important contribution to the field in the recent time, and are compatible with the overall profile of this book.

Paper [Jourdan95] describes an effort to determine the minimal number of functional units (FUs) for the maximal speed-up of modern superscalar microprocessors. Analysis includes MC 88110, SUN UltraSparc, DEC Alpha21164 (IO superscalars), plus IBM 604 and MIPS R10000 (OO superscalars). Basic characteristics of modern superscalar processors are given in Figure ILPS1.

<table>
<thead>
<tr>
<th>Processor</th>
<th>PPC 604</th>
<th>MC 88110</th>
<th>UltraSparc 95</th>
<th>DEC 21164 95</th>
<th>MIPS 10000 95</th>
</tr>
</thead>
<tbody>
<tr>
<td>DateShip</td>
<td>94</td>
<td>92</td>
<td>95</td>
<td>95</td>
<td>95</td>
</tr>
<tr>
<td>Degree</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>IntegerUnit</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>ShiftUnit</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>DivideUnit</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>MultiplyUnit</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>FPAvailUnit</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ConvertUnit</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>FPDivideUnit</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>FPMultiplyUnit</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>DataCachePort</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure ILPS1: Basic characteristics of modern superscalar processors (source: [Jourdan95])

Comment:
The most crucial issue here is the number of cache ports. Real two-port cache means that the cache memory complexity is doubled, which is the case with DEC Alpha 21164. However, MIPS 10000 single-port cache, under certain conditions, can behave as a two-port cache. This is because the cache is designed to include two distinct modules. If the compiler is smart enough, it will schedule data fetches in such a way that the two single-port modules are always used concurrently.

Major conclusion of the study is that the number of FUs in modern superscalar microprocessor architectures needed to exploit the existing ILP is from 5 to 9, depending on the application. If the degree of superscaling is four or more, the number of data cache ports becomes the bottleneck, and has to be increased accordingly.
Conditions of the analysis imply the applications corresponding to the Spec92 benchmark suite, a lookahead window of the degree 2 to 8, and OO execution architecture in all considered cases (where not included into the original architecture, the OO capability is simulated). Effects of cache miss have not been studied (a large enough cache is implied). For details, the interested reader is referred to the original paper [Jourdan95].

Paper [Hank95] describes a research effort referred to as region-based compilation. The analysis is oriented to superscalar, superpipelined, and VLIW architectures. Traditionally, compilers have been built assuming functions as units of optimization. Consequently, function boundaries are not changed and numerous optimization opportunities get hidden. Example of an undesirable function-based code partition is given in Figure ILPS2.

**Figure ILPS2:** Example of an undesirable function-based code partition (source: [Hank95])

**Legend:** Numbers refer to task units.

**Comment:**
If the two function codes are merged together, and the code partition into functions is redone by the compiler, chances are that much better code optimization conditions can be created.

Solution offered by this research is that compiler is allowed to repartition the code into more desirable units, in order to achieve more efficient code optimization. Block structure of a region-based compiler is given in Figure ILPS3.

**Figure ILPS3:** Example of a region-based compiler, inlining and repartition into regions (source: [Hank95])

**Comment:**
The essential elements which distinguish this compiler from traditional ones are the classifier unit and the router unit.

Conditions of this research imply that compilation units (CUs) are selected by compiler, not by the software designer. Optimal size of the CU helps for better utilization of employed transformations. Profiler support is used to determine regions of code which enable better op-
timization. All experiments have been based on the Multiflow technology for trace scheduling and register allocation.

Paper [Wilson96] proposes and analyzes the approaches for increasing the cache port efficiency of superscalar processors. Major problem is that, on one hand, cache ports are necessary for better exploitation of ILP, and on the other hand, they are expensive. The load/store unit data path of a modern superscalar microprocessor is given in Figure ILPS4.

![Figure ILPS4: The load/store unit data path in modern SS microprocessors (source: [Wilson96])](image)

Legend:
- CAB—Cache Address Buffer.

Comment:
This structure supports four different enhancements: (a) load all—if two or more loads from the same address are present in the buffer, the data which returns from memory will satisfy all of them; this will not cause program errors, because the memory disambiguation unit already removed all loads which depend on not-yet-completed store instructions, (b) load all wide—it is easy to widen the cache port (which means that “load all” can be applied to a larger data structure), so that the entire cache line (rather than a single word) can be returned from cache at a time; this is obtained by increasing only the number of interface sense amplifiers, which is not expensive, (c) keep tags—if a special tag buffer is included (which holds tags of all outstanding data cache misses), newly arrived cache accesses which are sure to miss (because their addresses match some of the addresses in the tag buffer) can be removed from the cache access buffer, thus leaving room for more instructions which are potentially successful (i.e., likely to hit), (d) line buffer—data returned from the cache can be buffered in some kind of L0 cache, which is fully associative, multi-ported, and based on the FIFO replacement policy; if a line buffer contains the data requested by the load, the data will be supplied from the line buffer (good for data with a high level of temporal locality). Note that the four enhancements can be superimposed, to maximize the performance.

Solution offered by this research is to increase the bandwidth of a single port, by using additional buffering in the processor and wider ports in caches. Authors have proposed four different enhancements.

Conditions of this research imply split primary caches (two-way set-associative external 8 KB large caches), a unified secondary cache (two-way set-associative 2 MB large cache), and a main memory. Four on-going cache misses are enabled by four special purpose registers called MSHR (miss status handling registers). Concrete numerical performance results imply operating system SimOS and benchmark suite Spec95.

Research performed at the Polytechnical University of Barcelona [Gonzalez96] reports some very dramatic results. They have assumed the architecture of Figure ILPS5, and they
have tried to measure the real ILP in selected SPEC95 benchmarks. They have varied the parameters like: (a) reorder buffer size (1024, 2048, 4096, 8192, 16384, and infinite), (b) memory ordering (in-order load, out-of-order load), (c) number of memory ports (1, 2, 4, no restrictions), and (d) register pressure (64, 128, 256, 512, infinite register count). The width of the fetch engine has been chosen to be 64, which is much more than in the contemporary microprocessors.

![Block diagram of the simulated processor](source: [Gonzalez96]).

Legend:
IC—Instruction Cache;
F&D—Fetch and Decode;
RR—Register Renaming;
ROB—ReOrder Buffer;
RF—Register File;
EU—Execution Unit;
DC—Data Cache.

Comment:
Essentially, this is a traditional pipeline with two more stages included (register renaming and reorder buffer), as well as appropriate feedback links to these two new stages.

Surprisingly enough, an IPC (Instructions Per Clock) of 9 to 50 was achieved in an ideal machine with the infinite number of resources and no memory dependencies, as indicated in Figure ILPS6. If memory dependencies are taken into consideration, IPS drops down to the range from 4 to 19 for out-of-order load (Figure ILPS7), which is significantly more than what was reported in a number of previous studies. Impact of the number of memory ports is given in Figure ILPS8 and shows that 4-port memory should become mandatory as soon as permitted by technology. Impact of a smaller reorder buffer is given in Figure ILPS9. The register count limitation effects are summarized in Figure ILPS10, which advocated for 128 or even 256 flat (as opposed to windowed) registers.

![The IPC for various reorder buffer sizes and no memory dependencies](source: [Gonzalez96]).

Legend:
RB—Reorder Buffer.

Comment:
A dash is used to indicate that the value did not change, i.e. that the saturation point has been reached (saturation point is here defined as the value obtained by the infinite reorder buffer). Different benchmarks saturate at different reorder buffer sizes.
Figure ILPS7: The IPC for in-order and out-of-order load in the presence of memory dependencies (source: [Gonzalez96]).

Comment:
The column without memory dependencies has been repeated from the previous figure, so that the results can be compared. With most of the benchmarks, the drop in IPC is substantial.

<table>
<thead>
<tr>
<th></th>
<th>In order</th>
<th>Out-of-Order load</th>
<th>No dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Go</td>
<td>4.32</td>
<td>6.18</td>
<td>50.13</td>
</tr>
<tr>
<td>Compress</td>
<td>10.56</td>
<td>13.87</td>
<td>43.19</td>
</tr>
<tr>
<td>Vortex</td>
<td>5.17</td>
<td>5.76</td>
<td>28.80</td>
</tr>
<tr>
<td>Li</td>
<td>4.14</td>
<td>5.75</td>
<td>18.11</td>
</tr>
<tr>
<td>Fpppp</td>
<td>8.50</td>
<td>8.57</td>
<td>28.40</td>
</tr>
<tr>
<td>Applu</td>
<td>6.40</td>
<td>6.47</td>
<td>31.43</td>
</tr>
<tr>
<td>Wave5</td>
<td>13.81</td>
<td>18.56</td>
<td>18.56</td>
</tr>
<tr>
<td>Swim</td>
<td>4.16</td>
<td>4.83</td>
<td>16.71</td>
</tr>
<tr>
<td>Turb3d</td>
<td>9.03</td>
<td>9.03</td>
<td>9.03</td>
</tr>
</tbody>
</table>

Figure ILPS8: The IPC with 1, 2, 4, or infinitely many memory ports (source: [Gonzalez96]).

Comment:
For comparison purposes, the out-of-order load column has been repeated from the previous figure, as the case without any restrictions on the number of memory ports. No benchmark saturates at 2 memory ports, which is used in most contemporary microprocessors.

<table>
<thead>
<tr>
<th></th>
<th>1 Port</th>
<th>2 Ports</th>
<th>4 Ports</th>
<th>No dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Go</td>
<td>2.27</td>
<td>3.86</td>
<td>5.30</td>
<td>6.18</td>
</tr>
<tr>
<td>Compress</td>
<td>5.84</td>
<td>9.06</td>
<td>11.50</td>
<td>13.87</td>
</tr>
<tr>
<td>Vortex</td>
<td>2.12</td>
<td>3.63</td>
<td>4.94</td>
<td>5.76</td>
</tr>
<tr>
<td>Li</td>
<td>2.08</td>
<td>3.56</td>
<td>5.04</td>
<td>5.75</td>
</tr>
<tr>
<td>Fpppp</td>
<td>2.26</td>
<td>4.16</td>
<td>6.55</td>
<td>8.57</td>
</tr>
<tr>
<td>Applu</td>
<td>2.51</td>
<td>3.79</td>
<td>4.80</td>
<td>6.47</td>
</tr>
<tr>
<td>Wave5</td>
<td>5.22</td>
<td>9.82</td>
<td>18.55</td>
<td>18.56</td>
</tr>
<tr>
<td>Swim</td>
<td>2.93</td>
<td>3.84</td>
<td>4.54</td>
<td>4.83</td>
</tr>
<tr>
<td>Turb3d</td>
<td>4.46</td>
<td>8.94</td>
<td>9.01</td>
<td>9.03</td>
</tr>
</tbody>
</table>

Figure ILPS9: The IPC as a function of a limited size reorder buffer (source: [Gonzalez96]).

Comment:
Very few benchmarks saturate at 128 or 256 entry reorder buffer.

<table>
<thead>
<tr>
<th></th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>∞</th>
</tr>
</thead>
<tbody>
<tr>
<td>Go</td>
<td>4.29</td>
<td>5.43</td>
<td>6.02</td>
<td>6.17</td>
<td>6.18</td>
</tr>
<tr>
<td>Compress</td>
<td>4.11</td>
<td>5.37</td>
<td>6.92</td>
<td>10.12</td>
<td>13.87</td>
</tr>
<tr>
<td>Vortex</td>
<td>4.78</td>
<td>5.65</td>
<td>5.74</td>
<td>5.74</td>
<td>5.76</td>
</tr>
<tr>
<td>Li</td>
<td>4.39</td>
<td>5.42</td>
<td>5.75</td>
<td>5.75</td>
<td>5.75</td>
</tr>
<tr>
<td>Fpppp</td>
<td>5.83</td>
<td>7.17</td>
<td>7.45</td>
<td>7.61</td>
<td>8.57</td>
</tr>
<tr>
<td>Applu</td>
<td>4.37</td>
<td>5.25</td>
<td>5.51</td>
<td>5.57</td>
<td>6.47</td>
</tr>
<tr>
<td>Wave5</td>
<td>4.52</td>
<td>5.90</td>
<td>6.77</td>
<td>7.58</td>
<td>18.56</td>
</tr>
<tr>
<td>Swim</td>
<td>3.46</td>
<td>4.40</td>
<td>4.82</td>
<td>4.83</td>
<td>4.83</td>
</tr>
<tr>
<td>Turb3d</td>
<td>7.11</td>
<td>8.11</td>
<td>8.39</td>
<td>8.39</td>
<td>9.03</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>∞</th>
</tr>
</thead>
<tbody>
<tr>
<td>Go</td>
<td>4.74</td>
<td>5.97</td>
<td>6.17</td>
<td>6.18</td>
<td>6.18</td>
</tr>
<tr>
<td>Compress</td>
<td>5.45</td>
<td>8.81</td>
<td>12.55</td>
<td>13.87</td>
<td>13.87</td>
</tr>
<tr>
<td>Vortex</td>
<td>5.37</td>
<td>5.74</td>
<td>5.74</td>
<td>5.74</td>
<td>5.76</td>
</tr>
<tr>
<td>Li</td>
<td>4.98</td>
<td>5.75</td>
<td>5.75</td>
<td>5.75</td>
<td>5.75</td>
</tr>
<tr>
<td>Fpppp</td>
<td>6.46</td>
<td>7.43</td>
<td>7.61</td>
<td>7.75</td>
<td>8.57</td>
</tr>
<tr>
<td>Applu</td>
<td>4.93</td>
<td>5.53</td>
<td>5.58</td>
<td>5.63</td>
<td>6.47</td>
</tr>
<tr>
<td>Wave5</td>
<td>5.86</td>
<td>7.35</td>
<td>7.60</td>
<td>7.62</td>
<td>18.56</td>
</tr>
</tbody>
</table>
### Figure ILPS10: The IPC as a function of a limited physical register count (source: [Gonzalez96]).

**Comment:**
Register utilization is a function of the optimizing compiler characteristics.

The overall conclusion of the paper is that the available ILP is much longer compared to what current microprocessors can achieve. The major obstacle on the way to considerably better microprocessor performance are memory dependencies.

One of the major problems of the superscalar microprocessors in particular, and microprocessors in general, is their flexibility. Execution units are fixed in their design: they execute a given set of operations, on a given set of data types. Recent advances in FPGA (Field Programmable Gate Array) technology enable several reconfigurable execution units to be implemented in a superscalar microprocessor. Of course, full benefits of reconfigurability can be achieved only if appropriate compilers are developed.

A survey of reconfigurable computing can be found in [Villasenor97]. The first attempt at reconfigurable computing dates back to the late 60s proposal by Gerald Estrin of UCLA, which was highly constrained by technology capabilities of those days. The latest attempts include a number of designs oriented to FPGAs with over 100,000 logic elements, including the effort of John Wawrzynek of UC Berkeley. An earlier research effort of Eduardo Sanchez and associates at the EPFL in Lausanne, Switzerland starts from the concept of reconfiguration at program or program segment boundaries [Iseli95], and progresses toward reconfiguration at the single instruction boundary level. Such a development is made possible by the latest technology trends, like those by André Deffon and Thomas Knight at MIT, which imply FPGAs storing multiple configurations, and switching among different configurations in a single cycle (order of only tens of nanoseconds).

An important new trend in instruction level parallelism related research implies the techniques which optimize the system design and performance by turning to complexity reduction and dynamic speculation [Palacharla97, Moshovos97].

### 3. About the Research of the Author and His Associates

The research of the author and his colleagues is described in [Milutinovic87a] and [Milutinovic87b]. It is on the HLL oriented ILP. The research was performed at Purdue University (at winter time) and Montenegro Coast (at summer time) under the sponsorship of NCR World Headquarters.

The major goal of this research was to analyze ILP in the context of HLL environment requirements on one side (on average, more than one machine level operation per HLL statement) and GaAs technology requirements on the other side (high ratio of off chip and on chip delays in conditions when on-chip transistor count is relatively small).

In GaAs, going off chip to fetch a new instruction is relatively expensive (measured in time units); consequently, it pays off to fetch several operations at a time. This fits the requirements of GaAs technology where multiple simple execution units take longer to execute (simplicity of design traded off for speed). Also, the simpler the basic operations, the larger the number of sequential operations before a branch, which helps exploit better the available MLP.
Resulting machine, referred to as vertical migration microprocessor architecture, was designed and simulated to details. The term vertical migration reflects the fact that HLL primitives (statements with the minimal number of operations) can be mapped into the machine level instructions, using the one-to-one correspondence.
Branch Prediction Strategies

This chapter includes three sections. The first one is oriented to basic issues (background). The second one is oriented to advanced issues (state-of-the-art). The third one is oriented to the research of the author and his associates (basic ideas and pointers to references).

1. Basic Issues

A straightforward method to cope with negative effects of branching (as far as execution time) is multiple path execution. Machines like IBM 370 Model 168, and IBM 3033 were among the first to adopt this type of strategy. The strategy boils down to doubling of the resources used, at each branch encountered, prior to the first branch outcome resolved. After that, computation results from the wrong branch path are flushed, and the doubling is applied to the next branch in order.

The rest of the presentation is oriented to branch prediction algorithms based on the correlation principle. Correlation principle means that future outcome of a branch is usually predictable, based on past outcomes of the same or related branches. Branch prediction strategies of this type are classified as hardware, software, or hybrid.

Hardware BPS (branch prediction strategy) implies dynamic prediction based on BTB (branch target buffer) and/or BPB (branch prediction buffer). The later is sometimes referred to as BHT (branch history table). Each branch has two essential parameters: target address and branch condition. The first one is taken care of by the BTB, while the second one is taken care of by the BPB. Of course, as it will be seen later, in addition to BTB and BPB, appropriate additional resources are also necessary. All hardware BPS schemes assume that the outcome of a branch is stable over a period of time.

Software BPS implies static prediction based on pre-annotating techniques (to point to the parallelism to be exploited) and pre-arranging techniques (to detect and eliminate dependencies, in order to increase the level of ILP). All software BPS schemes assume that each HLL (high-level language) program context is characterized with typical outcomes of branches.

Hybrid BPS implies that hardware and compiler are cooperating together. The most popular approaches are based on predicated (conditional) and speculative (renaming) instructions. Note that some authors classify predicated and speculative instructions in different ways.
1.1. Hardware BPS

In the rest of the text, terms hardware BPS and dynamic BPS will be used interchangeably. Dynamic decisions in this context are typically done in hardware. Actually, each BPS can also be implemented (dynamically, at run-time) in software; however, in this particular environment, there is no time for the slower software implementation.

Typical BTB includes one entry per one recently executed branch instruction. That entry contains several fields: (a) branch instruction address—full address or only low-order bits, (b) one bit telling about the outcome of the branch instruction, and (c) address of the branch target.

The fact that only low-order bits can be used means that several branches may map into the same BTB entry; however, if the number of low-order bits is high enough, due to the locality principle, it is very likely that the information in the entry will refer to the right branch instruction.

The fact that only one bit is used to tell about the outcome of the branch instruction can be utilized with two purposes: (a) to eliminate all entries referring to recently executed branch instruction, for cases when the branch was not taken, and (b) to eliminate the one-bit entry telling about the outcome. Consequently, most BTBs include entries only for taken branches.

Typical BPB does not include the branch target address. This means less complexity, but a slow down, since the branch target address has to be looked-up elsewhere. Also, BTB must include entries for both taken and not-taken branches, which means more complexity if a joint BTB/BPB is to be designed, because such a unified resource must have entries for both taken and not-taken branches; consequently, some machines include both a BTB and a BPB.

The PowerPC 620 is one of the machines including both a BTB and BPB. Actually, PowerPC 620 includes a variation of BTB, which has one or more target instructions, instead of or in addition to the target address. Because of the added complexity, a BTB access takes longer, which may complicate the critical path design. However, this solution enables the effect referred to as branch folding. Branch folding means that unconditional branches always take zero cycles to execute (if BTB hit, the instruction is already in the CPU) and conditional branches may also take zero cycles to execute (assuming that condition evaluation is fast enough).

The CRISP machine includes another variation which enables branch folding for indirect branches. Their target address varies at run time; this complicates the conventional design, which implies one fixed target address per entry. The CRISP solution by Ditzel and McLellan starts from a statistical analysis saying that procedure returns contribute with about 90% to the total number of indirect branches. It includes a stack of return addresses, and return addresses are pushed/popped at call/return. The top of the stack is treated as the target address field of the BTB. Consequently, only the top-of-the-stack target address is used in conjunction with each specific branch. If the depth of the stack is larger or equal to the nesting level of procedures, the target address prediction will be absolutely correct.

The DLX machine by Hennessy and Patterson uses the same solution. It includes a return buffer for the nesting depth of up to 16. This number was obtained from statistical analysis of SPEC89, aimed at the optimal complexity/performance ratio.

The simplest predictor is referred to as the 2-bit predictor, introduced by Jim Smith. It is shown in Figure BPSU1. The 2-bit predictor yields a considerably better performance than the one-bit predictor. This is because the 2-bit predictor mispredicts only at one of the two branches forming a loop, while the one-bit predictor mispredicts at both branches. A three-bit
predictor yields a performance which is only slightly better, at the 50% larger cost. Consequently, only 2-bit predictors make sense, both in the case of the simplest predictor of Figure BPSU1, and the more complex predictors to be discussed later.

![Figure BPSU1: States of the 2-bit predictor; avoiding the misprediction on the first iteration of the repeated loop (source: [Hennessy96])](image)

**Legend:**
Nodes—States of the scheme,
Arcs—State changes due to branches.

**Comment:**
State 11 means branch very likely. State 10 means branch likely. State 01 means branch unlikely. State 00 means branch very unlikely. Note that two (rather than three) mispredictions move the state machine from 11 to 00, and vice versa. The best initial state is “branch likely,” because a randomly chosen branch instruction is more likely to branch then not, but not very likely to branch. A condition-controlled loop is typically executed several times. Last execution of the loop condition statement must result in a misprediction (that misprediction can not be avoided, unless a special loop count estimation algorithm is incorporated, like the one in [Chang95]). If a one-bit predictor is used, the predictor bit gets inverted on the misprediction, and the first next execution of the loop condition statement will also result in a misprediction (in spite of the fact that loops are very unlikely to execute only once). However, if a 2-bit predictor is used, an “inertia” is incorporated into the system, and the predictor needs two mispredictions before it switches from 11 to 00. If average loop count is N, the 2-bit predictor, compared to the one-bit predictor, has the misprediction which is at least \((2/N - 1/N)\)% better (a number which justifies the exclusive use of 2-bit predictors). On the other hand, a three-bit predictor brings a negligible performance improvement and a 50% complexity increase, compared to the 2-bit predictor (another fact which fully justifies the exclusive use of the 2-bit predictor).

Prediction accuracy of a BTB depends on its size, as indicated in Figure BPSU2. This figure shows that before the saturation point is met, logic design strategy of the BTB may help, since the set-associative approach works somewhat better than the direct-mapped approach. However, after the saturation point is reached, logic design strategy does not matter any more. This figure also shows that the saturation point is reached at about 2K entries in the BTB. Several other studies claim that a 4K-entry BTB is about equally as good as an infinite BTB. Note that it is not sufficient that the branch instruction is located in the BTB; what is also important is that the prediction must be correct. In other words, in real designs, the cost of a misprediction is typically the same as the cost in the case of a BTB miss!
Figure BPSU2: Average branch-target buffer prediction accuracy; a BTB with 2K entries is about the same in performance as an infinite BTB (source: [Johnson91])

Legend:
CPB—Percentage of Correctly Predicted Branches,
4WSA—4-Way Set-Associative,
DM—Direct Mapped,
NE—Number of Entries.

Comment:
It is important to notice that the set-associative approach is better than the direct-mapped approach only if the BTB is too small, or not large enough.

Another solution is presented in Figure BPSU3, where the prediction related information is included into the instruction cache, in addition to the standard information, which is the addressing information and the code. Prediction related information includes two fields: (a) successor index field, and (b) branch entry index field.

Figure BPSU3: Instruction Cache Entry for Branch Prediction (source: [Johnson91])

Legend:
I—Cache entry.

Comment:
Efficiency of the approach depends on the number of instructions in each cache entry. Other approaches to the incorporation of prediction into cache memory are possible, too. Their elaboration is left as an exercise for the students.

The successor index field contains two subfields: (a) address of the next cache entry predicted to be fetched, and (b) address of the first instruction (in that entry) which is predicted to be executed. The length of the successor index field depends on cache size (the number of instructions which fits into the cache). Relative size of the two subfields depends on the number of instructions per cache entry. For example, a 1MB direct-mapped cache for a machine with 64-bit instructions, and 8 instructions per cache entry, requires a 17-bit successor index field.
(N₁ = 14 bits to address the cache entry and N₂ = 3 bits to address an instruction within the entry). This is so because a 1MB cache holds 128K 8-byte (or 64-bit) instructions, and these instructions are organized in 16K cache entries (8 instructions per cache entry). To address one of the 16K cache entries, one needs 14 bits. To address one instruction within an entry, one needs 3 bits.

The branch entry index field specifies the location (within the current cache entry) of the branch instruction which is predicted to be taken. Consequently, instructions beyond the branch point are predicted not to be executed.

This organization represents a way to incorporate the prediction related information into the cache. The successor index field specifies where to start the prefetching from (instructions before the one pointed to by the successor index field are not needed). The branch entry index field specifies the branch instruction that the prediction information refers to (instruction beyond the one pointed to by the branch entry index field is likely not to be executed).

An improvement of 2-bit predictors is referred to as two-level predictor (a more accurate name would be two-level 2-bit predictor, since 2-bit predictor is an element of a two-level predictor). The scheme was introduced by Yeh and Patt. Two-level predictors use the information on the behavior of other branches (at other addresses), in order to do prediction about the currently executing branch instruction (at the current address). Practically always (although theoretically not always) two-level predictors show better performance compared to 2-bit predictors. A two-level 2-bit (2,2) branch predictor is shown in Figure BPSU4.

![Figure BPSU4](image)

**Figure BPSU4:** A (2,2) Branch Predictor using a 2-bit global history to select one of the four 2-bit predictors (source: [Hennessy96])

**Legend:**
GBH—Global Branch History,
PO—Prediction Outcome.

**Comment:**
Global branch history register of the size 9 or 10 bits, and the size of one single (vertical) vector branch predictor equal to 2048 or 4096 entries, seems to be a good price/performance compromise.

The example (2,2) branch predictor from Figure BPSU4 includes four vectors of 2-bit predictors. The first one is for the case when the most recent two branches (on any two addresses) were (0,0), the second one is for the case when the most recent two branches (on any two addresses) were (0,1), etc. In this context, (0,0) means that the most recent two branches (on any addresses) were not taken, etc. Information about the outcome of the most recent
branches (on any addresses) is kept in a register called GBH (global branch history). Therefore, for a given branch, prediction depends both on the address of the branch (horizontal entry into the matrix of 2-bit predictors) and the contents of the GBH register (vertical entry into the matrix of 2-bit predictors).

The described example belongs to the category of global predictors, because there is only one GBH, and its contents refers to most recent branches at any addresses. It will be seen later that one can also talk about per-address predictors, where the number of GBH registers is equal to the number of entries in each vertical vector of 2-bit predictors.

The basic rationale behind global predictors is explained using the code example from Figure BPSU5a and the explanation from Figure BPSU5B. Figure BPSU5a includes a section of HLL (high-level language) code and its MLL (machine-level language) equivalent. From the explanation in Figure BPSU5b one can see that if previous branch (B1) is taken, there is a high probability that the next branch (B2) will also be taken. Branches B1 and B2 are related through the semantics of the code, and that is what global predictors rely on.

HLL:
if \( (d = 0) \)
   \(d = 1; \)
if \( (d = 1) \)

MLL (d assigned to \( r_1 \)):
   \texttt{bnez } \( r_1, \ l_1 \) ; branch \( B_1 \) (\( d \neq 0 \))
   \texttt{addi } \( r_1, r_0, #1 \) ; \( d = 0, \text{so } d \leftarrow 1 \) (note: \( [r_0 = 0] \))
   \texttt{l1: subi } \( r_3, r_1, #1 \)
   \texttt{bnez } \( r_3, l_2 \) ; branch \( B_2 \) (\( d \neq 1 \))
   ...

\( l_2: \)

**Figure BPSU5a**: Example code (source: [Hennessy96])

**Legend:**
HLL—High-Level Language,
MLL—Medium-Level Language.

**Comment:**
This example implies that register \( r_0 \) is hardwired to zero, which is typical for a number of RISC microprocessors.

<table>
<thead>
<tr>
<th>Dinit</th>
<th>( D = 0? )</th>
<th>( B_1 )</th>
<th>Dbefore( B_2 )</th>
<th>( D = 1? )</th>
<th>( B_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Yes</td>
<td>Not taken</td>
<td>1</td>
<td>Yes</td>
<td>Not taken</td>
</tr>
<tr>
<td>1</td>
<td>No</td>
<td>Taken</td>
<td>1</td>
<td>Yes</td>
<td>Not taken</td>
</tr>
<tr>
<td>2</td>
<td>No</td>
<td>Taken</td>
<td>2</td>
<td>No</td>
<td>Taken</td>
</tr>
</tbody>
</table>

**Figure BPSU5b**: Example explanation—illustration of the advantage of a two-level predictor with one-bit history;
if \( B_1 \) is not\_taken, then \( B_2 \) will also be not\_taken, which can be utilized to achieve better prediction (source: [Hennessy96])

**Legend:**
Dinit—Initial \( D \),
Dbefore\( B_2 \)—Value of \( D \) before \( B_2 \).

**Comment:**
It is important to underline that a number of other program segments would result into the same pattern of taken/not-taken relationship, which is a consequence of programming discipline and compiler design. In other words, parts of code which are semantically unrelated can be strongly related as far as the prediction related issues.
In general, one can talk about an \((M,N)\) predictor; it uses the behavior of the last \(M\) branches to select one of the \(2^M\) predictor vectors of the length \(L\), each one consisting of \(N\)-bit predictors. One study claims that it is not unreasonable that \(M\) goes all the way up to \(M = 9\). Previous discussion argues that it does not make sense to go beyond \(N = 2\).

Note that each 2-bit predictor corresponds to a number of different branches, because only a subset of address bits is used to access different 2-bit predictors in one vertical vector of 2-bit predictors. Consequently, the prediction may not correspond to the branch currently being executed, but to another one with the same set of low-order address bits. However, most of the time, the prediction will correspond to the branch currently being executed, because of the locality principle. In some cases, the prediction will not correspond to the branch currently being executed, but the scheme will still work, because the same programming discipline results in a similar pattern of branchings/nonbranchings.

Figure BPSU6a represents another viewpoint of looking at the above mentioned types of branch predictors. Different viewpoints enable the reader to create a better understanding of the issue.
Figure BPSU6a: Schemes 2bC, GAs, and PAs (source: [Evers96])

Legend:
PHT—Pattern History Table;
BHSR—Branch History Shift Register.

Comment:
Common characteristic of all three schemes is that the stress is on increased performance, rather than decreased complexity.
The basic scheme is now referred to as 2bC, as it is named in a number of research papers. It is seen that only a subset of $n$ branch address bits is used to access the single vector of $2^n$ 2-bit registers ($n = \log L$). The cost (complexity) of the scheme is given by:

$$C(2bC) = 2 \cdot 2^n \text{ bits.}$$

The global scheme is now referred to as GAs, as it is named in a number of research papers. Again, only a subset of lower $n$ address bits is used to access one of the vertical $2^m$ vertical branch prediction vectors ($m = M$). The one to be accessed is determined by the contents of the BHSR (branch history shift register). Terms BHSR and GBH refer to the same thing. One horizontal vector of 2-bit predictors is referred to as PHT (pattern history table). The cost of the scheme is given by:

$$C(GAs) = m + 2^{m+n+1} \text{ bits.}$$

The per-address scheme is here referred to as PAs, as it is named in a number of research papers. Everything is the same, except that each PHT is associated with a different BHSR. Consequently, the cost of the scheme is given by:

$$C(PAs) = m \cdot 2^n + 2^{m+n+1} \text{ bits}$$

assuming the same number of local branch history registers and rows in PHT.

Figure BPSU6b covers four different schemes of a much smaller cost, and (most of the time) of only a slightly lower performance. The simplification is based either on an interesting run-time property or on an efficient compile-time effort.
Cost (bits) = \( m \cdot 2^n + 2^{n+1} \)

Cost (bits) = \( m \cdot 2^n + 2^{n+1} \)

Cost (bits) = \( m \cdot 2^n + 2^{n+1} \)

Cost (bits) = \( m \cdot 2^n + 2^n \)

Figure BPSU6b: Schemes gshare, pshare, GSg, and PSg (source: [Evers96])

Legend:
BHSR—Branch History Shift Register.

Comment:
Common characteristic of all three schemes is that the stress is on decreased complexity, rather than increased performance.

In the case of gshare (introduced by McFarling—for details see [McFarling95]), the lowest \( n = m \) bits of the branch address are exclusive ORed with the contents of the single global m-
bit BHSR. The obtained value points to one of $2^m$ 2-bit predictors. The exclusive OR operation enables that each vertical vector from previous schemes gets substituted with a single 2-bit predictor. Consequently, the complexity of the scheme drops down considerably. Fortunately, empirical studies say that performance of the scheme drops down only slightly, due to the above discussed code locality and programming culture issues. The cost of this scheme is given by:

$$C(\text{gshare}) = m + 2^{m+1} \text{ bits.}$$

In the case of pshare, the lowest $n = m$ bits of the branch address are exclusive ORed with the per-address BHSR corresponding to the same address as the branch instruction being currently executed. Basically, everything is the same, except that the global treatment is substituted by the per-address treatment. The cost of this scheme is given by:

$$C(\text{pshare}) = m \cdot 2^m + 2^{m+1} \text{ bits.}$$

In the case GSg, predictors are based on compile-time prediction (rather than run-time prediction). Consequently, predictors are one-bit wide (rather than 2-bits wide). Scheme GSg is a global scheme. This means that the address of the current branch is not relevant. What is relevant is the contents of the $m$-bit BHSR, which points to one of the $2^m$ one-bit predictors. This scheme only has a theoretical value. It does not have any practical value, in spite of its extremely low cost function:

$$C(\text{GSg}) = m + 2^m \text{ bits.}$$

Finally, in the case of PSg (introduced, in various forms, independently by Lee and Smith, by Sechrest, and by Patt), the lowest $n$ bits of the branch address are used to select one of the $2^n m$-bit BHSRs of the per-address type. The selected m-bit BHSR is used to point to one of the one-bit predictors, inside the array of one-bit predictors; the length of this array is $2^m$. This scheme has some practical value, and is used in low-cost and hybrid schemes. Its cost is given by:

$$C(\text{PSg}) = m \cdot 2^m + 2^m \text{ bits.}$$

Note that the warm up time of the schemes based on one-bit predictors is equal to zero; this characteristic has been utilized in some hybrid branch predictors to be discussed later.

### 1.2. Software BPS

In the rest of the text, terms software BPS and static BPS will be used interchangeably. Static decisions in this context are typically done at compile-time. Actually, each static decision can also be implemented (dynamically, at run-time) inside the operating system; however, in this particular environment, there is no time for the slower operating system execution.

Static branch prediction schemes (in the narrow sense) use the information which is gathered before program execution, either by static code inspection or by appropriate profiling.

The simplest schemes in this category assume that conditional branches are either always taken (as was the case with the Stanford MIPS-X experimental RISC machine), or always not taken (as was the case with the Motorola MC88000 commercial RISC machine).

A more sophisticated solution has been used in the PowerPC architecture; compiler can use two types of branches—one which provides better execution time if the branch is taken and one which provides better execution time if the branch is not taken. In this context, compiler can consult a profiler, before the appropriate conditional branch instruction is scheduled.
Static branch prediction schemes (in the wide sense) include also those based on pre-annotating, software scheduling, local and global code motion, trace scheduling, loop unrolling, software pipelining, and other more sophisticated techniques. All these schemes should only conditionally be treated as static branch prediction schemes. Often, their primary goal is to eliminate branches or to minimize penalties of wrong decision, rather than to predict them; however, the final consequence may be the same.

Pre-annotating implies the insertion of special instructions, to help about the predefined goal, either at coding time, or at profiling time, or at compile time.

Software scheduling implies rearrangement of code at compile time, with hints from the programmer or the profiler. The motion is either local (within the basic block boundaries) or global (across the basic block boundaries). Basic block starts at one of the three points: (a) at the very beginning of the code, (b) at the instruction following a branch, and (c) at the labeled instruction—labeled instruction is a possible target of a branch. Basic block ends at one of the following three points: (a) at the very end of the code, (b) at the branch instruction—the branch instruction itself is treated as being a part of the basic block it terminates, and (c) at the instruction immediately before the labeled instruction. Each code rearrangement must be accompanied by appropriate code compensations, so that semantic structure of the code is not violated, no matter which way the execution proceeds.

Trace scheduling is the principal technique used in conjunction with VLIW (very large instruction word) architectures. Here, global code motion is enhanced with techniques to detect parallelism across conditional branches, assuming a special type of architecture (VLIW).

Loop unrolling is a technique used on any type of architecture, to increase the amount of sequentially executable code, i.e. to increase the size of basic blocks. Although the technique can be used in conjunction with any type of architecture, it gives the best results in conjunction with VLIW architecture and trace scheduling.

Software pipelining (symbolic loop unrolling) is a technique to pipeline operation from different loop iterations. Each iteration of a software pipelined loop includes instructions from different iterations of the original loop.

Note that many of the software techniques can be implemented in hardware, directly or indirectly. For example, the Tomasulo algorithm is a hardware equivalent of the software pipelining algorithm.

Software BPS will not be further elaborated here. For more information on these issues, the interested reader is referred to specialized literature.

1.3. Hybrid BPS

Hybrid BPS includes techniques with elements of both hardware BPS and software BPS. Since the hardware/software boundary is often ambiguous, the classification to hardware, software, and hybrid schemes has to be considered conditionally.

In this book, predicated instructions and speculative instructions are treated as the basic two approaches to hybrid BPS. However, in other sources, these two approaches are classified differently, and/or other possible approaches are treated as a part of the hybrid BPS group.

1.3.1. Predicated Instructions

Predicated instruction includes a condition which is evaluated during its execution. If the condition is evaluated true—normal execution proceeds. If the condition is evaluated false—noop execution proceeds (a cycle is wasted). Actually, this is an if-then construct with a mini-
mal body (one instruction body). Efficient utilization of predicated instructions requires the compiler and the hardware to cooperate, and that is why the predicating is here classified into the hybrid approaches.

The most common predicated instruction in modern microprocessors is predicated register-to-register move. However, predicated instructions are not a new invention. A form of predicated instruction can be found even in the first microprocessor of the x86 series—in the Intel 8086. The Intel 8086 instruction set includes the conditional REP prefix which can be treated as a primitive form of predicating.

Predicated instructions help to eliminate branches in some contexts: (a) where one has if-then with minimal body, or (b) where the code can be rearranged to create if-then with minimal body. These contexts happen in numerical codes (for example, when the absolute value is to be computed) and in the symbolic code (for example, when the repetitive search is to be applied).

Usefulness of predicated instructions is limited in a number of cases: (a) when the moving of a predicated instruction across a branch creates a code slow down; this is because cancelled instruction does take execution cycles, (b) when the moment of condition evaluation comes too late; the sooner the condition is evaluated, the better the speed of the code, (c) when the clock count of predicated instructions is too large; this happens easily in some architectures, and (d) when exception handling may become a problem, due to the presence of a predicated instruction.

Still, as indicated above, it is rather a rule than an exception, that modern microprocessor architectures do include predicated instructions. The above mentioned predicated register-to-register move is included into the architectures like DEC Alpha, SGI MIPS, IBM PowerPC, and SUN Sparc. The PA (Precision Architecture) approach of HP is that any register-to-register instruction can be predicated (not only the move instruction).

1.3.2. Speculative Instructions

Speculative instruction is executed before the processor knows if it should execute or not, i.e. before it is known if the prior branch instruction is taken or not. Control unit and optimizing compiler in microprocessors which implement speculative execution support the following scenario: (a) First, branch is predicted, (b) Second, the next instruction—either the target address instruction or the next address instruction, depending on the outcome of the prediction—is made speculative and is executed, (c) Third, some run-time scheduling is done, in order to optimize the code, i.e., to increase the efficiency of speculation, and (d) Four, if prediction was a miss, the recovery action is invoked and completed.

Some of the actions defined above are hardware responsibility, others are compiler (or, in principle, even operating system) responsibility; still the others are either a combined responsibility or can be treated one way (hardware) or the other way (compiler). That is the reason why this book classifies speculating into the hybrid group.

There are two basic approaches to speculation: (a) Compiler schedules a speculative instruction and hardware helps recover, if it shows up that speculation was wrong (here, speculation is done at compile time), and (b) Compiler does a straightforward code generation and the branch prediction hardware is responsible for speculation (here, speculation is done at run time).

Exception handling is less critical with speculative instructions, compared to predicated instructions (remember, the previous section specifies that exception handling is one of the limiting factors of predicated execution).
In principle, there are two types of exceptions: (a) program errors or similar, when exception causes the program to terminate, and (b) page faults or similar, when exception causes program to resume. No matter which type of exception is involved, in principle, there are three techniques which can be used to handle the problem, if speculation is used in a microprocessor architecture. However, only the first technique (to be explained next) is directly applicable to both types of exceptions. The other two techniques are directly applicable only to the second type of exception.

The three techniques used to handle exceptions in architectures with speculation, when a speculative instruction causes an exception, are: (a) Hardware and/or operating system are expanded with constructs to handle the exception; these constructs are invoked each time an exception happens, (b) Each register (in the set of general purpose registers) is expanded with a set of special status bits called poison bits—these bits are set whenever a speculative instruction writes a register; also, these bits are reset when the instruction no longer has the speculative status; at the time of exception, appropriate poison bits are tested, and a fault is generated if some other instruction, beyond the speculative instruction, selects a “poisoned” register for read, and (c) The architecture is expanded with a mechanism called “booster;” it is essentially a type of hardware renaming mechanism, and it is responsible for moving instructions past the branches, labeling each instruction during the period while it is speculative, and guarding the results of the labeled instructions inside a special purpose renaming buffer (in this way, results of speculative instructions are kept in a special purpose register file, and poison bits are not needed as a part of the general register file).

Speculation can be implemented in hardware or in software.

Hardware-based speculation is complex-to-design (which is not the problem for high production volume microprocessors) and transistor count consuming (which is not the problem in conditions when one has to find efficient use for the growing number of transistors on a single VLSI chip). On the other hand, hardware-based speculation offers the speed advantage (which is so important for the microprocessor market) and the reusability advantage (which is so important for fast design of new generations of microprocessors). Consequently, hardware-based speculation has been the solution of choice for modern microprocessors.

It is interesting to note that hardware based speculation has its roots in the research of 60s. Hardware based speculation was used in a number of CDC and IBM commercial products of those days.

The Thornton’s scoreboarding algorithm (developed for CDC 6600) is an early approach to dynamic scoreboarding. It does not include speculation, but it can be modified to include speculation.

The Tomasulo’s renaming algorithm (developed for IBM 360/91) is also an early approach to dynamic scoreboarding. It includes a primitive form of speculation, and it can be modified to include very sophisticated forms of speculation.

This book assumes that the reader is fully introduced into the details of the Thornton’s and the Tomasulo’s algorithm. If not, he/she is referred either to the original papers [Thornton64 and Tomasulo67] or to a well-known textbook [Hennessy96].
2. Advanced Issues

This part contains the author’s selection of research activities which, in his opinion, (a) have made an important contribution to the field in the recent time, and (b) are compatible with the overall profile of this book.

Numerous research papers describe efforts to combine two or more branch prediction strategies, in order to increase the probability of prediction. Such predictors, referred to as hybrid predictors, include two or more of the predictors described in the sections on branch predictors, plus a selection mechanism.

McFarling has introduced a dynamic (run-time) selection mechanism for a two-component hybrid BPS (it is referred to as branch selection); at run time, for each branch, it uses a 2-bit up-down saturating counter, to determine which of the two component predictors is more appropriate to use in the case of that specific branch; this selection mechanism can be easily expanded for the case of a multi-component hybrid BPS.

Chang, Hao, Yeh, and Patt have introduced a static (compile-time) selection mechanism for multi-component hybrid BPS (it is referred to as branch classification); at compile time, for each individual branch instruction, the branch predictor is determined which suits that particular branch the best. Consequently, each component branch predictor does prediction for the branches for which it is best suited.

In a follow up research effort, using a profiler, their compiler classifies branches into three categories: (a) mostly-taken, (b) mostly-not-taken, and (c) others. Authors propose the use of their compile time selection mechanism (branch classification) for the first two groups of branches, and the McFarling’s run time selection mechanism (branch selection) for the third group of branches.

Multi-component branch predictors cost more (very few resources in each individual component predictor can be efficiently shared among several component predictors); however, they perform better. The question is, how much better, in realistic conditions, when periodic context switches impact the behavior of individual predictors. The most efficient predictors need lots of time to warm up. On the other hand, predictors with zero or short warm up times are not nearly as efficient at steady state, as is the case with the most efficient but slowly warming predictors.

Obviously, the question is what component predictors to select, how to combine them for the best possible overall performance, and how to obtain reliable information on the overall performance, in conditions of periodic context switches. Several papers are dedicated to these issues.

A paper by Evers, Chang, and Patt introduces the multi-hybrid BPS, referred to here as MH-BPS. For the same implementation cost, it provides better performance compared to some two-component hybrid BPS schemes.

As indicated in Figure BPSS1, it is based on an array of 2-bit up-down predictor selection counters (PSC) similar to those from the McFarling’s branch selection mechanism, except that McFarling includes two 2-bit selection counters per branch and Evers/Chang/Patt include N 2-bit selection counters per branch; each BTB entry is extended with one PSC.
**Figure BPSS1:** Predictor Selection Mechanism (source: [Evers96])

**Legend:**
- BTB—Branch Target Buffer,
- PSC—Prediction Selection Counter,
- P₁—Predictor 1,
- Pₙ—Predictor n,
- PE—Priority Encoding,
- PO—Prediction Outcome.

**Comment:**
The priority encoding mechanism can be made flexible, so that different priority encoding algorithms can be evaluated (a good exercise for students).

Initial value of all PSC entries in Figure BPSS1 is 3, and a priority logic is used is several predictors are equal (see the order of component predictors in Figure BPSS2). If, among the predictors for which the contents of the PSC was 3, at least one was correct, the PSCs for all incorrect predictors were decremented. If none of the predictors with PSC = 3 was correct, the PSCs of all correct predictors are incremented. This algorithm guarantees that at least one PSC is equal to 3. Complexity of the described multi-hybrid selection mechanism is 2CL, where L is the number of entries in the BTB, and C is the number of component predictors.

Figure BPSS2 defines two things: (a) the priority order for the included component predictors—2bC has the highest priority and AlwaysTaken has the lowest priority, and (b) the component cost for each component predictor included into a given version of the MH-BPS. Different versions of MH-BPS differ in the overall bit count. The simplest version of MH-BPS from Figure BPSS2 includes 11KB. The most complex version of MH-BPS includes 116 KB. If one bit takes four transistors, the overall transistor count is in the range of about 350 kTr to about 3.5 MTr, which is more than the entire Intel Pentium.

<table>
<thead>
<tr>
<th>HybPredSiz [kB]</th>
<th>~11</th>
<th>~18</th>
<th>~33</th>
<th>~64</th>
<th>~116</th>
</tr>
</thead>
<tbody>
<tr>
<td>CompCost [kB]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SelectMech</td>
<td>2</td>
<td>2.5</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2bC</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>GAs</td>
<td></td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Gshare</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>Pshare</td>
<td>4</td>
<td>5.25</td>
<td>7.5</td>
<td>20</td>
<td>36.25</td>
</tr>
<tr>
<td>loop</td>
<td></td>
<td></td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>AlwaysTaken</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Figure BPSS2:** Multi-Hybrid Configurations and Sub-Optimal Priority Ordering 95.22/95.65 (source: [Evers96])

**Legend:**
- HybPredSiz—Hybrid Predictor Size,
The research of [Evers96] demonstrates that the optimal priority encoding algorithm provides the hit ratio which is only slightly larger compared to the hit ratio provided by the priority encoding algorithm which is selected in [Evers96]—95.22% versus 95.65%.

The quoted numbers set the stage for a small digression discussing the overall complexity of the MH-BPS. As already indicated, the complexity of 116 KB means more than the transistor count of the most single chip microprocessors from 80s. However, the transistor count of VLSI chips keeps growing, and their use has to be defined. If the performance improvement provided by MH-BPS is higher than the performance improvement in case of other possibilities, then the MH-BPS may become a standard element of many future microprocessors on the chips with more than 10 MegaTransistors. Therefore, the question boils down to the exact performance benefit of the MH-BPS. The answer can be found in [Evers96]. For the predictor size of approximately 64 KB, the MH-BPS achieves prediction accuracy of 96.22%, compared to 95.26% for the best two-component BPS of the same cost. At the first glance, this difference does not look spectacular (only 0.96%, or less than 1%). However, what matters is the difference in misprediction percentage, which drops from 4.74% to 3.78%, or over 25%. It is the misprediction which is costly and takes away the cycles of the execution time. Therefore, at the second glance, the difference does look spectacular, and places this research among the most exciting ones.

Note that the above data are obtained for SPECint92 (which is, according to many, not a large enough application suite, for this type of research, and includes user code only), and for the systems in which the contents of the prediction tables is flushed after periodical context switches (which is, according to many, not the optimal way to treat prediction tables after periodical context switches).

The table in Figure BPSS2 tells that the multi-hybrid predictor of 11 KB includes, in addition to the selection mechanism of about 2 KB, the following component predictors: 2bC, Gshare, Pshare, and AlwaysTaken. The multi-hybrid predictor of 116 KB includes a somewhat more costly selection mechanism of about 3 KB, plus the following component predictors: 2bC, GAs, Gshare, Pshare, loop, and AlwaysTaken (“loop” is a simple scheme which predicts the number of iterations for each loop-branch—for details see [Chang95]).

Authors claim that the selection of component predictors was guided by the following rationales: (a) large dynamic predictors have better accuracy at steady-state, but longer warm up time after context switch, (b) smaller dynamic predictors have worse accuracy at steady-state, but shorter warm up time after context switch, which means better accuracy during the initial period after context switch, (c) static predictors have zero warm up time, which means the best accuracy for a very short period immediately after context switch, and (d) a price/performance analysis has eliminated the predictors not included into the table of Figure BPSS2, due to their marginal price/performance.

All predictors taken into consideration in this study are summarized in Figure BPSS3, from a point of view which is different compared to the previous presentation of the same facts (more points of view brings much deeper understanding of the issues).
<table>
<thead>
<tr>
<th>Predictor</th>
<th>Description</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSg(m)</td>
<td>A modified version of the per-address variation of the Two-level Adaptive Branch Predictor consisting of 2K m-bit branch history registers and a single pattern history table (each PHT entry uses one statically determined hint bit instead of a 2bC). The version of PSg used in this study is the PSg(algo).</td>
<td>$2^{11}m + 2^m$</td>
</tr>
<tr>
<td>gshare(m)</td>
<td>A modified version of the global variation of the Two-level Adaptive Branch Predictor consisting of a single m-bit global branch history and a single pattern history table.</td>
<td>$m + 2^{m+1}$</td>
</tr>
<tr>
<td>pshare(m)</td>
<td>A modified version of the per-address variation of the Two-level Adaptive Branch Predictor consisting of 2K m-bit branch history registers and one pattern history table. As in the gshare scheme, the branch history is XORed with the branch address to select the appropriate PHT entry.</td>
<td>$2^{11}m + 2^{m+1}$</td>
</tr>
<tr>
<td>loop(m)</td>
<td>An AVG predictor where the prediction of a loop’s exit is based on the iteration count of the previous run of this loop. A 2K entry array of two m-bit counters is used to keep the iteration counts of loops. In this study, $m = 8$.</td>
<td>$2^{12}m$</td>
</tr>
</tbody>
</table>

**Legend:**
- **Always Taken**—Branch Always Taken (MIPS 10000),
- **Always Not Taken**—Branch Always Not Taken (Motorola 88110).

**Comment:**
This figure includes alternative descriptions of the branch prediction algorithms covered in this book. Receiving information from different sources is an important prerequisite for better understanding of essential issues.

The study in [Gloy96] is based on the IBS traces which include both system and user code (as indicated before, SPECint92 includes only user code) and a larger number of static branches (which means a more realistic environment). It also analyzes the systems in which the prediction tables are not flushed after the periodic context switches (which many believe is a better way to go, since different contexts are coded by the programmers using the same software design methodologies, and consequently produce code with similar run time prediction related characteristics).

The study by Gloy, Young, Chen, and Smith from Harvard University implies the model of BPS shown in Figure BPSS4 (another way of representing the same model), and it includes the component predictors shown in Figure BPSS5. The impact of zero warm up time and short warm up time predictors is smaller if no flushing is involved. This fact had an impact on the selection of component predictors shown in Figure BPSS5.
Comment:
This figure includes an alternative method of modeling the branch prediction system covered in this book. Receiving information from different sources is an important prerequisite for better understanding of essential issues.

Figure BPSS5: Explanation of four BPS approaches (source: [Gloy96])
Legend:
BranchAddr—Branch Address,
BHSR—Branch History Shift Register,
BHT—Branch History Table,
PO—Prediction Outcome.
Comment:
This figure singles out the four schemes used in the research of [Gloy96]. Note that [Gloy96] and [Evers96] have chosen a different set of schemes in their research.

Major conclusions of the study are twofold: (a) First, better prediction accuracy results are obtained if prediction tables are not flushed after periodic context switches, (b) Second, re-
sults for traces which include both user code and system code differ from the results based only on user code.

The study in [Sechrest96] is based on extremely long custom made traces (both user and system code) and it claims that even the longest standard application suites give the traces which are not nearly as long as needed to measure precisely enough the real accuracy of various branch predictors. The study assume the model of BPS which is shown in Figure BPSS6 (still another way to represent the same basic model), and the authors have used the benchmarks presented in Figure BPSS7.

**Figure BPSS6:** Yet another model of BPS (source: [Sechrest96])

**Comment:**
This figure includes a still another alternative method of modeling the branch prediction system covered in this book. Receiving information from more sources is an important prerequisite for better understanding of essential issues. The reader should make an effort to understand the real reasons for using different symbolics.

<table>
<thead>
<tr>
<th>B</th>
<th>DI</th>
<th>DCB (%[TI])</th>
<th>SCB</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>compress</td>
<td>83,947,354</td>
<td>11,739,532 (14.0%)</td>
<td>236</td>
<td>13</td>
</tr>
<tr>
<td>eqtott</td>
<td>1,395,165,044</td>
<td>342,595,193 (24.6%)</td>
<td>494</td>
<td>5</td>
</tr>
<tr>
<td>espresso</td>
<td>521,130,798</td>
<td>76,466,489 (14.7%)</td>
<td>1784</td>
<td>110</td>
</tr>
<tr>
<td>gcc</td>
<td>142,359,130</td>
<td>21,579,307 (15.2%)</td>
<td>9531</td>
<td>2020</td>
</tr>
<tr>
<td>xlisp</td>
<td>1,307,000,716</td>
<td>147,425,333 (11.3%)</td>
<td>489</td>
<td>48</td>
</tr>
<tr>
<td>sc</td>
<td>889,057,008</td>
<td>150,381,340 (16.9%)</td>
<td>1269</td>
<td>157</td>
</tr>
<tr>
<td>groff</td>
<td>104,943,750</td>
<td>11,901,481 (11.3%)</td>
<td>6333</td>
<td>459</td>
</tr>
<tr>
<td>gs</td>
<td>118,090,975</td>
<td>16,308,247 (13.8%)</td>
<td>12852</td>
<td>1160</td>
</tr>
<tr>
<td>mpeg_play</td>
<td>99,430,055</td>
<td>9,566,290 (9.6%)</td>
<td>5598</td>
<td>532</td>
</tr>
<tr>
<td>nroff</td>
<td>130,249,374</td>
<td>22,574,884 (17.3%)</td>
<td>5249</td>
<td>228</td>
</tr>
<tr>
<td>real_gcc</td>
<td>107,374,368</td>
<td>14,309,867 (13.3%)</td>
<td>17361</td>
<td>3214</td>
</tr>
<tr>
<td>sdet</td>
<td>42,051,812</td>
<td>5,514,439 (13.1%)</td>
<td>5310</td>
<td>508</td>
</tr>
<tr>
<td>verilog</td>
<td>47,055,243</td>
<td>6,212,381 (13.2%)</td>
<td>4636</td>
<td>850</td>
</tr>
<tr>
<td>video_play</td>
<td>52,508,059</td>
<td>5,759,231 (11.0%)</td>
<td>4606</td>
<td>757</td>
</tr>
</tbody>
</table>

**Figure BPSS7:** Benchmarks—SPEC versus IBS (source: [Sechrest96])

**Legend:**
B—Benchmarks,
DI—Dynamic Instructions,
DCB—Dynamic Conditional Branches (percentage of total instructions),
TI—Total Instructions,
SCB—Static Conditional Branches,
N—Number of Static Branches Constituting 90% of Total DCB.

**Comment:**
Note the minor difference in the number of dynamic instructions, between the benchmarks in the upper part of the table (SPEC) and the benchmarks in the lower part of the table (IBS). Also, note that the number of static branches constituting 90% of total dynamic conditional branches is extremely small in the case of SPEC, and much larger (on average) in the case of IBS. This means that in IBS a much larger percentage of branch population has an impact on the overall happenings in the system. Authors of [Sechrest96] believe that the later size difference is crucial for correct understanding of the essential issues (like, impact of aliasing, impact of warm-up, etc.). In other words, what matters (when it comes to proper performance evaluation) is the size of the branch population, not the size of the code.

Sechrest, Lee, and Mudge claim that control of aliasing (when many branches map into the same entry of the branch history table, due to the lower number of address bits used) and interbranch correlation (when various branches impact each other) are crucial for the prediction success of a scheme.

An important new trend in branch prediction research implies techniques which reduce negative branch history interference and improve target prediction for indirect branches [Sprangle97, Chang97].

3. About the Research of the Author and His Associates

Some of the research directions of the author and his colleagues are described in [Milutinovic96c], and cover the graduate research topics in both BPS for single chip microprocessor systems and multiple node heterogeneous systems.

Among other issues, the first research topic [Petrovic97] tries to figure out what happens if different MH-BPS schemes are used for user code and system code. The research concludes that, under certain conditions, the code execution time improvement goes up to about 10%. For details, see the author’s WWW presentation.

Among other issues, the second research topic [Ekmecic97] tries to figure out what happens if different MH-BPS schemes are used in heterogeneous systems to improve the next-task prediction and to speed up its allocation. The research concludes that, under certain conditions, the code execution time improvement goes up to almost 10%. For details, see the author’s WWW presentation.
The Input/Output Bottleneck

The so called input/output bottleneck is defined as a discrepancy between the larger speed of processing elements in the system and the smaller speed of input/output elements of the system. According to the overall structure of the book, after the basic issues are discussed, selected advanced issues will be covered, together with a brief comment on the contributions from the author and his colleagues.

1. Basic Issues

The overall system performance is frequently limited by I/O devices. The slow down comes for several reasons. Two of them are the most important: (a) monitoring of the I/O process consumes processor cycles, and (b) if I/O supplies input data, the processing has to wait until data are ready.

1.1. Types of I/O Devices

The I/O devices are divided into three major groups: (a) data presentation devices at the user interface, for processor to user communications, (b) data transport devices at the network interface, for processor to processor communications, and (c) data storage devices at the storage interface, for processor to storage communications. Of course, devices with dual or even triple role are not uncommon in present day machines.

Figure IOBU1 defines data rates for traditional presentation devices. Figure IOBU2 defines data rates for traditional transport devices. Figure IOBU3 defines data rates for traditional data storage devices. The I/O bottleneck exists for years now as a “medium” to “high” bottleneck, which is threatening to become a “major” bottleneck. However, these threats have not materialized, primarily due to technological advances. Consequently, the numbers in the three figures have to be considered conditionally, and as a lower bound.

<table>
<thead>
<tr>
<th>Devices</th>
<th>Data Rates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sensors</td>
<td>1 B/s–1 KB/s</td>
</tr>
<tr>
<td>Keyboard Entry</td>
<td>10 B/s</td>
</tr>
<tr>
<td>Communications Line</td>
<td>30 B/s–200 KB/s</td>
</tr>
<tr>
<td>CRT Display</td>
<td>2 KB/s</td>
</tr>
<tr>
<td>Line Printer</td>
<td>1–5 KB/s</td>
</tr>
<tr>
<td>Tape Cartridge</td>
<td>0.5–2 MB/s</td>
</tr>
</tbody>
</table>

Figure IOBU1: Data rates for some traditional data presentation devices (source: [Flynn95])
**Legend:**
CRT—Cathode Ray Tube,
KB/s—KiloBytes Per Second,
MB/s—MegaBytes Per Second.

**Comment:**
The numbers in this table change over time. The change is especially dramatic in the case of communications lines.

<table>
<thead>
<tr>
<th>Devices</th>
<th>Data Rates</th>
<th>Maximal Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graphics</td>
<td>1 MB/s</td>
<td>1–5 s</td>
</tr>
<tr>
<td>Voice</td>
<td>64 KB/s</td>
<td>50–300 ms</td>
</tr>
<tr>
<td>Video</td>
<td>100 MB/s</td>
<td>≈ 20 ms</td>
</tr>
</tbody>
</table>

**Figure IOBU2:** Data rates for some traditional data transport devices (source: [Flynn95])

**Legend:**
s—second.

**Comment:**
The speed of the video depends on the type of encoding used, and it is expected to raise during the years to come.

<table>
<thead>
<tr>
<th>Devices</th>
<th>Access Time</th>
<th>Data Rate</th>
<th>Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk</td>
<td>20 ms</td>
<td>4.5 MB/s</td>
<td>1 GB</td>
</tr>
<tr>
<td>Tape</td>
<td>O(s)</td>
<td>3–6 MB/s</td>
<td>0.6–2.4 GB (per cartridge)</td>
</tr>
</tbody>
</table>

**Figure IOBU3:** Data rates for some traditional data storage devices (source: [Flynn95])

**Legend:**
s—second.

**Comment:**
This is perhaps the field where technological changes of I/O devices are most dramatic. Still the speed gap between the central processing units and the data storage devices is widening.

Data presentation devices can be successfully made autonomous in their operation, and typically require minimal processor interaction. This is less of the case for data transport devices. Most of the processor workload comes from data storage devices. Consequently, most of the attention in the text to follow is dedicated to data storage devices (for both uniprocessor and multiprocessor environments).

Technological progress of data storage devices is especially fast. For example, at the time of writing of this book, an EIDE disk has an average access time of 11.5 ms, it supports the 16.6 MB/s data transfer rate, and the capacity of 4 GB. Since technology data change so quickly in time, the interested reader should check the WWW presentations of major disk technology vendors, if state of the art information is needed (for example: http://www.wdc.com/products/drivers/drive_specs/AC34000.html).

Also, at the writing of this book, tape technology is characterized with average access time $O(0.1)$ s, effective data transfer rate of 1.2 MB/s, and capacity of 20 GBs. For a state of the art information, see the WWW presentations of major tape vendors (for example, http://www.interface_data.com/idsp2140.html).

The CD-ROM technology has become very popular. At the time of writing this book, it is characterized with average access time of 150 ms, data transfer rate of 1800 KB/s to 2400 KB/s (speed 12x and 16x), and capacity of 650 MB (standardized). For state of the art information, see the WWW presentations of major vendors (for example, http://www.teac.com/dsp/cd/cd_516.html).
The new DVD-ROM standard implies access time of 4.7 ns, data rate of 9.4 MB/s, and capacity of 17 GB. For details, see the WWW presentations of major manufacturers (e.g., http://www.toshiba.com/taisdpd/dvdrom.htm).

1.2. Types of I/O Organization

The I/O devices in general and data storage devices in particular can be organized in three different ways: (a) Program-controlled I/O, (b) Interrupt-driven I/O, and (c) DMA-managed I/O.

Internal microprocessor design is crucial for efficient support of I/O. The P/M (processor/memory) bus represents the major interface between the processor and its I/O. Consequently, design of the P/M bus represents a major challenge during the microprocessor design process.

Issues of importance, when designing an efficient I/O interface are: (a) Finding the physical location, (b) Finding the path to the physical location, and (c) Finding the data, with no or only a minimal processor involvement.

Major types of I/O coprocessors for uniprocessor machines are: (a) Multiplexer channel, based on many low-speed devices, (b) Selector channel, based on one high speed device with capabilities to do data assembly and disassembly efficiently, and (c) Block channel, based on the combination of the first two approaches.

Major types of I/O coprocessors for multiprocessor machines, where it is important that devices are accessible to all processors in the system, no matter where they are physically located, are: (a) via a single low-speed asynchronous bus, (b) via multiple high-speed synchronous buses, and (c) via a combination of the above, with some intelligence added into the communications process.

1.3. Storage System Design for Uniprocessors

Access time is the major problem with storage devices. This problem can be made less pronounced if appropriate storage buffers are added. If these buffers exploit some type of locality principle, they are called disk caches. Disk caches are typically done in a semiconductor technology, which means a shorter access time. They are useful for two reasons: (a) because the spatial locality is present in the disk data access pattern (access happens in sequences, as far as data addresses), which can be utilized for prefetch purposes; and (b) because the disk timing access pattern includes the no-activity periods (access happens in bursts, as far as data transfer), which can be utilized for prediction purposes.

Figure IOBU4 defines three possible locations for disk cache buffers: (a) in the disk subsystem, (b) in the storage controller, and (c) in the main memory. Figure IOBU5 gives the typical miss ratios for three different locations of disk cache buffers.
Figure IOBU4: Three possible locations for disk cache buffers (source: [Flynn95])

Legend:
P—Processor,
IOP—Input/Output Processor.

Comment:
In disk caches, the spatial locality component is much more dominant than the temporal locality component. In processor caches, both types of locality are present, spatial more with complex data structures, and temporal more with single variables, like loop control, process synchronization, and semaphore variables.

Figure IOBU5: Miss ratios for three different locations of disk cache buffers (source: [Flynn95])

$C_D$—Disk,
$C_{IOP}$—Storage controller,
$C_U$—Cache in memory.

Comment:
If disk access prediction algorithms are used, each of the three different locations implies a different algorithm type, which enlarges the performance differences of the three different approaches (predictors in the memory have access to more information related to prediction, compared to those located in the I/O processor, and especially to those located in the disk itself).

Disk access implies two different activities, read and write. Different methods have been used to improve the read and the write access.

Disk arrays represent a method to improve disk read. Bytes of each block are distributed across all disks. Consequently, the time to read a file and the time to do the buffer-to-processor transfer become better. In many cases of interest, this speedup is about linear. This means, if N disks are used, the speedup will be equal to about N.

System structure and access structure typical of disk arrays are given in Figure IOBU6. Numerics which demonstrate the speedup derived from a disk array organization are given in Figure IOBU7.

---

**Figure IOBU6:** Structure of a disk array (source: [Flynn95])

**Legend:**
P—Processor,
s—number of disks acting as a single unit.

**Comment:**
This figure implies that latency time is larger than the transfer time; that relationship is technology dependent, and may change over time.

\[
T_{\text{transfer}} = n \cdot \frac{T_{\text{read}}}{s}
\]

For (1, s) configurations, \( n = E(f) \) and
\[ T_{\text{service}} = T_{\text{latency}} + \frac{E(f)}{s} T_{\text{read}} \]
\[ T_{\text{transfer}} = \frac{E(f)}{s} T_{\text{read}} \]

For (1,16):
\[ T_{\text{service}} = 17.5 + \frac{3.4}{16} (2.6) = 17.5 + 0.55 = 18.1\text{ms} \]
\[ T_{\text{transfer}} = 0.55\text{ms} \]

For (1,8), we would have:
\[ T_{\text{service}} = 17.5 + \frac{3.4}{8} (2.6) = 18.6\text{ms} \]
\[ T_{\text{transfer}} = 1.1\text{ms} \]

**Figure IOBU7:** Numerics of a disk array (source: [Flynn95])

**Legend:**
- \( n \) — number of blocks in a file;
- \( s \) — number of disks acting as a single unit.

**Comment:**
Variations of performance related computations represent potentially good test questions.

Disk logs represent a method to improve disk write. Data are first collected in a log buffer, until their size becomes equal to the size of a disk access unit. Consequently, disk access is characterized with the minimal ratio of overhead time to transfer time.

### 1.4. Storage System Design for Multiprocessor and Multicomputer Systems

Major contributors to the overall traffic for I/O in multiprocessor and multicomputer systems are: (a) virtual memory and virtual I/O traffic, to support the requirement that each and every I/O device is accessible to all nodes in the system, and (b) meta computing and meta processing, to support the requirement that each and every I/O device is accessible to all heterogeneous architectures representing different nodes in the system.

Figure IOBU8 defines the I/O requirements of the so called Grand Challenge applications [Patt94]. These applications have been defined as the major research driving forces in the fields of I/O technology and heterogeneous computing.

<table>
<thead>
<tr>
<th>Application</th>
<th>I/O Requirements</th>
<th>Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Environmental and Earth sciences</td>
<td>Current 1 GB/model, 100 GBs/application; projected 1 Tbyte/application.</td>
<td>S</td>
</tr>
<tr>
<td></td>
<td>10 Tbytes at 100 model runs/application.</td>
<td>A</td>
</tr>
<tr>
<td>Eulerian air-quality modeling</td>
<td>100 MB–1 GBs/run.</td>
<td>S</td>
</tr>
<tr>
<td>4D data assimilation</td>
<td>3-Tbyte database.</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>Expected to increase by orders of magnitude with the Earth Observing System—1 Tbyte/day.</td>
<td></td>
</tr>
</tbody>
</table>

Computational physics
Particle algorithms
in cosmology and astrophysics
Radio synthesis imaging

- 1–10 GBs/file; 10–100 files/run.  S
- 20–200 MB/s.  IOB
- 1–10 GBs.  S
- HiPPI bandwidths minimum.  IOB
- 1 Tbyte.  A

Computational biology
Computational quantum materials

- 150 MB (time-dependent code)  S
- 3 GBs (Lanczos code).  IOB
- 40–100 MB/s.  IOB

Computational fluid and plasma dynamics
High-performance aircraft simulation
Computational fluid and combustion dynamics

- 4 GBs of data/4 h.  S
- 40 MB to 2 GB/s disk, 50–100 MB/s disk to 3 inch storage (comparable to HiPPI/ Ultra).  IOB
- 1 Tbyte  A
- 0.5 GB/s to disk, 45 MB/s to disk for visualization  IOB

**Figure IOBU8:** I/O requirements of Grand Challenge applications (source: [Patt94])

**Legend:**
S—Secondary,
A—Archival,
IOB—I/O Bandwidth.

**Comment:**
All Grand Challenge applications belong to the domain of scientific computing. With the recent innovations in the internet technology, applications from the business computing domain become even more challenging.

Different companies employ different disk interface models in order to solve the I/O bottleneck in multiprocessor and multicomputer systems. The start point reading on these issues is [Patt94]. It covers the traditional approaches, and most of them can be generalized using the architecture from Figure IOBU9.

**Figure IOBU9:** Parallel I/O subsystem architecture (source: [Patt94])

**Legend:**
$n$—number of drives.

**Comment:**
The field of I/O for parallel processing is progressing more through the technological advances, rather than through the advances in architecture.

The Intel Touchstone Delta model implies a 2D array (16 by 32) of processing element nodes with 16 I/O nodes on the sides of the array.
The Intel Paragon model uses inexpensive I/O nodes which can be placed anywhere within a mesh.

The Thinking machines CM-5 model is based on a fewer I/O processors, each one characterized with a relatively high I/O bandwidth.

The best sources of information on the state of the art in the field are conferences which include sessions on I/O and manuals on the latest products for the Grand Challenge applications.

At the time of the writing of this book, the Encore Infinity SP model, based on an internal architecture of the reflective memory type, according to many, is believed to be the best I/O pump on planet (the reflective memory model and the author’s involvement are discussed in a follow-up section).

Network interface technologies are considered crucial, especially for high demand applications like Grand Challenge or multimedia. Figure IOBU10 sheds some light on the capacities and characteristics of some traditional approaches.

<table>
<thead>
<tr>
<th>Type</th>
<th>Bandwidth</th>
<th>Distance</th>
<th>Technology</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fiber Channel</td>
<td>100–1,000 MB/s</td>
<td>LAN, WAN</td>
<td>Fiber optics</td>
</tr>
<tr>
<td>HiPPI</td>
<td>800 MB/s or 1.6 GB/s</td>
<td>≤ 25 m</td>
<td>Copper cables (32 or 64 lines)</td>
</tr>
<tr>
<td>Serial-HiPPI</td>
<td>800 MB/s or 1.6 GB/s</td>
<td>≤ 10 km</td>
<td>Fiber-optics channel</td>
</tr>
<tr>
<td>SCI</td>
<td>8 GB/s</td>
<td>LAN</td>
<td>Copper cables</td>
</tr>
<tr>
<td>Sonet/ATM</td>
<td>55–4.8 GB/s</td>
<td>LAN, WAN</td>
<td>Fiber-optics</td>
</tr>
<tr>
<td>N-ISDN</td>
<td>64 KB/s, 1.5 MB/s</td>
<td>WAN</td>
<td>Copper cables</td>
</tr>
<tr>
<td>B-ISDN</td>
<td>≤ 622 MB/s</td>
<td>WAN</td>
<td>Copper cables</td>
</tr>
</tbody>
</table>

Figure IOBU10: Network capacities and characteristics (source: [Patt94])

Legend:
LAN—Local Area Networks (up to several meters),
WAN—Wide Area Networks (up to several kilometers).

Comment:
Quoted numbers are the subject of fast changes due to technological advances; for state of the art numbers, the reader is referred to related www pages. Also, new standards emerge (for example, IEEE FireWire).

Once again, issues covered so far are only those which, according to this author, represent the major problems to be solved by the designers of future microprocessors and multimicroprocessors on the single chip. The latest developments, only on these issues, are covered in the next section.

2. Advanced Issues

This part contains the author’s selection of research activities which, in his opinion, have made an important contribution to the field in the recent time, and are compatible with the overall profile and mission of this book.

2.1. The Disk Cache Disk

Paper [Hu96] describes an effort to optimize the I/O write performance. The solution of their research implies a small log disk used as a secondary disk cache to build a disk hierarchy. A small RAM buffer collects write requests, and passes them to the log disk when it is idle. In this way, one obtains performance close to the same size RAM for the cost of a disk.
Conditions of their analysis imply that the temporal component of data is relatively high. The higher the temporal locality, the higher the performance of this approach.

2.2. The Polling Watchdog Mechanism

Paper [Maquelin96] describes an effort to improve the efficiency of handling of incoming messages in message passing environments. The solution of their research implies a hardware extension which limits the generation of interrupts to the cases where polling fails to provide a quick enough response.

Conditions of their research imply that the message arrival frequency is the criterion for selection of interrupt versus polling. This solution is promising in environments typical of the future distributed shared memory multimicroprocessors on a chip.

An important new trend in I/O research implies issues in minimization of negative effects of deadlocks, as well the minimization of negative effects of multiple failures in RAID architectures [Pinkston97, Alvarez97].

3. About the Research of the Author and His Associates

The research of the author and his colleagues is described in [Milutinovic95d and Milutinovic96c]. One of the avenues explored is efficient architectural support for handling of large routing tables in ATM. Another avenue of research is related to DSIO (distributed shared I/O on the top of distributed shared memory).

Details of the ATM related research can be found in the master thesis of Dejan Raskovic, co-mentored with Emil Jovanov [Raskovic95]. The major goal was to use hashing in order to minimize the VLSI area of the solution, in conditions when ATM allows, rarely enough, that the table look-up starting from the address computed by hash function, is not completed within one ATM cell time, which means the loss of one ATM cell.

Details of the DSIO can be found in the doctoral thesis of Dejan Raskovic [Raskovic97]. Initially, the major goal was to come up with a flexible solution which minimizes the PCI slot count problem in the next generation Encore I/O pump. Later, the research diverged into a more promising direction of efficient interconnection topologies for a two-dimensional reflective memory environment.
Multithreaded Processing

This chapter includes three sections. The first one is oriented to basic issues (background). The second one is oriented to advanced issues (state-of-the-art). The third one is oriented to the research of the author and his associates (basic ideas and pointers to references).

1. Basic Issues

This chapter gives an introduction to multithreaded processing; mainly the elements which are, in the author’s opinion, of importance for future microprocessors on the chip.

There are two major types of multithreaded machines: (a) coarse grained, based on task level multithreading, and (b) fine grained, based on the instruction level multithreading.

Task level multithreading implies that switching to a new thread is done on context switch. Instruction level multithreading implies that switching to a new thread is done on every cycle.

Principle components of a multithreaded machine are: (a) multiple activity specifiers, like multiple program counters, multiple stack pointers, etc., (b) multiple register contexts, (c) thread synchronization mechanisms, like memory-access tags, two-way joins, etc., and (d) mechanisms for fast switching between threads.

A common question is what is the difference between a thread and a process. An important difference between threads and processes is that each process has its own virtual address space while multiple threads run in the same address space; consequently, thread switching is faster than process switching. According to another definition, threads are mostly supported at the architecture level, and processes are mostly supported at the operating system level. For example, major multithreading constructs like start, suspension, and continuation, of a thread, are usually supported on the ISA (instruction level architecture) level. Major processing constructs like start, suspension, and continuation, of a process, are usually supported at the OS (operating system) level.

Projects oriented to multithreading include, but are not limited to, Tera (Smith at Tera Computers), Monsoon and T* (Arvind at MIT in cooperation with Motorola), Super Actor Machine (Gao at McGill University), EM-4 (Sakai, Yamaguci, and Kodama at ETL in Japan), MASA (Halstead and Fujita at Multilisp), J-Machine (Dally at MIT), and Alewife (Agarwal at MIT). Many of these machines include elements of other concepts, like dataflow, message passing multicompacting, distributed shared memory multiprocessing, etc. Consequently, all
above mentioned efforts can be classified in a number of different ways. Again, for details see the original papers or [Iannucci94].

1.1. Coarse Grained Multithreading

The first coarse grained MIMD (multiple instruction, multiple data) machine based on multithreading was HEP (Heterogeneous Element Processor). It was built at Denelcor back in 1978. Structure of the HEP system is given in Figure MTPU1. It includes up to 16 processing element modules (PEM), a number of data memory modules (DMM), an input/output controller (IOC), and a multistage interconnection network (ICN) based on a number of packet switching units (PSU), which can be bypassed if necessary, using a set of local access paths (LAP).

![Figure MTPU1: Structure of the HEP multiprocessor system (source: [Iannucci94])](image)

Legend:
PEM—Processing Element Module;
DMM—Data Memory Module;
PSU—Packet Switch Unit;
LAP—Local Access Path;
IOC—Input/Output Controller;
ICN—Interconnection Network.

Comment:
The HEP is based on packet switching. More recent heterogeneous processing projects tend to explore more shared memory [Ekmecic96 and Ekmecic97].

Internal structure of a PEM is given in Figure MTPU2. Each PEM can run up to eight user threads and up to eight system threads. Details can be found in the original papers or in [Iannucci94].
Coarse grained multithreading is important as a method of combining existing microprocessors into larger and more powerful systems. However, fine grained multithreading (to be discussed in the next section) is important for the existing instruction level parallelism, in order to make the single chip machines faster.

### 1.2. Fine Grained Multithreading

As far as this author is concerned, among various approaches to fine grained multithreading, the approach which is referred to as simultaneous multithreading (SMT) seems to be the most promising candidate for incorporation into the next generation microprocessors on a single chip.

In traditional fine grained multithreading, only one thread issues instructions in each cycle. In SMT, in each cycle, several independent threads issue instructions simultaneously to multiple functional units of a superscalar. This approach provides higher potentials for utilization of resources in a wide-issue superscalar architecture. A study by [Tullsen95] indicates that, on an 8-issue superscalar processor, the performance can be up to about 4 times better compared to the same superscalar processor without multithreading, and up to about 2 times better compared to traditional fine grained multithreading on the same superscalar architecture.
Figure MTPU3 describes the essence of SMT. The term horizontal waste refers to unused slots within one cycle of a superscalar machine. Horizontal waste is a consequence of the fact that, often, one thread does not include enough instruction level parallelism. The term vertical waste refers to the cases when an entire cycle is wasted, because various hazards have to be avoided at run time. In Figure MTPU3, which describes traditional fine grained multithreading, horizontal waste is equal to 9 slots, and vertical waste is equal to 12 slots; consequently, the total waste is equal to 21 slots. If SMT was used, other threads could fill in the unused slots, which, in turn, may bring the total waste down to zero.

In conclusion, superscaling is not efficient for vertical waste, traditional fine grained multithreading is not efficient for horizontal waste, and SMT is efficient in both cases.

Figure MTPU4 lists the major sources of wasted issue slots, and possible latency hiding and/or latency reducing techniques which can be used to cure the problems. It is essential to understand that all listed latency hiding and latency reducing techniques have to be used properly before one turns to SMT for further improvement of the run time performance. Consequently, one has to be careful about the interpretation to benefits of SMT—it may appear as being more efficient that realistically, unless all techniques from Figure MTPU4 have been used properly. The research from [Tullsen95] recognizes the problem and provides a realistic analysis which sheds an important light on SMT.

<table>
<thead>
<tr>
<th>Source of Wasted Issue Slots</th>
<th>Possible Latency-Hiding or Latency-Reducing Techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>instruction TLB miss, data TLB miss</td>
<td>decrease the TLB miss rates (e.g., increase the TLB sizes); hardware instruction prefetching; hardware or software data prefetching; faster servicing of TLB misses</td>
</tr>
<tr>
<td>I cache miss</td>
<td>larger, more associative, or faster instruction cache hierarchy; hardware instruction prefetching</td>
</tr>
<tr>
<td>D cache miss</td>
<td>larger, more associative, or faster data cache hierarchy; hardware or software prefetching; improved instruction scheduling; more sophisticated dynamic execution</td>
</tr>
<tr>
<td>branch misprediction</td>
<td>improved branch prediction scheme; lower branch misprediction penalty</td>
</tr>
<tr>
<td>control hazard</td>
<td>speculative execution; more aggressive if-conversion</td>
</tr>
<tr>
<td>load delays (first-level cache hits)</td>
<td>shorter load latency; improved instruction scheduling; dynamic scheduling</td>
</tr>
<tr>
<td>short integer delay</td>
<td>improved instruction scheduling</td>
</tr>
<tr>
<td>long integer, short FP, (multiply is the only long integer operation, divide is the only long floating point operation) shorter latencies;</td>
<td></td>
</tr>
</tbody>
</table>
long FP delays | improved instruction scheduling
---|---
memory conflict | (accesses to the same memory location in a single cycle)
| improved instruction scheduling

**Figure MTPU4:** Causes of wasted issue slots and related prevention techniques
(source: [Tullsen95])

**Legend:**
TLB—Translation Lookaside Buffer;
FP—Floating Point operation.

**Comment:**
It is crucial that appropriate prevention techniques are completely utilized before the simultaneous multithreading is used.

A performance comparison of SMT and various other multithreaded multiprocessor approaches is given in Figure MTPU5. The results are fairly optimistic in favor of SMT, in spite of the fact that the study is based on a somewhat idealized case of SMT. The study concludes that SMT paves the way to 8-issue and 16-issue superscalars, while the techniques used in microprocessors of mid to late 90s, applied to superscalar machines, do not enable designers to go beyond the 4-issue superscalars.

<table>
<thead>
<tr>
<th>Purpose of Test</th>
<th>Common Elements</th>
<th>Specific Configuration</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unlimited FUs:</td>
<td>Test A: FUs = 32</td>
<td>SM: 8 thread, 8-issue</td>
<td>6.64</td>
</tr>
<tr>
<td>equal total issue bandwidth,</td>
<td>IssueBw = 8, RegSets = 8</td>
<td>MP: 8 1-issue</td>
<td>5.13</td>
</tr>
<tr>
<td>equal number of register sets (processors or threads)</td>
<td>Test B: FUs = 16</td>
<td>SM: 4 thread, 4-issue</td>
<td>3.40</td>
</tr>
<tr>
<td></td>
<td>IssueBw = 4, RegSets = 4</td>
<td>MP: 4 1-issue</td>
<td>2.77</td>
</tr>
<tr>
<td></td>
<td>Test C: FUs = 16</td>
<td>SM: 4 thread, 8-issue</td>
<td>4.15</td>
</tr>
<tr>
<td></td>
<td>IssueBw = 8, RegSets = 4</td>
<td>MP: 4 2-issue</td>
<td>3.44</td>
</tr>
<tr>
<td>Unlimited FUs:</td>
<td>Test D:</td>
<td>SM: 8 thread, 8-issue, 10 FU</td>
<td>6.36</td>
</tr>
<tr>
<td>Test A, but limit SM to 10 FUs</td>
<td>IssueBw = 8, RegSets = 8</td>
<td>MP: 8 1-issue procs, 32 FU</td>
<td>5.13</td>
</tr>
<tr>
<td>Unequal issue BW:</td>
<td>Test E: FUs = 32</td>
<td>SM: 8 thread, 8-issue</td>
<td>6.64</td>
</tr>
<tr>
<td>MP has up to four times</td>
<td>IssueBw = 8, RegSets = 8</td>
<td>MP: 8 4-issue</td>
<td>6.35</td>
</tr>
<tr>
<td>the total issue bandwidth</td>
<td>Test F: FUs = 16</td>
<td>SM: 4 thread, 8-issue</td>
<td>4.15</td>
</tr>
<tr>
<td></td>
<td>IssueBw = 4, RegSets = 4</td>
<td>MP: 4 4-issue</td>
<td>3.72</td>
</tr>
<tr>
<td>FU utilization:</td>
<td>Test G: FUs = 8</td>
<td>SM: 8 thread, 8-issue</td>
<td>5.30</td>
</tr>
<tr>
<td>equal FUs, equal issue bw, unequal reg sets</td>
<td>IssueBw = 8</td>
<td>MP: 2 4-issue</td>
<td>1.94</td>
</tr>
</tbody>
</table>

**Figure MTPU5:** Comparison of various (multithreading) multiprocessors and an SMT processor (source: [Tullsen95])

**Legend:**
T—Throughput (instructions/cycle);
FU—Functional Unit.

**Comment:**
Note that the number of instructions per cycle scales up almost linearly with the increase of the issue width.

### 2. Advanced Issues

This part contains the author’s selection of research activities which, in his opinion, have made an important contribution to the field recently, and are compatible with the overall profile of this book.
Paper [Eickmeyer96] describes an effort at IBM to use an off-the-shelf microprocessor architecture in a form of multithreading. The effort is motivated by the fact that memory accesses are starting to dominate the execution time of uniprocessor machines.

Major conclusion of their study is that multithreading is an important avenue on the way to more efficient multiprocessor/multicomputer environments. For details, see [Eickmeyer96].

Conditions of their analysis imply object oriented programming for on-line transactions processing. This does not mean that the approach is not efficiently applicable to other environments of interest.

Paper [Tullsen96] describes a follow up effort on the SMT project, aimed at better characterization of the SMT design environment, so that more realistic performance figures are obtained. The major goal of the project is to pave the way for implementation which extends the conventional wide-issue superscalar architecture using the principles of SMT.

Major principles of their study are: (a) minimizing the changes to the conventional superscalar architectures, which makes SMT more appealing for incorporation into future versions of existing superscalar machines, (b) making any single thread to be only slightly suboptimal, which means that performance of a single thread can be sacrificed for at most about 2%, and (c) achieving the maximal improvement over the existing superscalar machines when multiple threads are executed. In other words, [Tullsen96] insists that performance of multiple threads should not be targeted at the cost of slowing a single thread for more than about 2%.

Conditions of their analysis imply a modified Multiflow compiler to work with eight threads, and a specific superscalar architecture along the lines of SGI 10000. This study takes into account the fact that the increased number of multiple activity specifiers (register files, etc.) either slows down the microprocessor clock or causes some operations which are of the one-cycle type in traditional superscalars to become of the two-cycle type in SMT. In spite of the relatively pessimistic timing conditions of the study, SMT still demonstrates a relatively optimistic throughput of 5.4, versus the traditional superscalar throughput of 2.5, in conditions when all architectural details are the same, except that SMT runs multiple threads, and traditional superscalar runs a single thread.

An important new trend in high-performance microprocessors implies the combination of simultaneous multithreading on one side and multithreaded vector architectures and/or multiscalar processor architectures on the other side [Espasa97, Jacobson97].

3. About the Research of the Author and His Associates

The research of the author and his colleagues is described in references [Milutinovic95d] and [Milutinovic96c]. The work includes elements of both coarse grained and fine grained multithreading.

In the coarse grained multithreading research track, various approaches to heterogeneous computing are being explored, mostly in the domain of run time allocation, based on local-area network architectures [Milutinovic85c and Milutinovic86a], as well as wide-area network architectures [Ekmebic95 and Ekmebic96].

In the fine grained multithreading research track, various approaches to wave pipelining are being explored, mostly in the domain of processing element design, oriented to arithmetic [Davidovic97] and input/output [Vuletic97].
In order to make this book more usable for graduate teaching, research efforts of the author and selected research efforts of others have been covered only conceptually, without details, and should be treated as homework case studies for students.
Caching in Shared Memory Multiprocessors

Shared memory multiprocessing (SMP) is one of the most widely used concepts in the implementation of modern multiprocessor workstations. The SMP architecture is essentially an MIMD architecture. It includes a shared memory, an interconnection network, and a set of independent processors with caches (only L1 or both L1 and L2). All processors share the same logical address space and it is implemented as a single physical unit. Many successful implementations are based on off-the-shelf memory chips, standard buses, and off-the-shelf microprocessors. This makes the concept relatively easy to implement.

1. Basic Issues

Programming model of SMP machines is relatively simple and straightforward. It resembles the programming model of single instruction single data (SISD) machines. Therefore, the existing code can be easily restructured for reuse, and the new code can be easily developed using the well-known and widely accepted programming techniques.

Unfortunately, the SMP systems are not scalable beyond some relatively low number of processors. This number changes over time and depends on the speed of the components involved (processors, bus, memory). For the current technology of components involved, the performance/price ratio starts dropping after the number of processors reaches 16; it starts dropping sharply after the number of processors crosses the count of 32. Consequently, SMP systems with more than 16 nodes are rarely being considered.

One of the major problems in implementing the SMP is cache consistency maintenance. This problem arises if two or more processors bring the same shared data item into their local private caches. While this data item is being read from local private caches of the processors-sharers, consistency of the system is maintained. However, if one of the processors-sharers executes a write and changes the value of the shared data item, all subsequent reads (of that same data item) may result in a program error. Maintenance of cache consistency is done using appropriate cache consistency maintenance protocols. They can be implemented in hardware, software, or using hybrid techniques. Here we will cover only the basic notions of hardware protocols. For advanced aspects of hardware protocols, interested reader is referred to [Tomasevic93]. Software protocols will not be elaborated in this book, due to their marginal use in modern computer systems today. For a detailed information on software proto-
ocols, interested readers are referred to [Tartalja96]. Hybrid protocols try to combine the best of the two extreme approaches (fully hardware approach versus fully software approach). They represent a promising new avenue for the on-going research.

There are two major approaches to hardware-based cache consistency maintenance: (a) snoopy protocols and (b) directory protocols. Both approaches are elaborated in the sections to follow, in the context of cache consistency maintenance.

As it will be seen later (in the section on distributed shared memory systems), one can also talk about the memory consistency maintenance, in systems with one logical address space, which is implemented using a number of different physical memory modules. Such systems are referred to as distributed shared memory (DSM) and they typically consist of a number of interconnected clusters—one cluster representing an SMP system. In SMP systems, snoopy protocols are the most efficient solutions for maintenance of cache consistency. In DSM systems, snoopy protocols continue to be the most efficient approach to cache consistency maintenance (at the SMP level), while the directory protocols represent the most efficient solutions for maintenance of memory consistency (at the DSM level). In order to justify these statements, details of snoopy and directory protocols are presented next.

1.1. Snoopy Protocols

In snoopy protocols, information related to cache consistency maintenance is fully distributed, because the cache consistency maintenance mechanism is built into the controllers of local private caches. Major elements of the cache consistency maintenance mechanism are broadcast and snooping. In other words, each cache controller uses the shared bus to broadcast all consistency maintenance related information, and all cache controllers constantly monitor (snoop) the bus; each individual cache controller fetches from the bus the information which is relevant for that particular processing node.

In general, snoopy protocols are ideally suited for bus-based multiprocessors; this is because broadcast is the only operation supported by typical busses, and snoopy protocols only need the broadcast operation. Snoopy protocols are of the low implementational cost; however, they are characterized with a limited scalability, for reasons discussed earlier.

There are two basic classes of snoopy protocols: (a) write-invalidate (WI), and (b) write-update (WU). The criterion for classification is the type of action utilized: to avoid inconsistency, or to force consistency. The two protocol classes are explained next. They are described using the examples in Figure SMPU1 and Figure SMPU2.

Figure SMPU1: Explanation of a write-invalidate protocol (source: [Tomasevic93]).
Legend:
M—memory,
C
i—cache memory of processor #i,
V—validity bit.

Comment:
Write invalidate protocols generate much less traffic on the interconnection network during the consistency maintenance activities. One bus cycle is enough to invalidate the entire block. During that cycle, active lines are address bus and invalidation control (one line). Data bus is not used, and could be utilized for other purposes. However, write invalidate protocols generate more bus traffic later, if and when an invalidated data item is needed again.

**Figure SMPU2:** Explanation of a write-update protocol (source: [Tomasevic93]).

Comment:
Write update protocols generate much more traffic on the interconnection network during the consistency maintenance activities. The number of bus cycles needed is equal to the number of words in the block. In each bus cycle, both address and data lines are busy, and can not be used for other purposes. However, write update protocols generate no bus traffic later, if and when the updated data are used again (by individual processors from their local caches).

1.1.1. Write-Invalidate Protocols
Write-invalidate protocols allow the processors to do a simultaneous read of all cached copies. However, only one processor has a permission to write, during a given time interval. Other processors are given the permission to write at other time intervals.

Write to a shared copy is preceded by an invalidation signal. After the broadcast of the invalidation signal is completed, all other shared copies are invalidated (validity bit V set to zero), and the originator of the invalidation signal has the permission to write to its shared copy (which is not shared any more, since other copies are invalid now). Note that invalidation comes first, and writing comes second. If the order of invalidation and writing is reversed, it could happen that some other processor reads a stale copy, during the time interval after the shared copy was written by one processor and the validity bits were set to zero in the other processors.

The memory copy (if any) has to be either invalidated (if the write invalidate protocol is of the write-back type), or updated (if the write invalidate protocol is of the write-through type). Term write-back denotes that memory is updated when the block with the newly written value is to be replaced. Term write-through denotes that memory is updated immediately after the new written value is deposited into the local private cache.
The example in Figure SMPU1 shows the read case (never a problem) and the write case (combined with the write-back approach). The new reader (cache number \(N\)) simply reads in the shared value and uses it. The new writer (cache number 2) first invalidates the other caches and the memory, and then it writes its own cache (invalidation signal is typically denoted as \text{inv}X).

Write-invalidate protocols are cheap to implement. They generate relatively low bus traffic—only the invalidation signal in the case of the example of Figure SMPU1. However, they are characterized with one important disadvantage. If the newly written copy is to be shared in the future, all other processors needing that copy will have to run through a read miss cycle, which slows down the code being executed. This problem is cured in the write-update protocols discussed next.

1.1.2. Write-Update Protocols

Write-update protocols allow multiple processors with write permission, and support the so-called distributed write approach. This means that the updated word is immediately broadcast to other sharers.

Main memory can be updated at the same time or not. In other words, in principle, both write-back and write-through approaches can be used. In practice, write-back is typically used for private data, and write-through is typically used for shared data.

The example in Figure SMPU2 shows the read case (never a problem) and the write case (combined with the write-through approach). The new reader (cache number \(N\)) simply reads in the shared value and uses it. The new writer (cache number 2) writes its own cache, but it also forces the newly written value into the other caches as well as the memory.

Write-update protocols are not so cheap to implement, since additional bus lines are needed. They generate more bus traffic—both control signals as well as data and addresses are being transferred across the bus in the case of the example of Figure SMPU2. However, as indicated earlier, other sharers of the same new value will have that value readily available when needed, and will not have to run through the potentially expensive read miss cycle. On the other hand, the updating may bring a data item which will never be used by a remote processor and may force the purge of a data item which may be needed later.

1.1.3. MOESI Protocol

The MOESI protocol is a consequence of an effort to combine the properties of the many existing snoopy protocols, and to enable different processors based on different protocols to be used as building blocks for a powerful SMP system. This protocol has been devised through the effort to develop the cache consistency maintenance superset protocol for the IEEE Futurebus standard.

Existing snoopy protocols imply that cache controllers are state machines based on some subset of the following five types of typical states: (a) \(M\) or modified, (b) \(O\) or owned, (c) \(E\) or exclusive, (d) \(S\) or shared, and (e) \(I\) or invalid. For a detailed explanation of existing snoopy protocols and their state machines, the interested reader is referred to [Tomasevic93].

As indicated symbolically in Figure SMPU3, the MOESI protocol includes all five states defined above. This fact is reflected by the name of the MOESI protocol, which includes the first letters of the states involved. The MOESI protocol is supported by seven special bus lines in the IEEE Futurebus.
Figure SMPU3: Explanation of the MOESI protocol (source: [Tomasevic93]).

Comment:
Different protocols cover different parts of the symbolic field of shapes. Ownership, validity, and exclusiveness are represented as three different circles.

The MOESI protocol supports some of the earlier protocols in their original form, like Berkeley and Dragon (see [Tomasevic93] for details); it supports some of the earlier protocols in a somewhat modified form, like Illinois and Firefly (see [Tomasevic93] for details). It also supports various memory update policies: (a) write-back, (b) write-through, and (c) non-caching.

As long as different modules of different architectures and different cache consistency maintenance protocols dynamically select only the actions permitted by the MOESI protocol, the coexistence is possible, and the system will consistent.

1.1.4. MESI Protocol

A subset of the MOESI protocol called MESI protocol is implemented in a number of 32-bit and 64-bit microprocessors. The MESI protocol includes only four states (M, E, S, and I) and represents a type of architectural support for incorporation of off-the-shelf microprocessors into the modern SMP systems.

The MESI protocol can be treated as a first step towards a future goal of having an entire SMP system on a single VLSI chip. This goal is believed to be implementable as soon as the on-chip transistor count crosses the 10 million threshold. Of course, with simpler nodes, the goal can be achieved sooner.

A partial list of machines implementing the MESI protocol or its supersets (for example, MOESI) includes, but is not limited to: (a) AMD K5 and K6, (b) Cyrix 6x86, (c) the DEC Alpha series, (d) the HP Precision Architecture series, (e) the IBM PowerPC series, (f) Intel Pentium, Pentium Pro, Pentium II, and Merced, (g) SGI 10000, and (h) SUN UltraSparc series.

1.2. Directory protocols

In the case of directory protocols, the responsibility for each cache coherence maintenance is typically built into a centralized controller. This controller is typically located in the main memory, or next to it (on the chip or on the board). One directory entry is associated with cache each block, and contains all consistency maintenance related information.

Figure SMPU4 includes an explanation of a typical directory protocol. In principle, reading is again not a problem; however, if a cache needs to write to a value, it first consults a central-
ized cache consistency maintenance related directory, and performs the desired action, which is either of the write invalidate type (most often) or the write update type (less often).

Figure SMPU4: Explanation of a directory protocol (source: [Tomasevic93]).

Legend:
ICN—interconnection network.

Comment:
Directory is consulted before each and every consistency maintenance activity. After the appropriate knowledge is acquired from the directory, only updates/invalidations which are necessary are performed. Consequently, broadcast is used very rarely; multicast and unicast to selected destinations are more appropriate solutions, on condition that interconnection network permits multicast and unicast (shared bus permits only broadcast). Note that the directory approach can be used both for cache consistency maintenance (less often), and for memory consistency maintenance (more often).

Actions involved in the cache consistency maintenance are of the following possible types: (a) unicast (one source to one destination), (b) multicast (one source to several but not all destinations), and (c) broadcast (one source to all destinations).

Since not only broadcast is involved, directory protocols are better suited for interconnection networks of a more general type (other than shared bus). However, in theory, directory protocols can be used on any type of interconnection network: (a) simplest, like bus, ring, or LAN (BRL in the rest of the text), (b) one of the reduced interconnection network types (like hypercube, N-cube, PM2I, etc. or (c) the most complex, like grid, mesh, or crossbar (GMC in the rest of the text).

In this context, bus implies parallel transfer on an open loop topology, ring implies parallel transfer on a closed loop topology, and LAN implies serial transfer on any topology (open, close, star, etc.). Furthermore, grid, mesh, and crossbar imply the same topology of lines (a number of horizontal and a number of vertical lines); however, processing nodes are positioned differently. In the case of a grid, processing nodes are located next to line crossings (messages go by the processing nodes). In the case of a mesh, processing nodes are located on the line crossings (messages go through the processing nodes). In the case of a crossbar, processing nodes are located on horizontal and/or vertical line terminations.
Information stored in the directory can be organized in a number of different ways. This organization determines the type and the characteristics of directory protocols. The three major types of directory protocols are: (a) full-map directory, (b) limited directory, and (c) chained directory. All of three of them will be briefly elaborated in the text to follow.

1.2.1. Full-Map Directory Protocols

Essence of the full-map directory protocols is explained using Figure SMPU5. Each directory entry (related to a cache block) includes \( N \) presence bits corresponding to \( N \) processors in the system, plus a single bit (dirty or D) which denotes if the memory is updated, or not (\( D = 0 \) means that the memory is updated, i.e. not dirty). This means that each directory entry includes \( N + 1 \) bits.

Figure SMPU5: Explanation of a full-map directory scheme (source: [Tomasevic93]).

Legend:
V—validity bit,
M—modified bit,
D—dirty bit,
K—processor which modifies the block.

Comment:
The full map directory approach is not scalable, and it is suitable only for the systems of a relatively small size (e.g., up to 16 nodes, as in the case of the Stanford DASH multiprocessor).

This type of protocol is denoted as Dir\((N)NB\), which means that each entry includes \( N \) presence related fields, and no broadcast operation is ever used.

Some of the consistency maintenance bits are kept in the local private cache memories; in the case of full-map protocols, two more bits are added per each cache block. Each cache block includes one validity bit (V) telling if the cached copy is valid (\( V = 1 \)), or not, and one modified bit (M) telling if the value in the cache compared to the value in the memory is different/modified (\( M = 1 \)), or not.

In Figure SMPU5 (left hand side) processor \( K \) reads a value, which is shared by processors 1 and \( K \). The sharing status is reflected through the fact that two out of the \( N \) bits in the centralized directory entry are set to one. Also, since the value in the memory and the values in the caches are the same, memory bit D is set to zero (\( D = 0 \)), and cache bits M are set to zero (\( M = 0 \)). Of course, the V bit for all shares is set to one (\( V = 1 \)).

In Figure SMPU5 (right hand side) processor \( K \) writes a new value into its own cache. Consequently, all but one of the bits in the presence vector are set to zero. Since a write-back approach is assumed in the example of Figure SMPU5, the dirty bit in the corresponding
memory entry is set to one \((D = 1)\), telling that the memory is not up to date any more. Since a write-invalidate approach is assumed in the example of Figure SMPU5, the \(V\) bits of other caches in the system get set to zero; in such condition, the \(M\) bits in other cashes become irrelevant (typically, designs are such that \(V=0\) causes \(M = 0\), as well).

The full-map protocol is characterized with the best performance, compared to other directory protocols. This is because the coherence traffic is the smallest, compared to other directory protocols. However, this protocol is characterized with a number of drawbacks.

First, the size of the directory storage is \(O(M \cdot N)\), where \(M\) refers to the number of blocks in the memory, and \(N\) refers to the number of processors in the system. This is essentially an \(O(N^2)\) complexity, which means that the protocol is not scalable. It is not able to support very large systems, due to a large cost which grows as \(O(N^2)\).

Second, adding a new node requires system changes, like the widening of the centralized controller, etc. Consequently, the protocol is not flexible for expansion, i.e. the cost per added node is considerably larger than the cost of the node alone.

Third, the centralized controller is a potential fault-tolerance bottleneck, as well as a performance-degradation factor, especially if more nodes are present in the system.

The protocols to follow eliminate some or all of the drawbacks of full-map protocols, at the expense of performance reduction, due to an increased consistency maintenance related traffic.

### 1.2.2. Limited Directory Protocols

Essence of the limited directory protocols is explained using Figure SMPU6, which includes a straightforward modification of a full-map protocol, called \(\text{Dir}(i)NB\). Each directory entry (related to a cache block) includes the same single dirty bit \((D)\) plus only \(i\) \((i < N)\) presence fields; each presence field being of the length \(\log_2 N\) bits. In total, this is \(1 + i \log_2 N\) bits (remember that the full-map protocol entries include \(1 + N\) bits). In reality, such an approach means less bits per entry. If \(N\) is large enough, and \(i\) is small enough, the size of one directory entry is less, compared to the full-map protocol. For example, if \(N = 16\) and \(i = 2\), a \(\text{Dir}(i)NB\) limited directory protocol includes 9 bits, and a full-map directory protocol includes 17 bits. In other words, if \(N = 16\), the \(\text{Dir}(i)NB\) limited directory protocols are less costly if \(i < 4\).

![Figure SMPU6](https://example.com/figure-smpu6.png)

**Figure SMPU6**: Explanation of the no-broadcast scheme (source: [Tomasevic93]).

**Comment:**

Note that the coherence overhead exists even for read operations. The amount of overhead decreases if appropriate operating system support is incorporated.

The described approach is possible, because several studies have shown that typically a block is shared only by a relatively small number of processors. In many cases, it is only a
producer and a consumer. The question is what happens if a need arises for a cache block to be shared by more than \(i\) processors, i.e., what type of mechanism is used to handle the presence vector overflow (which happens when the number of sharers becomes larger than \(i\)).

Actually, there are two types of limited directory protocols. The one described so far, as indicated, is denoted as Dir\((i)NB\). Another one is denoted as Dir\((i)B\), which means that each entry includes \(i\) presence related fields, and that the broadcast operation is used in the protocol.

In both schemes, some of the consistency maintenance bits are kept in the local private cache memories. As in the case of full-map protocols, two more bits are added per each cache block. Same as before, each cache block includes one validity bit \((V)\) telling if the cached copy is valid \((V = 1)\), or not, and one modified bit \((M)\) telling if the value in the cache compared to the value in the memory is different/modified \((M = 1)\), or not.

In both cases, the protocols are scalable, as far as the directory storage overhead, which is \(O(M\log N)\), or essentially \(O(N\log N)\). In both cases, protocols are less inflexible, because not always a new node means changes in the central directory (adding a new node may not change the prevalent pattern of sharing). However, in both cases, performance is affected by increased sharing. Also, the centralized directory continues to be the bottleneck of many kinds.

### 1.2.2.1. The Dir\((i)NB\) Protocol

In Figure SMPU6, which explains Dir\((i)NB\), there is a restriction for reading (which is a consequence of the fact that \(i < N\)), meaning that the number of copies for simultaneous read is also limited to \(i\). For example (left hand side of the figure), processor \(N\) reads a value, which is shared by processors 1 and \(K\). After the read miss, the value will be brought into cache \(N\) (right hand side of the figure), and the corresponding pointer field will be updated (\(K\) is substituted by \(N\)), but the copy in another cache (\(K\) in this example) has to be invalidated. In other words, the sharing status had to be changed, because two out of the \(i = 2\) fields (in the centralized directory entry) have already been in use. Also, since the value in the memory and the values in the active caches continue to be the same, memory bit \(D\) continues to be set to zero \((D = 0)\), and cache bits \(M\) continue to be set to zero \((M = 0)\). Of course, the \(V\) bit for all active sharers is set to one \((V = 1)\).

Because of the restriction on the number of simultaneously cached copies, there is an overhead even for read sharing. This brings a performance degradation, due to an increased miss ratio, which may become especially dramatic in the case of intensive sharing of read-only and read-mostly data. The term intensive sharing denotes situation when more than \(i\) nodes keep accessing the same value (for reading purposes) in some random order (the worst case is when they rotate their access order).

The scheme to follow eliminates the restrictions for reading, at some minimal the cost increase, and a potential performance degradation in some applications.

### 1.2.2.2. The Dir\((i)B\) Protocol

In Figure SMPU7, which explains Dir\((i)B\), there is no restriction on the number of read sharers. For example, processor \(N\) reads a value (left hand side of the figure), which is shared by processors 1 and \(K\). Processor \(N\) is again allowed to cache a copy. However, the sharing status will not be reflected through a change in the centralized directory. The two pointers remain unchanged, and continue to be set to one and \(K\), respectively. However, this protocol includes one more control bit in each directory entry, referred to as the broadcast bit \((B)\), and
that one gets set to one \((B = 1)\), to reflect the fact that the number of sharers is now larger than \(i\) (pointer overflow). The meaning of the two control bits in cache controllers is the same as before.

**Figure SMPU7:** Explanation of the broadcast scheme (source: [Tomasevic93]).

**Legend:**
- D—dirty bit,
- B—broadcast bit.

**Comment:**
Note that the coherence overhead does not exist now for read operations; however, the system state related information is not precise any more (it becomes fuzzy, unless appropriate operating system support is incorporated).

In this context, there are no restrictions on the number of simultaneous readers. However, if a write happens in conditions of a pointer overflow \((B = 1)\), an invalidation broadcast signal will be generated. Of course, the need for invalidation broadcast increases the write latency, plus some of the invalidation broadcasts are not necessary which wastes a fraction of the communications bandwidth.

All protocols described so far share some common drawbacks: (a) too much is centralized, which is a bottleneck, (b) adding a new node requires centralized changes, which is inflexible, and (c) the size of the central directory is still relatively large. The chained directory protocols, to be described next, remove these problems, at the price which is negligible in a number of common applications.

### 1.2.3. Chained Directory Protocols

In the case of chained directory protocols, directories are distributed across all caches, in the form of a chain of pointers (shared copies of a block are chained into a linked list). As indicated in Figure SMPU8, only the head of the list \((H)\) is in the memory, and occupies \(\log N\) bits, so that it can point to one of the \(N\) nodes.
Figure SMPU8: Explanation of chained directory schemes (source: [Tomasevic93]).

Legend:
H—header.

Comment:
The chained directory protocols are very frequently used in new designs, especially those based on the IEEE Scalable Coherent Interface standard, which supports the chained directory approach.

This means that the three above mentioned problems have been cured. First, almost nothing is centralized (only the vector of link lists heads), and there is no central bottleneck. Next, the system is absolutely flexible for expansion, and no changes in the centralized directory are needed, if a new node has to be added. Finally, the size of the centralized directory is down at the minimum, i.e. the storage cost is defined by $O(\log N)$.

Figure SMPU8 gives an example based on the singly-linked lists. After a new reader shows up (processor 1), it is added to the linked list, as the first node after the head of the list. All searches start from the head of the list; consequently, due to the locality principle, the most recently added read sharer is the most likely one to be reading in the near future. End of the linked list is denoted with a terminator symbol (CT).

A performance can be improved at a minimal cost increase, if a doubly-linked list is used. Research at MIT has shown that such an approach leads to performance which is close to the one provided by full-map directory protocols, at the cost which continues to be $O(\log N)$.

2. Advanced Issues

This part contains the author’s selection of research activities which, in his opinion, have made an important contribution to the field in the recent time, and are compatible with the overall profile of this book. Here we concentrate only on extended pointer schemes and protocols for efficient elimination of negative effects of false and passive sharing.

Other problems of interest include reduction of contention on shared resources. A good survey of techniques for reduction of contention in shared memory multiprocessors is given in [Stenstrom88].

2.1. Extended Pointer Schemes

The directory protocol of [Agarwal90, Agarwal91] is described in Figure SMPS1. It is referred to as the LimitLESS Directory Protocol, and represents an effort to rationalize the storage problems of typical directory schemes, at a minimal performance deterioration. It is based on standard limited directory protocols, like $\text{Dir}(i)\text{NB}$ or $\text{Dir}(i)\text{B}$, with a software mechanism
to handle pointer overflow. On a pointer overflow, an interrupt is generated, and the hardware pointers (of the cache block which is about to become shared more than supportable by hardware) get stored to memory, in order to free up directory space for lower-intensity sharers. Figure SMPS1 (left hand side) shows that the directory structure is basically the same as in standard limited directory protocols, except that a trap bit (T) is included to indicate the overflow processing status. Figure SMPS1 (right hand side) shows that some space has to be allocated in memory, for the software maintained table of overflow pointers.

**Figure SMPS1:** LimitLESS directory protocol (source: [Agarwal90], [Agarwal91]).

**Legend:**
- D—dirty bit;
- T—trap bit;
- SwMaintTableOverPtrs—Software Maintained Table of Overflow Pointers.

**Comment:**
In essence, the LimitLESS directory protocol represents a synergistic interaction between a limited directory protocol and the operating system. Consequently, the information about system state continues to be precise, even if the number of sharers exceeds the number of directory entries.

The approach of [Simoni90, Simoni91, Simoni92] is described in Figure SMPS2. It describes an effort to improve the characteristics of chained directory protocols, by using dynamic pointer allocation. Each directory entry includes a short and a long part. The Short Directory includes only a small directory header—one for each shared cache block. This header includes two fields: (a) dirty state bit, and (b) head link field. The Long Directory includes two types of structures: (a) one linked list of free entries of the so called pointer/link store (referred to as the Free List Link), and (b) numerous linked lists for shared cache blocks, with two fields per link list element (processor pointer and forward link). On read, the new sharer is added at the head position (the space for the new sharer is obtained by reallocating an entry from the Free List Link). On write, the linked list is traversed for invalidation purposes (and the space is freed up by returning the entries into the Free List Link). Essentially, this protocol represents a hardware implementation of the LimitLESS protocol.

**Figure SMPS2:** Dynamic pointer allocation protocol (source: [Simoni90, Simoni91, Simoni92]).

**Legend:**
X—CacheBlock;
LT—list terminator;
DirtyState (1 bit);
HeadLink (22 bits);
Pointer (8 bits);
Link (22 bits);
P/L—Pointer/Link Store.

Comment:
In essence, dynamic pointer allocation protocol relies on the operating system support, in order to maintain the absolute minimum of memory storage for specification of sharers.

The approach of [James90] is described in Figure SMPS3. This approach represents another effort to improve the characteristics of chained directory protocols, by using doubly linked lists. This effort resulted in the creation of a new IEEE SCI (Scalable Coherent Interface) standard, also referred to as IEEE 1592-1992 [IEEE93]. This protocol is based on the previously described chained directory protocols, and specifies a number of details of interest for its incorporation into industrial products. Reading a shared value implies: (a) finding the current head of the list in memory, and (b) adding the specifiers of the reading node, if it is the first reading of that node. Writing to a shared value implies: (a) sending the invalidation signal sequentially down the list and (b) creating the new list header for the modified block. Advantages of doubling the links are: (a) on block replacement, the corresponding linked list element has to be extracted from the list, and that operation is much easier to execute if the linked lists is bi-directional, and (b) possibility of a bi-directional traversal brings up a number of potential opportunities. Figure SMPS3 indicates that the major fields in memory entries are: (a) validity bit (V), and (b) memory header pointer (MemHeadPtr). Major fields in cache entries are: (a) status bit (S), (b) forward pointer (ForwPtr), and (c) backward pointer (BackPtr).

![Figure SMPS3: An explanation of the SCI protocol (source: [James90]).](image)

Legend:
V—validity (1 bit);
MemHeadPtr (8 bits);
State (5 bits);
ForwPtr (8 bits);
BackPtr (8 bits).

Comment:
A detailed description of the IEEE Scalable Coherent Interface 1592-1992 standard can be found in the reference [IEEE92].

2.2. The University of Pisa Protocols

Paper [Prete91] describes an effort to minimize the number of unnecessary write-backs related to the replacement of modified shared copies. The approach of [Prete91] is referred to as RST (Reduced State Transitions), and represents a modification of a write-update protocol.

Paper [Prete95] describes an effort to minimize the negative effects of false sharing, which occurs when cache blocks are relatively large, and several objects or processes share the same
cache block, as a consequence of process migration or unsophisticated compiler. The approach of [Prete95] is referred to as USCR (Useless Shared Copies Removal).

Paper [Prete97] describes an effort to minimize the negative effects of passive sharing, which happens when a process moves from one processor to another, and leaves behind private values which, after migration, formally become shared. The approach of [Prete97] is referred to as UPCR (Useless Private Copies Removal).

The approaches of the University of Pisa have been introduced in early and mid 90s; however, related experiences have been used lately in a number of state of the art microprocessor designs intended for multimicroprocessor systems. This is why they have been included into the state-of-the-art group of protocols.

An important new trend in SMP research is related to hardware fault containment and efforts to understand better the effects of communication latency, overhead, and bandwidth [Teodosiu97, Martin97].

3. About the Research of the Author and His Associates

The research of the author and his colleagues is described in a number of research and survey papers dating back to the year 1990. Their list is available at the author’s WWW presentation. Some of them are displayed in their full version.

The first effort is referred to as the WIN protocol which tries to use the word level invalidation in an effort to eliminate negative effects of false sharing. For more information on the details, see [Tomasevic92a] and [Tomasevic92b].

The follow up effort tries to incorporate the split temporal spatial cache concept into the SMP and DSM environments, and to formalize it into an advanced MOESI protocol referred to as MOESIA. For more information, see the on-going research reports of Davor Magdic.

The most recent effort concentrates on the cache injection approach, aimed at the reduction of the probability that the first read in SMP and DSM systems results in a miss. For more information, see the on-going research reports of Aleksandar Milenkovic.
Distributed Shared Memory

The Distributed Shared Memory approach represents an effort to combine the best characteristics of two different approaches: (a) SMP—shared memory multiprocessing and (b) DCS—distributed computing systems. As already indicated, SMP has one major good characteristic (simple to use programming model, based on one shared address space), and one major bad characteristic (it is not scalable). As it is well known, DCS has one major good characteristic (it is scalable), and one major bad characteristic (difficult to use programming model, based on a number of distinct address spaces, and message passing between these spaces).

The major characteristics of DSM are: (a) simple to use programming model, and (b) it is scalable. The simple programming model is a consequence of the fact that DSM is also based on a single logical address space. The scalability is a consequence of the fact that different portions of the address space (which is logically unique) are implemented in different processing nodes (which means physically distinct).

This approach has emerged relatively recently. However, some systems back in 80s, and even back in 70s, do include properties of today’s DSM systems. Some examples dating back into 70s, and elaborated in early 80s, include the Encore’s RMS (Reflective Memory System) system [Gould81], the Modcomp’s MMM (Mirror Memory Multiprocessor) system [Modcomp83], the University of Belgrade’s Lola system [Protic85], and the University of Pisa’s MuTeam system [Corsini87].

1. Basic Issues

Basic structure and organization of a DSM system are given in Figure DSMU1. The backbone of the system is a system level interconnection network (ICN) of any type. This means that some systems include a minimal ICN of the BRL (bus, ring, or LAN) type. Other systems include a maximal ICN of the GMC (grid, mesh, or crossbar) type, or some reduced ICN, which is in between, as far as cost and performance.
Figure DSMU1: Structure and organization of a DSM system (source: [Protic96a]).

Legend:
ICN—Interconnection network.

Comment:
Note that each cluster (node) can be a uniprocessor system, or a relatively sophisticated multiprocessor system. In a number of designs, both clusters and interconnection network are off-the-shelf systems. Only the DSM mechanisms are custom made, in hardware, software, or a combination of the two.

Nodes of a DSM system are referred to as clusters in Figure DSMU1. In reality, a cluster can include a single microprocessor, or an entire SMP system. Systems can be uniform (all clusters implemented as a single microprocessor, or all clusters implemented as an SMP) or hybrid (different clusters based on different architectures).

Basic elements of a cluster are: (a) one or more processors with their level 1 (L1) and level 2 (L2) caches, (b) a part of memory with the memory consistency maintenance directory, and (c) interconnection network interface, which is more or less sophisticated.

Finally, the logical DSM memory address space is obtained by combining the memory address spaces of physical modules placed in different processing nodes. Address spaces of different physical modules can logically overlap (which is the case in systems with replication), or they can be logically non-overlapping (which is the case in systems with migration), or the overlap can be partial (as indicated in the bottom left corner of Figure DSMU1).

1.1. The Mechanisms of a DSM System and Their Implementation

The term mechanisms of a DSM system covers all engineering aspects necessary for proper functioning of the concept. Typical DSM mechanisms include: (a) Internal organization of shared data and related system issues, (b) Granularity of consistency maintenance and related system issues, (c) Access algorithms and related issues, (d) Property management and related issues, (e) Cache consistency maintenance on the cluster level, and (f) Memory consistency maintenance on the system level.
The DSM mechanisms can be implemented in three basic ways: (a) software, (b) hardware, and (c) a combination of the two.

First DSM implementations were in software, on platforms such as message passing multiprocessors or clusters of workstations. Software implementation can be placed on various layers of the software system: (a) library routines, (b) optimizing compiler, (c) operating system, outside the kernel, and (d) operating system, inside the kernel. These implementations were easy to build and cheap to implement, but relatively slow and inefficient. Consequently, researchers turned to hardware implementations.

Hardware implementations were fast and efficient, but relatively complex and expensive. Therefore, the researchers had to turn to hybrid implementations, where some mechanisms are implemented in hardware and others in software.

According to many, hybrid DSM is the best way to go. Each particular mechanism is to be implemented in the layer (software or hardware) that it is best suited for. This means the best price/performance ratio. In some cases, hybrid implementations start from a full hardware approach, and then migrate selected mechanisms into software (typically, those which are complex and expensive to implement in hardware). In other cases, hybrid implementations start from a full software approach, and then migrate selected mechanisms into hardware (typically, those which are slow if implemented in software and frequently used in applications of importance).

The sections to follow give brief explanations of the major DSM mechanisms. For detailed explanations, interested readers are referred to [Protic97] and [Protic98]. These references include both the details and the references to original research publications.

1.2. The Internal Organization of Shared Data

In principle, shared data can belong to any data type. They can be non-structured or structured. If structured, they can be structured as traditional complex data types or as objects in line with modern programming languages.

Shared data are being accessed during the program segments called critical code. These critical code segments have to be properly protected, which is the subject of a follow up section of this book.

Organization of shared data is an issue of interest for the optimizing compiler design and the system programming. It is not to be confused with the type and granularity of consistency maintenance units, which is the subject of the next section.

1.3. The Granularity of Consistency Maintenance

As far as the cache memory consistency maintenance, the granularity is typically determined by the internal design of the microprocessors used in different clusters of the system. These microprocessors are typically off-the-shelf machines, and cache block sizes are in the range from four to eight, or 16 words.

As far as the memory consistency maintenance, the granularity is typically determined by the system designers, and it can take a number of different sizes: (a) word, (b) block, (c) small page, (d) large page, (e) object, (f) segment.

Hardware approaches typically utilize relatively small units of consistency maintenance (word, block, or small page). This is so because hardware can easily support smaller granularity, which is good for issues like space utilization, data thrashing, and false sharing.
Software approaches typically utilize relatively large units of consistency maintenance (large page, objects, or segments). This is so because software can not easily support smaller granularity; however, they can easily support mechanisms which check and improve on the issues like space utilization, data thrashing, and false sharing. Also, software algorithms can support semantic data structures like objects and/or segments, which is good for some applications.

Hybrid approaches typically utilize medium size units of consistency maintenance (small page or large page). This is so because hybrid approaches can easily combine the best of the two extreme solutions, for the best price/performance on the given technology/application prerequisites.

1.4. The Access Algorithms of a DSM System

There are three basic access algorithms used in DSM systems: (a) SRSW (Single Reader Single Writer), (b) MRSW (Multiple Reader Single Writer), (c) MRMW (Multiple Reader Multiple Writer). Each one will be elaborated here shortly.

The SRSW algorithm implies that only one node is allowed to read during a given period of time, and only one node is allowed to write during the same period of time. This algorithm is well suited for typical producer-consumer relationships in two-node systems. If applied to multiple-node systems, the management mechanism (to be discussed in the next section) must allow the migration of memory consistency units (pages, in most of the cases). In that case, if another node needs to read or write, it first makes an action to migrate the memory consistency unit of interest into its own physical memory, which makes it become the owner of that memory consistency unit, and then it reads or writes. In DSM systems, SRSW algorithm is used only in conjunction with the other two algorithms, but never as the only algorithm in the system. In such schemes, typically, SRSW is used only for some data types (which behave as SRSW), and other two algorithms are used for other data types (which demonstrate a behavior not suitable for SRSW).

The MRSW algorithm implies that several nodes are allowed to read during a given period of time, but only one node is allowed to write during a given period of time. In the case of reading, no migration of memory consistency units is needed. Migration is needed only in the case of writing. This approach is compatible with typical application scenarios, where multiple consumers use data generated by one producer. Consequently, this type of algorithm is used in most of the traditional DSM machines.

The MRMW algorithm implies that all of most nodes are allowed both to read and to write during a given period of time. In systems with replication, in write-through update-all systems (like RMS or MMM), whenever a node writes to a shared variable, this writing is also sent to all other nodes which potentially need that value, and the remote nodes get updated after a network delay. If the network serializes all writes in the system, data consistency will be preserved, and each node can read shared variables from its own local physical memory. In systems with replication, in write-back systems, and in systems in which the interconnection network does not serialize the writes, additional action (on the software or hardware part) is needed, in order to preserve data consistency. In systems with migration, appropriate mechanisms have to be applied (in software or in hardware), to help preserve the data consistency.
1.5. The Property Management of a DSM System

Except in all-cache systems (COMA,† like KSR or DDM) and many low-complexity systems (UMA,‡ like RMS or Memory Channel), with caches only and no main memory, each unit of consistency is typically associated with a home node (an important characteristic of CC-NUMA§ systems). The home node for a given unit of consistency (read “page,” in most of the systems) is responsible for the management of ownership of that unit of consistency (different processing nodes can own it during different time intervals).

Nodes can be assigned to units of consistency in a number of different ways. One way is, at compile time, to assign unit #0 to node #0, unit #1 to node #1, etc. In principle, it helps the application execution time, if the home node is the same node as the most frequent owner. Consequently, a good optimizing compiler always tries to do static assignment (compile time) of units of consistency to nodes, in such a way that the home for each unit of consistency is its most frequent owner. Another way is, at run time, to assign nodes according to a pre-defined algorithm. An approach which is sometimes used is to assign unit #0 to the first node that needs it, unit #1 to the first node that needs it, etc. This approach starts from the assumption that the first user is the most likely to be the most frequent user.

In SRSW algorithms, when a node has to read or write, it first consults the home of the memory consistency unit that the variable to be read/written belongs to, and requests that the ownership of the entire memory consistency unit is passed to it (i.e., to the node which is to read or write). After the ownership is granted, and the unit of consistency is migrated, the reading or writing can happen. In MRSW algorithms, the same procedure happens only on writing. In MRMW algorithms this procedure is not needed on either reading or writing, but other appropriate actions are needed, in order to preserve the data consistency.

The management mechanism can be either centralized or distributed. If distributed, the responsibility for different units of consistency can be fixed (at compile time or run time) or variable (which means changeable at run time). Knowledge about current nodes for different units of consistency can be kept in the form of traditional tables or linked lists. Knowledge about the concrete node which is the home or owner for a specific unit of consistency can be public, or only the node involved can have that knowledge. Different management strategies and more have been elaborated in the Yale Ph.D. thesis of Kai Li.

1.6. The Cache Consistency Protocols of a DSM System

As far as the cache consistency maintenance, both research prototypes and industrial products heavily rely on the mechanisms supported by the off-the-shelf microprocessors built in. This means that either WI (write-invalidate), or WU (write-update), or TS (type-specific) protocols can be used. Type-specific means that WI is used for some data types, and WU for other data types.

As indicated earlier, snoopy protocols are used at the cache consistency maintenance level, and directory protocols are used at the memory consistency maintenance level. Consistency maintenance on the memory level is always based on one of the chosen memory consistency models; consequently, the next section sheds more light on different memory consistency models.

† COMA = Cache Only Memory Access
‡ UMA = Uniform Memory Access
§ CC-NUMA = Cache Consistent Non-Uniform Memory Access
1.7. The Memory Consistency Protocols of a DSM System

Memory consistency models (MCMs) can be restricted (strict or sequential) or relaxed (processor, weak, release, lazy, lazy release, entry, AURC, scope, or generalized). The more relaxed a memory consistency model, the more difficult it is to program a DSM machine. However, the performance gets better. Memory consistency models determine the timing when different nodes in the system have to become aware about changes on variables that they share and potentially need in the future.

In principle, memory consistency protocols can be based in either update or invalidation. In other words, in order to keep the memory consistent, protocols can use either update or invalidation of memory consistency units.

The strict MCM implies that all nodes become immediately aware of all shared data changes. This type of MCM has only a theoretical value, unless relative interconnection network (ICN) delays are negligible.

The sequential MCM implies that all nodes become aware of all shared data changes, after a delay introduced by the ICN. Sequential MCM is typically supported on systems which use BRL (bus, ring, or LAN) in the ICN. All processors see each particular data access stream, as well as the global data access stream in the same order.

The processor MCM implies that all nodes see each particular data access stream in the same order; however, each node sees the global data access stream in a different order. Processor MCM is typically supported on systems which use BLR and include ICN access buffers, which forward data to the ICN using different speeds and priorities.

The weak MCM (as well as other MCMs to follow) implies that special synchronization primitives are incorporated into the program code. In between the synchronization points, the memory consistency is not maintained, and programmers have to be aware of that fact, to avoid semantic problems in the code. Synchronization points are referred to as either: (a) acquire, at the entry point of a critical section, and (b) release, at the exit point of the critical section. In the weak MCM, consistency is maintained at both synchronization points, and the code is not allowed to proceed after the synchronization points, unless the memory consistency has been established. Synchronization points by themselves have to follow the sequential consistency rule (all nodes seeing the same global order of accesses).

The release MCM implies that memory consistency has to be established only at release points, and the release points by themselves have to follow the processor consistency rule. At each release point, all other processors/processes in the system get updated with the changes to variables made prior to the release point of the critical code section just completed. This generates some potentially unnecessary ICN traffic, which may have a negative impact on the speed of the application code.

The lazy release MCM implies that memory consistency has to be established only at the next acquire point, which means that the ICN traffic will include only the updated variables that might be used by the critical code section which is just about to start. This means less ICN traffic, but more buffering space at each processor, to keep the updated variables until the start of all critical code sections to follow. The better execution speed of the application code is paid by more data buffering at all nodes. Note that some of the updated variables will not be used in the critical code section to follow. Nevertheless, such variables will contribute to the ICN traffic, because they might be used, and consequently have to be passed over to the next critical code section, which is about to start running on another processor.
The entry MCM implies that each shared variable (or each group of shared variables) is protected by a synchronization variable (a critical section is bounded by a pair of synchronization accesses to the synchronization variable). Consequently, updated variables will be passed over the network if and only when absolutely needed by the follow-up critical code sections. Such an approach leads to the performance which is potentially the best. However, concrete performance depends a lot on the details of the practical implementation. The first implementation of entry consistency (Midway project at CMU) requires the programmer to be the one to protect the specific shared variables (or the groups of shared variables). If such implementation of entry MCM is compared with lazy release MCM, differences are small; for some of the SPLASH-2 applications [Woo95] entry MCM works better, and for other SPLASH-2 applications lazy release MCM works better [Adve96]. This author believes that, with an appropriate implementation, entry MCM is able to provide better performance for most SPLASH-2 applications.

In principle, implementations of various MCMs can be done predominantly in hardware or predominantly in software. Figure DSMU2 summarizes the discussion on MCMs. For each of the six different MCMs discussed so far, two projects (or two groups of projects) have been specified: (a) one which refers to the first predominantly hardware implementation of the given MCM, and (b) one which refers to the first predominantly software implementation of the given MCM. The figure also includes the names of the major researchers, as well as the names of universities/industries involved. Information in this figure has to be considered conditionally, because different researchers classify different approaches in different ways.

<table>
<thead>
<tr>
<th>MEMORY CONSISTENCY MODELS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementations of MCMs (hw-mostly vs. sw-mostly):</td>
</tr>
<tr>
<td>Sequential: MEMNET+KSR1 vs. IVY+MIRAGE</td>
</tr>
<tr>
<td>Delp/Farber(Delaware)+Frank(KSR) vs. Li(Yale)+Fleish/Popek(UCLA)</td>
</tr>
<tr>
<td>Processor: RM vs. PLUS</td>
</tr>
<tr>
<td>Gould/Encore/DEC(USA) vs. Bisiani/Ravishankar/Nowatzyk(CMU)</td>
</tr>
<tr>
<td>Weak: TSO vs. DSB</td>
</tr>
<tr>
<td>SUN/HAL/SGI(USA) vs. Dubois/Scheurich/Briggs(USC)</td>
</tr>
<tr>
<td>Release: Stanford-DASH vs. Eager-MUNIN</td>
</tr>
<tr>
<td>Gharachorloo/Gupta/Hennessy(SU) vs. Carter/Bennett/Zwaenepoel(Rice)</td>
</tr>
<tr>
<td>LazyRel: AURC vs. TREADMARKS</td>
</tr>
<tr>
<td>Iftode/Dubnicki/Li(Princeton) vs. Zwaenepoel/Keleher/Cox(Rice)</td>
</tr>
<tr>
<td>Entry: SCOPE vs. MIDWAY</td>
</tr>
<tr>
<td>Iftode/Singh/Li(Princeton) vs. Bershad/Zekauskas/Sawdon(CMU)</td>
</tr>
</tbody>
</table>

Figure DSMU2: Memory consistency models

Comment:
It is important to know the names behind the major achievements in the field of interest. Conditionally speaking, each memory consistency model can be implemented mostly in hardware and/or mostly in software. For all major memory consistency models, for both hardware-mostly and software-mostly implementations, one can find here the name of the project, the names of major contributors, and the name of the university. In the case of industry, individual names have been omitted.

Chronologically, software implementations of relaxed MCMs have come first. Here is an explanation. Basic idea behind software-implemented DSM, often referred to as SVM (Shared Virtual Memory), is to emulate a cache coherence protocol at the granularity of pages rather than cache blocks. In such conditions (coarser granularity of memory coherence units), major issues that affect performance are false sharing and fragmentation. False sharing happens when two semantically unrelated processors/processes share the same page (accessing unrelated data which fall into the same page), and one of them writes data. Data of the other
will be unnecessarily invalidated and it (the other one) will have to bring over the ICN the data that it already has. If data update is used, the ICN traffic will increase again, due to the updates which are not needed because they refer to data which are not being used by the processor/process being updated. Page fragmentation happens when a processor/process needs only a part of the page, because there is no finer granularity available. Both problems are alleviated through the use of relaxed MCMs, which permit the system to delay the coherence activity from the time when the modification happens to the time of the next synchronization point of interest (release or acquire or both). Postponing the coherence activity means that some of the ICN traffic will not happen. On the other hand, hardware implemented DSM typically uses cache blocks as memory coherence units, which means that impacts of false sharing and fragmentation are not so severe, and that there is no urgent need to do relaxed MCMs in hardware, except for uniformity reasons when everything is done in hardware (like relaxed MCM in DASH), or when the speedup of a software solution is needed (like AURC and SCOPE in SHRIMP).

The major MCMs will now be revisited from another point of view, using pictures and specific examples (definitions of all mnemonics used in the pictures have been given in figure captions).

### 1.7.1. Release Consistency

Stanford DASH project includes a predominantly hardware implementation of the release consistency, which is depicted in the upper part of Figure DSMU3. As soon as a value is generated (which is potentially needed by another process on another processor), the update is performed. Consequently, values $W(x)$, $W(y)$, and $W(z)$ are being updated at different times.

![Figure DSMU3: Release consistency—explanation (source: [Lenoski92 + Carter91]).](image)

**Legend:**
- $W$—write,
- $R$—read,
- REL—release,
ACQ—acquire.

**Comment:**

Note that Dash (hardware implementation) and Munin (software implementation) create different interconnection network traffic patterns. Dash accesses the network after each write, which means a more uniform traffic. Munin accumulates the information from all writes within the critical section, and accesses the network at the release point, which means bursty traffic. This difference is due to inherent characteristics of hardware (low overhead for single data broadcast, multicast, or unicast) and software (high overhead for single data broadcast, multicast, or unicast). Note that release consistency, in general, no matter if implemented in hardware or software, updates/invalidates more processors than needed, which is a drawback to be overcome by the lazy release consistency, at the cost of extra buffering in the system. In theory, release consistency can be based on either update or invalidate approaches.

The Rice Munin project includes a predominantly software implementation of the release consistency, which is depicted in the middle part of Figure DSMU1. All updated values are being collected in a special-purpose buffer, and the update is done only at the release point, which is more convenient in software implementations. Obviously, the software implementations create a bursty ICN traffic.

The example in the lower part of Figure DSMU3 underlines the fact that release consistency broadcasts the updated values to all potentially interested processors, in spite of the fact that many of them will not need the data. This picture is provided to serve as the contrast to the picture which explains lazy release consistency, and follows next.

1.7.2. Lazy Release Consistency

The first predominantly software implementation of lazy release consistency was done as a part of the TreadMarks project at Rice [Keleher94]. The lazy release consistency is best explained using the example in Figure DSMU4. This figure covers the same example as Figure DSMU3.

It is immediately seen that lazy release consistency generates considerably less ICN traffic. As indicated earlier, this potential performance improvement is paid with extra buffering in each processor—a price which is fully justified by the improved performance.

The LRC (lazy release consistency) implementation of TreadMarks is a multiple writer protocol using invalidation (other implementations are possible, as well). Each writer records all changes that it makes on all shared pages, since the last acquire point. This is done by comparing a copy of the original page (twin) and the modified original page; this comparison generates a data structure called diff. At the next acquire point, pages modified by other processes get invalidated. At the first access to an invalidated page, diffs are collected and applied in the proper causal order, to rebuild the coherent page.
**1.7.3. Entry Consistency**

The first predominantly software implementation of entry consistency was done as a part of the Midway project at CMU [Bershad93]. The entry consistency is best explained using the example in Figure DSMU5. This figure covers the same example as Figure DSMU4 and Figure DSMU3.

**Figure DSMU4:** Lazy release consistency—explanation (source: [Keleher94]).

**Comment:**
Updates/invalidates happen at the next acquire point. Only the shared variables related to the acquiring process are being updated/invalidated, which means less interconnection network traffic, at the cost of extra buffering in each processor (remaining updates/invalidates related to other shared variables changed by the previous process have to be held in a buffer, until the time comes when they are needed by a future acquiring process). This figure is related to a software implementation of lazy release (TreadMarks); software implementation is noticeable through the fact that a single update/invalidate burst happens at the next acquire point. In theory, lazy release consistency can be based on either update or invalidate approaches.

**Figure DSMU5:** Entry consistency—explanation (source: [Bershad93]).

**Legend:**
→—protected by,
NO—not needed.

**Comment:**
This figure is related to a software implementation of entry consistency (Midway). Through a help of the programmer, at the next acquire point, updates/invalidates are done only for the shared variables related to a given lock (S1 in this figure), which means even less interconnection network traffic, on average. In theory, entry consistency can be based on either update or invalidate approaches, although some people believe that entry consistency is inherently oriented to update, because the Midway implementation uses the update approach.
It is immediately seen that (in this idealized example) entry consistency generates even less ICN traffic than lazy release consistency. As indicated earlier, this potential performance improvement is paid (in the Midway implementation) with extra effort on the programmer side.

The EC (entry consistency) implementation of Midway is a single writer protocol using updating (other implementations are possible, as well). Entry consistency guarantees that shared data becomes consistent only when the processor acquires a synchronization object. Designers of the Midway system were aware of the fact that advantages of entry consistency may be difficult to obtain if limited programming effort is possible; consequently, in addition to entry consistency, Midway also supports release and processor consistency.

1.7.4. Automatic Update Release Consistency

The automatic update release consistency (AURC) was developed at Princeton, as a part of the SHRIMP project [Blumrich94, Iftode96a]. Some researchers consider it as an implementation of lazy release consistency, which includes elements of hardware support. Others treated it as a new MCM.

Figure DSMU6 shows an example which depicts the difference between the classical lazy release consistency (TreadMarks) and the automatic update release consistency (SHRIMP-2). The upper part of the figure includes a classical lazy release example. The middle part of the figure includes an AURC example based on the Copyset-2 mechanism. The lower part of the figure includes an AURC example based on the Copyset-N mechanism.

---

**Figure DSMU6:** AURC—explanation (source: [Iftode96a]).

**Legend:**
- RdFt—read fault,
- AuUp—automatic update.

**Comment:**
Automatic update release consistency implemented in the SHRIMP-2 project is best treated as a hardware-supported implementation of lazy release. Hardware supports the point-to-point communications between a writing node and the owner node for the page being written into (copyset-2); however, this point-to-point mechanism, enhanced by software, can be used as a building block for a more general communications pattern (copyset-N).

As indicated in Figure DSMU7, Copyset-2 mechanism implies automatic hardware-based update of the tables in the home processor, as soon as a value is updated. Copyset-N mechanism is a logical extension of Copyset-2 mechanism (this means that Copyset-2 mechanism is directly supported in hardware, and Copyset-N is synthesized out of the appropriate number of actions of the Copyset-2 mechanism). Details of AURC are given in Figure DSMU7, using a version of AURC (the AURC idea has undergone several modifications). Details not covered in Figure DSMU7 are left to the interested reader, to be developed as a homework.

### AURC

1. The hardware update mechanism works on the point-to-point basis. Upon a write to a location, only one remote node is updated—the home, no matter how many different nodes share that same data item.

2. The home for a specific page is determined at run time, as the node which writes the first into a page. It is assumed that the first writer will become the most frequent writer.

3. On the next acquire point, the acquiring node sends messages to the homes, to obtain the current versions of all pages (in jurisdiction of these homes) to be needed during the critical section which is just about to start.

4. Homes return the current version number of each page, and the requesting node compares it with the version number of the corresponding page in its possession.

5. The version at the node might be older, but still usable, if its version corresponds to the last update made by the same lock.

   The current version might be #21, and the node may have the version #18, but #18 may be the last version written to by the lock (process), which is related to the acquire point in question.

   If the node already has the last version of interest for it, the page will not be ported over the network; otherwise, it will.

6. The major issue here is to minimize the negative effects of false sharing, in cases when page sizes are relatively large, and several processes (locks) share the same page.

7. Home keeps the copy set (list of sharers) and the updated “version vector” telling which locks were writing to a particular version of each page, in the set of pages for which he/she is the home.

   Each node keeps just one element of the “version vector.”

   This is the vector element related to the pages replicated in that specific node.
8. During the execution of the code in the critical section, the node will update both itself and the home, on each write, as indicated earlier. Others will be updated after they acquire the lock and after they contact the home to obtain the last update of the page. This decreases the ICN traffic, in comparison with some other reflective (write-through-update) schemes, which update ALL sharers using a hardware reflection mechanism (RM and/or MC).

9. The above implements the lazy release consistency model through a joint effort of hardware and software (unlike TreadMarks, which does the entire work in software).

10. The AURC scheme has been implemented in the SHRIMP-2 project at Princeton.

**Figure DSMU7: AURC—Revisited**

**Comment:** The AURC idea was going through a number of different versions; version selected for presentation here is the one which seems to be the most suitable from the point of view of the final goal of this book, which is to present the facts of importance for implementation of a DSM on a chip, together with a number of dedicated accelerators for the most frequently used application functions.

### 1.7.5. Scope Consistency

The scope consistency (ScC) was developed at Princeton, also as a part of the SHRIMP project [Ifode96b]. Some researchers consider it to be an implementation of entry consistency, which includes elements of hardware support, and excludes programmer interaction (which represents a very important improvement). Others treat it as a new MCM.

The scope consistency is based on the same basic hardware support as the AURC, except that a more detailed bookkeeping is maintained, which brings it conceptually closer to entry consistency, and avoids the need for programmer support. Details of scope consistency are given in Figure DSMU8, using a version of scope consistency (the idea has undergone several modifications). Details not covered in Figure DSMU8 are left to the interested reader, to be developed as a homework.

**SCOPE**

1. The SCOPE is the same as the AURC, except for the following differences.

2. The version update vector at the home includes one more field, telling which variables (addresses) were written into, by a given lock; this is repeated for each version of the page.

3. Consequently, the home is not checked at the next acquire point, but after that point, when the variable is invoked, which corresponds to the ENTRY consistency model.

4. However, unlike with the ENTRY, the activity upon invoking a variable is related to the entire page, i.e. the entire page is brought over, if so necessary.
5. This means page-based maintenance and more traffic, rather than object-based maintenance and less traffic (like in Midway). In principle, Midway can also do page-based maintenance, but it does not.

6. The SCOPE is faster, although it is page-based rather than object-based, since it does a part of the protocol in hardware, using the same automatic update hardware as the AURC.

7. In conclusion, SCOPE can be treated as a hw/sw implementation of ENTRY.

8. Its advantage over ENTRY is in complete avoidance of compiler assistance.

9. The SCOPE was not implemented in any of the Princeton DSM machines.

10. The SCOPE research brings up a number of new ideas: merge, diff-avoid.

**Figure DSMU8: SCOPE—Revisited**

**Comment:**
The SCOPE idea was going through a number of different versions, and only one of them is selected for presentation here.

### 1.8. A Special Case: Barriers and Their Treatment

A closer look at some of the typical parallel applications (e.g., those found in the SPLASH-2 suite) reveals that many shared reads and writes frequently occur outside of explicit critical sections. We will briefly discuss the treatment of such memory references by the more relaxed memory consistency models mentioned above.

The aforementioned applications (frequently of scientific and engineering nature) often use barriers as a means of synchronization, in addition to using ordinary locks. Of $N$ processes participating in a parallel computation, each time one of them encounters a barrier, it will block until all the other $N - 1$ processes reach the same barrier. At that point, all $N$ of them get released. Barriers are used to separate phases in parallel computation; each such phase is called an epoch. A barrier can be intuitively understood as a point where total consistency is established, including the data written outside of explicit critical sections (with an exception that this is not necessarily true for the entry consistency model). It may be helpful to view a barrier as a release-acquire pair, i.e. as if a process performed a release at barrier arrival, and an acquire at barrier departure.

To illustrate the treatment of shared accesses both inside and outside of (explicit) critical sections, let us consider the following sequence of actions performed by three processors, P1, P2, and P3, in parallel:

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>BARRIER</td>
<td>BARRIER</td>
<td>BARRIER</td>
</tr>
<tr>
<td>$A := 1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$t_1$</td>
<td>$acquire \ S1$</td>
<td>...</td>
</tr>
<tr>
<td>$B := 2$</td>
<td>$acquire \ S1$</td>
<td>$acquire \ S2$</td>
</tr>
<tr>
<td>$t_2$</td>
<td>$release \ S1$</td>
<td></td>
</tr>
<tr>
<td>$t_3$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$t_4$</td>
<td>$release \ S1$</td>
<td>$release \ S2$</td>
</tr>
<tr>
<td></td>
<td>$a := A$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$b := B$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$C := 3$</td>
<td></td>
</tr>
<tr>
<td>$t_5$</td>
<td>BARRIER</td>
<td>BARRIER</td>
</tr>
</tbody>
</table>
Assume that variables A, B, and C are shared and the others are private; assume further that all reads and writes not shown are local. Symbol \( t_i \) denotes a point of time at which a synchronization operation (acquire, release, barrier) occurs. Different models treat this code in different ways, and they are ordered here by the amount of extra programming effort required:

- **release consistency:** \( A = 1 \) and \( B = 2 \) are guaranteed to be visible to P2 and P3 after \( t_2 \), and no later than immediately after \( t_3 \). \( C = 3 \) will be visible to P2 and P3 immediately after \( t_5 \).

- **lazy release consistency:** \( A = 1 \) and \( B = 2 \) must be seen by P2 immediately after \( t_3 \). However, this is not the case with P3, as it uses a different lock: while the interval \( t_1-t_2 \) on P1 clearly must precede the interval \( t_3-t_4 \) on P2; nothing can be claimed about the relative order of the interval \( t_1-t_2 \) on P1 and the interval \( t_3-t_4 \) on P3. Therefore, \( A = 1 \) and \( B = 2 \) will be visible to P3 only after \( t_5 \). Had P3 tried to acquire lock S1 sometime between \( t_4 \) and \( t_5 \), the values of \( A \) and \( B \) would have been propagated. \( C = 3 \), as for the release consistency, will be visible after \( t_5 \).

- **scope consistency:** \( B = 2 \) is guaranteed to be visible to P2 immediately after \( t_3 \), but no further commitments exist within the epoch. The model does not ensure visibility of \( A = 1 \) before \( t_5 \). \( A = 1 \) and \( C = 3 \) will be visible to P2 and \( A = 1, B = 2, C = 3 \) to P3 only after \( t_5 \).

- **entry consistency:** As in the previous case, \( B = 2 \) is guaranteed to be visible to P2 immediately after \( t_3 \) (provided that the lock S1 guards \( B \)), but no further assumptions are made, not even to what will be visible *after* the epoch. In order to ensure visibility of shared variables updated outside of critical sections, the programmer must either a) use read-only locks to read the shared but not propagated values when they are needed, or b) must explicitly bind them to the barrier, or c) bind the entire address space to the barrier (the latter two when visibility in the next epoch is the goal).

### 1.9. Existing Systems

The best way to give a brief overview of existing systems is through a table with itemized characteristics. Software implementations are summarized using the table in Figure DSMU9. Hardware implementations are summarized using the table in Figure DSMU10. Hybrid software/hardware implementations are summarized using the table in Figure DSMU11.

<table>
<thead>
<tr>
<th>Name and Reference</th>
<th>Type of Implementation</th>
<th>Type of Algorithm</th>
<th>Consistency Model</th>
<th>Granularity Unit</th>
<th>Coherence Policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>IVY [Li88]</td>
<td>user-level library + OS modification</td>
<td>MRSW</td>
<td>sequential</td>
<td>1KB</td>
<td>invalidate</td>
</tr>
<tr>
<td>Mermaid [Zhou90]</td>
<td>user-level library + OS modifications</td>
<td>MRSW</td>
<td>sequential</td>
<td>1KB, 8KB</td>
<td>invalidate</td>
</tr>
<tr>
<td>Munin [Carter91]</td>
<td>runtime system + linker + library + preprocessor + OS modifications</td>
<td>type-specific (SRSW, MRSW, MRMW)</td>
<td>release</td>
<td>variable size objects</td>
<td>type-specific (delayed update, invalidate)</td>
</tr>
<tr>
<td>Midway [Bershad93]</td>
<td>runtime system + compiler</td>
<td>MRMW</td>
<td>entry, release, processor</td>
<td>4KB</td>
<td>update</td>
</tr>
<tr>
<td>TreadMarks [Keleher94]</td>
<td>user-level</td>
<td>MRMW</td>
<td>lazy release</td>
<td>4KB</td>
<td>update, invalidate</td>
</tr>
<tr>
<td>Blizzard [Schoinas94]</td>
<td>user-level + OS kernel modification</td>
<td>MRSW</td>
<td>sequential</td>
<td>32-128 byte</td>
<td>invalidate</td>
</tr>
<tr>
<td>Mirage [Fleisch89]</td>
<td>OS kernel</td>
<td>MRSW</td>
<td>sequential</td>
<td>512 byte</td>
<td>invalidate</td>
</tr>
</tbody>
</table>
Clouds
[Ramachandran91] OS, out of kernel MRSW inconsistent, sequential 8KB discard segment when unlocked.

Linda
[Ahuja86] language MRSW sequential variable (tuple size) implementation dependent.

Orca

Figure DSMU9: A summary of software-implemented DSM (source: [Protic96a]).

Legend: OS—Operating System.
Comment: Note that the bulk of software DSM research starts in mid to late 80s. This author especially likes the selective approach of Munin, which uses different approaches for different data types, in almost all major aspects of DSM, and introduces the release memory consistency model. TreadMarks introduces the lazy release memory consistency model. Midway introduces the entry memory consistency model.

<table>
<thead>
<tr>
<th>Name and Reference</th>
<th>Cluster Configuration + Network</th>
<th>Type of Network</th>
<th>Type of Algorithm</th>
<th>Consistency Model</th>
<th>Granularity Unit</th>
<th>Coherence Policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memnet [Delp91]</td>
<td>single processor, Memnet device</td>
<td>token ring</td>
<td>MRSW</td>
<td>sequential</td>
<td>32 bytes</td>
<td>invalidate</td>
</tr>
<tr>
<td>Dash [Lesk92]</td>
<td>SGI 4D/340 (4 PEs, 2-L caches), loc. mem.</td>
<td>mesh</td>
<td>MRSW</td>
<td>release</td>
<td>16 bytes</td>
<td>invalidate</td>
</tr>
<tr>
<td>SCI [James94]</td>
<td>arbitrary</td>
<td>arbitrary</td>
<td>MRSW</td>
<td>sequential</td>
<td>16 bytes</td>
<td>invalidate</td>
</tr>
<tr>
<td>KSR1 [Frank93]</td>
<td>64-bit custom PE, 1+D caches, 32M loc. mem.</td>
<td>ring-based hierarchy</td>
<td>MRSW</td>
<td>sequential</td>
<td>128 bytes</td>
<td>invalidate</td>
</tr>
<tr>
<td>DDM [Hagerrsten92]</td>
<td>4 MC88110s, 2-L caches, 8-32M local memory</td>
<td>bus-based hierarchy</td>
<td>MRSW</td>
<td>sequential</td>
<td>16 bytes</td>
<td>invalidate</td>
</tr>
<tr>
<td>Merlin [Maples90]</td>
<td>40-MIPS computer</td>
<td>mesh</td>
<td>MRMW</td>
<td>processor</td>
<td>8 bytes</td>
<td>update</td>
</tr>
<tr>
<td>RMS [Luci95]</td>
<td>1-4 processors, caches, 256M local memory</td>
<td>RM bus</td>
<td>MRMW</td>
<td>processor</td>
<td>4 bytes</td>
<td>update</td>
</tr>
</tbody>
</table>

Figure DSMU10: A summary of hardware-implemented DSM (source: [Protic96a]).

Legend: RM—Reflective Memory,
RMS—Reflective Memory System,
SCI—Scalable Coherent Interface,
DDM—Data Diffusion Machine.

Comment: Note that the bulk of hardware DSM research starts around the year 90, except for some pioneering work like RMS (reflective memory system) at Gould and Encore. Stanford Dash introduces the first hardware implementation of the release memory consistency model.

<table>
<thead>
<tr>
<th>Name and Reference</th>
<th>Cluster Configuration + Network</th>
<th>Type of Algorithm</th>
<th>Consistency Model</th>
<th>Granularity Unit</th>
<th>Coherence Policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLUS [Bisani90]</td>
<td>M88000, 32K cache, 8-32M local memory, mesh</td>
<td>MRMW</td>
<td>processor</td>
<td>4K bytes</td>
<td>update</td>
</tr>
<tr>
<td>Galactica Net [Wilson94]</td>
<td>4 M88110s, 2-L caches 256M local memory, mesh</td>
<td>MRMW</td>
<td>multiple</td>
<td>8K bytes</td>
<td>update/ invalidate</td>
</tr>
<tr>
<td>Alewife [Chaiken94]</td>
<td>Sparcle PE, 64K cache, 4M local mem, CMMU, mesh</td>
<td>MRSW</td>
<td>sequential</td>
<td>16 bytes</td>
<td>invalidate</td>
</tr>
</tbody>
</table>
In software implementations, the majority of newer systems support OS (operating system) level and OC (optimizing compiler) level approaches. Most of the systems use the MRSW algorithms; however, new systems typically explore the MRMW algorithms. Sequential MCM is still represented a lot, in spite of its simplicity; newer approaches are typically oriented to more sophisticated MCMs. Granularity of the memory consistency unit tends to be larger (pages of the size below 1KB are rarely used). Invalidate consistency maintenance protocols seem to be more frequently used, compared to update protocols. Of course, the conclusions from tables of this sort will change, as the time goes by, and new research ideas find their way into research prototypes and/or industrial products.

In hardware implementations, the majority of systems support clusters with multiple processors (typically SMP). As far as the type of the ICN, the variety is large. The MRSW algorithm is more frequently used than the MRMW algorithm. Again, sequential consistency prevails. The granularity unit is at most 128 bytes. The consistency maintenance protocols are usually of the invalidate type. Again, the conclusions from tables of this sort will change, as the time goes by.

In hybrid implementations, the majority of systems is based on SMP multiprocessor clusters, using off-the-shelf microprocessors. Since these machines are of a newer-date, the MRMW and MRSW algorithms are used about equally often. For the same reason, there is a variety of MCMs used. The granularity units vary from as low as 16 bytes to as high as 8KB. Update and invalidate consistency protocols are being used about equally often.

These three tables assume the classification of DSM systems according to the type of implementation (hardware, software, or hybrid), as the major classification criterion. Other classifications are possible, too. One widely used classification is based on the access pattern characteristics, as the major classification criterion. In that classification, the major classes are: (a) UMA or uniform memory access, (b) NUMA or non-uniform memory access, and (c) COMA or cache-only memory access. Each class can be further subdivided into a number of subclasses: (a) F-COMA or Flat COMA versus H-COMA or Hierarchical COMA, (b) CC-NUMA or cache-coherent NUMA versus NCC-NUMA or not-cache-coherent NUMA, etc.

### 1.10. New Research

The field is already well established and new developments are not expected to be very exciting. Mostly, researchers work on fine improvements in performance, and on implementa-

<table>
<thead>
<tr>
<th>FLASH [Kuskin94]</th>
<th>MIPS T5, 1+D caches, MAGIC controller, mesh</th>
<th>MRSW</th>
<th>release</th>
<th>128 bytes</th>
<th>invalidate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Typhoon [Reinhard94]</td>
<td>SuperSPARC, 2-L caches, NP controller</td>
<td>MRSW</td>
<td>custom</td>
<td>32 bytes</td>
<td>invalidate custom</td>
</tr>
<tr>
<td>Hybrid DSM [Chandra93]</td>
<td>FLASH-like</td>
<td>MRSW</td>
<td>release</td>
<td>variable</td>
<td>invalidate</td>
</tr>
<tr>
<td>SHRIMP [Iftode96a]</td>
<td>16 Pentium PC nodes, Intel Paragon routing network</td>
<td>MRMW</td>
<td>AURC, scope</td>
<td>4KB</td>
<td>update/ invalidate</td>
</tr>
</tbody>
</table>

**Figure DSMU11:** A summary of hybrid hardware/software-implemented DSM (source: [Protic96a]).

**Legend:**
CMMU—Cache Memory Management Unit, NP—Network Protocol.

**Comment:**
Note that the bulk of hybrid DSM research starts in early to mid 90s. Princeton SHRIMP-2 project introduces the AURC and the SCOPE memory consistency models.
tions which are better suited for new technologies. The major crown of all these efforts will be the birth of a DSM system on a single VLSI chip, which will happen after the on-chip transistor count reaches 1 billion, or even before (if simpler processors are used within the SMP clusters).

The section to follow summarizes some of the recent developments and gives references to the original literature, so that interested readers can get a closer insight. The selection of examples to follow has been created to be in line with the major message of this book (DSM on a single VLSI chip, as the ultimate goal of microprocessor industry, according to this author).

The first commercial microprocessor with large on-chip DRAM [Geppert97] was announced in 1996 by Mitsubishi Electronics America in Sunnyvale, California (M32R/D). It is a 32-bit machine with a 16 MB DRAM on the same chip. This DRAM can be used to implement a part of distributed shared memory. Such a development is an important milestone on the way to complete DSM on a single chip, together with appropriate accelerators (like MMX, or similar). Also, such a development supports the major mission of this book and serves as an evidence of correctness of the ideas promoted by this book.

2. Advanced Issues

This part contains the author’s selection of research activities which, in his opinion, have made an important contribution to the field in the recent time, and are compatible with the overall mission of this book.

Papers [Nowatzyk93] and [Saulsbury96] describe efforts at SUN Microsystems to come up with a DSM architecture which represents a good candidate for future porting to a DSM-on-a-single-chip environment. Major conclusion of their study is that a siliconless motherboard, as a first next step towards the final goal, is achievable once the feature size drops down to 0.25 micrometers. Major highlights are summarized in Figure DSMS1.

<table>
<thead>
<tr>
<th>Origin and Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nowatzyk, et al.</td>
</tr>
<tr>
<td>SUN Microsystems</td>
</tr>
<tr>
<td>Major Highlights</td>
</tr>
<tr>
<td>- Going towards the DSM based workstation (S3.mp)</td>
</tr>
<tr>
<td>- Going towards the siliconless motherboard (LEGO)</td>
</tr>
<tr>
<td>- Using many less powerful CPUs, rather than a few brainiacs</td>
</tr>
<tr>
<td>- since the performance is limited by the “memory wall”</td>
</tr>
<tr>
<td>- Simulation studies oriented to 0.25 μm 256 Mbit DRAM</td>
</tr>
</tbody>
</table>

**Figure DSMS1:** The S3.mp and beyond

**Comment:**
The LEGO project from Sun Microsystems can be treated as an earliest effort towards a DSM on a chip, with a number of on-chip accelerators.

Paper [Lovett96] describes an effort at Sequent Corporation to come up with a commercially successful CC-NUMA machine based on the SCI standard chips. Major conclusion of their study is that once Intel becomes able to place four P6 machines on a single die (quad Pentium Pro), it will be possible to have a much more efficient implementation of their STiNG architecture (small letter “i” comes from “iNTEL inside”). Major highlights are summarized in Figure DSMS2.
The Sequent STiNG

Origin and environment
- Lovett + Clapp
- Sequent Computer Systems, Beaverton, Oregon, USA
- A CC-NUMA for the commercial market (1996)
- Major highlights:
  - Combines 4 quads using SCI
  - Quad is based on Intel P6
  - Quad includes up to 4GB of system memory, 2 PCI buses for I/O, and a LYNX board for SCI interface and system-wide cache cons
  - Architecture similar to Encore Mongoose (1995)
  - Processor consistency MCM
  - Application: OLTP

Figure DSMS2: The Sequent STiNG
Legend:
OLTP—On-Line Transactions Processing.
Comment:
The STiNG product from Sequent can be treated as one of the first and the most successful implementations based on the chips which support the SCI standard.

Paper [Savic95] describes an effort at Encore Computer Systems to come up with a board which one can plug into a PC (personal computer), to enable it to become a node in DSM systems of the RMS type. Major conclusion of their study is that the board (implemented using FPGA VLSI chips and fully operational) can be ported into a single standard-cell VLSI chip, which means that the RMS approach might be the first one to fit within the single chip boundaries. Major highlights are summarized in Figure DSMS3.

The IFACT RM/MC for Networks of PCs

Origin and Environment
- Milutinović + Tomašević + Savić + Jovanović + Grujić + Protić + Aral + Gertner + Natale + Gupta + Tran + Grant
- Supported by Encore on a contract for DEC
- Major highlights:
  a) Basic research and design in 1993
  b) Board implementation and testing in 1994
  c) Five different concept improvements for higher node counts:
    - Efficient integration of RM and MC
    - Write filtering
    - Transmit FIFO priorities
    - Caching of reflective memory regions
    - Duplication of critical resources

Figure DSMS3: The IFACT RM/MC for PC Environments
Comment:
The IFACT RM/MC for PC environments project at Encore can be treated as the first effort to implement a board which turns a PC into a DSM node (based on the reflective memory approach).

Paper [Gillett96] describes an effort at DEC to come up with a support product for their client-server systems, using the principles of RMS (this effort can be treated as a follow up effort after [Savic95], done on the top of a contract with Encore). Major conclusion of their study is that RMS still represents a successful way to go, in spite of the fact that the concept has been around for such a long time (as long as appropriate innovations are incorporated, like those selected by DEC). Major highlights are summarized in Figure DSMS4.
The DEC MC for NOWs

Origin and Environment
- Gillett
- A follow-up on the “Digital/Encore MC team” (1994/95)
- Major highlights
  a) A PCI version of the IFACT RM/MC board
  b) Digital UNIX cluster team: Better advantage of MC
  c) Digital HPC team: Optimized application interfaces (including PVM)
  d) Reason for adoption:
     - Performance potentials over 1000 times the conventional NOW
     - No compromise in cost per added node
     - Computer architecture for availability
     - Error handling at no cost to the applications

Figure DSMS4: The DEC Memory Channel Architecture

Legend:
NOW—Network Of Workstations;
HPC—High Performance Computing;
PVM—Parallel Virtual Machine.

Comment:
The DEC memory channel product is treated as one of the most successful market oriented products based on the reflective memory approach, done as a follow up effort, after a contract with Encore.

Paper [Milutinovic96] describes another effort at Encore Computer Systems to come up with further improvements of the RMS concept, for better exploitation in the I/O environment (in order for their Infinity SP “I/O pump” to continue to be, in their word, the “Fastest I/O pump on Planet.”). Major conclusion of the study is that the RMS can be further improved if it is combined with more sophisticated MCMs, and if an appropriate layer is added, which can be viewed as distributed shared I/O on the top of distributed shared memory. Major highlights are summarized in Figure DSMS5.

The IFACT RM/MC for Infinity SP

Origin and Environment
- Milutinović + Protić + Milenković + Rašković + Jovanović + Denton + Aral
- Supported by Encore, on a contract for IBM
- Major highlights:
  a) Basic research in 1996
  b) Goal: Continuing to be the highest performance I/O processor on planet
  c) Five different ideas introduced for higher performance:
     - Separation of temporal and spatial data in DSM
     - Direct cache injection mechanisms in DSM
     - Distributed shared I/O on top of DSM
     - Moving to more sophisticated memory consistency models (MIN)
     - Moving to more sophisticated memory consistency models (MAX)

Figure DSMS5: The IFACT RM/MC for Infinity SP

Comment:
The major goal of the IFACT RM/MC for Infinity SP project at Encore is to make the reflective memory approach more competitive in the performance race with other approaches acquired by industry.

Paper [Reinhardt96] describes an effort at the University of Wisconsin to come up with an approach based on NOWs and additional commodity components. Major conclusion of their study is that the level of success of a new product depends on the level of utilization of commodity components in both hardware and software domains; however, appropriate architec-
tural changes have to be done first (decoupling of functional hardware). Major highlights are summarized in Figure DSMS6.

<table>
<thead>
<tr>
<th>Origin and Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reinhardt + Pfle + Wood</td>
</tr>
<tr>
<td>University of Wisconsin</td>
</tr>
<tr>
<td>Hardware supported software DSM in NOW</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Major highlights</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Decoupling the functional hardware, for higher off-the-shelf utilization</td>
</tr>
<tr>
<td>- Typhoon-0: An off-the-shelf protocol processor + network interface</td>
</tr>
<tr>
<td>- Typhoon-1: Higher level of hardware integration</td>
</tr>
<tr>
<td>- Basic DSM functions to decouple:</td>
</tr>
<tr>
<td>- Messaging and networking (doing internode communications)</td>
</tr>
<tr>
<td>- Access control (detecting memory references for non-local action)</td>
</tr>
<tr>
<td>- Protocol processing (maintaining the global consistency)</td>
</tr>
<tr>
<td>- Commodity components in MN+AC; now also for PP (FPGAs + μPs)</td>
</tr>
</tbody>
</table>

Figure DSMS6: Typhoon and beyond

Comment:
Typhoon is a university project oriented to DSM based on off-the-shelf components in both hardware and software domains, which is the most cost-efficient approach for future DSM systems.

An interesting homework for the reader is to estimate the needed transistor count for all DSM systems mentioned so far, having in mind both the transistor count of the actual components used, as well as transistor count if only the needed resources are used. Such an exercise can bring a better understanding of the future referred to as DSM-on-a-VLSI-chip.

Another type of exercise for students, used in classes of this author, for the above mentioned systems, and for new ones to come, is to prepare one-academic-hour lectures, to explain the details to their colleagues, using the presentation strategy explained in [Milutinovic95b] and [Milutinovic96c]: (a) problem being attacked by the chosen research project, (b) essence of the existing solutions, and what is to be criticized with them, form the point of view defined in the problem statement, (c) essence of the proposed approach, and why it is expected to be better in conditions of interest, which are defined as a part of the problem statement, (d) details which deserve attention, (e) performance evaluation results, and (f) complexity evaluation results.

An important new trend in DSM research implies the building of commercially successful machines as well as the usage of IRAM approach to achieve energy efficient architectures [Fromm97, Laudon97].

3. About the Research of the Author and His Associates

The research of the author and his colleagues is described in numerous papers published after 1.1.1990. The author’s WWW presentation contains a partial list, as well as the full text for some of the papers.

Former efforts (projects fully completed) are related, but not limited, to: (a) design of an RMS board for PC environment (as already mentioned [Savic95]) and (b) design of new conceptual improvements to be used in an RMS board for the Infinity SP products of Encore Computer Systems [Protic98]. Also, (c) a simulation based comparison to determine the quality of RMS [Grujic96], in relation to the approaches taken by the potential competitors,
namely the SCI-like and the KSR-like approaches (the KSR—Kendal Square Research—is a concept of the COMA type, which did not make in its first attempt to succeed on the market, but represents a successful concept which, in the opinion of this author, will reappear and succeed, sooner or later).

Current efforts are related, but not limited, to: (a) design of the cache injection concept which helps to minimize the penalty of the first read misses in SMP and especially DSM systems, (b) characterization of the parallel applications, in order to determine the type of program which is better suited for lazy release MCM, and the type of program which is better suited for entry MCM. Also, (c) two Ph.D. research efforts have tried to come up with better consistency maintenance protocols for SMP and/or DSM [Tartalja97 and Protic97].
EPILOGUE
Each design of a new system implies a number of questions to be answered. Where to look for these answers? The answers are always in the past; one just has to know how to read the past, which is not an easy task to do.
Case Study #1: Surviving the Design of an MISD Multimicroprocessor for DFT**

This case study covers an early research and development experience of the author: absolutely all details of research and development (from the project requirements till the final prototype ready for production) were generated by the author alone (except for wire-wrapping, which was done by a technician). This is the project which brings the author to the firm belief that special purpose multimicroprocessor accelerators are an important component of future “one billion transistor machines on a single chip.”

1. Introduction

This case study is on the subject of microprocessor and multimicroprocessor based design of data modems. Presentation starts with a relatively simple uniprocessor example which is well suited for educational purposes. It ends with a relatively complex example which demonstrates the beauty of multimicroprocessor based digital signal processing.

All the work presented here was done back in 70s, completely by the author alone, during his full-time professional employment, which went in parallel with his M.Sc. and Ph.D. studies on related topics. The author was responsible for the project administration and management, project conceptualization and consulting, as well as project design and testing, all the way from idea to market.

The presentation to follow is based on the following references: [Milutinovic78, Milutinovic79, Milutinovic80, Milutinovic85a, and Milutinovic86a]. It includes both the technical facts and the postfinalization thoughts. Some of the theoretical foundations can be found in [Helstrom68].

** DFT = Discrete Fourier Transform.
2. Low-Speed Data Modem Based on a Single Processor

The example to follow corresponds to a 2400b/s CCITT V.26/A data modem. Presentation is divided into two parts—one on the transmitter design and one on the receiver design. As usual for this book, text and text comments give the global view; figures and figure captions explain the specific details.

2.1. Transmitter Design

Line signal is based on differential coding with dibits (one dibit corresponds to a pair of bits). Coding table used in this project is given in Figure X1. Discrete phase jumps create the line signal spectrum which is too wide for a voice-band transmission channel. Consequently, dibits have to be weighted with an envelope which eliminates time domain discontinuities.

<table>
<thead>
<tr>
<th>Dibit</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>( dP )</td>
<td>0°</td>
<td>+90°</td>
<td>+180°</td>
<td>+270°</td>
</tr>
</tbody>
</table>

**Figure X1:** Correspondence between dibit values and phase changes (source: [Milutinovic78]).

**Legend:**

\( dP \)—phase difference.

For this purpose, a raised-cosine envelope is typically used. In order to make the envelope incorporation related processing easier, adjacent dibits are processed in two different processing channels, and superimposed in order to form the desired line signal without discontinuities. Mathematical formula describing the line signal is given in Figure X2a. Time domain waveform of the line signal, together with its two components from two different processing channels, is given in Figure X2b.

\[
s(t) = \begin{cases} 
  b + \sin \left( \frac{2\pi(t - c)}{T_E} \right) & 0 \leq t \leq T_E - 2c, \\
  0, & T_E - 2c \leq t \leq T_E, \quad n = 1, 2, 3, 4
\end{cases}
\]

**Figure X2:** Line signal, a) formula, b) waveform (source: [Milutinovic78]).

**Legend:**
$T_e$—envelope interval,
ω—radial frequency of the carrier,
b, c—constants.

Overlapping of partial signals in two different processing channels is 50% of the time. In such conditions, the line signal spectrum has the form as indicated in Figure X3, which is compatible with the voice-band telephone channel. If the carrier frequency is 1800 Hz, about 99% of the line signal spectrum is located between 800 Hz and 2800 Hz. Consequently, the sampling rate of 5600 Hz is the lowest one which satisfies the requirement that sampling rate be twice the signal bandwidth. In practice, sampling rate has to be higher. For this project, the sampling rate of 9600 Hz was selected.

![Figure X3: Line signal spectrum (source: [Milutinovic78]).](image)

**Legend:**
f—frequency,
$G(f)/G_{max}(f^*)$—normalized signal spectrum.

In this particular case, the ratio of sampling rate and carrier frequency is an integer number (six), so only four different signal shapes can exist in any of the two processing channels, on the length of one envelope interval, as indicated in Figure X2b. This is good for implementations based on look-up tables, since the number of table entries is proportional to the number of different signal shapes which can appear in each of the two processing channels.

If sampling rate is equal to 9600 Hz, each of the four possible channel symbols (signal shapes on the length of one envelope interval) is represented with 16 samples (four out of 16 are equal to zero). Sample values for each of the four different channel symbols are given in Figure X4.

<table>
<thead>
<tr>
<th>$P_0 = 0^\circ$</th>
<th>$P_0 = +90^\circ$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.000 = 00000000</td>
<td>0.000 = 00000000</td>
</tr>
<tr>
<td>10.461 = 00001010</td>
<td>–25.223 = 11100111</td>
</tr>
<tr>
<td>–42.045 = 11010110</td>
<td>–42.124 = 11010110</td>
</tr>
<tr>
<td>–84.793 = 10101011</td>
<td>34.983 = 00100011</td>
</tr>
<tr>
<td>–0.223 = 00000000</td>
<td>119.032 = 01110111</td>
</tr>
<tr>
<td>126.704 = 01111111</td>
<td>52.832 = 00110101</td>
</tr>
<tr>
<td>101.886 = 01100110</td>
<td>–101.314 = 10011011</td>
</tr>
<tr>
<td>–52.117 = 11001100</td>
<td>–127.000 = 10000001</td>
</tr>
<tr>
<td>–119.031 = 10001001</td>
<td>–0.447 = 00000000</td>
</tr>
<tr>
<td>–35.460 = 11011101</td>
<td>84.595 = 01010101</td>
</tr>
<tr>
<td>41.886 = 00101010</td>
<td>42.281 = 00101010</td>
</tr>
</tbody>
</table>
Hardware and software necessary for implementation of the described modem transmitter are extremely simple, i.e., straightforward to design and inexpensive to implement.

Hardware includes a processor (8-bit CPU is enough), a small ROM (for the stored transmitter program and the stored channel symbol samples; 64 8-bit samples all together), an input port (for CCITT binary interface signals), and output port (as an interface to the D/A converter), a D/A converter (8-bit resolution is enough), and low-pass filter (to act as a signal integrator).

Software includes a kernel routine which superimposes the early samples of the later channel signal and the later samples of the early channel signal, plus the routines for input, differential coding, various “logistics”, and output.

An interesting homework implies that hardware and software of the transmitter are fully designed and tested (in a simulation environment), and then implemented and tested (in a real environment). The choice of the microprocessor/microcontroller or an FPGA/VLSI chip is left to each individual reader.

### 2.2. Receiver Design

This section describes a receiver design, based on a suboptimal detection technique, and an extremely simple hardware/software. Theoretical explanations of the statistical signal detection process are given in Figure X5.

According to [Helstrom68],

the probability of correct detection ($Q_d$) of the phase ($\psi$) of the signal $s(t; \psi)$ with unknown
phase, and buried in Gaussian noise with flat power density ($N_0 = 1$) is given by:

$$Q_0(\psi) = Q(d, R_0/d^2)$$

where $d^2$ and $R_0$ are SNR and phase decision threshold, respectively.

Provided the above conditions are satisfied, the impulse responses $h(t)$ of 4 filters matched to the signals $s(t; \psi); \psi = 0^\circ, +90^\circ, +180^\circ,$ and $+270^\circ$ are given by:

$$s(t) = \begin{cases} s(T_D - t; \psi), & \psi = 0^\circ, 90^\circ, 180^\circ, 270^\circ, \\ 0, & \text{elsewhere}, \end{cases}$$

$T_D$ corresponds to dibit rate.

**Figure X5:** Theoretical foundations of statistical signal detection (source: [Milutinovic78]).

**Legend:**
- $\psi$—signal phase,
- $s(t, \psi)$—signal which is to be detected.

Optimal detection is based on four matched filters—one filter for each of the four different channel signal shapes. Matched filtering implies convolution, which is a relatively complex mathematical operation to perform. However, if the transmission medium is characterized with a high signal to noise ratio and a low distortion and interference factor, a suboptimal detection based on binary matched filtering can be used. Consequently, a much simpler implementation hardware/software can be used. More on suboptimal detection in general, and binary matched filtering in particular, can be found in [Milutinovic80b, Milutinovic80c, Milutinovic84, Milutinovic85a, and Milutinovic88b].

The binary matched filtering is best applied during the non-overlapping time interval on the middle portion of each dibit interval. Signs of four different samples on the middle portion of each dibit interval define the current phase uniquely, as indicated in Figure X6. More than four samples would provide better performance in real conditions. Of course, a relatively stable timing extractor circuitry is needed in the system. Fortunately, it is relatively simple to implement for the type of signal involved in this example.

<table>
<thead>
<tr>
<th>$P_0$</th>
<th>Signs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0^\circ$</td>
<td>++++</td>
</tr>
<tr>
<td>$+90^\circ$</td>
<td>++ --</td>
</tr>
<tr>
<td>$+180^\circ$</td>
<td>-- --</td>
</tr>
<tr>
<td>$+270^\circ$</td>
<td>-- ++</td>
</tr>
</tbody>
</table>

**Figure X6:** Correspondence between sample signs and initial phase of the line signal (source: [Milutinovic78]).

**Legend:**
- $P_0$—initial phase.

Selection of sampling instants can be optimized, in order to maximize the signal to noise ratio of the detection. Figure X7 defines the maximal likelihood function to be used for selection of optimal sampling instants. Figure X8 gives one outcome of the maximal likelihood function in the noisy environment.

Optimum sampling instants are given by the abscissa of local maxima of the following function:
\[ f_s(t) = \begin{cases} \sum_{n=1}^{4} \left[ b + \sin \left( \frac{2\pi(t - c)}{T_E} \right) \cos \left( \omega_s t + \frac{(n-1)\pi}{2} \right) \right], & 0 \leq t \leq T_E - 2c; \\ 0, & T_E - 2c \leq t \leq T_E; \end{cases} \]

**Figure X7:** Optimum sampling instants for maximal signal to noise ratio (source: [Milutinovic78]).

**Legend:**

\(T_E\)—envelope interval.

**Figure X8:** Maximal likelihood function with noise (source: [Milutinovic78]).

**Legend:**

\(T_E\)—envelope interval.

Hardware requirements for implementation of the receiver are minimal. The core components are the microprocessor (an 8-bit machine is more than satisfactory), a hard limiter with a tri-state comparator (on the line side), and a latch (on the terminal side). See Figure X9 for one possible structure of the underlying hardware.

**Figure X9:** System structure (source: [Milutinovic78]).

**Legend:**

SU—synchronization unit,
L1/2—latches,
TB/C—tristate buffers,
I—interpolator.
The entire software can fit into a 1 KB PROM. It can be synchronized from the timing extractor, using the interrupt mechanism. Note that modern data modems include not only a “data pump,” but also a number of other functions.

An interesting homework implies that hardware and software of the receiver are fully designed and tested (in a simulation environment), and then implemented and tested (in a real environment). The choice of the microprocessor/microcontroller or an FPGA/VLSI chip is left to each individual reader. Note that the design can be optimized if the transmitter and the receiver can share the resources.

3. Medium-Speed Data Modem Based on a Single Processor

This section describes the design of a 4800b/s data modem. The 4800b/s modem is somewhat more complex than the 2400b/s modem. Its block diagram is given in Figure X10.

![Block Diagram of 4800b/s Data Modem](image)

**Figure X10:** Structure of a 4800 b/s CCITT compatible modem (source: [Milutinovic78]).

**Legend:**
- **TI**—terminal interface,
- **Sc**—scrambler,
- **Cod**—coder,
- **Mod**—modulator,
- **DAC**—digital to analog converter,
- **IF**—interpolating filter,
- **LI**—line interface,
- **AGC**—automatic gain control,
- **BPF**—band pass filter,
- **StEq**—statistical equalizer,
- **ADC**—analog to digital converter,
- **Dem**—demodulator,
- **Det&Dec**—detector and decoder,
- **TE**—timing extractor,
- **Desc**—descrambler,
- \(\{a_i\}\)—input data stream,
- \(\{b_i\}\)—scrambled data stream,
- \(S[t]\)—line signal.
On the transmitter side, in addition to the line signal generation, one has to implement a scrambler, as well. On the receiver side, the signal to noise margin is smaller, and one has to implement an optimal detector, together with an equalizer, and a descrambler.

### 3.1. Transmitter Design

Terminal interface is best implemented using specialized chips. Scrambler is best implemented using microprocessor code (if the microprocessor is fast enough), or an FPGA chip (if the extra hardware cost can be tolerated). The scrambler block diagram is given in Figure X11. Line signal coding and modulation follows the same theory as in the previous example. Carrier frequency is equal to 1800 Hz. Before shaping, about 95% of signal power occupies the interval from 200 Hz to 3400 Hz. After shaping, using a 50% raised cosine envelope, with overlapping of two channel signals during 33% of the time, over 95% of signal power is in the interval from 500 Hz to 3100 Hz. Sampling rate is again 9600 Hz.

![Scrambler Structure](image)

**Figure X11:** Scrambler Structure (source: [Milutinovic78]).

**Legend:**

\[ P_5(x) = 1 + x^{-6} + x^{-7} \] — generating polynomial with additional guard against repeating patterns of 1, 2, 3, 4, 6, 9, and 12 bits,

\( \{a_i\} \) — input data stream,

\( \{b_i\} \) — output data stream,

Note: if strap BC is used instead of strap AB, the scrambler turns into the descrambler; the clock input is implied.

The CCITT V.27 coding scheme implemented in the modem under consideration is given in Figure X12. The left-hand digit of the tribit is the one to occur the first in the data stream. The phase change is calculated as the actual on-line phase shift in the transition region, from the center of one signaling element to the center of the following signaling element.

<table>
<thead>
<tr>
<th>Tribit values</th>
<th>Phase values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1</td>
<td>0°</td>
</tr>
<tr>
<td>0 0 0</td>
<td>45°</td>
</tr>
<tr>
<td>0 1 0</td>
<td>90°</td>
</tr>
</tbody>
</table>
The two components making the line signal are shown in Figure X13, the detailed mathematical explanation of the line signal is given in Figure X14, while the internal organization of program and data are given in Figure X15. All necessary explanations are given in the captions of the related figures.

**Figure X12:** The CCITT V.27 coding scheme implemented in the modem under consideration (source: [Milutinovic78]).

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
<th>135°</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>180°</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>225°</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>270°</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>315°</td>
</tr>
</tbody>
</table>

**Figure X13:** Two components making the line signal (source: [Milutinovic78]).

Legend:
- φ₀—initial phase of the signal, in the center of the signaling interval,
- Tₛ—sampling period in the transmitter,
- Tₑ—envelope interval,
- TOV—overlapping interval (of two consecutive symbols),
- Tₑ—carrier period,
- TB—signaling interval,
- cy—basic cycle of the microprocessor system,
- X₁—the first occurring bit of the tribit X₂X₃X₁
  (in this example, it is causing the appearance of the line symbol with φ₀ = 0°),
- 010—the tribit occurring as the next to X₂X₃X₁ in the stream
  (in this example, it is causing the appearance of the line symbol with φ₀ = 90°),
- INTREQ—signal on the microprocessor pin INT,
- t₄₈, t₉₈, tₑ—the interrupt request instants (in the transmitter),
- tw₁, tw₂—signaling limits (interrupt request instants in the receiver).

\[
s_A(t) = \begin{cases} 
0; & -3T \leq t \leq -2T \\
1/2 + \cos\left(\frac{2\pi}{6T}t\right) \sin\left(2\pi f_c t - \frac{n\pi}{4}\right); & -2T \leq t \leq 2T \\
0; & 2T \leq t \leq 3T 
\end{cases}
\]

\[n = 20, K, 7; \quad f_c = 1800 \text{Hz}; \quad T = (1/4800 \text{Hz})\]
The analytical expression for the signal on the envelope interval can have only 8 different forms. There are 12 samples in total, on condition that the 9.6 kHz sampling frequency is chosen.

The values of these samples (8 groups with 12 samples in each group) normalized with the constant 127 are given by the following expression:

\[
s_A(i, n) = \begin{cases} 
0; & i = 1, 2 \\
(254/3) \left[ 1/2 + \cos \left( \frac{\pi n}{6} \right) \right] \sin \left( \frac{3\pi i}{8} + \frac{n\pi}{4} \right); & i = 3, K, 9 \\
0; & i = 10, 11, 12 
\end{cases}
\]

\[-127 \leq s_A(i, n) \leq 127; \quad i = 1, K, 12; \quad n = 0, K, 7\]

**Figure X14:** Analysis of the line signal (source: [Milutinovic78]).

**Figure X15:** Memory organization of the transmitter design (source: [Milutinovic78]).

Legend:
- ■—memory occupied by code (program or data),
- □—free memory space,
- ProgramMemory—memory space occupied by program code,
- DataMemory—memory space occupied by data code,
- Page No. 0—memory page with initialization routines,
- Page No. 1—memory page with the precoding routine,
- Page No. 2—memory page with the signal forming routine,
- Page No. 3—memory page with 8 possible groups of signal samples (existing on the envelope interval).

Block scheme of the transmitter hardware is given in Figure X16. This hardware includes a microprocessor, a ROM which is organized as in Figure X15, and no RAM memory (internal registers of the microprocessor are enough to fulfill the need for temporary storage).
Figure X16: Transmitter structure (source: [Milutinovic78]).

Legend:
MSB—the most significant bit,
LSB—the least significant bit,
RTS—request-to-send,
$S(t)$—line signal,
$S_0(t)$—zero-order polynomial-based approximation of the line signal,
$T$—bit interval,
$S_s$—synchronization switch ($S_s = \text{ON}$: modem master; $S_s = \text{OFF}$: terminal master),
S/P—serial-to-parallel converter,
IF—second-order low-pass filter as interpolating filter,
DAC—eight-bit digital-to-analog converter,
R—appropriate resistor.

The basic clock of this example is equal to 320.5 ns. This means that the number of basic clock intervals (defined by the 3.12 MHz clock frequency) is within one signaling interval (defined by the frequency of 1600 Hz) is equal to 1950. In the case of a faster microprocessor, this basic clock budget would be much higher, and would enable the scrambler code to be included.

Wiring at the input port of the microprocessor system enables a number of different coding schemes to be implemented. The CCITT V.27 coding scheme is achieved with straps in Figure X15 arranged as AD+BE+CF. Output port is interfacing a D/A converter based on the offset binary coding scheme.

Initialization code flow chart is given in Figure X17, while the operation code flow chart is given in Figure X18. An interesting homework is to implement and test the assembly/machine
level coding which realizes the entire transmitter. An advanced homework implies that the hardware/software be redesigned using a state-of-the-art technology.

**Figure X17:** Flow chart of the transmitter initialization code. (source: [Milutinovic78]).

**Legend:**

DAC—Digital-to-Analog Converter.
3.2. Receiver Design

As indicated earlier, optimal signal detection is a must for medium or higher transmission rates. A symbolic representation of the detection process is given in Figure X19, and its mathematical explanation in Figure 20.
Figure X19: Symbolics of the detection process (source: [Milutinovic78]).

Legend:

$q\Phi$—previous signal in “quasiphase,”
$qQ$—previous signal in “quasi quadrature,”
$\Phi$—inphase projection of current signal,
$Q$—quadrature projection of current signal,
sign $Q$—first bit to occur at the receiver output,
sign $\Phi$—second bit to occur at the receiver output,
logical value of $|\Phi|>|Q|$—third bit to occur at the receiver output,
TRUE $\rightarrow$ bit 1,
FALSE $\rightarrow$ bit 0.

After the formation of quantities $\Phi$ and $Q$, sign $Q$ is the first bit to be sent from the modem receiver, followed by sign $\Phi$, and bit ‘0’ finally, as the third bit, if $|\Phi|>|Q|$ is false, or bit ‘1’ if $|\Phi|>|Q|$ is true.

Phase and quadrature projections of the signal are formed as

$$\Phi = \sum_{i=1}^{N} s_i \ast s_i q^{qQ(-T)} \quad (1)$$

$$Q = \sum_{i=1}^{N} s_i \ast s_i q^{Q(-T)} \quad (2)$$

Figure X20: Mathematics of the detection process (source: [Milutinovic78]).

Legend:

$N$—number of samples ($N = 4$),
$s_i$ ($i = 1, \ldots, N$)—signal samples from the current signaling interval,
$s_i q^{(\Phi(-T))}$ ($i = 1, \ldots, N$)—signal samples from the previous signaling interval (after the $-22.5^\circ$ phase shifter),
$s_i q^{Q(-T)}$ ($i = 1, \ldots, N$)—signal samples from the previous signaling interval (after the $+67.5^\circ$ phase shifter).

This example implies a sampling rate of 7200 Hz, which means that four statistically independent signal samples are available within one line signal carrier interval without overlapping (of signals from two adjacent tribit intervals). This is clearly described in Figure X21.
Figure X21: Clocking of the detection process (source: [Milutinovic78]).

Legend:
- $T_c$—carrier period,
- $T_{baud}$—baud period,
- $T_s$—sampling period,
- $t_A$, $t_B$—symbol limits,
- $t_1$, $t_2$, $t_3$, $t_4$—nominal sampling instants.

Details of the sampling process, for all eight outcomes of the line signal, and the two local carriers (in-phase and quadrature), are given in Figure X22 symbolically, and in Figure X23 numerically.

Figure X22: Sampling of the detection process (source: [Milutinovic78]).

Legend:
During the demodulation procedure it is necessary to calculate the following:

\[
\Phi = s_1(\pm 135\degree) \cdot s_2(\pm 22.5\degree) + s_3(\pm 135\degree) \cdot s_4(\pm 22.5\degree) \\
+ s_3(\pm 135\degree) \cdot s_3(\pm 22.5\degree) + s_4(\pm 135\degree) \cdot s_4(\pm 22.5\degree)
\]

\[
Q = s_1(\pm 135\degree) \cdot s_1(\pm 67.5\degree) + s_2(\pm 135\degree) \cdot s_2(\pm 67.5\degree) \\
+ s_3(\pm 135\degree) \cdot s_3(\pm 67.5\degree) + s_4(\pm 135\degree) \cdot s_4(\pm 67.5\degree)
\]

Taking the specific case into account, the following is obtained:

\[
\Phi = 0 \cdot (\cos 67.5\degree) + 1 \cdot (\cos 22.5\degree) + 0 \cdot (\cos 67.5\degree) - 1 \cdot (\cos 22.5\degree) \\
= -2 \cos 22.5\degree = -1.848
\]

\[
Q = 0 \cdot (\cos 22.5\degree) + 1 \cdot (\cos 67.5\degree) + 0 \cdot (\cos 22.5\degree) - 1 \cdot (\cos 67.5\degree) \\
= +2 \cos 67.5\degree = 0.765
\]

So, if sign \(Q = 0\) is sent to the terminal first, sign \(\Phi = 1\) second, and bit ‘1’ (since \(|\Phi| > |Q|\)) last, the receiver output will correspond to the transmitter input, i.e. tribit \(011\).

If envelope shaping is introduced, the first and the fourth product in expressions (5) and (6) are decreased relative to the second and third, and nothing else is changed.

**Figure X23:** Numerics of the detection process (source: [Milutinovic78]).

Block diagram of the receive hardware is given in Figure X24 and the organization of the ROM and RAM memory in Figure X25. The concrete mapping of ports into the memory address space reflects the state of the technology at the time of the design.

**Figure X24:** Hardware for the detection process (source: [Milutinovic78]).

**Legend:**
P1, P2, P3—input ports,
P4—output port,
M1, M2—multiplier inputs,
M3—high-order byte of the multiplier output,
S&H—sample-and-hold circuit,
AD—analog-to-digital converter,
CL—control logic generating support signals for S&H and AD,
CS —chip-select signal for port P1,
Φ2TTL—basic microprocessor clock: 3.12 MHz,
S/H —control signal for S&H circuits,
SOC—control signal for ADs,
MSB—the most significant bit,
P/S—parallel-to-serial converter,
PhSh—analog phase-shift circuit,
A, B, C, D, E, F—straps for matching the coding scheme of the receiver to that of the transmitter; nominal strap configuration AD + BE + CF,

Comment:
The propagation time through all phase-shift circuits must be the same.

![Diagram of Memory Organization](image)

**Legend:**
INIC—initialization routine,
REC—main program in the receiver,
SAMP—memory space for “old” and “new” envelope samples,
$qQ(S_i)$—symbolical memory address of the $i$-th sample from the “quasiquadrature” channel;
$(i = 1, \ldots, 4),$
$q\Phi(S_i)$—symbolical memory address of the $i$-th sample from the “quasiphase” channel;
$(i = 1, \ldots, 4).

Flow chart of the initialization code is given in Figure X26. The operation code is written on the assumption that the microprocessor will halt after the completion of all necessary activities between two adjacent signal samples. After the next signal sample is ready, an interrupt occurs and the operation code restarts again. Flow chart of the operation code is given in Figure X28. Again, redesign using a state-of-the-art technology is left to the reader as a homework.
**Figure X26:** Flow chart of the receiver initialization code (source: [Milutinovic78]).

**Figure X27:** Timing of the receiver code (source: [Milutinovic78]).

**Legend:**
- REC—time interval covered by the main program in the receiver,
- HALT—time interval when the processor is in the halt state,
- $t_a, t_b$—limits of the signaling interval,
- $t_1, t_2, t_3, t_4$—sampling instants.
Finally, after a piece of engineering is completed, one has to determine the complexity and the performance of the system. In real world, the complexity question is answered by the cost analysis. The performance question is often answered through an error probability versus the signal to noise ratio family of curves, as indicated in Figure X29.
Another question which is related both to complexity (cost factor) and performance (error rate) is the maximal quantization noise which can be tolerated in given circumstances. Results of the related analysis are given in Figure X30.

**Figure X30:** Impact of the quantization noise (source: [Milutinovic78]).

Legend:
- $P_e$—error probability,
- $j$—jitter in percent of the signaling interval time,
- ADC—analog-to-digital converter.

4. Medium-Speed Multimicroprocessor Data Modem for High Frequency Radio

This example, from the computer architecture point of view, is essentially an MISD (Multiple Instructions Single Data) design. From the processing algorithm point of view, it is es-
sentially a DFT (Discrete Fourier Design) using only a subset of possible output values. All algorithmic details for both the transmitter and the receiver are given and explained in Figure X31. The overall structure of the design is given and explained in Figure X32.

During a single signaling interval lasting $T = 13.33$ ms, composite signal is given by:

$$s(t) = \sum_{n=1}^{16} \sin\left[2\pi f_n (t - kT) + \phi_{n,k}\right], \quad kT \leq t \leq (k + 1)T, \quad k = 0, 1, 2, K$$

Symbols $f_n = 55 \cdot (15 + 2n)$ Hz, $n = 1, K, 16$, and $\phi_n = l\pi/4, l = 0, K, 7$, refer to subchannel carrier frequency and its initial phase, respectively. After being corrupted by multiplicative and additive disturbances, the signal received is decomposed on the basis of mutual orthogonality of subsignals over the middle $T_0$ portion of the signaling interval.

The frame is reconstructed according to the signs of $\Phi_{n,k}$ and $Q_{n,k}$, where

$$\Phi_{n,k} = \int_{kT+t_1}^{kT+t_1+T_0} s(t) \cos\left[2\pi f_n (t - kT) + \phi_{n,k-1}\right]dt$$

$$Q_{n,k} = \int_{kT+t_1}^{kT+t_1+T_0} s(t) \sin\left[2\pi f_n (t - kT) + \phi_{n,k-1}\right]dt$$

$n = 1, K, 16$

$k = 0, 1, 2, K$

Here $t_1$ being related to the beginning of the signaling interval. In the case of digital realization, with the samples $s(t_j), j = 0, 1, 2, \ldots$ being $\Delta T_j = 1/(7040$ Hz) apart, it is:

$$\Phi_{n,k} = \sum_{j=1}^{J=64} s(t_j) \cos(2\pi f_n t_j + \phi_{n,k-1}) = \sum_{j=1}^{J=64} \phi_{j,n,k}$$

$$Q_{n,k} = \sum_{j=1}^{J=64} s(t_j) \sin(2\pi f_n t_j + \phi_{n,k-1}) = \sum_{j=1}^{J=64} Q_{j,n,k}$$

$n = 1, K, 16$

$k = 0, 1, 2, K$

Finally, the phase difference between the received signal carrier ($\phi_{n,k}$) and the local oscillator ($\phi_{n,k-1}$) is measured and the local oscillator phase is updated in all subchannels in order to meet differentially coherent detection requirements.

Measuring accuracy of $2.8125^\circ$ proves to be satisfactory.

**Figure X31:** Mathematics of the HF data modem signal forming and detection (source: [Milutinovic78]).
Figure X32: Structure of the receiver (source: [Milutinovic78]).

Legend:
- \( \{a_k\} \) — transmitting bit stream \( k = 0, 1, \ldots \),
- \( \{b_k\} \) — receiving bit stream \( k = 0, 1, \ldots \),
- \( C(f, t) \) — fading channel transfer function,
- \( n(t) \) — additive noise,
- T.E. — timing extraction,
- DPL — digital phase locking,
- OSC — crystal oscillator,
- AGC — automatic gain control,
- AFC — Doppler effect correction,
- F1 — 75 Hz clock with phase locked to the incoming signal frame transitions,
- F2 — 2400 Hz clock synchronized with F1.

4.1. Transmitter Design

Transmitter design follows the same strategy as in the previous two examples. The only minor complication is that sinusoidal signals from 16 different channels have to be superimposed before the signal is sent out. On the other hand, this type of signal involves no envelope based shaping. Consequently, the design is easier, compared to the previous examples, and will not be further elaborated here.

4.2. Receiver Design

The most sophisticated part of the receiver design is the signal detector. Its structure is given in Figure X33. It includes a master processor which receives an interrupt request from the timing extractor unit. After the interrupt request is accepted, the appropriate preparational work is performed, and then the master starts sending interrupts periodically to each one of the 16 slave processors.

Altogether, master processor sends 64 interrupt requests to slave processors, as 64 triggers for accepting 64 new samples of the line signal. These samples are created by the A/D converter in the master processor, and are passed to all 16 slave processors, through one output port in the master processor and 16 input ports in 16 slave processors (one port per processor).

Each slave processor calculates the in-phase and the quadrature component of the line signal projection onto the local carrier in the channel corresponding to that particular slave processor, as explained in Figure X31. All the in-phase components (16 altogether) and all quadrature components (16 altogether) are passed back to the master processor, using 16 output ports in 16 slave processors (one port per processor) and 16 input ports in the master processor.
Final estimation of 16 phases for 16 channel signals of the composite line signal is done by the master processor, using the formula specified in Figure X32. Exact timing of all above described activities is described in Figure X35.

\[ \Delta \Phi_{n,k} = \arctan \left( \frac{Q_{n,k}}{\Phi_{n,k}} \right), \quad n = 1, K, 16, \quad k = 0, 1, 2, K \]

Figure X33: Structure of the detector (source: [Milutinovic78]).
Legend:
PO—output port,
PI—input port.

Figure X34: The detection formula (source: [Milutinovic78]).

Figure X35: Timing of the receiver code (source: [Milutinovic78]).

The line signal includes peaks which happen periodically, when it happens that all 16 channel signals are in phase, inside the same signaling interval. Consequently, the ratio of peak to average signal power is relatively large and some kind of limiter in the system could increase the performance for the given signal to noise ratio. Figure X36 gives some information on the impact of the limitation factor.

<table>
<thead>
<tr>
<th>L [%]</th>
<th>( Q_s ) &amp; ( \Phi_s )</th>
<th>( Q_w ) &amp; ( \Phi_w )</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>50</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>75</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>100</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Figure X36: Number of bits required in representation of final \( \Phi_{n,k} \) and \( Q_{n,k} \) for various values of the limitation factor \( L \) (source: [Milutinovic78]).
Presented example can serve the basis for three types of homework assignment. First, all hardware and software details have to be elaborated for the technology used originally in this particular example. Second, the example can be redone using a state-of-the-art technology. Third, an alternative design philosophy can be utilized, which might be better suited for the state-of-the-art technology.

5. Experiences Gained and Lessons Learned

As indicated in the preface of this book, all hardware schematics and all machine programs, for the 17-microprocessor design and the other examples presented in this case study, have been created by the author himself, without anybody’s help, except for the physical process of wire wrapping and/or board printing. Also, all testing of the design, of the wire wrapped and/or printed circuit boards, and of the entire system, including the field testing based on real high frequency radio transmission, have been performed by the author alone.

The crown of the entire project was the realization that performance was better compared to the same products of the competing companies on the world market, and that the market price could be much lower, comparing to the same competition! The total price of the components to build in was more than an order of magnitude smaller than the price of high frequency radio modems available on the market in those days.

The entire work was done immediately after the graduation, which meant no previous practical experience. The obvious question was how all this was possible, especially having in mind the fact that each printed circuit board and each piece of software worked from the first try; the testing with the real high frequency radio channel also worked from the first try. The lines to follow will try to create an answer.

First, since this was the author’s first independent fight with real implementations, he had created a high level of respect for all elements of the engineering work on the road to the final goal. Design with a component would start only after all its details were fully mastered from the data sheets; this meant reading the data sheets over and over again for dozens of times, before a design attempt would start. Also, all details of the development equipment would be fully mastered (from user manuals) before an attempt to use that equipment.

Second, testing in early phases of the design was given the highest priority in the entire project. A careful testing on the functional level was done, both in ideal cases and in the cases of all possible transmission disturbances that the author was able to envision with his knowledge in those days. Next, after the hardware schematics and machine code were completed, an exhaustive logical testing using software packages was done, over and over again, until the vision was created that everything is finally OK. Progress through the hardware and the software was step-by-step, using the most incremental steps that one can imagine, and doing exhaustive (and often unnecessary) testing at each particular step (of progress through the schematic and/or the code).

Third, everything was documented on-line, and a working day would never be completed, unless all related documenting was completed. Simply, facts would hit the paper while everything was still clear in the mind and memory. Each element of the documentation was double checked.

Fourth, the project has become a part of the life and the life has become a part of the project. Often, an awareness about a wrong assumption would wake up the author in the middle of the night. During the critical phases of the design and testing, he was completely useless
for the family life. Also, if something looked like ready for doing, the author would let a night’s sleep help the idea to re-mature, before the action is started.

All these time consuming activities did not mean that the project took more time than initially scheduled. On the opposite, it was one of the rare projects in the laboratory which was fully completed well before the originally scheduled deadline.

No piece of engineering is completed without any errors made. These errors are in the technical and non-technical domains. As far as the examples described above, a technical error was to start with the algorithmic approach of the competing companies, before it was checked if there exists another algorithmic approach (not taken by others, but present in the open literature) which could potentially bring a better performance (in conditions of the high frequency fading) for a smaller implementational complexity (in conditions of the state-of-the-art technology). On the other hand, some non-technical errors were made as well; the joy of success (for some of the partial goals on the way to the final goal) was sometimes shared with others in the laboratory in a way which could be treated by older colleagues as an attempt to demonstrate that enthusiasm of a youth means more than the experience of a lifetime. Sometimes it takes decades to realize that it is actually the opposite.
Case Study #2: Surviving the Design of an SIMD Multimicroprocessor for RCA††

This case study covers an early research and development experience that this author was a part of. The author was responsible for the architecture-technology interface. This is another project which brings the author to the firm belief that special purpose multimicroprocessor accelerators are an important component of future “one billion transistor machines on a single chip.”

1. Introduction

The case study described here represents a pioneering effort to design and implement a GaAs systolic array for applications in the domain of radar and communications systems. The author was invited to help on this project after the 200 MHz GaAs RISC microprocessor design project was completed.

First, the author’s role was to help on the general research and development strategies of RCA by creating the ideas and soliciting the proposals [Milutinovic85c]. Second, the author’s role was to help on the specific research and development tactics of RCA by creating the suggestions and evaluating the proposals on how to best utilize the existing experiences from the 200 MHz GaAs RISC microprocessor design project which was just completed [Milutinovic85c]. Third, during the concrete design and testing process, the author’s role was of the consulting nature; the major design and testing role was taken by others, and the technical details described later in this text are to be credited mostly to professor Jose Fortes [Fortes86].

The presentation to follow is based predominantly on the following references: [Fortes86], [Helbig89], [Milutinovic85c], and [Milutinovic86c]. For algorithmic details and the architectural details, the readers are referred to references: [Bierman77], [Manzingo80], and [Kung88].

†† RCA = Radar and Communications Applications or Radio Corporation of America, as appropriate in the given context.
2. GaAs Systolic Array Based on 4096 Node Processor Elements

Adaptive signal processing is of crucial importance for advanced radar and communications systems. In order to achieve real-time throughput and latencies, one is forced to use advanced semiconductor technologies (e.g., gallium arsenide, or similar) and advanced parallel architectures (e.g., systolic arrays, or similar).

The systolic array described here was designed to support two important applications: (a) adaptive antenna array beamforming, and (b) adaptive Doppler spectral filtering. In both cases, in theory, the system output is calculated as the product of the signal vector \( \mathbf{x} \) (complex \( N \)-dimensional vector) and the weight vector \( \mathbf{w} \) (optimal \( N \)-dimensional vector).

Complex vector \( \mathbf{x} \) is obtained by multiplying \( N \) input samples with the corresponding window weighting function consisting of \( N \) discrete values. Optimal vector \( \mathbf{w} \) is obtained as:

\[
\mathbf{w} = \mathbf{R}^{-1}\mathbf{s}^* = \mathbf{M}^{-1}\mathbf{v}^* .
\]

Symbol \( \mathbf{R} \) refers to the \( N \)-by-\( N \) inverse covariance matrix of the signal with the \((i,j)\)-th component defined as:

\[
r_{ij} = x_i^* x_j^T ,
\]

and symbol \( \mathbf{s}^* \) refers to the \( N \)-dimensional vector which defines the antenna direction (in the case of adaptive antenna beamforming) or Doppler peak (in the case of adaptive Doppler spectral filtering).

Symbols \( \mathbf{M} \) and \( \mathbf{v} \) represent scaled values of \( \mathbf{R} \) and \( \mathbf{s} \), respectively. In practice, the scaled values \( \mathbf{M} \) and \( \mathbf{v} \) may be easier to obtain, and consequently the remaining explanation is adjusted.

The core of the processing algorithm is the inversion of a \( N \)-by-\( N \) matrix in real time. This problem can be solved in a number of alternative ways which are computationally less complex. The one chosen here includes the operations explained in Figure Y1a.

Positive semi-definite matrix \( \mathbf{M} \) can be defined as:

\[
\mathbf{M} = \mathbf{UDU}^T .
\]

Matrices \( \mathbf{U} \) and \( \mathbf{D} \) are defined using the formula:

\[
\mathbf{M}_K = \mathbf{U}_K \mathbf{D}_K \mathbf{U}_K^T = \sum_{i=1}^{K} x_i x_i^T ,
\]

which is recursively updated using the formula:

\[
\mathbf{U}_K \mathbf{D}_K \mathbf{U}_K^T = \mathbf{U}_{K-1} \mathbf{D}_{K-1} \mathbf{U}_{K-1}^T + x_K x_K^T .
\]
Figure Y1: Basic Operational Structure.

Legend:
SAA1—Cells involved in root covariance update, and the first step of back substitution;
SAA2—Cells involved in root covariance update, and in both steps of back substitution;
U—Lower triangular matrix with unit diagonal elements;
D—Diagonal matrix with positive or zero diagonal elements;
b—A scalar initially set to 1;
K—Iteration count.

Consequently, matrix inversion of a direct solution reduces to back-substitution of an iterative algorithm. This means that a simpler systolic array architecture can be developed to perform the task efficiently. The one which is the subject of this example implies that (as indicated in Figure Y1a):

\[ u_{ii} = 1. \]

The solution proposed by Fortes consists of a triangular systolic array with \( N(N - 1)/2 \) nodes, where \( N \) refers to the size of the problem, as defined above. Note that matrix elements along the diagonal differ in treatment from the rest of the matrix elements.

Realization based on a full systolic array implies three systolic waves of computation, as indicated in Figures Y1b, Y1c, and Y1d, for the specific case when \( N = 6 \). As indicated in [Fortes86], the three waves accomplish the following functions:

Wave #1 = Covariance matrix updating. The data vector \( x \) enters the array at the top, and propagates as indicated in Figure Y1b.

Wave #2 = The first step of back substitution. Input for the first step of back substitution enters the array at the top, and propagates as indicated in Figure Y1c. Output is an intermediate vector to be used in the third wave, for the final computation of the weights \( w \).
Wave #3 = The second step of back substitution. Input for the second step of back substitution enters the array at the bottom, and propagates as indicated in Figure Y1d. The weights $w$ are computed sequentially.

The full systolic array implementation can be optimized. Waves #1 and #2 can run simultaneously, which is not the case with wave #3, which has to wait for the first two waves to complete before its backwards propagation can start; also, wave #3 has to have access to old covariance values, which means that memory is required inside the systolic array. Consequently, if the array includes all necessary processing elements and data memory, each iteration of the algorithm can perform in two passes, as indicated in Figure Y1e:

Pass #1 = This pass includes the waves #1 and #2. At the end, the updated values of the covariance matrix $U$ are stored into the data memory of the systolic array (essentially, a shift register).

Pass #2 = This pass basically includes the wave #3. At the end, final values of weights $w$ are generated, one value per each clock cycle (pipelining is possible, as indicated later in the text).

Latency of this solution is of the order of $2N$, calculated from the new input data vector to the new output weights vector. Throughput of this solution is defined by the minimal time between two successive issuance of the input data vector (and can be made equal to the order of $N$, using the appropriate design), which is of importance for pipelining.

The solution also includes elements of fault-tolerant computing, which is of crucial importance for GaAs technology, due to its immaturity. Figure Y1f shows the basic array ($128 \times 128$) augmented with 16 extra rows and 16 extra columns, to make it fault tolerant (to a degree proportional to the investment). A reconfigured solution of the same basic size ($128 \times 128$), with one extra row and one extra column, and one faulty processor, is shown in Figure Y1g. The presented solution implies that the systolic array is periodically tested for faults, and reconfigured appropriately when a fault is detected. Of course, test vectors and data vectors can be interleaved, so that performance is not degraded. The solution of Fortes also provides graceful degradation, after the capabilities of extra rows and columns have been exhausted.

Finally, a larger triangular systolic array can be formed using smaller triangular systolic arrays and square systolic arrays as elements. Figure Y1h shows how a larger triangular array can be generated using smaller square arrays. The replication can also be done in time, using time multiplexing.

One of the roles of the author was to investigate the required redesign of the GaAs microprocessor from [Helbig89], in order to make it appropriate for this type of GaAs systolic array design, from both the complexity reduction and speed acceleration points of view; another role was to investigate the design modifications which could enable better latency and throughput. These tasks can be a nice homework exercise for interested readers of this book.

3. Experiences Gained and Lessons Learned

The major experience gained from this project was that it can be dangerous to run ahead of the time. Simply, the project was so much ahead of its time, and the failure was inevitable. In such complex projects, the sponsor prefers a fixed-amount contract to protect himself/herself from bad surprises. For the same reason, the implementor prefers a “floating-amount” contract.
The major lesson learned from this project was that the role of the market is much more important than initially envisioned by someone who is a scientist and an engineer before anything else. Another lesson learned was that, due to a hype that a new technology can create, even papers that describe a piece of non-completed engineering can obtain an award of one kind or another.
Case Study #3:
Surviving the Design of an MIMD Multimicroprocessor for DSM‡‡

This case study covers a recent research and development effort in which this author was a project leader. This project is the one which brought the author to the form belief that the solution for a future “one billion transistor chip” is to have an entire DSM system on a single chip.

1. Introduction

The case study described here represents a pioneering effort to come up with a printed circuit board which plugs into a personal computer and enables it to become a node in a distributed shared memory (DSM) system based on the reflective memory (RM) approach. The work also included research on the possible changes in the existing RM concept, and the existing RM boards, so that the concept and the board can be made better suited for the environments in which the number of nodes may be larger (initial RM systems included only up to 9 nodes). Actual design started after the research was over.

First, the role of the author was to create the project goals and the to define the project milestones. In the research/development part, the main goal was to come up with as many ideas as possible, aimed at performance improvement and/or complexity reduction (and the main role of the author was to help induce the inventivity among the members of the team which consisted of a younger colleague and several graduate students). In the design/implementation part, the main goal was to come up with an almost exhaustive testing strategy before the prototype is built so that testing on the prototype results in the minimal number of discovered design bugs (as opposed to a frequently used strategy by impatient engineers which implies that the major testing is done after the prototype hardware is implemented). Second, the role of the author was to evaluate the architectural and the algorithmic improvements generated by others. Third, the role of the author was to create evaluation strategies for the design and testing related details created by graduate students.

‡‡DSM = Distributed Shared Memory.
The presentation to follow is based predominantly on the following references: [Grujic96], [Milutinovic92], [Protic96a], and [Savic95]. The last reference describes the suggested improvements from the research/development part and defines the board blocks for the design/implementation part.

2. A Board Which Turns PC into a DSM Node
   Based on the RM Approach

   The RM approach is essentially a write-through update type of DSM. In theory, each node includes its private memory and a portion of the distributed shared memory. Addressing portions of the distributed shared memory are replicated. When a node writes to its own part of the distributed shared memory, the data also go onto the interconnection network (typically a bus or a ring), and gets written into the distributed shared memory of all nodes that might need or will need that particular data. Consequently, the reading is always satisfied in the local part of the distributed shared memory, and data consistency is preserved.

   The type of data consistency supported depends on the philosophy of the system software (see [Protic96a] for a survey of possible approaches to data consistency in DSM systems) and the concrete hardware design (there is a transmit FIFO buffer, as well as a receive FIFO buffer, on the interface between the node and the interconnection network—data may be deleted and/or bypassed while in a FIFO). The basic operational structure of an RM system is shown in Figure Z1a.

---

PC = Personal Computer; DSM = Distributed Shared Memory; RM = Reflective Memory.
Figure Z1: Basic Operational Structure.

Legend:
DMA—Direct Memory Access;
TMI—Transition Module Interface.

Comment:
An important characteristic of this approach is that it can be implemented using off-the-shelf and FPGA components only.

Suggested improvements (compared to the previous generation of boards for workstation environments) included, but were not limited, to the following: (a) caching of RM regions, (b) multiple RM buses, (c) urgent RM bus requests, (d) write filtering in the transmit FIFO, and (e) incorporation of multiple memory modules. Details can be found in [Savic95].

Suggested improvements did not represent a major technical contribution to the field, but did help improve the overall performance of the system, in conditions of a larger node count, and consequently a larger interconnection network traffic.

The basic operational structure of the implemented board is shown in Figure Z1b. The part below the TMI cable was reused from a previous design. The part above the TMI cable was reimplemented having in mind the impact of the above mentioned five suggested improvements. The personal computer interface was based on the EISA standard, since the PCI standard was non-existent at the time of the design of this board.

Interested readers can try to redesign the details of the basic solution, as well as the details of the suggested improvements. It is suggested, while thinking about the actual design, to try to come up with the own ideas on how to do better in conditions of the state-of-the-art technology.
3. Experiences Gained and Lessons Learned

An important experience gained from this project was that it is not enough to work fast, if one works on a subject which is potentially hot from the market point of view. In addition, one has to do a careful look ahead, which will put him on the right research and design track well before the others realize that the topic is potentially of high market interest. Running alone after a head start gives better chances than running in a group after a late start.

An important lesson learned from this project is that sometimes, for reasons beyond engineering, a company may decide to sell rights for a potentially successful product, and unless appropriate precautionary steps are made, the credit may go to others, and not to the pioneers.
RESEARCH PRESENTATION
METHODOLOGY
This part contains two texts which are potentially useful for young researchers: (a) about a research methodology and writing of research papers, and (b) about preparation and usage of transparencies for presentation of research results.
The Best Method for Presentation of Research Results

The major goal of this section is to serve as a guideline for organization of research presentations in oral or written form. Another important goal of this paper is to convince the researchers to use the author’s semantics-based layout strategy for transparencies. The major purpose of the entire effort is to make the research presentations as easy to comprehend as absolutely possible. Proper usage of the guidelines and strategies defined in this paper is a conditio sine qua non for those graduate students who have chosen that the author of this paper be their major professor. The same structure is being used for thesis work, as well as for conference and journal publications, or technical reports to research sponsors, both by graduate students and professional engineers.

1. Introduction

This paper focuses on a method for presentation of research results (in written and/or oral form) and focuses on the following issues:

(a) Selection of the title;
(b) Structure of the abstract;
(c) Structure of the figures and/or tables and their captions;
(d) Syntax of references;
(e) Structure of the written paper and the related oral presentation using transparencies;
(f) Semantics-based layout of transparencies for an oral presentation.

Intentionally, the entire text to follow has been made relatively short, so more people decide to read it. This paper represents the decades-long research experience of the author, and summarizes the mandatory requirements that he places before his graduate students.

The motivation to publish this paper (which is in use at the University of Belgrade for about half decade now) came after the repeated pattern at international conferences where lots of good research was presented in such a way that research results are obscured by poor pres-
entation. It was not possible to understand quickly, either the essence of the contribution, or the most important research details.

At a recent major set of computer science/engineering conferences, no single presentation was following either the guidelines presented below, or the semantics-based layout of transparencies to be defined below.

2. Selection of the Title

The selection of title should be both didactic and lapidaric. See a dictionary for the exact meaning of these two terms.

In this context, didactic means creating a title which enables an expert to figure out the essence of the basic idea and the main contribution, even without reading the paper; lapidaric means creating a title which induces the reader to think deeply over the “philosophy” of the contribution described in the paper. A relatively good example of a didactic and lapidaric title is:

APPLYING ENTRY AND LAZY RELEASE SELECTIVELY: TEMPORAL VERSUS SPATIAL DATA

This title is didactic since it is immediately obvious that the main idea is to apply the entry consistency model to temporal data and the lazy release consistency model to spatial data, for the performance which is better than applying only one of the two models to all data.

This title is also lapidaric, since one immediately starts thinking about how the selective application of two different consistency models was really implemented. An alternative (bad) title would be:

SOME ISSUES IN MEMORY CONSISTENCY MODELING

People would tend to stay away from a paper with such a title, since that kind of title might be viewed as an introduction into a contents-free paper, unless it comes from a well known expert who has a reputation of knowing what he/she is doing. Consequently, a good idea may not be noticed by the research community, and those who reinvent it at a later time will get the credit instead of the initial inventor.

3. Structure of the Abstract

Wherever possible, the abstract of a research paper should include the following five elements:

(a) Problem statement of the research under consideration;

(b) A short list of existing solutions and what is their drawback, from the point of view of the above defined problem statement;

(c) Essence of the proposed solution, and why it is expected to be better under the same conditions;

(d) What type of analysis was done to show that the proposed solution is really better than any of the existing ones, from both the performance and the complexity points of view (if one is an engineer, then both performance and complexity are equally important);
What are the major numerical highlights of the analysis (if one is an engineer, numbers are the “name of the game”).

If a 50-word abstract is required, then each part above should be about one sentence long; if a 500-word abstract is required, then each part above should be about 10 sentences long, etc. Of course, the language should be simple and concise, with declarative sentence structure, written primarily in the present tense.

4. Selection of the Keywords

Appropriate selection of keywords is important. It helps with the accessibility and retrievability of your paper/thesis, which in turn enables more researchers to become aware of your work, and to reference it in their work. A good number of keywords to include is one to five. One good classification of keywords in computer science and engineering can be found at the WWW presentation of ACM (http://www.acm.org/).

However, the field of computer science and engineering is quickly expanding. Consequently, often the author has to create new keywords. Good guidelines for creation of new keywords can be found at WWW presentations of major journal publishers (e.g., http://www.elsevier.nl/inca/homepage/sac/micpro/instant.htm#keyword).

Some of the major rules for creation of keywords are: (a) each keyword must describe a single concept, (b) avoid very general terms since they bear little information about your specific work, (c) try to use only adjectives and nouns in the singular form, (d) avoid the use of abbreviations, as much as possible, unless they are well established, and (e) avoid words like “and,” “or,” “of,” etc., since they may interfere with the database search software.

5. Structure of the Figures and/or Tables and the Related Captions

Figures and tables should include only language-independent mnemonics (derived from English language), which is especially important for non-English-speaking researchers, and for those writing for many languages, so it is easier to switch back and forth between languages.

All details must be clearly visible, even after the same figure is ported to a transparency for an oral presentation.

Captions deserve a special attention, which is neglected in a typical written presentation. The main issue is that reading only the figure captions of the paper can substitute the first rough reading of the entire paper. This goal is achieved more successfully if the caption includes the following five elements:

(a) Title with the main highlight, i.e. the main issue to be demonstrated by the corresponding figure/table;
(b) Legend, to explain all language-independent mnemonics inside the figure/table;
(c) Description, of one or more phenomena which deserve attention (e.g., curves $A$ and $B$ cross each other at $X = 16$);
(d) Explanation, of the essential reason for such a behavior (e.g., the curves cross each other because for higher values of $X$, the following happens ...);

(e) Implication, or what is to be kept in mind when designing/engineering a system to exploit the above noticed phenomenon (e.g., increasing the size of register file helps until the number of registers reaches a critical value; after that ...).

A book which insists on this type of reasoning is [Flynn95]; however, the approach has not been formalized, and this type of reasoning can not be found in figure/table captions. Writing a good caption of this type is extremely difficult for the one who writes the paper (and graduate students often show resistance to such an approach), but extremely useful for the one who reads the paper (and readers/reviewers often show appreciation for such an approach).

Also, this type of caption may become relatively long, and one might think that the limited paper space is not used rationally; however, the captions should include only the facts which are “local” to the figure/table, and these facts should never be repeated again in the main body of the paper. The main body of the paper should include only the “global” facts (e.g., comparing the findings from different figures, and similar).

A similar approach can be found in the famous books of Hennessy and Patterson (alphabetical order), except that their captions do not always have all five elements, and if they do include all five elements, these elements are not formally separated, which is a requirement of the methodology presented here.

All figure and figure captions should be completed before the actual writing of the paper starts.

6. Syntax of References

This is another item to be completed before the writing of the paper starts. As far as the syntax of references, it is most natural that one follows the syntax used by the most prestigious scientific journal in the field (e.g., IEEE Transactions on ...).

If an alternative approach seems to be better, this methodology suggests that one waits until the major journal accepts it.

As far as the method of pointing to a reference, the mnemonical approach with the entire name of the first author and the year is preferred (so the reader knows immediately what research group the paper comes from). Often, the name of the last author conveys that information more clearly, but it is not practical to use it, when pointing to a reference. Of course, if so required, the above method can be easily converted into the numeric form, mandatory in some journals.

An important reason for doing references before the actual writing starts is that one makes sure that no important reference is omitted; a task more difficult to do after the entire paper is completed.

7. Structure of the Written Paper and the Corresponding Oral Presentation

In the case of a research paper, whenever possible, one should first develop the skeleton of the paper/presentation, to include the following first level titles:
a) **Introduction**, to include the basic facts needed to tune the reader to the paper and/or presentation;

b) **Problem statement**, to define precisely the problem being attacked by the research under consideration, and why is that problem important;

c) **Existing solutions and their criticism**, to survey briefly the major existing solutions form the open literature and to underline their deficiencies from the point of view of interest for this research, which is defined in the above mentioned problem statement section;

d) **Proposed solution and why it is expected to be better**, to give the essence of the proposed solution (i.e., the essence of the idea which is to be introduced), followed by a logical and/or philosophical discussion about the expected benefits stemming from the idea;

e) **Conditions and assumptions of the research to follow**, to summarize the environment of interest. The term *conditions* refers to the specifiers of the real environment, and the term *assumptions* refers to the simplifications which simplify the analysis without any negative impacts on the validity and representativeness of the final results. It is useful for the reader if conditions and assumptions are itemized (e.g., application-, system-software-, architecture-, organization-, design-, and technology-related);

f) **Analytical analysis**, to show one or more of the following:

f1) proof of validity of the major idea of the paper/presentation;

f2) calculation of initial values for simulation analysis to follow;

f3) rough estimation of the performance;

f4) rough estimation of the complexity;

f5) something else which is relevant;

Analytical analysis will not give the final answers; however, it will help understanding the concept (it will be helpful both to the researcher and the reader);

g) **Simulational analysis**, to show performance (this should be the major and the longest part of the paper);

h) **Implementational analysis**, to show complexity

(for some types of research, this one could be the major and the longest part of the paper);

i) **Conclusion**, with the following three major elements:

i1) revisiting the major contribution from the performance/complexity point of view;

i2) stating who will benefit from the presented results;

i3) what are the newly open problems and research avenues.

One should keep in mind that some people read only the abstract and the conclusion;

j) **References**, as described above.

After the skeleton on the first level of titles is defined, one should develop the skeleton on the paragraph level; this means defining all subtitles on lower levels and the contents of all paragraphs under each lowest-level sub-title. Finally, the last thing to do is to specify the first sentence of each paragraph, which is the major one; other sentences of each paragraph are just to explain and/or justify the statement conveyed by the first sentence.

It is not before now that the writing can start, and it will be easy to do it; also, this approach enables that, after the complete skeleton is developed by a senior person (e.g., a major profes-
sor), the writing can be done by a junior person (e.g., a graduate student); any errors in writing will be localized at the paragraph level, and, as such, easy to fix.

The above applies to research papers. An important prerequisite for a good research paper is that a good survey paper is prepared first, to demonstrate that major solutions for the problem of interest are known.

In the case of a survey paper, the major requirement is to have a concepts part (to define the major issues), and the systems part (to define various algorithms and/or implementations, etc.). The concepts part should be preceded by a classification of concepts. The systems part should be preceded by a classification of systems. Each system in the systems part should be described/explained using the same template (e.g., origin, environment, essence, advantages, drawbacks, relevant details, performance consideration, complexity consideration, conclusion, trends, etc.). The choice of elements for the template is flexible. What is not flexible is that the same elements must be used in each template.

No matter what is the goal of the paper, sentences should be short, using simple words, and avoiding passive voice. This makes the text easier to read and comprehend. Consequently, more people will decide to read the entire paper rather than only the abstract and the conclusion.

8. Semantics-Based Layout of Transparencies

Major rules for doing the transparencies can be found in numerous books. Consequently, the stress here is on an issue which is extremely important, yet not mentioned in any of the books known to this author—the rule about the semantics-based layout of transparencies. This rule reads as follows.

*If a semantic entity must be spread over several lines, the breakdown of lines should be done in a semantic way.* In other words, if a “bullet” is to be spread over more than one line (often, three is the maximum which makes a good choice), each line should represent a separate thought.

As an illustration, two examples are shown next, one without and one with semantic splitting.

AN EXAMPLE WITHOUT SEMANTIC SPLITTING:

**TOPIC TITLE**
- Fixed/variable allocation scenarios based on the home property (page manager): DSM + DSIO
- Writes get satisfied on distance or locally, depending on what brings better performance
- Good if reads and writes are interleaved with similar probabilities of occurrence

AN EXAMPLE WITH SEMANTIC SPLITTING:

**TOPIC TITLE**
- Fixed/variable allocation scenarios, based on home property (page manager): DSM + DSIO
• Writes get satisfied on distance or locally, depending on what brings better performance
• Good if reads and writes are interleaved, with similar probabilities of occurrence

In other words, do not let the word-processor split the lines for you. Instead, do it by yourself, the right way! Semantic splitting is extremely useful for the audience, and its fast comprehension of the material. An experiment was performed by the author to prove that fact. In this experiment, the same subject was taught to two different groups of students, using two sets of transparencies, one with and one without semantic splitting of lines. A test would be given after the subject is completed. The experiment was repeated enough times, and the test results were considerably different, in favor of the transparencies based on semantic splitting.

As a consequence of this experiment, the author of this paper insisted that transparencies for his university courses and pre-conference tutorials are based on semantic splitting [Ekmečić97, Protic96a, Tartalja96, Tomasevic93].

Sometimes, semantic splitting seems impossible to do; however, in each such case, it turns out that an alternative way of expressing the thoughts is both easy to split and sounds much better.

9. Conclusion

This paper sets a standard for organization of research presentations, and defines the semantics-based layout of presentation transparencies. So far, almost without exception, others would start using the views expressed here (especially the semantics-based splitting for transparencies), as soon as they learn about them, which was a great source of pleasure and satisfaction for the author.

10. A Note

An earlier but wider version of this text can be found in [Milutinovic95]. For lower level details, the interested reader is welcome to contact the author directly.

11. Acknowledgments

The author is thankful to professors Mike Flynn of Stanford and Yale Patt of Michigan for their response to some of the ideas presented here; also, to professors Hennessy, Gupta, and their graduate students for numerous comments during the author’s last seminar at Stanford University. Also, to professor Jean-Loup Baer for his suggestion to publish a paper of this type with all the experiences incorporated, and to professor Nitin Vaidya for his efforts to help about the paper quality.

At last, but not least, the author is thankful to the graduate students of the University of Belgrade for their continuous efforts to educate their major professor, so he can keep up with the newest trends in the field (e.g., Jovanka Ciric, Goran Davidovic, Ilija Ekmečić, Aleksandar Janicijevic, Milan Jovanovic, Aleksandar Milenkovic, Zvezdan Petkovic, Milena Petrovic, Jelica Protic, and Dejan Raskovic).
12. References

[Ekmecic97] Ekmecic, I., Tartalja, I., Milutinovic, V.,
“Tutorial on Heterogeneous Processing: Concepts and Systems,”

[Flynn95] Flynn, M.J.,
“Computer Architecture,”
Jones and Bartlett, Boston, Massachusetts, 1995.

[Hennessy96] Hennessy, J.L., Patterson, D.A.,
“Computer Architecture: A Quantitative Approach,”
Morgan Kaufmann, San Francisco, California, 1996.

[Milutinovic95] Milutinovic, V.,
“A Research Methodology in the Field of Computer Engineering for VLSI,”
Proceedings of the IEEE International Conference on Microelectronics,
Nis, Serbia, Yugoslavia, September 1995.

[Milutinovic96] Milutinovic, V.,
“Surviving the Design of a 200 MHz RISC Microprocessor: Lessons Learned,”

[Patterson94] Patterson, D. A., Hennessy, J.L.,
“Computer Organization and Design,”

[Tartalja96] Tartalja, I., Milutinovic, V.,
“Tutorial on Cache Coherency Maintenance in Shared Memory Multiprocessors: Software Solutions,”
IEEE CS PRESS, Los Alamitos, California, 1996.

[Tomasevic93] Tomasevic, M., Milutinovic, V.,
“Tutorial on Cache Coherency Maintenance in Shared Memory Multiprocessors: Hardware Solutions,”

13. Epilogue

This section includes a list with some of the author’s journal papers which either helped create the research and presentation methodology which is the subject of this paper, or are based on the research and presentation methodology presented in this paper. The enclosed list includes only the papers published after 1.1.1990. and only from the IEEE periodicals.

1. Milutinovic, V.,
“Mapping of Neural Networks onto the Honeycomb Architecture,”

2. Milutinovic, V.,
“Tutorial on Microprogramming and Firmware Engineering,”
3. Perunicic, B., Lakhani, S., Milutinovic, V.,
   “Stochastic Modeling and Analysis of Propagation Delays in GaAs Adders,”

4. Milutinovic, V., Fura, D., Helbig, W.,
   “Pipeline Design Trade-offs in 32-bit Gallium Arsenide Microprocessor,”

5. Hoevel, L., Milutinovic, V.,
   “Terminology Risks with the RISC Concept in the Risky RISC Arena,”

6. Tomasevic, M., Milutinovic, V.,
   “Tutorial on the Cache Coherency Problem in Shared-Memory Multiprocessors:
   Hardware Solutions,”

7. Tomasevic, M., Milutinovic, V.,
   “A Survey of Hardware Solutions for Maintenance of Cache Consistency
   in Shared Memory Multiprocessor Systems,”
   IEEE MICRO (Part #1), October 1994.

8. Tomasevic, M., Milutinovic, V.,
   “A Survey of Hardware Solutions for Maintenance of Cache Consistency
   in Shared Memory Multiprocessor Systems,”
   IEEE MICRO (Part #2), December 1994.

9. Milutinovic, V., Petkovic, Z.,
   “Processor Design Using Silicon Compilation: Ten Lessons Learned
   from a RISC Design,”

10. Savic, S., Tomasevic, M., Milutinovic, V.,
    “Improved RMS for the PC Environment,”
    (A follow up paper will be published in an IEEE journal).

11. Ekmecic, I., Tartalja, I., Milutinovic, V.,
    “A Taxonomy of Heterogeneous Computing,”

12. Tartalja, I., Milutinovic, V.,
    “Tutorial on the Cache Coherency Problem in Shared-Memory Multiprocessors:
    Software Solutions,”

13. Tomasevic, M., Milutinovic, V.,
    “The Word Invalidate Protocol,”
    Microprocessor Systems, March 1996
14. Grujic, A., Tomasevic, M., Milutinovic, V.,
   “A Simulation Study of Hardware DSM Approaches,”
   IEEE Parallel and Distributed Technology, Spring 1996.

15. Milutinovic, D., Milutinovic, V.,
   “Mapping of Interconnection Networks for Parallel Processing
   onto the Sea-of-Gates VLSI,”
   IEEE Computer, Vol. 29, No. 4, April 1996.

16. Protic, J., Tomasevic, M., Milutinovic, V.,
   “A Survey of Distributed Shared Memory: Concepts and Systems,”
   IEEE Parallel and Distributed Technology, Summer 1996.

17. Tartalja, I., Milutinovic, V.,
   “A Survey of Software Solutions for Cache Consistency Maintenance
   in Shared Memory Multiprocessors,”

18. Protic, J., Tomasevic, M., Milutinovic, V.,
   “Tutorial on DSM: Concepts and Systems,”

19. Ekmecic, I., Tartalja, I., Milutinovic, V.,
   “A Survey of Heterogeneous Computing: Concepts and Systems,”
   Proceedings of the IEEE, August 1996.

20. Milicev, D., Petkovic, Z., Raskovic, D., Jelic, D., Jelisavic, M., Stevanovic, D.,
   Milenkovic, A., Milutinovic, V.,
   “Modeling of Modern 32-bit and 64-bit Microprocessors,”

21. Milutinovic, V.,
   “The Best Method for Presentation of Research Results in Computer Engineering,”
   IEEE TCCA Newsletter, September 1996.

22. Milutinovic, V.,
   “Some Solutions for Critical Problems of Distributed Shared Memory Systems:
   New Ideas to Analyze,”
   IEEE TCCA Newsletter, September 1996.

23. Milutinovic V., Tomasevic, M., Markovic, B., Tremblay, M.,
   “The Split Temporal/Spatial Cache Memory for Next Generation SuperMicroprocessors:
   Initial Performance Analysis,”
   (To be published). Conference version available from the Proceedings
   of the IEEE SCIzzL-5, Santa Clara, California, March 1996.

24. Milutinovic V., Tomasevic, M., Markovic, B., Tremblay, M.,
   “The Split Temporal/Spatial Cache Memory for Next Generation SuperMicroprocessors:
   Initial Complexity Analysis,”
   (To be published). Conference version available from the Proceedings
   of the IEEE SCIzzL-6, Santa Clara, California, September 1996.
A Good Method
to Prepare and Use Transparencies
for Research Presentations

This section††† summarizes some of the author’s experiences in preparation and usage of overhead transparencies for research presentations. Since one picture is worth a thousand words, the amount of the text is minimized (so one decides to read it), and the stress is on actual transparencies (so one can see the issues). Figures of this paper are in the form of ready to use transparencies, for teachers of methodology classes.

1. Introduction

The first figure defines the details of a well-structured title transparency. It contains three important elements: title, name(s) and affiliation(s) of the author(s), and the symbol of the research group (important in some environments). At first, such a title slide may look too dense. However, as indicated later, the same slide can be reloaded at the very end of presentation, so interested people can copy information they need.

The second figure defines the issues of a well structured introductory transparency. It contains three important elements: tuning the audience to the subject (for example, basic issues about the research field), showing the entire forest before individual trees are approached (for example, the table of contents), and making the introductory comments which help to avoid misunderstandings (important in some cases). An introductory anecdote is more than welcome (of course, it has to be well balanced and compatible with the subject; otherwise, it brings more harm than help).

2. Preparing the Transparencies

The third figure defines the ten essential elements of a well structured and prepared research presentation. Details on these elements and more could be found in [1]. Good work which is not well presented is not necessarily a good work! Note that the rule of thumb is to

††† A version of this text can be found at http://www.computer.org/tab/tcca/tcca.htm.
use $7 \pm 2$ lines per page, which means that this slide is a little too dense; however, the chosen density is justified by the fact that itemization includes ten elements.

The fourth figure sheds some light on the fact that each research problem is essentially a logical problem; consequently, the formal analysis should contain the seven elements of a generalized hypothesis testing. There is a strong analogy between the terms in the fourth figure and the typical engineering terms (variables, primitives, procedures, conditions and assumptions, subrules, rules, and implications).

The fifth and the sixth figure are related to semantic splitting of bulleted text which spans two or more lines, as discussed in [1]. The fifth figure gives a positive example (with semantic splitting). The sixth figure gives a negative example (without semantic splitting). Once a person gets used to semantic splitting, he/she can’t live without it any more!

The seventh and the eighth figure are about font size and line count, as discussed in numerous education related references [2], together with many other “tricks of the trade,” like single font, figures with all details visible from the most remote corner of the lecture hall, etc. The seventh figure gives a positive example (large enough font and a single-digit line count). The eighth figure gives a negative example and exaggerates in order to underline the essence. If you decide to use colors, remember, better no colors than badly composed colors.

3. Using the Transparencies

The ninth figure gives some advice related to the actual presentation. Each keyword carries lots of meaning, and leaves lots of room for the speaker to improvise when explaining its essence. Again, note that very dense transparencies are difficult to follow, unless the speaker spends enough time on each item.

Make an effort to verify the setting of the presentation desk and the screen, before the session starts. There should be enough space to lay out the transparencies and to use them without crossing the overhead beam. Also, there should be a space for the speaker to position himself/herself so that he/she can see both the audience and the screen without turning around and without blocking the view of the audience. If all these conditions are not satisfied (which is frequently the case if conferences are held in hotels, where the presentation desk and screen are typically set by people without Ph.D. in computer architecture), do not hesitate to do rearrangements (after asking for permission, if necessary).

Make sure that you look good during the presentation (and compatible with your general image). Do not forget to get combed, to check buttons, etc.

Let the introductory transparency be there while you are being introduced by the chairman, so the screen is not blank and people can get both oral and written information.

Check the screen after each new transparency is placed, in order to see if it is in line and fully visible.

Point with your pointer to the wall and not to the overhead projector; while pointing, make sure that you do not impair the clear view of the screen.

Avoid the SOS words and sounds while talking; they irritate the audience, and may hurt the presentation.

Watch your time; in the research environment overtime is considered one of the worst sins, and will hurt the speaker not careful with time.
Remember that the discussion after the presentation is when the speaker demonstrates who he really is (this is where many good presentations get blown away).

Do not promise more material on your work before you are explicitly asked for. If explicitly asked, suggest that the request be forwarded to you by e-mail.

Let the introductory transparency be there while you are waiting for post-presentation questions, so the probability is increased that the audience leaves with a lasting memory of your lecture.

4. Conclusion

The last transparency has a special weight and should include the three points indicated in the tenth figure. Finally, a concluding anecdote is welcome; its major purpose is to reiterate the major message of the paper in a way which creates smiles and is easy to remember.

For more details on the experiences presented here, and the related experiences of the author working with his younger colleagues, the reader is referred to [3] and [4]. Also, have a close look at [5], [6], [7], and [8].

5. Acknowledgment

The author is thankful to all the poor presentations that he enjoyed at conferences, and all the good suggestions that he received from students/colleagues.

6. References

[1] Milutinović, V.,
“The Best Method for Presentation of Research Results,”

[2] Milutinović, V.,
“A Good Method to Prepare and Use Transparencies in Research Presentations,”

[3] Milutinović, V.,
Surviving the Design of a 200 MHz RISC Microprocessor: Lessons Learned,

[4] Milutinović, V.,
Surviving the Design of Microprocessor and Multimicroprocessor Systems: Lessons Learned,
[5] Patterson, D.,
“How to Have a Bad Career in Research/Academia,”

[6] Parberry, I.,
“How to Present a Paper in Theoretical Computer Science:
A Speaker’s Guide to Students,”

[7] Chapman, D., Editor,
“How to do Research At the MIT AI Lab,”

[8] Kschischang, F.,
“Giving a Talk: Guidelines for the Preparation and Presentation of Technical Seminars,”
A Good Method
to Prepare and Use Transparencies
for Research Presentations

Veljko Milutinović

Department of Computer Engineering,
School of Electrical Engineering,
University of Belgrade,
POB 816, 11000 Belgrade, Serbia, Yugoslavia
Phone: +381-11-762-214, Fax: +381-11-762-215
E-mail: vm@ubbg.etf.bg.ac.yu
WWW: http://ubbg.etf.bg.ac.yu/~vm/

Σ

Figure 1: The three elements of a well-structured title page.

Introduction

• Tuning the audience to the subject
• Presenting the forest to the audience
• Preventing misunderstandings of any kind

😊

Figure 2: The three elements of a well-structured introduction (with an introductory anecdote).
The Ten Compile-Time Commandments

- Introduction (to tune the audience)
- Problem statement (and why it is important)
- Existing solutions (and their criticism)
- Proposed solution (and its essence)
- Conditions and assumptions (of the analysis to follow)
- Details of the solutions to be compared \((1 + k)\)
- Mathematical analysis (or analytical modeling)
- Simulation analysis (to show performance)
- Implementation analysis (to show complexity)
- Conclusion (from the performance/complexity viewpoint)

Figure 3: The ten elements of a well-structured research presentation.

The Magnificent Seven

- Elements of the set
- Operations
- Functions
- Axioms
- Lemmas
- Theorem(s)
- Corollaries

Figure 4: The seven elements of a well-structured formal analysis.
Fixed/variable allocation scenarios based on the home property (page manager): DSM + DSIO system approaches

- Writes get satisfied on distance or locally, depending on what brings better performance and smaller complexity
- Good if reads and writes are interleaved with similar probabilities of occurrence

Figure 5: Semantic line breaks: Good.

Figure 6: Word processor forced line breaks: Bad.
Title—36 pt
• Simple superscalar (four)—24 pt
• Branch prediction issues (2-bit history table)
• On-chip caches (primary and secondary)

Figure 7: Font size and line count: Good.

Title
• Simple superscalar going after high clock rate; four instructions fetching from an 8KB I cache, and placed in one of two I buffers (four instructions in each one).
• Issue goes from the buffers (in-order, no instructions bypassing at all); one must be empty before the other one can be used (simpler design, but slower issue).
• Branches predicted using a 2-bit history table associated within the I cache controller; I issue stalled if 2nd branch encountered before the 1st branch resolved.
• After fetch and decode, instructions are arranged according to the functional unit they will use; issue follows after the operands are ready (from registers or via bypassing).
• Four FUs: int with shifter, int with branchevalor, FP adder, FP multiplier; integer instructions update registers in-order; FP instructions update files out-of-order; not all FP exceptions are precise.
• On-chip primary caches: 8KB each (I + D, dm for 1cp cache access); a 6-entry MAF with MissAddresses/TargetRegs for up to 21 missed loads (merge).
• On-chip secondary cache: 96KB (shared, 3w/sa); in most of the current microprocessors this cache memory is off the chip.

Figure 8: Font size and line count: Bad.
The Ten Run-Time Commandments

- *Do not hesitate* to redo the presentation desk prior to session start, in order to optimize your efficiency and comfort.
- *Make sure* that you look good during the presentation.
- *Let* the introductory transparency be there while you are being introduced.
- *Check* the screen after each new transparency.
- *Point* at the wall, not at the transparency.
- *Avoid* the SOS words.
- *Watch* your time!!
- *Remember*, discussion is where you show who you really are.
- *Do not promise*; just suggest the audience to ask by e-mail.
- *Reload* the introductory transparency immediately after you finish.

Figure 9: The ten elements of a well-structured presentation.

Conclusion

- Performance versus complexity
- Who will benefit
- Newly open problems…

Figure 10: The three elements of a well-structured conclusion (with a final anecdote).
The list to follow includes only the references used in this book. For information on the ongoing research in the field, the reader is referred to a WWW presentation which includes a number of interesting and useful pieces of information related to the general field of computer architecture (http://www.cs.wisc.edu/~arch/www/#Participation).


[Ekmecic95] Ekmecic, I., Tartalja, I., Milutinovic, V.,
“A Taxonomy of Heterogeneous Computing.”

[Ekmecic96] Ekmecic, I., Tartalja, I., Milutinovic, V.,
“A Survey of Heterogeneous Computing: Concepts and Systems,”
Proceedings of the IEEE, August 1996.

[Ekmecic97] Ekmčić, I.,
“The LOCO++ Approach to Task Allocation in Heterogeneous Computer Systems,”

[Espasa97] Espasa, R., Valero, M.,
“Multithreaded Vector Architectures,”

[Evers96] Evers, M., Chang, P.-Y., Patt, Y.,
“Using Hybrid Branch Predictors to Improve Branch Prediction Accuracy in the Presence of Context Switches,”

[Fleisch89] Fleisch, B., Popek, G.,
“Mirage: A Coherent Distributed Shared Memory Design,”

[Flynn95] Flynn, M. J.,
Computer Architecture: Pipelined and Parallel Processor Design,
Jones and Bartlett Publishers, Boston, Massachusetts, 1995.

[Forman94] Forman, G. H., Zahorjan, J.
“The Challenges of Mobile Computing,”

[Fortes86] Fortes, J., Milutinović, V., Dock, R., Helbig, W., Moyers, W.,
“A High-Level Systolic Architecture for GaAs,”

[Frank93] Frank, S., Burkhardt III, Rothnie, J.,
“The KSR1: Bridging the Gap Between Shared Memory and MPPs,”

[Fromm97] Fromm, R., Perissakis, S., Cardwell, N., Kozyrakis, C., McEachern, B., Patterson, D.,
Anderson, T., Yelick, K.,
“The Energy Efficiency of IRAM Architectures,”

[Geppert97] Geppert, L.,

[Gillett96] Gillett, R. B.,
“Memory Network for PCI,”


Jovanović, M., Milutinović, V., “Bypassing and Deletion for Better Performance,” *Encore*, Fort Lauderdale, Florida, USA, **1996**.


[Milutinovic87a] Milutinovic, V., Lopez-Benitez, N., Hwang, K.,
“A GaAs-Based Microprocessor Architecture for Real-Time Applications.”

[Milutinovic87b] Milutinovic, V.,
“A Simulation Study of the Vertical-Migration Microprocessor Architecture,”

[Milutinovic88a] Milutinovic, V.,
“A Comparison of Suboptimal Detection Algorithms Applied to the Additive Mix of Orthogonal Sinusoidal Signals,”

[Milutinovic88b] Milutinovic, V., Crnkovic, J., Houstis, C.,
“A Simulation Study of Two Distributed Task Allocation Procedures,”

[Milutinovic92] Milutinovic, V.,
“Avenues to Explore in PC-Oriented DSM Based on RM,”
*ENCORE Internal Report (Solicited Expert Opinion)*,
ENCORE, Fort Lauderdale, Florida, USA, **December 1992**.

[Milutinovic95a] Milutinovic, V.,
“A New Cache Architecture Concept: The Split Temporal/Spatial Cache Memory,”
UBG-ETF-TR-95-035, Belgrade, Serbia, Yugoslavia, **January 1995**.

[Milutinovic95b] Milutinovic, V., Petkovic, Z.,
“Ten Lessons Learned from a RISC Design,”
*Computer*, **March 1995**, p. 120.

[Milutinovic95c] Milutinović, V.,
“New Ideas for SMP/DSM,”
*UBTR*, Belgrade, Serbia, Yugoslavia, **1995**.

[Milutinovic95d] Milutinović, V.,
http://ubbg.etf.bg.ac.yu/~vm/ieee90.html **1995**.

[Milutinovic96a] Milutinovic, V., Markovic, B., Tomasevic, M., Tremblay, M.,
“The Split Temporal/Spatial Cache Memory: Initial Performance Analysis,”

[Milutinovic96b] Milutinovic, V., Markovic, B., Tomasevic, M., Tremblay, M.,
“The Split Temporal/Spatial Cache Memory: Initial Complexity Analysis,”

[Milutinovic96c] Milutinovic, V.,
“Some Solutions for Critical Problems in Distributed Shared Memory,”
*IEEE TCCA Newsletter*, **September 1996**.
“The Best Method for Presentation of Research Results,”
IEEE TCCA Newsletter, September 1996.

“MIPS R10000 Microprocessor User’s Manual, Version 2.0”
MIPS Technologies, Mountain View, California, USA, 1996.

“Mirror Memory System,”
Internal Report, Modcomp, Inc., Fort Lauderdale, Florida, USA,
December 1983.

“Dynamic Speculation and Synchronization of Data Dependencies,”

“Exploiting Instruction Level Parallelism in Processors by Caching Scheduled Groups,”

“S3.mp: A Multiprocessor in Matchbox,”

“Complexity-Effective Superscalar Processors,”

“Tuning the Pentium Pro Microarchitecture,”

“The I/O Subsystem—A Candidate for Improvement,”

M. Sc. Thesis, University of Belgrade,

Peterson, L. L., Davie, B. S.,
Computer Networks,
Morgan Kaufmann, San Francisco, California, 1996.

“On Deadlocks in Interconnection Networks,”

“RST: Cache Memory Design for a Titlay Coupled Multiprocessor System,”

“Reducing Coherence-Related Overhead in Multiprocessor Systems,”
Proceedings of the IEEE/Euromicro Workshop
[Prete97] Prete, C. A., Prina, G., Giorgi, R., Ricciardi, L.,
“Some Considerations About Passive Sharing in Shared-Memory Multiprocessors,”

[Protic85] Protic, J.,
“System LOLA-85.”
Lola Technical Notes (in Serbian), Belgrade, Serbia, Yugoslavia, *December 1985.*
email: jeca@etf.bg.ac.yu.

[Protic96a] Protic, J., Tomasevic, M., Milutinovic, V.,
“Distributed Shared Memory: Concepts and Systems,”
pp. 63–79.

[Protic96b] Protić, J., Milutinović, V.,
“Combining LRC and EC: Spatial versus Temporal Data,”
*Encore*, Fort Lauderdale, Florida, USA, 1996 (jeca@etf.bg.ac.yu).

[Protic97] Protic, J., Tomasevic, M., Milutinovic, V.,
“Tutorial on Distributed Shared Memory (Lecture Transparencies),”

[Protic98] Protic, J.,
“A New Hybrid Adaptive Memory Consistency Model,”

[Prvulovic97] Prvulovic, M.,
“Microarchitecture Features of Modern RISC Microprocessors—An Overview,”

[Ramachandran91] Ramachandran, U., Khalidi, M., Y., A.,
“An Implementation of Distributed Shared Memory,”
pp. 443-464.

[Raskovic95] Raskovic, D., Jovanov, E., Janicijevic, A., Milutinovic, V.,
“An Implementation of Hash Based ATM Router Chip,”

[Raskovic97] Raskovic, D.,
“Distributed Shared I/O,”

[Reinhardt94] Reinhardt, S., Larus, J., Wood, D.,
“Tempest and Typhoon: User-Level Shared Memory,”

[Reinhardt96] Reinhardt, S. K., Pfle, R. W., Wood, D. A.,
“Decoupled Hardware Support for DSM,”

[Rexford96] Rexford, J., Hall, J., Shin, K. G.,
“A Router Architecture for Real-Time Point-to-Point Networks,”
[Sanchez97] Sanchez, F. J., Gonzalez, A., Valero, M.,
“Software Management of Selective and Dual Data Caches,”

[Saulsbury96] Saulsbury, A., Pong, F., Nowatzyk, A.,
“Missing the Memory Wall: The Case for Processor/Memory Integration,”

[Savic95] Savic, S., Tomasevic, M., Milutinovic, V., Gupta, A., Natale, M.,
Gertner, I.,
“Improved RMS for the PC Environment,”
Microprocessors and Microsystems,

“Fine-grain Access Control for Distributed Shared Memory,”

[Sechrest96] Sechrest, S., Lee, C. C., Mudge, T.,
“Correlation and Aliasing in Dynamic Branch Predictors,”

[Seznec96] Seznec, A.,
“Don’t use the page number, but a pointer to it,”

[Sheaffer96] Sheaffer, G.,
“Trends in Microprocessing,”
Keynote Address, YU-INFO-96, Brezovica, Serbia, Yugoslavia, April 1996.

[Simha96] “R4400 Microprocessor product information”
MIPS Technologies, Mountain View, California, USA, 1996.

[Simoni90] Simoni, R.,
“Implementing a Directory-Based Cache Coherence Protocol,”

[Simoni91] Simoni, R., Horowitz, M.,
“Dynamic Pointer Allocation for Scalable Cache Coherence Directories,”

[Simoni92] Simoni, R.,

[Smith95] Smith, J. E., Sohi, G.,
“The Microarchitecture of Superscalar Processors,”
[Sprangle97] Sprangle, E., Chappell, R.S., Alsup, M., Patt, Y.,
“The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference,”

[Stenstrom88] Stenstrom, P.,
“Reducing Contention in Shared-Memory Multiprocessors,”

[Stiliadis97] Stiliadis, D., Varma, A.,
“Selective Victim Caching: A Method to Improve the Performance of Direct-Mapped Caches,”

[Stojanović95] Stojanović, M.,
“Advanced RISC Microprocessors,”
Internal Report, Department of Computer Engineering, School of Electrical Engineering, University of Belgrade, Belgrade, Serbia, Yugoslavia, December 1995.

[Sun95] “SuperSPARC Data Sheet: Highly Integrated 32-Bit RISC Microprocessor,”
Sun Microelectronics, Mountain View, California, USA, 1995.

[Sun96] “UltraSPARC-I High Performance, 167 & 200 MHz, 64-bit RISC Microprocessor Data Sheet,”
Sun Microelectronics, Mountain View, California, USA, 1996.

[Sun97] “UltraSPARC-II High Performance, 250 MHz, 64-bit RISC Processor Data Sheet,”
Sun Microelectronics, Mountain View, California, USA, 1997.

[Tanenbaum90] Tanenbaum, A. S.,
Structured Computer Organization,

[Tartalja97] Tartalja, I.,
“The Balkan Schemes for Software Based Maintenance of Cache Consistency is Shared Memory Multiprocessors,”

[Teodosiu97] Teodosiu, D., Baxter, J., Govil, K., Chapin, J., Rosenblum, M., Horowitz, M.,
“Hardware Fault Containment in Scalable Shared-Memory Multiprocessors,”

[Thornton64] Thornton, J. E.,
“Parallel Operation on the Control Data 6600.”

[Tomasevic92a] Tomasevic, M., Milutinovic, V.,
“A Simulation Study of Snoopy Cache Coherence Protocols,”

[Tomasevic92b] Tomasevic, M.,
“A New Snoopy Cache Coherence Protocol,”
ABOUT THE AUTHOR
This part has been prepared at the request of the publisher, and includes the following elements: (a) A list of industrial cooperations, (b) A list of publications from IEEE periodicals, and (c) A list of citations in papers and books on computer architecture. For more information, the interested reader is referred to the author's WWW presentation (http://ubbg.etf.bg.ac.yu/~vm/).
Selected Industrial Cooperation with US Companies (since 1990)

Note:
The results/publications to follow
are a direct or an indirect consequence of the industrial cooperation (R&D) listed here.
In all these cases,
generated ideas resulted in new products or improvements of existing products.

PURDUE University Research Foundation, West Lafayette, Indiana:

HAWAII University Research Foundation, Honolulu, Hawaii:

NCR Headquarters, Dayton, Ohio (and NCR Germany):
4 R&D topics in shared memory multiprocessing (1990+1991)
4 R&D topics in acceleration chips for multimedia PC (1991)

ENCORE Computer Systems, Fort Lauderdale, Florida (and ENCORE Massachusetts):
4 R&D topics in distributed shared memory for PC environment (1992+1993+1994)
4 R&D topics in reflective memory multiprocessing (1996)

TD Technology, Cleveland, Ohio (and MARUBENI/UNISYS + NIHON/MITSUBISHI Japan):
1 R&D topic in modeling for HLL simulation (1992)

AT&T Headquarters, Murray Hill, New Jersey:
1 R&D topic in computer architecture (1994)

QSI in Santa Clara, California (and NEC Japan):
1 R&D topic in stochastic routing for ATM (1995)

ET Communications, San Francisco, California:
1 R&D topic in logic synthesis for silicon compilation (1996)

SUN Microsystems, Palo Alto, California:
1 R&D topic in cache memory (1996)

INTEL Corporation, Santa Clara, California:
1 R&D topic in cache memory (1996)
# Selected Publications in IEEE Periodicals (since 1990)

**Note:**
Papers from non-IEEE journals have not been listed here; listed papers span the areas from advanced processor design and data communications/networking to cache consistency and distributed shared memory.


25. Milutinovic, V.,
   *Surviving the Design of Microprocessor and Multimicroprocessor Systems: Lessons Learned*,
   IEEE Computer Society Press, Los Alamitos, California, USA, 1998 (accepted).

26. V. Milutinovic, B. Markovic, M. Tomasevic, M. Tremblay,
   “The Split Temporal/Spatial Cache Memory,”
   *IEEE Transactions on Computers*, 1997 (conditionally accepted).

27. Milutinovic, V.,
   “A Research Methodology in the Field of Computer Engineering for VLSI,”
   *IEEE Transactions on Education*, 1997 (conditionally accepted).
   Conference version available from the Proceedings of the IEEE MIEL-95, Nis, Serbia, Yugoslavia, pp. 811–816.

28. Ekmecic, I., Tartalja, I., Milutinovic, V.,
   *Tutorial on Heterogeneous Processing: Concepts and Systems*,
   (conditionally accepted).

29. Davidovic, G., Ciric, J., Ristic-Djurovic, J., Milutinovic, V., Flynn, M. J.,
   “A Comparison of Adders Based on Wave Pipelining,”

30. Vuletic, M., Aleksic, M., Ristic-Djurovic, J., Milutinovic, V., Flynn, M. J.,
   “Per Window Switching of Window Characteristics: Wave Pipelining vs. Classical Design,”

31. Milutinović, V.,
   “Issues in the Theory and Practice of Cache Memory Research: Instead of the Guest Editor’s Introduction,”

32. Milutinović, V.,
   “A Good Method to Prepare and Use Transparencies in Research Presentations,”
General Citations

SCI—over 50 (excluding self-citations);
BOOKS—over 100 (including textbooks, monographs, as well as M.Sc. and Ph.D. theses);

Textbook Citations

Note:
This list includes all textbooks available at the Stanford University Bookstore Index in Fall 1996 (only the textbooks published on or after 1990), which include the term Computer Architecture in their title (or subtitles), and cover the general field of computer architecture.

Legend:
Position $X$—position in the ranking of referenced authors (s = shared position);
$Y$ citations—number of citations in the textbook (na = not applicable).

Flynn, M. J., Computer Architecture, Jones and Bartlett, USA (96)
position 1 (12 citations)

Bartee, T. C., Computer Architecture and Logic Design, McGraw-Hill, USA (91)
position 1 (2 citations)

Tabak, D., RISC Systems (RISC Processor Architecture), Wiley, USA (91)
position 1s (6 citations)

Stallings, W., Reduced Instruction Set Computers (RISC Architecture), IEEE CS Press, Los Alamitos, California, USA (90)
position 1s (3 citations)

position 3s (2 citations)

van de Goor, A. J., Computer Architecture and Design, Addison Wesley, Reading, Massachusetts, USA (2nd printing, 91)
position 4s (3 citations)

Tannenbaum, A., Structured Computer Organization (Advanced Computer Architectures), Prentice-Hall, USA (90)
position 5s (4 citations)

position 7s (2 citations)

Stallings, W., Computer Organization and Architecture, Prentice-Hall, USA (96)
position 9s (3 citations)

Murray, W., Computer and Digital System Architecture, Prentice-Hall, USA (90)
position 10s (2 citations)

Wilkinson, B., Computer Architecture, Prentice-Hall, USA (91)
position >10 (2 citations)

Decegama, A., The Technology of Parallel Processing (Parallel Processing Architectures), Prentice-Hall, USA (90)
position >10s (2 citations)
Baron, R. J., Higbie, L., *Computer Architecture*, Addison-Wesley, USA (92)

*position >10s (1 citation)*

Tabak, D., *Advanced Microprocessors (Microcomputer Architecture)*, McGraw-Hill, USA (95)

*position >10s (1 citation)*


*position >10s (1 citation)*


*na (0 citations)*


*na (0 citations)*

Kain, K., *Computer Architecture*, Addison-Wesley, USA (95)

*na (0 citations)*

Shiva, S., Pipelined and Parallel Computer Architectures, Harper Collins, USA (96)

*na (0 citations)*

Heuring, V., Jordan, H., Computer Systems Design and Architecture, Addison Wesley Longman, USA (97)

*na (0 citations)*
A Short Biosketch of the Author

Dr. Veljko Milutinovic is a faculty member in the Department of Computer Engineering, School of Electrical Engineering, University of Belgrade, Belgrade, Serbia, Yugoslavia. Before that, for over a decade, he was on various faculty positions in the Department of Computer Engineering, School of Electrical Engineering, Purdue University, West Lafayette, Indiana, USA. Before that, he received his Ph.D., M.Sc., and B.Sc. from the University of Belgrade. He published over 50 papers in IEEE periodicals, and presented over 100 papers at conferences worldwide. He is the single author of 4 books (which were translated into several languages), and editor or co-editor of 16 books (two of them in co-operation with two Nobel Laureates). His work is referenced more than 50 times in the Science Citation Index (excluding self-citations and citations of the former co-authors). As far as the textbooks with the term Computer Architecture in the title or sub-titles, according to Stanford University Bookstore Index, he is the most referenced author in 5 widely used textbooks, and among the most referenced authors in most of the other textbooks from the same group. As an invited professor he taught graduate courses or presented research lectures at all of the top 10 universities in the USA, plus over 100 other universities/companies worldwide. He consulted for companies like Intel, Sun Microsystems, NCR, RCA, Encore, Unisys, IBM, QSI, Hewlett-Packard, Fairchild. Honeywell, Delco, Aerospace Corporation, Electrospace Corporation, etc. His major strength are hardware prototype implementations of which several were implemented by himself alone (including the first DPSK multiprocessor system with 17 microprocessors), or within a team (including the first 200MHz GaAs RISC), as a project leader (including the first RDSM board for personal computers).