# Transaction Models and Algorithms for Improved Transaction Throughput

Samuel Kaspi

This thesis is presented in fulfillment of
the requirements for the degree of
Doctor of Philosophy

Computer and Mathematical Sciences
School of Communications and Informatics
Faculty of Engineering and Science

Victoria University Of Technology
2002

# Abstract

Currently, e-commerce is in its infancy, however its expansion is expected to be exponential and as it grows, so too will the demands for very fast real time online transaction processing systems. One avenue for meeting the demand for increased transaction processing speed is conversion from disk-based to in-memory databases. However, while in-memory systems are very promising, there are many organizations whose data is too large to fit in in-memory systems or who are not willing to undertake the investment that an implementation of an in-memory system requires. For these organizations an improvement in the performance of disk-based systems is required.

Accordingly, in this thesis, we introduce two mechanisms that substantially improve the performance of disk-based systems. The first mechanism, which we call a contention-based scheduler, is attached to a standard 2PL system. This scheduler determines each transaction's probability of conflict before it begins executing. Using this knowledge, the contention-based scheduler allows transactions into the system in both optimal numbers and an optimal mix. We present tests that show that the contention-based scheduler substantially outperforms standard 2PL concurrency control in a wide variety of disk-based hardware configurations. The improvement though most pronounced in the

throughput of low contention transactions extends to all transaction types over an extended processing period.

We call the second mechanism that we develop to improve the performance of disk-based database systems, enhanced memory access (EMA). The purpose of EMA is to allow very high levels of concurrency in the pre-fetching of data thus bringing the performance of disk-based systems close to that achieved by in-memory systems. The basis of our proposal for EMA is to ensure that even when conditions satisfying a transaction's predicate change between pre-fetch time and execution time, the data required for satisfying transactions' predicates are still found in memory. We present tests that show that the implementation of EMA allows the performance of disk-based systems to approach the performance achieved by in-memory systems. Further, the tests show that the performance of EMA is very robust to the imposition of additional costs associated with its implementation.

# Declaration

This thesis contains no material which has been accepted for the award of any other degree or diploma in any university. The material presented in this thesis is the product of the author's own independent research under the supervision of Professor Clement H.C. Leung.

Some of the materials presented in this thesis has been published in various publications. This thesis is less than 100,000 words in length.

Samuel Kaspi

June 28 2002

# A List of External Publications

1. Kaspi, S., “Optimizing Transaction Throughput in Databases Via an Intelligent Scheduler”*, Proceedings of the 1997 IEEE International Conference on Intelligent Processing Systems*, Beijing, October, 1997, pp.1337 – 1341

2. Kaspi, S., “The Use of Contention-Based Scheduling For Improving The Throughput of Locking Systems”, *Research Communications, Advances In Databases And Information Systems 5th East - European Conference ADBIS' 2001*, A. Caplinskas andJ. Eder (Eds.), Vol. 1, Vilnius, Lithuania, September 2001, pp 115-124

3. Leung, C.H.C. and S. Kaspi, “A Flexible Paradigm for Semantic Integration in Cooperative Heterogeneous Databases” *Proceedings of FGCS '94, ICOT*, Tokyo, December 1994

# Acknowledgements

I would like to express my appreciation and gratitude to my supervisor, Professor Clement H.C. Leung for enabling me to accomplish this work by providing me guidance in all aspects of my research.

I am grateful to my employer, Victoria University for encouraging me to undertake this work with both financial support and grants in the form of paid leave.

I would like to apologize to my friends and family for the long periods of time during which my research made me unavailable to them. I am thankful that my friends have made the effort to keep in contact with me and that my wife hasn't killed me for constantly getting up in the middle of the night to monitor my experiments.

Although my father died early in the history of this research, I owe him more than I can say. He was a wonderful person and a model of perseverance and dedication.

# Tables of Contents

**Abstract**.....i

**Declaration**.....ii

**List of External Publications**.....iii

**Acknowledgements**.....iv

**Table of Contents**.....vi

**List of Figures and Tables**..... ...x

**Chapter 1 Introduction**..... 1

**1.1 Objective**..... 1

**1.2 Summary of Contributions**..... 2

**1.3 Thesis Structure**..... 4

**Chapter 2 An Overview of Transaction Processing**..... 8

**2.1 Introduction**..... 8

**2.2 Two Phase Locking**..... 9

2.2.1 INTRA-TRANSACTION PARALLELISM..... 12

2.2.2 FIELD CALLS AND ESCROW LOCKING..... 14

**2.3 Optimistic Concurrency**..... 17

**2.4 Running Priority and Wait Depth Limited**..... 19

**2.5 Performance Comparisons**..... 19

**2.6 Further Reading**..... 21

**Chapter 3 An Analysis of the Behavior of Concurrency Control Mechanisms ....................... 23**

**3.1 Introduction ........................................................................................................ 23**

**3.2 An Overview of 2PL ........................................................................................... 24**

3.2.1 OTHER BEHAVIORAL PROPERTIES OF 2PL CONCURRENCY CONTROL .................................................................... 28

**3.3 Alternative Concurrency Control Mechanisms .................................................. 31**

3.3.1 OPTIMISTIC CONCURRENCY CONTROL ........................................................................................................ 32

3.3.1.1 VARIABILITY OF TRANSACTION SIZE AND CONTENTION AND PERFORMANCE ...................................................... 35

3.3.2 WAIT DEPTH LIMITED (WDL) ..................................................................................................................... 36

3.3.3 ACCESS INVARIANCE AND WASTED WORK IN DISK-BASED SYSTEMS .................................................................. 38

**3.4 Evaluation Systems ............................................................................................ 40**

3.4.1 THE DATABASE AND TRANSACTION SUBSYSTEMS ........................................................................................... 40

3.4.2 HARDWARE SUBSYSTEMS ............................................................................................................................ 41

3.4.3 THE CONCURRENCY CONTROL SUBSYSTEMS ................................................................................................... 43

3.4.4 THE PROCESSING SUBSYSTEM ....................................................................................................................... 44

**3.5 Performance Results ........................................................................................... 45**

3.5.1 THE PERFORMANCE OF IN-MEMORY SYSTEMS .................................................................................................. 45

3.5.1.1 THE PERFORMANCE OF IN-MEMORY SYSTEMS WITH VERY FAST PROCESSORS .................................................. 50

3.5.2 THE PERFORMANCE OF DISK-BASED SYSTEMS ................................................................................................. 53

3.5.3 A COMPARISON OF THE PERFORMANCE OF IN-MEMORY AND DISK-BASED SYSTEMS ......................................... 65

**3.6 Thrashing in 2PL Concurrency Control Systems ............................................... 71**

**3.7 Summary ............................................................................................................. 80**

**Chapter 4. New Strategies for Improving the Performance of 2PL concurrency Control ..... 82**

**4.1 Introduction ........................................................................................................ 82**

**4.2 Improving the Performance of 2PL by Manipulating Contention ......................... 83**

4.2.1 MECHANICAL PRINCIPLES ............................................................................................................................ 84

**4.3 Scheduling Heuristics .......................................................................................... 86**

4.3.1 THE MEASUREMENT OF CONTENTION ............................................................................................................ 87

4.3.2 SCHEDULING BY TRANSACTION TYPE ............................................................................................................ 90

4.3.3 SETTING TRANSACTIONS PRIORITY ................................................................................................................ 94

4.3.4 THE DETERMINATION OF AGGREGATE CONTENTION ........................................................................................ 95

**4.4 Controlling Thrashing** ........ **98**

**4.5 Summary** ........ **103**

**Chapter 5. Enhanced Memory Access** ........ **105**

**5.1 Introduction** ........ **105**

**5.2 An Overview of the Principles of EMA** ........ **106**

**5.3 Classification of Update Predicates** ........ **112**

**5.4 Mechanics of the Implementation of EMA** ........ **118**

5.4.1 THE MANAGEMENT OF DATA ........ 123

**5.7 Summary** ........ **127**

**Chapter 6. Validation of the Contention-Based Scheduler** ........ **129**

**6.1 Introduction** ........ **129**

**6.2 Performance Results for the Contention-Based Scheduler** ........ **131**

**6.3 Allowing For Errors in the Measurement of Contention** ........ **135**

**6.4 Restricted Number of Transactions** ........ **139**

**6.5 The Performance of the Scheduler Over Extended Time Periods** ........ **143**

**6.6 Controlling Thrashing** ........ **151**

**6.7 Very High Concurrencies** ........ **156**

**6.8 Summary** ........ **162**

**Chapter 7. Validation of EMA** ........ **164**

**7.1 Introduction** ........ **164**

**7.2 The Performance Under EMA with 0 Costs** ........ **166**

**7.3 Adding Costs to EMA** ........ **182**

**7.4 2PL EMA Systems With and Without Thrashing Control** ........ **192**

**7.5 Summary** ........ **201**

**Chapter 8. Conclusion** ........ **202**

**8.1 Contributions and Achievements** ........ **202**

**8.2 Future Research** ........ **207**

**Bibliography** ........ **209**

**Appendix A** ........ **218**

**Appendix B** ........ **222**

# List of Figures and Tables


TABLE 2.1. TWO EXAMPLES OF TRANSACTIONS OPERATING CONCURRENTLY UNDER 2PL. ............ 10

FIGURE 2.1. THEORETICAL UPPER LIMITS ON ACTIVE TRANSACTIONS UNDER 2PL. ............ 12

FIGURE 2.2. FIRST PHASE OF FIELD CALLS AND ESCROW LOCKS. ............ 16

FIGURE 2.3. SECOND PHASE OF FIELD CALLS AND ESCROW LOCKS. ............ 16

FIGURE 2.4. UPPER BOUNDS ON EXPECTED ACTIVE TRANSACTIONS FOR ESSENTIAL BLOCKING POLICIES. ............ 18

FIGURE 3.1. AN EXAMPLE OF TRANSACTION ACTIVITY IN THE FIRST CYCLE ............ 25

FIGURE 3.2. AN EXAMPLE OF TRANSACTION ACTIVITY IN THE SECOND CYCLE. ............ 26

FIGURE 3.3. AN EXAMPLE OF ESSENTIAL BLOCKING ............ 34

FIGURE 3.4. THE THREE INITIAL (TEMPORARY) CASES OF TRANSACTION STATES. ............ 37

FIGURE 3.5. WDL(1) ALGORITHM GIVEN THE STATES OUTLINED IN FIGURE 3.2 ............ 38

FIGURE 3.6. 1248 PROCESSORS EACH OPERATING AT 4 MIPS ............ 46

FIGURE 3.7. 96 PROCESSORS EACH OPERATING AT 100 MIPS ............ 47

FIGURE 3.8. A COMPARISON OF THROUGHPUTS UNDER 2PL, OPTIMISTIC KILL AND WDL CONCURRENCY CONTROL METHODS FOR A SYSTEM WITH 96 PROCESSORS EACH OPERATING AT 200 MIPS ............ 47

FIGURE 3.9. A COMPARISON OF THROUGHPUTS OF THE LARGEST TRANSACTIONS ($T_4$) UNDER OPTIMISTIC KILL AND WDL CONCURRENCY CONTROL METHODS FOR A SYSTEM WITH 1248 PROCESSORS EACH OPERATING AT 4 MIPS ............ 49

FIGURE 3.10. A COMPARISON OF THROUGHPUTS OF THE LARGEST TRANSACTIONS ($T_4$) UNDER OPTIMISTIC KILL AND WDL CONCURRENCY CONTROL METHODS FOR A SYSTEM WITH 96 PROCESSORS EACH OPERATING AT 100 MIPS ............ 49

FIG 3.12. A COMPARISON OF THROUGHPUT UNDER 2PL CONCURRENCY CONTROL IN SYSTEMS COMPOSED OF 96 PROCESSORS EACH OPERATING AT 200 MIPS, 20 PROCESSORS EACH OPERATING AT 1000 MIPS AND 10 PROCESSORS EACH OPERATING AT 2000 MIPS ............ 51

FIGURE 3.13. A COMPARISON OF THROUGHPUT UNDER WDL CONCURRENCY CONTROL IN SYSTEMS COMPOSED OF 96 PROCESSORS EACH OPERATING AT 200 MIPS, 20 PROCESSORS EACH OPERATING AT 1000 MIPS AND 10PROCESSORS EACH OPERATING AT 2000 MIPS ............ 51

FIGURE 3.14. A COMPARISON OF THROUGHPUT UNDER OPTIMISTIC CONCURRENCY CONTROL IN SYSTEMS COMPOSED OF 96 PROCESSORS EACH OPERATING AT 200 MIPS, 20 PROCESSORS EACH OPERATING AT 1000 MIPS AND 10 PROCESSORS EACH OPERATING AT 2000 MIPS ............ 52

FIG 3.15. A COMPARISON OF THROUGHPUTS UNDER 2PL, OPTIMISTIC KILL AND WDL CONCURRENCY CONTROL METHODS FOR A SYSTEM WITH 1248 PROCESSORS EACH OPERATING AT 4 MIPS AND WITH DISK ACCESS AT 15 MILLISECONDS PER ACCESS ............ 54

Figure 3.16. A comparison of throughputs under 2PL, optimistic die-kill and WDL concurrency control methods for a system with 96 processors each operating at 100 MIPS and with disk access at 15 milliseconds per access ........ 55
Figure 3.17. A comparison of throughputs under 2PL, optimistic die-kill and WDL concurrency control methods for a system with 96 processors each operating at 200 MIPS and with disk access at 15 milliseconds per access ........ 55
Figure 3.18. A comparison of throughputs under 2PL, optimistic die-kill and WDL concurrency control methods for a system with 20 processors each operating at 1000 MIPS and with disk access at 15 milliseconds per access. ........ 56
Figure 3.19. A comparison of throughputs under 2PL, optimistic die-kill and WDL concurrency control methods for a system with 10 processors each operating at 2000 MIPS and with disk access at 15 milliseconds per access ........ 56
Figure 3.20. A comparison of the throughput of large transactions given no access invariance under optimistic kill and WDL concurrency control methods for a system with 1248 processors each operating at 4 MIPS and with disk access at 15 milliseconds per access ........ 58
Figure 3.21. A comparison of the throughput of large transactions given access invariance under optimistic kill-die and WDL concurrency control methods for a system with 96 processors each operating at 100 MIPS and with disk access at 15 milliseconds per access........ 59
Figure 3.22. A comparison of the throughput of large transactions given access invariance under optimistic kill-die and WDL concurrency control methods for a system with 96 processors each operating at 200 MIPS and with disk access at 15 milliseconds per access........ 59
Figure 3.23. A comparison of the throughput of large transactions given access invariance under optimistic kill-die and WDL concurrency control methods for a system with 20 processors each operating at 1000 MIPS and with disk access at 15 milliseconds per access........ 60
Figure 3.24. A comparison of the throughput of large transactions given access invariance under optimistic kill-die and WDL concurrency control methods for a system with 10 processors each operating at 2000 MIPS and with disk access at 15 milliseconds per access........ 60
Figure 3.25. The throughput of systems of systems operating under 2PL concurrency control........ 62
Figure 3.26. The throughput of systems of systems operating under WDL concurrency control ........ 62
Figure 3.27: The throughput of systems of systems operating under optimistic concurrency control..... 63
Table 3.1. The time in seconds required to complete a transaction by CPU speed and transaction type .. 65
Figure 3.28. A comparison of in-memory and disk-based systems with each configuration having 1248 processors operating at 4 MIPS per processor ........ 66
Figure 3.29. A comparison of in-memory and disk-based systems with each configuration having 96 processors operating at 100 MIPS per processor ........ 67
Figure 3.30. A comparison of in-memory and disk-based systems with each configuration having 96 processors operating at 200 MIPS per processor ........ 67
Figure 3.31. A comparison of in-memory and disk-based systems with each configuration having 20 processors operating at 1000 MIPS per processor........ 68

FIGURE 3.32. A COMPARISON OF IN-MEMORY AND DISK-BASED SYSTEMS WITH EACH CONFIGURATION HAVING 10 PROCESSORS OPERATING AT 2000 MIPS PER PROCESSOR ........ 68
FIGURE 3.33. A COMPARISON OF IN-MEMORY SYSTEMS WITH 96 PROCESSORS OPERATING AT 100 AND 200 MIPS PER PROCESSOR, 20 PROCESSORS OPERATING AT 1000 MIPS PER PROCESSOR AND 10 PROCESSORS OPERATING AT 2000 MIPS PER PROCESSOR OPERATING UNDER 2PL CONCURRENCY CONTROL WITH THEIR EQUIVALENT DISK-BASED SYSTEMS OPERATING AT 0 CONTENTION AND A CONCURRENCY OF 100 ........ 70
FIGURE 3.34: THROUGHPUT OF $T_2$ TRANSACTIONS PER CYCLE OVER 8 CYCLES ........ 76
FIGURE 3.35. THROUGHPUT OF $T_3$ TRANSACTIONS PER CYCLE OVER 8 CYCLES ........ 76
FIGURE 3.36. THROUGHPUT OF THE SYSTEM WITH FIXED SIZE VARIABLE CONTENTION TRANSACTIONS PER CYCLE OVER 8 CYCLES ........ 77
FIGURE 3.37: THROUGHPUT OF $T_4$ TRANSACTIONS PER CYCLE OVER 8 CYCLES ........ 77
FIGURE 4.1. A DIAGRAMATIC ILLUSTRATION OF THE CONTENTION-BASED SCHEDULER ........ 85
FIGURE 4.2. THE CYCLICAL RELATIONSHIP BETWEEN PEAK AGGREGATE CONTENTION AND THROUGHPUT ........ 97
FIGURE4.3. AN ILLUSTRATION OF A THREE WAY DEADLOCK ........ 99
FIGURE 5.1. A SAMPLE STATE FOR A PRE-FETCHING SYSTEM. ........ 107
FIGURE 5.2. AN ILLUSTRATION OF THE BASIC PRINCIPLE BEHIND EMA. ........ 109
TABLE 5.1. AN EXAMPLE OF THE POSSIBLE EFFECTS OF CHANGES TO A TRANSACTION'S PREDICATE ........ 117
FIGURE 5.3: AN OVERVIEW OF THE ACTIVITY OF TRANSACTION ON ENTRY INTO AN EMA SYSTEM. ........ 119
FIGURE 5.4. THE COMPOSITION OF TRANSACTION MARKERS IN THE EMA SYSTEM . ........ 120
FIGURE 5.5. EXIT OF THE OLDEST TRANSACTION ........ 121
FIGURE 5.6. EXIT OF A TRANSACTION THAT IS NOT THE OLDEST TRANSACTION. ........ 122
FIGURE 5.7. THE PROCEDURE FOLLOWED BY A COMPLETING COMMITTING TRANSACTION. ........ 124
FIGURE 5.8. THE PROCEDURE FOLLOWED WHEN A PRE-FETCHED ITEM HASHES TO THE SAME ADDRESS AS AN EXISTING ITEM AND THE TIME-STAMP OF THE CURRENT DATA ITEM IS OLDER THAN THE TIME-STAMP OF THE OLDEST TRANSACTION. ........ 125
FIGURE 5.9. THE PROCEDURE FOLLOWED WHEN A PRE-FETCHED ITEM HASHES TO THE SAME ADDRESS AS AN EXISTING ITEM AND THE TIME-STAMP OF THE CURRENT DATA ITEM IS YOUNGER THAN THE TIME-STAMP OF THE OLDEST TRANSACTION. ........ 126
FIGURE 5.10. THE PROCEDURE FOLLOWED WHEN A PRE-FETCHED ITEM HASHES TO THE SAME ADDRESS AS AN EXISTING ITEM, THE TIME-STAMP OF THE CURRENT DATA ITEM IS YOUNGER THAN THE TIME-STAMP OF THE OLDEST TRANSACTION, AND OVERFLOW ALREADY HAS MORE THAN 1 ITEM ........ 127
FIGURE 6.1. A COMPARISON OF TOTAL THROUGHPUTS UNDER THE CONTENTION BASED SCHEDULER AND STANDARD 2PL CONCURRENCY CONTROL. ........ 132
FIGURE 6.2. A COMPARISON OF THROUGHPUT UNDER THE CONTENTION BASED SCHEDULER AND STANDARD 2PL CONCURRENCY CONTROL BY TRANSACTION TYPE. ........ 133
FIGURE 6.3. A COMPARISON OF TOTAL THROUGHPUTS UNDER THE CONTENTION BASED SCHEDULER AND STANDARD 2PL CONCURRENCY CONTROL WITH AN ALLOWANCE FOR ERROR IN THE MEASUREMENT OF CONTENTION. ........ 137

FIGURE 6.4. A COMPARISON OF THROUGHPUTS BY TRANSACTION TYPE WITH AN ALLOWANCE FOR ERROR IN THE MEASUREMENT OF CONTENTION BY THE SCHEDULER ........ 138
FIGURE 6.5. A COMPARISON OF TOTAL THROUGHPUTS UNDER THE CONTENTION BASED SCHEDULER AND STANDARD 2PL CONCURRENCY CONTROL WITH AN ARRIVAL RATE OF 1000 TRANSACTIONS PER SECOND. ........ 141
FIGURE 6.6 A COMPARISON OF THROUGHPUTS UNDER THE CONTENTION BASED SCHEDULER AND STANDARD 2PL CONCURRENCY CONTROL WITH AN ARRIVAL RATE OF 1000 TRANSACTIONS PER SECOND BY TRANSACTION TYPE. ........ 142
FIG 6.7. TOTAL THROUGHPUT PER SECOND OVER A RANGE OF PROCESSING PERIODS ........ 148
FIG 6.8. THROUGHPUT PER SECOND OVER A RANGE OF PROCESSING PERIODS BY TRANSACTION TYPE ........ 149
TABLE 6.1. RATIO OF AVERAGE THROUGHPUT PER SECOND OVER A RANGE OF PROCESSING PERIODS FOR A STANDARD 2PL SYSTEM ........ 150
FIGURE 6.9. TOTAL THROUGHPUT PER SECOND OVER 1 MINUTE IN THE 1248 PROCESSOR BY 4 MIPS SYSTEM ........ 153
FIGURE 6.11. TOTAL THROUGHPUT ........ 157
FIGURE 6.12. THROUGHPUT OF $T_1$ TRANSACTIONS ........ 158
FIGURE 6.13. THROUGHPUT OF $T_2$ TRANSACTIONS ........ 159
FIGURE 6.14. THROUGHPUT OF $T_3$ TRANSACTIONS ........ 160
FIGURE 6.15. TTHROUGHPUT OF $T_4$ TRANSACTIONS ........ 161
FIGURE 7.1. A COMPARISON OF TOTAL THROUGHPUTS IN SYSTEMS CONTAINING 96 PROCESSORS EACH OPERATING AT 100 MIPS. ........ 167
FIGURE 7..2 A COMPARISON OF THROUGHPUTS IN SYSTEMS CONTAINING 96 PROCESSORS EACH OPERATING AT 100 MIPS BY TRANSACTION TYPE. ........ 168
TABLE 7.1. A MILLISECOND BREAKDOWN OF THE PERFORMANCE OF 2PL EMA ........ 170
FIGURE 7.3. A COMPARISON OF THE THROUGHPUTS OF IN-MEMORY AND EMA (WITH NO COST) UNDER BOTH WDL AND OPTIMISTIC CONCURRENCY CONTROL FOR SYSTEMS WITH A CONFIGURATION CONTAINING 96 PROCESSORS OPERATING AT 100 MIPS PER PROCESSOR. ........ 172
FIGURE 7.4. A COMPARISON OF TOTAL THROUGHPUTS IN SYSTEMS CONTAINING 96 PROCESSORS EACH OPERATING AT 200 MIPS ........ 174
FIGURE 7.5. A COMPARISON OF THE THROUGHPUT IN SYSTEMS CONTAINING 96 PROCESSORS EACH OPERATING AT 200 MIPS BY TRANSACTION TYPE. ........ 175
FIGURE 7.6. A COMPARISON OF TOTAL THROUGHPUTS IN SYSTEMS CONTAINING 20 PROCESSORS EACH OPERATING AT 1000 MIPS ........ 177
FIG .7.7. A COMPARISON OF THROUGHPUTS IN SYSTEMS CONTAINING 20 PROCESSORS EACH OPERATING AT 1000 MIPS BY TRANSACTION TYPE ........ 178
FIGURE 7.8. A COMPARISON OF TOTAL THROUGHPUTS IN SYSTEMS CONTAINING 10 PROCESSORS EACH OPERATING AT 2000 MIPS ........ 180
FIGURE 7.9. A COMPARISON OF THE THROUGHPUTS IN SYSTEMS CONTAINING 10 PROCESSORS EACH OPERATING AT 2000 MIPS BY TRANSACTION TYPE. ........ 181
FIGURE 7.10. TOTAL THROUGHPUT WITH DIVERSE COSTING REGIMES IN SYSTEMS CONTAINING 96 PROCESSORS EACH OPERATING AT 100 MIPS ........ 184

FIGURE 7.11. THROUGHPUT WITH DIVERSE COSTING REGIMES IN SYSTEMS CONTAINING 96 PROCESSORS EACH OPERATING AT 100 MIPS BY TRANSACTION TYPE. ........ 185
FIGURE 7.12. TOTAL THROUGHPUT WITH DIVERSE COSTING REGIMES IN SYSTEMS CONTAINING 96 PROCESSORS EACH OPERATING AT 200 MIPS ........ 186
FIGURE 7.13. THROUGHPUT WITH DIVERSE COSTING REGIMES IN SYSTEMS CONTAINING 96 PROCESSORS EACH OPERATING AT 200 MIPS BY TRANSACTION TYPE ........ 187
FIGURE 7.14. TOTAL THROUGHPUT WITH DIVERSE COSTING REGIMES IN SYSTEMS CONTAINING 20 PROCESSORS EACH OPERATING AT 1000 MIPS ........ 188
FIG 7.15. THROUGHPUT WITH DIVERSE COSTING REGIMES IN SYSTEMS CONTAINING 20 PROCESSORS EACH OPERATING AT 1000 MIPS BY TRANSACTION TYPE ........ 189
FIGURE 7.16. TOTAL THROUGHPUT WITH DIVERSE COSTING REGIMES IN SYSTEMS CONTAINING 10 PROCESSORS EACH OPERATING AT 2000 MIPS ........ 190
FIGURE 7.17. THROUGHPUT WITH DIVERSE COSTING REGIMES IN SYSTEMS CONTAINING 10 PROCESSORS EACH OPERATING AT 2000 MIPS BY TRANSACTION TYPE ........ 191
FIGURE 7.18. TOTAL THROUGHPUT IN 2PL SYSTEMS WITH WITHOUT THRASHING CONTROL IN SYSTEMS CONTAINING 96 PROCESSORS EACH OPERATING AT 100 MIPS ........ 193
FIGURE 7.19. THROUGHPUT IN 2PL SYSTEMS WITH WITHOUT THRASHING CONTROL IN SYSTEMS CONTAINING 96 PROCESSORS EACH OPERATING AT 100 MIPS BY TRANSACTION TYPE ........ 194
FIGURE 7.20. TOTAL THROUGHPUT IN 2PL SYSTEMS WITH WITHOUT THRASHING CONTROL IN SYSTEMS CONTAINING 96 PROCESSORS EACH OPERATING AT 200 MIPS ........ 195
7.21. THROUGHPUT IN 2PL SYSTEMS WITH WITHOUT THRASHING CONTROL IN SYSTEMS CONTAINING 96 PROCESSORS EACH OPERATING AT 200 MIPS BY TRANSACTION TYPE ........ 196
FIGURE 7.22. TOTAL THROUGHPUT IN 2PL SYSTEMS WITH WITHOUT THRASHING CONTROL IN SYSTEMS CONTAINING 20 PROCESSORS EACH OPERATING AT 1000 MIPS. ........ 197
FIGURE 7.23. THROUGHPUT IN 2PL SYSTEMS WITH WITHOUT THRASHING CONTROL IN SYSTEMS CONTAINING 20 PROCESSORS EACH OPERATING AT 1000 MIPS BY TRANSACTION TYPE. ........ 198
FIGURE 7.24.. TOTAL THROUGHPUT IN 2PL SYSTEMS WITH WITHOUT THRASHING CONTROL IN SYSTEMS CONTAINING 10 PROCESSORS EACH OPERATING AT 2000 MIPS ........ 199
FIGURE 7.25. THRASHING CONTROL IN SYSTEMS CONTAINING 10 PROCESSORS EACH OPERATING AT 2000 MIPS BY TRANSACTION TYPE. ........ 200
FIGURE A.1. A UML CLASS DIAGRAM OF OUR SIMULATION PROGRAMS ........ 219
FIGURE A..2. IN-MEMORY 2000 MIPS ........ 221
TABLE B.1. COST BENEFIT OF DIVERSE THRASHING ........ 224
FIGURE B.1. 1248 PROCESSORS BY 4 MIPS DISK BASED SYSTEM ........ 225
FIGURE B.2. 96 PROCESSORS BY 100 MIPS DISK BASED SYSTEM ........ 225
FIGURE B.3. 96 PROCESSORS BY 200 MIPS DISK BASED SYSTEM ........ 226
FIGURE B.4. 20 PROCESSORS BY 1000 MIPS DISK BASED SYSTEM ........ 226
FIGURE B.5. PROCESSORS BY 2000 MIPS DISK BASED SYSTEM ........ 227

# *Chapter 1*

## *Introduction*

### 1.1 Objective

The main objective of this thesis is to develop and test algorithms that substantially increase the throughput of transactions in disk-based database management systems that comply with the ACID (Atomicity, Consistency, Isolation and Durability) properties. The focus will be on developing algorithms that can work in conjunction with (though not necessarily exclusively) two-phase concurrency control (2PL).

While it is possible to achieve very large increases in transaction throughputs by converting from disk-based to in-memory databases, it is likely that disk-based DBMS' (Data Based Management System) will continue to be dominant in the foreseeable future. There are several reasons for this. Firstly, there are many organizations whose data is too large to fit into current or soon to be released in-memory systems. For other

organizations, the total cost of conversion does not justify the switch to in-memory systems. That is, while the cost of memory is relatively cheap, the cost of converting data and existing applications that access that data is prohibitive for many organizations. Finally, in-memory systems are new and most organizations are inherently conservative. Thus, most organizations will wait for in-memory systems to become well established before they consider them.

It is possible to increase the throughput levels of some databases by relaxing the ACID properties (Atomicity, Consistency, Isolation and Durability) by which most DBMS' are constrained. Such a relaxation allows a higher level of effective concurrency than is possible in DBMS' where ACID is enforced. However, such a relaxation risks compromising the integrity of an organization's data and as such is only acceptable where the integrity of data is not critical.

Except for some experimental systems, to our knowledge there are no commercial DBMS' that do not implement some variation of 2 phase locking (2PL). This is despite the fact that alternatives to 2PL such as optimistic concurrency control have been available for around 20 years. Given the reluctance of DBMS vendors and users to abandon two phase locking (2PL), it is likely that only algorithms that improve the performance of 2PL systems will be commercially acceptable.

## 1.2 Summary of Contributions

The contributions of this thesis to improving the performance of transaction processing in database systems are summarized as follows:

1. The demonstration that the potential performance of concurrency control mechanisms in in-memory systems, rather than the performance of different concurrency control methods under equivalent disk-based hardware should form the benchmark for judging the merit of a transaction processing system. We believe that the demonstration of the huge gap in performance between any concurrency control mechanism in in-memory systems and the best performance of the most successful conventional concurrency control mechanism in disk-based systems makes this conclusion inescapable. That is, we feel that determining whether, in a disk-based system, one type of concurrency control mechanism can yield even several hundred percent improvements in performance pales into insignificance against the many thousand percent improvements in performance that can be achieved by a change from a disk-based to an in-memory system.

2. The development of the contention based scheduler, which significantly improves the performance of disk-based 2PL systems. The basic premise behind the contention-based scheduler is that in database systems containing transactions with varying contentions, the behavior of transactions varies consistently but predictably according to their contention. It is thus possible for a scheduler that can measure transactions' contention as they arrive, to sort these transactions into queues of transactions with similar contentions and then to manipulate the number of transactions allowed into the system by their contention class. By doing so, the contention-based scheduler can dramatically increase the system's throughput in 2PL systems.

3. The development of the enhanced memory access (EMA) system, which allows the performance of disk –based concurrency control mechanisms to approaches that of in-memory systems. EMA is a variation of the access invariance and pre-fetch schemes outlined in [16] and [17]. However, unlike access invariance, EMA does not assume a constant database state or that a transaction will access the same set of objects in all its execution histories. Rather, by controlling the length of time that data resides in memory, it guarantees that having pre-fetched its data, any object that a transaction requires will always be in memory even if the set of data required at actual execution time varies from that established during pre-fetch. This allows a very large number of transactions to be pre-fetched and then executed entirely in memory thus dramatically improving system throughput. EMA improves the performance of disk-based systems (using either 2PL, WDL or optimistic concurrency control) to near that achieved by in-memory systems.

## 1.3 Thesis Structure

No research is conducted in a vacuum and an understanding of any piece of research requires some understanding of the history of developments in the area being researched and its current state of the art. Accordingly in chapter 2 we present an overview and literature survey of the field of transaction processing.

In chapter 3, we examine the behavior of three concurrency control mechanisms and their performance under various hardware configurations. The concurrency control

mechanisms examined are two-phase locking (2PL), optimistic concurrency control and wait depth limited (WDL). Particular attention is given to the problem of thrashing in 2PL systems.

The hardware configurations used for testing fall into two basic categories: disk-based systems and in-memory (or main memory) systems. For each in-memory configuration, there is a disk-based configuration with an equivalent number of processors with the same speed per processor. The number of disks available to the disk-based systems using these processor configurations are based on the algorithm used in [17] and [51] which applies Little's law and using total CPU processing power, the number of expected disk accesses and a CPU and disk utilization ratio of 75/20 as parameters. An assumption underlying this calculation is that the cache held sufficient data to ensure that 0.625 of items required in a first access is in memory.

These hardware configurations, database subsystems, processing and transaction subsystems outlined in chapter 3, also form the basis of tests presented later in the thesis. The main focus of this thesis is high capacity systems and we believe that the range of hardware configurations tested is sufficiently wide so as to represent a reasonable sample of high capacity configurations that are or will soon be commercially available.

The results presented in chapter 3 suggest that for those organizations that can fit all their data in memory, the best way to improving the performance of their transaction processing systems lies in switching to in-memory systems rather than improving the

performance of their disk-based systems. However, many organizations cannot satisfy this requirement, that is, they cannot fit all their data in memory and for these organizations, improvement in the performance of their transaction processing systems can only come about by improving the performance of disk-based systems. The search for such improvements is the primary purpose of the remainder of our thesis.

In chapter 4 we introduce the contention-based scheduler. We develop this scheduler because while in-memory systems are now becoming available, disk-based hardware using 2PL concurrency control are dominant in the market place and are likely to remain so for some time. This scheduler operates by measuring transactions' contention as they arrived sorting these transactions into queues of transactions with a similar contention and then manipulates the number of transactions allowed into the system by the contention class.

In chapter 5 we present a modified variation of access invariance that we called enhanced memory access, (EMA). The purpose of EMA is to allow very high levels of concurrency in the pre-fetching of data thus bringing the performance of disk-based systems close to that achieved by in-memory systems. The basis of accesses invariance is that conditions satisfying a predicate at pre-fetch time do not change between pre-fetch time and the time when actual execution takes place. This assumption only holds at limited concurrencies. EMA does not assume unchanged conditions between pre-fetch time and execution time, but rather, ensures that even where such changes occur, the data required to satisfy transactions' predicates are still found in memory. This allows pre-

fetching at extremely high concurrencies. Consequently throughput can be increased to levels near those achieved by in-memory systems.

In chapter 6 we compare the performance of our contention-based scheduler against the performance achieved by disk-based systems that do not use our scheduler while in chapter 7 we compare the performance of our EMA systems against the performance achieved by in-memory systems. Our systems are tested both with and without mechanisms to control threshing. In Chapter 8 we conclude our thesis.

# *Chapter 2*

## *An Overview of Transaction Processing*

### 2.1 Introduction

In commercial databases, transactions are generally small and independent of each other. Thus, one could expect that processing them at a concurrency level limited by hardware capacity would maximize the throughput of transactions. However, hardware capacity is not the only constraint limiting transaction processing. Because transactions modify data, where data integrity is crucial, as it is in most database applications, transactions must conform to certain generally accepted constraints known by their acronym as the ACID properties. With respect to concurrency, the limiting properties are Atomicity and Isolation.

To ensure adherence to these properties, the concurrency of transactions needs to be managed. In modern databases, the upper limits to concurrency are set by this need to guarantee integrity rather than by hardware capacity.

## 2.2 Two Phase Locking

By far the most widely used protocols for controlling concurrency are those known as two-phase locks (2PL). As the name suggests, transactions under this concurrency management scheme are executed in two phases. In the first phase, a transaction attempts to acquire locks for all the objects that it requires. If it is successful, it can then access these objects and modify them if this is required. Upon completion, it commits its updates and then in the second phase, it releases its locks. If the transaction cannot obtain all of its required locks, it goes to “sleep” until it is “woken” up when the transaction upon which it is waiting completes, commits its data and releases all its locks.

Since only a write changes the database, a transaction, $T_2$, is denied a lock for an object, $o_1$, only if another transaction, $T_1$, has previously obtained a lock on $O_1$ for a write or if $T_1$, has previously obtained a lock on $O_1$ for a read and $T_2$ requires $O_1$ for a write.

The operation of 2PL is illustrated in table 2.1 below. In table 2.1, in example 1, the set of objects required by the two objects is disjoint. Thus, each transaction can execute concurrently and complete normally. In example 2, when $T_2$ tries to get a lock for object $O_1$ at time 4, it finds that its request is denied and it has to wait until $T_1$ unlocks its objects at time 5. Note that this is so even though $T_1$ has finished using object $O_1$ at time 2.

However, under 2PL, unlocking cannot be interleaved but must be completed in one phase at the end of the transaction.

The major impediment to 2PL's performance is its tendency to develop long chains of waiting transactions. To illustrate this problem, let us assume three transactions - $T_1$, $T_2$ and $T_3$. Let us assume that $T_1$ arrives first and obtains all its locks. Transaction $T_2$ arrives next but finds that one of its required objects has been locked by $T_1$ and it thus has to wait. When transaction $T_3$ arrives, it finds that an object that it requires has been locked by $T_2$ and it thus has to wait until $T_2$ completes.

| transaction | time | object | operation | transaction | time | object | operation |
|---|---|---|---|---|---|---|---|
| $T_1$ | 1 | $O_1$ | Readlock | $T_1$ | 1 | $O_1$ | Writelock |
| $T_2$ | 2 | $O_2$ | Readlock | $T_2$ | 2 | $O_2$ | Readlock |
| $T_1$ | 2 | $O_1$ | Read | $T_1$ | 2 | $O_1$ | Write |
| $T_1$ | 3 | $O_3$ | WriteLock | $T_1$ | 3 | $O_3$ | WriteLock |
| $T_2$ | 3 | $O_2$ | Read | $T_2$ | 3 | $O_2$ | Read |
| $T_1$ | 4 | $O_3$ | Write | $T_2$ | 4 | $O_1$ | WriteLock Failed - wait |
| $T_2$ | 4 | $O_4$ | Writelock | $T_1$ | 4 | $O_3$ | Write |
| $T_1$ | 5 | $O_1$ | Unlock | $T_1$ | 5 | $O_1$ | Unlock |
| $T_2$ | 5 | $O_4$ | Write | $T_1$ | 5 | $O_3$ | Unlock, Wake $T_2$ |
| $T_1$ | 6 | $O_3$ | Unlock | $T_2$ | 6 | $O_1$ | Writelock |
| $T_2$ | 6 | $O_2$ | Unlock | $T_2$ | 7 | $O_1$ | Write |
| $T_2$ | 7 | $O_4$ | Unlock | $T_2$ | 8 | $O_2$ | Unlock |
| | | | | $T_2$ | 8 | $O_1$ | Unlock |

**Table 2.1. Two examples of transactions operating concurrently under 2PL.**

This tendency to develop long and deep queues of waiting transactions makes 2PL extremely sensitive to data contention. Where data contention is low, 2PL works

reasonably well. However, in high contention environments the maximum effective level of concurrency is severely limited. The limits to concurrency in 2PL systems are discussed in [15] and [47]. Figure 2.1 below shows the theoretical limits placed on concurrency in 2PL systems by data contention, $p$. These are taken from [15].

To improve the performance of 2PL various schemes have been developed. These fall into two broad categories –

1. Those schemes which attempt to minimize the collision cross section of transactions. These algorithms can be divided into two further categories. Firstly, algorithms such as transaction chopping as found in [46] and secondly, algorithms such as field calls [21] and escrow locking [43] and [40]. The aim of transaction chopping algorithms is to allow intra-transaction parallelism while the aim of field calls and escrow locking is to minimize the time that locks are held by adding semantics to lock execution.

2. Schemes which attempt to minimize the effects of long wait chains –for example running priority and wait depth limited (WDL) as found in [17] and [50].

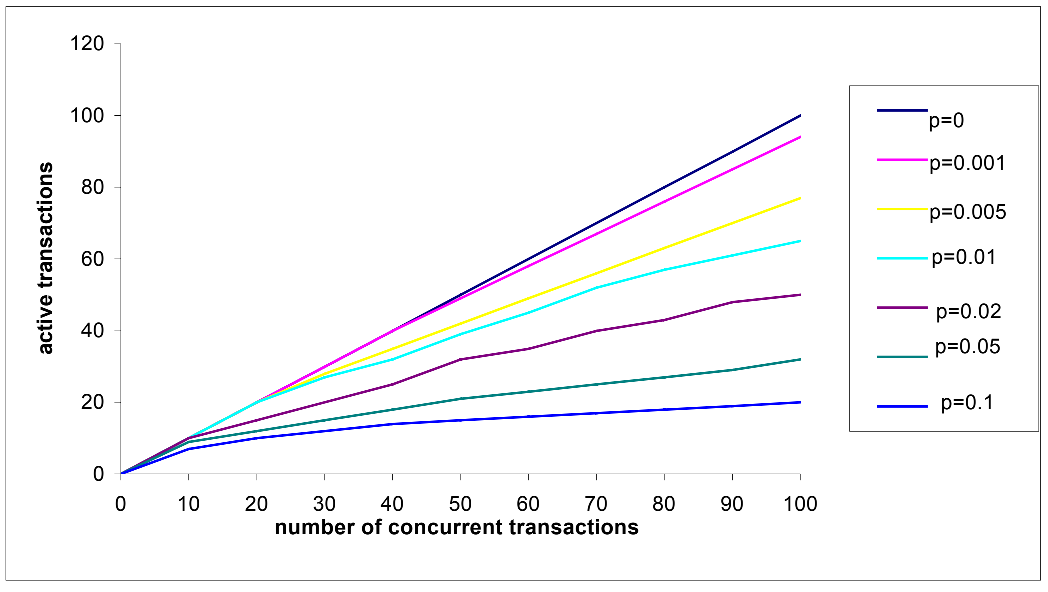



**Figure 2.1. Theoretical upper limits on active transactions under 2PL.**

### *2.2.1 Intra-Transaction Parallelism*

In transaction processing, concurrency is generally treated as the number of independent transactions that can run concurrently. In theory, particularly where larger transactions are concerned, it is also theoretically possible to break down transactions into their components and perform these in parallel. Implementing intra-transaction parallelism would increase the overall level of effective concurrency by increasing the number of processes that could be run at any one time without collision. Thus for example, given 10 processors, 10 non- partitioned transactions could run concurrently with each transaction having the potential to conflict with nine other transactions. If each transaction were partitioned into five sub-processes with each sub-process running concurrently, all 10

processors would be utilized with two transactions running. Thus, each transaction could conflict with only one other transaction.

Intra-transaction parallelism would also increase the overall level of effective concurrency by reducing the time profile during which collisions would take effect. Using the above example, if it took 10 un-partitioned transactions 1 second to run, where a conflict occurred, a waiting transaction would have to wait for up to a second to obtain its locks. With partitioned transactions, if each process took a fifth of a second to run, a waiting transaction would have to wait a fifth of a second.

However, intra-transaction parallelism is difficult to implement with flat transactions. The issues here include the problem of guaranteeing isolation within the sub-transactions as where two parallel sub-transactions within a transaction may conflict with each other and the increase in the complexity of the lock manager particularly the deadlock detection mechanism.

Research into intra-transaction parallelism includes [46] which investigates transaction-chopping algorithms. These algorithms under certain restrictive assumptions guarantee the serializability of sub-transactions. Unsurprisingly, simulation results presented in this paper find that where data contention is high, chopping up transactions and running the sub transactions concurrently increases throughput significantly. Unfortunately, the assumptions that underlie these algorithms are too restrictive to make these algorithms generally applicable.

Thus the current state of intra-transaction processing is that "[intra-transaction] parallelism without nested transactions is so complex that few are likely to want to use it outside of very limited contexts. Parallel SQL systems such as Teradata and NonStop SQL, use parallelism either for read only queries or in very carefully controlled update cases" [22].

### *2.2.2 Field Calls and Escrow Locking*

Field calls were developed by [21] as a response to the problem of hot spots. That is, the small proportion of data in a database that is disproportionately accessed by transactions and which consequently causes most of the data contention in the database. [43] and [40] extended this work to include escrow reads.

The basic idea behind this set of protocols is that transactions are given short-term locks during which their anticipated changes are tested against the database. If a condition is met, the transaction and its transform are written to a log and the short term locks are released. The data in the database is not changed at this point. The next transaction then follows the same process. At commit time, the log is accessed and each transaction again obtains its locks. Its predicate is again tested and if the condition is met, its changes to the database are given effect. Since locks are held for only a short period, the collision cross section is greatly reduced. The basic operation of field calls/escrow locking is shown in Figures 2.2 and 2.3 below.

In Figure 2.2 in the first phase, the intended transformation for each transaction is tested against the current state. In this case, assuming there is only one type of item, for each transaction, the transformation is a subtraction from stock (purchase). Thus transactions 1 to 4 reduce stock by 10, 8, 7 and 25 respectively. Since current stock is positive (1000), and the sum of all these transactions is less than 1000, all transformations are written to the log. In Figure 2.3 in the second phase, the transformations recorded in the log are tested against the predicate. If they are successful, the transaction is committed and the state of the database is changed. Thus for example, current stock minus transaction 1 is greater than 0 and thus transaction 1 is committed. The next transaction is then tested against the new state.

While according to [22] field calls/escrow reads have been successful, there is no published data to confirm this. Indeed other than the works listed in this section there seem to be no published work on field calls/escrow reads. As well, they seem to have certain problems, which would seem to limit their applicability. These include –

1. Transactions cannot see the value of the data that they are accessing.
2. Field calls and escrow calls find cannot operate where there is a requirement for sequence numbers to increase monotonically.
3. It is not always possible to set a valid condition that could be tested by a predicate.

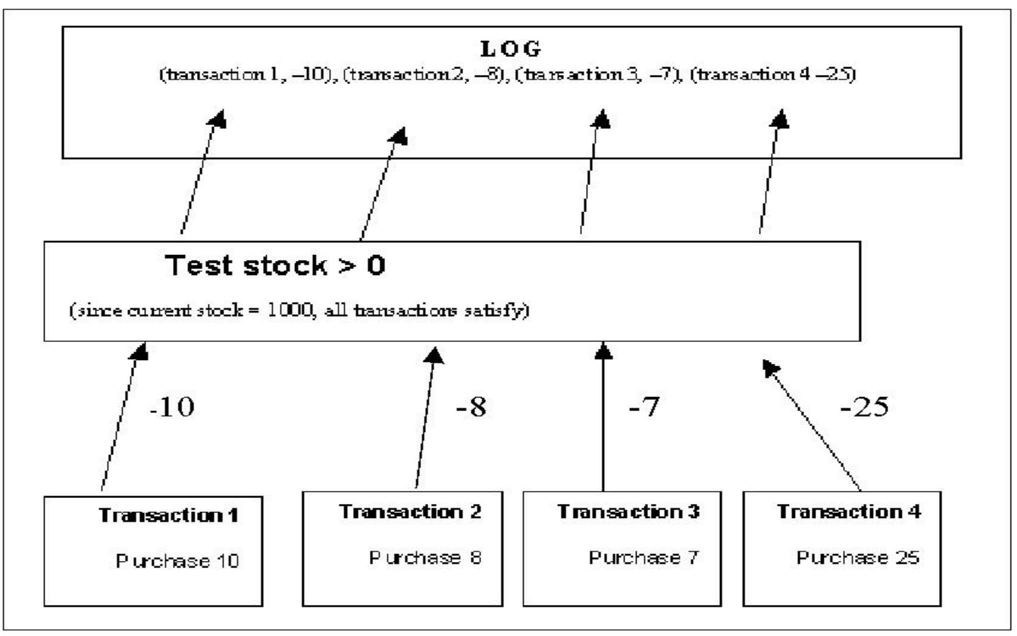



**Figure 2.2. First phase of field calls and escrow locks.**

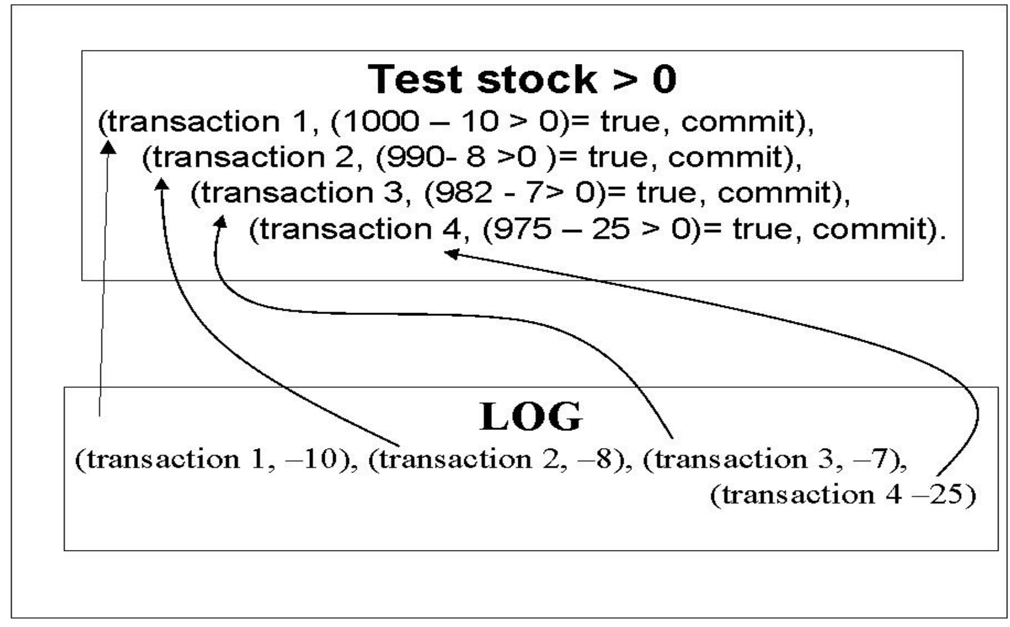



**Figure 2.3. Second phase of field calls and escrow locks.**

## 2.3 Optimistic Concurrency

Optimistic concurrency [1] is so called because it assumes that conflict will not occur. Thus it allows all transactions to acquire and process their data objects (as against 2PL which because it assumes that transactions will conflict is also known as a “pessimistic” concurrency control system). It is only at committal time that transactions are validated. If a clash has occurred, the offending transaction is aborted and restarted – thus “blocking” under optimistic algorithms is essential blocking since no transaction has to wait for a transaction that is not performing useful work. However, a transaction that is aborted has performed its work needlessly.

From the preceding discussion it would seem to logically follow, that in high contention database working with infinite hardware resources, optimistic algorithms would perform better than 2PL systems since wasted work is of little concern. This intuitive conclusion was formalized analytically in [15]. This work concluded that given infinite resources as concurrency is increased, under essential blocking policies such as optimistic algorithms, the effective level of concurrency increases indefinitely at $O(\log (n))$. Figure 2.4 below, taken from [15], shows the theoretical upper limit on the number of active transactions given different values of $p$ (contention), varying levels of concurrency, optimistic concurrency control and infinite resources.

[1] Timestamp algorithms work along a similar principle and according to [22: pp 235] are a “degenerate form of optimistic concurrency”.

As can be seen in Figure 2.4, at all levels of $p$, throughput increases with concurrency though at an ever-diminishing rate. As would be expected, for any level of concurrency, throughput is lower at high values of $p$ than at lower values of $p$. However, a comparison with Figure 2.1 shows that essential blocking is less sensitive to contention – $p$, than 2PL.

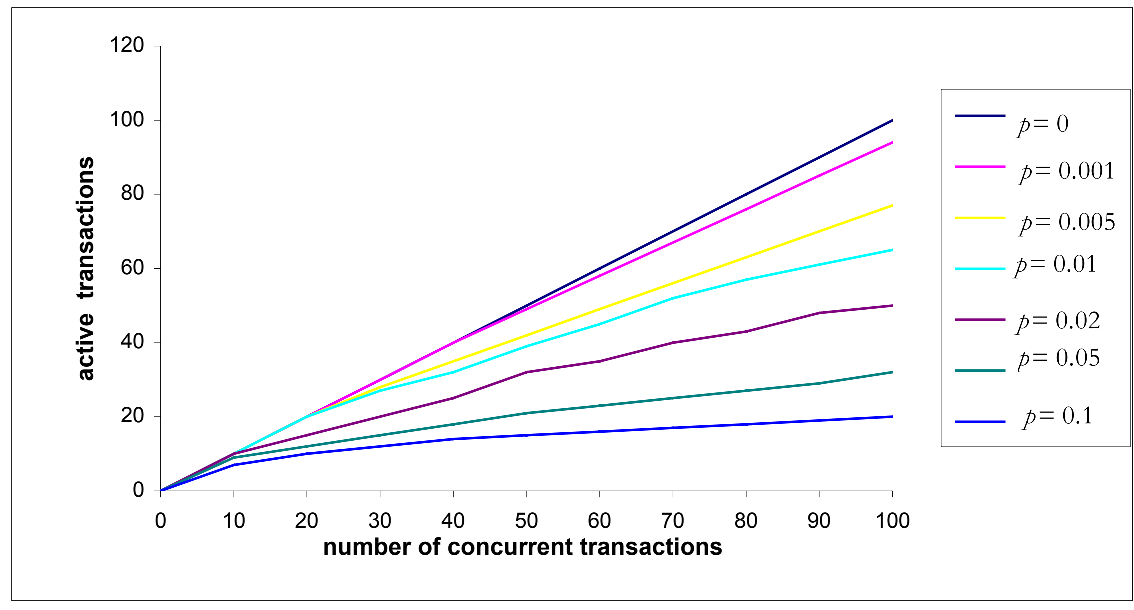



**Figure 2.4. Upper bounds on expected active transactions for essential blocking policies.**

While optimistic concurrency control outperforms 2PL in systems with infinite resources, in systems with finite resources, at some resource level below infinity, the cost of wasted work exceeds the benefits of essential blocking. Recent work to overcome this problem of wasted work includes [51]. Here a hybrid optimistic/2PL system is proposed that ensures that transactions that are blocked in their optimistic phase are guaranteed success in their next execution by a pre-claiming of locks.

## 2.4 Running Priority and Wait Depth Limited

Running priority/wait depth limited (WDL) are a novel set of algorithms that attempt to minimize the length of wait chains by aborting non-active transactions that are blocking other transactions. The basis of these algorithms is as follows - Given transactions $T_1$, $T_2$, if transaction $T_2$ is blocked by transaction $T_1$, if $T_1$ is performing useful work and no transaction is being blocked by $T_2$, then $T_2$ waits. If however, $T_1$ is itself waiting for another transaction (say $T_0$), then it is aborted and restarted at some later stage and $T_2$ starts performing useful work immediately.

While both WDL and optimistic methods remove the problem of wait chains as encountered under 2PL systems, running priority/ WDL have an advantage over optimistic systems in that not all conflicting transactions are restarted. Thus the amount of wasted work under running priority/WDL is less than under optimistic systems. However, as against this, running priority/WDL does not have the benefits of essential blocking.

## 2.5 Performance Comparisons

As indicated in the previous section, given infinite resources, essential blocking methods should outperform 2PL while at some more limited resource levels, 2PL should outperform optimistic methods.

Simulations conducted by [2] confirm this deduction. [2] show that for low contention environments, all concurrency control algorithms perform reasonably well. However in high contention environments, the performance of different algorithms becomes resource dependent. With low resource availability (1 CPU and 2 disks)[2], 2PL outperforms optimistic algorithms. As would be expected from the preceding discussion as the level of resource availability increases, the relative performance of optimistic algorithms improves and at 25 CPU's and 50 disks, exceeds that of 2PL.

[17] and [50] tested several variations of running priority and WDL against both optimistic and 2PL algorithms in a high contention environment. The tests in [17] were conducted for systems containing the following configurations – 4 processors each operating at 50 million instructions per second (MIPS) , 4 processors each operating at100 MIPS, 4 processors each operating at 200 MIPS and 256 processors each operating at 4 MIPS.

In all these tests, all the other algorithms outperformed 2PL. WDL performed best in all tests except for those where the configuration was four processors operating at 200 MIPS – here, the optimistic algorithms performed best (since wasted work was of relatively little importance).

What is of interest besides the results, are the parameters used. At the time that [17] was written, the level of hardware performance required for optimistic methods to

---

[2] In this study, it was assumed that each transaction took 500 milliseconds to execute. That is the speed of the CPU was fixed but the number of CPU's was varied.

outperform 2PL was available but rare. Today, this level of hardware capability is considered passé. Despite this, there are very few "real" systems operating under anything other than a 2PL algorithm.

## 2.6 Further Reading

For a general overview and analysis of many of the issues discussed in the previous sections, [4] is considered a seminal text. [22] is a more recent general text. However, despite its rigorous treatment of most areas of transaction processing, [22]'s treatment of optimistic concurrency schemes is extremely brief and it is clear from their comments that the authors are not well disposed to these schemes.

A more recent brief comparison of 2PL, time-stamp and optimistic control schemes is given in [5].

As indicated above, [22] is a good general text that provides a good overview of nearly all aspects of transaction processing including some areas not discussed here such as logging and recovery which is also dealt with in [7], [8], [11] and check-pointing in optimistic systems which is also dealt with in [38] and [49].

With regard to optimistic schemes, for historical interest, one can read early work on optimistic concurrency in [33], [37] and [25]. A more recent work in the area is [1].

Locking schemes are evaluated in [47] and [48].

As indicated earlier, [2] compares the performance of various concurrency control schemes under various conditions. As well, it presents a rigorous analysis of the principles of building correct models for simulating transaction processing.

# *Chapter 3*

## *An Analysis of the Behavior of Concurrency Control Mechanisms*

## 3.1 Introduction

In this chapter we examine the behavior of three concurrency control mechanisms and their performance under diverse hardware configurations. The concurrency control mechanisms discussed are two phase locking (2PL), wait depth limited (WDL) and optimistic concurrency control. While these concurrency control mechanisms have been around for a while (though only 2PL is commercially used) and their behavior has been well documented we discuss these mechanisms for two reasons. Firstly because, while the performance of these mechanisms has been tested in the context of disk-based systems, the recent advent of in-memory systems makes it important to re-investigate the relative performance of diverse hardware/concurrency control mechanism combinations.

Secondly, as a framework for some of the algorithms presented in later chapters aimed at improving the performance of each of these concurrency control mechanisms.

The hardware configurations used in this chapter fall into two basic categories - disk-based systems and in-memory (or main memory) systems. In the latter, as the name implies, all data is held in main memory and thus transactions do not access disks.

## 3.2 An Overview of 2PL

We begin our discussion of 2PL by considering systems whose transactions are uniform in both size and contention. For simplicity, we also assume that for any permitted level of concurrency, there are always sufficient new transactions in the arrival queue to fully satisfy the system's capacity for processing new transactions.

At time 0, the system is empty. Thus, the number of new transactions that can be allowed into the system is equal to the permitted concurrency level, $n$, of the system. With this entry of $n$ transactions into the system, the system's first cycle has begun. In this cycle, each transaction attempts to acquire locks for all of its objects. If a transaction is successful in obtaining all its required locks, it proceeds to completion. We refer to these transactions as active transactions. If a transaction is blocked, it ceases execution at the point that it is blocked (goes to sleep).

This is illustrated in Figure 3.1 below. In Figure 3.1, transaction $T_2$ is blocked by transaction $T_1$ for object $O_2$ and goes to sleep. It does not, at this stage, attempt to obtain locks for objects $O_6$ and $O_7$.

When active transactions complete, they release their locks and exit the system. At this point, the start of the second cycle, new transactions are allowed into the system to replace the completed transactions. Completing transactions release their locks. As a result, transactions that are no longer blocked are woken well. Woken transactions resume acquiring locks from the point at which they were blocked. As in the first cycle, transactions that are acquiring all locks are active transactions whilst those that are blocked go to sleep at the point that they are blocked. This is illustrated in Figure 3.2 below.

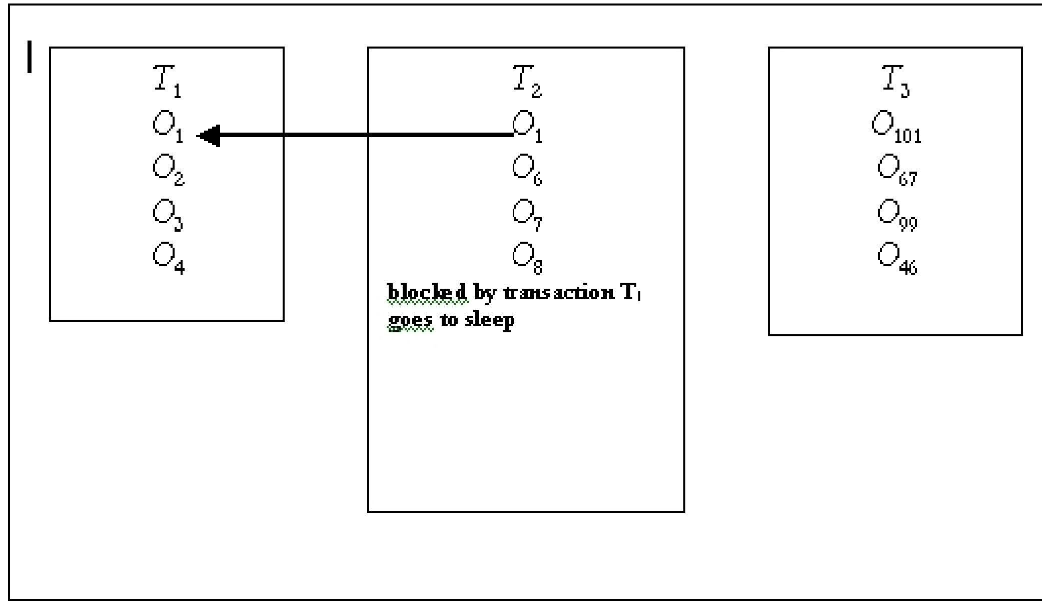



**Figure 3.1. An example of transaction activity in the first cycle**

In Figure 3.2, transactions $T_1$ and $T_3$ as illustrated in Figure 3.1 complete and are replaced by transactions $T_4$ and $T_5$. Transaction $T_2$ is woken up by the completion of transaction $T_1$. All transactions attempt to acquire locks for their required objects with transaction $T_2$ resuming from the point where it was blocked. In this diagram, transactions $T_4$ and $T_2$ succeed in obtaining all their locks while transaction $T_5$ is blocked by transaction $T_2$ for object $O_7$ and goes to sleep. Subsequent cycles follow the same process as outlined for the second cycle.

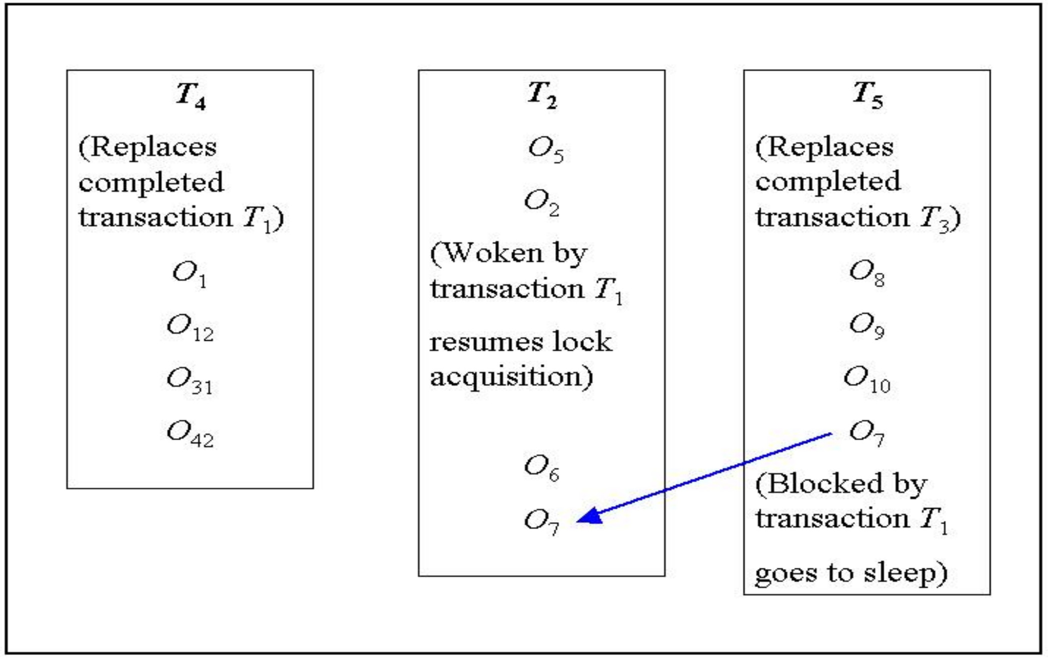



**Figure 3.2. An example of transaction activity in the second cycle.**

From the preceding it follows that apart from hardware considerations, the throughput of 2PL systems depends on how many transactions are successful in obtaining all their locks. This in turn, depends on the probability of conflict between transactions –

or, as it is commonly known, contention and the length of time that transactions are allowed to remain blocked before they are timed out (that is, the cyclical behavior of transactions). The factors governing contention are –

1. The size of the database – the smaller the database, the greater the likelihood of conflict between transactions requesting objects from this database. This extends to hotspots –that is, subsets of the database where objects belonging to these subsets are more commonly accessed than other objects.

2. The number of objects required by transactions – the greater the number of objects required by transactions, the greater the probability of conflict between transactions. For systems with variable size transactions this implies a difference in the behavior of transactions since smaller lower contention transactions have a higher probability of success than higher contention transactions. The effect of variability in transactions' contention and size on the behavior of 2PL systems is investigated in more detail in the next chapter and in [47].

3. The number of transactions in the system (concurrency) either processing or blocked. For any level of contention above 0, the higher the concurrency, the greater the probability of conflict. Thus, as concurrency increases, while the absolute number of successful transactions may increase, the success rate of transactions decreases and eventually becomes negative – that is for any contention level above 0, any increase in concurrency eventually leads to a

decrease in throughput. The level of concurrency at which this downturn occurs depends on contention.

### 3.2.1 Other Behavioral Properties of 2PL Concurrency Control

According to [15] an implication of the cyclical behavior of transactions in 2PL systems is that unless some preventative action is taken, throughput will decrease in every cycle and will eventually reach 0. This is because in every cycle the proportion of transactions in the wait chain grows and the proportion of transactions woken by completing transactions decreases. That is, the proportion of transactions that are blocked by transactions that are themselves in the wait chain grows with every successive cycle – such transactions are not woken by the completion of active transactions. According to [47] this very rarely occurs unless the degree of lock contention is permanently greater than 0.226 in systems with fixed size transactions. In systems with variable size transactions where the mean lock contention is below 0.226, the random arrival of large transactions may temporarily lead to a lock contention greater than 0.226 thus destabilizing the system and leading to thrashing.

It is well known that 2PL concurrency control systems are susceptible to the problem of deadlock, a state that occurs when two transactions block each other and where consequently, neither transaction can be woken. As shown in [48], the incidence of deadlock is quite small and is proportional to the degree of concurrency, the fourth power of the number of requested locks and the inverse of the second power of the size of the

database. Since it occurs so infrequently, the cost of wasted work caused by the breaking of deadlocks is insignificant.

As indicated above, in 2PL systems, for any contention level above 0, there are constraints limiting the number of transactions which can be processed in a given time period by either increasing concurrency or permitting the system to proceed cyclically without interference. Here, we propose that in 2PL systems the concurrency constraint is more restrictive than the cyclical constraint. That is, we propose that for any equivalent total processing capacity, a greater throughput is achieved by increasing the number of cycles that can be completed in a given time than by increasing the number of concurrent transactions processing in the same period of time.

We illustrate this proposition using the equation found in [15] to predict transaction throughput, that is, throughput is approximated by

$$n(1-p/2)^{(n-1)} \tag{3.1}$$

Where $n$ is the concurrency level and $p$ is the level of contention. For simplicity we assume uniform transactions and that all transactions have an equal probability of being blocked. Thus, given two transactions $T_1$, which is active, and $T_2$, which is blocked, a new transaction arriving at the system has an equal probability of being blocked by either $T_1$ or

$T_2$. As vehicles for our illustration we use 3 in-memory hardware systems with the following configurations -

1. A system with 20 CPUs each operating at 20 MIPS.
2. A system with 40 CPUs each operating at 10 MIPS.
3. A system with 60 CPUs each operating at 6.667 MIPS.

It will be noted that each of the systems has the same total processing power – 400 MIPS. We assume that each system operates at its maximum concurrency, which given that all systems are in-memory, is equal to its number of processors in each system. It will also be noted that given that in the first system each processor is twice as fast as in the second system and three times as fast as in the third system. Thus, the first system goes through two cycles in the time the second system goes through 1 cycle and 3 cycles in the time the third cycle goes through 1 cycle.

Using the equation presented above, the predicted throughput for the first system in the first cycle is 17.17 that of the second system in the first cycle is 29.24, and that of the third system in the first cycle is 37.35. Because the first system is twice as fast as the second system it completes two cycles in the time the second system completes 1. Since in the first system 17.17 transactions completed in the first cycle, 17.17 new transactions enter the system and the number of released transactions that attempt to obtain their locks is (20 -17.17)(17.7/20)=2.43. That is, given that each transaction has an equal probability of blocking another transaction, the number of potentially released blocked transactions is the total number of blocked transactions, which given a concurrency of 20

is (20-17.17), times the proportion of blocked transactions blocked by completing transactions (17.17/20). Thus, in the first system in the second cycle, 19.6 transactions attempt to acquire their locks and 16.83 are successful. Consequently, 33.29 transactions complete in the first system over two cycles at a concurrency of 20 as against 29.24 in the second system over one cycle. By similar calculations, 57.3 transactions complete in the first system over 3 cycles as against 37.35 transactions in the third system over one cycle.

Changing the contention per transaction to 0.001 yields a total predicted throughput in the first system of 39.62 over two cycles and 59.43 over three cycles. The corresponding predicted throughputs in the first cycle for the other two systems are 39.22 and 58.26 for the second and third systems respectively. A contention per transaction of 0.064, yields a total predicted throughput in the first system of 19.27 over two cycles and 26.48 over three cycles. The corresponding predicted throughputs in the first cycle for the other two systems are 11.25 and 8.80 for the second and third systems respectively. Thus, theoretically our proposition holds for all levels of contention greater than 0 but becomes more pronounced as contention increases.

## 3.3 Alternative Concurrency Control Mechanisms

In the previous section we examined the behavior of transactions in 2PL systems and saw that one of its characteristics is the formation of chains of transactions blocked by transactions that are themselves blocked. This characteristic reduces throughput in 2PL

systems and, given a high enough contention, may lead to thrashing in later cycles. This implies that methods that eliminate or reduce wait chains will increase throughput relative to that which can be obtained under 2PL concurrency control. In this section we briefly examine two well know alternatives to 2PL that attempt to increase throughput by eliminating or reducing wait chains. The alternative concurrency controls that we examine are optimistic concurrency control and wait depth limited (WDL).

### *3.3.1 Optimistic Concurrency Control*

Conceptually, optimistic concurrency control is quite simple. Here, transactions acquire all their required objects and process to completion unhindered. It is only on completion, when the transactions modifications require committal, that the transaction is tested. If during its' processing, the transaction has conflicted with a committed transaction it is aborted and restarted. If no such conflict has occurred, then the transaction is committed.

There are two features of this system that improve its performance relative to 2PL. Firstly, because transactions that conflict with committing transactions are aborted and then restarted, there is no chain of transactions waiting behind locked transactions. Secondly is the fact that to be aborted, a transaction must not only conflict with another transaction, the transaction with which it conflicts must be a committed transaction. This property is known as essential blocking and is illustrated in Figure 3.3 below. In Figure 3.3, $T_2$ conflicts with $T_1$ and $T_3$ conflicts with $T_2$**.** Because $T_1$ completes first it has no

clash with a committed transaction and thus commits. When $T_2$ is tested, it fails because it conflicts with a committed transaction. Thus, $T_2$ is thus aborted. This allows $T_3$, which under 2PL would have been blocked, to continue to committal unhindered.

The benefit of essential blocking is that it does not impede the completion of transactions unnecessarily. Consequently, the number of successful transactions is unbounded and increases logarithmically with concurrency. In [15] an upper and lower bound is provided for the number of successful transactions at each concurrency level. Where p is the level of contention and n the level of concurrency, these are defined by the two equations. The lower bound is defined as –

$$1 + \frac{1-p}{p} ln( p(n-1)+1 ) \tag{3.2}$$

while the upper bound is defined as –

$$1 + \frac{1}{p} ln( p(n-1)+1 ) \tag{3.3}$$

An interesting consequence of equations 3.2 and 3.3, is that while they allow higher throughputs at higher concurrencies than does 2PL, they imply that as for 2PL, for any equivalent total processing capacity, a greater throughput is achieved by increasing the

number of cycles that can be completed in a given time than by increasing the number of concurrent transactions processing in the same period of time. That is, using equations 3.2 and 3.3, for any level of contention and concurrency, throughput will be the same in every cycle. Thus, in an in-memory system, doubling processor speeds doubles the number of cycles completed in a given time period and thus doubles throughput. However, doubling concurrency only increases throughput logarithmically.

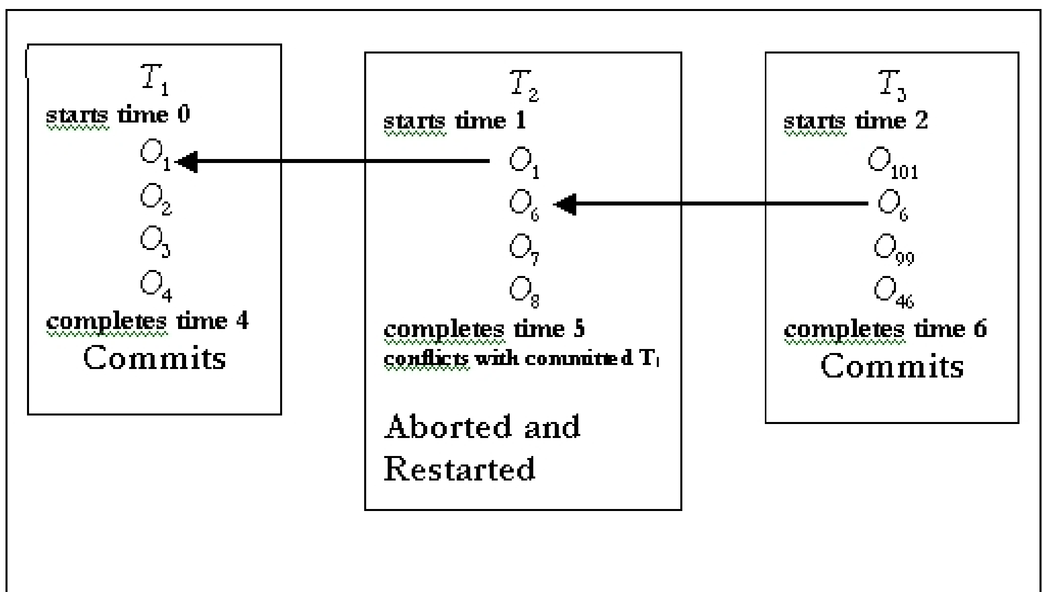



**Figure 3.3. An example of essential blocking**

There are two ways with which optimistic concurrency control can be implemented. These are the kill method and the die method. In the kill method, a successful transaction checks to see if there is any conflict with a processing transaction, and, if so, kills it. In the example shown in Figure 3.3, $T_1$ would kill $T_2$. In the die method, a completing transaction checks if it has conflicted with a committed transaction, and, if so, commits suicide. In the

example shown in Figure 3.3, $T_2$ would find that it has conflicted with committed $T_1$ and would commit suicide.

#### 3.3.1.1 Variability of Transaction Size and Contention and Performance

An implicit assumption of the description presented in the previous section is that all optimistic systems behave uniformly. That is, all systems with the same mean contention *p*, level of concurrency *n* and hardware capability, will have the same throughput. However, both variability in transactions size and contention have a significant impact on system performance.

Contention determines the probability that a transaction will encounter conflict with another transaction. Thus, the lower a transaction's contention the lower the likelihood that it will encounter conflict and that it will either block or be blocked by another transaction.

A transaction's size determines how much processing time a transaction requires. The smaller a transaction the less processing time it requires and the greater its probability of completing successfully. This is because where two conflicting transactions start at the same time, the smaller one will finish first and get the first opportunity to commit and kill the larger transaction. Alternatively, in a die system, the smaller transaction will finish first and commit, thus, when the larger transaction completes, upon testing it will be forced to commit suicide.

While it is generally considered that within a system variability in transactions contention is proportional to their size, in databases with hotspots this is not always so.

Further, even where transactions contention varies proportionally with their size, as indicated above, size and contention have distinct influences on performance.

### *3.3.2 Wait Depth Limited (WDL)*

Running priority/wait depth limited (WDL) are a novel set of algorithms that attempt to minimize the length of wait chains by aborting non-active transactions that are blocking other transactions. The basis of these algorithms is as follows - Given transactions $T_1$, $T_2$, if transaction $T_2$ is blocked by transaction $T_1$, if $T_1$ is performing useful work and no transaction is being blocked by $T_2$, then $T_2$ waits. If however, $T_1$ is itself waiting for another transaction (say $T_0$), then it is aborted and restarted at some later stage and $T_2$ starts performing useful work immediately.

A more precise description of WDL (with a wait depth of 1), taken from [17] is shown in Figures 3.4 and 3.5 below. In these Figures, given transactions $T^i$ and $T^j$, $T_0^i$ $T_1^i$ $\cdots T_n^i$, represent transactions waiting on $T^i$, while $T_0^j$ $T_1^j$ $\cdots T_m^j$, represent transactions waiting on $T^j$. $L$ is a length function – that is given two transactions, the transaction whose L value is greater is the longer transaction. In Case a, both transactions $T^i$ and $T^j$ are active. In Case b $T^j$ waits on $T^i$ and in Case c, $T^j$ waits on $T_0^i$, which in turn, is waiting on $T^i$.

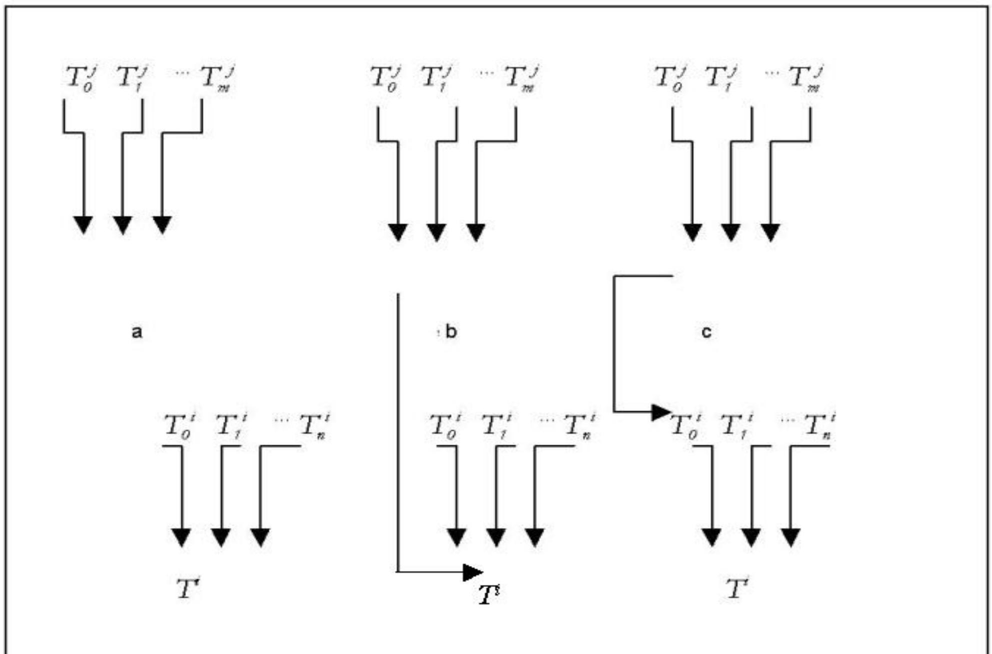



**Figure 3.4. The three initial (temporary) cases of transaction states.**

As indicated by Figure 3.5, under WDL, longer transactions are given priority over shorter transactions, and, herein lays the major difference between WDL with a wait depth of 1 and running priority. This preference given to larger transactions reduces the quadratic effect that is, the tendency of longer transactions to suffer a disproportional rate of restarts relative to shorter transactions. As indicated in the previous section, this problem also occurs in optimistic systems.

Thus, under WDL, larger transactions do not suffer the disadvantages that they face in optimistic and running priority systems. This tends to reduce the problem of wasted work since large transactions have in general performed more work when aborted than have smaller transactions. As well, when compared to optimistic systems, since not all blocked transactions are restarted, one would expect less wasted work under both running priority and WDL than under optimistic systems.

**Case a:**

Both transactions $T^i$ and $T^j$ are active. Since under WDL(1) no wait depth of greater than one is allowed, no transaction in the set $T_0^i\ T_1^i\ \ldots T_n^i$ or in the set $T_0^j\ T_1^j\ \ldots T_{n1}^i$ can have a transaction waiting for them. In this case transactions $T_0^i\ T_1^i \ldots\ T_n^i$ wait for $T^i$ to complete while $T_0^j\ T_1^j \ldots\ T_{n1}^i$ wait for $T^j$ to complete.

**Case b:**

1. $m=0$: $T^j$ waits since the wait tree is 1.
2. $m>0$: restart $T^j$ unless $L(T^j) \geq L(T^i)$ and for each $k$, $L(T^j) \geq L(T_k^j)$. In this case, restart $T^i$.

**Case c:**

1. $m=0$: restart $T^j$ unless $L(T^j) \geq L(T^i)$ and $L(T_0^i) \geq L(T^j)$. In this case restart $T^i$.
2. $m>0$: restart $T^j$ unless $L(T^j) \geq L(T_k^j)$ and for each $k$, $L(T^j) \geq L(T_k^j)$. In this case, restart $T^j$.

**Figure 3.5. WDL(1) algorithm given the states outlined in Figure 3.2**

### *3.3.3 Access Invariance and Wasted Work in Disk-based Systems*

The cost of wasted work is largely dependant on the speed at which the work can be done. An important factor in determining speed is the speed of the processor, however far more crucial in disk-based systems is the amount of disk I/O that has to be redone as a result of starting a transaction. Thus for example let us assume two disk -based systems each with an equal number of processors with each processor operating at 200 MIPS and with disk access times of 15 milliseconds per access. In the first system data retrieved by aborted transactions is kept in cache. In the second system all objects whether required by new or restarted transactions require disk access. Let us further assume uniform transactions requiring 16 objects with each object requiring memory 20000 instructions to

process. Let us now say that on both systems 2 identical transactions have to be restarted after 8 objects have been acquired.

In the first system, since objects previously fetched by restarting transactions only require memory access, the total work lost is (8 objects x 20000 instructions per object)/200MIPS= 0.0008 seconds. In the second system, since all objects have to be re-accessed from disk, the total work lost is (8 objects x 20000 instructions per object)/200MIPS plus (8 objects x 15 milliseconds per access)= 0.0008 + 0.12= 0.1208 seconds. Thus, despite having the same processor speed, the cost of work lost in the second system is 151 times that lost in the first system.

The assumption that makes pre-fetching viable, is that the data pre-fetched by a transaction will be the same data that the transaction requires when restarted. For this assumption to hold, the following conditions must hold, either the data required by a restarted transaction has not been modified by a conflicting transaction or, where the data required by a restarted transaction has been modified by a conflicting transaction the modification has not been sufficient to invalidate the predicate of the restarted transaction.

The assumption outlined above together with its conditions is outlined in [16] and [17]. The term coined for this assumption in these works is access invariance. These works maintain that for limited levels of concurrency, the conditions required for access concurrency hold but that as concurrency is increased, the degree of access invariance decreases. No measure is given correlating concurrency levels and the degree of access

invariance. Accordingly, in this chapter, we will assume arbitrarily, that access invariance holds for concurrency levels up to 100 and then ceases to hold. In chapter 5 we will examine how pre-fetching can be extended to much higher levels of concurrency.

## 3.4 Evaluation Systems

Each of the systems in this chapter is composed of the following subsystems - a concurrency control subsystem, a database subsystem, a hardware subsystem a processing subsystem and a transaction subsystem. The programs for conducting these simulations were constructed by the author as outlined in Appendix A. The specification of these subsystems is outlined in the remainder of this section.

### *3.4.1 The Database and Transaction Subsystems*

There are two object stores available to transactions from which to choose their objects -$D_1$, which contains 1000 objects, and $D_2$, which contains a million objects. Characteristics of objects in these data stores are –

1. In each data store each object is unique.
2. Data stores are disjoint.
3. Given a data store $D$, any two objects $O_i$ and $O_j$ in that data store have an equal probability of being required in any access. Thus for $D_1$ which contains

1000 objects, each object in $D_1$ has a 1/1000 chance of being required in an access to $D_1$. Similarly, any object in $D_2$, which contains a million objects, has a 1/1000000 chance of being required in an access to $D_2$.

Each of our systems System contains four transaction types or classes - $T_1$, $T_2$, $T_3$ and $T_4$. These transaction types require 1,2,4 and 8 objects respectively from $D_1$ and 3, 6, 12 and 24 objects respectively from $D_2$. $T_1$ transactions represent $20\%$ of transactions $T_2$ transactions represent $20\%$ of transactions, $T_3$ transactions represent 35% of transactions and $T_4$ transactions represent $25\%$ of transactions.

### 3.4.2 Hardware Subsystems

As indicated earlier in this chapter, the basic division in the hardware subsystems tested in this chapter is between in-memory systems and disk -based systems. The in-memory subsystems used in this chapter are –

1. A subsystem containing 1248 processors each operating at 4MIPS.
2. A subsystem containing 96 processors each operating at 100MIPS.
3. A subsystem containing 96 processors each operating at 200MIPS
4. A subsystem containing 20 processors each operating at 1000 MIPS.
5. A subsystem containing 10 processors each operating at 2000 MIPS.

The subsystems outlined in parts 3, 4 and 5 above have nearly the same total processing power. The total processing power of the third system is 96 processors x 200MIPS = 19200 MIPS, that of the fourth subsystem is 20 processors x 1000 MIPS = 20000 MIPS while that of the fifth subsystem is 10 processors x 2000 MIPS = 20000 MIPS. Besides measuring the performance of systems with very fast processors, given their similar total processing power, systems 3, 4 and 5, allows us to compare the performance of systems with fewer and faster processors as against those with more numerous but slower processors.

The disk-based subsystems used in this chapter are equivalent to the first three systems outlined above - with the exception that in these latter systems, while some data resides in cache, some disk access is required. The cache holds sufficient data to ensure that 0.625 of items required in a first accesses are in memory. For restarted transactions or transactions in process, all items required remain in cache and are not flushed till the transaction commits. These parameters are in accord with those found in [17] and [51].

The number of disks per system is as follows- the 96 processor 200 MIPS per processor subsystem has 15880 disk arms as have the 20 processor 1000 MIPS per processor and 10 processor 2000 MIPS per processor systems. The 96 processor 100 MIPS per processor subsystem has 7940 disk arms while the 1248 processor 4 MIPS per processor subsystem has 3970 disk arms. In all disk-based systems, disk accesses are uniformly distributed (no skew). The determination of this number of disk arms is based on the algorithm used in [17] and [51] which applies Little's law using total CPU

processing power, the number of expected disk accesses and a CPU and disk utilization ratio of 75/20 as parameters.

### *3.4.3 The Concurrency Control Subsystems*

Three basic concurrency control subsystems are used in this chapter these are 2PL, wait depth limited (WDL) and optimistic.

The 2PL subsystem is a standard 2PL system (including a deadlock breaking mechanism) except that for simplification, all requests for locks are requests for a write and the granularity of lock acquisition and release is a single lock.

The WDL subsystem is as described in section 3.2.2. with a depth of 1 and the length cost  as described in [17]. As with the 2PL system, all requests for locks are requests for a write and the granularity of lock acquisition and release is a single lock.

As per section 3.2.3, access invariance and the pre-fetch properties consequent to it are assumed to hold for the disk-based systems whose processors operate at 100, 200, 1000 and 2000 MIPS. Thus, for these disk-based systems, restarted transactions find all the objects that they acquired prior to restarting in cache.  Given the limited concurrency at which these systems are run this is a reasonable assumption. The disk-based system with 1248 processors operating at 4 MIPS is tested both without the access invariance assumption. This is because this subsystem is tested to a concurrency of 1250, and at higher levels of concurrency access invariance is unlikely to hold.

For testing the performance of optimistic systems we use two methods. For the in-memory systems we use a pure kill system since this minimizes the amount of wasted work by killing a transaction immediately. For the disk-based systems we use a hybrid die-kill system as outlined in [17] and [51]. In this system, transactions in their first phase die at the end of their processing cycle if they have conflicted with a previously committed transaction. By dieing after processing transactions are able to pre-fetch all their required objects. Transactions that have died and are restarted are killed if they conflict with a previously committed transaction.

This variation allows transactions to maximize the benefits of pre-fetching while minimizing the cost of wasted work for transactions that have already pre-fetched all their data. This of course assumes that access invariance and its consequent pre-fetch property holds. Again, access invariance is a reasonable assumption for the 100,200, 1000 and 2000 MIPS per processor systems. However, once again, this assumption is unlikely to hold at the higher concurrencies at which the 4 MIPS system is tested. Consequently, this subsystem is tested with a kill system that assumes no access invariance. As with the previous systems, all objects are acquired for a write and the granularity of object acquisition is a single object.

### *3.4.4 The Processing Subsystem*

The processing system used in this chapter has three stages initialization, processing and completion. The initialization stage, whether for new or restarted transactions requires 100000 instructions per transaction regardless of the transaction's

size. This stage requires CPU processing only. Similarly, the completion phase requires 50000 CPU instructions per committing transaction. For disk-based systems, it is assumed that once a committed transaction is placed in a buffer, the system is ready for the next transaction; thus, no disk access is required in this phase. The processing phase requires 20000 CPU instructions per data item, which includes the overheads for concurrency control. For the disk-based systems, 0.375 of objects required by a transaction need disk access (excluding objects pre-fetched by restarting transactions). Each disk access requires 15 milliseconds.

Since we are interested in systems with high throughputs, a large number of transactions need to be available, thus the arrival rate of transactions is 50000 transactions per second arriving at a constant rate. All systems are tested for one second at each concurrency level. At each concurrency level for each system the results shown are an average over 40 runs.

## 3.5 Performance Results

In this section we present and analyze the results of the tests outlined in the previous section. We first look at the in-memory systems, then the disks based systems and then compare the performances of the in-memory and disk-based systems.

### *3.5.1 The Performance of In-Memory Systems*

Figures 3.6 to 3.8 below show the results of the in memory-systems with 1248 processors operating at 4 MIPS per processor, 96 processors operating at 100 MIPS per processor and 96 processors operating at 200 MIPS per processor respectively.

The two most obvious and unsurprising features of these results are –

1. That at concurrencies over 20, optimistic and WDL concurrency control methods outperform 2PL with the peak reached by optimistic and WDL methods exceeding the peak achieved by 2PL by around tenfold in the case of the 4MIPS by 1248 processor system and by around fourfold for the other two systems.

2. That extremely high levels of throughput can be achieved by the in-memory systems with throughput reaching close to a massive 20000 transactions per second in the system with 96 processors operating at 200 MIPS per processor and using either optimistic or WDL concurrency control.

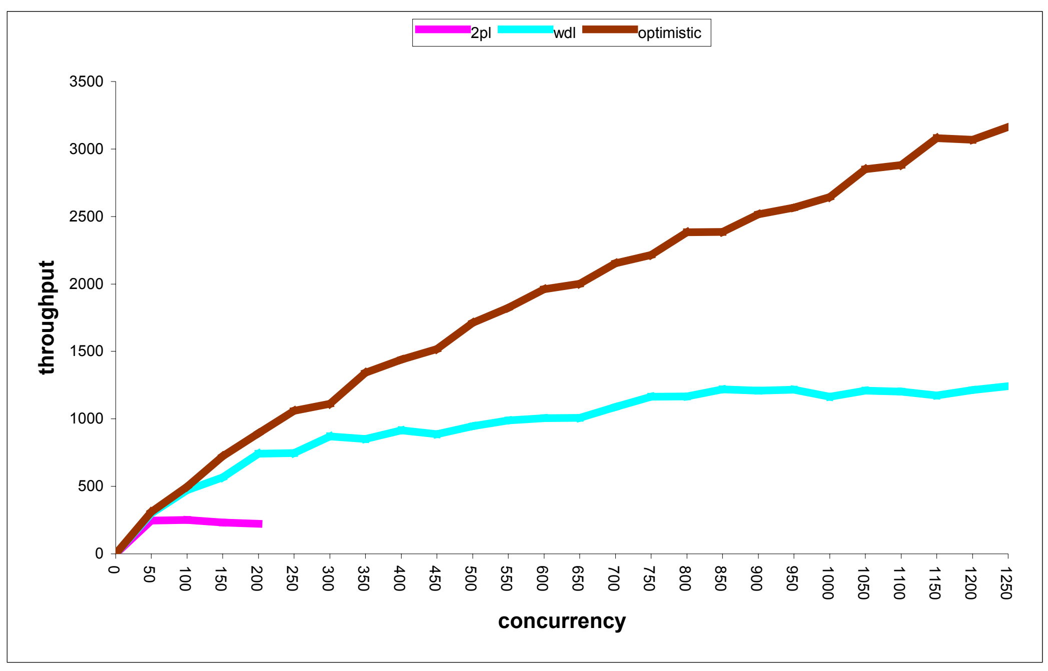



**Figure 3.6. 1248 processors each operating at 4 MIPS**

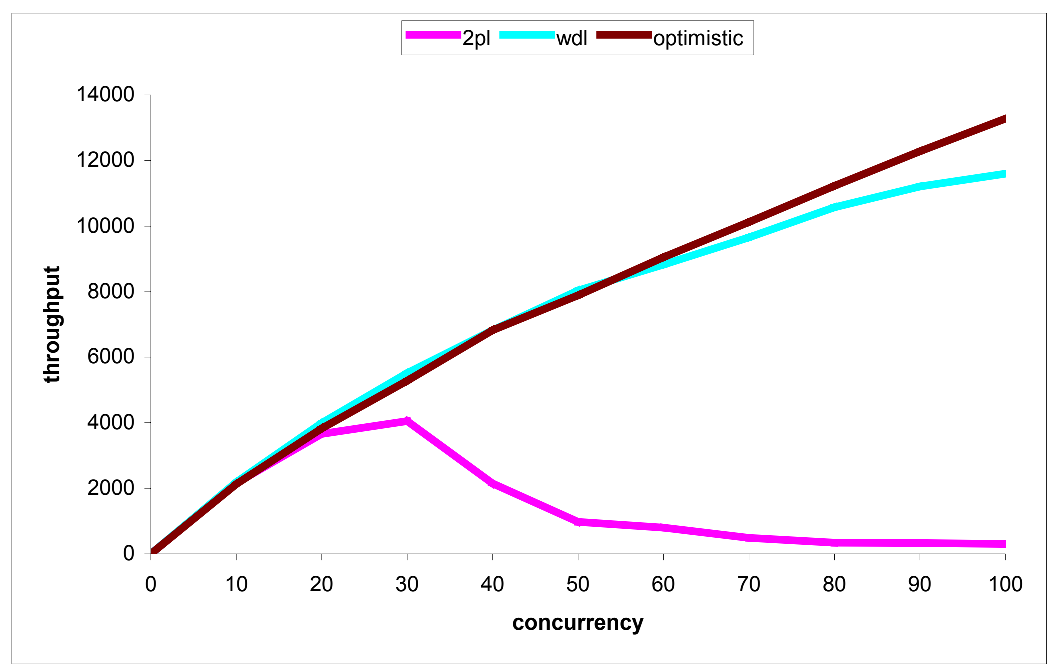



**Figure 3.7. 96 processors each operating at 100 MIPS**

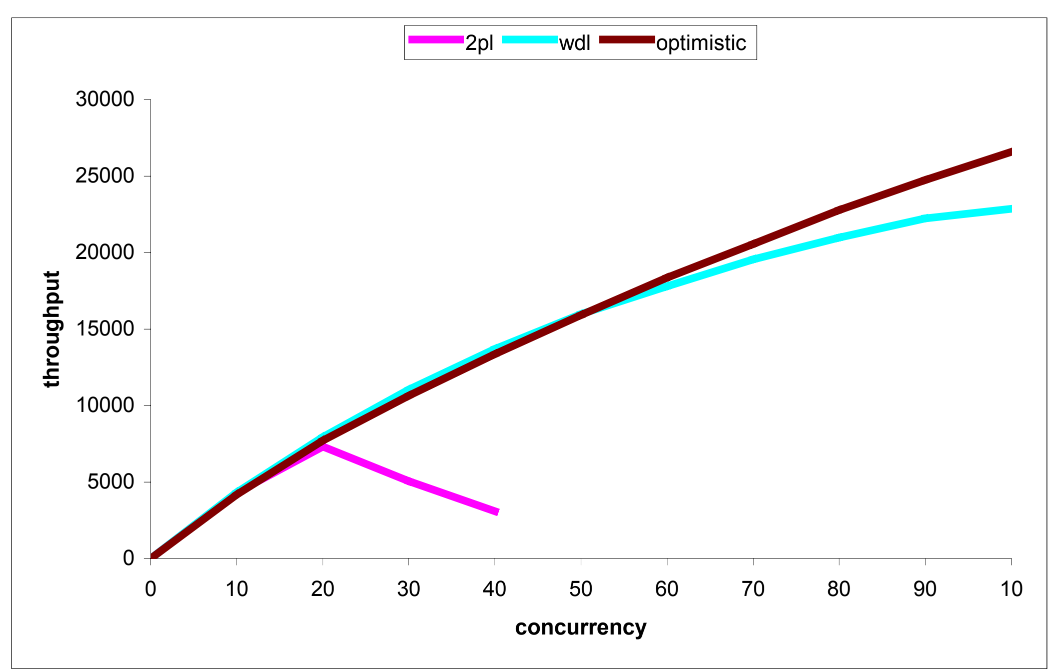



**Figure 3.8. A comparison of throughputs under 2PL, optimistic kill and WDL concurrency control methods for a system with 96 processors each operating at 200 MIPS**

The superior performance of WDL and optimistic concurrency control methods over 2PL at higher concurrencies, particularly in systems with abundant resources, is by now well known. This superiority is due to their elimination of the wait queues that characterize 2PL systems. A benefit of optimistic systems relative to WDL is essential blocking while a cost of optimistic systems relative to WDL is the minimization of wasted work. Given the negligible cost of wasted work in in-memory systems, one would expect that the essential blocking property of optimistic systems would lead it to outperform WDL in in-memory systems. As Figures 3.6 to 3.8 show, with our transaction model, this expectation is confirmed at concurrencies over 50.

One aspect of the relative performance of optimistic and WDL concurrency control systems that is not shown in Figures 3.6 to 3.8 is their performance vis a vis the throughput of higher contention transactions. As indicated earlier in this chapter, in optimistic systems, one would expect the throughput of larger transactions to suffer since these transactions have a disproportionate tendency to be killed by smaller transactions. In WDL systems this problem is overcome by giving larger transactions priority over smaller transactions in clashes between non-active transactions. Thus, one would expect WDL to outperform optimistic systems in the throughput of larger transactions.

Figures 3.9 to 3.11 below break down the results shown in Figures 3.6 to 3.8 showing the performance of our systems in the throughput of the largest transaction types in our systems (transactions belonging to the class $T_4$). As Figures 3.9 to 3.11 show, the expectation that WDL should outperform optimistic concurrency control in the throughput of the largest/highest contention transactions is not fulfilled. Indeed, at concurrencies of around 50 and over, optimistic concurrency control outperforms WDL in the throughput of the largest/highest contention transactions.

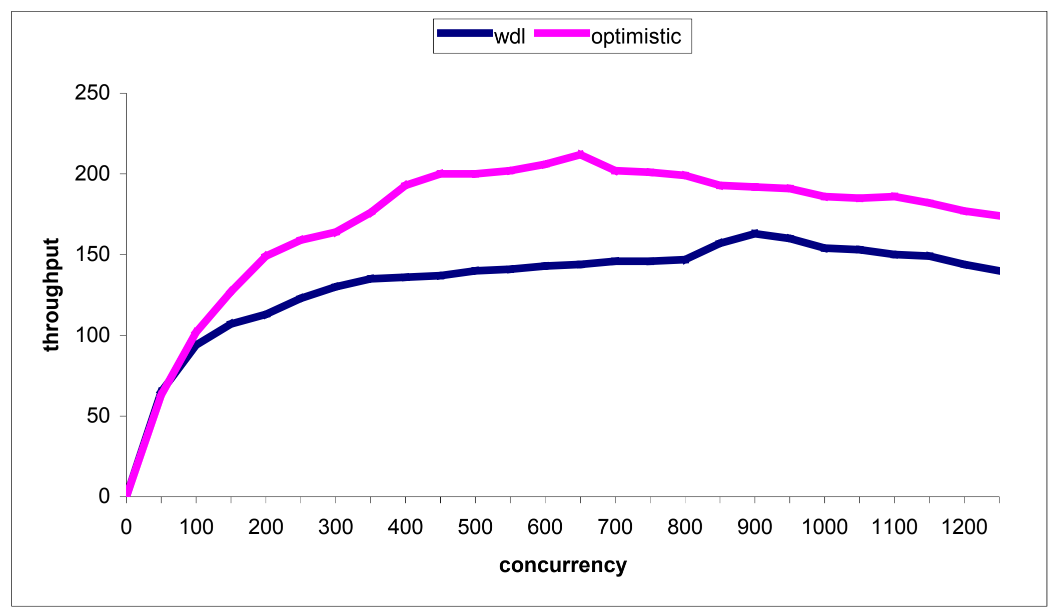



**Figure 3.9. A comparison of throughputs of the largest transactions ($T_4$) under optimistic kill and WDL concurrency control methods for a system with 1248 processors each operating at 4 MIPS**

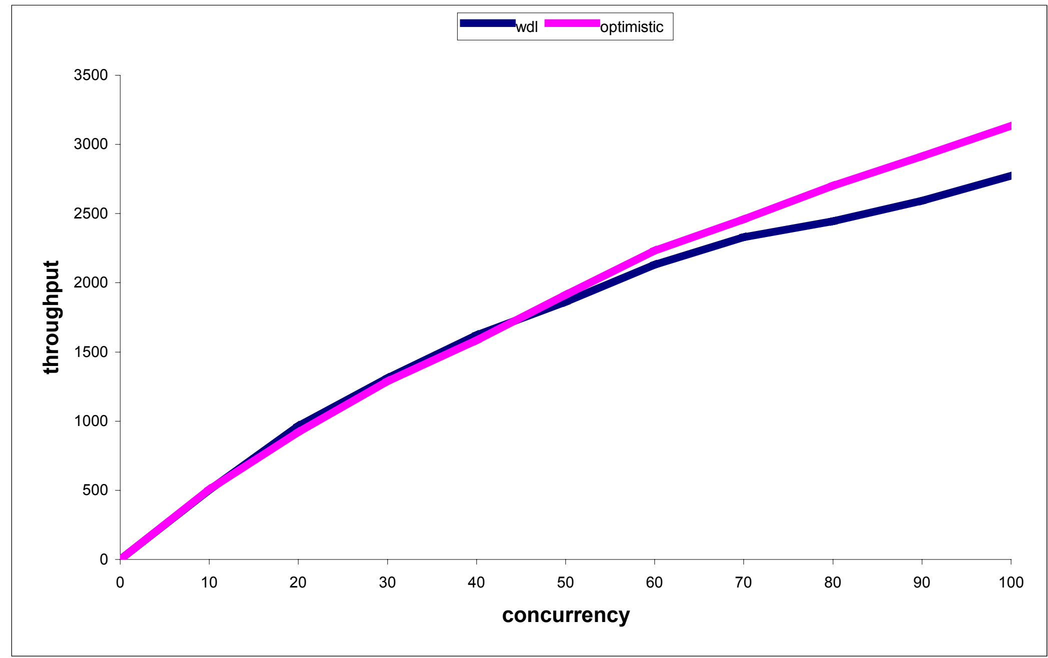



**Figure 3.10. A comparison of throughputs of the largest transactions ($T_4$) under optimistic kill and WDL concurrency control methods for a system with 96 processors each operating at 100 MIPS**

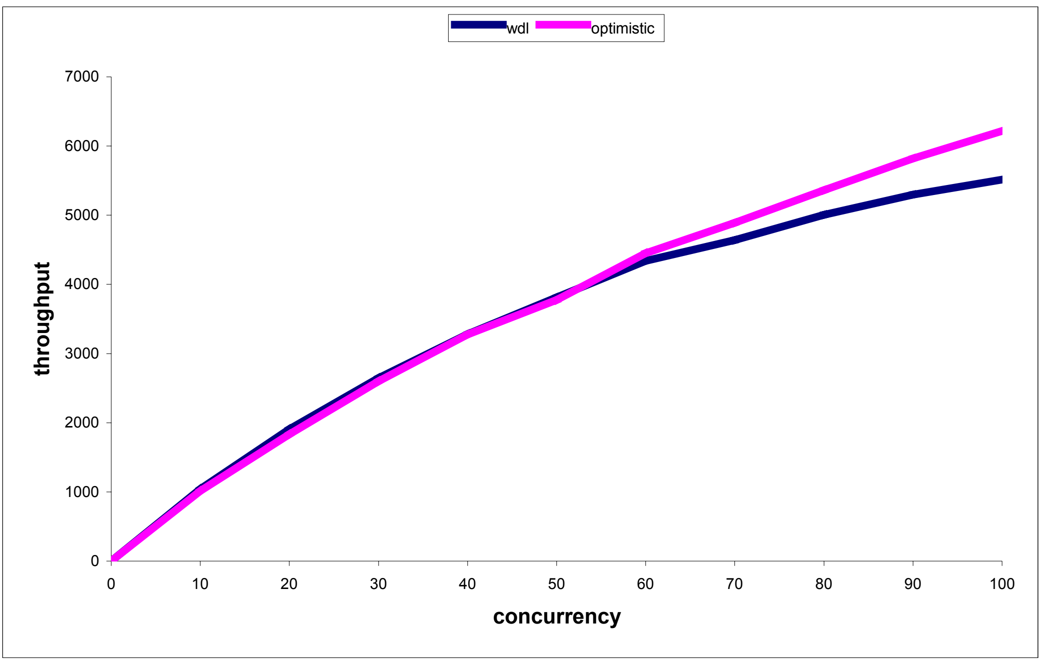



**Fig 3.11: A comparison of throughputs of the largest transactions ($T_4$) under optimistic kill and WDL concurrency control methods for a system with 96 processors each operating at 200 MIPS**

### *3.5.1.1 The Performance of In-Memory Systems with Very Fast Processors*

To validate the proposals outlined in section 3.2 and 3.3.1 and to examine the performance of very fast in-memory systems, we present Figures 3.12 to 3.14 below. In these graphs the total processing power of each of the systems is about the same (between 19200 and 20000 MIPS). However one system reaches this total with 96 processors each operating at 200 MIPS, another with 20 processors each operating at a 1000 MIPS and the third with 10 processors each operating at 2000 MIPS. Figure 3.12 compares the performance of these systems under 2PL, Figure 3.13 compares the performance of these systems under WDL and Figure 3.14 compares the performance of these systems under optimistic concurrency control.

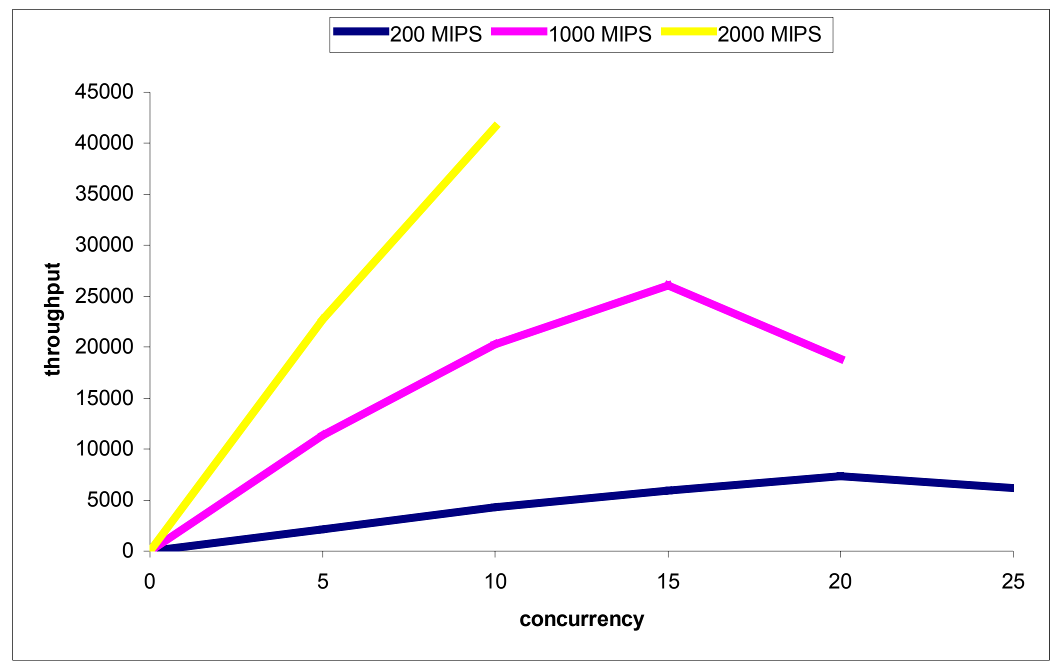



**Fig 3.12. A comparison of throughput under 2PL concurrency control in systems composed of 96 processors each operating at 200 MIPS, 20 processors each operating at 1000 MIPS and 10 processors each operating at 2000 MIPS**

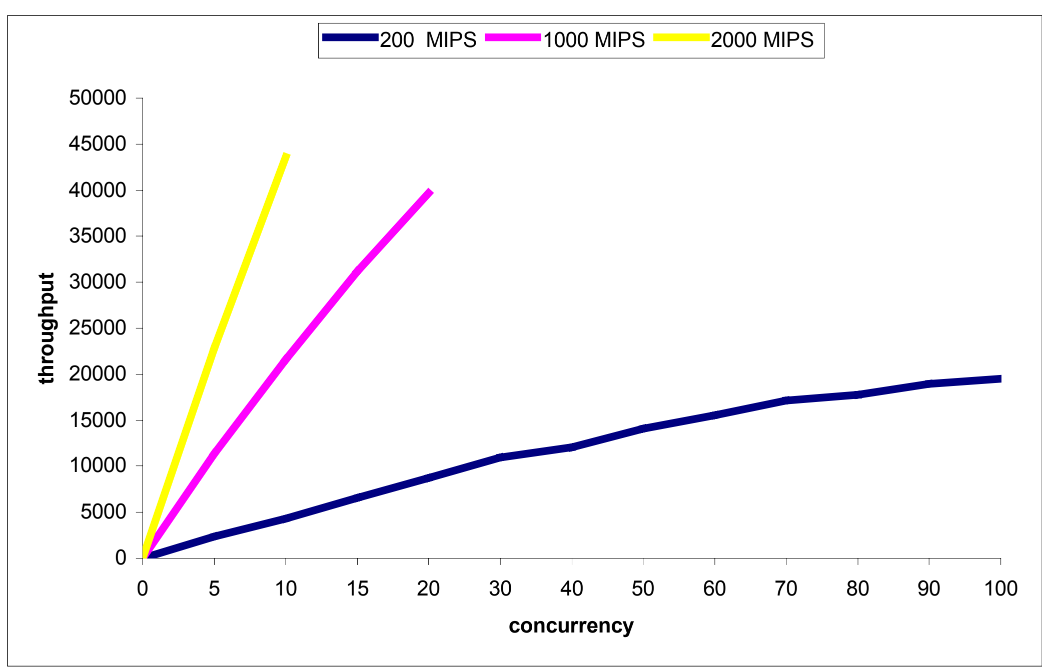



**Figure 3.13. A comparison of throughput under WDL concurrency control in systems composed of 96 processors each operating at 200 MIPS, 20 processors each operating at 1000 MIPS and 10processors each operating at 2000 MIPS.**

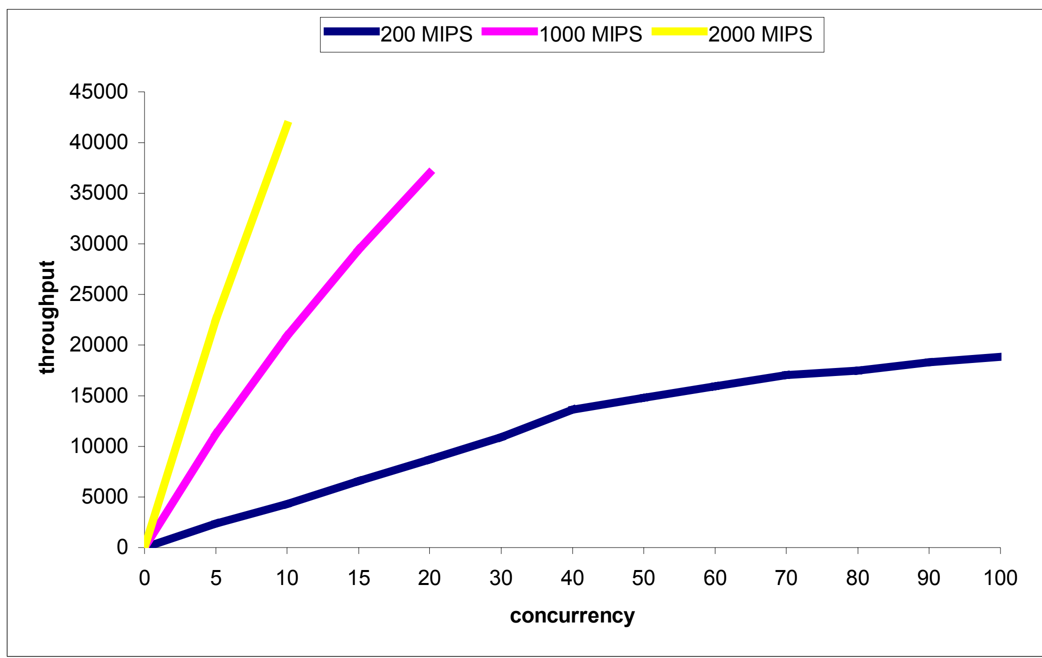



**Figure 3.14. A comparison of throughput under optimistic concurrency control in systems composed of 96 processors each operating at 200 MIPS, 20 processors each operating at 1000 MIPS and 10 processors each operating at 2000 MIPS.**

Apart from the massive throughputs achievable by in-memory systems with few but very fast processors, these graphs show that under each of the concurrency control mechanisms, although the total processing power of each of the systems is the same, throughput is highest under the fastest system operating at 2000 MIPS per CPU at low concurrencies and is lowest under the slowest system operating at 200 MIPS per CPU at relatively high concurrencies. This tends to confirm the proposition made earlier in this chapter that increasing the speed of processing yields higher throughput than increasing concurrency.

With regards to mean throughputs achieved, as before optimistic and WDL outperform 2PL but only at concurrencies over 10. Thus, for in-memory systems with a

small number of processors, in terms of mean throughput, there is no benefit in considering concurrency control mechanisms other than 2PL while for in-memory systems with a larger number of processors and a large number of available transactions there are quite significant gains in throughput to be had by implementing concurrency control mechanisms other than 2PL.

### *3.5.2 The Performance of Disk-based Systems*

Figures 3.15 to 3.19 below show the results of the disk-based systems as described in section 3.4.2. These systems have equivalent CPU power to the systems discussed in the preceding section – that is, 1248 processors operating at 4 MIPS per processor, 96 processors operating at 100 MIPS per processor, 96 processors operating at 200 MIPS per processor, 20 processors operating at 1000 MIPS per processor and 10 processors operating at 2000 MIPS per processor and. As indicated in section 3.3.3, the access invariance assumption is only valid at lower concurrencies, thus, the first system, that with 1248 processors operating at 4 MIPS per processor assumes no the access invariance and uses the kill optimistic method. In the latter systems, as indicated earlier, we assume that access invariance holds and the optimistic method used is a hybrid die-kill system. As well, in all these systems, we show the performance of these systems for equivalent sized transactions with no contention.

As with the in-memory systems, the performance of the optimistic and WDL concurrency control methods is considerably better than that of 2PL concurrency control in all the disk-based systems though in the case of the 4 MIPS per processor system this

superiority only becomes evident past concurrencies of 50. An interesting observation is that the superiority of optimistic concurrency control over WDL exhibited in the in-memory systems is maintained in the disk-based systems. This seems counter-intuitive given that the relative cost of wasted work, optimistic concurrency control's major liability, would seem to be much higher in disk-based systems. However, deeper analysis indicates that there are factors acting to reduce the cost of wasted work in our disk-based optimistic systems. In the massively parallel system, where access invariance does not hold and the kill method is used, a conflicting transaction is killed immediately thus reducing the amount of work wasted by killed transactions. In our other systems, where access invariance does hold and the optimistic kill-die system is used, all restarted transactions are executed in memory thus minimizing the cost of wasted work. In both cases, the effect is to reduce the relative cost of wasted work vis' a vis' the benefits of essential blocking.

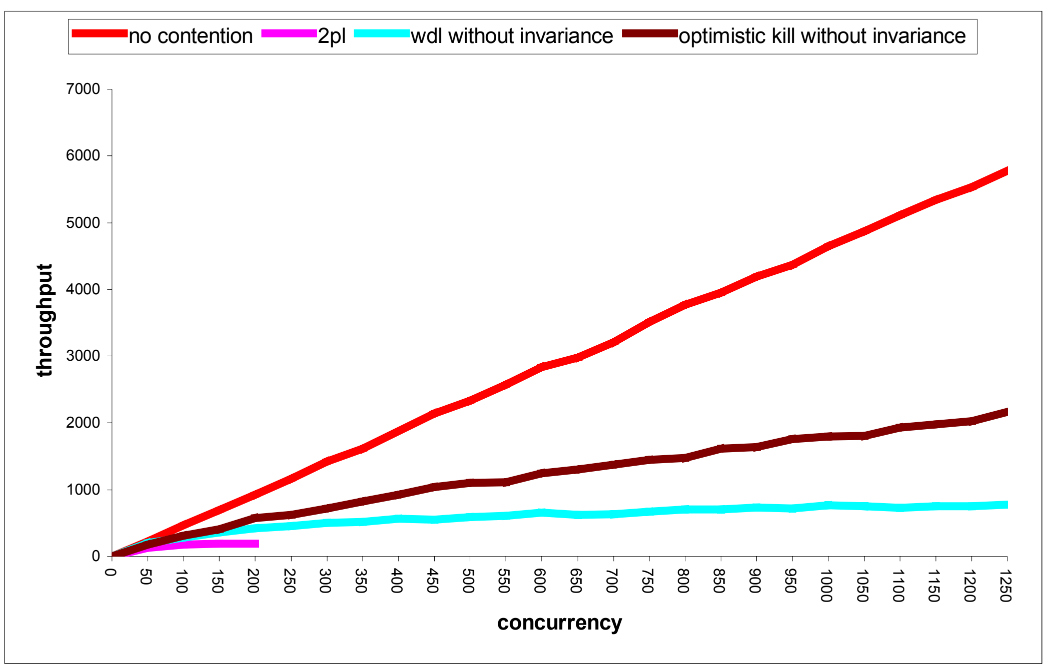



**Fig 3.15. A comparison of throughputs under 2PL, optimistic kill and WDL concurrency control methods for a system with 1248 processors each operating at 4 MIPS and with disk access at 15 milliseconds per access**

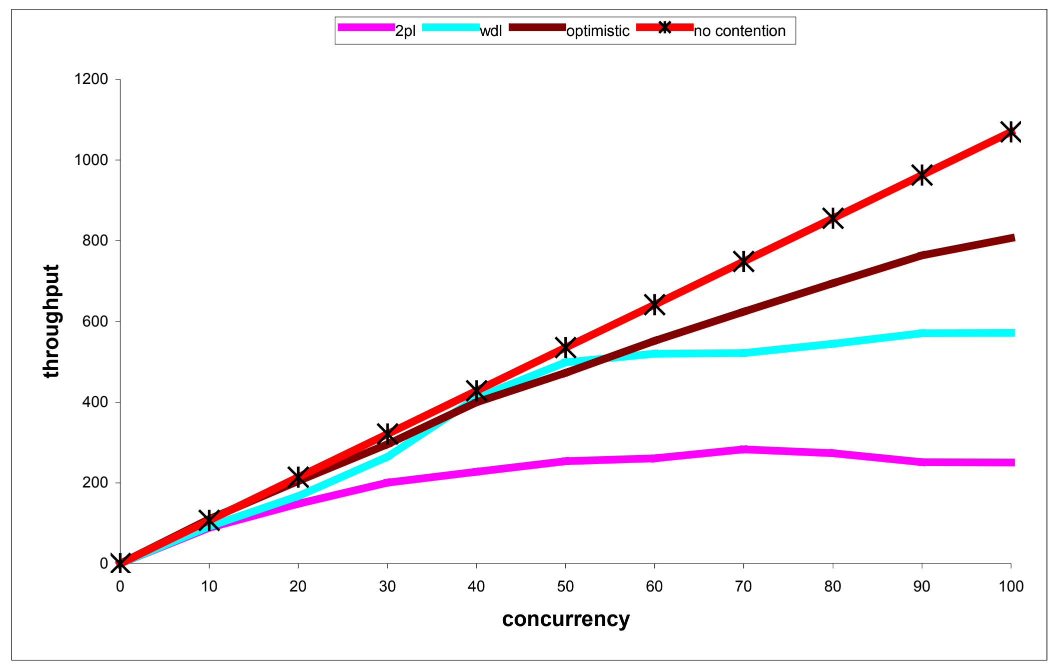



**Figure 3.16. A comparison of throughputs under 2PL, optimistic die-kill and WDL concurrency control methods for a system with 96 processors each operating at 100 MIPS and with disk access at 15 milliseconds per access**

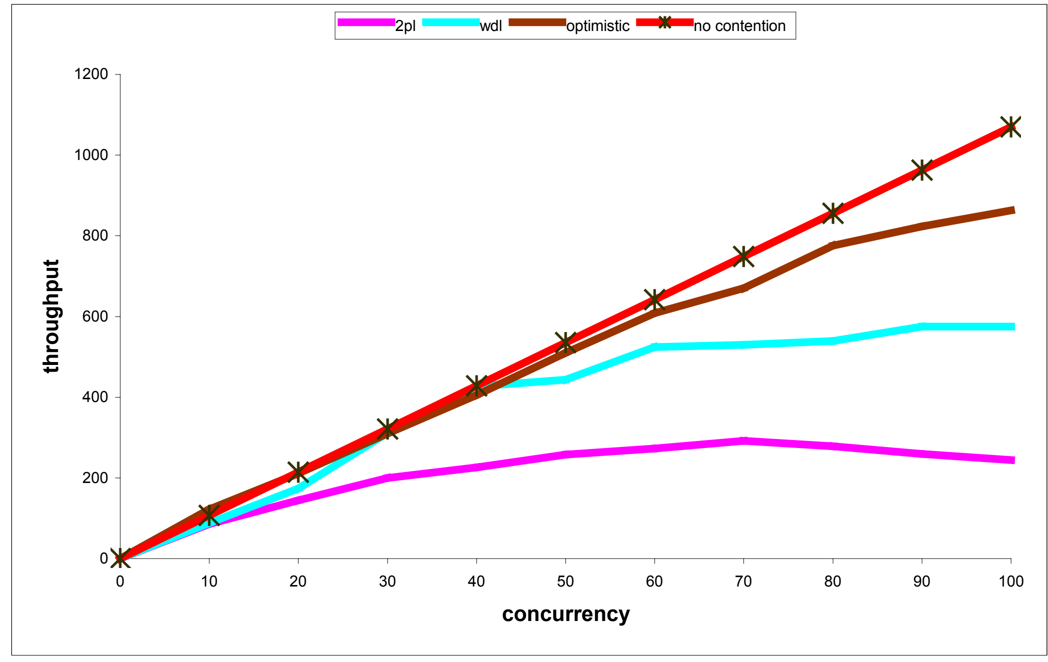



**Figure 3.17. A comparison of throughputs under 2PL, optimistic die-kill and WDL concurrency control methods for a system with 96 processors each operating at 200 MIPS and with disk access at 15 milliseconds per access**

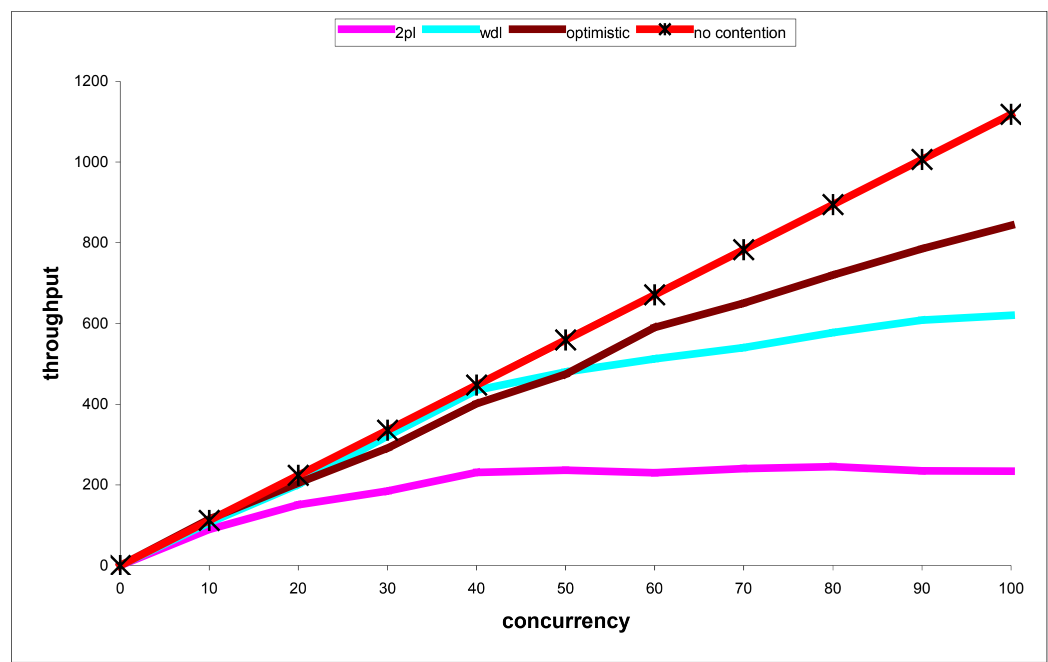



**Figure 3.18. A comparison of throughputs under 2PL, optimistic die-kill and WDL concurrency control methods for a system with 20 processors each operating at 1000 MIPS and with disk access at 15 milliseconds per access.**

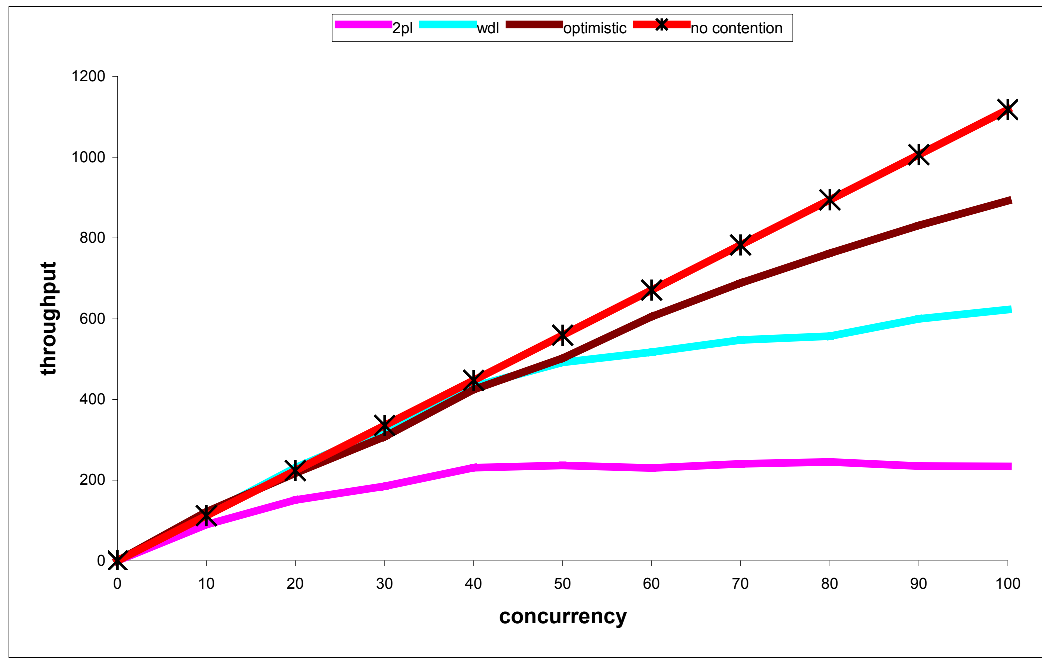



**Figure 3.19. A comparison of throughputs under 2PL, optimistic die-kill and WDL concurrency control methods for a system with 10 processors each operating at 2000 MIPS and with disk access at 15 milliseconds per access**

As indicated previously, one of the expected advantages of WDL over optimistic systems is that WDL redresses the disadvantage that larger transactions have relative to smaller transactions by giving larger locked transactions priority over smaller locked transactions. Figures 3.20 to 3.24 below break down the results of the preceding graphs and examine the relative performance of the optimistic and WDL concurrency control in the throughput of the largest transactions in the system - those belonging to transaction class $T_4$ (Here we concentrate on the performance of the highest contention transaction type. A more detailed breakdown of throughputs by all transaction types is presented in the following chapter).

As these graphs show, while the relative performance of WDL and optimistic concurrency control is as expected in the slow 4 MIPS per processor system, in the faster systems, optimistic concurrency control outperforms WDL in the throughput of the larger transactions.

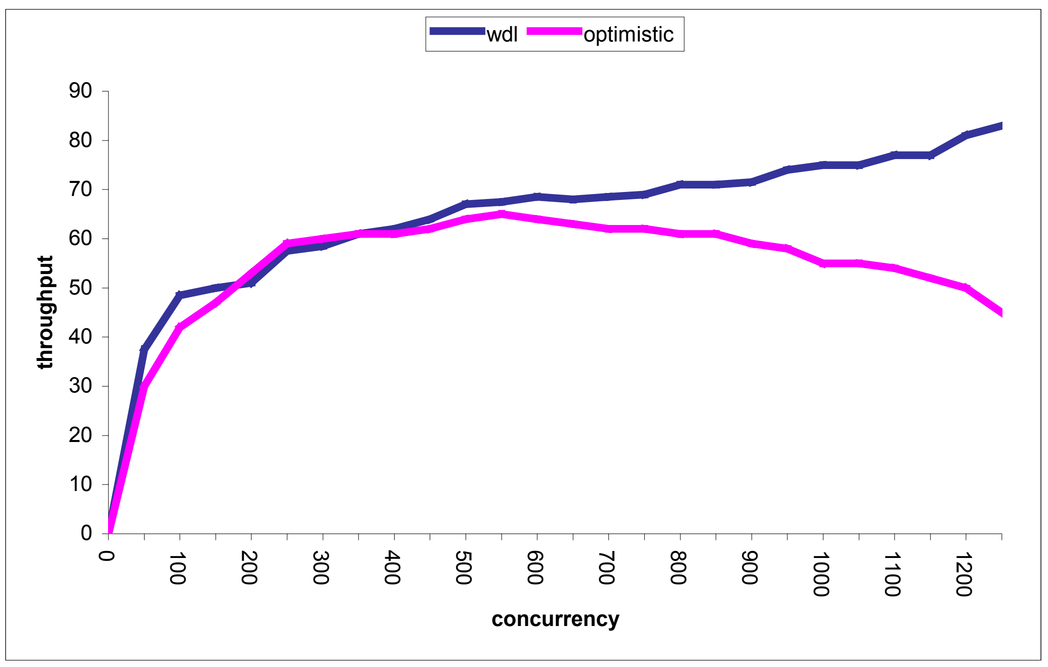



**Figure 3.20. A comparison of the throughput of large transactions given no access invariance under optimistic kill and WDL concurrency control methods for a system with 1248 processors each operating at 4 MIPS and with disk access at 15 milliseconds per access**

The explanation for this difference in behavior lies in the fact that access invariance holds at the lower concurrencies at which the faster systems operate whilst it does not hold at the higher concurrencies of the slower system. In the slower system, given that access invariance does not hold, there is no advantage in a conflicting transaction executing to completion since the objects acquired may need to be re-fetched in any case and thus the kill method is used. Here, a restarted transaction needs the same execution time on restart as it did initially. Larger transactions are more likely to be killed than smaller transactions and face the same disproportionate probability of being killed on restart. Thus, at high concurrencies, with no access invariance, WDL outperforms the kill optimistic method in the throughput of larger transactions.

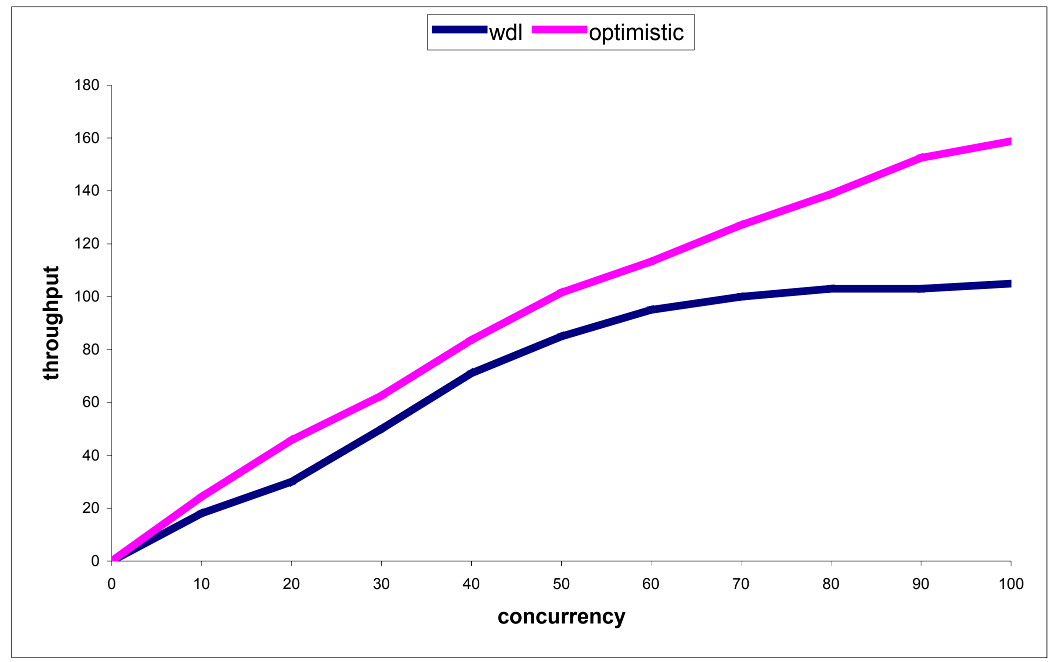



**Figure 3.21. A comparison of the throughput of large transactions given access invariance under optimistic kill-die and WDL concurrency control methods for a system with 96 processors each operating at 100 MIPS and with disk access at 15 milliseconds per access**

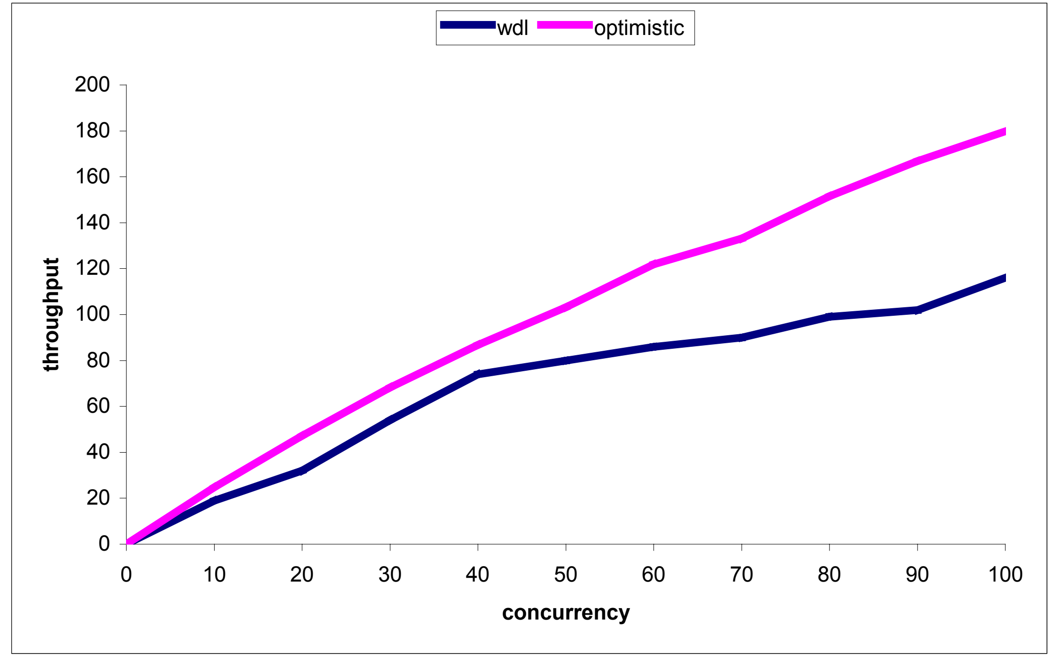



**Figure 3.22. A comparison of the throughput of large transactions given access invariance under optimistic kill-die and WDL concurrency control methods for a system with 96 processors each operating at 200 MIPS and with disk access at 15 milliseconds per access.**

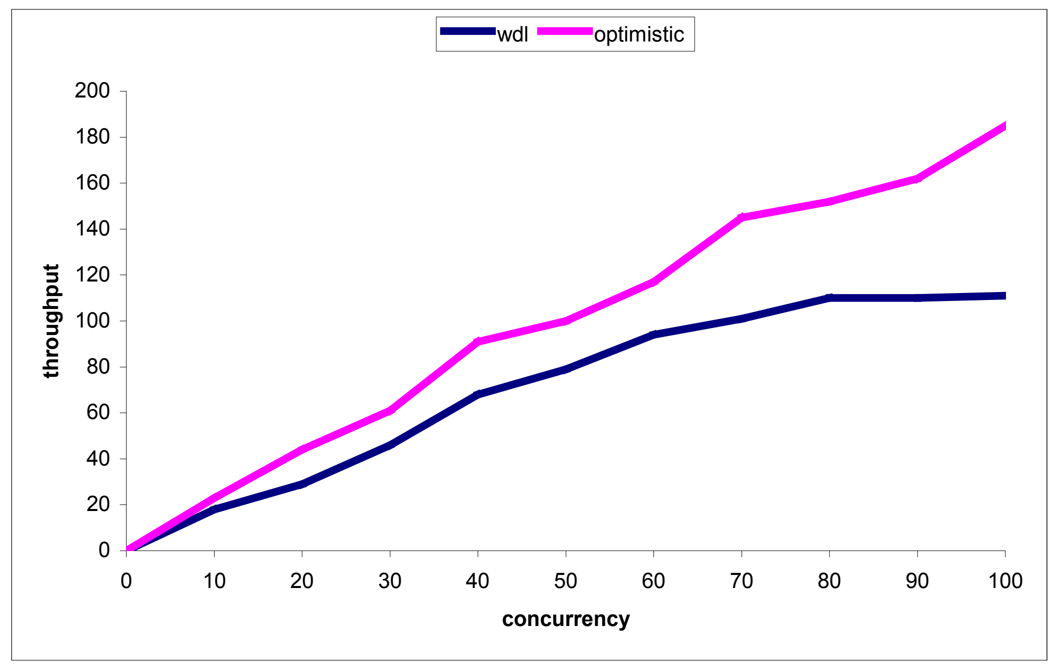



**Figure 3.23. A comparison of the throughput of large transactions given access invariance under optimistic** kill-die **and WDL concurrency control methods for a system with 20 processors each operating at 1000 MIPS and with disk access at 15 milliseconds per access.**

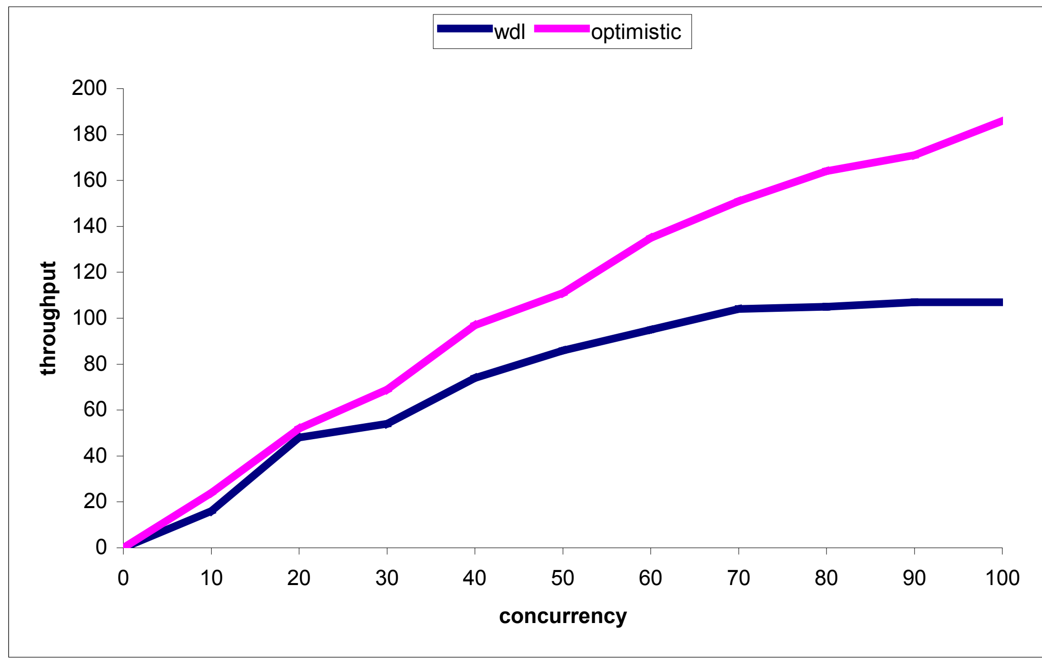



**Figure 3.24. A comparison of the throughput of large transactions given access invariance under optimistic** kill-die **and WDL concurrency control methods for a system with 10 processors each operating at 2000 MIPS and with disk access at 15 milliseconds per access**

In the systems where access invariance holds, a transaction that dies and restarts only needs to re-execute in memory. This makes the use of the die/kill optimistic concurrency control viable at the lower concurrencies used in the faster systems. Since the time taken to re-execute in memory for the longer transaction is less than the time that shorter transactions executing for the first time require when disk access is taken into account, longer transactions though more likely to die initially are more likely to succeed on restart. This advantage in execution time on restart conferred on larger transactions under the die/kill concurrency control method seems to be greater than the priority advantage conferred on the larger transactions by WDL.

In the previous section where the performance of the in-memory systems was examined, it was clear that the speed of processors was more important than the number of processors in determining system throughput. However, as Figures 3.25 to 3.27 below show, the importance of CPU speed is greatly reduced in disk-based systems. Figure 3.25 shows the relative performance of the disk-based systems under 2PL concurrency control, Figure 3.26 compares their performance under WDL and Figure 3.27 compares their performance under optimistic concurrency control.

In Figures 3.26 and 3.27, the throughput of the systems composed of 1248 processors each operating at 4 MIPS is shown at the peak achieved for these systems – that is at a concurrency of 1250. As well, as before, for the systems composed of 1248 processors each operating at 4 MIPS access invariance does not hold while for the faster systems operating at lower concurrencies it does. Consequently, as before, in Figure 3.27, the kill optimistic method is used for massively parallel system while the die-kill method is used for the other systems.

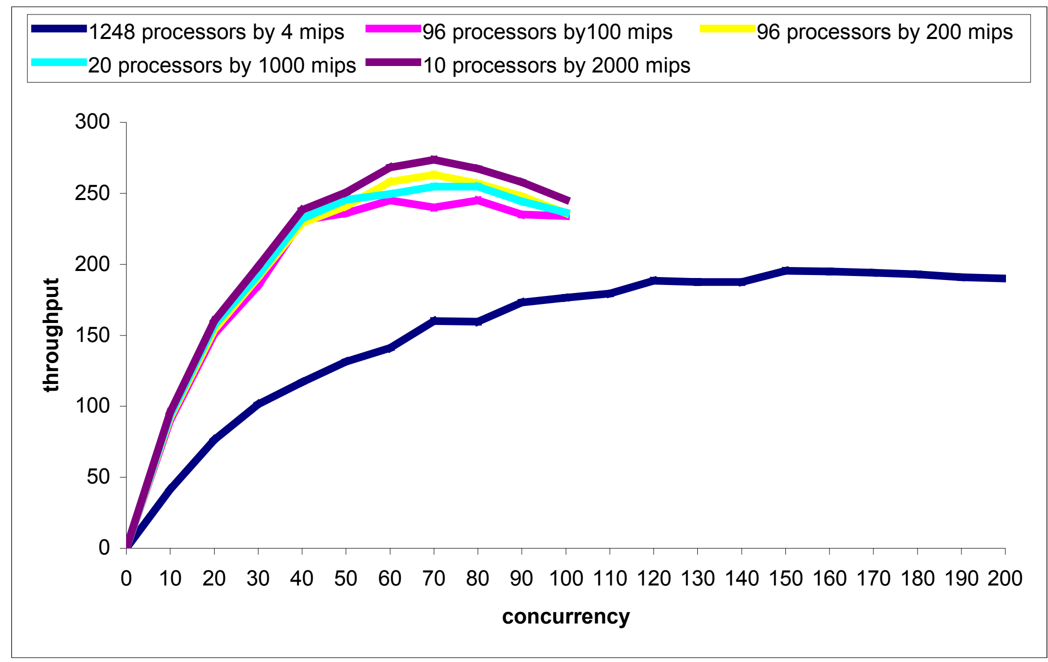



**Figure 3.25. The throughput of systems of systems operating under 2PL concurrency control**

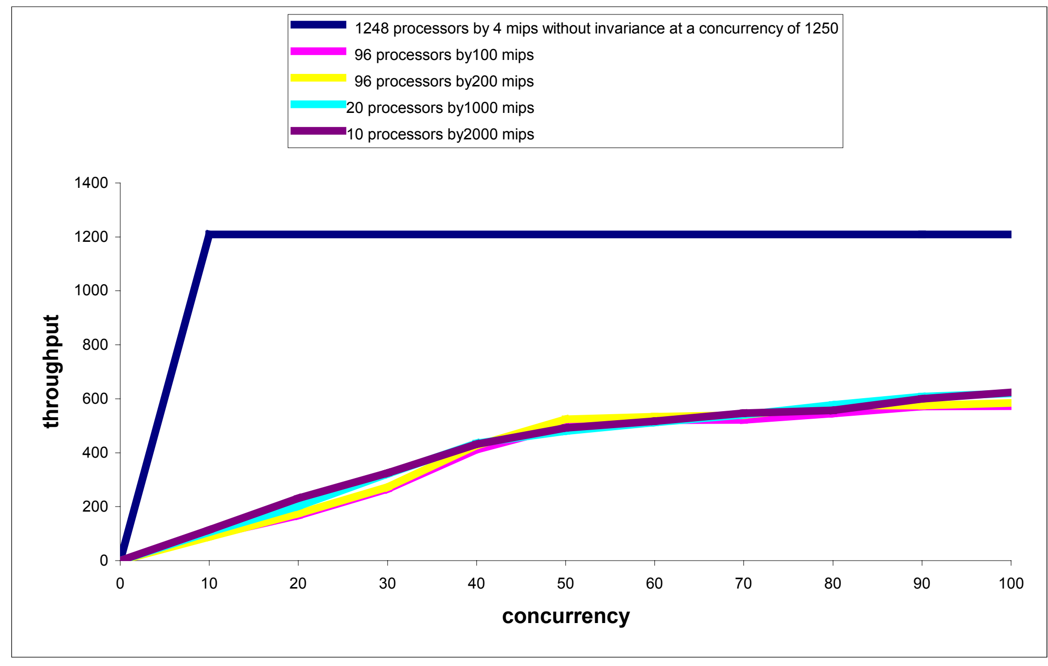



**Figure 3.26. The throughput of systems of systems operating under WDL concurrency control**

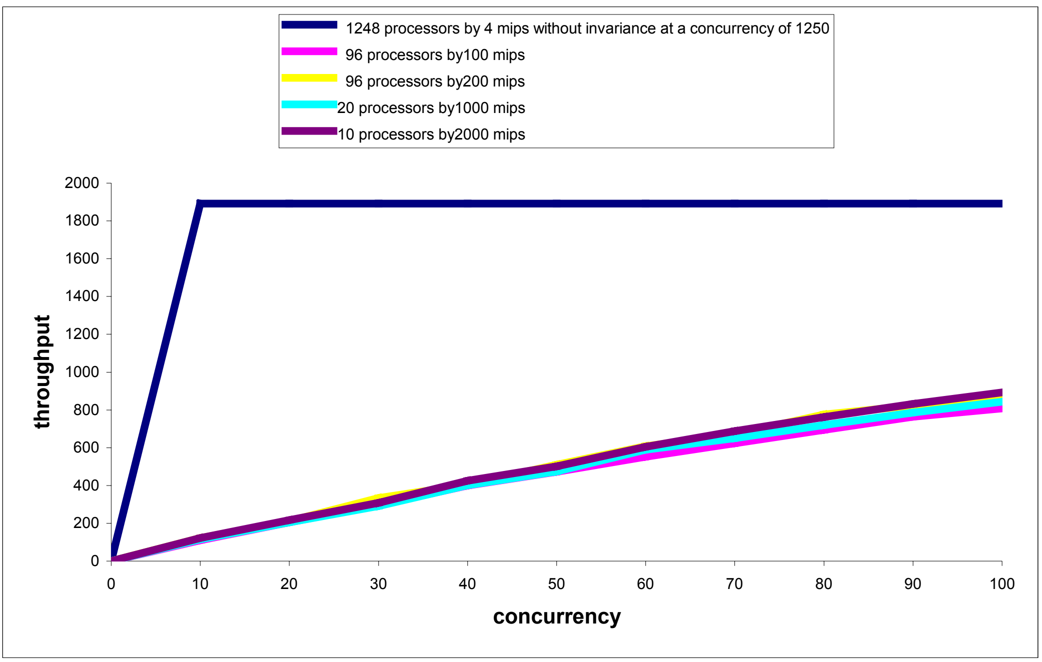



**Figure 3.27: The throughput of systems of systems operating under optimistic concurrency control**

As Figure 3.25 shows, under 2PL concurrency control, the systems with processors operating at 100 MIPS and over have very similar throughputs. Thus, the increase of CPU performance does not seem to have a significant impact on throughput. Similarly, while the performance of the massively parallel system is lower than the other systems, the deficiency is not proportional to its deficiency in processing speed[3]. The degree of variability in total throughputs in different runs displayed by the in-memory systems is absent in the disk–based systems. This is because with disk access, even the fastest of our systems completes relatively few cycles in one second and thus never reaches a point at which thrashing occurs.

---

[3] The massively parallel system's large number of processors does not affect its performance since its peak throughput is reached at a concurrency of 150.

Under both WDL and optimistic concurrency control, the performance of the systems with processor speed over 100 MIPS is virtually the same while the massively parallel system operating at its peak concurrency substantially outperforms the other systems. This is despite the fact that besides its slower processor speeds it is also operating without the benefits of access invariance – that is, its restarted transactions have the same processing time requirements as new transactions. This indicates that even without the benefits of access invariance, there are increases in throughput available at high concurrencies under both WDL and optimistic concurrency.

The explanation for the small effect that CPU speed has on system throughput lies in the disproportionate cost of disk access in determining the amount of time a transaction requires to complete. Given the CPU speeds, disk speed and cache hit rate parameters outlined in section 3.4.2, table 3.1 below shows the time required to complete a transaction assuming no clash occurs. As this table shows, despite the variation in CPU speed, the time required to complete a transaction is almost the same in the 100, 200, 1000 and 2000 MIPS disk-based systems. As well, despite the fact that the CPU speed of the 4 MIPS system is a five hundredth of the 2000 MIPS system, the time required to complete an average transaction is only half that of the 2000 MIPS system.

| | CPU SPEED | | | | |
|---|---|---|---|---|---|
| | 4 MIPS | 100 MIPS | 200 MIPS | 1000 MIPS | 2000 MIPS |
| Transaction Type | | | | | |
| $T_1$ | 0.070625 | 0.024425 | 0.023463 | 0.0226925 | 0.02259625 |
| $T_2$ | 0.115 | 0.0478 | 0.0464 | 0.04528 | 0.04514 |
| $T_3$ | 0.20375 | 0.09455 | 0.092275 | 0.090455 | 0.0902275 |
| $T_4$ | 0.38125 | 0.18805 | 0.184025 | 0.180805 | 0.1804025 |
| Average | 0.20375 | 0.09455 | 0.092275 | 0.090455 | 0.0902275 |

**Table 3.1. The time in seconds required to complete a transaction by CPU speed and transaction type**

### *3.5.3 A Comparison of the Performance of In-Memory and Disk-based Systems*

Figures 3.28 to 3.32 compare the results of the disk-based and in-memory systems by hardware configuration. In Figure 3.28 there are two hardware configurations - each system has 1248 processors operating at 4 MIPS per processor, however one configuration is in-memory and the other is disk-based. Figure 3.29 shows the equivalent comparisons for systems with 96 processors operating at 100 MIPS per processor, Figure 3.30 compares the results for the hardware configurations with 96 processors operating at 200 MIPS per processor, Figure 3.31 compares the results for the hardware configurations with 20 processors operating at 1000 MIPS per processor and Figure 3.32 compares the results for the hardware configurations with 10 processors operating at 2000 MIPS per processor.

Overall, in the massively parallel systems, as expected, the in-memory systems outperform the disk-based systems. However, the degree of the advantage is dependant

on the concurrency control method chosen. Thus, as Figure 3.28 shows, the performance of in-memory and disk-based systems under 2PL concurrency control is comparable while the advantage of the in-memory systems under WDL and optimistic concurrency control are more pronounced being almost double their equivalent in the disk- based systems

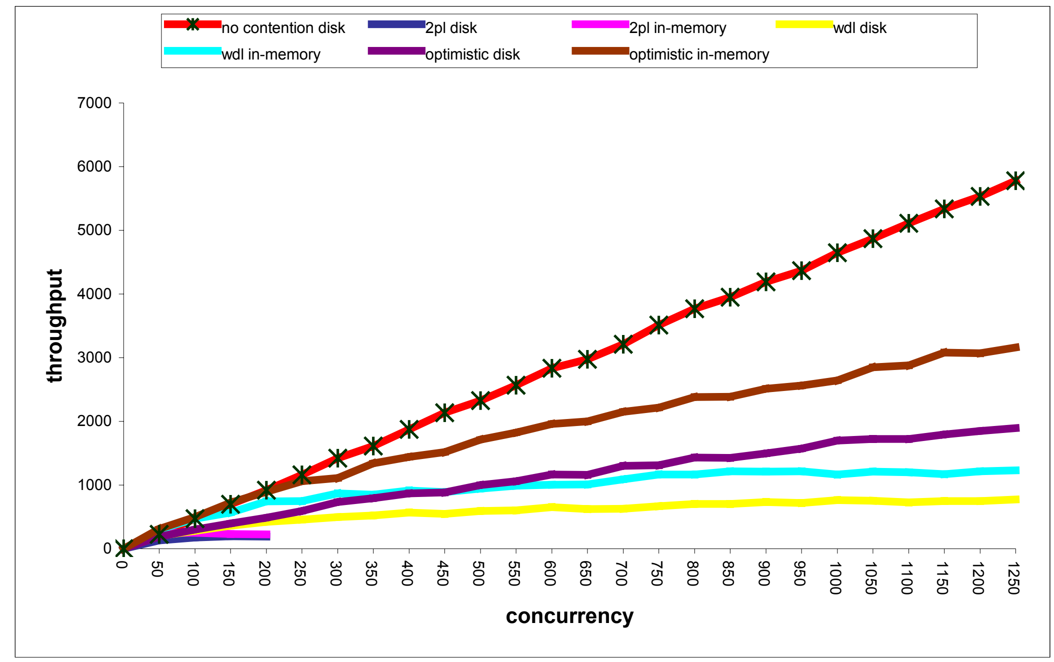



**Figure 3.28. A comparison of in-memory and disk-based systems with each configuration having 1248 processors operating at 4 MIPS per processor**

In the systems with fast processors, the in-memory systems outperform the disk-based systems for any concurrency control mechanism. Thus, as Figures 3.29 to 3.32 show, for all the fast processor systems, 2PL, the worst performed concurrency control method in the in-memory systems substantially outperforms the best results of the best performing concurrency control methods used by the disk-based systems (optimistic die-kill). In the case of the 100 MIPS per processor systems, the in-memory system using 2PL has a throughput more than 5 times that of the best result in the equivalent disk-based system operating under the die-kill optimistic control mechanism. In the 200 MIPS per

processor system the ratio is over 8:1, in the 1000 MIPS per processor the ratio is over 24:1 and in the 2000 MIPS per processor system the ratio is over 46:1.

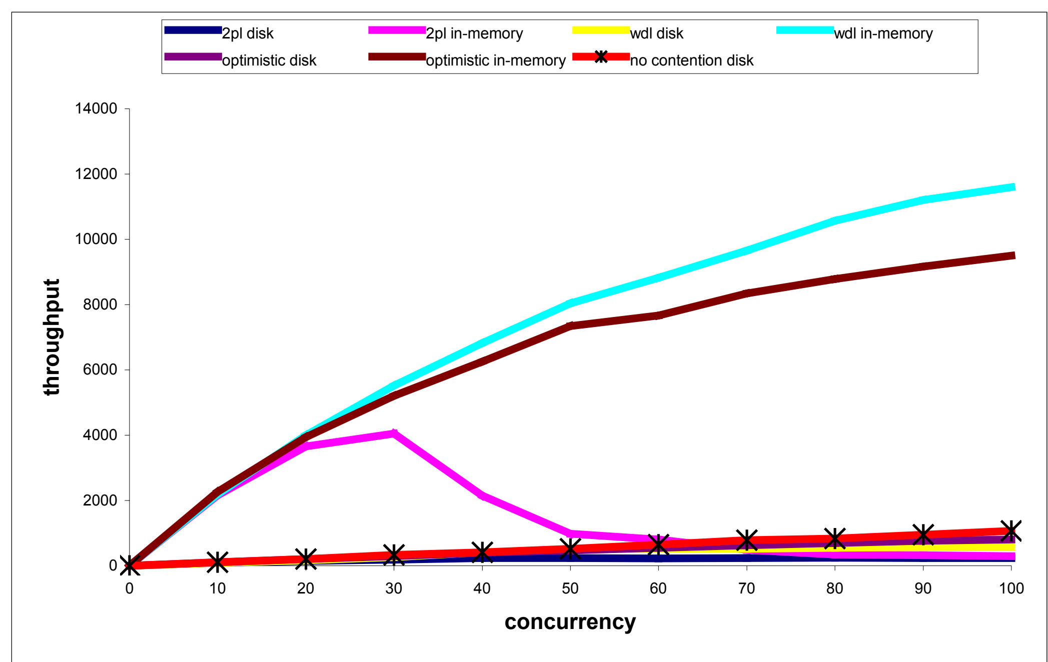



**Figure 3.29. A comparison of in-memory and disk-based systems with each configuration having 96 processors operating at 100 MIPS per processor**

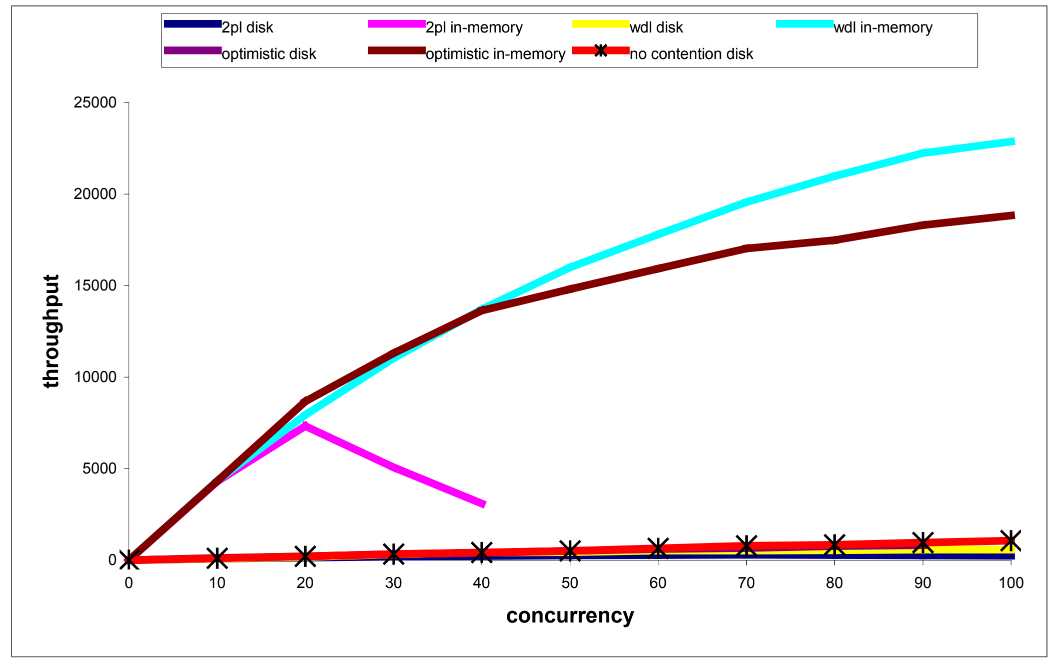



**Figure 3.30. A comparison of in-memory and disk-based systems with each configuration having 96 processors operating at 200 MIPS per processor**

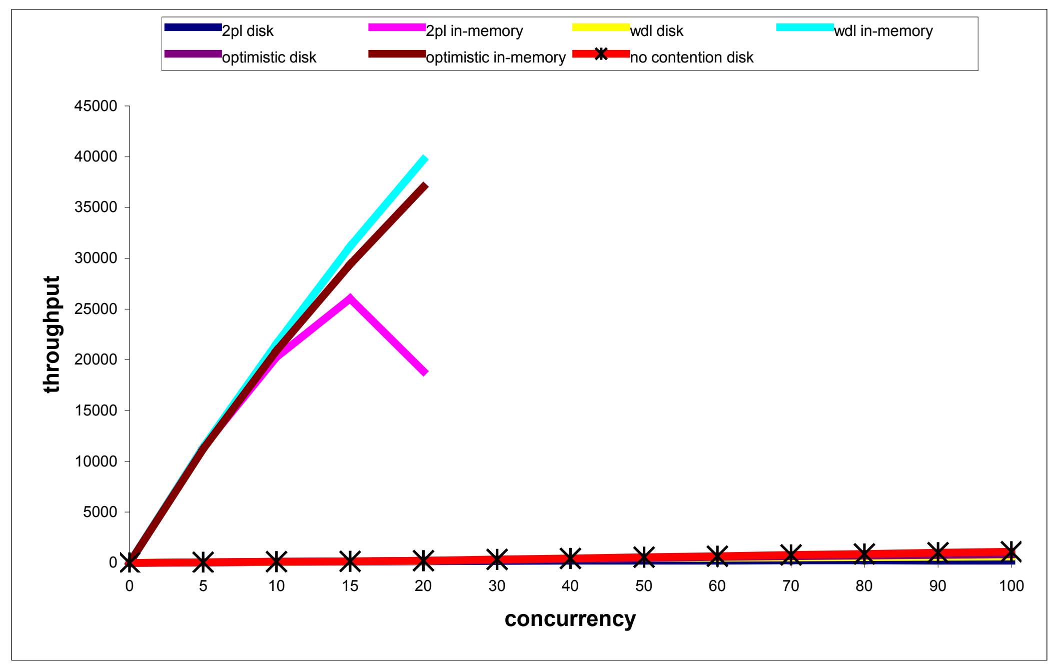



**Figure 3.31. A comparison of in-memory and disk-based systems with each configuration having 20 processors operating at 1000 MIPS per processor**

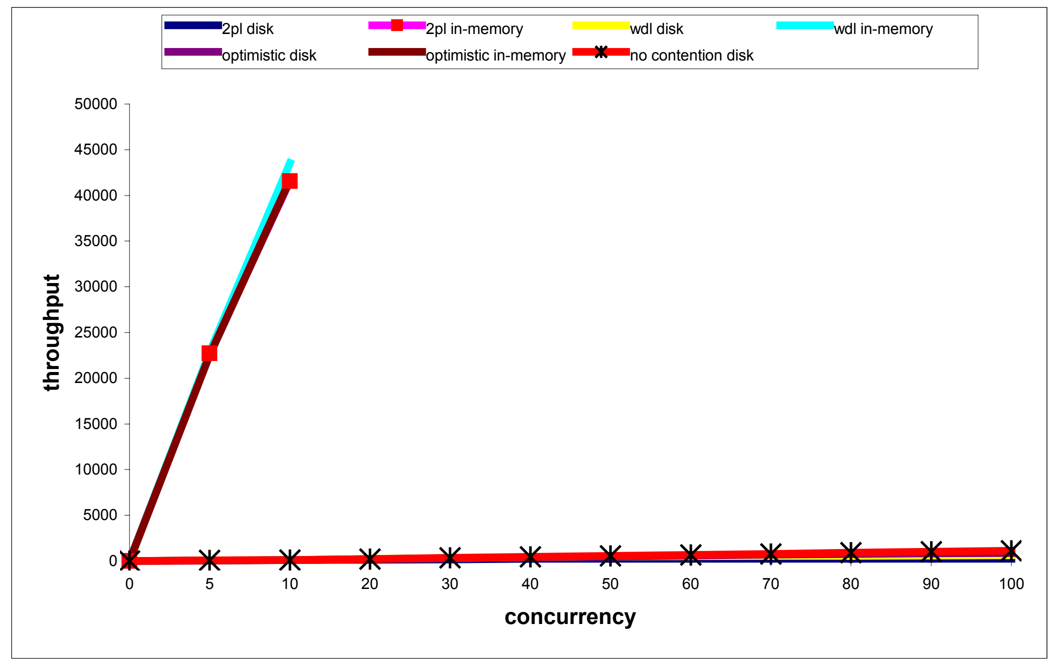



**Figure 3.32. A comparison of in-memory and disk-based systems with each configuration having 10 processors operating at 2000 MIPS per processor**

In terms of performance under comparable concurrency control mechanisms, under WDL concurrency control, the throughput of the 100 MIPS per processor in-memory system at a concurrency of 100 is over 20 times that of the equivalent disk-based system operating at the same concurrency. This ratio is 1:40 for the 200 MIPS per processor systems,1:63 for the 1000 MIPS per processor systems and 1:70 for the 2000 MIPS per processor systems.[4]

Under optimistic concurrency control, the peak throughput of the 100 MIPS per processor in-memory system is nearly 12 times that of the equivalent disk-based system while this ratio is 22:1 for the 200 MIPS per processor systems, 44:1 for the 1000 MIPS systems and 47:1 for the 2000 MIPS systems.

This difference in the potential performance of in-memory systems and disk-based systems operating at limited concurrency is further highlighted by Figure 3.33, which compares hardware configurations operating under 2PL concurrency control. Besides comparing the performance of hardware systems with very fast processors under 2PL Figure 3.33 also shows the throughput of these fast processor disk-based systems operating at a concurrency of 100 with 0 contention. As this graph shows, not only do the 2PL in-memory systems outperform the disk-based optimistic and WDL concurrency control systems, they also outperform the disk-based systems that operate with 0 contention. Here, the 2PL in-memory system with 96 processors operating at 100 MIPS

---

[4] In the case of the 1000 MIPS per processor in-memory systems, peak concurrency is achieved at a concurrency of 20 while for the 2000 MIPS per processor in-memory systems, peak concurrency is achieved at a concurrency of 10 – that is, peak concurrency is equal to the number of processors in the respective systems.

per processor has more than double the throughput of its equivalent disk-based system with 0 contention while the 2PL in-memory system with 200 MIPS processors has more than triple the throughput of its equivalent disk-based system with 0 contention. For the 1000 MIPS systems this ratio is 12:1 while for the 2000 MIPS systems this ratio is 20:1.

Since operation at 0 contention sets the upper limit on throughput for any hardware configuration at any given concurrency level, in systems with fast processors, clearly, no disk-based concurrency control mechanism can approach the performance of even the worst performing concurrency control mechanism without in some way enabling very high levels of concurrency.

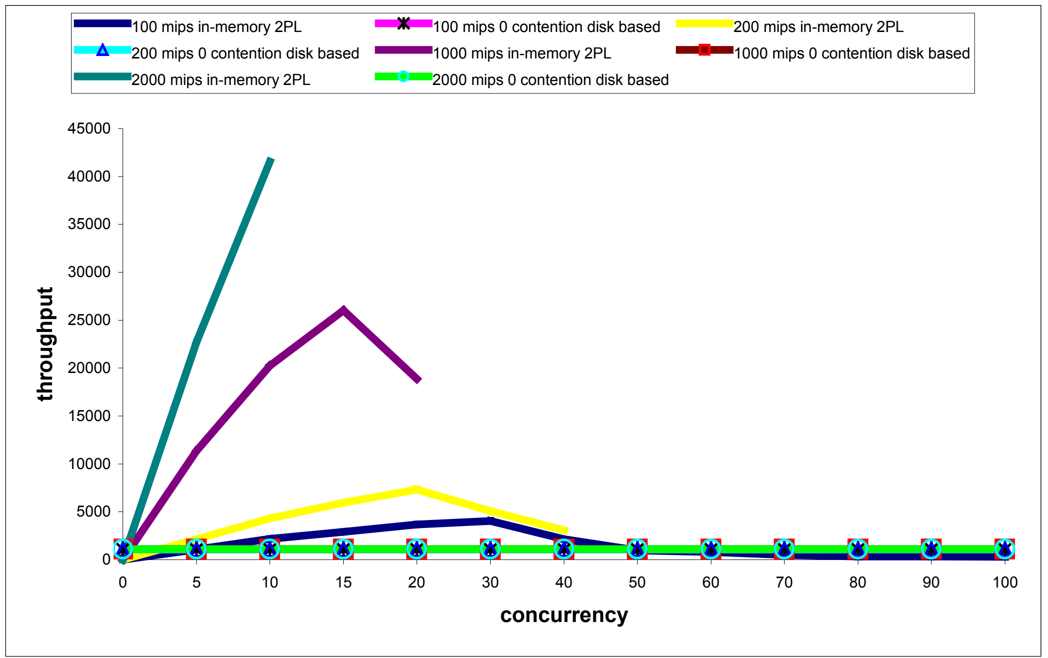



**Figure 3.33. A comparison of in-memory systems with 96 processors operating at 100 and 200 MIPS per processor, 20 processors operating at 1000 MIPS per processor and 10 processors operating at 2000 MIPS per processor operating under 2PL concurrency control with their equivalent disk-based systems operating at 0 contention and a concurrency of 100**

## 3.6 Thrashing in 2PL Concurrency Control Systems

So far in this chapter we have shown the overall results for the throughput of 2PL systems under diverse hardware configurations and compared these results to those achieved under alternative concurrency control mechanisms. However we have not investigated some of the behavioral characteristics of 2PL systems outlined in section 3.2. In particular, we have not investigated the cyclical behavior of 2PL systems and the varying performance of different transaction types in systems with multiple transaction types. This latter aspect of 2PL behavior is investigated in more detail in the next chapter where the varying behavior of different transaction types is critical to the algorithms presented for improving the performance of disk-based 2PL systems. In this chapter we concentrate on the problems of thrashing and deadlock.

As indicated in section 3.2.1, according to [15] in 2PL systems, unless some preventative action is taken, throughput will decrease in every cycle and will eventually reach 0. That is, thrashing is inherent in all 2PL systems. This is because in every cycle the proportion of transactions in what is commonly known as a wait chain grows and the proportion of transactions woken by completing transactions decreases. However, according to [47] this occurs very infrequently in systems with fixed size transactions unless the degree of lock contention is greater than 0.226. In systems with variable size transactions, the random arrival of large transactions may temporarily lead to a lock contention greater than 0.226 thus destabilizing the system and leading to thrashing.

A validation of the hypothesis in [15] would require a demonstration that for any contention level and at any concurrency, throughput decreases in every successive cycle

while a validation of the hypothesis in [47] requires a demonstration that in systems with fixed size transactions thrashing occurs very infrequently at lock contentions below 0.226 though it may occur more frequently at lock contentions below 0.226 in systems with variable sized transactions if a random arrival of large number of larger transactions temporarily raises lock contention over 0.226.

An initial inspection of the results presented in the previous section seems to indicate a conflict between these results and the hypothesis in [47]. At a concurrency of 5, none of the systems exhibited thrashing which is consistent with [47]. However, at a concurrency of 10, 6 runs, (three in the in-memory 20 by 1000 and three in the10 by 2000 MIPS per processor systems) exhibited thrashing. At this concurrency for these two systems, the incidence of thrashing seems to be randomly distributed in terms of time of occurrence with the earliest incidence occurring after 0.064 seconds and the latest incidence occurring after 0.96 seconds. In the 20 by 1000 MIPS system, 11 out of 40 runs thrashed at a concurrency of 15 and 25 out of 40 of the runs thrashed at a concurrency of 20.

The variability in transaction contention/size in our systems does not account for this relatively high incidence of thrashing. At a concurrency of 10, even if all transactions in the system at the time thrashing began were the highest contention $T_4$ transactions, then by the measurement used in [47], lock contention would be around 0.096 while at a concurrency of 20, if all transactions in the system were $T_4$ transactions, lock contention would be around 0.2. In both cases this is significantly below the threshold of 0.226 prescribed by [47].

In fact, in the 10 by 2000 MIPS system, none of the runs in which thrashing occurred at a concurrency of 10 contained 10 $T_4$ transactions at the time thrashing began. At this concurrency the largest number of $T_4$ transactions in the system at the time thrashing occurred was 7 the lowest was 3 and the mean was around 4 – the same mean as for those runs in which thrashing did not occur. Similarly, in the 20 by 1000 MIPS system, the mean number of $T_4$ transactions in the system at the time of thrashing began was 8 as against the mean of 6.5 for runs in which no thrashing occurred. In this system at a concurrency of 20, in runs in which thrashing occurred the mean number of $T_4$ transactions was 10.2 as against a mean of 9.5 for those runs in which thrashing did not occur.

To confirm these findings we ran the 1000 and 2000 MIPS per processor with fixed size/contention transactions using the medium sized $T_3$ transactions in one set of tests and the large sized $T_4$ transactions in another set of tests. We had 40 runs at each concurrency of 5 and 10 in each system. We had an additional 40 runs at each concurrency of 15 and 20 in the 1000 MIPS by 20 processors system.

In the tests using only $T_3$ transactions, as in the tests using variable sized transactions, no thrashing occurred at a concurrency of 5. In each of the other tests thrashing occurred once in each test. This was a considerably better performance than

that achieved in the tests using variable sized transactions and substantially conformed to the hypothesis in [47].

In the tests using only $T_4$ transactions as in the tests using variable sized and $T_3$ transactions, no thrashing occurred at a concurrency of 5. However at higher concurrencies the number of runs at which thrashing occurred was quite high. For example at a concurrency of 10, 15 out of the 40 runs in the 1000 MIPS per processor system and 27 out of 40 runs in the 2000 MIPS per processor thrashed.

While an initial reading of these results do not seem to support the hypothesis in [47], they also tend to conflict with the hypothesis presented by [15] that throughput diminishes with every cycle – these results did not display this tendency at low concurrencies. To investigate the performance of 2PL systems at higher concurrencies we ran the in-memory 1248 by 4 MIPS system in 4 sets of tests. Each set of tests ran for 8 cycles (a cycle being the time required to process a transaction when no conflict occurred) with the number of transactions completed in each cycle recorded.

The first set of tests used only low contention $T_2$ transactions at concurrencies starting at 50 and incrementing by 50 to 1000 while the second set of tests used only medium contention $T_3$ transactions at concurrencies starting at 10 and incrementing by 10 to 200. The third set of tests used a slightly modified version of the variable sized

transaction model used elsewhere in this chapter. In this modification access to the low contention database $D_2$ was used to make all transactions the same size. Thus as before, $T_1$, $T_2$, $T_3$ and $T_4$ transactions accessed 1,2,4 and 8 objects respectively from database $D_1$ (containing 1000 objects). However in this model, $T_1$, $T_2$, $t_3$ and $T_4$ transactions accessed 15,14,12 and 8 objects respectively from database $D_2$ (containing 1000000 objects). Thus while contention of each transaction class varied, the size of all transactions was the same. This variation was used to make cycles easily identifiable since all transactions require the same processing time. The fourth set of tests used only high contention $T_4$ transactions at concurrencies starting at 10 and incrementing by 10 to 100. The results of these tests are presented in Figures 3.34 to 3.37 below.

The results shown in these graphs partially confirm the hypothesis in [15] that once a threshold concurrency is passed, throughput drops consistently in each successive cycle. The threshold concurrency at which this occurs varies inversely with the contention of the transactions in the system. Thus, with $T_2$ transactions this threshold occurs at a concurrency of around 150, for $T_3$ transactions this threshold occurs at a concurrency of around 30 and with $T_4$ transactions this threshold occurs at a concurrency of around 10.

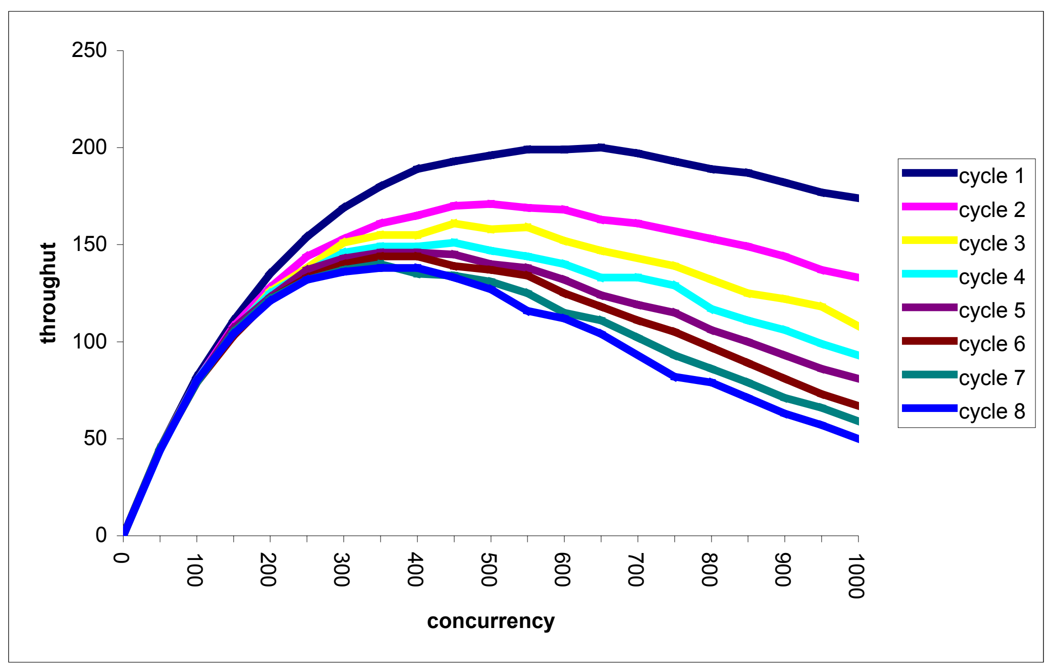



Figure 3.34: Throughput of $T_2$ transactions per cycle over 8 cycles

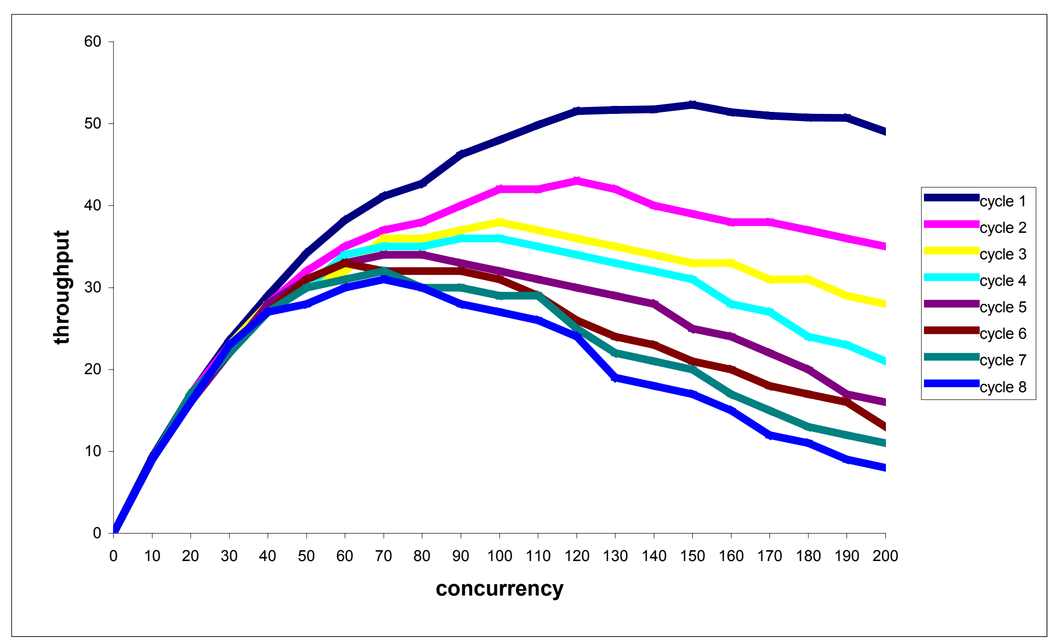



Figure 3.35. Throughput of $T_3$ transactions per cycle over 8 cycles

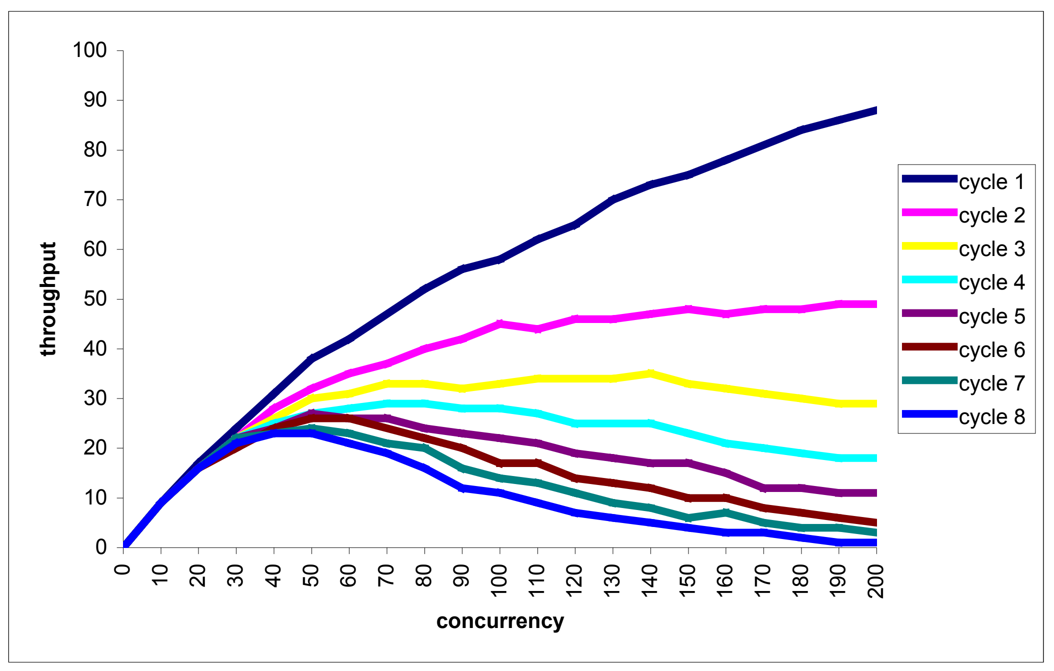



**Figure 3.36. Throughput of the system with fixed size variable contention transactions per cycle over 8 cycles**

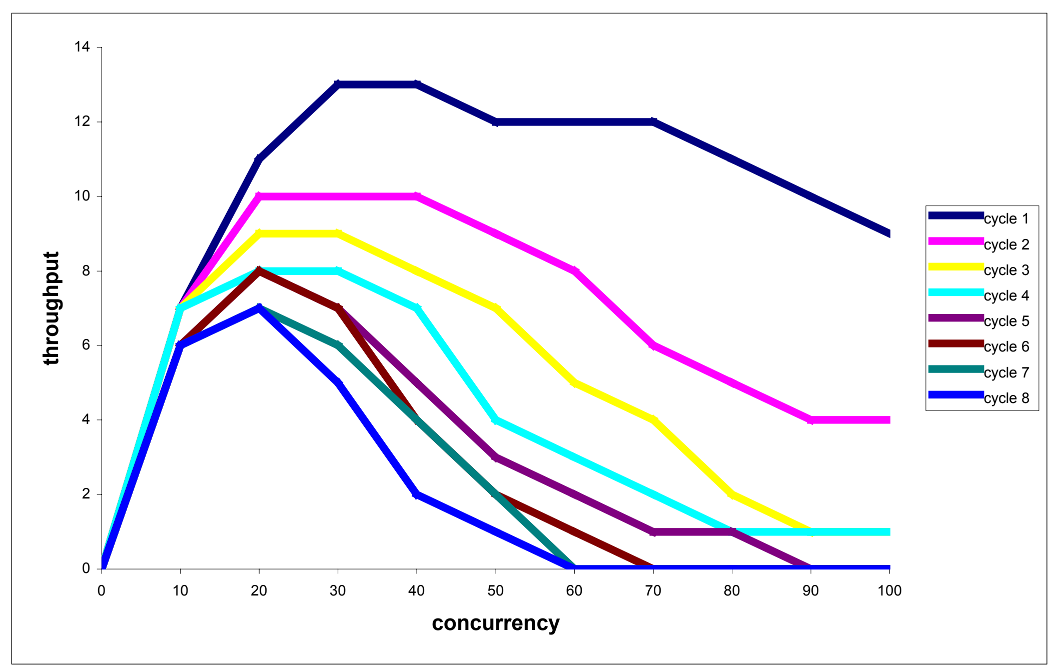



**Figure 3.37: Throughput of $T_4$ transactions per cycle over 8 cycles**

Interestingly, in the variable contention fixed size system while the size and mean number of items required from the high contention database is the same as for the system composed entirely of $T_3$ transactions, the behavior of the systems is quite different. The former has a higher initial throughput in the first cycle but has a more precipitous fall in throughput in later cycles. As well, in the variable contention fixed size system, the threshold concurrency is around 20, 10 below the system composed entirely of $T_3$ transactions.

While they seems to be conflict, the results presented in this section can be reconciled with the hypothesis in [47] if one makes a distinction between the incidence of thrashing and the probability of thrashing. While our results show a fairly high incidence of thrashing at low concurrencies, they also show that at any point in time, the probability of thrashing is indeed extremely low.

For example, in the tests using only $T_3$ transactions at a concurrency of 10, of the 349607 cycles completed, only 2 - that is 0.000286% of cycles completed exhibited thrashing. In the tests using the variable sized transactions, using the mean required processing time per transaction as the length of a cycle, of the 331765 cycles completed, only 6 - that is 0.001809% of cycles completed exhibited thrashing. In the tests using only $T_4$ transactions at a concurrency of 10, of the 113030 cycles completed, only 41 - that is 0. 036273% of cycles completed exhibited thrashing.

Thus, on the one hand, a very significant number of runs in the high-speed in – memory systems exhibited thrashing within 1 second of commencing operation while on the other hand, the proportion of cycles in which thrashing occurred was very small. These two seemingly contradictory conclusions can be reconciled by the extremely high number of cycles completed by the very fast in-memory systems. That is, while the probability of thrashing is very small at low concurrencies, given a constant arrival of sufficient medium to high contention transactions, the system will eventually thrash even at low concurrencies. In in-memory systems where a very large number of cycles can be completed very quickly, this thrashing will often occur within 1 second of operation.

The acceptability of this small probability of thrashing depends on the nature of the system. If the system is such that the availability of transactions enables continuous processing at a concurrency of around 10 and over for a very large number of cycles, then even this small probability of thrashing is unacceptable since it nevertheless translates to a high incidence of thrashing over time - particularly in high-speed in-memory systems where it occurs very quickly. On the other hand, if the availability of transactions is such that they can be cleared before the arrival of a new packet of transactions, then the small probability of thrashing also translates to a low incidence of thrashing over time and is thus likely to be acceptable to users.

## 3.7 Summary

In this chapter we outlined three concurrency control methods 2PL, WDL and optimistic and examined their performance relative to each under diverse hardware configurations. The salient points raised by the results presented in this chapter are-

1. In all our tests, WDL and optimistic concurrency control outperformed 2PL by a wide margin on equivalent hardware at concurrencies of 20 and over. This is consistent with the well-known observation that in systems with abundant hardware capacity, 2PL performs relatively badly.

2. The rates of throughput achievable by in-memory systems under 2PL, WDL and optimistic concurrency control are massive with throughputs of over 40000 transactions per second being achieved in the fastest of our in-memory systems.

3. While average throughput under 2PL concurrency control in the in-memory systems with fast processors was quite high, this performance was subject to quite large variations even at low concurrencies. Indeed, in some runs, throughput in even the fastest systems was not very much greater than that achieved by the equivalent disk-based systems.

4. Throughputs under all concurrency control mechanisms in the in-memory systems with speeds over 100 MIPS per CPU were well above the best results achieved by any concurrency control method in the disk-based systems.

With 2PL in-memory systems capable of addressing 3 gigabytes currently available and systems capable of addressing 6 gigabytes of memory due to be released soon,

these results seem to indicate that for many organizations, the best way to improving the performance of their transaction processing systems lies in switching to in-memory systems rather than improving the performance of their disk-based systems. This is particularly so since the cost of memory is now relatively low.

However, for those organizations that have very large databases that cannot fit into 6 gigabytes, the option-of an in-memory database is not viable. For these organizations, the only way of increasing throughput is by improving the performance of disk-based concurrency control. In the following chapters we examine how the performance of disk-based systems can be improved by effectively harnessing very high concurrencies.

# Chapter 4

## *New Strategies for Improving the Performance of 2PL Concurrency Control*

## 4.1 Introduction

For those organizations that have very large databases that cannot fit into an in-memory database, the only way of increasing throughput is by improving the performance of disk-based concurrency control. This is particularly pertinent for 2PL concurrency control, which remains virtually the only concurrency control mechanism commercially used. As well, while in-memory systems are now becoming available, disk-based systems are dominant and are likely to remain so for some time. Because of this commercial importance of disk-based 2PL systems, in this chapter, we investigate how the throughput of such systems can be improved by manipulating contention.

As indicated in the previous chapter, 2PL systems are susceptible to thrashing. This problem is particularly pronounced at lock contentions greater than 0.226 but, as shown in the previous chapter, can also be a serious problem at much lower lock contentions if the availability of transactions is such that the system can continuously process for a very large number of cycles. In this chapter we also address the issue of managing thrashing.

## 4.2 Improving the Performance of 2PL by Manipulating Contention

Under standard 2PL concurrency control, arriving transactions are allowed into the system on a first come first serve basis. Only the permitted level of concurrency restricts entry into the system. We propose a contention-based scheduler that measures transactions' contention as they arrive sorts these transactions into queues of transactions of similar contention and then manipulates the number of transactions allowed into the system by the contention class. By doing so, the contention-based scheduler can dramatically increase the system's throughput.

The rationale behind this manipulation is that lower contention transactions have a lower probability of being involved in a conflict. It is thus possible to increase the effective level of concurrency by permitting a high proportion of low contention transactions entry into the system while restricting the number of high contention transactions allowed entry into the system. This is because the increase in throughput gained by increasing the

number of lower contention transactions is larger than the throughput lost by reducing the higher contention transactions.

Besides increasing the throughput of lower contention transactions, because the number of transactions of each type allowed into the system is determined by the scheduler rather than by the arrival of transactions, the system is not destabilized by the random arrival of a large number of larger transactions thus reducing the probability of a cusp catastrophe and consequent system thrashing. As such, this contention-based scheduler is an extension of a suggestion in [47] that limiting the number of larger transactions can control thrashing.

### *4.2.1 Mechanical Principles*

A description of the operation of the contention-based scheduler is as follows. As transactions arrive, the scheduler does a pre-fetch for each transaction to establish its contention and then places it on its appropriate queue. Such a pre-fetch is only for the purposes of estimating contention and does not actually acquire or release locks. Having sorted transactions according to their contention, the scheduler ensures that no more than the prescribed number of each transaction type is allowed into the system. The schema for such a scheduler is shown in Figure 4.1 below.

In Figure 4.1, at time 1, a new transaction $T_n$ arrives. The contention-based scheduler determines the transaction's contention and having determined this puts it at the end of its appropriate queue (time 2). At time 3, an active transaction $T_i$ completes. The scheduler notes this completion and adjusts its computation of the numbers and

proportions of transactions released into the system. At time 4, it calculates which type of transaction is to be released into the system. It then goes to the appropriate queue and releases the transaction at the head of this queue into the system.

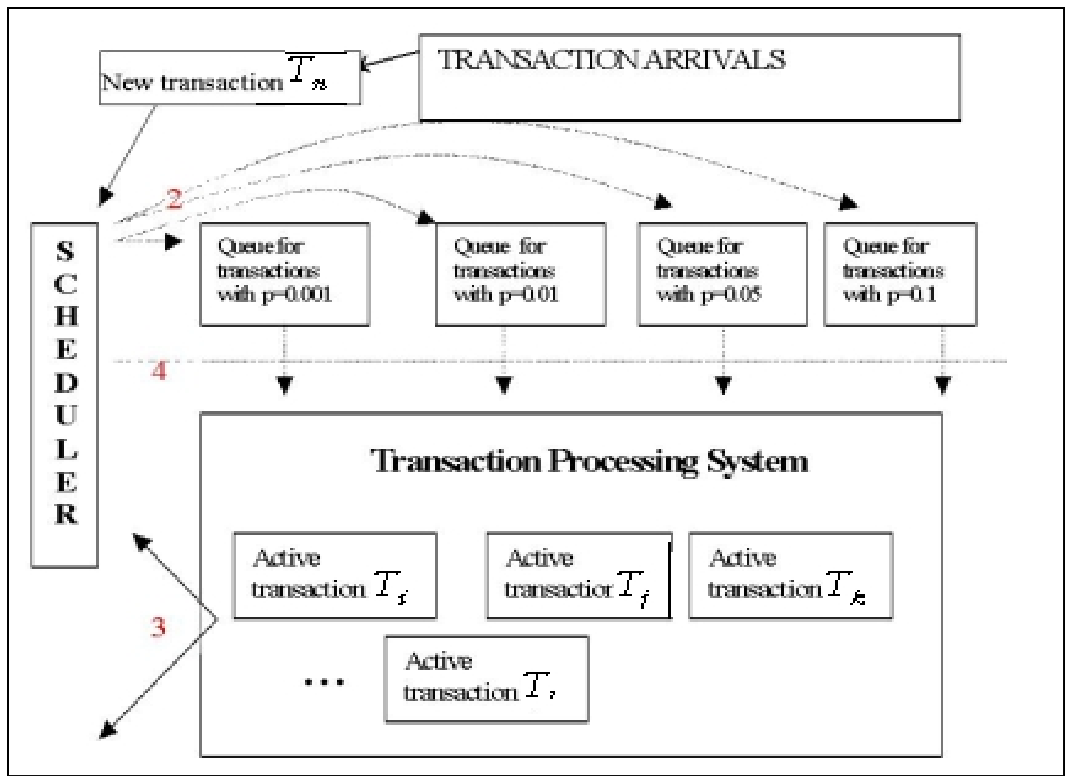



**Figure 4.1. A diagramatic illustration of the contention-based scheduler**

As indicated above, in order to determine each transaction's contention the contention-based scheduler does a pre-fetch to measure a transaction's contention-size. In order to be viable, this pre-fetch needs to be effected at a minimal cost. This can be accomplished by implementing the pre-fetch as a memory only operation. Here, a transaction does a pseudo-execution. If a required item is found in cache its contribution to contention is calculated. If an item is not found in cache, its contribution to contention is assigned a default value based on the average contention of objects found on disk.

Where both the number and contention of objects that need to be acquired from disk is unknown because the choice of object is dependent on predicate satisfaction, the estimation of both the number and contention of the objects can be assigned a default value. One heuristic for determining this default could be that the where the number of objects required from disk is unknown, this number could be set by dividing the number of objects a transaction has acquired from cache by the proportional hit rate of objects found in cache.

For example, let us assume that in a system with a cache hit-ratio of 0.625, a transaction that has acquired 16 objects from cache requires an unknown number of objects from disk. Under the suggested heuristic, the 16 objects acquired from cache would be divided by 0.625 thus determining the approximate number of objects required from disk. Since in general, all high contention items are in cache, any imprecision in assigning a default contention to items not found in cache is likely to be of minimal consequence. The robustness of the contention-based scheduler to imprecision will be shown in a later chapter when the results for the performance of the contention-based scheduler are presented.

## 4.3 Scheduling Heuristics

In section 4.1 we showed that the contention-based scheduler sorts transactions into queues by their contention. What we did not determine was how to measure each transaction's contention in a system with variable sized/contention transactions and how

to determine the appropriate number of each transaction type to allow into the system. In this section, we present a set of equations to measure transactions' contention and some algorithms for determining the maximum numbers of each transaction type to allow into the system.

### 4.3.1 The Measurement of Contention

In order to determine how many transactions of each type to allow into the system, the contention based-scheduler needs to be able to measure contention. While there are several equations that measure contention such as those found in [15], [47] and [48] all of these have deficiencies Vis a Vis our requirements. In [47] and [48] the measurement of contention is dependent on the level of concurrency while for our requirements we need a measure of contention that is independent of concurrency. As well, all the equations that we have seen, assume that data only comes from one data store and that all objects in that data store have an equal probability of being accessed. However, in typical applications, not all data objects are accessed with equal probability. The equations presented here address these problems. Our equation for measuring any transaction class $T_a$'s contention is -

$$p(T_a) = S\sum_{k=i}^{j} A(O_k) \tag{4.1}$$

Here $O_i$ to $O_j$ are the objects that transaction $T_a$ requires, $A(O_i)$ to $A(O_j)$ are each object's probability of being required in any access and $S$ is the mean size of all

transactions in the system. In a database where all objects are accessed with the same probability, for any object, its $A$ is simply $1/D$ where $D$ is the size of the database. In a database where objects do not have the same probability of being required in an access, each object's $A$ is determined historically. Alternatively, a database can be partitioned into groups of objects. Here, where $D_x$ is the group to which objects $O_i$ belongs and $k$ is the probability that in any access the object required will come from $D_x$, an object $O_i$'s $A$ can be calculated by -

$$A(O_i) = k / D_x \qquad (4.2)$$

Equation 4.2 assumes that each object in a group has the same probability of being required in an access as any other object in that same group and that each object in a group has a different probability of being required in an access to any object in a different group.

To illustrate our equations, let us assume a database where there are two data stores- $D_1$ with 1000 objects. Let us further assume that the system consists of four transaction types -transactions of type $T_1$ which represent 20% of transactions and require 1 object from $D_1$ and 3 objects from $D_2$, transactions of type $T_2$ which represent 20% of transactions and require 2 objects from $D_1$ and 6 objects from $D_2$, transactions of type $T_3$ which represent 35% of transactions and require 4 objects from $D_1$ and 12 objects from $D_2$ and transactions of type $T_4$ which represent 25% of transactions and require 8 objects from $D_1$ and 24 objects from $D_2$.

Since one in four accesses is to an object in $D_1$, $k$ for $D_1$ is 0.25 and thus for any object $O_i$ in $D_1$, by equation 3.2, $A(O_i)$ evaluates to –

1/(1000/0.25)=0.00025.

By similar calculations, $A(O_j)$ for $D_2$ is 0.00000075. The average size of transactions, $S$, is 16. Thus, the contention of transactions of type $t_1$, $p(T_1)$ evaluates to –

(16*(1*0.00025))+(16*(3*0.000000075))= 0.0040036

By similar calculations $p(T_2)$ evaluates to 0.0080072, $p(T_3)$ evaluates to 0.0160144 and $p(T_4)$ evaluates to 0.0320288.

Thus, besides allowing for the calculation of contention by transaction class, equations 4.1 and 4.2 allows us to calculate contention in systems containing sets of objects with different access probabilities such as databases with hotspots. In our example $D_2$ is the standard set of objects while $D_1$ is the set of objects that are hotspots.

Our estimation for the first cycle throughput of any transaction type, say transaction class $T_a$ ,is-

$$(n\, r(T_a))(1 - p(T_a)\, /2)^{(n-1)} \qquad (4.3)$$

This modifies the equation in [15 ] by adding the expression $r(T_a)$ and replacing mean contention $p$ ,with $p(T_a)$ . Here, $r(T_a)$ is the proportion of transactions that belong

to class $T_a$ and $p(T_a)$ is, as per equation 4.1, the contention of each transaction belonging to transaction class $T_a$. As in [15 ], $n$ is the level of concurrency. These changes allow for prediction of throughput by transaction type rather than in aggregate. Thus for example, given the transaction model used to illustrate equations 4.1 and 4.2 and given a concurrency of 70, the predicted throughput of $T_1$ transactions in the first cycle is

$$(70 \times 0.2) \times (1-0.0040036/2)^{(70-1)} = 12.$$

By similar calculations, at this concurrency, the estimated first cycle throughput of $T_2$, $T_3$ and $T_4$ transactions is 11, 14 and 6 respectively and total throughput is thus 43.

### 4.3.2 Scheduling by Transaction Type

The algorithms presented in this section are heuristic and while they improve performance, they are not necessarily optimal. There are two reasons for this. Firstly, in systems where transactions vary in size or contention, there is a problem in determining the contention at which peak throughput occurs since each transaction type reaches its peak at a different concurrency level[5]. The second reason our algorithms are heuristic is because while the contention-based scheduler may set the upper limits for the number of any transaction type allowed into the system, there is no guarantee that there will always be sufficient numbers of a particular transaction type to satisfy this upper limit. That is,

---

[5] In systems with homogeneous transaction types there is a regular relationship between aggregate contention and peak throughput in each cycle.

some transaction types may be cleared at a faster rate than their permitted level in the system.

Given these limitations, we present 2 heuristic algorithms. Our first algorithm is as follows. Firstly we determine each transaction class 'contention as per equation 4.1 as well as the mean system contention. We then determine the notional system concurrency at an aggregate contention of 2.6 given the mean contention. That is, where $m$ is the mean system contention, notional system concurrency $C_n$ is –

$$C_n = 2.6/m \tag{4.4}$$

The reason for choosing 2.6 as the aggregate contention is that it approximates the total contention at which throughput peaks. The derivation of this peak aggregate contention is dealt with in more detail in a subsequent subsection.

In our next step we calculate each transaction class 'contention as if that class was the only transaction type in the system. For any transaction type $T_a$ our approximation of transaction type $T_i$s' "isolated" contention is –

$$I(T_a) = p(T_a)(p(T_a)/m) \tag{4.5}$$

For each transaction type $T_a$ , where $r(T_a)$ is the proportion of all transactions that naturally belong to $T_a$ (as in equation 4.3) we define $n(T_a)$ as-

$$n(T_a) = r(T_a)C_n \tag{4.6}$$

Next, we calculate each transaction class' $U$ by dividing its $I$ as derived in equation 4.5, by the $I$ of the lowest contention class. For any transaction class, say $T_a$, we then define its $B$ as-

$$B(T_a) = U(T_a)n(T_a) \tag{4.7}$$

In our final step, we determine the maximum number of each transaction type allowed into the system. For any transaction class $T_a$ in a system with transaction classes $i$ to $j$, the maximum number of transactions of a class $T_a$ allowed into the system is –

$$Q(T_a) = (\sum_{k=i}^{j} B(T_k)\ r(T_a)) /\ U(T_a) \tag{4.8}$$

We use the transaction model outlined in section 4.3.1 to illustrate equations 4.1 and 4.2 to illustrate the calculation of the maximum number of each transaction type allowed into the system by our first algorithm. As shown in section 4.3.1, the contentions of the transaction classes in our illustration model are around 0.004, 0.008, 0.016 and 0.032 for transaction classes $T_1$, $T_2$ , $T_3$ and $T_4$ respectively. Given the natural proportion of each transaction class as outlined in section 4.3.1, $m$ evaluates to 0.016 and thus, under equation 4.4 $C_n$ =2.6/$m$ =162.5.

By equation 4.5, $I(T_1)$ = 0.004*(0.004/0.016) = 0.001 and by similar calculations we get $I(T_2)$ =0.004, $I(T_3)$ = 0.016 and $I(T_4)$. Since $T_1$, $T_2$ , $T_3$ and $T_4$ transactions represent 20%, 20%, 35% and 25% of transactions respectively and since by equation 4.4

$C_n$ evaluates to 162.5, by equation 4.6, $n(T_1)$ = 162.5*0.2=32.5 and by similar calculations $n(T_2)$ = 32.5, $n(T_3)$ =56.875 and $n(T_4)$ =40.625. For our illustration model $U(T_1)$=1, $U(T_2)$=4, $U$ $(T_3)$=16 and $U(T_4)$ =64. Thus, by equation 4.7 $B(T_1)$ =1* 32.5=32.5. Similarly, $B(T_2)$= 130, $B(T_3)$= 910, $B(t_4)$= 2600 and $\sum_{k=1}^{4} B(T_k)$ =3672.5. Finally, by equation 4.8, $Q(T_1)$=( 3672.5*0.2)/1=734.5 while $Q(T_2)$= 183.5, $Q(T_3)$= 80.4 and $Q(T_4)$= 14.34375.

We now use our running transaction model, algorithm 1 and equations 4.1 to 4.8 to illustrate the predicted effect of our scheduler on throughput. Because each transaction has the same proportion of accesses to each of the two data stores, the frequency of access under our scheduling is unchanged from that derived in section 4.3.1 and remains 0.00025. However, because of the disproportionate number of low contention transactions allowed into the system, the mean transaction size, $S$, is changed from the 16 derived in section 4.3.1 to 6 under our scheduler. This changes the contentions of each transaction class to 0.0015, 0.003, 0.006 and 0.012 for transactions $T_1$, $T_2$, $T_3$ and $T_4$ respectively. As a result, by equation 4.3, the estimated first cycle throughputs of each transaction class are 344, 41, 4 and 0.3 for transaction types $T_1$, $T_2$, $T_3$ and $T_4$ respectively. This is against the first cycle throughputs without the scheduler which, at a concurrency of 70, in section 4.3.1 had an estimated throughput of 12, 11, 14 and 6 for transaction types $T_1$, $T_2$, $T_3$ and $T_4$ respectively.

Our second algorithm is simpler and is constructed as follows. Firstly we derive $I$ for each transaction class as per equation 4.5. For each transaction class, we divide the aggregate contention of 2.6 by that transaction class' $I$. This gives a notional peak concurrency for each transaction class if it were the only transaction class in the system. Finally, we multiply each transaction class' peak by the proportion of its natural representation in the system giving the maximum number of transactions of a particular class allowed into the system. Thus, for any transaction class $T_a$, the maximum number of $T_a$ transactions allowed into the system is –

$$Q(T_a) = (2.6 / I(T_a))\ r(T_a) \qquad (4.9)$$

Thus, under our second algorithm, $Q(T_1)$=520, $Q(T_2)$=130, $Q(T_3)$= 57 and $Q(T_4)$=10.

### 4.3.3 Setting Transactions priority

While the algorithms presented above may increase aggregate throughput, higher contention transactions "subsidize" lower contention transactions. To compensate the higher contention transactions for their decreased success rate, we institute a priority system such that a transaction with a higher contention has a higher priority than a transaction with a lower contention. Thus, if a $T_4$ transaction conflicts with a $T_3$ transaction, it is the $T_4$ transaction, which obtains the lock. If the conflict occurs after the $T_3$ transaction has gained its lock, then the $T_3$ transaction is rolled back to the point where it acquired the lock on the disputed item and the $T_4$ transaction is given the lock. Similarly,

$T_3$ transactions have priority over $T_1$ and $T_2$ transactions and $T_2$ transactions have priority over $T_1$ transactions.

The effect of this prioritization is to increase the success rate of higher contention transactions without dramatically affecting the success rate of lower contention transactions. Consequently, the throughput of each transaction class under the scheduler should either increase or be close to the throughput of that transaction class under systems without the scheduler. The only difference between conventional rollback and the rollback suggested in this chapter is that with the contention-based scheduler a transaction's priority would be determined by its contention. This requires automatic checkpointing by the DBMS such that a checkpoint is automatically inserted into the transaction by the DBMS every time a new object is acquired. As well, to avoid problems associated with restoring rolled back values, all a transaction's updates are done in its own workspace and are not applied to the actual data until committal. Thus, a rollback by a transaction only involves a move back to the checkpoint and does not require restoring original object values[6].

### *4.3.4 The Determination of Aggregate contention*

As indicated in section 4.3.2, an important part of our algorithms is determining the aggregate contention at which throughput peaks. This is quite problematic since firstly, as indicated in the previous chapter, past a certain threshold, throughput per cycle falls in

---

[6] It should also be noted that rollback violates one of the principles of 2PL – that is, that all locking is done in one phase and unlocking in another. However, rollback is so widely used, that we do not consider this violation to be a serious problem to our algorithm.

every successive cycle and the rate at which it falls increases with concurrency. This results in total throughput peaking at different concurrencies/aggregate contentions depending on the number of cycles run.

This is illustrated in Figure 4.2 below, which is a distillation of the graphs presented in the previous chapter and shows both actual peak contentions as well as regressions on the mean using the power and log regressions built into the "EXEL" spreadsheets charting facilities. Here, in every cycle aggregate throughput at every concurrency is calculated and the concurrency at which throughput is highest recorded. This concurrency is then divided by contention per transaction giving the aggregate peak contention. The contentions used are calculated in accordance with equation 4.1. Thus, the tests using only $T_2$ transactions have a contention of around 0.004 per transaction, the test using only $T_3$ transactions have a contention of around 0.016 per transaction and the tests using only $T_4$ transactions have a contention of around 0.064 per transaction.

As Figure 4.2 shows, for fixed size transactions, whatever the contention per transaction, there is quite a high level of consistency in the aggregate level contention at which peak throughput is achieved. However, this peak aggregate level of contention changes with every cycle. Thus the most desirable level of concurrency/aggregate contention depends on the nature of the system. If the system operates with transactions arriving in packets that can be cleared before the arrival of a new packet of transactions then a level of aggregate contention of around 2.6 (the peak in the first cycle) is desirable. If, on the other hand, the number of available transactions allows for a long period of

continuous processing then a lower level of contention is desirable. In our algorithm we arbitrarily opt for the peak in the first cycle.

It should be observed that while the algorithms used for the contention-based scheduler are aimed at improving the performance of systems containing transactions with varying contentions, the peak aggregate contention upon which the number of each transaction type allowed into the system is determined, is calculated with reference to fixed sized transactions. This is necessary since in multiple transaction class systems, in any cycle, there is no unique aggregate contention at which the performance of each transaction type is maximized. In general, a high level of concurrency favors lower contention transactions while a low level of concurrency favors higher contention transactions. Thus, the peak contentions achieved with fixed transactions is used as a reasonable compromise.

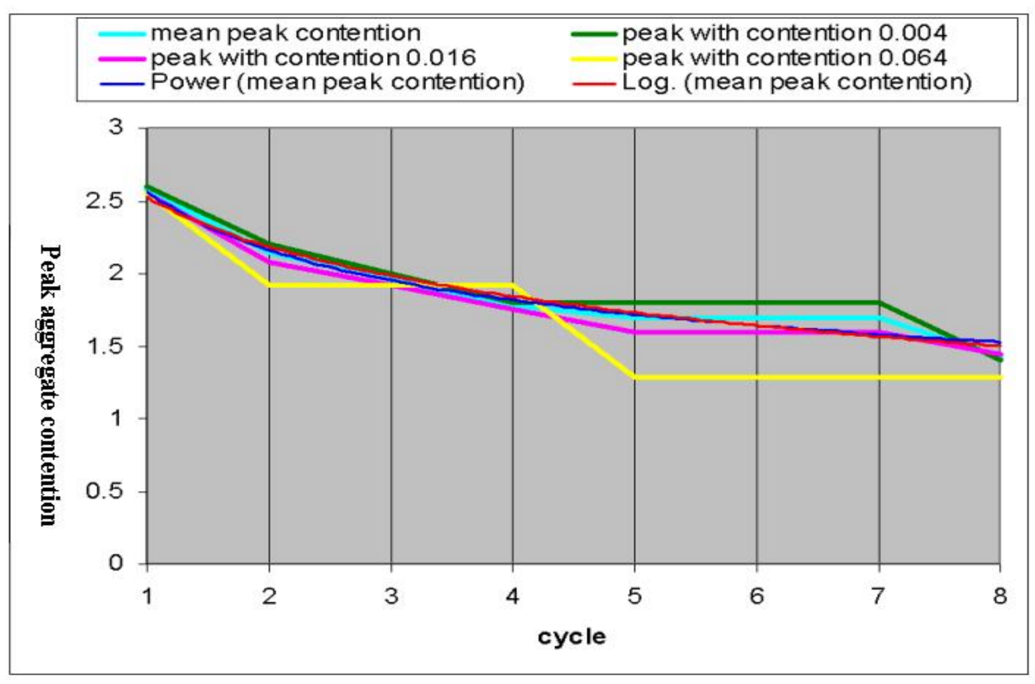



**Figure 4.2. The cyclical relationship between peak aggregate contention and throughput**

## 4.4 Controlling Thrashing

While our scheduling algorithms increase throughput over an extended period of time relative to that achieved under standard 2PL, because they are essentially a front end for a 2PL concurrency control system, they are still susceptible, to thrashing. Thus, in this section we investigate mechanisms to control thrashing. Before outlining our proposal, we first outline the causes of thrashing in a locking system.

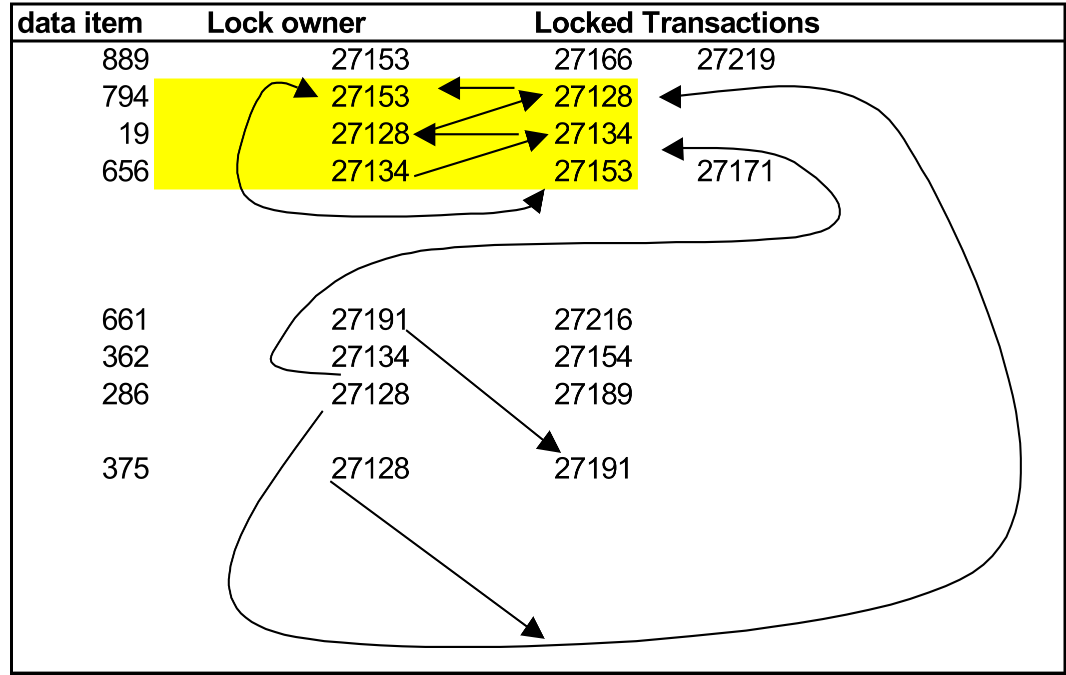
It is well known that thrashing occurs when all transactions are locked so that none can proceed. If deadlocks are unresolved, then the two transactions involved in the deadlock become the root of the thrashing problem as other transactions form a chain behind the two deadlocked transactions with no transaction being able to resume without the deadlock being broken. Traditional 2PL systems have deadlock breaking mechanisms to prevent such situations but they do not have mechanisms to break a circular deadlock involving 3 or more transactions such as illustrated in Figure 4.3.

In Figure 4.3, which is constructed from actual data from one of our tests[7], three transactions are involved in a three-way deadlock. Transaction 27153 is locked by transaction 27134 at data item 656. Transaction 27134 is in turn blocked by transaction 27128 at data item 19 and transaction 27128 is in turn blocked by transaction 27153 at

---

[7] In-memory, 2 billion MIPS per processor at a concurrency of 10.

data item 794. All other transactions form a chain behind these three transactions. The tendency of circular deadlocks of depth 2 and over to occur increases with both contention and concurrency.



| data item | Lock owner | Locked Transactions | |
|---|---|---|---|
| 889 | 27153 | 27166 | 27219 |
| 794 | 27153 | 27128 | |
| 19 | 27128 | 27134 | |
| 656 | 27134 | 27153 | 27171 |
| 661 | 27191 | 27216 | |
| 362 | 27134 | 27154 | |
| 286 | 27128 | 27189 | |
| 375 | 27128 | 27191 | |

**Figure4.3. An illustration of a three way deadlock**

Traditional 2PL systems are generally only equipped to deal with deadlocks involving two transactions thus rendering themselves susceptible to circular deadlocks involving multiple transactions at their root. Limiting concurrency reduces the number of restarts required to eliminate thrashing as well as keeping the cost of processing restarts to memory only processing by enabling access invariance. However, this is an unsuitable solution for our scheduling algorithms whose effectiveness is based on a high level of effective concurrency. Given that a relatively high level of concurrency increases the risk of thrashing, finding an economical way of managing thrashing is required where a high level of concurrency is used.

One possible method of controlling thrashing while allowing a high level of concurrency, is to keep a graph of all conflicts (as in Figure 4.3) and restart only the transactions that are at the root of a deadlock. In Figure 4.3 this would mean restarting transaction 27153 or transaction 27134 or transaction 27128. Theoretically, this process can be extended to an arbitrary depth. However, the cost of navigating to trace the root of any potential circular deadlock incurs a cost. Once the root of a lock chain becomes excessively deep it is cheaper to restart all transactions than it is find the root of the conflict and restart it only.

An alternative thrashing control strategy is to allow the proportion of blocked transactions to reach a threshold and then restart all blocked transactions. The rationale behind this method is the finding in [47] that 2PL systems become unstable once the proportion of blocked transactions exceeds 0.378.

However, keeping the proportion of blocked transactions below 0.378 by restarting the most recently blocked transaction does not ensure that thrashing will not occur. For example, in Figure 4.3, restarting any of the 10 blocked transactions other than transaction 27153 or transaction 27134 or transaction 27128 will not solve the thrashing problem. Thus, if one cannot identify the root of the thrashing problem, the only solution is to restart all blocked transactions.

However even this policy does not ensure that thrashing will not occur. For example, let us assume the situation shown in Figure 4.3. Let us further assume that all transactions are restarted if on restart and obtain their locks in the same sequence as they did originally except that transaction 27153 obtains its lock on item 656 before transaction 27134. With this solution the root of the thrashing problem would be eliminated and if the system proceeded without further interference all the transactions would eventually be cleared.

However, the policy of restarting all blocked transactions once the proportion of blocked transactions exceeds 0.378 would not allow our example system to clear without interference. Until transaction 27153 cleared, all the other transactions would return to their blocked state. The system perceiving that the blocked proportion exceeds 0.378 would then restart all the blocked transactions again. Thus immediate restart is susceptible to a high number of restarts. This example also indicates another problem with restarting all blocked transactions once a threshold is passed - that transactions will be restarted whether or not a circular deadlock exists. That is, transactions are restarted even though if left alone the system would clear.

Two solutions to the problems outlined above are –

1. Increase the proportion of blocked transactions permitted before restarting transactions. Increasing the threshold allows systems that are not involved in a circular deadlocked more time to clear thus reducing unnecessary restarts. The cost of this method is that it allows transactions to remained blocked for an

extended time even when thrashing is inevitable due to the existence of a circular deadlock.

2. Change the sequence that transactions obtain their locks thus reducing the probability that the same wait chain will recur. One method of doing this is by time-stamping each restarted transaction with the system clock plus a random number between 0 and the processing time required by an average transaction. A transaction marked for restart does not commence until the system clock equals its time-stamp. The cost of this method is that it delays the restart of some transactions thus increasing their response time.

For thrashing control used in conjunction with our scheduling algorithms we also suggest the following modification - instead of setting a system wide threshold of blocked transactions, we set this proportion separately for each transaction class. That is, if a threshold of 0.356 is set for blocked transactions, if the proportion of blocked $T_4$ transactions exceeds 0.356 and all other transaction types are inside their thresholds, then only blocked $T_4$ transactions are restarted. We do this since for systems using our contention-based scheduler, our priority system makes the wait chains of higher contention transactions independent of the lower contention transactions in the system. Thus there is no sense in restarting high contention transactions if the low contention transactions exceed their threshold since the restart of the low contention transactions will have no effect on the higher contention transactions.

## 4.5 Summary

The primary purpose of this chapter was to introduce the contention-based scheduler. This scheduler operated by measuring transactions' contention as they arrived sorted these transactions into queues of transactions with a similar contention and then manipulated the number of transactions allowed into the system by the contention class. For its operation, the contention based scheduler needed-

1. A mechanism that could enable it to estimate transaction's contention reasonably cheaply. In section 4.2.1 we developed this mechanism via a pseudo in-memory access of data.
2. A method for calculating transactions' contention that was independent of concurrency and that could calculate contention in situations where not all objects in data stores have an equal probability of being accessed. We presented equations that addressed this problem in section 4.3.1.
3. A method for determining the maximum number of transactions of each type allowed into the system. As indicated in section 4.3.2, any method for determining the maximum number of transactions of each type allowed into the system is necessarily heuristic rather than deterministic. Accordingly, the methods presented in section 4.3.2 were heuristic.
4. A method to compensate the higher contention transactions for their subsidy to lower contention transactions. In section 4.3.3 we developed a priority system such that a transaction with a higher contention had a higher priority than a transaction with a lower contention in lock acquisition.

5. While the contention-based scheduler substantially improved performance, it did not address the problem of thrashing to which 2PL based systems are susceptible. Accordingly, in section 4.4 we analyzed the causes of thrashing and proposed some simple mechanisms for dealing with it.

# *Chapter 5*

## *Enhanced Memory Access*

### 5.1 Introduction

In chapter 3 we demonstrated that in-memory systems substantially outperform disk-based systems. Thus, it seems that for those organizations that can fit their data into memory, the best way to improving the performance of their transaction processing systems lies in switching to in-memory systems rather than improving the performance of their disk-based systems.

However, for those organizations that have very large databases that cannot fit into memory, the option-of an in-memory database is not viable. For these organizations, the only way of increasing throughput is by improving the performance of disk-based concurrency control. In chapter 4 we presented the contention-based scheduler as a way of improving the performance of disk-based systems. This scheduler accepted the generally held view that there is a tradeoff between access invariance and high levels of concurrency. In this chapter we challenge this assumption and examine how a modified variation of access invariance that we call enhanced memory access, (EMA) can be used to allow very high levels of concurrency in the pre-fetching of data and how this pre-fetching can yield close to in-memory performance in disk-based systems.

## 5.2 An Overview of the Principles of EMA

EMA is an extension of the concept of pre-fetching as presented in [17]. Here, all transactions are executed without concurrency control to pre-fetch their data into memory. Once transactions have been pre-fetched, their committable execution can be effected entirely in memory. Figure 5.1 below, illustrates the basic operation of a pre-fetching system.

**Committable Execution**

| | Processor | Processor | Processor | Processor | Processor | Processor | Processor | Processor | Processor | Processor |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| Transaction | $T_1$ | $T_2$ | $T_3$ | $T_4$ | $T_5$ | $T_6$ | $T_7$ | $T_8$ | $T_9$ | $T_{10}$ |

**Pre-fetching Transactions Performing Disk Access**

| $T_{11}$ | $T_{12}$ | $T_{13}$ | $T_{14}$ | $T_{15}$ | $T_{16}$ | $T_{17}$ | $T_{18}$ | $T_{19}$ | $T_{20}$ | $T_{21}$ |
|---|---|---|---|---|---|---|---|---|---|---|
| ... | | | | | | | | | | |

**Pre-fetching Transactions Waiting for Memory Time**

| $T_i$ | $T_j$ | ... | $T_k$ | |
|---|---|---|---|---|

**Figure 5.1. A Sample state for a pre-fetching system.**

In Figure 5.1, transactions $T_1$ to $T_{10}$ have successfully pre-fetched and are engaged in committable execution. While these transactions are processing, no memory time is available. However, those transactions that are engaged in disk access to pre-fetch their data can continue in their efforts. Should any of transactions $T_1$ to $T_{10}$ be blocked, then memory time is made available to those transactions requiring such time for their pre-fetching. If any of transactions $T_1$ to $T_{10}$ completes, then additional memory time is made available to those transactions requiring such time for their pre-fetching – unless there are transactions that have completed their pre-fetching, say transactions $T_{11}$ to $T_{21}$, that can take the place of the committing transactions and begin committable execution

themselves. Given a sufficient number of available transactions, such a system can yield throughputs near those achieved by in-memory systems.

The major impediment to the success of the process outlined above is that, as indicated in previous chapters, at high concurrencies, access invariance does not hold. That is, at high concurrencies, there is a high probability that the conditions satisfying predicates which determine what data is pre-fetched will change by the time a query is ready to embark on a committing execution. Thus data that has been pre-fetched no longer satisfies the required predicates consequently requiring disk accesses. For this reason, where pre-fetching is used, the number of items pre-fetched is limited.

The basis of our proposal for EMA is to allow pre-fetching to the maximum limit physically possible rather than ensuring that conditions satisfying a predicate at pre-fetch time do not change between pre-fetch time and committal execution. Further, EMA ensures that even where conditions do change, the data required to satisfy a predicate are also found in memory. While there are several policies and mechanisms that are required for a full implementation of EMA, the basic policy that needs to be implemented is that no data item can be flushed from memory unless the time-stamp of its last access exceeds the timestamp of the oldest transaction in the system. This basic principle is illustrated in Figure 5.2 below - a more detailed exposition of the required mechanisms will be provided in a later section.

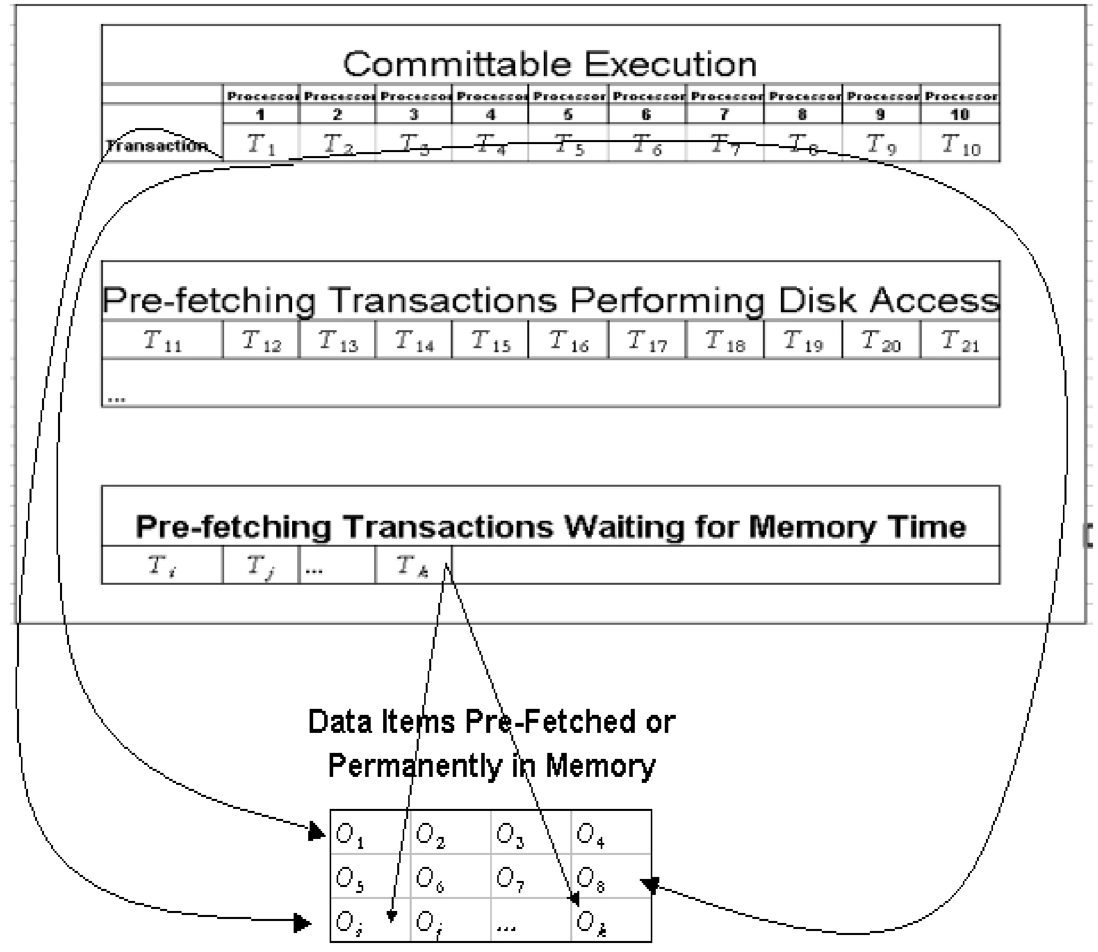



**Figure 5.2. An illustration of the basic principle behind EMA.**

In Figure 5.2, $T_k$ is the newest transaction entering the system - it requires objects $O_i$ and $O_k$. Object $O_i$ is already in memory having been pre-fetched by transaction $T_1$. Thus, transaction $T_k$ merely updates the time-stamp on object $O_i$ and both pre-fetches and timestamps object $O_k$. Let us assume that transaction $T_1$ is committing. On committal, it changes the time-stamps and values of the objects that it has used ($O_1$, $O_8$ and $O_i$). For example, this update changes the timestamp and value of object $O_i$ to reflect that the committal by transaction $T_1$ was the last access to it. Naturally, a committing transaction is the only transaction that can change the value of a data item that is

available to all transactions. All other transactions can only modify copies of objects in their own workspace.

Object $O_i$ cannot be flushed from memory until transaction $T_k$ completes since its update time-stamp is newer than transaction $T_k$'s time-stamp. Thus, when transaction $T_k$ begins its committing execution it can access the most recent value of object $O_i$ from memory - (by a committing execution we mean an execution that has the potential to commit rather than merely a pre-fetch execution. We do not mean the act of committal).

The memory requirements of such a system are quite modest relative to the capacity of modern hardware systems. Equation 5.1 below can be used for approximating the maximum possible memory required to support EMA.

$$E = ((F\ n)\ 3\ )\ G \tag{5.1}$$

In equation 5.1, $E$ is memory required, $F$ is the average number of objects required per transaction, $n$ is the permitted concurrency (including pre-fetching) and presuming that the granularity of retrieval into memory is a page, $G$ is the page size (in bytes). To illustrate our equation let us assume that a system has sufficient available transactions to operate at a pre-fetch concurrency of 15000 transactions and that its transactions are homogeneous with each transaction requiring 16 objects. Let us further assume for simplicity that the objects required by each transaction are almost disjoint and that objects are retrieved into memory in pages of 2048 bytes. Given these assumptions,

in our example $F=$16, $n$=15000 and $G=$ 2048. Thus by equation 5.1 $E$ for this system is ((16 x 15000)x 3) x 2048 =1474560000 bytes = 184320000 kb =184.3 mb.

While $E$ indicates the maximum possible memory requirements, in most situations, the amount of memory required is in fact far less. For example, if a system has 7500 hotspot objects permanently in memory and 6 billion objects on disk, and if the objects in memory account for 0.625 of all object access requirements, then the amount of memory necessary to hold all the required data is approximately 71 megabytes. Memory capacities of this order of magnitude are now quite common in home personal computers.

It will be noted that in equation 5.1 we use a constant 3. To illustrate why we use the constant 3, we use the state described by Figure 5.2 and call the time at which this state occurs time $time_0$. At time $time_1$, when transaction $T_k$ is ready to commit, 15000 new transactions have pre-fetched data. However, none of the objects with a time-stamp younger than $T_k$'s has been flushed. Thus, at a pre-fetch concurrency of 15000, a memory space for approximately twice the number of objects required by 15000 transactions is required. As well, since a pre-fetching transaction can only modify copies of data in its private workspace, the total space required is triple the number of objects required by 15000 transactions.

## 5.3 Classification of Update Predicates

While the basic EMA principle described above allows a large proportion of transaction to be processed in memory with a guarantee of integrity, the guarantee of integrity does not extend to all transactions. In this section we investigate and classify transactions by the nature of their predicates, determine any shortfalls, if any, to their execution by the EMA system outlined above. Where shortfalls exist, we suggest remedies.

The first type of transaction we consider is the standard type of update where an object is nominated and if it satisfies the condition specified in the transaction's predicate, then the transaction applies its update otherwise it does not. We call the predicate in such a transaction $predicate_1$. All the requirements for dealing with transactions with predicates of type $predicate_1$ are adequately catered for by the EMA policy as specified above. An example of a transaction with $predicate_1$ using SQL syntax is –

```
update accounts set balance=balance-withdrawal
   where account_number=12345
      and
         balance> withdrawal
```

The second type of transaction we consider is one where a choice is offered and the object best meeting the required criteria is updated. We call the predicate in such a transaction $predicate_2$. An example of a transaction with $predicate_2$ is –

```
update property  set status to 'rented'
where property_id in ('p55', 'p77')
and price =
  (select min(price) from property
        where property_id in ('p55', 'p77'))
```

In this case, both items are pre-fetched. If item $p55$ is chosen as the candidate at pre-fetch time and if as a result of the execution of another committing transaction $p77$ has the lower rental by the time the transaction with the above predicate executes its committing execution, then since the updated data is still in memory, all the requirements for dealing with transactions with predicates of type $predicate_2$ are adequately catered for by the EMA policy as specified so far.

The third type of transaction we consider is similar to $predicate_2$ except that because the number of objects to be evaluated is so large, it is impractical to retrieve and retain them all in memory and so objects are compared two at a time with the unsuccessful one being discarded. We call the predicate in such a transaction $predicate_3$. An example of a transaction with $predicate_3$ (using SQL syntax) is –

*update property set status to 'rented'*

*where property_id in*

*(select min(price) from property*

*where status !='rented')*

In the example above, if the database contained the identities of all rental properties in New York or London, then it would be impractical to retrieve all these properties into memory and retain them. Rather, two properties at a time would be retrieved and the unsuccessful one discarded until such time as the lowest rental property was found. This type of query may cause problems for the EMA method as specified so far. If for example, another transaction rented the property selected before the above transaction began its potentially committing execution, then the pre-fetched property would no longer satisfy the predicate and the system would be forced to re-execute (Note that if another query does not update status but lists a new property that's cheaper than the current property, then no further disk access is required).

One solution to the problem faced by a transaction with a predicate of type $predicate_3$, is for the system to interpret a pre-fetch as one that retrieves several candidates. That is, one ensures that a reasonable range of items are in memory such that if the preferred or several candidates are invalidated by the activity of other transactions, alternative candidates are still guaranteed to be in memory.

The last type of transaction we deal with, occurs where there is a fork determining the choice of items to be retrieved, that is, where the satisfaction of a predicate determines which objects will be retrieved and updated. We call the predicate in such a

transaction $predicate_4$. As an example of a transaction based on a $predicate_4$, consider a caterer choosing a set course for a function that will be either fish based or beef based with the choice being dependant on the relative price of fish and beef. The pseudo code for such a query could be -

*If (Beef price< Fish price)*

*Buy beef, carrots, broccoli, potatoes, and red wine*

*Else*

*Buy fish, lettuce, tomato, cucumber, and white wine.*

In this case, if the price of beef at pre-fetch time is lower than the price of fish, then beef, fish, carrots, broccoli, potatoes and red wine will be fetched into memory. If, before the transaction begins its potentially committing execution, another transaction reduces the price of fish below that of beef, then lettuce, tomato, cucumber and white wine may need to be fetched into memory. A simple solution to this type of predicate is to pre-fetch all the possible options. In the above example this would mean pre-fetching both the fish based and meat based menus.

In terms of memory requirements, the above solution would not be too costly. For example, let us assume -

1. That the average transaction size without any forks is 16 items,
2. 25% of transactions entail forked decisions (a very generous allowance),
3. For all forked decisions the number of items pre –fetched is double when all forks are executed rather than when the best candidates are pre-fetched.

Given these assumptions, 75% of transactions require 16 items and 25% require 32 items. Thus if all forks are considered, the average size of pre-fetched transactions rises from 16 to 20 items. Using equation 5.1, this would increase the maximum total storage required from 184 megabytes to 230 megabytes. This is still well within the capacity of many home personal computers. Besides the extra memory requirements, this solution also increases the time required for pre-fetching simply because more items need to be pre-fetched.

An alternative to the solution posited above is to just to live with the possibility that some transactions may require data to be re-fetched due to invalidation. This option has smaller memory requirements than the preceding solution and is unlikely to significantly increase execution costs.

For example, let us assume as above, that 25% of transactions have forked decisions. Thus for 75% of transactions, once their data has been pre-fetched no more disk access is required. For the 25% of transactions that are forked, the requirement for further disk access only arises if the conditions satisfying their predicate are changed. In our menu example, either the price of fish must fall or the price of beef must rise. Further, the changes must be of sufficient magnitude to change the result of the predicate's evaluation. Table 5.1 below shows some of the possibilities that could occur between the time the fish-beef predicate is pre-fetched and the time the transaction begins its potentially committing execution. Note, that in table 5.1 only 2 of the possibilities shown change the initial pre-fetch predicate's result – that is, only two of the changes would change the menu from a beef based menu to a fish based menu.

| | beef price | fish price | menu |
|---|---|---|---|
| prefetch | 8 | 10 | beef |
| change 1 | none | none | beef |
| change 2 | 9 | none | beef |
| change 3 | 10 | 11 | beef |
| change 4 | none | 9 | beef |
| change 5 | 7 | 8 | beef |
| change 6 | none | 7 | fish |
| change 7 | 11 | none | fish |

**Table 5.1. An example of the possible effects of changes to a transaction's predicate**

Let us be extremely generous and allow the value of a predicate to change between pre-fetch time and the time of actual execution in 50% of cases. Thus at most 12.5% of transactions will require further disk access after pre-fetch (that is half of the 25% of transactions that require forked decisions). Let us again be extremely generous and allow all changes to a predicate value to be of sufficient magnitude so as to invalidate the original pre-fetch. According to [17] in most databases around 0.625 of accesses are to cache resident items. If we apply this proportion to the above example then for the 12.5 % of transactions whose predicate is invalidated between pre-fetch time and potentially committing execution time, 0.625 of the new objects that they require will be in memory as hot spots.

Thus, after pre-fetch, the proportion of accesses that are disk accesses is –

$$(12.5 \times 0.375) \text{ or } (0.125 \times 0.375) = 0.046875.$$

That is, even under the most severe assumptions, over 95 % of data will not require further disk access after pre-fetch.

## 5.4 Mechanics of the Implementation of EMA

In order for the EMA to be viable, it requires a mechanism that can perform the following functions economically –

Determine the youngest transaction in the system at the time a transaction commits,

Determine the oldest transaction currently in the system,

Record the time of the most recent access for any data object,

Flush unwanted data items from memory.

In this section we present one possible implementation of an EMA. Figure 5.3 below, gives an overview of the activity of transactions on entering the system (prior to commencing pre-fetch activity) and how this activity determines the youngest (and in one special case, oldest transaction) in the system at any point in time.

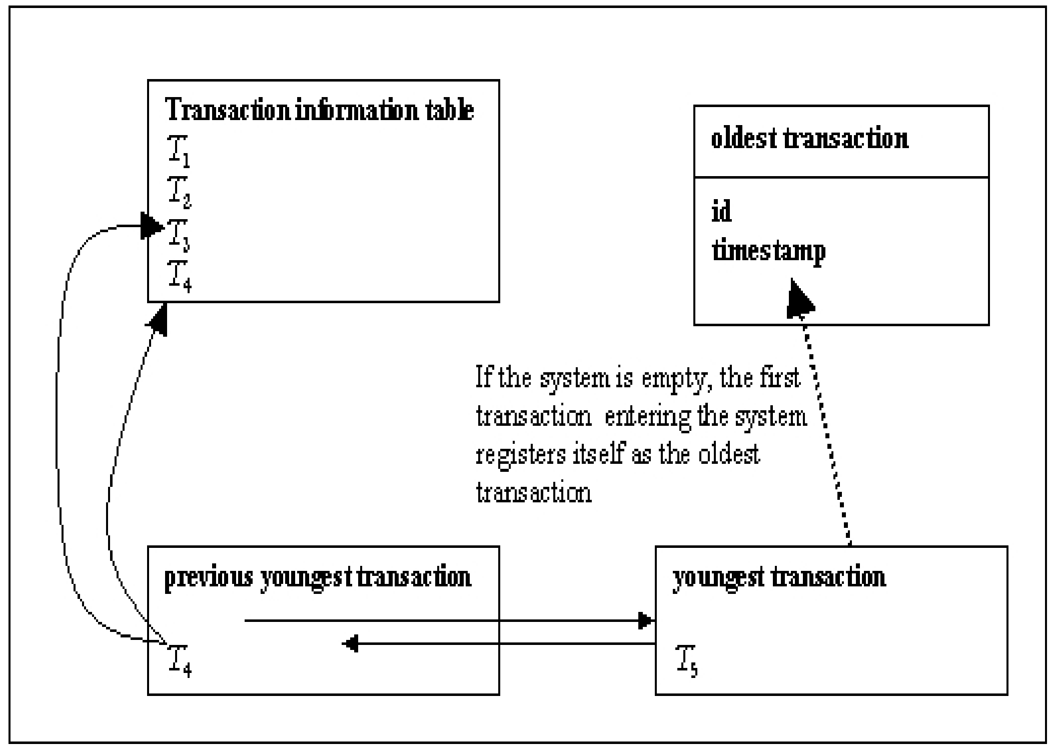



**Figure 5.3: An overview of the activity of transaction on entry into an EMA system.**

As Figure, 5.3 shows, the system has fixed locations for the oldest, the current youngest and the second youngest transactions in the system. If the system is empty, the first transaction registers itself as the oldest transaction and copies itself to the second youngest transaction's location. From then on, each transaction entering the system follows the same procedure. There are 5 steps involved in this procedure. These are –

1. The transaction copies its details to the youngest transaction location.
2. The transaction accesses the second youngest transaction location and registers its information in the older transaction's next youngest transaction slot.
3. The transaction copies the older transaction's details in its own next oldest transaction slot.

4. The transaction copies the older transaction's information to the memory location indicated by its hash address.

5. The transaction copies itself to the previous youngest location.

Thus, each transaction has access to its next and preceding transaction information and can access this information in a decentralized way – that is, no long queues are formed in accessing this information since the only common resource for which a semaphore is required in this phase is that for the youngest transaction location and there are relatively few instructions required before this resource is freed.

The composition of the locations identifying the youngest transaction, second youngest transaction and each transaction's marker in the transaction information table is the same and is shown in Figure 5.4 below. The procedures performed by transactions on transaction markers on completion and exiting from the system are shown in Figure 5.5 and 5.6 below.

id:
Timestamp:
hash address of the owning transaction:

id of next oldest
hash address of next oldest transaction:

id of next youngest
hash address of next youngest transaction:

**Figure 5.4. The composition of transaction markers in the EMA system .**

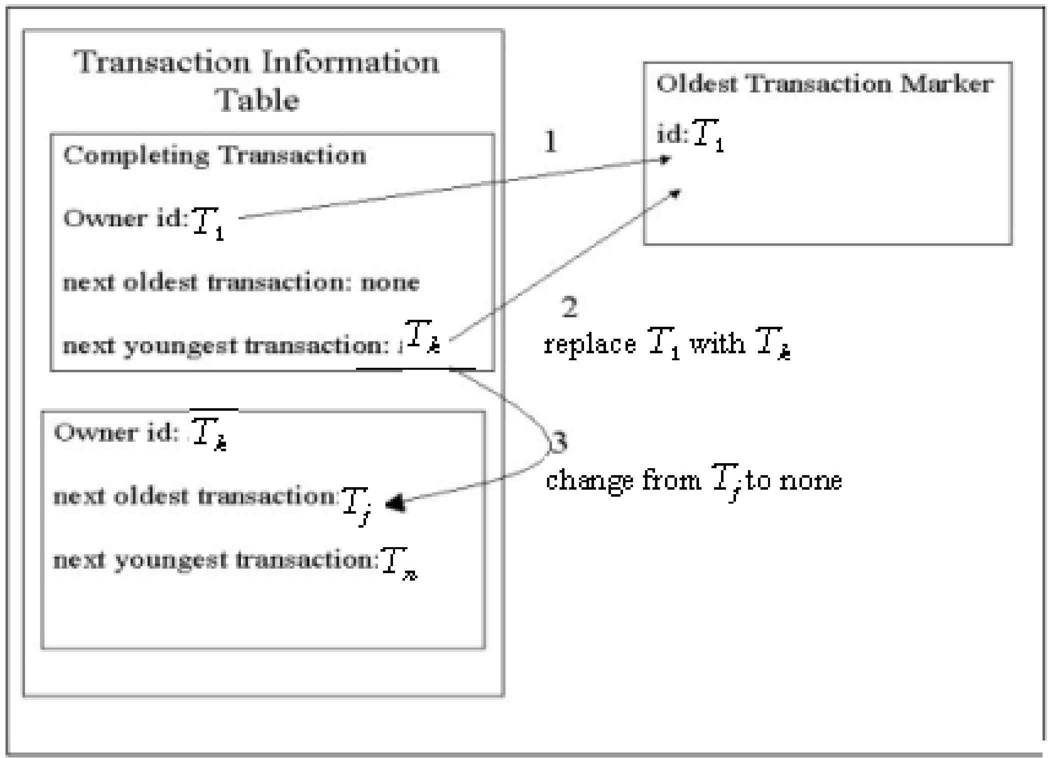



**Figure 5.5. Exit of the oldest transaction**

Figure 5.5 shows the procedure that occurs when the exiting transaction is the oldest in the system while Figure 5.6 shows the procedure that occurs when the exiting transaction is not the oldest in the system. As Figure 5.5 shows, on exiting the system, a transaction, $T_j$, checks the oldest transaction location. It finds that the identity shown on this location is the same as its own (that is, $T_j$ is the oldest transaction). $T_j$ copies the identity shown in its next youngest transaction slot into the oldest transaction location $T_k$ (that is, $T_k$ is now the oldest transaction) and then deletes itself from the transaction information table.

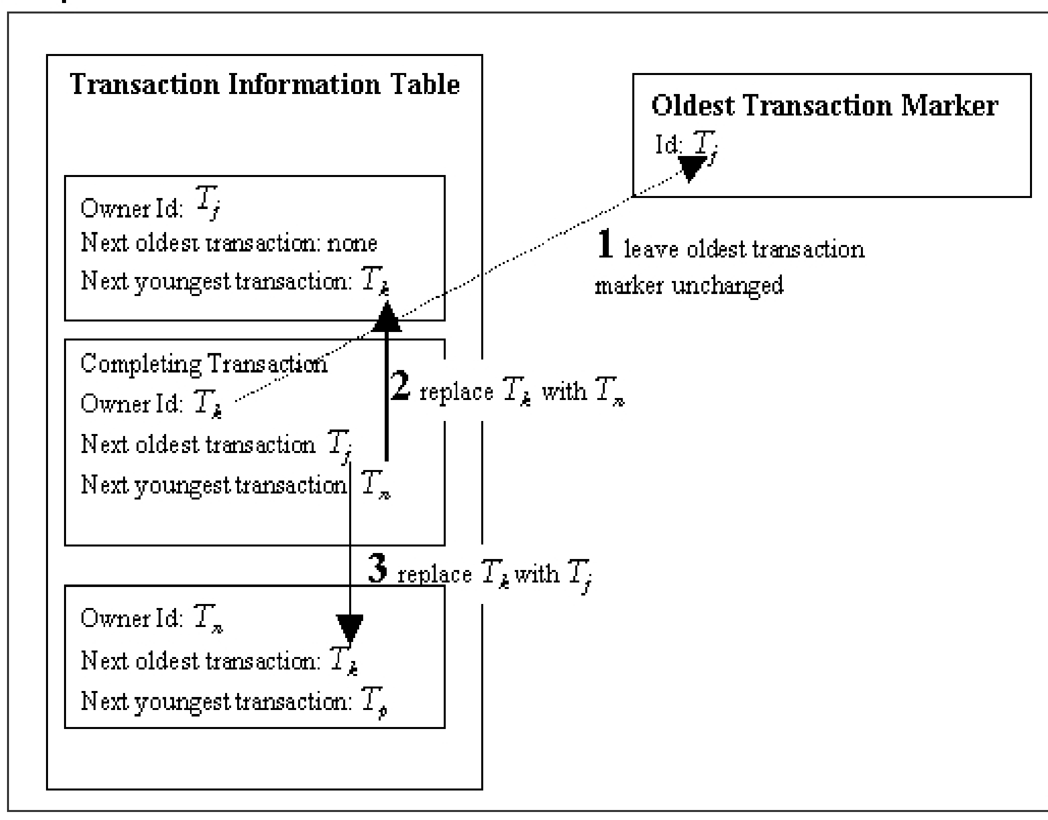



**Figure 5.6. Exit of a transaction that is not the oldest transaction.**

In Figure 5.6, the exiting transaction, $T_k$, finds that it is not the oldest transaction and thus does not change the value of the oldest transaction location. It accesses the transaction marker shown in its older transaction slot -$T_j$, and replaces the identity shown on $T_j$'s younger transaction slot from $T_k$ to $T_n$. It then accesses its own younger transaction - $T_n$ and replaces the identity shown on $T_n$'s older transaction slot from $T_k$ to $T_j$. $T_k$ then deletes itself from the transaction information table. Following the procedures illustrated by Figures 5.3 to 5.6, it is quite economical to identify both the oldest and youngest transactions at any given moment.

### *5.4.1 The Management of Data*

Under our proposed system, the management of data has the following features –

1. A fixed number of memory locations are allocated for the storage of data to data with each location having an overflow.

2. Each data item is time-stamped every time it is accessed for a pre-fetch, data access or committal with the newest access time-stamp overwriting the previous time-stamp. While a data value may be flushed by a non-committing transaction, only a committing transaction can change the value of shared data in memory. In addition, when a transaction completes, it changes a shared data item's time-stamp to that of the youngest transaction in the system.

3. If a new item is pre-fetched from disk and hashes to a memory location that contains no data, then the item along with its time-stamp is written to the memory location. If a new item is pre-fetched from disk and hashes to the same address as an existing item, if the time-stamp of the current data item is older than the time-stamp of the oldest transaction, the current data item is flushed and is replaced by the newer data item. If the time-stamp of the existing data item is younger than the oldest transaction, the new data item is put into an overflow for items hashing to that address. If on hashing to a location the overflow already has more than 1 item, than each of the items in the overflow is tested to see whether they should be flushed.

Figures 5.7 to 5.10 below, illustrate this operation. Figure 5.7 illustrates the procedure followed by a completing committing transaction while Figures 5.8 to 5.10 illustrate the importation of data from disk to memory and the flushing of data from memory. The implementation of EMA outlined in this section meets all the necessary requirements in that it allows for the quick recording and easy maintenance of the identities and time-stamps of the newest and oldest transactions in the system.

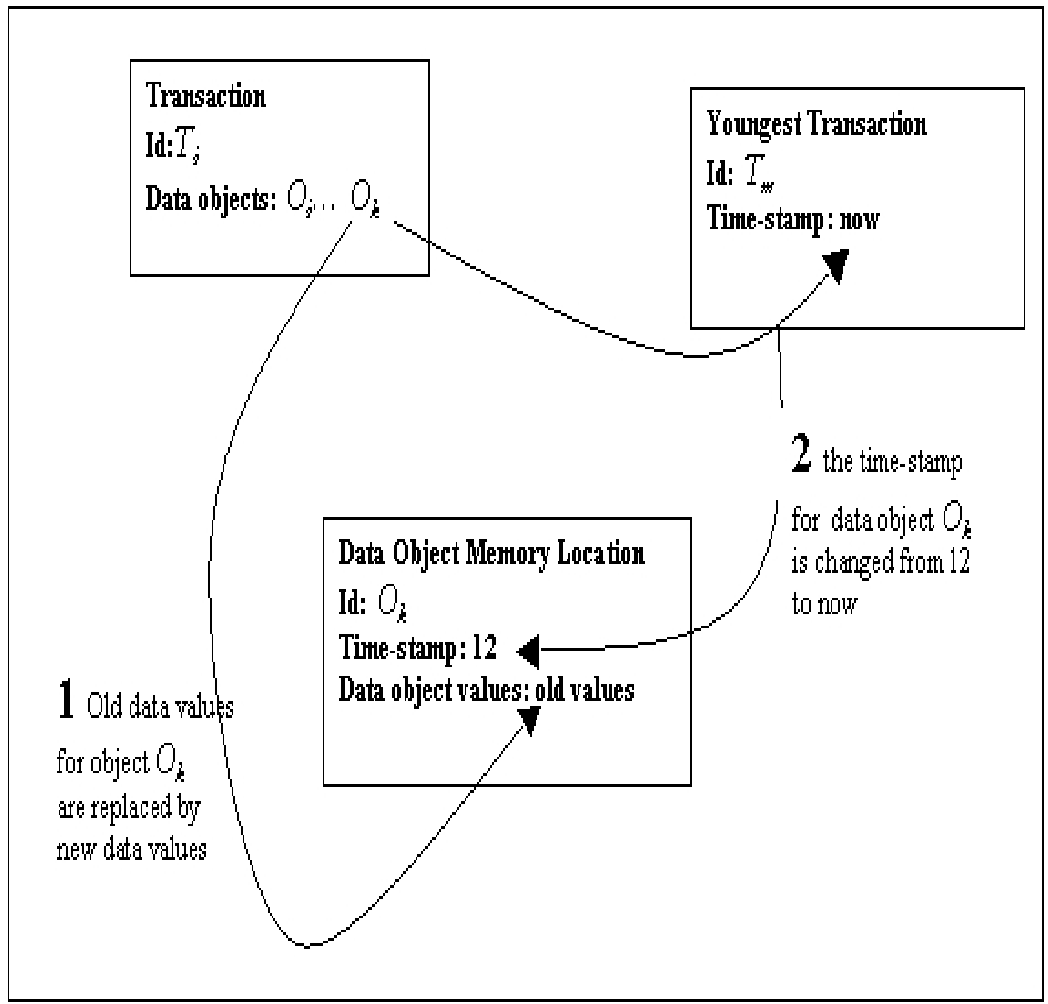



**Figure 5.7. The procedure followed by a completing committing transaction.**

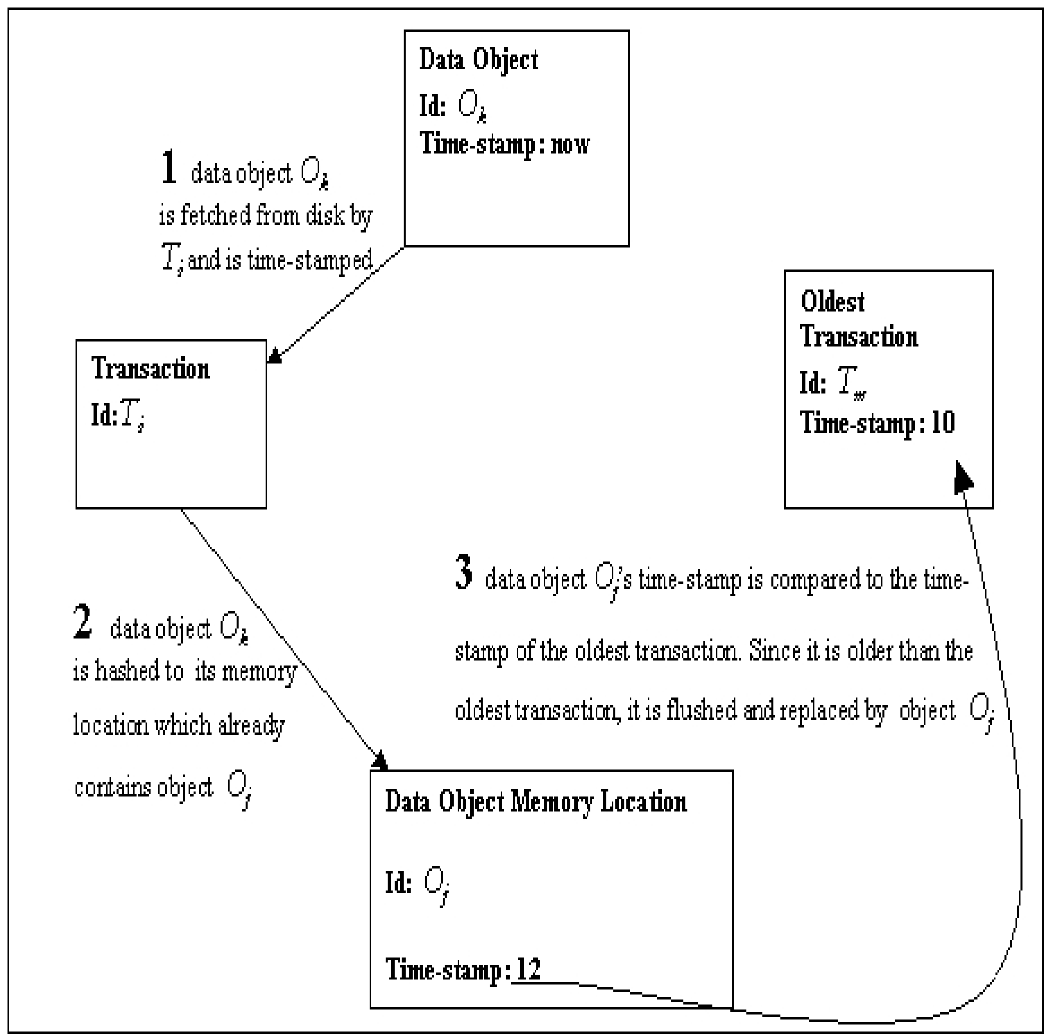



**Figure 5.8. The procedure followed when a pre-fetched item hashes to the same address as an existing item and the time-stamp of the current data item is older than the time-stamp of the oldest transaction.**

Given easy access to these values, the management of data in memory becomes relatively simple. We do not suggest that the implementation of EMA outlined in this section is the only possible implementation or even the best possible implementation. However, it does illustrate that an economical implementation is possible and that consequently, an implementation of EMA is viable.

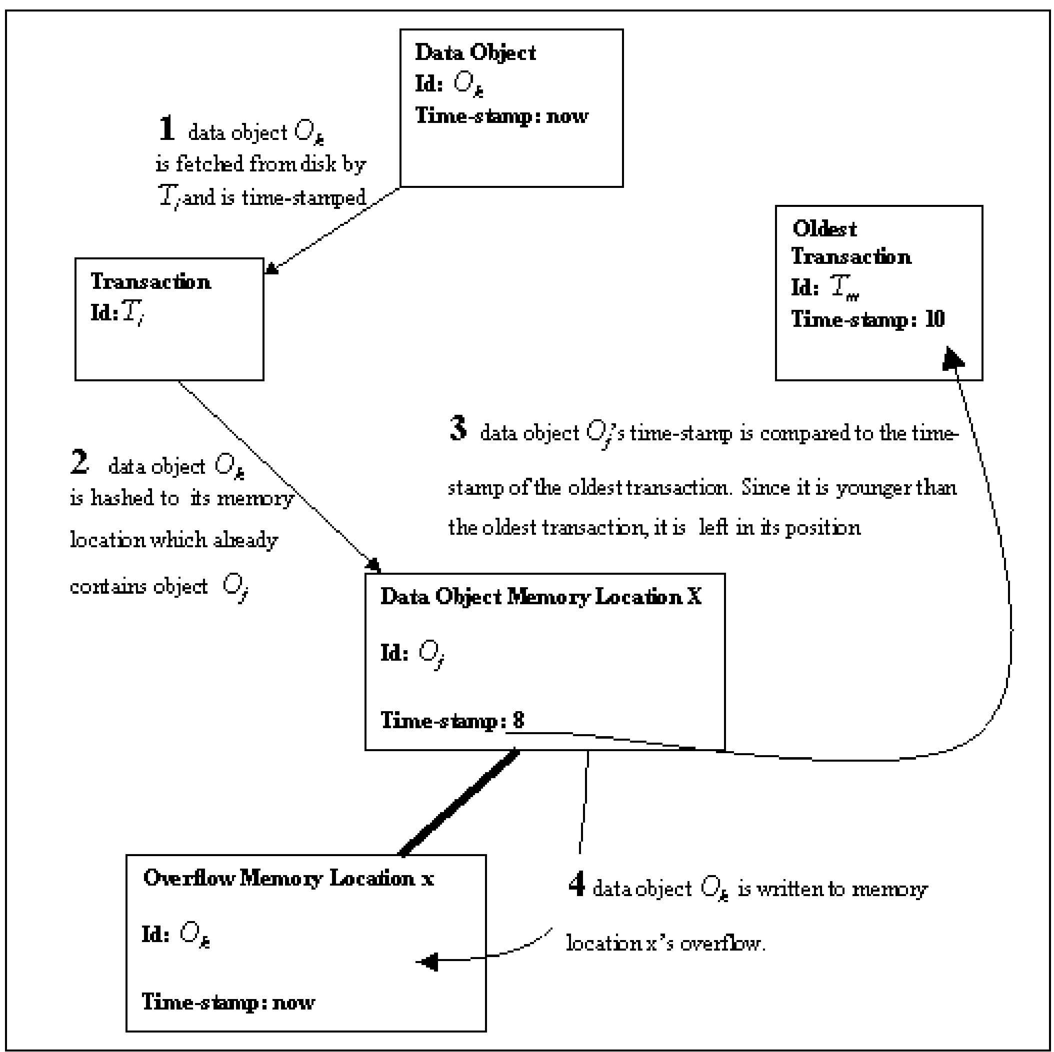



**Figure 5.9. The procedure followed when a pre-fetched item hashes to the same address as an existing item and the time-stamp of the current data item is younger than the time-stamp of the oldest transaction.**

It should be noted that this implementation handles situations where pre-fetched data is required by an executing transaction and situations where pre-fetched items are not used because they fail to meet predicates on execution. In both cases, these items will eventually be flushed when an item is hashed to their location leading to a check of their time-stamp.

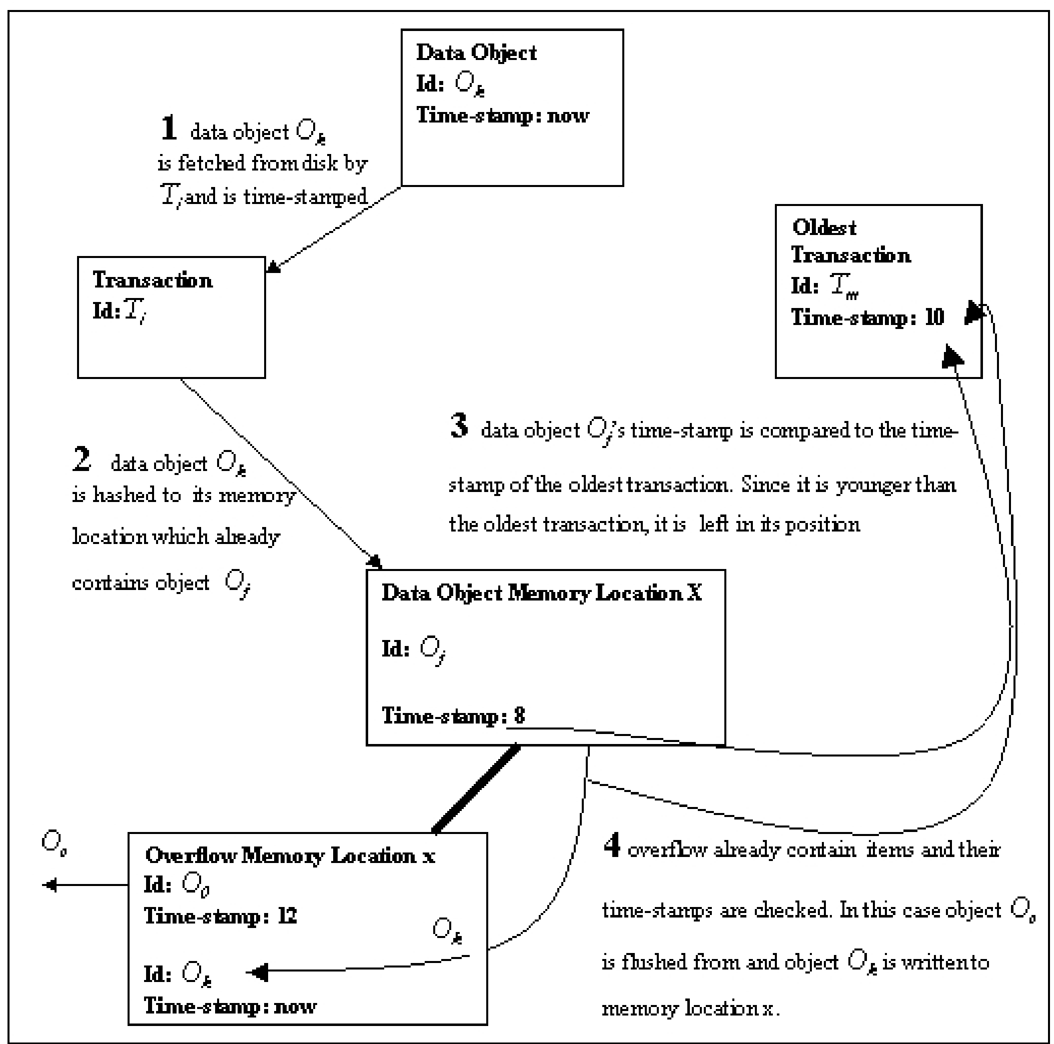



**Figure 5.10. The procedure followed when a pre-fetched item hashes to the same address as an existing item, the time-stamp of the current data item is younger than the time-stamp of the oldest transaction, and overflow already has more than 1 item.**

## 5.7 Summary

The purpose of this chapter was to introduce and test the concept of enhanced memory access. The basis of our proposal for EMA was to allow pre-fetching to the maximum limit physically possible ensuring that when even where the conditions required to satisfy transactions' predicates changed between pre-fetch time and real execution

time, the data required to satisfy transactions' predicates would be found in memory. This required that no data item could be flushed from memory unless the time-stamp of its last access exceeded the timestamp of the oldest transaction in the system. To ensure that required data would not be flushed from memory and that un-required data would be flushed from memory, EMA required a mechanism that could perform the following functions economically –

1. Determine the youngest transaction in the system at the time a transaction commits,
2. Determine the oldest transaction currently in the system,
3. Record the time of the most recent access for any data object,
4. Flush unwanted data items from memory.

We also investigated and classified transactions by the nature of their predicates and outlined policies that would allow EMA to economically deal with transactions that are based on predicate types $\mathrm{predicate}_3$ or $\mathrm{predicate}_3$.

We presented an implementation of EMA that implemented all the requirements outlined above.

# *Chapter 6*

## *Validation of the Contention-Based Scheduler*

### 6.1 Introduction

In this chapter we test the performance of our contention-based scheduling algorithms. In general, the evaluation systems in this chapter are almost identical to the systems used in chapter 3 except that we use the contention based algorithms to determine the number of transactions allowed into the system. That is, the hardware, transaction, database and processing subsystems are the same as outlined in section 4 of chapter 3.

Besides being the same as that used in chapter 3, the transaction subsystem used in this chapter is the same as that used to illustrate the examples in chapter 4. Thus, as in section 4.3.2 in chapter 4, under the first of our scheduling algorithms, algorithm 1, as quantified by equations 4.1 to 4.8, the maximum number of each transaction type allowed into the system is 734, 184, 80 and 14 for transaction types $T_1$, $T_2$, $T_3$ and $T_4$ respectively. Similarly, as in section 4.3.2 in chapter 4, under the second of our scheduling algorithms, algorithm 2, as quantified by equation 9, the maximum number of each transaction type allowed into the system is 520, 130, 57 and 10 for transaction types $T_1$, $T_2$, $T_3$ and $T_4$ respectively.

In section 6.2 we compare the performance of our scheduling algorithms against standard 2PL when the arrival rate of transactions is very high – 20000 transactions per second. Thus the scheduler's performance is not limited by the availability of transactions. As well, in this section we assume that the schedulers can measure the size and contention of transactions with perfect accuracy prior to allowing transactions into the system. In section 6.3 we modify the conditions used in section 6.2 by introducing a margin of error in the schedulers' measurement of the size and contention of transactions. In section 6.4 we modify the conditions used in section 6.2 by restricting the arrival rate of transactions to 1000 transactions per second. In the tests presented in sections 6.2 to 6.4 all systems are run for 1 second. In section 6.5, we compare the performance of contention- based schedulers against standard 2PL when the tests are conducted over an extended time period. In section 6.6 we introduce thrashing control strategies to our scheduler and standard 2PL systems. Because the aim of our scheduling algorithms is to improve the performance of disk-based locking systems, most of this chapter is devoted to

comparing the performance of our scheduling algorithm against standard 2PL. In section 6.7 we compare the performance of systems using our scheduling algorithms to the performance of systems using WDL and optimistic concurrency control.

## 6.2 Performance Results for the Contention-Based Scheduler

The results presented in this section, compare the results of locking systems using our scheduling algorithms against the best results achieved by standard 2PL systems using the same transaction model described in chapters 3 and 4 with transactions arriving at a rate of 20000 transactions per second. Thus the scheduler's performance is not limited by the availability of transactions. As well, in this section we assume that the schedulers can measure the size and contention of transactions with perfect accuracy prior to allowing transactions into the system. Figure 6.1 shows total throughput while Figure 6.2 breaks up total throughput by transaction class. The peak results for the standard 2PL locking method is achieved at a concurrency of 100 in the case of the 1248 processors by 4 MIPS per processor system and at a concurrency of 70 in the other systems.

As Figure 6.1 shows, given a very high arrival rate of transactions and perfect measurement of transactions' contention the improvement in total throughput achieved by our scheduling algorithms over standard 2PL is very large under all hardware configurations. In the massively parallel system, throughput achieved under scheduling algorithm 1 is around 8 times the total throughput achieved by the corresponding standard 2PL system and under algorithm 2, total throughput is over 9 times the total throughput achieved by the standard 2PL system. In the 96 by 100 MIPS system, throughput

achieved under scheduling algorithm 1 is around 11.5 times the total throughput achieved by the corresponding standard 2PL system and around 11 times the total throughput achieved by the corresponding standard 2PL system in the case of scheduling algorithm 2. Under all the other hardware configurations, total throughput under either of our algorithms is around 15 times the total throughput achieved by the standard 2PL system with a slightly better performance being achieved under algorithm 1.

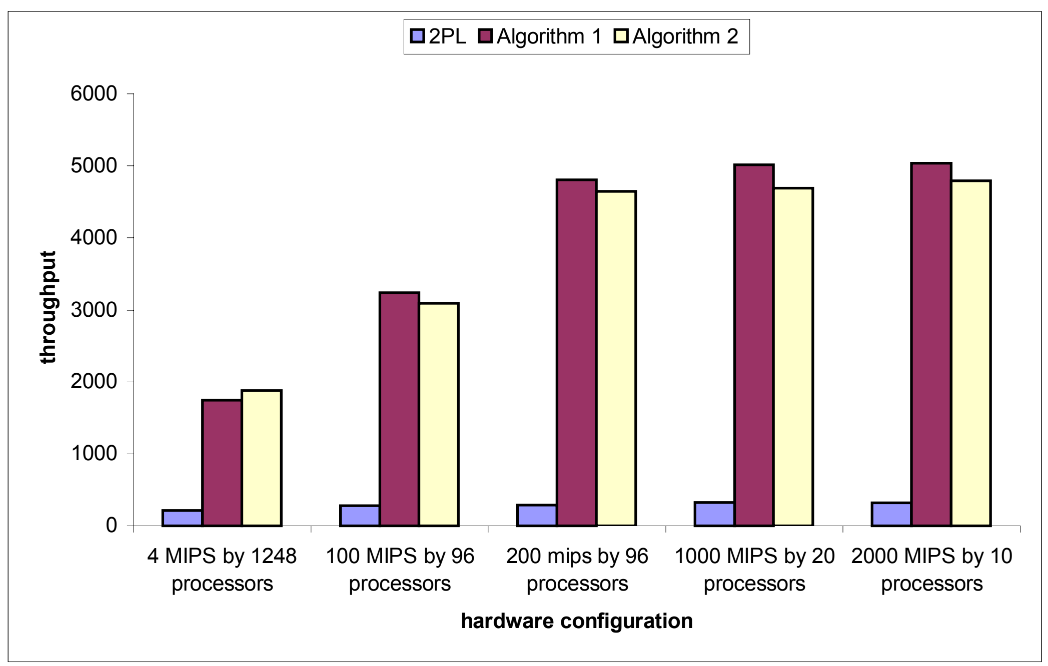
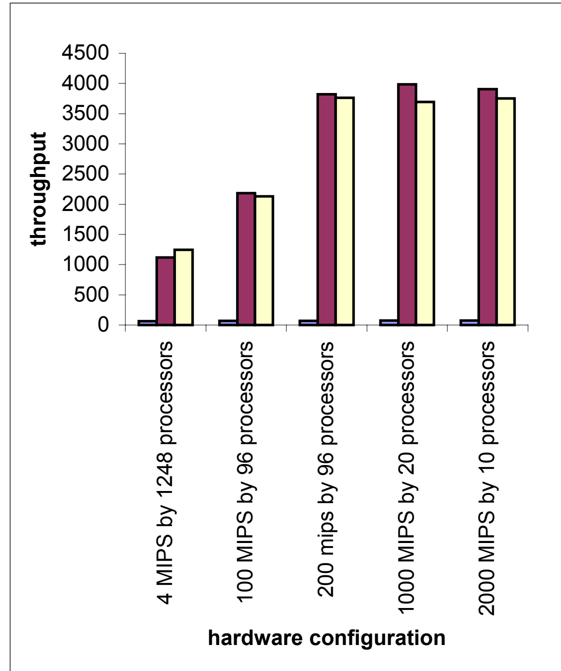
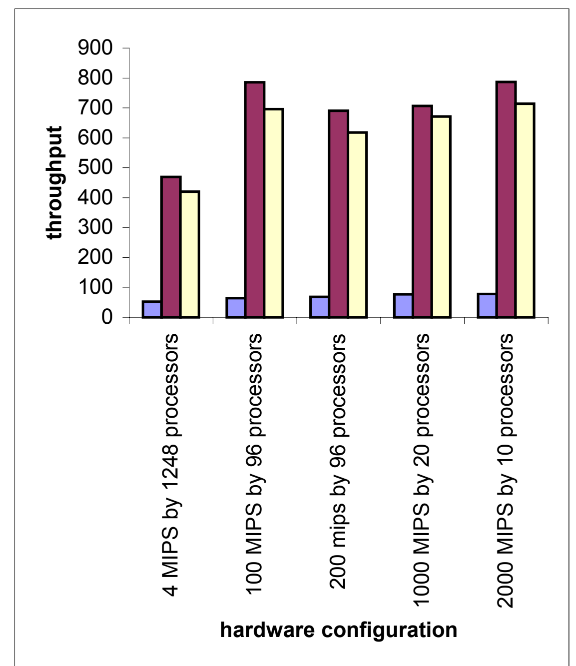



**Figure 6.1. A comparison of total throughputs under the contention based scheduler and standard 2PL concurrency control.**

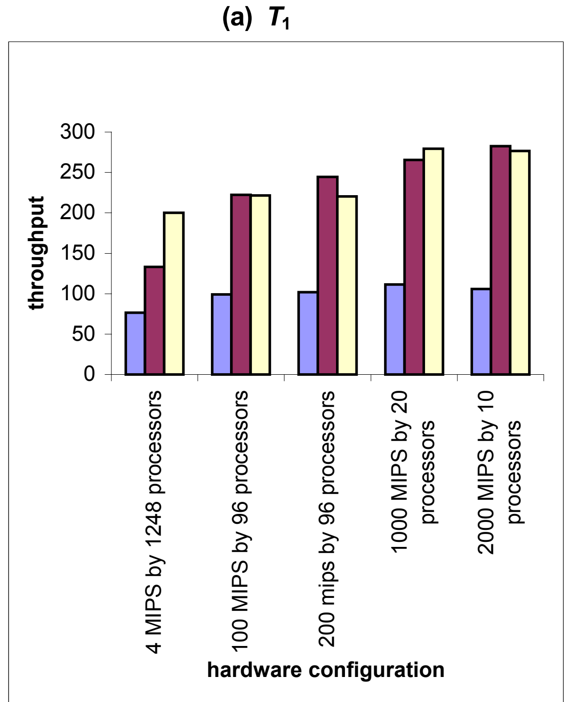
Figure 6.2 shows that as expected, this improvement is most pronounced in the throughput of the lower contention $T_1$ and $T_2$ transactions though the improvement is quite substantial for the medium contention $T_3$ transactions.



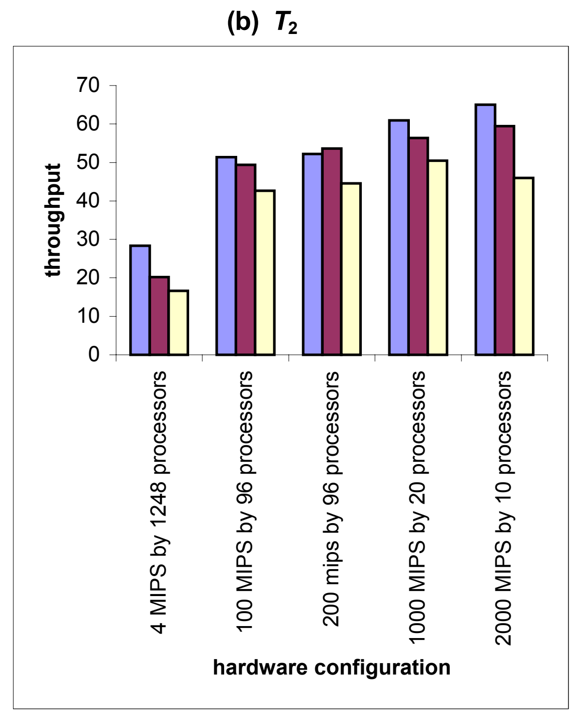

**(a)** $T_1$




**(b)** $T_2$




**(c)** $T_3$




**(d)** $T_4$

**Figure 6.2. A comparison of throughput under the contention based scheduler and standard 2PL concurrency control by transaction type.**

□ 2PL ■ Algorithm 1 □ Algorithm 2

In the massively parallel system, throughput of $T_1$ transactions under either scheduling algorithm is around 20 times that achieved by the corresponding standard 2PL system while the corresponding improvement in the throughput of $T_1$ transactions in the 96 by 100 MIPS system is around thirty fold. Under all the other hardware configurations, throughput of $T_1$ transactions under either algorithm is around 50 times that achieved by the corresponding standard 2PL system. The throughput of $T_2$ transactions, under either of our algorithms under all hardware configurations is around 10 times that achieved by the 2PL system under the equivalent hardware configuration while for $T_3$ transactions, throughput under either of our algorithms under all hardware configurations is around 2.5 times that achieved by the 2PL system under the equivalent hardware configuration.

The only transaction type where standard 2PL outperforms our algorithms is the highest contention $T_4$ transactions and here, apart for the system composed of 1248 processors operating at 4 MIPS, the superiority of the standard 2PL system is marginal particularly relative to algorithm 1. Thus, under the system composed of 1248 processors operating at 4 MIPS the standard 2PL system has a throughput of $T_4$ transactions that is 1.4 times that achieved under algorithm 1 and 1.7 that achieved under algorithm2. Under the other hardware configurations, the standard 2PL system has a throughput of $T_4$ transactions that is 1.03 times that achieved under algorithm 1 and 1.2 that achieved under algorithm 2.

The reason for the relatively poor performance of our scheduler in the throughput of $t_4$ transactions in the system composed of 1248 processors operating at 4 MIPS is because of the slow processor speeds in this system. As indicated earlier, the scheduler measures contention by accessing memory resident objects and allocating a default contention to items not found in memory. The time required for this memory access in a system composed of slow processors significantly increases the total processing time of large transactions whereas in systems composed of fast processors, the time required for this memory access is relatively insignificant even for larger transactions.

## 6.3 Allowing For Errors in the Measurement of Contention

As indicated in chapter 4, our scheduling algorithms measure contention by reading memory resident data and giving default contention vales to disk resident data. In the previous section it was assumed that the assigning of default values had no effect on the measurement of contention. In general this level of precision is highly unlikely. However, while the assigning of default values to disk resident data is not absolutely precise, given the small contribution of such objects to contention it is unlikely to cause significant error.

A more significant source of error is our scheduling algorithms high level of concurrency and the consequent abandonment of access invariance. Thus, it is possible that the satisfaction of a transaction's predicate may change between the time that its contention is measured and the time that it is executed. Consequently, it is possible that a

transaction may gain or lose high contention objects between the time that it is measured and the time that it is executed.

In the tests presented in this section, 20% of $T_1$ transactions are placed on the $T_2$ transaction queue. 10% of $T_2$ transactions are placed on the $T_1$ queue and 10% of $T_2$ transactions are placed on the $T_3$ queue. 10% of $T_3$ transactions are placed on the $T_2$ queue and 10% of $T_3$ transactions are placed on the $T_4$ queue. 20% of $T_4$ transactions are placed on the $T_3$ transaction queue. When a transaction completes, its actual contention is re-measured and the appropriate number of transactions of each type in the system is re-calculated. That is, if a $T_4$ transaction is initially measured as a $T_3$ transaction, on completion its actual contention is re-calculated and the number of $T_4$ transactions in the system is reduced by 1 thus allowing an additional $T_4$ transaction into the system.

This simulates errors that could arise due to either assigning imprecise default values or due to changes in a transaction's contention between the time contention is measured and the time the transaction is actually executed. While the selection of 20% as the margin of error for testing is arbitrary, we believe that this margin is as high as could reasonably be expected to occur. In the results presented below, Figure 6.3 compares total throughputs while Figure 6.4 breaks up total throughput by transaction class.

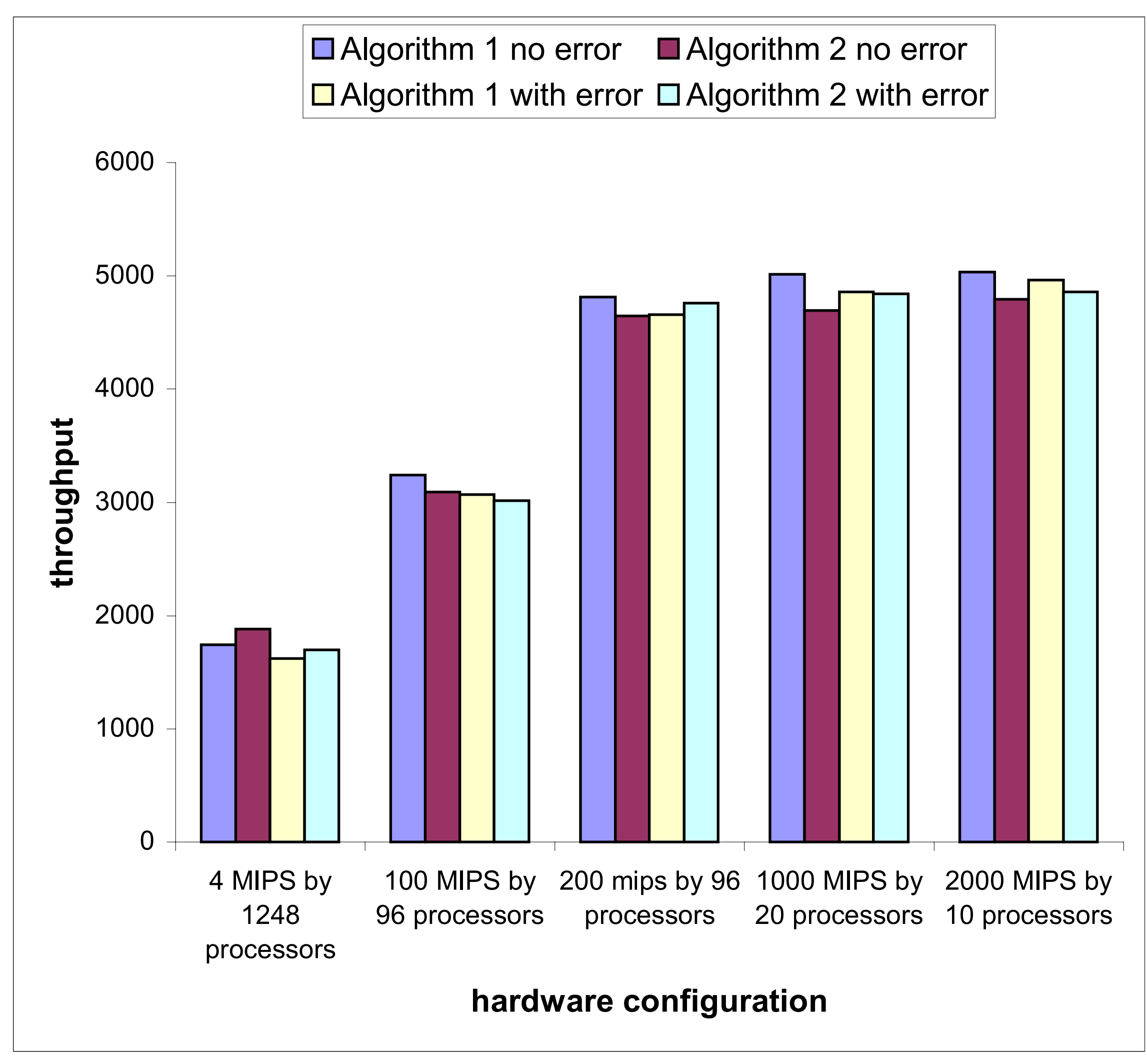
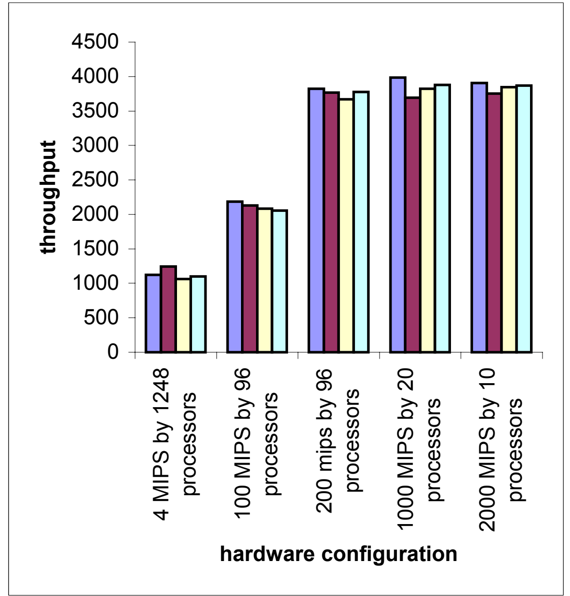
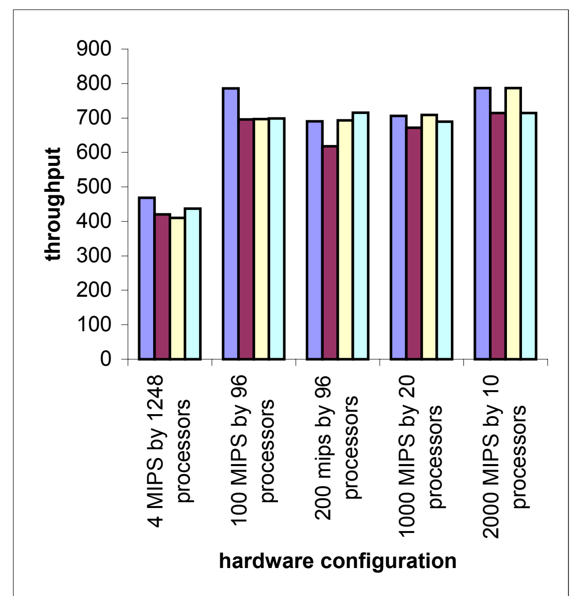


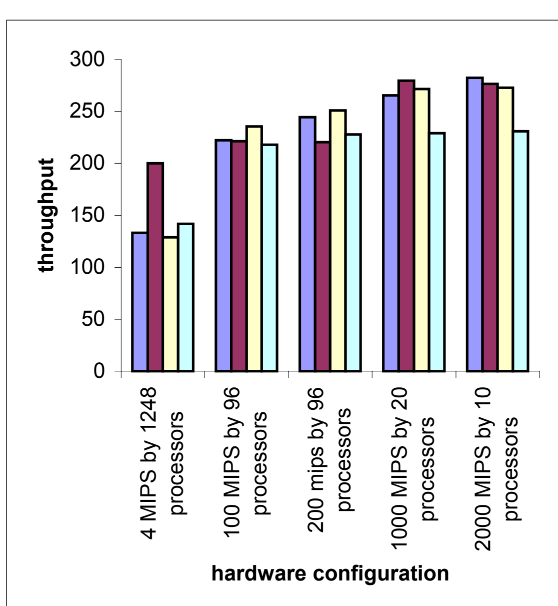

**Figure 6.3. A comparison of total throughputs under the contention based scheduler and standard 2PL concurrency control with an allowance for error in the measurement of contention.**



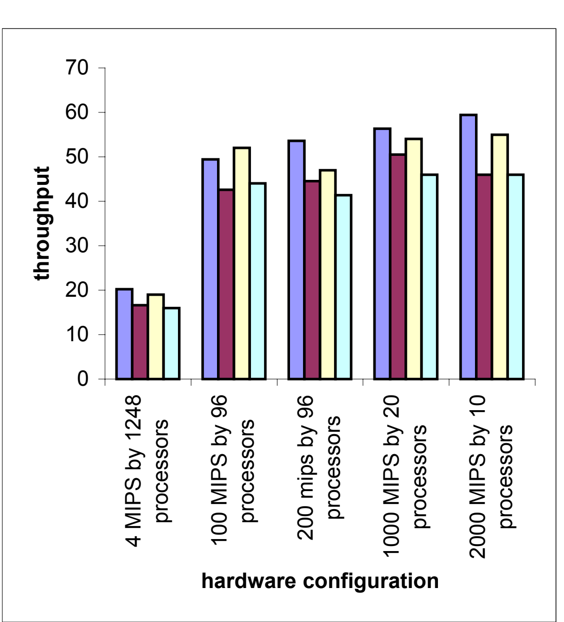

**(a)** $T_1$




**(b)** $T_2$




**(c)** $T_3$




**(d)** $T_4$

**Figure 6.4. A comparison of throughputs by transaction type with an allowance for error in the measurement of contention by the scheduler**

Algorithm 1 no error | Algorithm 2 no error
Algorithm 1 with error | Algorithm 2 with error

The results presented in Figures 6.3 and 6.4 indicate that the performance of our scheduling and rollback algorithms is very robust even when one allows a significant margin of error in the classification of transactions' contention. Thus, under all the tested hardware configurations, total throughput under both algorithms when error in measurement occurs is very similar to total throughput under equivalent hardware configuration and algorithm when no error in measurement occurs. The breakdown of total throughput by transaction type in Figure 6.4 indicates that this robustness extends to all transaction types in all the hardware configurations tested. Overall, then, the observations about the relative performance of our scheduling algorithms and standard 2PL made in the previous section hold even when a significant margin of error occurs in the measurement of contention.

## 6.4 Restricted Number of Transactions

While the results thus far have shown that the performance improvement achievable under our scheduling algorithms under diverse hardware configurations is substantial and robust, it could be argued that it is dependent on the availability of a very large number of transactions and that given a more restricted number of transactions its benefits would be limited. That is, with a restricted number of transactions and the consequent relatively low availability of low contention transactions whose performance fares best under our scheduling algorithms, one may reasonably expect that the relative advantage of our scheduling algorithms would diminish.

To determine the extent to which our scheduler maintains its advantage over standard 2PL concurrency control when the number of available transactions is restricted, we re-test the systems presented in section 6.2 but reduce the arrival rate of transactions from 20000 transactions per second to 1000 transactions per second. Given that the tests are run over 1 second, and given the relative proportion of each transaction type arriving at the system, approximately 200 $T_1$, 200 $T_2$, 350 $T_3$ and 250 $T_4$ transactions are available for processing. The results of these tests are shown in Figures 6.5 and 6.6 below.

As in previous sections, the first graph shows total throughput while Figure 6.6 breaks up total throughput by transaction class. As in section 6.2, the peak results for the standard 2PL locking method is achieved at a concurrency of 100 in the 1248 processors by 4 MIPS per processor system and at a concurrency of 70 in the other systems.

As these graphs show, even with a restricted number of transactions, the performance gains obtained by using our scheduling algorithms are quite large with total throughput under either of our algorithms being on average over double that achieved under standard 2PL. As expected, even with the reduced availability of lower contention transactions, these transactions still fare best under our algorithm.

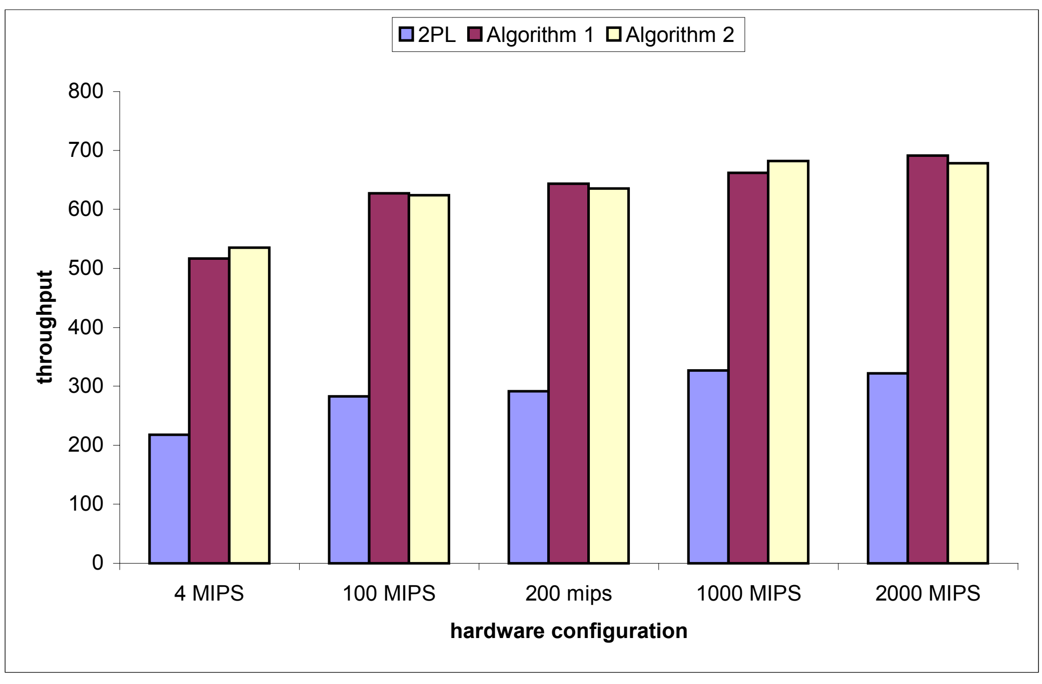
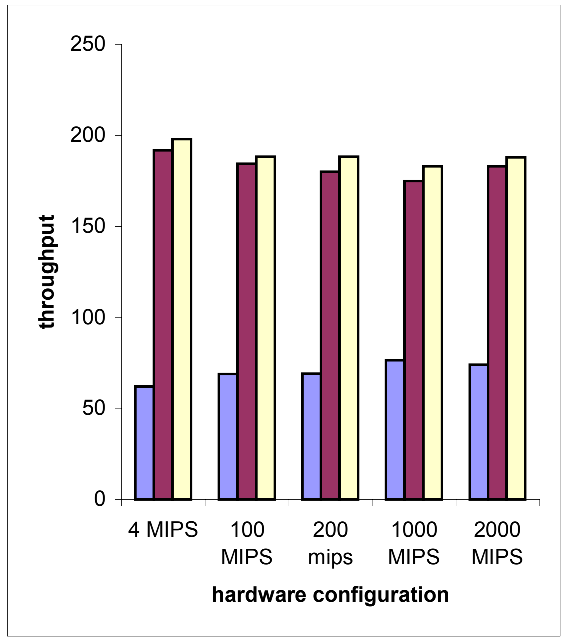


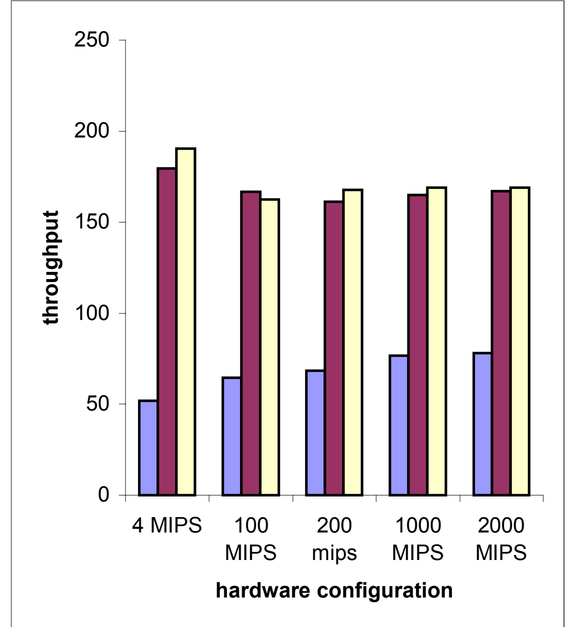

**Figure 6.5. A comparison of total throughputs under the contention based scheduler and standard 2PL concurrency control with an arrival rate of 1000 transactions per second.**

The throughput of the medium and high contention transactions under our algorithm are very similar to that achieved at the higher arrival rate. This is reasonable given that higher contention transactions have priority over lower contention transactions and therefore their performance is unaffected by the reduction in the number of lower contention transactions.

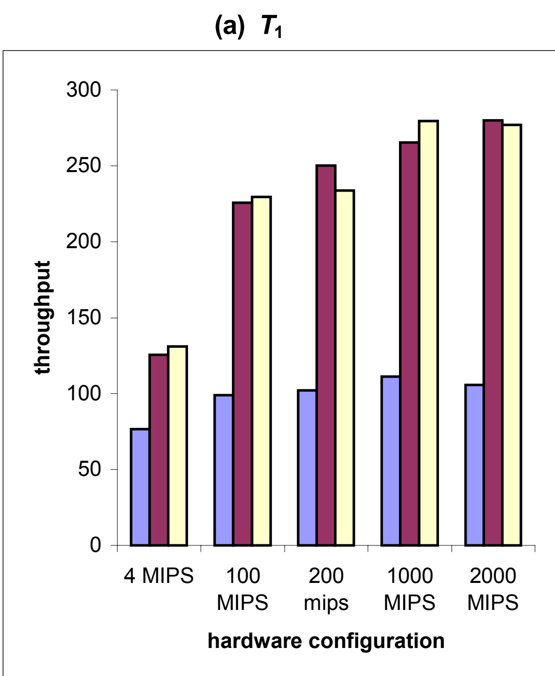
As in the results presented in section 6.2, the only transaction type where standard 2PL outperforms our algorithms is the highest contention $T_4$ transactions and again, apart for the system composed of 1248 processors each operating at 4 MIPS, the superiority of the standard 2PL system is marginal particularly relative to algorithm 1.



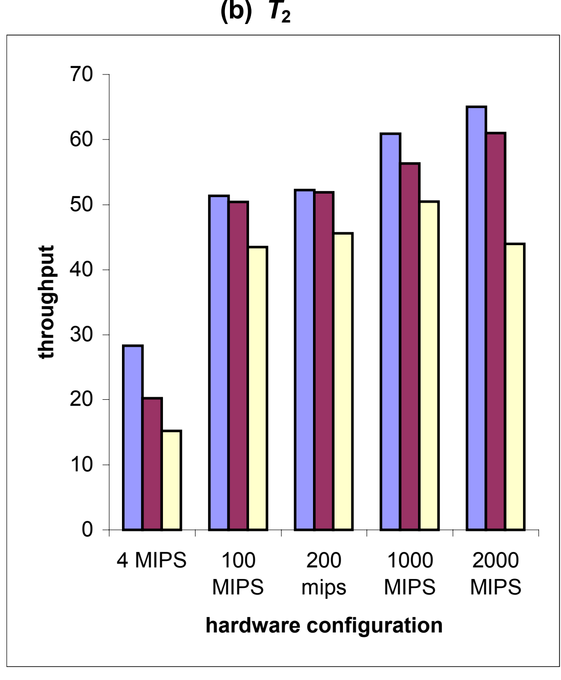

(a) $T_1$




(b) $T_2$




(c) $T_3$




(d) $T_4$

**Figure 6.6 A comparison of throughputs under the contention based scheduler and standard 2PL concurrency control with an arrival rate of 1000 transactions per second by transaction type.**

As before, the reason for the relatively poor performance of our scheduler in the throughput of $T_4$ transactions in the system composed of 1248 processors operating at 4 MIPS is because the slow processor speeds in this system increases the relative cost of evaluating transactions' contention.

## 6.5 The Performance of the Scheduler Over Extended Time Periods

In the preceding sections, we have presented results of tests run over 1 second. There are two reasons why one may expect the relative results to change over an extended period.

1. The major cost of the contention-based scheduler is the memory-based calculation of transactions' contention. This is particularly costly for systems with slow processors. However, as time progresses, if there is surplus capacity in the system such that some surplus resources can be devoted to calculating transactions' contention, the relative cost of the calculation of contention would decrease and one would expect an improvement in the relative performance of transactions under the scheduler relative to that achieved under standard 2PL concurrency control. This is particularly true for larger transactions for which the calculation of contention is particularly costly.

For example in the 1248 by 4 MIPS system under algorithm 1, a maximum of 1012 transactions are allowed to be active. Thus, if at a moment in time, all active

transactions are undergoing a memory operation, 236 processors can be evaluating the contention of non-active transactions. If at a moment in time, all active transactions are undergoing a disk operation, 1248 processors can be evaluating the contention of non-active transactions. Thus when a transaction completes, a new transaction can replace it without having to wait for its contention to be calculated.

2. As indicated in chapter 3, once a threshold of concurrency is passed, throughput per cycle starts to diminish with every successive cycle. As shown in chapter 3, for the transaction model used in that chapter which is the same as the transaction model used in this chapter, that threshold is a concurrency of 20. Thus, as one extends the time period over which tests are conducted, one would expect that the concurrency at which total throughput peaks would decrease in the standard 2PL systems. In the case of the scheduler, under algorithm 1, 80 $T_3$ and 14 $T_4$ transactions are allowed into the system while under algorithm 2, 57 $T_3$ and 10 $T_4$ transactions are allowed into the system.

The results in the previous chapter indicated that for fixed sized $T_3$ transactions the threshold concurrency is around 30 and for $T_4$ transactions the concurrency threshold is around 7 transactions. As well, the concurrency threshold is reduced in systems with variable sized transactions. Thus, since under both algorithm1 and 2, the number of $T_3$ and $T_4$ transactions is above the threshold, one would expect that over an extended processing period, the performance of the scheduler would deteriorate.

To compare the relative performance of standard 2PL and our scheduling algorithms over time, we run the 1248 by 4 MIPS system for 3 seconds, 6 seconds, 20 seconds and 1 minute. The results are averaged on a per second basis. We then compare the results to those achieved with a 1 second processing time. Thus for example, the results of the tests for 20 seconds are divided by 20 giving the average results per second. The transaction model used is the same as that used in the preceding section with the transaction arrival rate restricted to 1000 transactions per second.

We only extend the processing time of the 1248 by 4 MIPS system and restrict the arrival rate of transactions since an extension of the allowed processing time and the number of available transactions to the other systems requires more resources than are available to us. Nevertheless, we feel that this model adequately illustrates the points that need to be made. The results of extending this period of operation on throughput are shown in Figures 6.7 and 6.8 below. As in other tests presented in this chapter, Figure 6.7 shows total throughput while Figures 6.8 breaks down total throughput by transaction type.

As in previous tests, the peak results for the standard 2PL model are shown – that is, the concurrency at which the best results are achieved are shown. As indicated earlier, for this model with processing time set to 1 second, this is achieved at a concurrency of 100. However, with processing time extended to 3 seconds, peak results over the 3 seconds are achieved at a concurrency of 50. With processing time extended to 6

seconds, peak results over the 6 seconds are achieved at a concurrency of 40, with processing time extended to 20 seconds, peak results over the 20 seconds are achieved at a concurrency of 30 and with processing time extended to 1 minute, peak results over the minute are achieved at a concurrency of 20. For comparison, we also show the average result per second when concurrency is maintained at 100 as the length of the time allowed for processing is extended.

As these graphs show, under standard 2PL, while throughput over 1 second is maximized at a concurrency of 100, as the period of processing is extended, the concurrency at which throughput is optimized over the entire processing period decreases. Alongside this decreasing optimal concurrency there is also a decrease in average throughput per second as the processing period is extended. This pattern conforms to expectations since as indicated earlier, once 2PL systems exceed a threshold concurrency their throughput tends to decrease in every successive cycle. Interestingly, this rate of decrease tends to be higher the lower the contention/smaller the size of the transaction.

This is clearly shown by table 6.1. Here the average throughput of every transaction type over each processing period is expressed as a ratio of the throughput achieved when the processing period is 1 second. Thus, over 1 minute, the peak throughput of $T_1$ transactions per second is about 25% that achieved when the processing time is only 1 second and over a minute, the peak throughput of $T_4$ transactions per second is about 25% that achieved when the processing time is only 1 second.

This tendency for the rate of throughput of higher contention transactions to fall at a slower rate than that of lower contention transactions seems to be contrary to expectations since as indicated in the previous chapter, in systems with fixed sized transactions, in successive cycles, the throughput of higher contention transactions seem to decrease at a faster rate than does that of lower contention transactions. The explanation for this seemingly contrary phenomenon is that over time, in systems with variable size/contention transactions, larger transactions exhibit a consistent tendency to increase their representation in the system. Thus, in this system, on arrival, 20% of transactions are $T_1$, 20% are $T_2$, 35% are $T_3$ and 25% are $T_4$. However, at the end of 1 minute of processing, of the transactions executing in the system at a concurrency of 20, on average, 8.75% of transactions are $T_1$, 16.75% are $T_2$, 37% are $T_3$ and 37.5% are $T_4$. Similarly, at the end of 20 seconds of processing at a concurrency of 30, on average, 8.3% of executing transactions are $T_1$, 11% are $T_2$, 33.16% are $T_3$ and 47.5% are $T_4$, while at the end of 6 seconds of processing at a concurrency of 40, on average, 6.3% of executing transactions are $T_1$, 11% are $T_2$, 32.5% are $T_3$ and 50.25% are $T_4$ and at the end of 3 seconds of processing at a concurrency of 50, on average, 6% of executing transactions are $T_1$, 10% are $T_2$, 37.4% are $T_3$ and 46.6% are $T_{4.}$

In the short term, this tendency for larger transactions to usurp resources increases their throughput as a proportion of total throughput thus explaining the relatively slow rate of decline in their throughput. In the longer term, (beyond the minute tested here), this usurpation would tend to lead to thrashing by taking the system's contention beyond the cusp catastrophe threshold (which using our measure of contention is around 0.5).

As is the case with standard 2PL, under our algorithms, average throughput per second starts to decrease as the processing period is extended past 6 seconds. However, despite this decreasing average throughput per second, the performance of our algorithms relative to standard 2PL improves – that is, the rate of decline of throughput under our algorithms is less than the rate of decline under 2PL. Under our algorithms, as under standard 2PL, the decrease is most precipitous with the lower contention transactions. However, this is not due to usurpation of resources since our scheduling algorithms maintain a constant number of transactions of each type in the system. Here, the decrease is due to the increasing length of the wait chain. It should be noted that since our algorithms give larger transactions priority in lock acquisition, larger transactions cannot be locked behind shorter transactions. Thus, as the processing time is extended and the length of the wait chain grows, shorter/lower contention transactions are at a greater disadvantage.

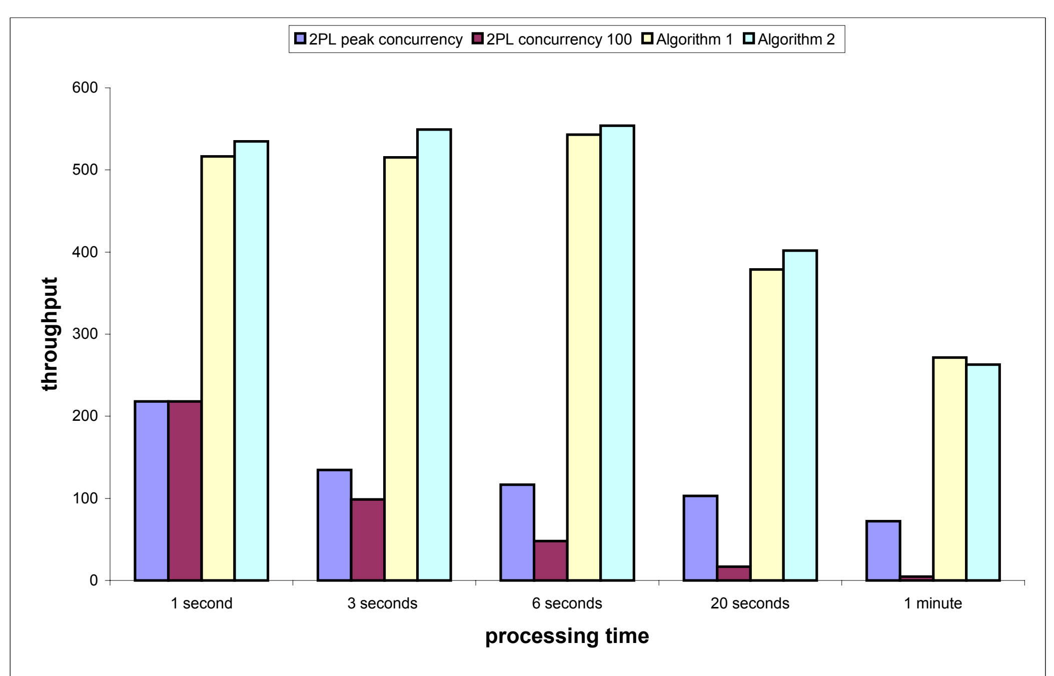
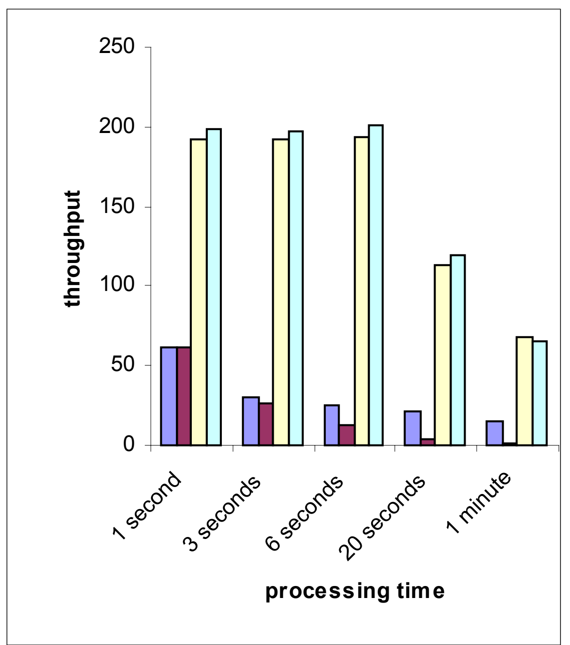
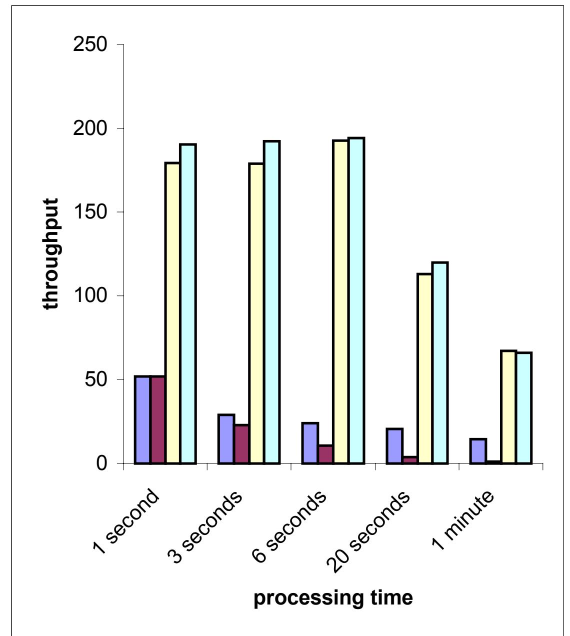


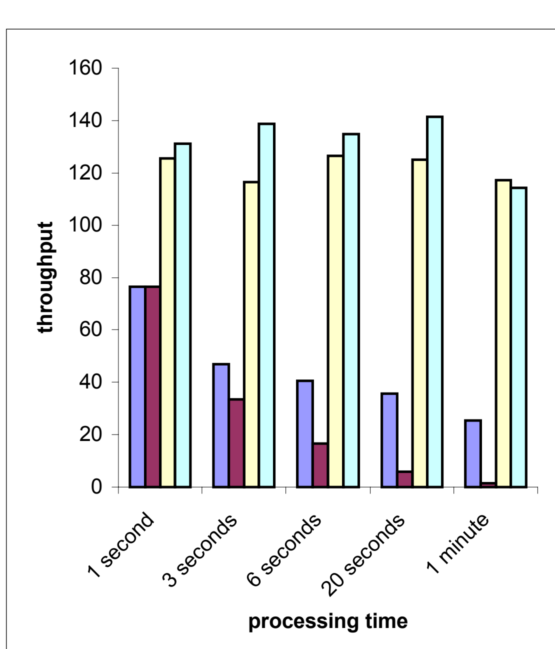

**Fig 6.7. Total throughput per second over a range of processing periods**



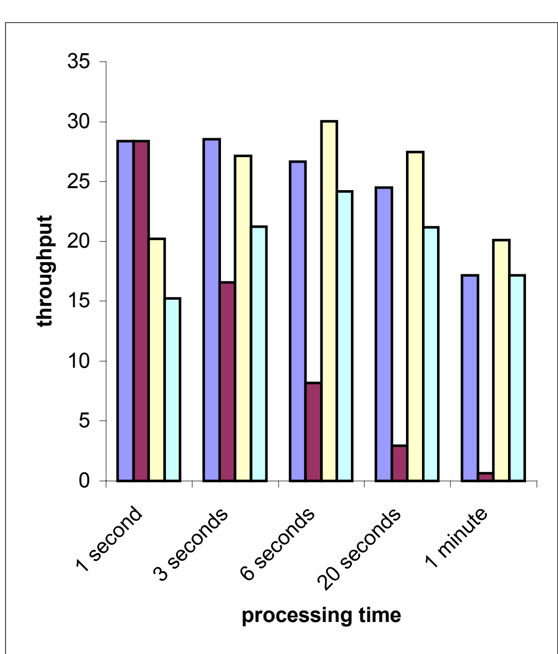

(a) $T_1$




(b) $T_2$




(c) $T_3$




(d) $T_4$

**Fig 6.8. Throughput per second over a range of processing periods by transaction type**

2PL peak concurrency | 2PL concurrency 100 | Algorithm 1 | Algorithm 2

| | 1 second | 3 seconds | 6 seconds | 20 seconds | 1 minute |
|---|---|---|---|---|---|
| total | 1 | 0.616743 | 0.5345566 | 0.47227638 | 0.331709 |
| $T_1$ | 1 | 0.485776 | 0.4067364 | 0.35360306 | 0.245102 |
| $T_2$ | 1 | 0.556091 | 0.4657666 | 0.40033751 | 0.278094 |
| $T_3$ | 1 | 0.61329 | 0.528976 | 0.4677451 | 0.333627 |
| $T_4$ | 1 | 1.006467 | 0.9403292 | 0.86269841 | 0.604938 |

**Table 6.1. Ratio of average throughput per second over a range of processing periods for a standard 2PL system**

Under our algorithms, initially, the average throughput per second of the largest/highest contention $T_4$ transactions increases as the processing time is increased. However, eventually, as the processing time is further increased the average throughput per second of these transactions decreases. The initial increase occurs because, as indicated earlier, excess resources are used to calculate contention thus decreasing the relative cost of such calculations over time. It is only once the processing period is extended past 6 seconds that the average throughput per second of these transactions begins to decrease. As well, while under the transaction model used in this section the throughput of $T_4$ transactions is initially higher under standard 2PL than it is under our algorithms, at a processing period of 6 seconds and over, the throughput of $T_4$ transactions under our algorithm 1 exceeds the throughput of these transactions under standard 2PL. The reasons why our algorithms outperform standard 2PL in the throughput of higher contention transactions over an extended period are twofold –

1. Our algorithms give higher transactions priority in lock acquisition while standard 2PL does not.

2. As the processing time is extended, optimal concurrency in standard 2PL is reduced. Thus for example, in the model used in this section, peak concurrency with 1 minute of processing is 20. As indicated above, at this concurrency, given larger transactions tendency to usurp resources, on average, 37.5% or 7.5 transactions are the highest contention $T_4$ transactions while under our scheduling algorithms there are 14 $T_4$ transactions in the system under algorithm 1 and 10 $T_4$ transactions in the system under algorithm 2. Thus over a lengthened processing period, our algorithms actually allow a larger number of high contention transactions into the system than does standard 2PL with its reduced concurrency.

## 6.6 Controlling Thrashing

As indicated in chapter 4, while our scheduling algorithms increase throughput over an extended period of time relative to that achieved under standard 2PL, because they are essentially a front end for a 2PL concurrency control system, they are still susceptible, to thrashing. In chapter 4 we outlined several strategies to combat thrashing. We now apply the two most successful disk-based thrashing control strategies – a 0.356 threshold with and without staggered restarts, to our scheduling algorithms (a detailed analysis of the results which determined our choice of strategies is given in Appendix A). That is, transactions are restarted if the proportion of blocked transactions exceeds 0.356. For the tests using staggered restarts, we firstly time-stamping each blocked transaction with the system clock plus a random number between 0 and the processing time required by an average transaction. A transaction marked for restart does not commence until the system

clock equals its time-stamp. For the tests that do no used staggered restarts, all blocked transactions are restarted immediately once the 0.356 threshold is passed. As suggested in chapter 4, we modify these algorithms for use with our scheduling algorithms by setting a separate threshold of 0.356 for each transaction type rather than setting a system wide threshold.

One would expect that using both our scheduling algorithm and thrashing control, as the processing time is extended and the initial cost of calculating contention is amortized, the relative throughput of large, high contention transactions would improve. To test this, we run the 4 MIPS by 1248 processor system for 1 minute with transactions arriving at a rate of 1000 per second. The results for 1 minute are then averaged on a per second basis. As in section 6.5, this restriction of the platform used is because an extension of the allowed processing time and the number of available transactions to the other systems requires more resources than are available to us. Nevertheless, we feel that this model adequately illustrates the points that need to be made.

Figures 6.9 and 6.10 below show the results of our tests. Figure 6.9 shows total throughput while Figures 6.10 breaks down total throughput by transaction type. The results shown by these graphs indicate that under our scheduling algorithm, as under standard 2PL, the use of thrashing control increases performance significantly under each hardware configuration.

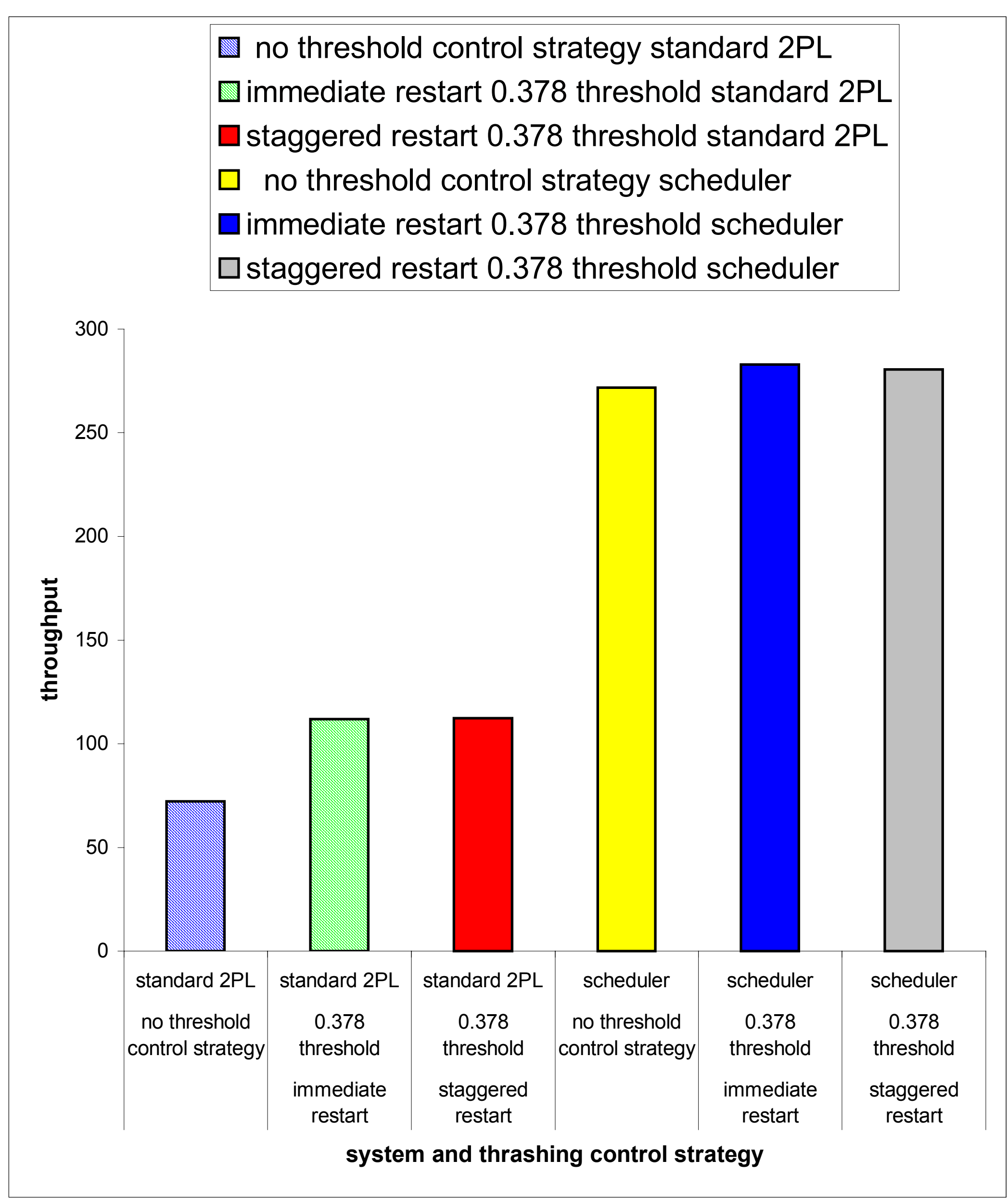
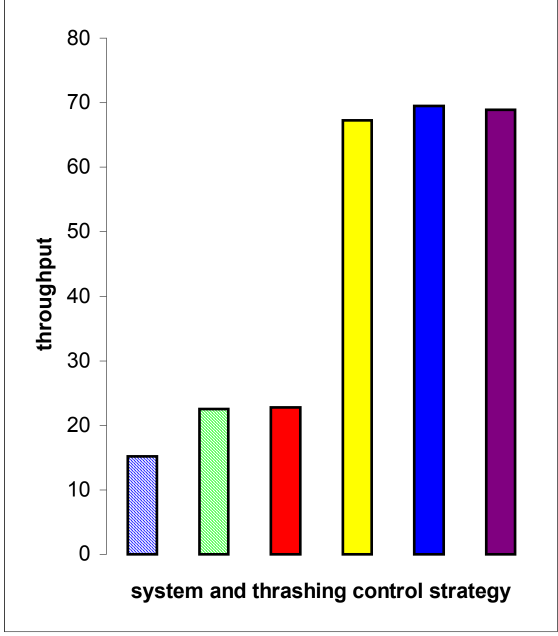
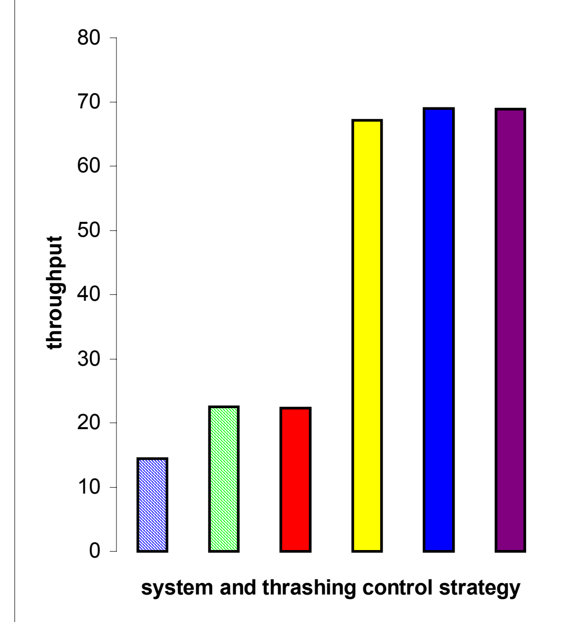


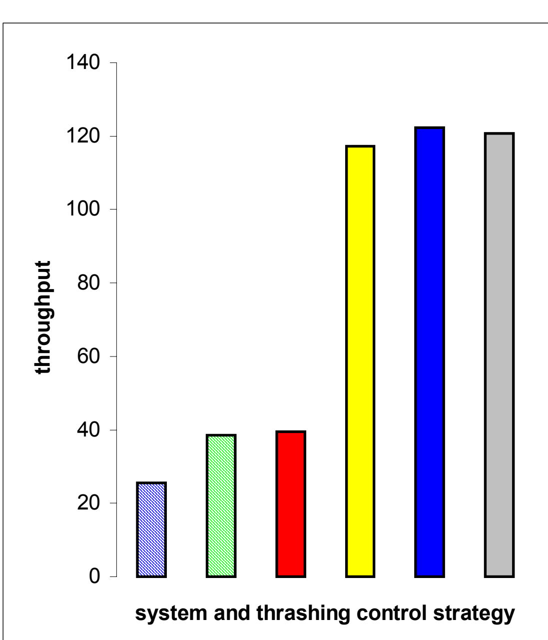

**Figure 6.9. Total throughput per second over 1 minute in the 1248 processor by 4 MIPS system**

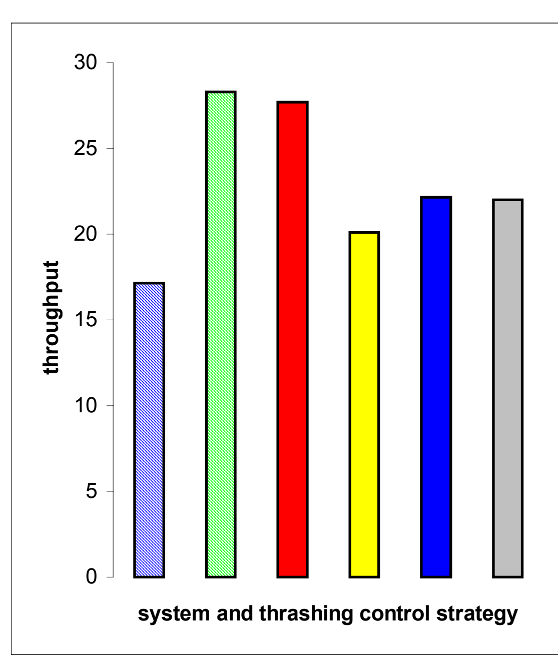



**(a)** $T_1$

**(b)** $T_2$




**(c)** $T_3$

**(d)** $T_4$

**Figure 6.10. Throughput per second over 1 minute in the 1248 processor by 4 MIPS system by transaction type**

| | |
|---|---|
| no threshold control strategy standard 2PL | immediate restart 0.378 threshold standard 2PL |
| staggered restart 0.378 threshold standard 2PL | no threshold control strategy scheduler |
| immediate restart 0.378 threshold scheduler | staggered restart 0.378 threshold scheduler |

As expected, the results in Figures 6.9 and 6.10 indicate that the advantage of those systems using thrashing control over those not using thrashing controlled is maintained when the processing time is extended for 1 minute. Also, as expected, in those systems using both our scheduling algorithm and thrashing control, there is an improvement in the throughput of the highest contention $T_4$ transactions relative to those systems not using thrashing control. Thus, the throughput of $T_4$ transactions under our scheduling algorithm with thrashing control exceeds the throughput of this transaction type in systems using our scheduling algorithm but not using thrashing control.

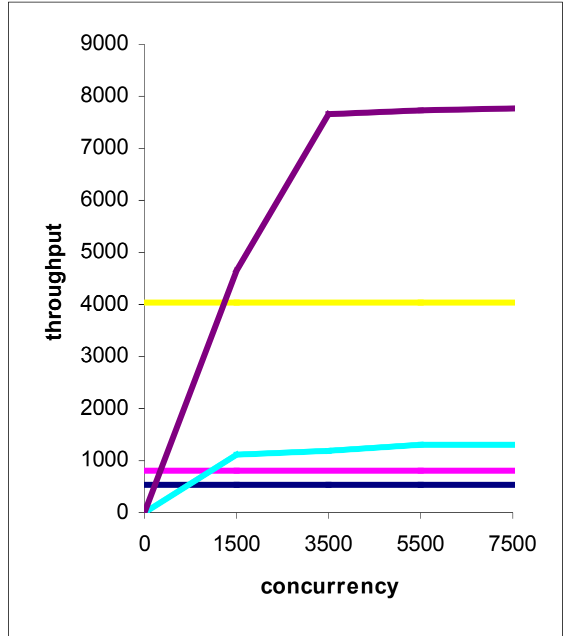
However, the most significant improvement in the throughput of the highest contention transactions comes from the standard 2PL systems using thrashing control. As Figure 6.10 shows, the throughput of $T_4$ transactions is significantly higher in the standard 2PL system with thrashing control than it is under any other system. The reason for this is, that as explained in section 6.5, in 2PL systems, larger transactions because of their longer processing time and tendency to be blocked tend to usurp resources. This increased incidence of blocked high contention transactions makes a policy that releases blocked transactions particularly successful with respect to these high contention transactions.

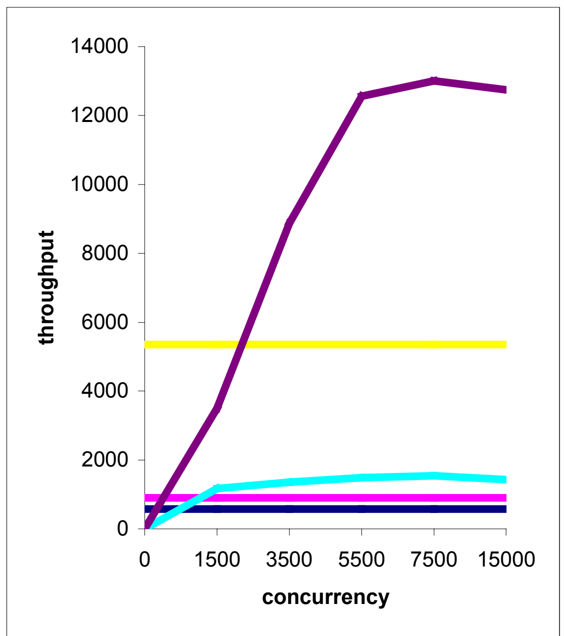
Overall, then, our scheduling algorithm with thrashing control has a significantly better performance than any of the other systems tested in this chapter. This extends to all transaction types except the highest contention transactions where the best results are obtained by the standard 2PL systems with thrashing control.

## 6.7 Very High Concurrencies

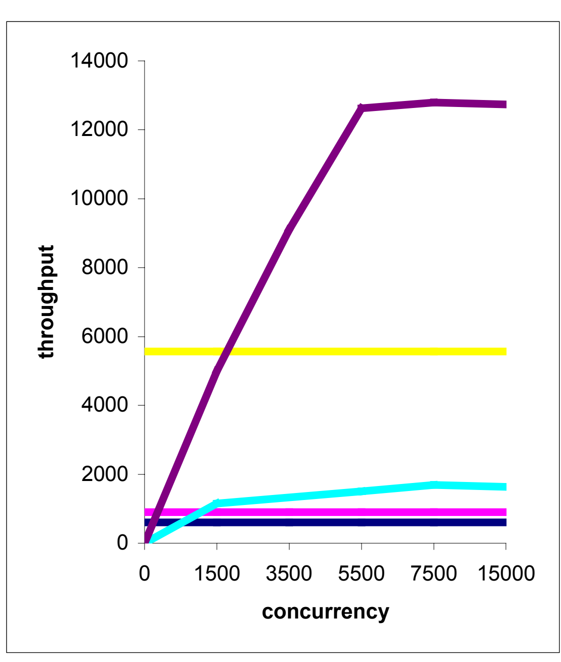
In chapter 3, the best disk based performance was achieved by the massively parallel system using WDL and optimistic concurrency control operating at very high concurrencies. In the massively parallel system, the advantages of very high concurrencies seem to outweigh the cost of the loss of access invariance in the performance of WDL and optimistic concurrency control. These results suggest that that the option of using WDL or optimistic concurrency control and operating at very high concurrencies would also be suitable for systems with the capacity to achieve high concurrencies via interleaving -even though it means sacrificing the benefits of access invariance.

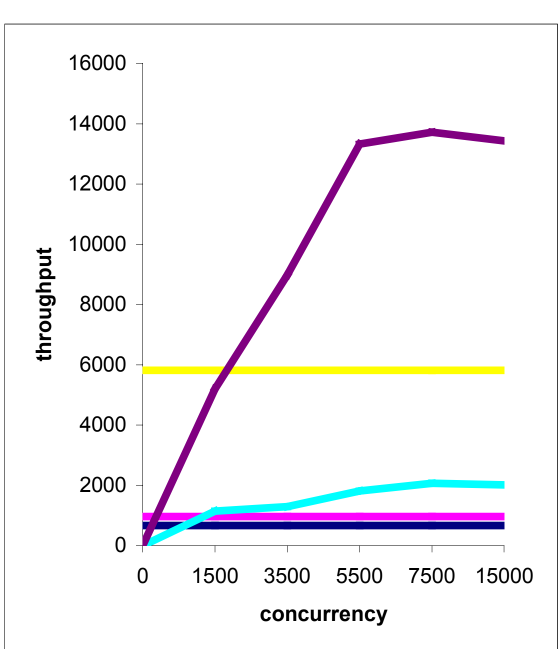
In this section investigate this possibility by using optimistic and WDL concurrency control and extending the concurrency of the disk-based systems described in chapter 3 to the maximum concurrency possible under these systems. These maximums are 7500 for the 96 processors at 100 MIPS per processor system and 15000 for the 96 processors at 200 MIPS per processor, 20 processors at 1000 MIPS per processor and 10 processors at 2000 MIPS per processor systems. At these very high concurrencies, no access invariance is assumed. These results are juxtaposed against the results presented for these systems in chapter 3 at concurrencies of 100 where access invariance is assumed to hold. As well, we compare these results to the results achieved in these systems using our scheduler with thrashing control. These results of the tests described in the previous section are shown in Figures 6.11 to 6.16 below. Figures 6.11 and 6.12 show total throughput for all the hardware configurations tested in this section. Figures 6.13 to 6.16 break down the results shown in Figure 6.11 by transaction class.




**(a) 100 MIPS per processor** **(b) 200 MIPS per processor**

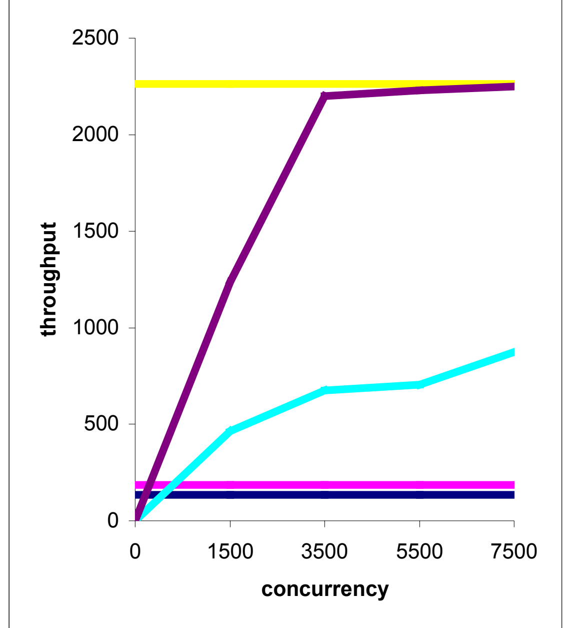


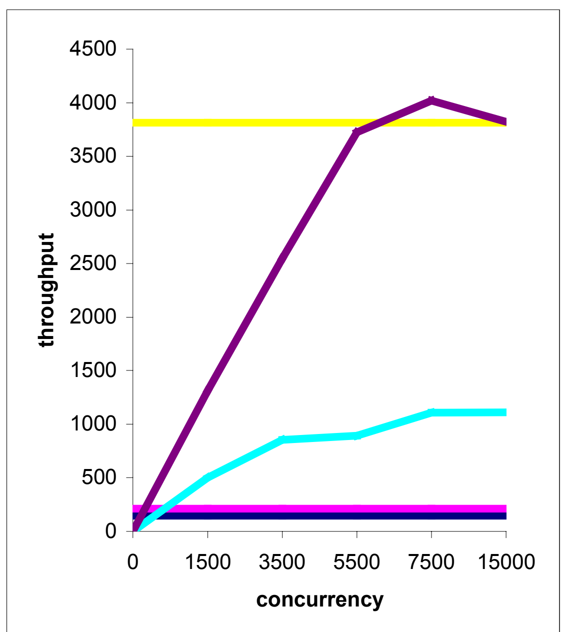




**(a) 1000 MIPS by 20 processors** **(b) 2000 MIPS by 10 processors**

**Figure 6.11. Total throughput**

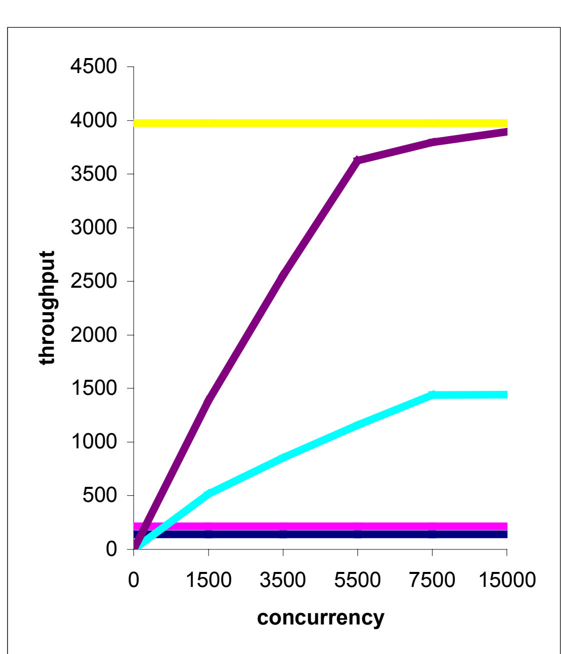
WDL with invariance concurrency 100
Optimistic kill/die with invariance concurrency 100
Scheduler Algorithm 1 with thrashing control
WDL no invariance
Optimistic kill no invariance

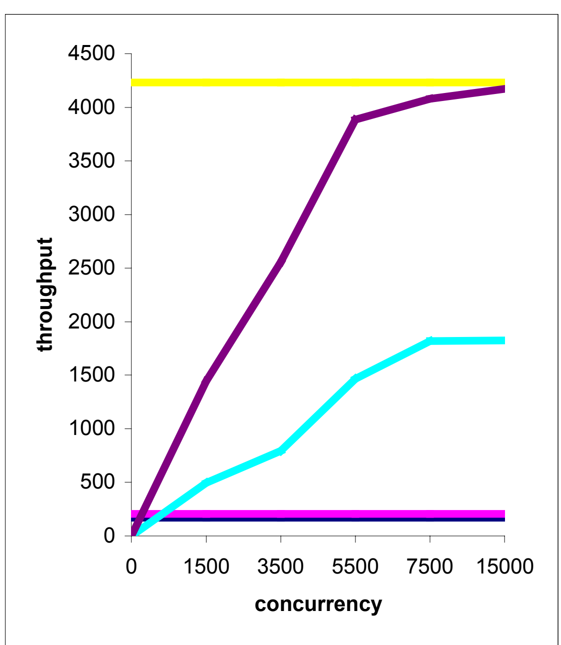



**(a) 100 MIPS by 96 processors**

**(b) 200 MIPS by 96 processors**

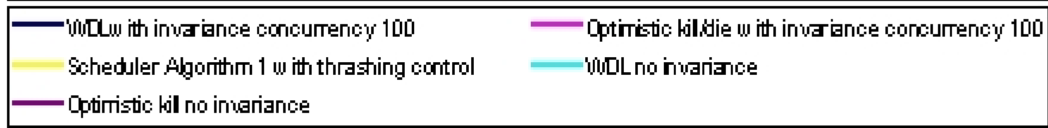


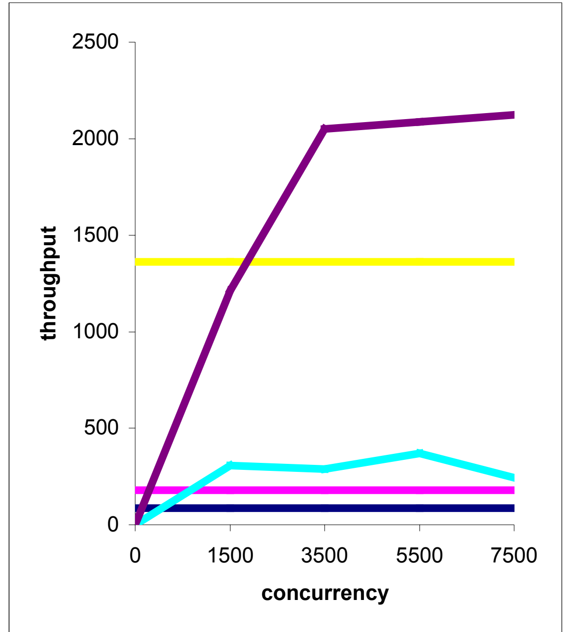




**(c) 1000 MIPS by 20 processors**

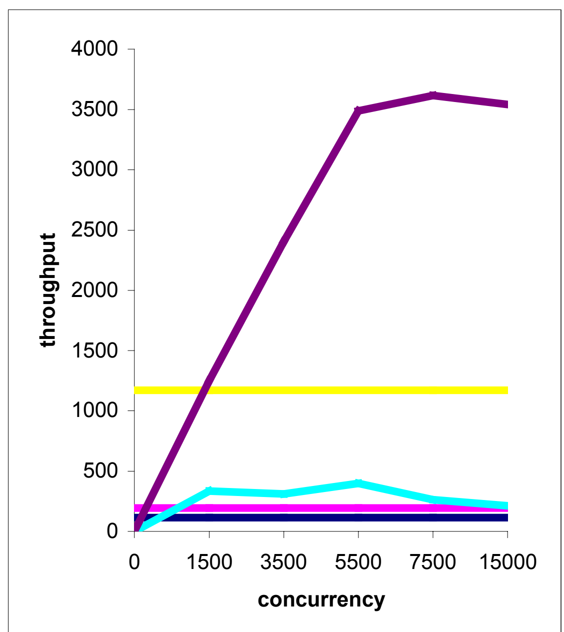
**(d) 2000 MIPS by 10 processors**

**Figure 6.12. Throughput of $T_1$ transactions**

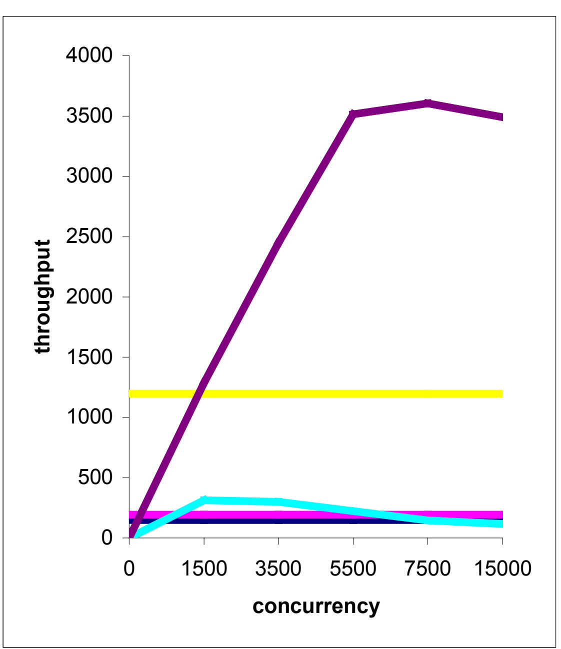




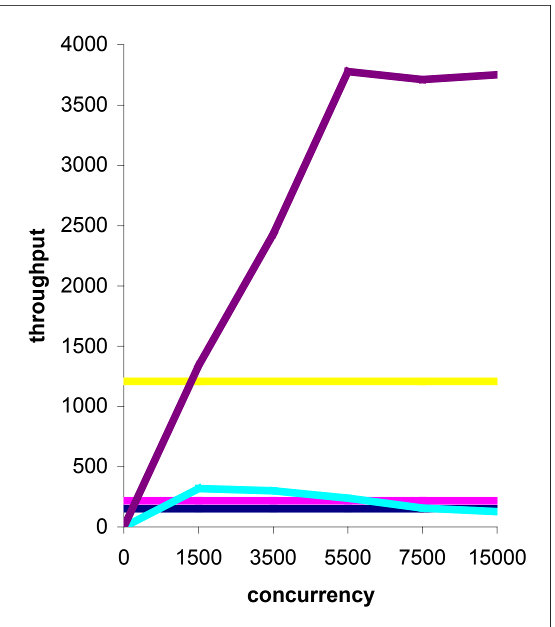

(a) **100 MIPS by 96 processors**




(b) **200 MIPS by 96 processors**

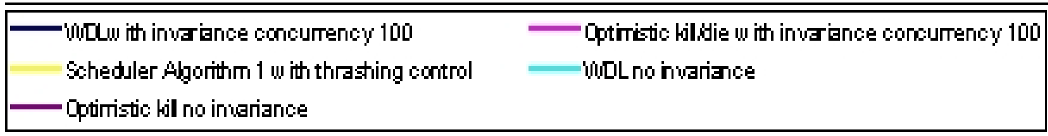


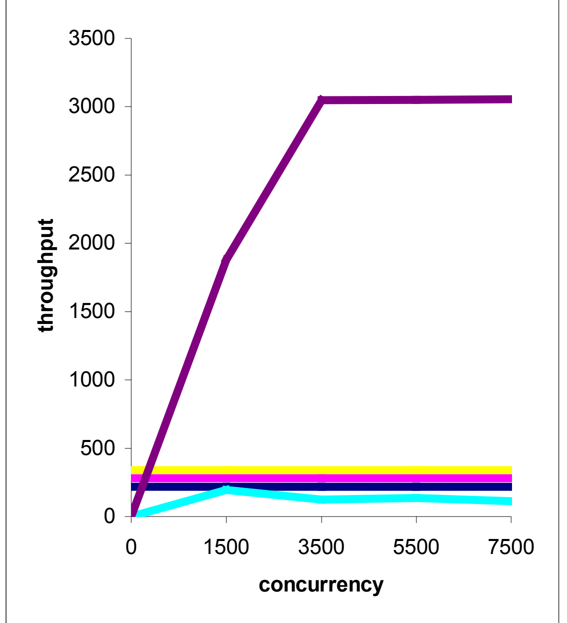

(c) **1000 MIPS by 20 processors**

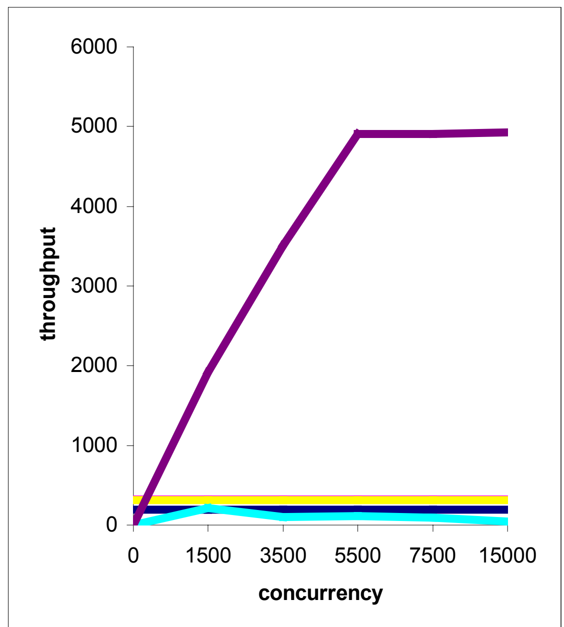



(d) **2000 MIPS by 10 processors**

**Figure 6.13. Throughput of $T_2$ transactions**

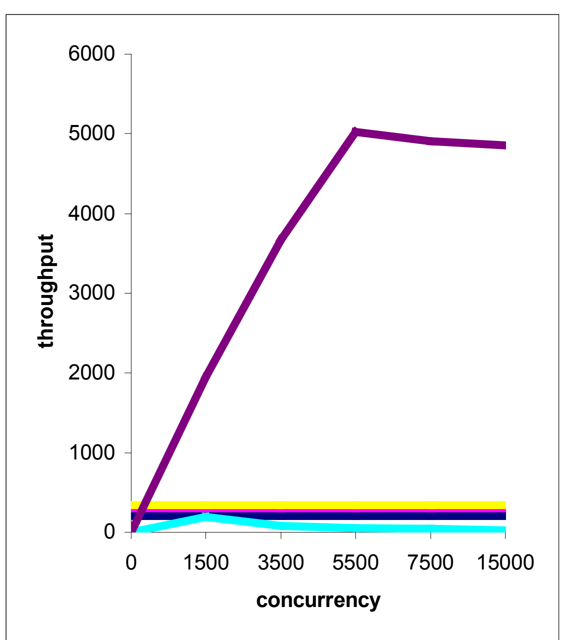


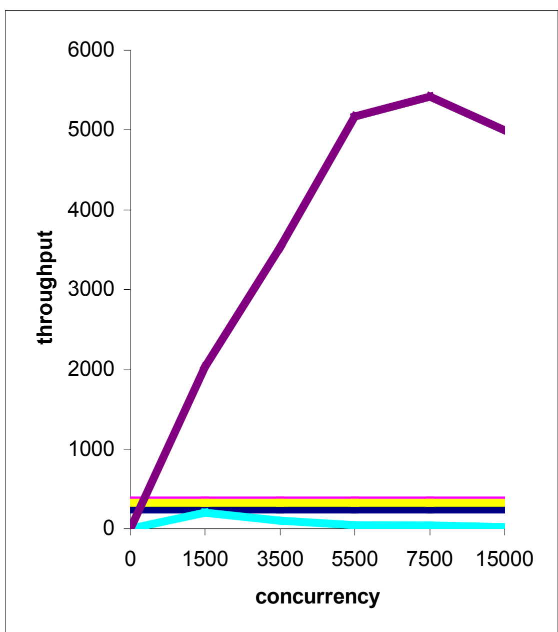



**(a) 100 MIPS by 96 processors**

**(b) 200 MIPS by 96 processors**

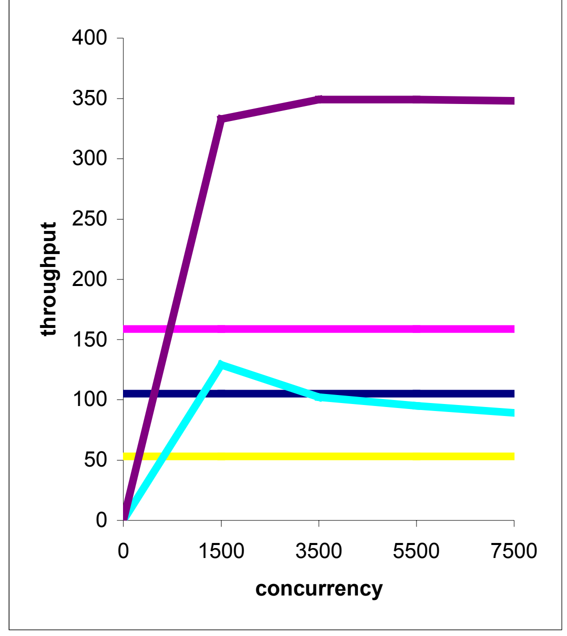


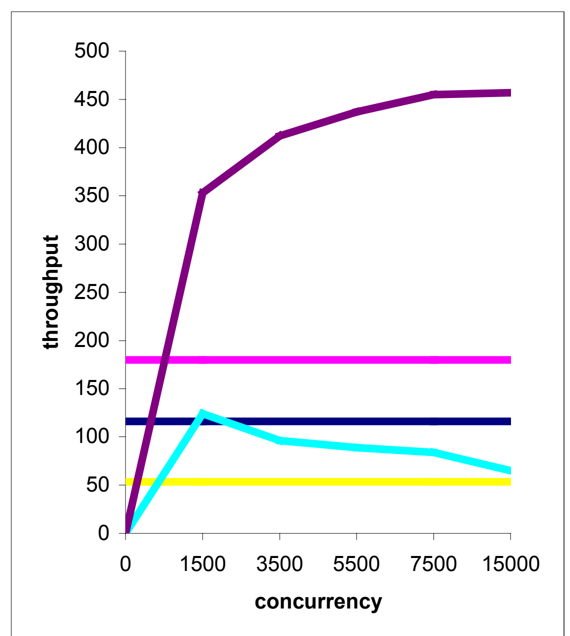




**(c) 1000 MIPS by 20 processors**

**(d) 2000 MIPS by 10 processors**

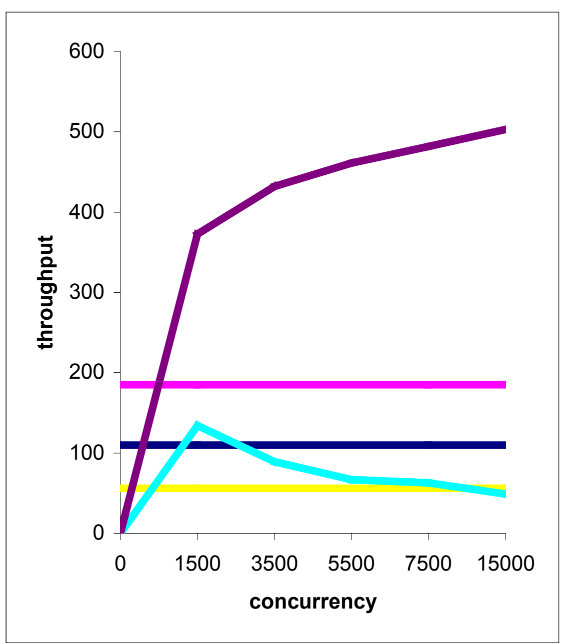
**Figure 6.14. Throughput of $T_3$ transactions**

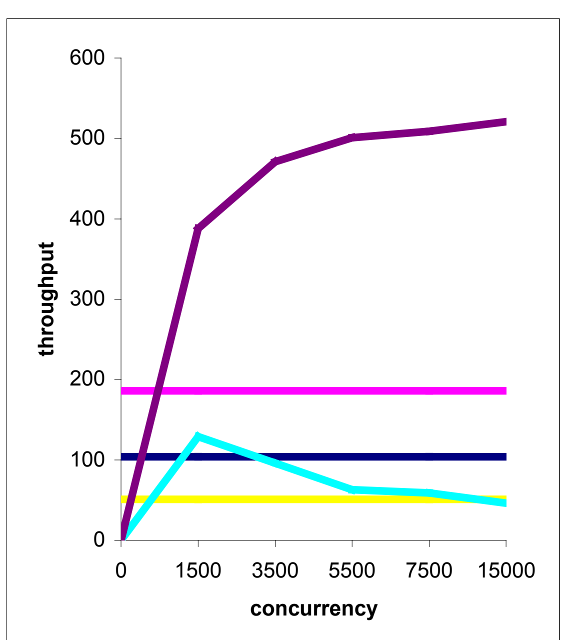



**(a) 100 MIPS by 96 processors**

**(b) 200 MIPS by 96 processors**

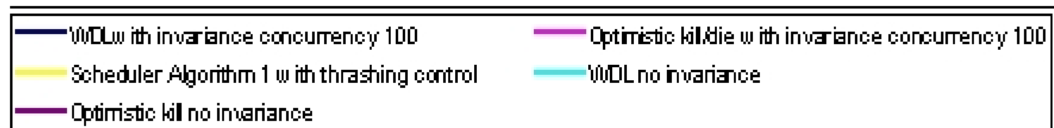



**(c) 1000 MIPS by 20 processors**

**(d) 2000 MIPS by 10 processors**

**Figure 6.15. Tthroughput of $T_4$ transactions**



The pattern for all these systems is very similar. Under WDL without access invariance, throughput peaks at a concurrency of around 1500 while under optimistic concurrency control without invariance, total throughput peaks at the maximum concurrency physically possible for the configuration in question. Peak throughputs without access invariance and the consequent cost of disk access for restarted transactions are much greater than total throughput for WDL and optimistic concurrency control at lower concurrencies with access invariance and with the consequent memory only execution of restarted transactions.

## 6.8 Summary

Tests presented in this chapter showed that the contention-based scheduler substantially outperformed standard 2PL concurrency control in a wide variety of disk-based hardware configurations and under a wide variety of conditions. The improvement though most pronounced in the throughput of low contention transactions extended to all transaction types over an extended processing period.

While the contention-based scheduler substantially improved performance, it did not address the problem of thrashing to which 2PL based systems are susceptible. Accordingly, in section 6.6 we presented results of tests where thrashing was prevented by restarting transactions once the proportion of blocked transactions exceeded 0.356. The results showed that these thrashing control policies besides successfully preventing

thrashing also increased throughput when used either by the standard 2PL system or by our scheduler.

The success of our contention-based scheduler was due to its ability to effectively harness higher levels of concurrency than is possible under standard 2PL. In section 6.7 we compared its performance against other concurrency control mechanisms operating at very high concurrencies. The results presented in this section suggest that where the hardware capacity permits, implementing kill optimistic concurrency control and extending the concurrency of the system's operation to the physically maximum limit achieve the best results. That is, in hardware systems with abundant resources, the tradeoff between high concurrency and access invariance favors the use of optimistic systems at high concurrencies. However, given users obvious preference for 2PL systems, our 2PL based scheduling algorithm performs quite creditably and yields a higher throughput than WDL.

# Chapter 7

## Validation of EMA

### 7.1 Introduction

In this chapter we test the performance of our EMA systems. Our primary comparisons are between disk-based EMA systems and in-memory systems running on hardware configurations of equivalent power in terms the number of processors and their speeds.

Three broad series of tests are presented in this chapter. The first, in section 7.2, compares the performance of 2PL, WDL and optimistic concurrency control under EMA

and in-memory to each other for equivalent hardware configurations under the assumption of 0 additional costs involved in administering EMA. In the second series of tests, in section 7.3, for each hardware configuration, under each concurrency control mechanism, EMA is run under three different costing regimes. In the first two series of tests (sections 7.2 and 7.3), where 2PL is used, it is used with thrashing control implemented. In the last series of tests (section 7.4), we compare the performance of 2PL concurrency control in in-memory and disk-based EMA systems with and without thrashing control. The thrashing control mechanism used is the immediate re-start of all blocked transactions once the proportion of blocked transactions exceeds 0.378 as discussed in chapter 5 (For further analysis of thrashing see Appendix B).

The hardware, transaction, database and processing subsystems used in this chapter are the same as outlined in section 4 of chapter 3 and as used in chapter 6. The difference between the systems used in this chapter and those used in previous chapters is that we use the EMA mechanism to pre-fetch data and that the massively parallel system is omitted. The reason for this omission is that the benefit of EMA lies in the disparity between the speeds of disk and memory. Under EMA, each transaction is executed twice, once to pre-fetch its data and once to actually commit its actions. With fast processors, the vast majority of a transaction's processing time involves waiting while data is accessed from disk and thus the cost of a double execution is negligible. However, in the massively parallel system with its slow processor speeds, the differential between memory speed and disk access speed is relatively small and the memory cost of double execution is relatively high and thus an implementation of EMA is unprofitable.

When optimistic concurrency control is used with EMA we only use optimistic kill concurrency control. We do not use either the die or die-kill optimistic method because the rationale behind their use is that virtual execution allows unsuccessful transactions to pre-fetch their data. Since under EMA data is pre-fetched prior to a transaction's actual execution, further virtual execution merely results in extra wasted work.

## 7.2 The Performance Under EMA with 0 Costs

In this section we compare the performance of 2PL, WDL and optimistic concurrency control under EMA and in-memory to each other for equivalent hardware configurations under the assumption of 0 additional costs involved in administering EMA.

As well as comparing the performance of in-memory and EMA systems against each other, this series of tests also compares EMA and in-memory systems to the peak performance of the disk-based optimistic kill method without EMA or access invariance operating at the maximum physical concurrency allowed by the hardware. Optimistic kill operating at the maximum physical concurrency allowed by the hardware is used as a reference, because as shown in the previous chapter, it yields a significantly better performance than any other concurrency control mechanism in the disk-based systems we have presented till now.

For each hardware configuration, there is a graph showing the total throughputs achieved and a graph that breaks up total throughput by transaction type. Thus, Figure 7.1 below compares total throughputs in systems containing 96 processors each operating at

100 MIPS while Figure 7.2 breaks this result up by transaction type.

The results shown by Figure 7.1 indicate that to a concurrency of around 50, the systems operating under EMA and in-memory have a comparable total throughput. The total throughput of the optimistic kill concurrency control at a concurrency of 7500 is similar to that of the EMA and in-memory systems at a concurrency of 50. Past a concurrency of 50, the in-memory optimistic and WDL systems show superior performance with the superiority increasing with concurrency while the best-performed EMA systems are those using WDL and optimistic concurrency control.

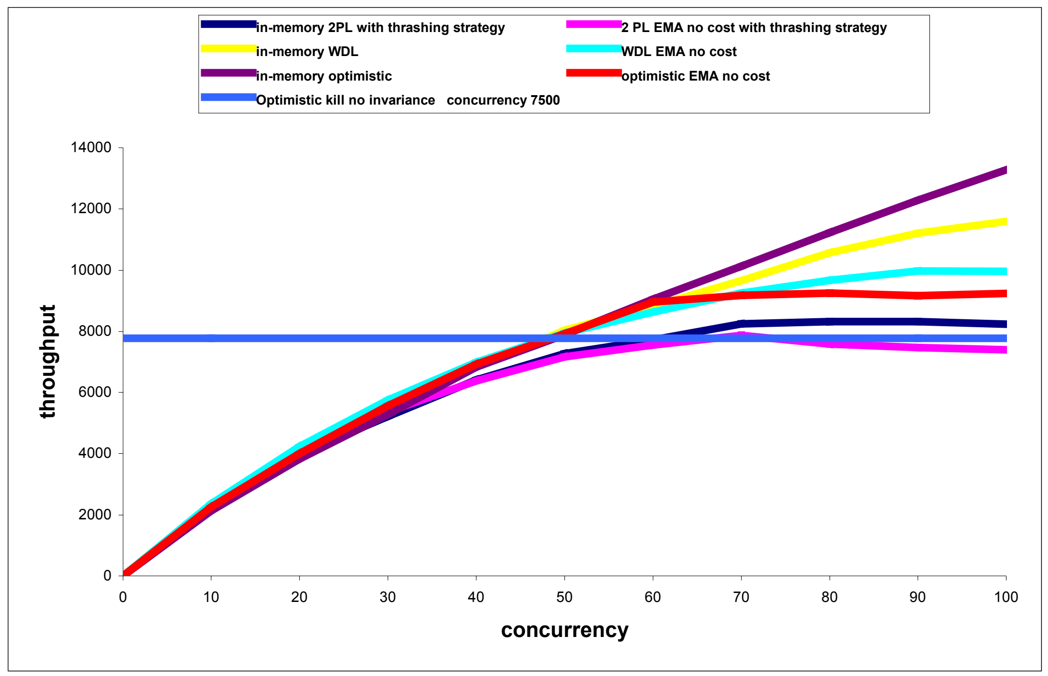
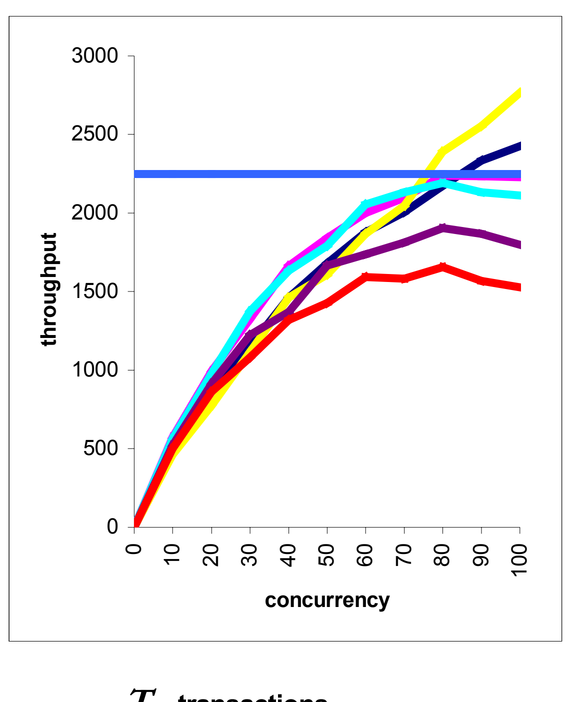
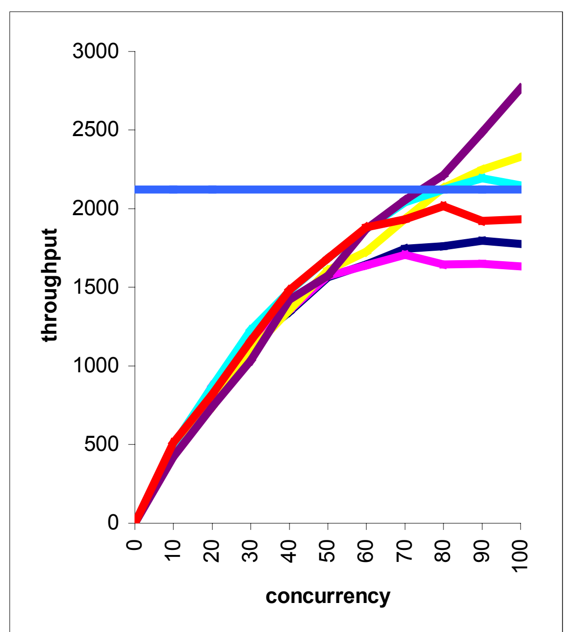


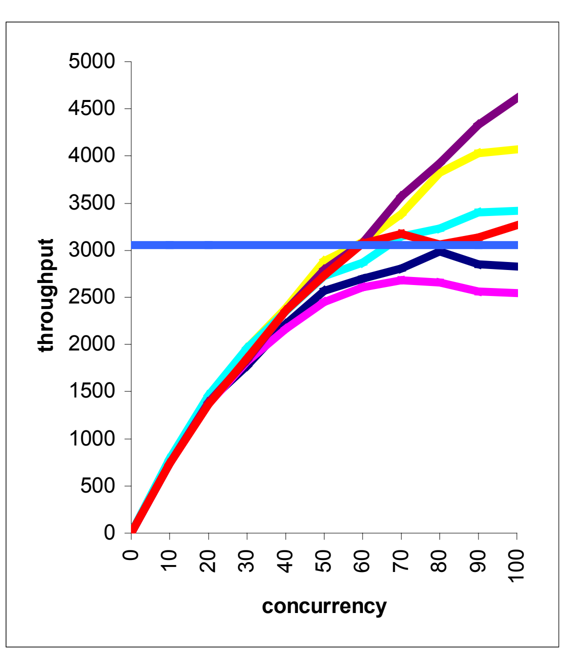

**Figure 7.1. A comparison of total throughputs in systems containing 96 processors each operating at 100 MIPS.**

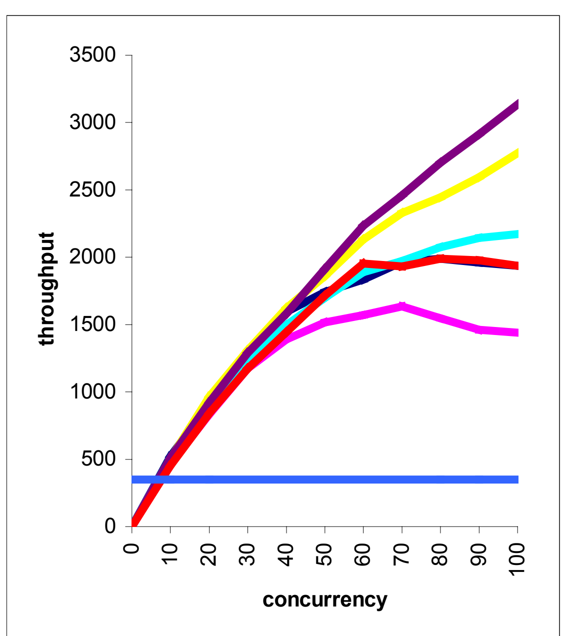



$T_1$ transactions

$T_2$ transactions

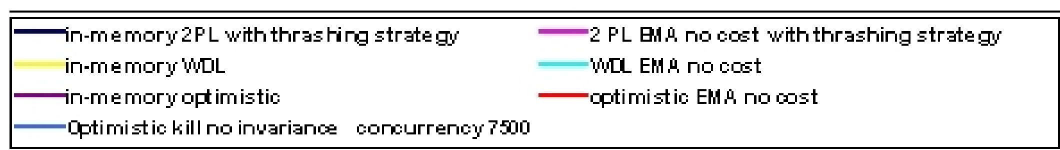



$T_3$ transactions

$T_4$ transactions

**Figure 7..2 A comparison of throughputs in systems containing 96 processors each operating at 100 MIPS by transaction type.**



In general, the patterns observed for total throughput under EMA and in-memory systems hold in the breakdown of total throughput by transaction type as shown in Figure 7.2. However, while overall the breakdowns conform to the total result, there are several interesting patterns that are worthy of comment.

The first of these is that the throughput of EMA relative to in-memory is best for the smallest/ lowest contention transactions and is worst for the largest/highest contention transactions. Thus, as Figure 7.2 shows, in the 96 by 100 MIPS systems, up to a concurrency of 80, concurrency control mechanisms under EMA have a similar throughput of the lowest contention $T_1$ transactions to the in-memory systems using the same concurrency control mechanism. Even past a concurrency of 80, the advantage of in-memory systems over EMA systems in the throughput of $T_1$ transactions is quite small. However, the advantage of the in-memory system increases progressively with transaction size/contention.

The reason for this pattern of behavior is that given the same arrival time, under EMA, smaller/lower contention transactions complete their pre-fetching phase and begin and complete their real execution phase before larger transactions complete their pre-fetch phase. Thus, in the initial stages of operation, small transactions only compete for locks with other small transactions. Since smaller transactions have low contentions, the number of conflicts is low and consequently the success of completing transactions is high leading to a high throughput of low-contention transactions. As higher contention

transactions complete their pre-fetching phase and begin actual execution, contention increases with a consequent increase in lock conflicts and a reduction in the success rate of completing transactions. This is more clearly illustrated by table 7.1 which gives a millisecond breakdown of the performance of the 2PL EMA system with a thrashing strategy and no added costs at a concurrency of 100.

| | throughput | | | | |
|---|---|---|---|---|---|
| **time** | **total** | $T_1$ | $T_2$ | $T_3$ | $T_4$ |
| 0.20 | 2302 | 873 | 658 | 714 | 57 |
| 0.40 | 3488 | 1087 | 871 | 1146 | 384 |
| 0.60 | 4694 | 1332 | 1100 | 1563 | 699 |
| 0.80 | 5985 | 1576 | 1362 | 1998 | 1049 |
| 1.00 | 7373 | 1841 | 1650 | 2498 | 1384 |

**Table 7.1. A millisecond breakdown of the performance of 2PL EMA**

As table 7.1 shows, nearly 1/2 the throughput of $T_1$ and $T_2$ transactions and nearly 1/3 of the throughput of $T_3$ transactions over 1 second occurs in the first 20 milliseconds while only 1/24$^{th}$ of the throughput of $T_4$ transactions over 1 second occurs in the first 20 milliseconds.

While the margin of performance between in-memory systems and those using EMA increases with the size/contention of the transaction types in the system, so too does the margin of performance between those systems using EMA and the optimistic kill

system operating at a concurrency of 7500. This is most noticeable in the throughput of the highest contention $T_4$ transactions where the EMA systems have a peak throughput that is between around 350% and 450% greater than the optimistic kill system operating at a concurrency of 7500. As indicated in previous chapters, in optimistic systems, small transactions tend to finish before larger ones and thus, there is a disproportionate tendency for small transactions to kill larger ones. The probability of a larger transaction being killed increases with concurrency. This, together with the fact that under optimistic kill without invariance a restarted transaction has the same disk access requirements as it had prior to restart, accounts for the vastly superior performance of EMA systems over standard optimistic kill systems in the throughput of the highest contention transactions.

As indicated above, till a threshold concurrency is reached, with the threshold depending on the contention/size of the transaction class under consideration, the throughput of in-memory systems and systems operating under EMA is quite close. It is only once the threshold concurrency is passed that the performance of the in-memory systems becomes increasingly superior for WDL and optimistic concurrency controls. The reason for this is that an upper ceiling of performance is imposed by hardware limitations. That is, under EMA, given the need to process each transaction twice (once for pre-fetch and once for committal) and given the number of processors available, the overall upper limit of throughput for WDL and optimistic concurrency control under EMA for this configuration is approached at a concurrency of 50. Till this concurrency, the cost of pre-fetching is low since of the 100 available processors, at any one time only 50 can be involved in the “real” execution of transactions thus allowing the other processors to pre-

fetch transactions.

EMA with optimistic concurrency control approaches the performance ceiling earlier than and has a lower peak throughput than its equivalent under WDL concurrency control. This is despite the fact that in our in-memory systems, optimistic concurrency has a higher throughput than WDL. This is more clearly illustrated by Figure 7.3 below distills Figure 7.1 and only shows the total throughputs of WDL and optimistic concurrency control.

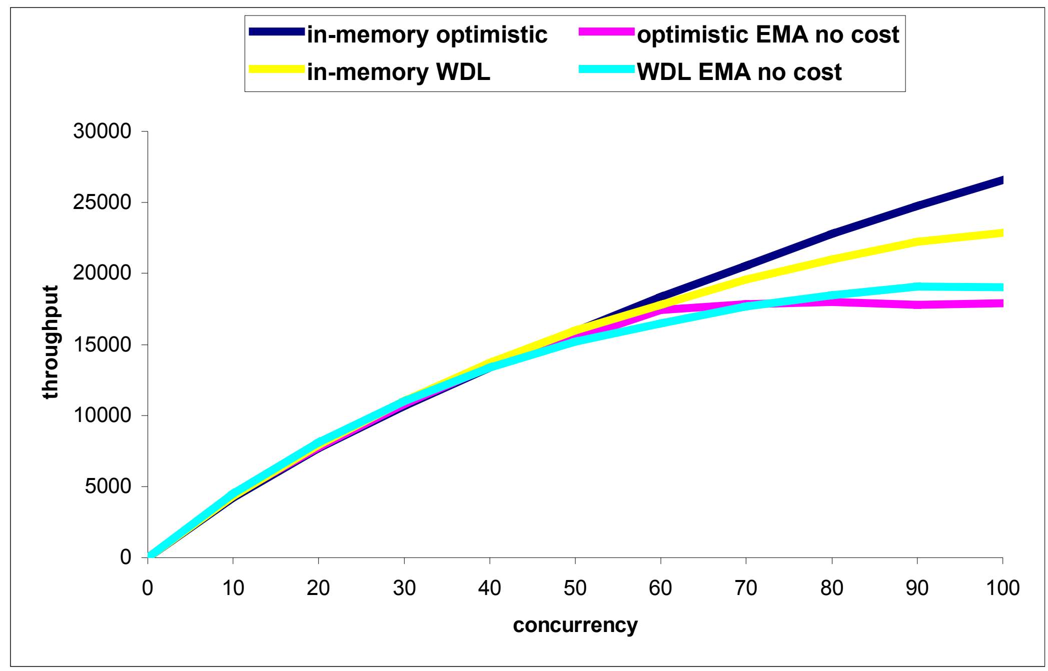



**Figure 7.3. A comparison of the throughputs of in-memory and EMA (with no cost) under both WDL and optimistic concurrency control for systems with a configuration containing 96 processors operating at 100 MIPS per processor.**

The reason for this phenomenon is that that under optimistic concurrency control, once a processor is busy processing a transaction that has already been pre-fetched, it cannot execute any pre-fetches. However, under WDL, a processor that is processing a blocked transaction can execute pre-fetches while it is waiting for its transaction to be unblocked. This causes EMA under optimistic concurrency control to reach its throughput ceiling at a lower concurrency and a lower throughput than would be the case under WDL concurrency control.

The gap between total throughput in in-memory and EMA does not occur under 2PL concurrency control. Thus, as Figure 7.1 shows, total throughputs using 2PL concurrency control with either EMA or in-memory is very similar for all concurrencies. This is because the upper limit on throughput imposed by 2PL concurrency control is reached before the limit imposed by the hardware. That is, despite the fact that the 2PL systems use thrashing control, their performance is still impaired by 2PL's propensity for creating queues of waiting transactions. This propensity increases with contention and concurrency and in the configurations configuration containing 96 processors operating at 100 MIPS per processor with the transaction model as outlined, causes both the EMA and in-memory systems to reach their peak at around the same concurrency.

The performances of the systems containing 96 processors each operating at 200 MIPS are shown in Figures 7.4 and 7.5 below. The relationships between the in-memory

and EMA systems in this configuration are virtually identical to those exhibited in the systems containing 96 processors each operating at 100 MIPS except that a higher throughput is achieved at every concurrency because of the higher processor speeds.

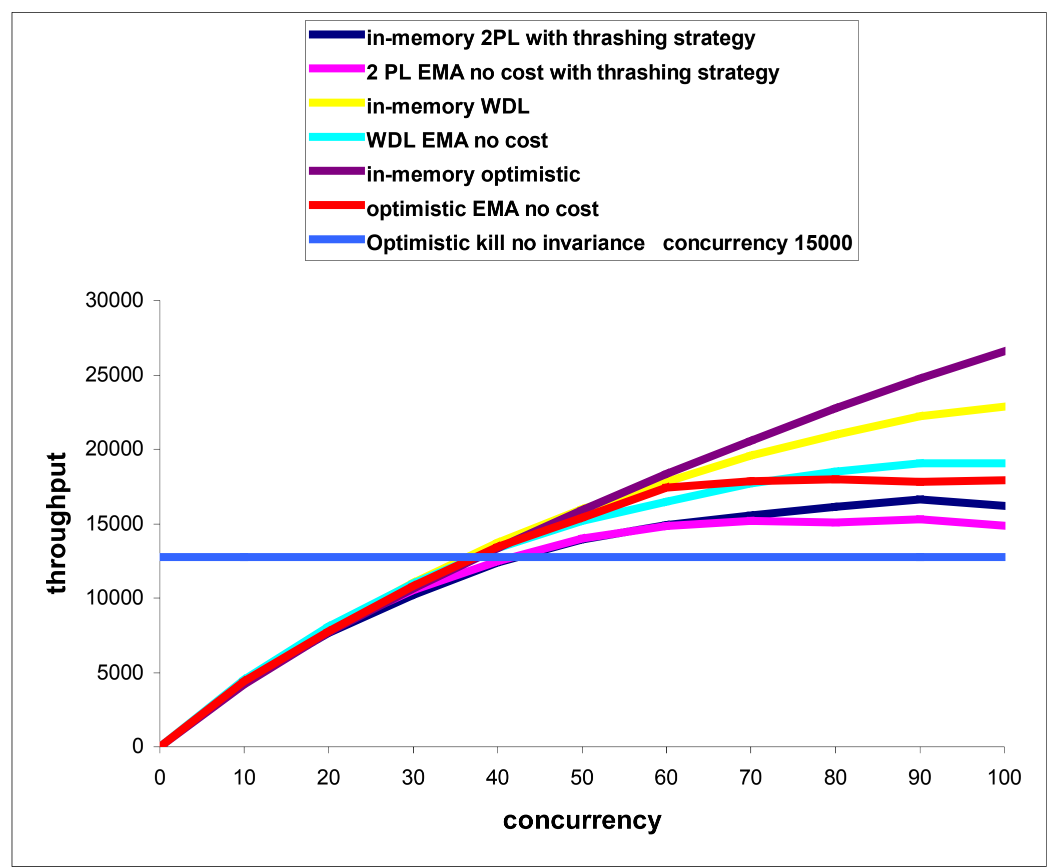
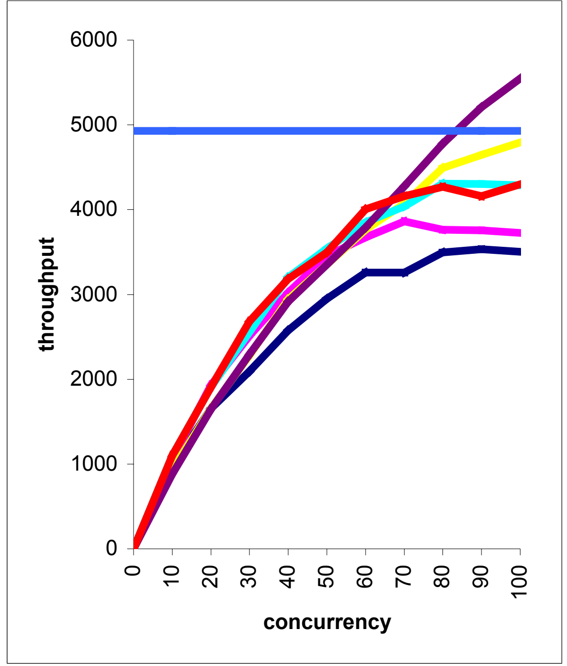
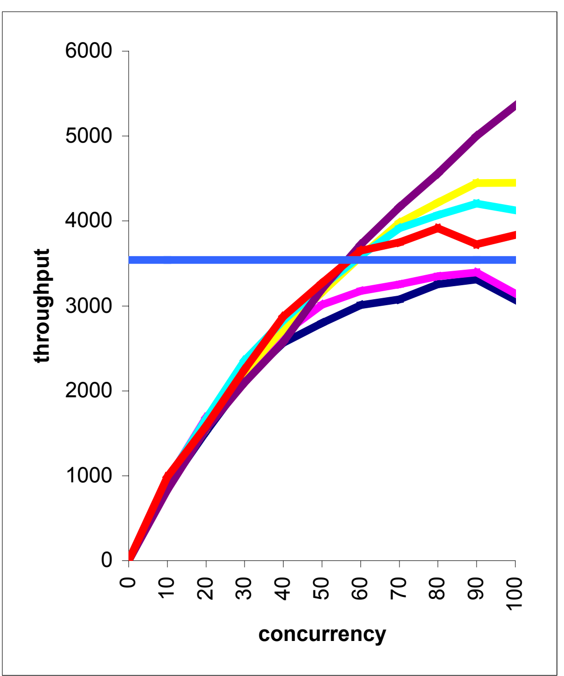


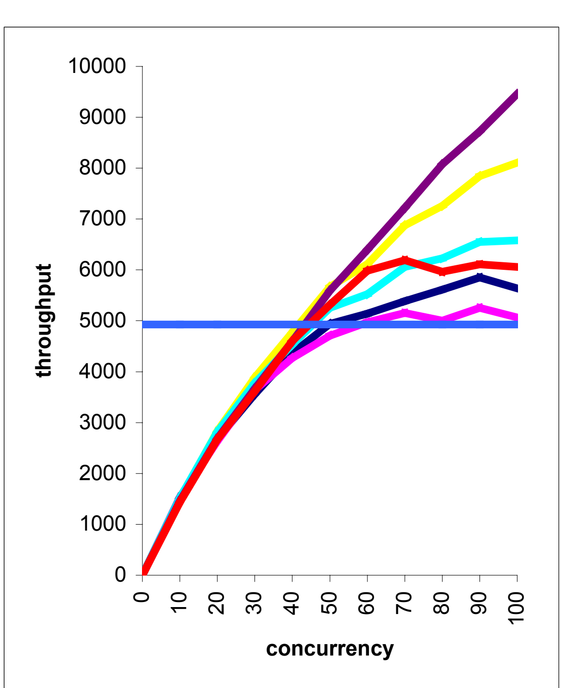

**Figure 7.4. A comparison of total throughputs in systems containing 96 processors each operating at 200 MIPS**



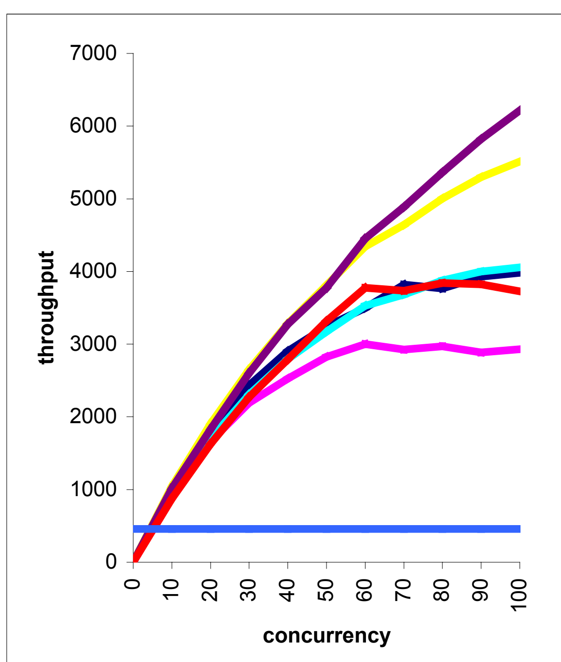

$T_1$ transactions




$T_2$ transactions




$T_3$ transactions




$T_4$ transactions

**Figure 7.5. A comparison of the throughput in systems containing 96 processors each operating at 200 MIPS by transaction type.**

However, the gap in performance between the EMA systems the optimistic kill system operating at a concurrency of 15000 is extended – particularly for the larger $T_3$ and $T_4$ transactions. Thus, in the throughput of the largest $T_4$ transactions in the previous configuration with processors operating at 100 MIPS each, the gap between the optimistic kill operating at high concurrencies and the EMA systems was between 350% and 450% while under this configuration, with processors operating at 200 MIPS each, the gap between optimistic kill operating at high concurrencies and the EMA systems is between 650% and 900%. This is because throughput under EMA can expand almost linearly with processor speeds. However, throughput under standard optimistic kill can only expand logarithmically with concurrency with increased processor speeds achieving little beyond enabling a higher level of concurrency.

The performances of the systems with the configurations containing 20 processors each operating at 1000 MIPS are shown in Figures 7.6 and 7.7 below. As under the previous hardware configurations the EMA systems using WDL or optimistic concurrency control reach a hardware-imposed ceiling on throughput. Here, the ceiling is reached after a concurrency of around 10. Because of the low concurrencies involved, the in-memory 2PL system does not reach the ceiling imposed by 2PL's propensity to create wait queues. However, as with the WDL and optimistic systems, because of the small number of processors available, the 2PL EMA system also approaches the hardware imposed throughput ceiling at a concurrency of around 10.

As in the previous configurations it is noteworthy that for every transaction type,

WDL using EMA outperforms optimistic concurrency used in conjunction with EMA. As well, under this hardware configuration, 2PL used in conjunction with EMA also outperforms optimistic concurrency used in conjunction with EMA. As previously explained the main reason between the performance of WDL using EMA (and in this configuration 2PL using EMA) and optimistic concurrency using EMA, is that in optimistic concurrency there is no opportunity for interleaving pre-fetching while a transaction is being executed while under WDL and 2PL, processors with blocked transactions can spend their time usefully by pre-fetching transactions.

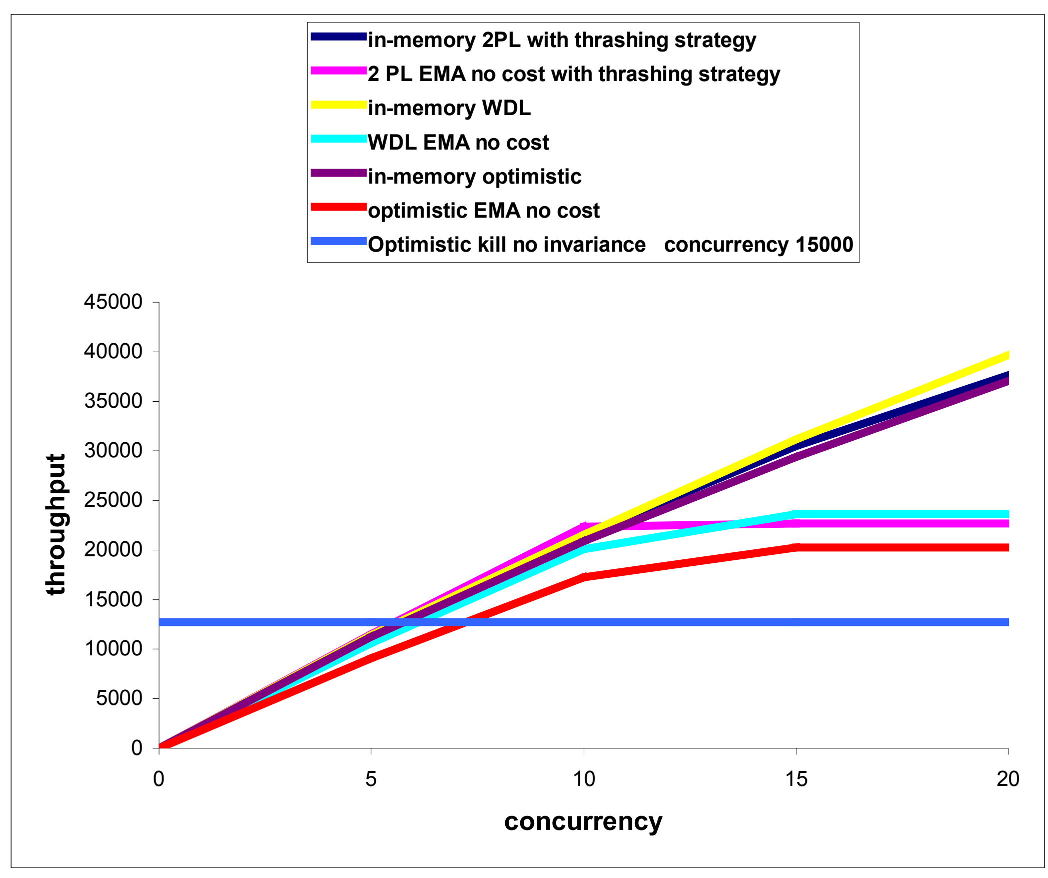
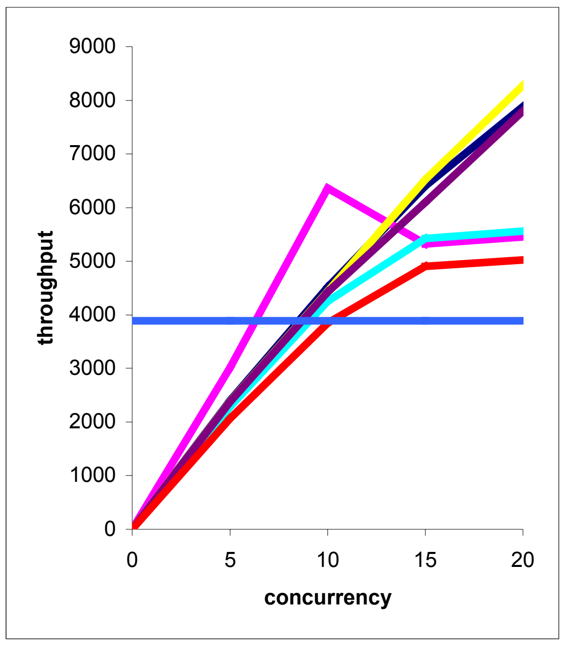
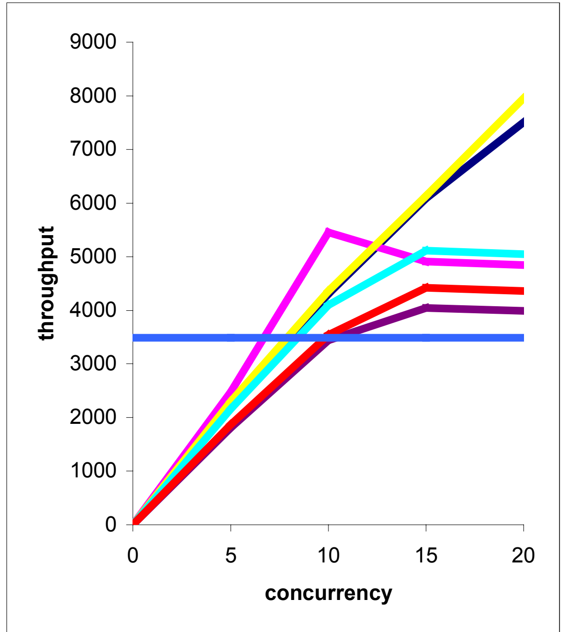


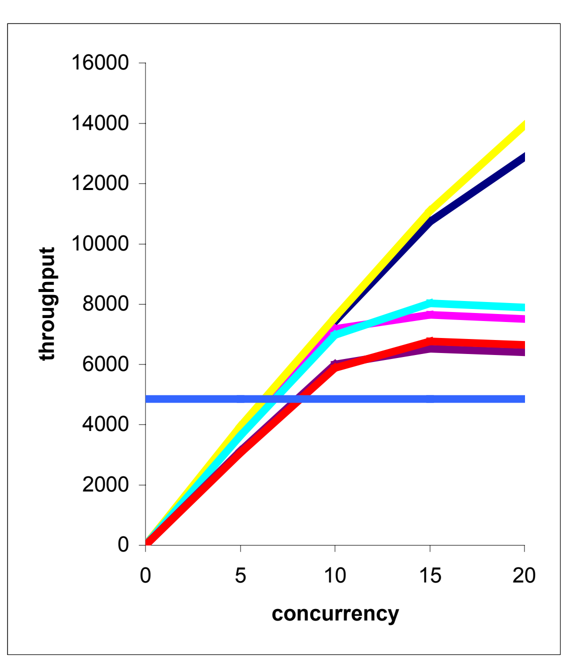

**Figure 7.6. A comparison of total throughputs in systems containing 20 processors each operating at 1000 MIPS**

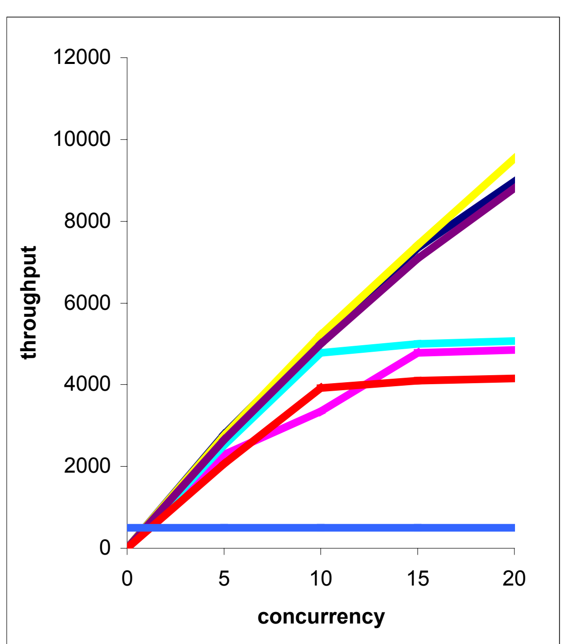



$T_1$ **transactions**

$T_2$ **transactions in**




$T_3$ **transactions**

$T_4$ **transactions**

**Fig .7.7. A comparison of throughputs in systems containing 20 processors each operating at 1000 MIPS by transaction type**

| | |
|---|---|
| in-memory 2PL with thrashing strategy | 2 PL EMA no cost with thrashing strategy |
| in-memory WDL | WDL EMA no cost |
| in-memory optimistic | optimistic EMA no cost |
| Optimistic kill no invariance concurrency 7500 | |

Once again, the gap in performance between the EMA systems and the optimistic kill system operating at a concurrency of 15000 is extended with the increase in processor speeds. In this configuration the gap is now significant for all transaction types although it is still greatest for the larger $T_3$ and $T_4$ transactions.

Thus, the gap in the throughput of the largest $T_4$ transactions has expanded from between 350% and 450% under the configuration with processors operating at 100 MIPS each, to between 650% and 900% under the configuration with processors operating at 200 MIPS each to between 825% and 1000% under the configuration with processors operating at 1000 MIPS each. As previously explained, this increasing gap occurs because throughput under EMA can expand almost linearly with processor speeds. However, throughput under standard optimistic kill can only expand logarithmically with concurrency with increased processor speeds achieving little beyond enabling a higher level of concurrency.

The performances of the systems containing 10 processors each operating at 2000 MIPS are shown in Figures 7.8 and 7.9 below.

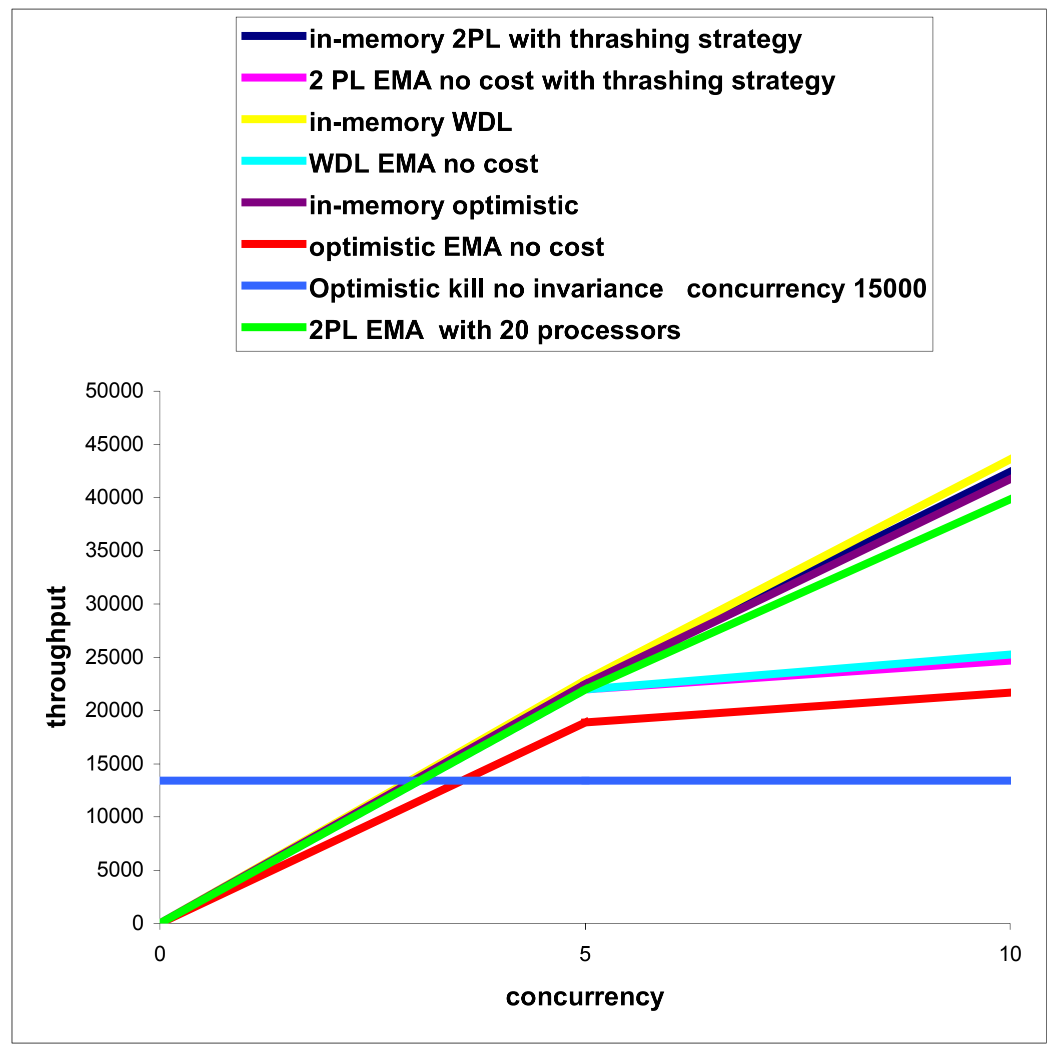



**Figure 7.8. A comparison of total throughputs in systems containing 10 processors each operating at 2000 MIPS**

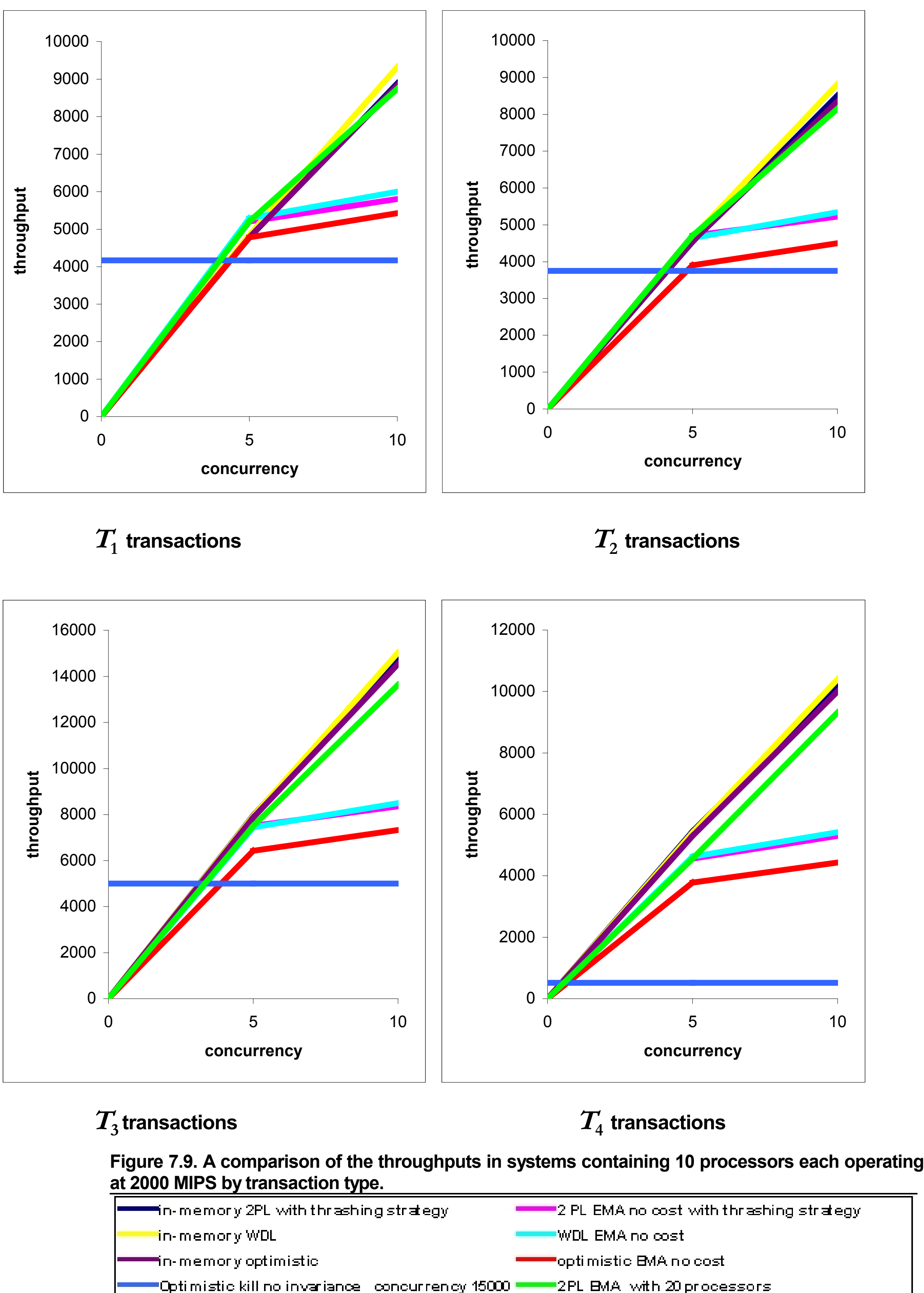



**Figure 7.9. A comparison of the throughputs in systems containing 10 processors each operating at 2000 MIPS by transaction type.**

The patterns exhibited are almost identical to those under the previous hardware configuration except that because of the lower number of processors, the hardware ceiling is reached at a lower concurrency and because of the faster processors throughputs are higher. As an indication of the scalability of EMA, 2PL EMA is also run with 10 extra processors and 15880 extra disks (giving a total of 20 processors at 2000 MIPS each and 31760 disks. As shown by Figures 7.8 and 7.9, the extra processors and disks bring the performance of 2PL EMA with 20 processors each operating at 2000 MIPS, close to that of the in-memory systems with 10 processors each operating at 2000 MIPS.

## 7.3 Adding Costs to EMA

The results presented in the preceding section assumed that there is no extra cost involved in administering EMA. In this section, for each hardware configuration, under each concurrency control mechanism, EMA is run for under three different costing regimes.

The first, as in the tests outlined above, assumes a 0 additional cost in implementing EMA.

The second costing regime assumes that once its data is pre-fetched, a transaction requires no further disk access. However, it is also assumed that 25% of transactions require forked decisions and that each fork is of equal length. Thus, 25% of transactions

require two pre-fetches even though only one of these pre-fetches is used in actual execution. As well, each transaction is penalized 5000 instructions per data item as an overhead cost involved in maintaining EMA. In the results presented below this costing scheme is called costing 1.

Under the third costing scheme 10% of pre-fetched transactions have to re-access their data from disk once a transaction begins its real execution. In the results presented below this costing scheme is called costing 2. As in costing 1, in costing 2 each transaction is penalized 5000 instructions per data item as an overhead cost involved in maintaining EMA. The added costs allowed by costing regimes 1 and 2 are extreme as shown by the analysis of section 5.3. Thus, this series of tests provide a good indication of the robustness of EMA.

The results of this series of tests of EMA are shown in Figures 7.10 to 7.17 below. As in the previous section, there are 2 graphs for each hardware configuration – one showing total throughput and the other breaking down total throughput by transaction type. In the graphs shown in this section, the number of items shown on each graph is quite large and consequently the legend is quite large and will not fit on the graphs breaking down throughput by transaction type. The first series of 2 graphs show the results for the disk-based configurations containing 96 processors operating at 100 MIPS per processor.

The results shown in Figures 7.10 and 7.11 indicate that for the hardware configuration containing 96 processors each operating at 100 MIPS, the addition of significant costs to EMA only affects its performance marginally. In fact, the type of concurrency control used seem to affect performance for each transaction type more than the costs applied to EMA.

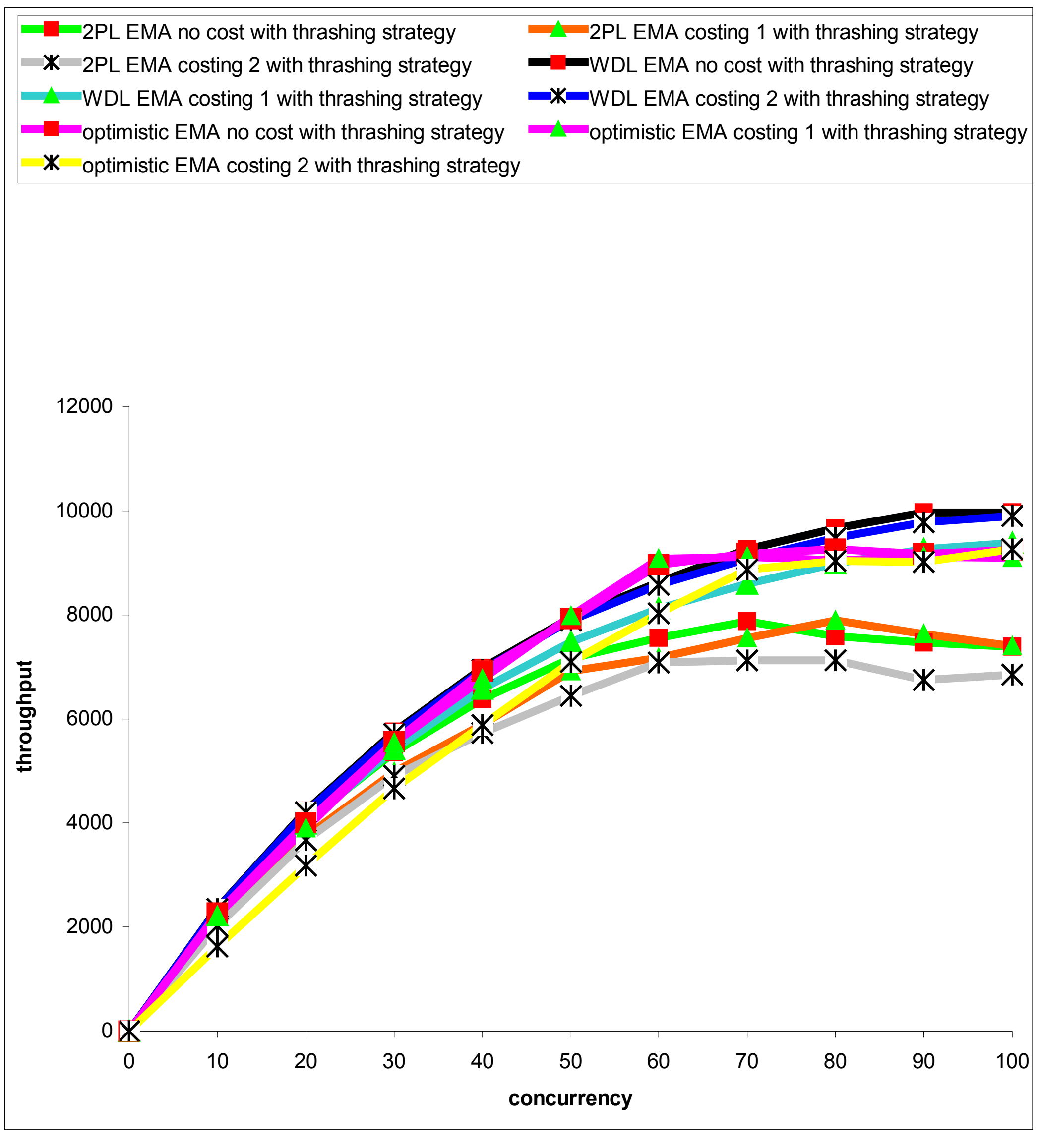
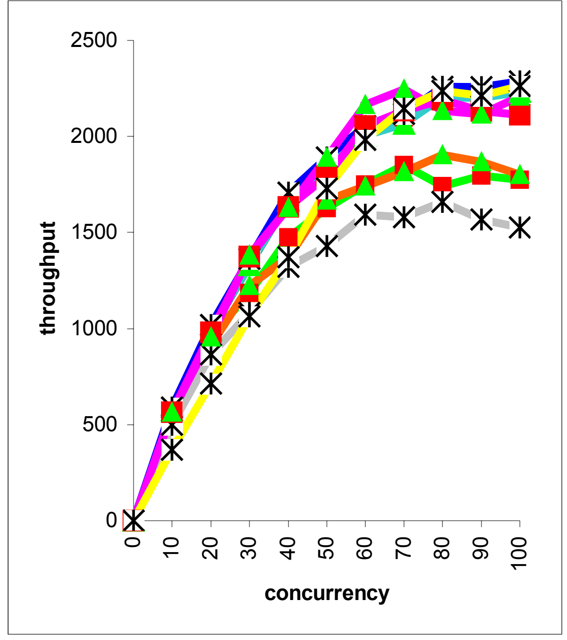
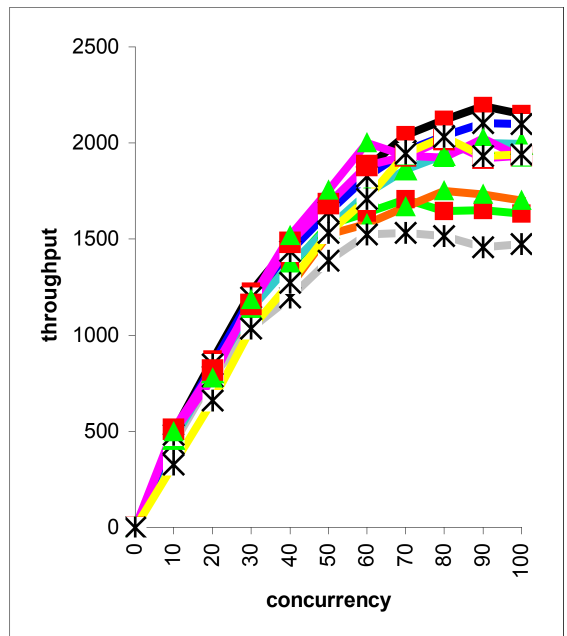


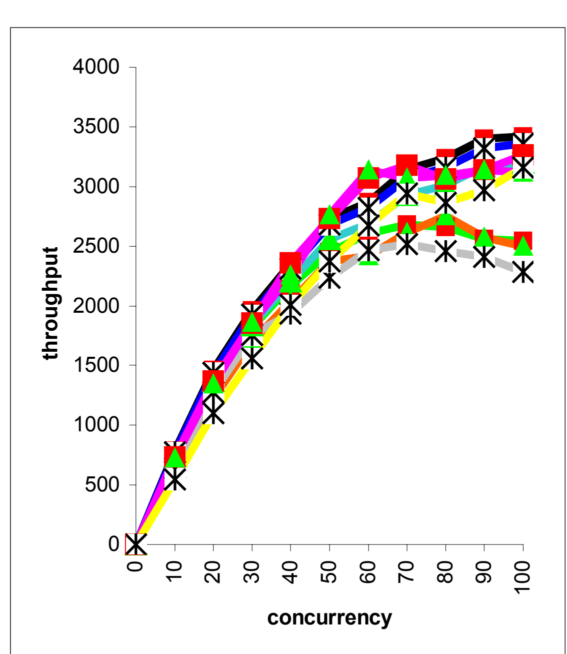

**Figure 7.10. Total throughput with diverse costing regimes in systems containing 96 processors each operating at 100 MIPS**

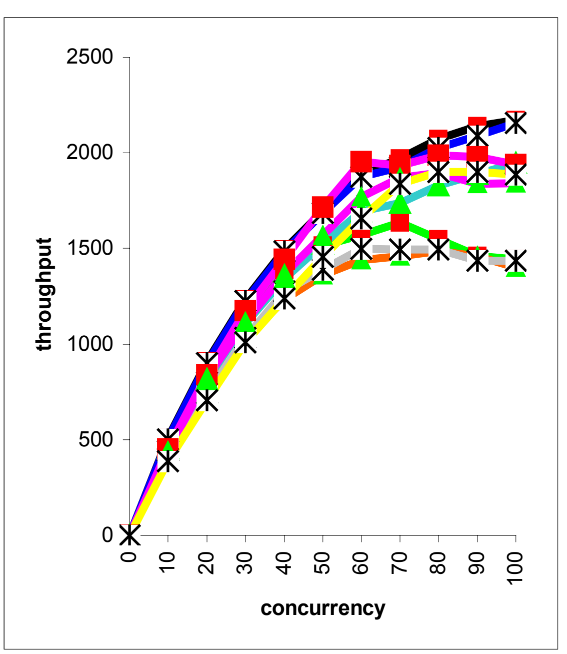



$T_1$ transactions

$T_2$ transactions




$T_3$ transactions

$T_4$ transactions

**Figure 7.11. Throughput with diverse costing regimes in systems containing 96 processors each operating at 100 MIPS by transaction type.**

The pattern shown in Figures 7.10 and 7.11 is repeated for each hardware configuration as shown by the following graphs. In all hardware configurations, costing regime 2 seems to be particularly robust. This can be explained by the fact that the repeated disk access by some transactions at execution time is partially offset by the increased ability of the system to process pre-fetches while waiting for the executing transactions to re-access their data from disk.

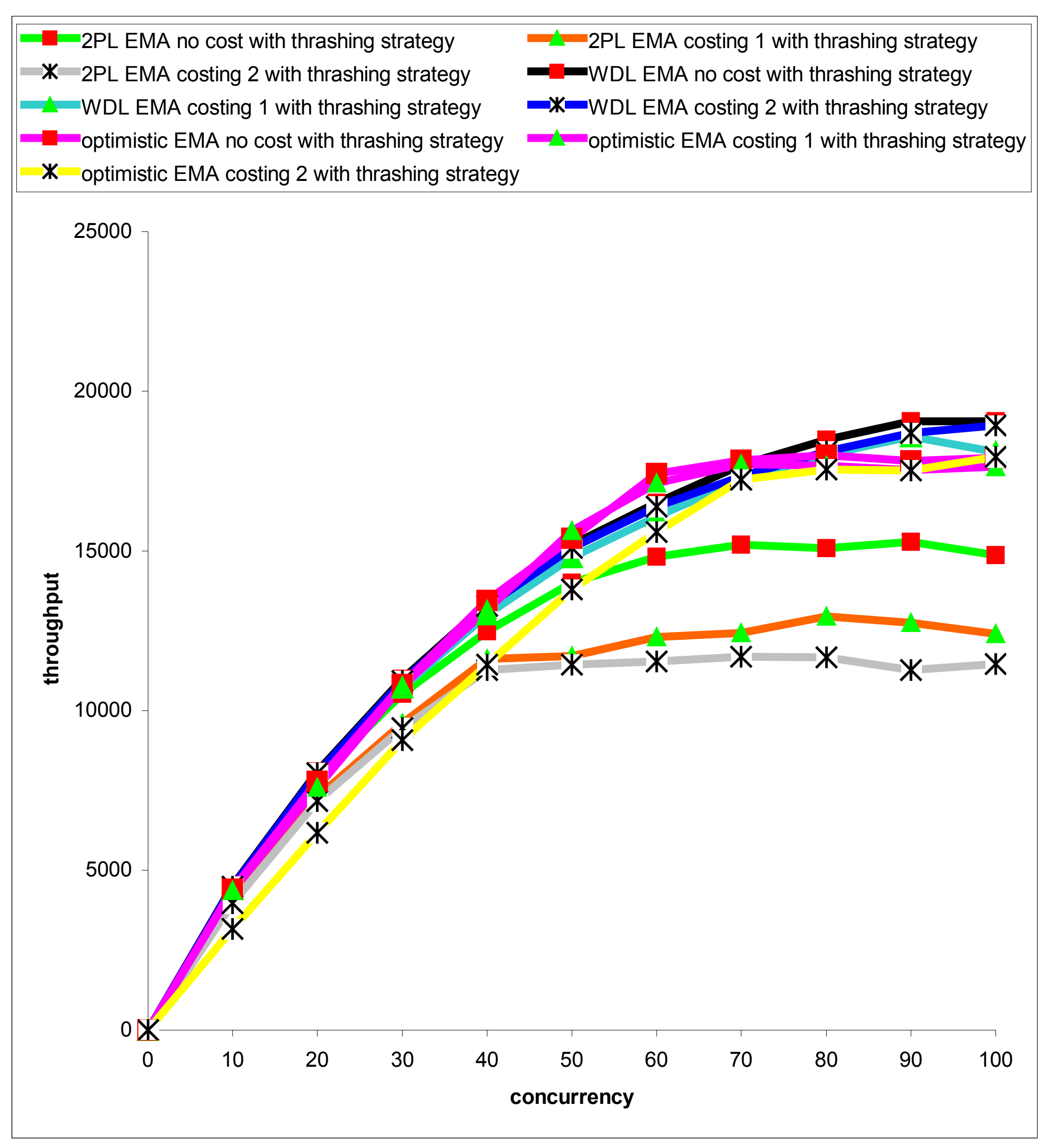
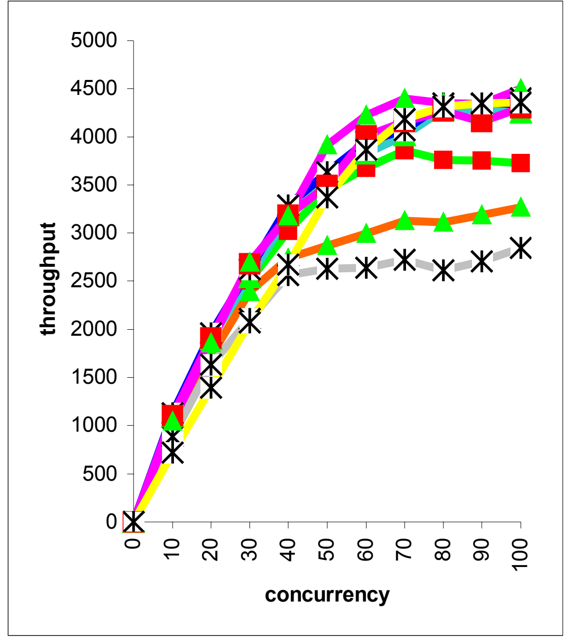
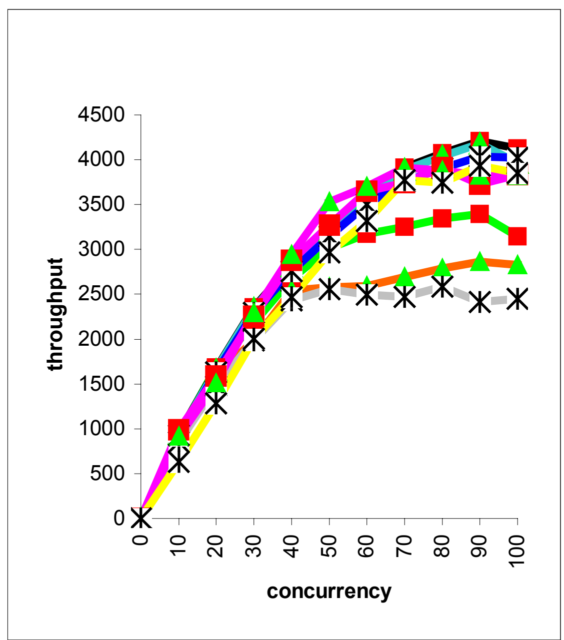


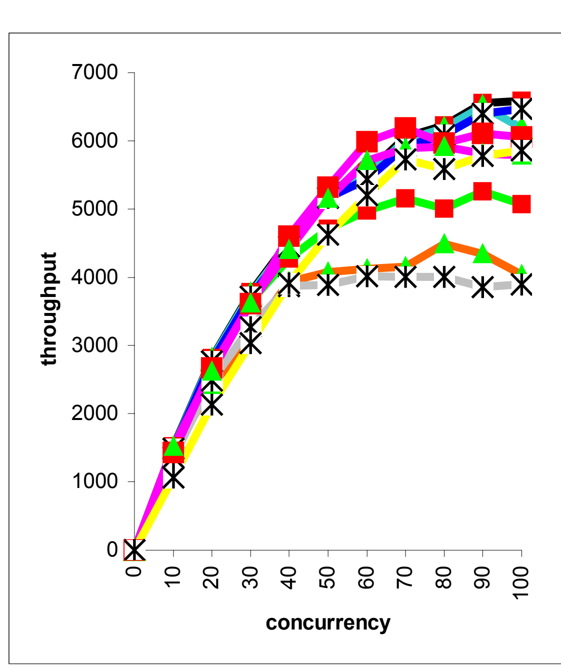

**Figure 7.12. Total throughput with diverse costing regimes in systems containing 96 processors each operating at 200 MIPS**



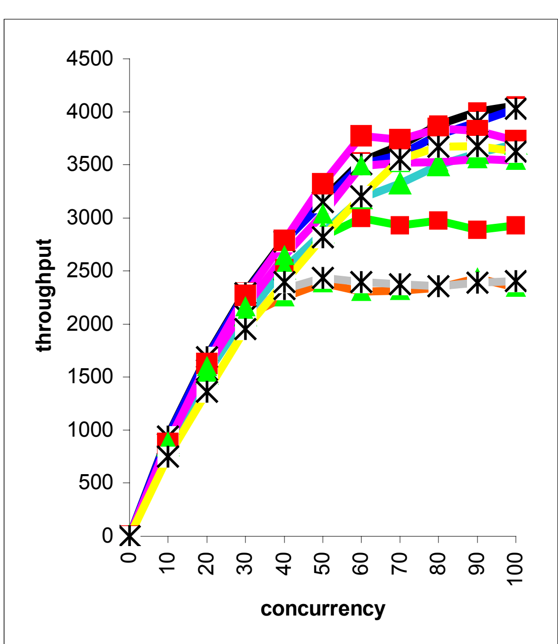

$T_1$ transactions




$T_2$ transactions




$T_3$ transactions




$T_4$ transactions

**Figure 7.13. Throughput with diverse costing regimes in systems containing 96 processors each operating at 200 MIPS by transaction type**

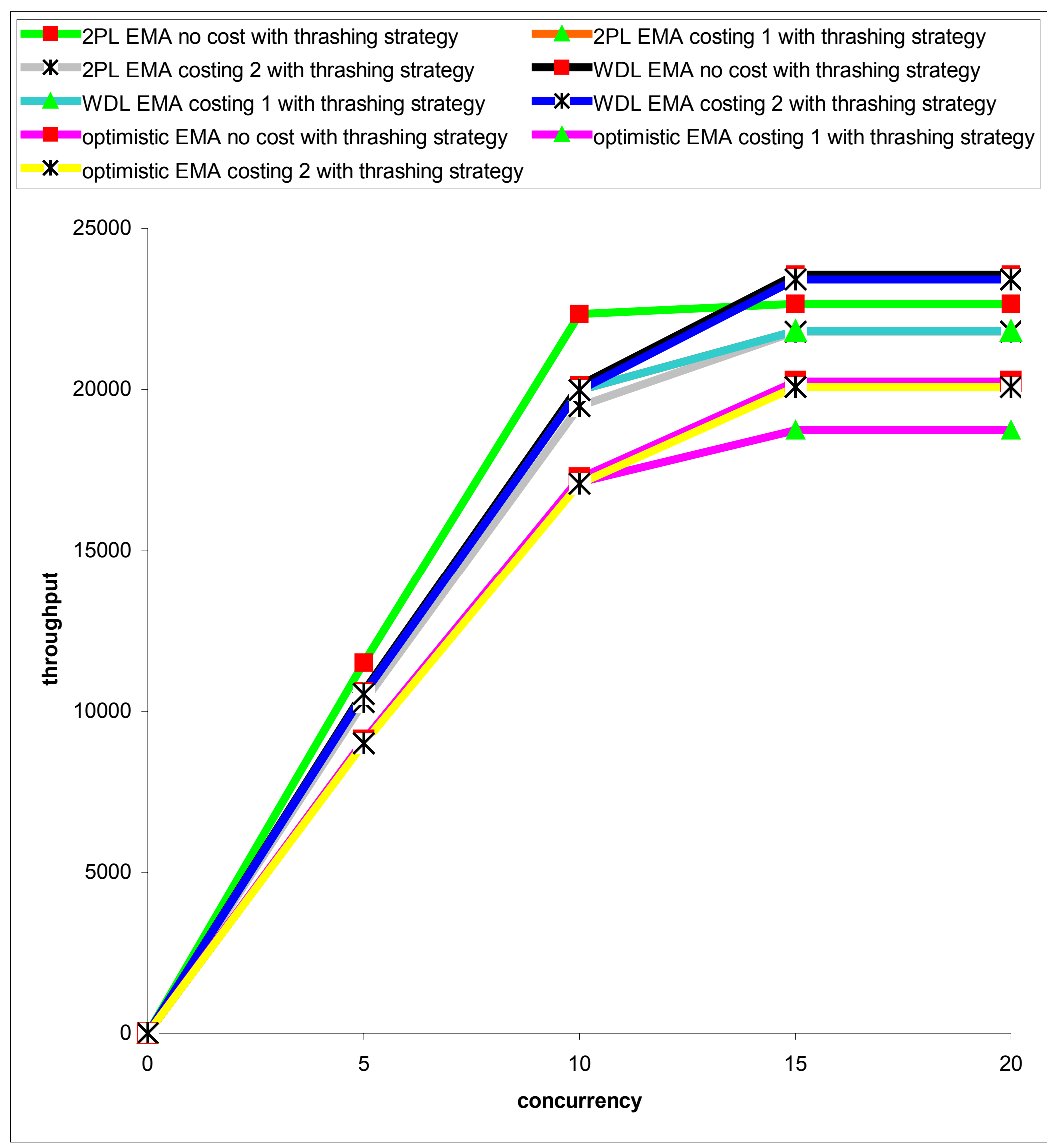



**Figure 7.14. Total throughput with diverse costing regimes in systems containing 20 processors each operating at 1000 MIPS**

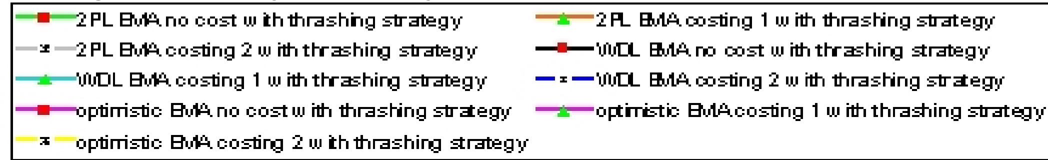
$T_1$ transactions

$T_2$ transactions

$T_3$ transactions

$T_4$ transactions

**Fig 7.15. Throughput with diverse costing regimes in systems containing 20 processors each operating at 1000 MIPS by transaction type**



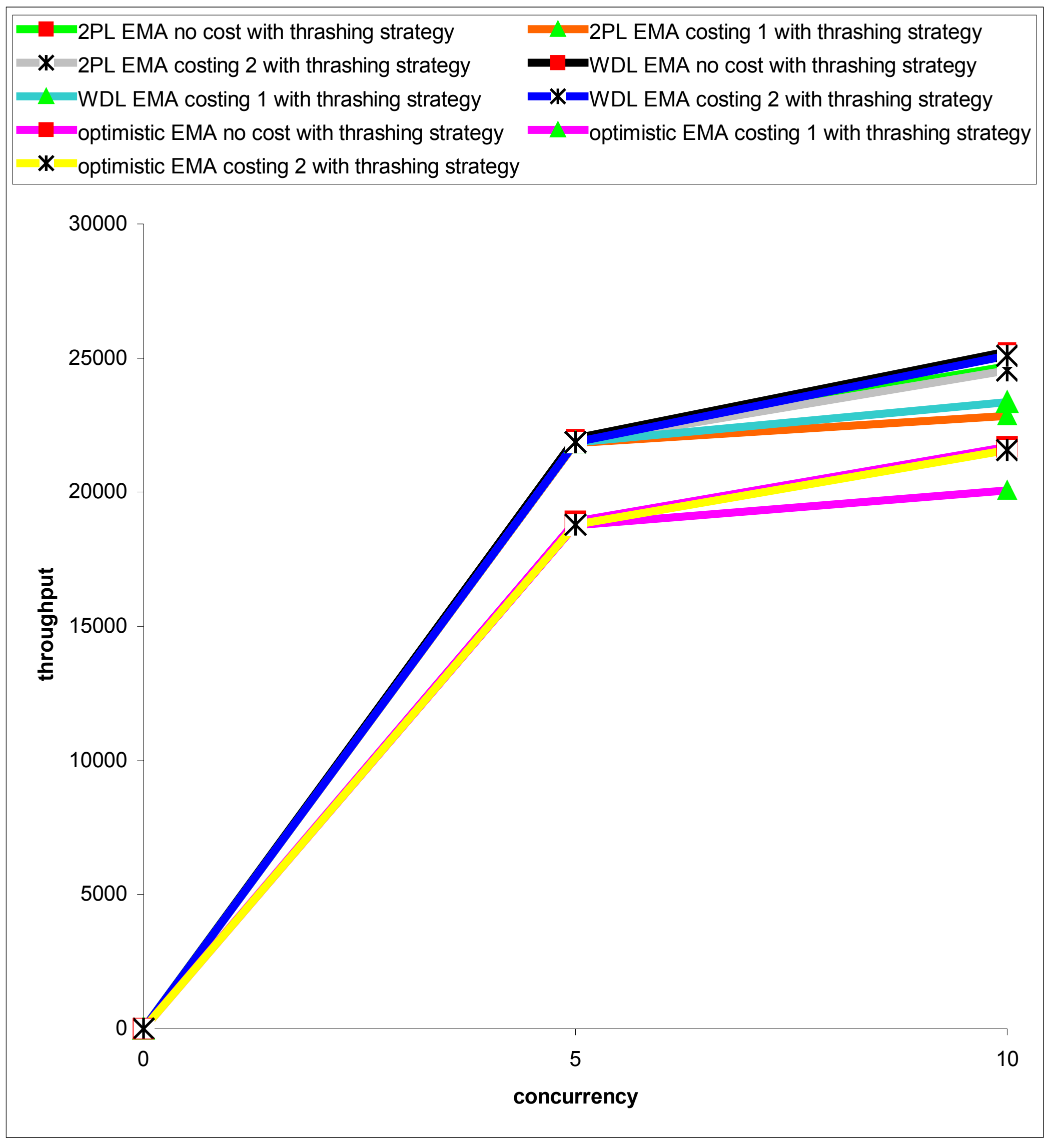
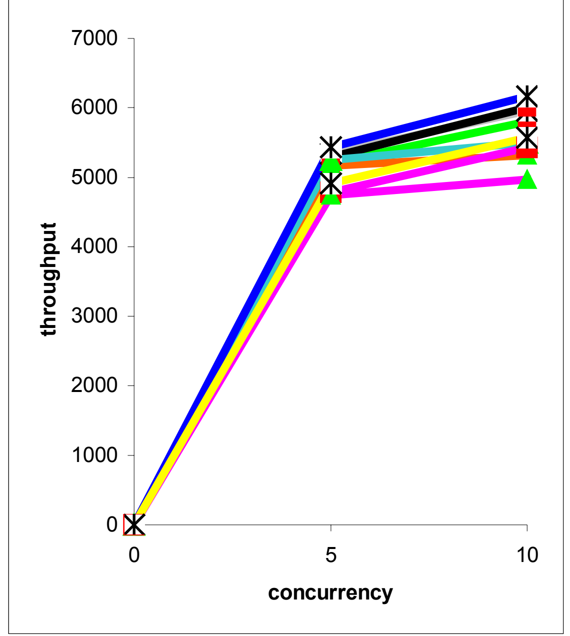
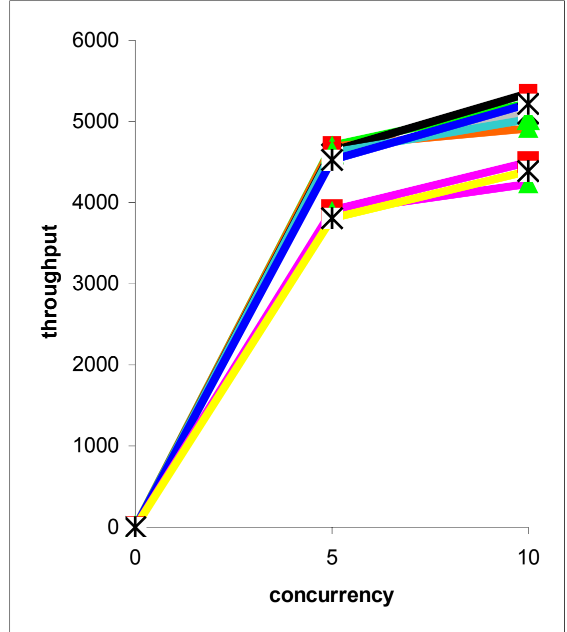



**Figure 7.16. Total throughput with diverse costing regimes in systems containing 10 processors each operating at 2000 MIPS**




$T_1$ transactions




$T_2$ transactions

throughput

concurrency

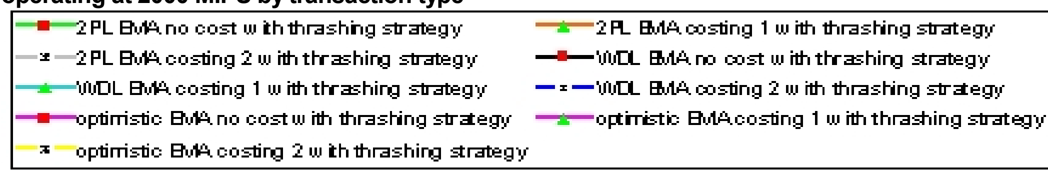
$T_3$ transactions

throughput

concurrency

$T_4$ transactions

**Figure 7.17. Throughput with diverse costing regimes in systems containing 10 processors each operating at 2000 MIPS by transaction type**



Overall, since under all tested hardware configurations the addition of significant costs to EMA only affects its performance marginally, one can conclude that this series of tests indicates that the performance of EMA is robust to significant implementation costs.

## 7.4 2PL EMA Systems With and Without Thrashing Control

As indicated in chapters 3 and 4, 2PL systems are particularly prone to thrashing. This problem increases with contention and concurrency, but, as shown in chapter 3, even in systems with low concurrencies a very low probability of thrashing at any point in time can still translate to an unacceptable incidence of thrashing given a sufficient number of transactions and processing cycles. In this section we compare the performance of in-memory and EMA 2PL systems with and without thrashing control. The thrashing control mechanism used is the immediate re-start of all blocked transactions once the proportion of blocked transactions exceeds 0.378 since this seems to be the most successful method in high-speed systems. For a comparison of the performance of the various thrashing control strategies see Appendix B.

The results of this series of tests are shown in Figures 7.18 to 7.25 below. As in the previous sections, there are 2 graphs for each hardware configuration – one showing total throughput and the other breaking down total throughput by transaction type. The first series of 2 graphs show the results for the disk-based configurations containing 96 processors operating at 100 MIPS per processor. The second series of 2 graphs show the

results for the disk-based configurations containing 96 processors operating at 200 MIPS per processor. The third series of 2 graphs show the results for the disk-based configurations containing 20 processors operating at 1000 MIPS per processor while the last series of 2 graphs show the results for the disk-based configurations containing 10 processors operating at 2000 MIPS per processor.

Past a concurrency of around 20, the results for the configurations containing 96 processors operating at 100 MIPS per processor and 96 processors operating at 200 MIPS per processor show a dramatic superiority in the performance of the 2PL systems using thrashing control as against those not using thrashing control with the peak throughput of the systems using thrashing control being around double that of the systems not using thrashing control. This pattern is consistent for all transaction types.

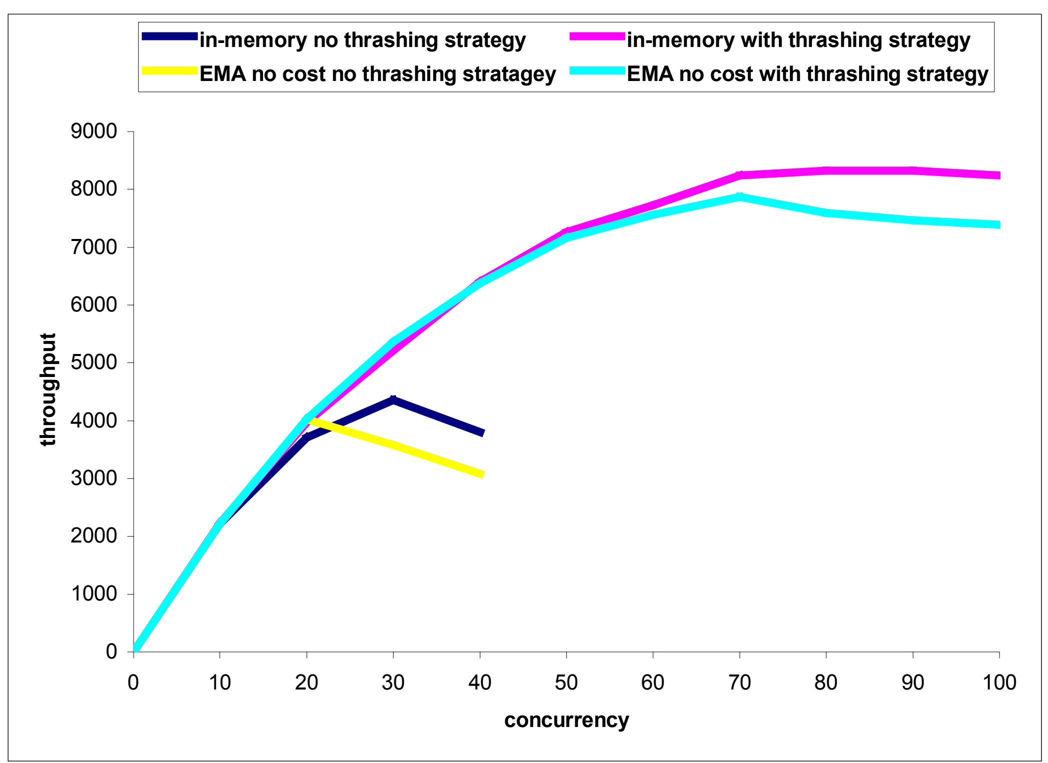
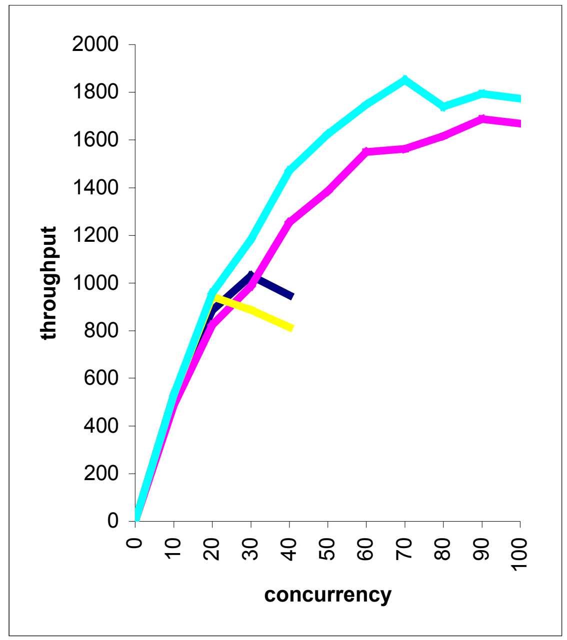
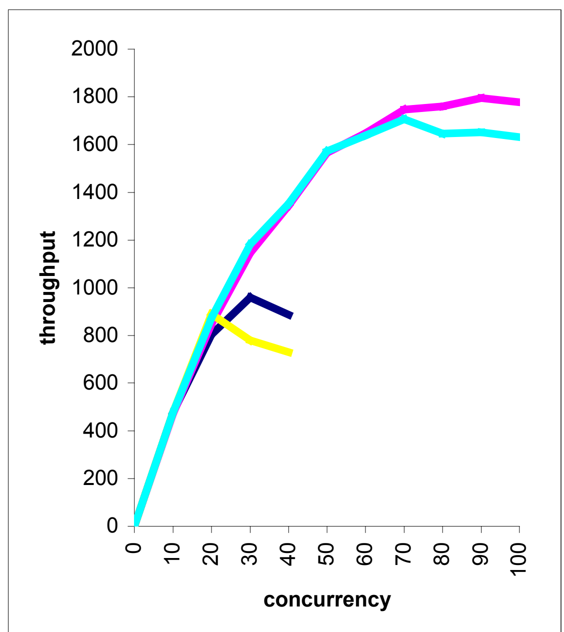


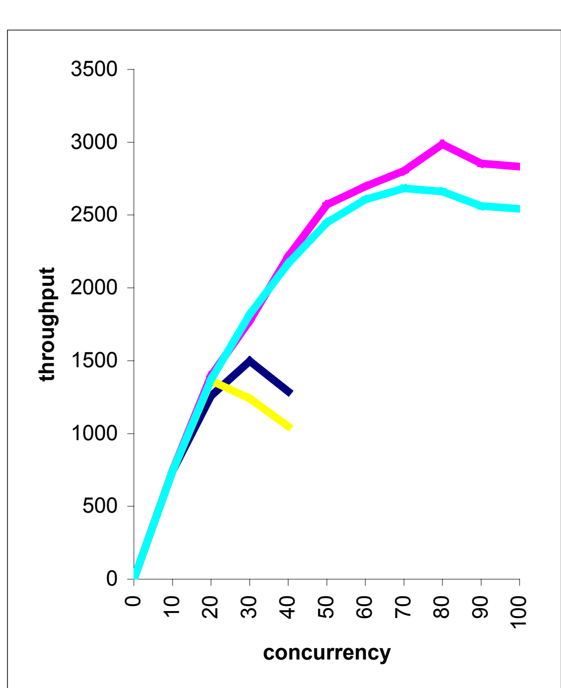

**Figure 7.18. Total throughput in 2PL systems with without thrashing control in systems containing 96 processors each operating at 100 MIPS**

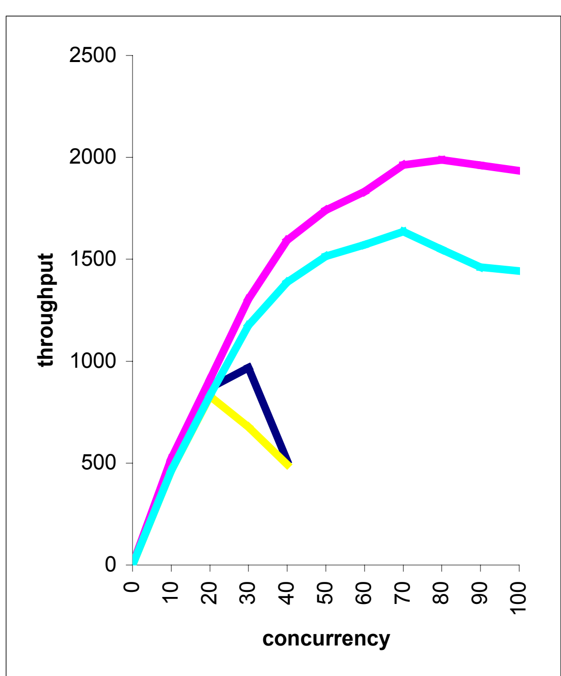



$T_1$ **transactions**

$T_2$ **transactions**




$T_3$ **transactions**

$T_4$ **transactions**

**Figure 7.19. Throughput in 2PL systems with without thrashing control in systems containing 96 processors each operating at 100 MIPS by transaction type**

in-memory no thrashing strategy
in-memory with thrashing strategy
EMA no cost no thrashing stratagey
EMA no cost with thrashing strategy

The restart of transactions has two effects on performance, firstly it limits the length of wait for queues thus increasing throughput and secondly it eliminates thrashing thus increasing the average of throughputs achieved. In the case of the configurations containing 96 processors operating at 100 MIPS per processor and 96 processors operating at 200 MIPS per processor, the latter reason dominates.

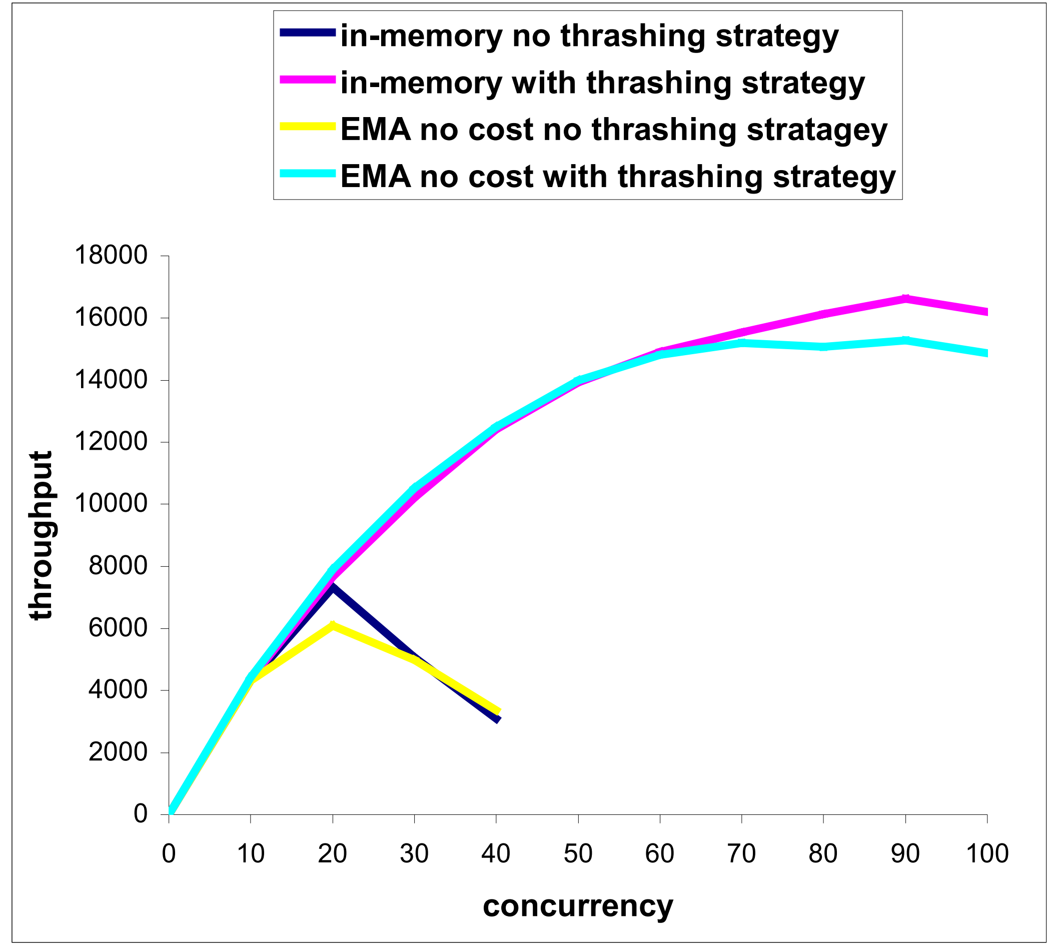
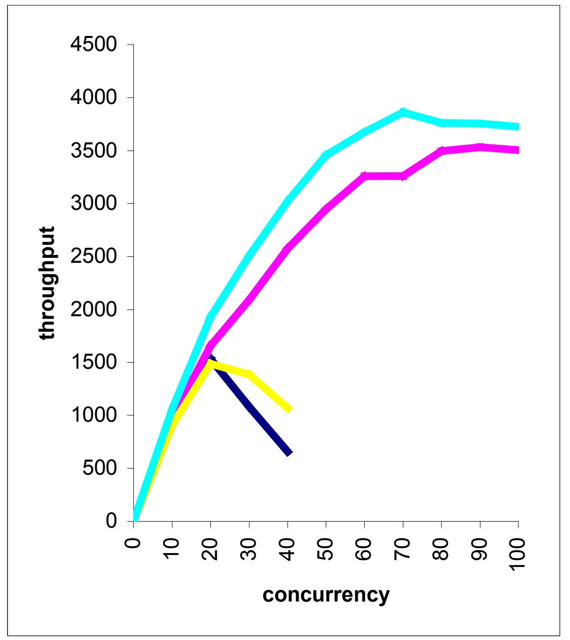
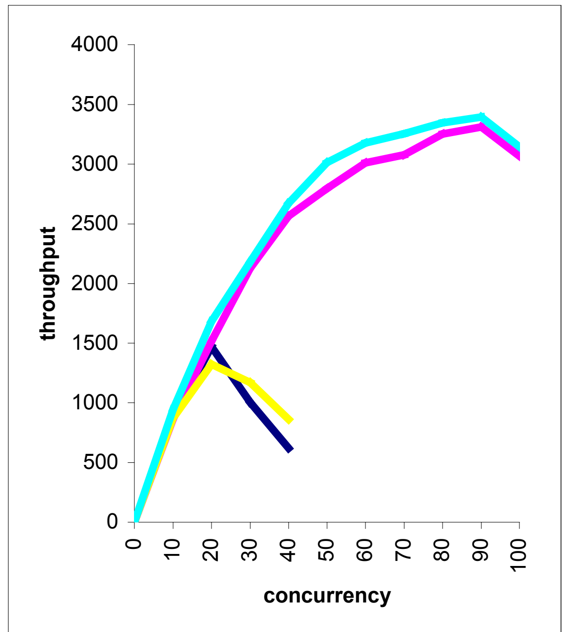
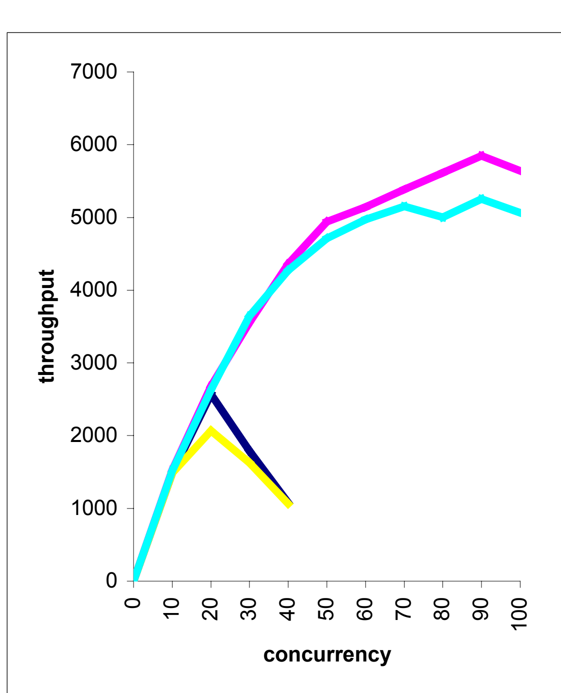


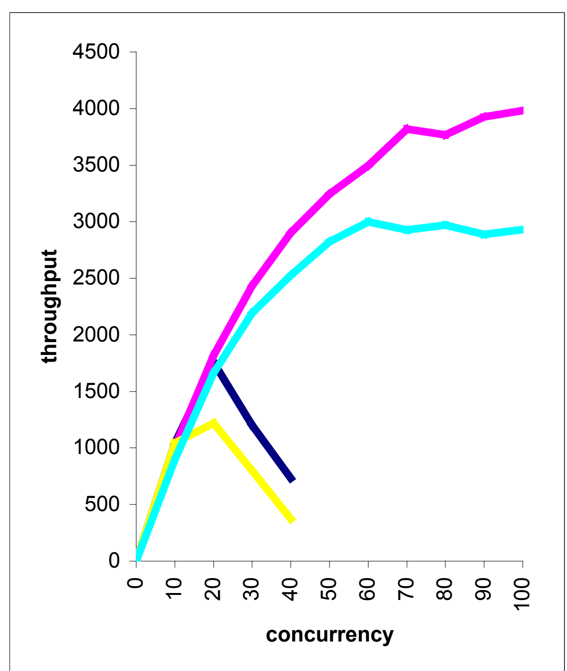

**Figure 7.20. Total throughput in 2PL systems with without thrashing control in systems containing 96 processors each operating at 200 MIPS**




$T_1$ **transactions** $T_2$ **transactions**




$T_3$ **transactions** $T_4$ **transactions**

**7.21. Throughput in 2PL systems with without thrashing control in systems containing 96 processors each operating at 200 MIPS by transaction type**

in-memory no thrashing strategy
in-memory with thrashing strategy
EMA no cost no thrashing stratagey
EMA no cost with thrashing strategy

For example, in the configuration containing 96 processors operating at 200 MIPS per processor under EMA at a concurrency of 20, total throughput without thrashing control implemented is 6238. This is an average over 40 runs. Of these 40 runs, 26 complete without thrashing and the average throughput for these runs is 7522. The remaining14 runs thrash and the average throughput for these is 3597. The average throughput for this configuration under EMA at a concurrency of 20 with thrashing control implemented is 7907 – that is just over 400 more than the runs without thrashing control implemented where thrashing did not occur. Thus at this concurrency the effect of thrashing control in eliminating thrashing is an increase in throughput of around 3500 transactions while the effect of thrashing control in shortening wait queues is an increase in throughput of around 400 transactions.

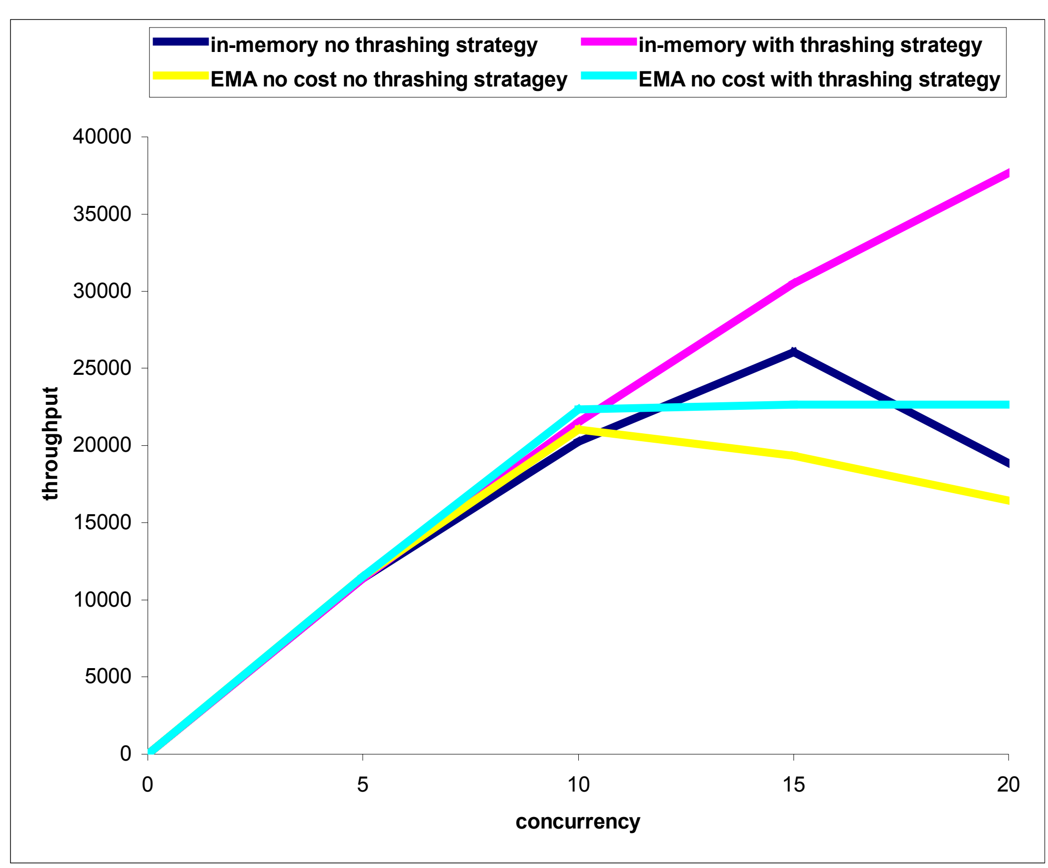
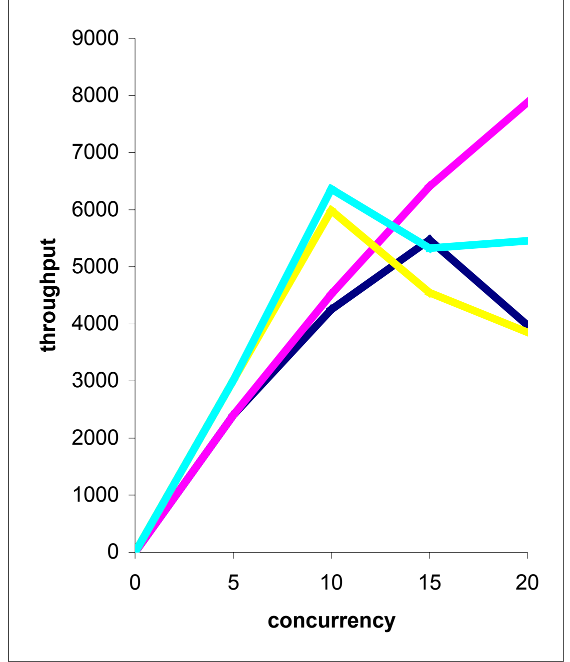
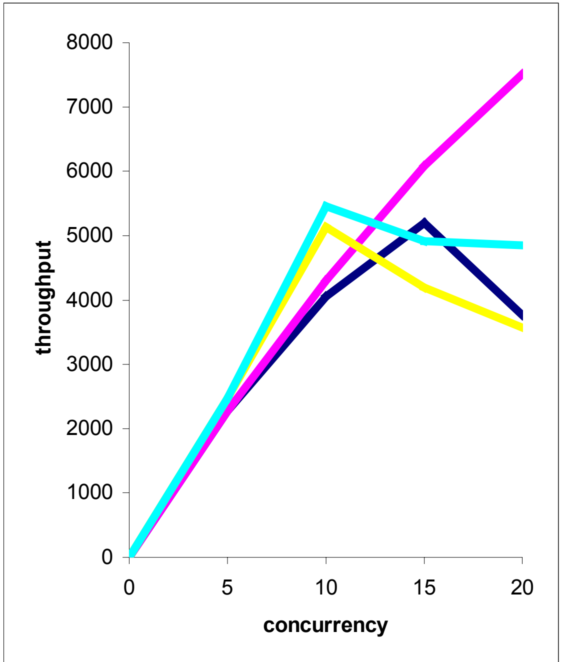


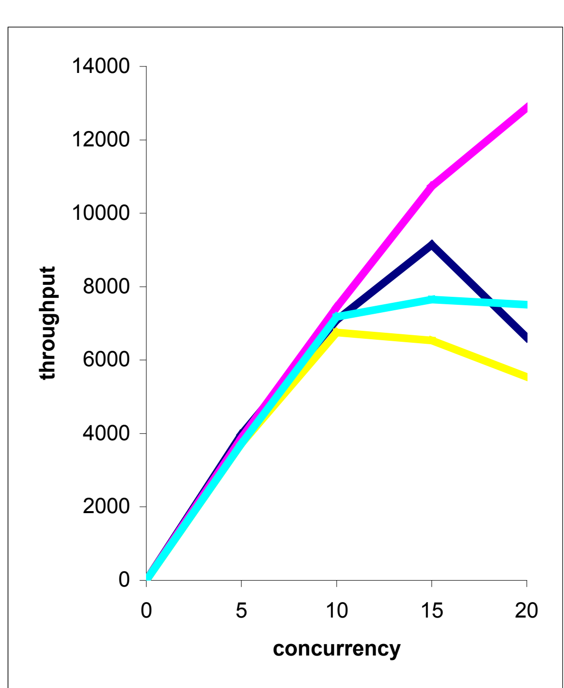

**Figure 7.22. Total throughput in 2PL systems with without thrashing control in systems containing 20 processors each operating at 1000 MIPS.**



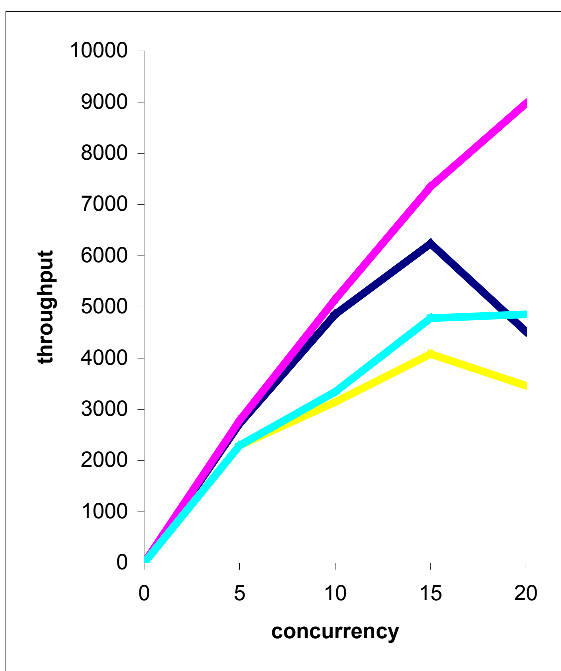

$T_1$ **transactions**




$T_2$ **transactions**




$T_3$ **transactions**




$T_4$ **transactions**

**Figure 7.23. Throughput in 2PL systems with without thrashing control in systems containing 20 processors each operating at 1000 MIPS by transaction type.**

in-memory no thrashing strategy | in-memory with thrashing strategy
EMA no cost no thrashing stratagey | EMA no cost with thrashing strategy

The advantage of thrashing control diminishes with the reduction of concurrency. Thus in the configurations composed of 20 processors operating at 1000 MIPS, the advantage of thrashing control only becomes marked in the in-memory system past a concurrency of 15. Similarly, while the advantage of thrashing control for EMA is clear, the performance of EMA is affected more by the ceiling imposed by the hardware than it is by the implementation of thrashing control. In the configurations composed of 10 processors operating at 2000 MIPS, because of the very low concurrencies involved, the effect of thrashing control on throughput is very small and differences in performance are dominated by hardware considerations.

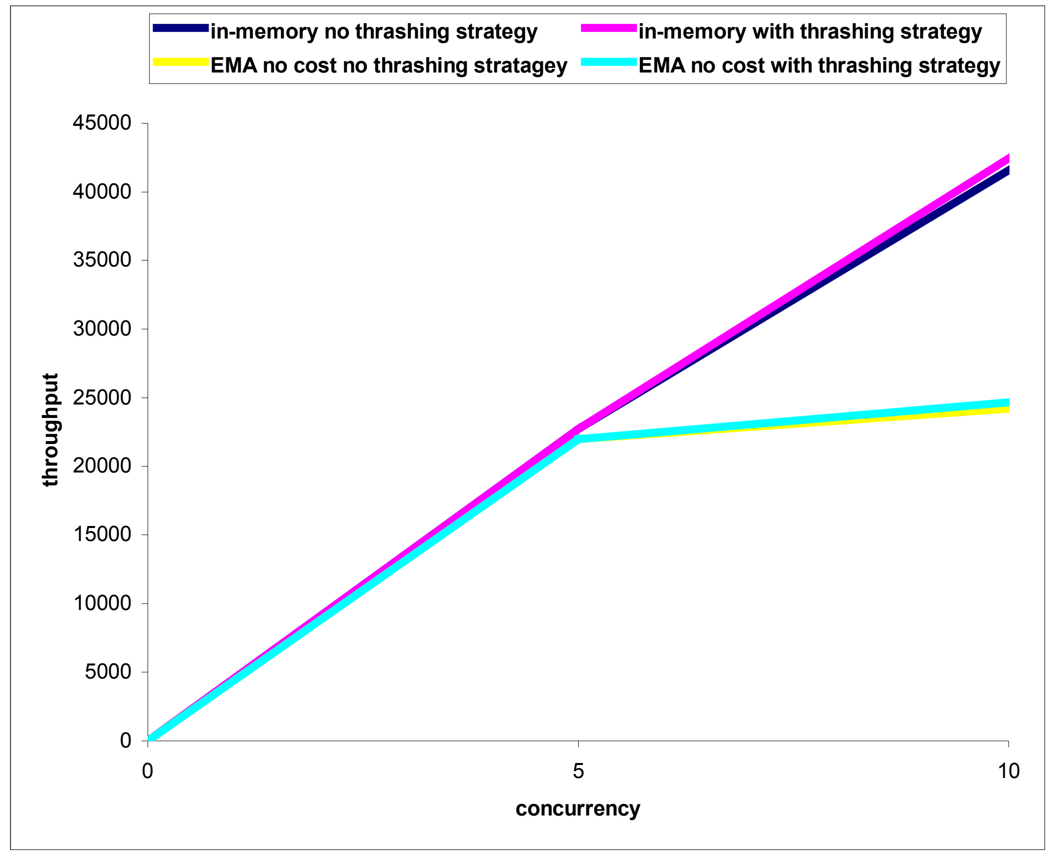
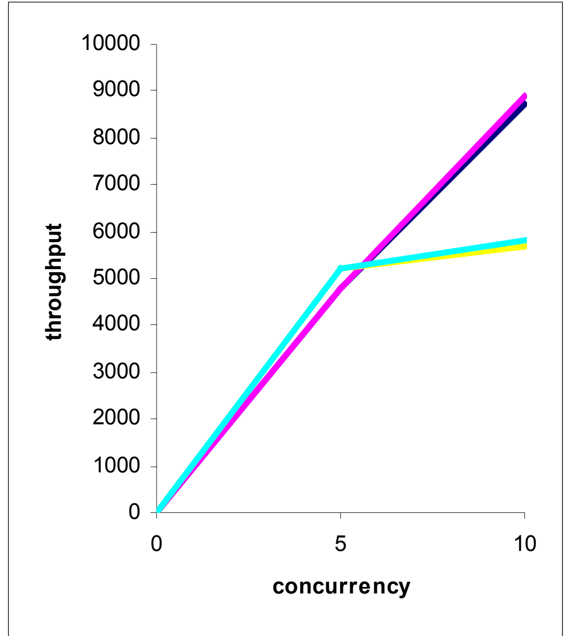
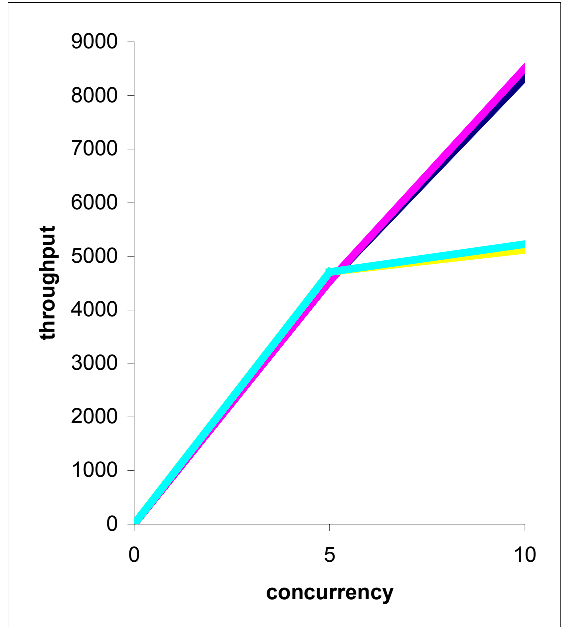


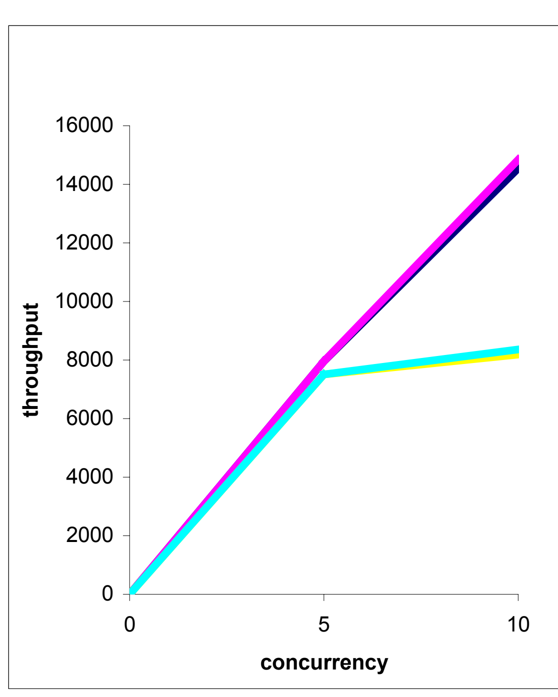

**Figure 7.24.. Total throughput in 2PL systems with without thrashing control in systems containing 10 processors each operating at 2000 MIPS**



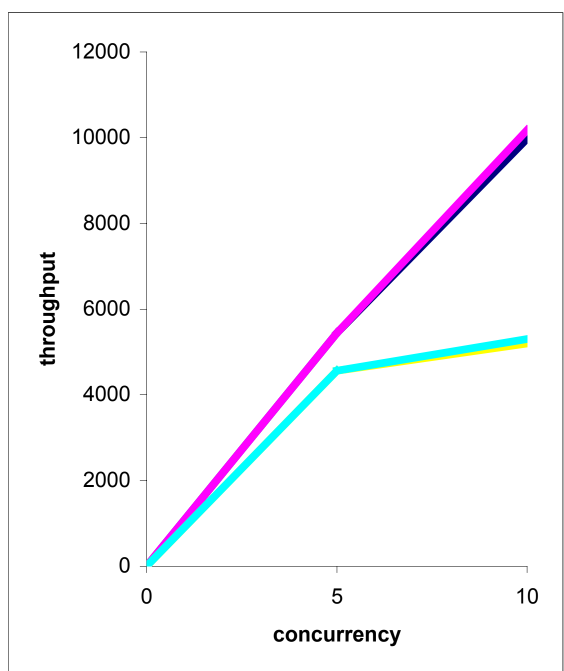

$T_1$ transactions




$T_2$ transactions




$T_3$ transactions




$T_4$ transactions

**Figure 7.25. Thrashing control in systems containing 10 processors each operating at 2000 MIPS by transaction type.**

in-memory no thrashing strategy | in-memory with thrashing strategy
EMA no cost no thrashing stratagey | EMA no cost with thrashing strategy

## 7.5 Summary

The result of the tests presented in this chapter indicate that when used in conjunction with all concurrency control mechanisms, EMA can increase the throughput of disk-based systems to levels quite close to those achieved by in-memory system using an equivalent concurrency control mechanisms. This performance is far better than can be achieved with any disk-based concurrency control mechanism that does not use EMA.

Further, results presented in section 7.3 showed that the performance of EMA was very robust to the imposition of additional costs associated with its implementation. Indeed, the addition of cost penalties far in excess of what one could reasonably expect in commercial applications reduced the performance of EMA by a very small margin. Besides preventing thrashing, the use of thrashing control, improved the performance of 2PL systems. In the systems with the fastest hardware at low concurrency, the performance of the 2PL systems was comparable to that achieved with WDL and optimistic concurrency control. At higher concurrencies, with thrashing control, the performance of 2PL was respectable when compared to WDL. However without thrashing control, the throughput of 2PL actually declined once the peak concurrency threshold was passed.

# *Chapter 8*

## *Conclusion*

## 8.1 Contributions and Achievements

In recent years, there have been several important developments in the general computing industry. These developments have also impinged on both the actual and potential performance requirements and capacity of transaction processing systems. The start of an expected explosion in e-commerce has already led to applications such as on-line share trading, which require much higher throughputs than have been available in the past. The development of cheap memory has made systems with memory capacities that in the past were considered huge, quite commonplace. The combination of these two factors has led to the recent development of in-memory database systems. The aim of this thesis has been to examine the potential performance of in-memory systems under

various hardware configurations and concurrency control schemes relative to equivalent disk-based systems and to determine whether there was any way of bringing the performance of disk-based systems closer to that of in-memory systems. We summarize the major points and contributions of this thesis as follows.

- **Disk-Based Versus In-Memory Systems**

In this thesis we tested a number of concurrency control mechanisms operating in a variety of hardware configurations. The most interesting results of these tests were the extraordinary rates of throughput achievable by in-memory systems. Under 2PL, WDL and optimistic concurrency control, massive throughputs of over 40000 transactions per second were achieved in some of our in-memory systems. Throughputs under all concurrency control mechanisms in the in-memory systems with speeds over 100 MIPS per CPU were well above the best results achieved by any concurrency control method in the disk-based systems.

The results of these tests indicate that the potential performance of concurrency control mechanisms in in-memory systems, rather than the performance of different concurrency control methods under equivalent disk-based hardware should form the benchmark for judging the merit of a transaction processing system. We believe that the gap in performance between any concurrency control mechanism in in-memory systems and the best performance of the most successful conventional concurrency control mechanism in disk-based systems makes this conclusion inescapable.

- **Contention-Based Scheduler**

The results outlined above suggest that for those organizations that could fit all their data in memory, the best way of improving the performance of their transaction processing systems is to switch to in-memory systems rather than improve the performance of their disk-based systems. However, many organizations cannot fit all their data in memory and for these organizations, improvement in the performance of their transaction processing systems can only come about by improving the performance of disk-based systems.

The first mechanism we developed to improve the performance of disk-based 2PL systems was the contention-based scheduler. This scheduler operated by measuring transactions' contention as they arrived sorting these transactions into queues of transactions with a similar contention. It then manipulated the number of transactions allowed into the system by the contention class. We outlined the mechanisms required by this scheduler for its operation. These mechanisms included –

1. A mechanism that could enable the scheduler to estimate transaction's contention reasonably cheaply.
2. A method for calculating transactions' contention that was independent of concurrency and that could calculate contention in situations where not all objects in data stores have an equal probability of being accessed
3. A method for determining the maximum number of transactions of each type allowed into the system.

4. A means of compensating higher contention transactions for their subsidy to lower contention transactions.

Tests showed that the contention-based scheduler substantially outperformed standard 2PL concurrency control in a wide variety of disk-based hardware configurations. The improvement though most pronounced in the throughput of low contention transactions extended to all transaction types over an extended processing period.

- **EMA**

The second mechanism we developed to improve the performance of disk-based systems was the enhanced memory access (EMA) system. Indeed, we believe that our proposal for EMA, which improves the performance of disk-based systems to near that achieved by in-memory systems is probably the major contribution of our thesis.

The purpose of EMA was to allow very high levels of concurrency in the pre-fetching of data thus bringing the performance of disk-based systems close to that achieved by in-memory systems. The basis of our proposal for EMA was to ensure that even if conditions for satisfying a predicate changed between pre-fetch time and the time when actual execution took place, the data required to satisfy transactions' predicates were nevertheless still be found in memory. This pre-fetching could be used as a front end for 2PL, WDL or optimistic concurrency control.

The main policy that had to be effected to implement EMA was that no data item could be flushed from memory unless the time-stamp of its last access exceeded the timestamp of the oldest transaction in the system. Besides this policy, consideration was

also given as to what sort of data needed to be pre-fetched in order to ensure that no matter how a predicate's value changed, all required data was in memory. We presented an implementation that addressed all the requirements of EMA.

In this implementation, a new transaction entering the system is time-stamped on arrival and begins an immediate virtual execution to pre-fetch all its required objects. If an object that it requires is already in memory having been previously acquired by another transaction, the new transaction updates that object's timestamp. If an object is not found in memory, it is retrieved from disk and time-stamped. An object's timestamp is also changed by a transaction that has used it. Naturally, a committing transaction is the only transaction that can change the value of a data item that is available to all transactions. All other transactions can only modify copies of objects in their own workspace. An object cannot be flushed from memory until its timestamp is older than that of the oldest transaction in the system. This ensures that a current version of all the objects that a transaction requires are in memory by the time it begins its actual execution with modest memory requirements elative to the capacity of modern hardware systems.

Tests showed that the implementation of EMA allowed the performance of disk-based systems to approach that achieved by in-memory systems. Further, the tests showed that the performance of EMA was very robust to the imposition of additional costs associated with its implementation.

- **Thrashing**

We showed that in systems using 2PL concurrency control, thrashing could be a serious problem even at low lock contentions if the availability of transactions is such that

the system can continuously process for a very large number of cycles. In in-memory systems, this large number of cycles could occur in a very short period of real time. To control this thrashing we suggested several devices. Testing showed that the most successful thrashing control policy was to simply restart all blocked transactions once the proportion of locked transactions exceeded 0.378. This policy was most successful in in-memory and EMA systems. A variation of this policy, which restarted blocked transactions by transaction class, was successfully used with the contention-based scheduler.

## 8.2 Future Research

While the EMA system outlined in this thesis improves the throughput of disk-based systems quite dramatically, the proposal presented in this thesis is open to further research. One of the issues that need to be determined more precisely than has been done in this thesis is the actual nature of predicates that are used in commercial applications. For example, for our tests we arbitrarily set a fixed number of transactions that had forked decisions – the actual proportions of such transactions that are common in commercial applications needs to be established.

Another approximation used in this thesis was the costs of overheads for EMA. In our tests we arbitrarily set the overhead costs to 5000 instructions per object processed. Further research is required for a more precise assessment of actual costs.

Another question that remains open with regards to EMA is whether the implementation suggested in this thesis is the best one. For example in the implementation outlined in this thesis, the management of data and transactions was decentralized – that is, each transaction had a set of required activities which included the maintenance of transaction information and the flushing of data. An alternative implementation could have a dedicated processor to control the management of data and transaction information. For example, a garbage collector could do data flushing periodically.

# *Bibliography*

1. Adya, A., Gruber, R., Liskov, B. and Maheshwari, U.,“ Efficient Optimistic Concurrency Control Using Loosely Synchronised Clocks”, Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, Vol. 24 No. 2, San Jose California, May 1995, pp 23-34

2. Agrawal, R., Carey, M.J., and Livney, M, “Concurrency Control Performance Modelling: Alternatives and Implications”, ACM Transactions on Database Systems, Vol.12, No.4, December 1987, pp 609 - 654

3. Bell, D.A., “Difficult Data Placement Problems”, The Computer Journal, Vol 27, No. 4, 1984, pp 315-320

4. Bernstein, P.A., Hadzilacos, P. and Goodman, N., Concurrency Recovery and Control in Database Systems, Addison-Wesley, Reading MA, 1987

5. Bhargava, B., "Concurrency Control in Database Systems"”, IEEE Transactions on Knowledge and Data Engineering, Vol. 11 No. 1, January/February, 1999, pp 3-16

6. Bohannon, P., McIlroy, P., and Rastogi, R., "Main-Memory Index Structures with Fixed-Size Partial Keys", Proceedings of the ACM SIGMOD International Conference on Management of Data, Vol. 30, No 2, Santa Barbara, California, May 21 - 24, 2001, pp 163-174

7. Borr, A., " Transaction Monitoring in Encompass [TM]: Reliable Distributed Transaction Processing", Proceedings of the 7th International Conference on Very Large Data Bases, Cannes, France, September 1981, pp 155 -165

8. Borr, A., "Robustness to Crash in a Distributed Database: A Non Shared-Memory Multi-Processor Approach", Proceedings of the 10th International Conference on Very Large Data Bases, Singapore, August 1984, pp 445 -453

9. Carey, M.J., Krishnamurthi, S. and Livny, M., "Load Control for Locking: The 'half and half' Approach", The 9th ACM Symposium on the Principles of Database Systems, Nashville Tenessee, April, 1990, pp 72-84

10. Chakrabarti, K., and Mehrotra, S., "Efficient Concurrency Control in Multidimensional Access Methods", Proceedings of ACM SIGMOD International Conference on Management of Data, Vol. 28, No. 2, Philadephia, Pennsylvania, June, 1999, pp25-36

11. Daniels, D.S., Spector, A.Z. and Thompson, D.S., "Distributed Logging for Transaction Processing", ACM SIGMOD Record, Vol. 16, No. 3, December, 1987, pp 82 - 95

12. Griffeoen, j, Anderson, T, Breitbart, Y. and Vingralek, R., “DERBY: A Memory Management System for Distributed Main Memory Databases”, retrieved March 5 2000 <http://www.dcs.uky.edu/~griff/papers/derby-ride96/html.html>

13. DeWitt, D.J., Katz, R.H., Olken, F., Shapiro, L.D., Stonebraker, M. and Wood, D.A., “Implementation Techniques for Main Memory Database Systems”, ACM SIGMOD Record, Vol. 14 No.2, 1984, pp 1-8,

14. DeWitt, D., and Gray, J. “ Parallel Database Systems: The Future of High Performance Database Systems”, Communications of the ACM, Vol. 35, No. 6, June 1992, pp 85-98

15. Franaszek, P., and Robinson, J.T., “Limitations of Concurrency in Transaction Processing”, ACM TODS, Vol. 10, No.1, March 1985, pp 1 - 28

16. Franaszek,P., Robinson,J.T.and Thomasian,A., “Access Invariance and Its Use in High-Contention Environments”, Proceedings of the 6th International Data Engineering Conference, Los Angeles, Feb 1990, pp 47 - 55

17. Franaszek,P., Robinson,J.T.and Thomasian,A.,“Concurrency Control for High Contention Environments”, ACM TODS, Vol.17, No.2, June 1992, pp 304 - 345

18. Franklin, M.J., Carey, M.J., and Livny, M., “Transactional Client-Server Cache Consistency: Alternatives and Performance”, ACM TODS, Vol.22, No., September 1997,pp 315-363

19. Garcia-Molina, H. and K. Salem, K. , "Main Memory Database Systems", IEEE Transactions on Knowledge and Data Engineering, Vol.4, No.6, December 1992, pp 509-516

20. Gawlick, D., "Processing Hot Spots in High Performance Systems", IEEE Database Engineering, 1985, pp 249 -251

21. Gawlick, D., and Kinkade, D., " Varieties of Concurrency control in IMS/VS FastPath", IEEE Database Engineering, Vol. 8, No. 2, 1985, pp 3-10

22. Gray, J., and Reuter, A. Transaction Processing: Concepts and Techniques, Morgan-Kaufmann, San Mateo, California, 1993.

23. Gruenwald, L., Chen,Y.W., and Huang, J., "Effects of Update Techniques on Main Memory Database System Performance", IEEE Transactions on Knowledge and Data Engineering, Vol. 10, No. 5, September/October 1998

24. Takahiro , Kaname , Masahiko and Shojiro, "Database Migration: A New Architecture for Transaction Processing in Broadband Networks", IEEE Transactions on Knowledge and Data Engineering, Vol. 10, No. 5, September/October 1998, pp 839-854

25. Harder, T. "Observations on Optimistic Concurrency Control Schemes", Information Systems, Vol.9, No. 2, 1984, pp 111-120

26. Haritsa,J.R., and Seshadri, S., "Real -Time Concurrency Control", SIGMOD Record, Vol.25 No. 1, March 1996, pp 13 - 17

27. Hasse, C., and Welkum, G., " A Performance Evaluation of Multi-Level Transaction Management", Proceedings of the 17th International Conference on Very Large Data Bases, Barcelona, September 1991, pp 55 -66

28. Heiss, H., and Wagner, R., " Adaptive Load Control in Transaction Processing Systems", Proceedings of the 17th International Conference on Very Large Data Bases, Barcelona, September 1991, pp 47 - 54

29. Joshi, A.M., " Adaptive Locking Strategies in a Multi-node Data Sharing Environment", Proceedings of the 17th International Conference on Very Large Data Bases, Barcelona, September 1991, pp 181 -191

30. Kaspi, S., "Optimizing Transaction Throughput in Databases Via an Intelligent Scheduler", *Proceedings of the 1997 IEEE International Conference on Intelligent Processing Systems*, Beijing, October, 1997, pp.1337 – 1341

31. Kaspi, S., "The Use of Contention-Based Scheduling For Improving The Throughput of Locking Systems", forthcoming in, *Proceedings of ADBIS 2001*, Vilnius, Lithuania, September 2001

32. Kornacker, M., Mohan, C. and Hellerstein, J.M., "Concurrency and recovery in generalized search trees", Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data, Vol. 26, No 2, Tucson Arizona, June 1997, pp 62-72

33. Kung, H.T. and Robinson, J.T., "On Optimistic Methods for Concurrency Control", ACM Transactions on Database Systems, Vol.2, No.4, June 1981, pp 213 - 226

34. Leung, C.H.C., "Parallel Paradigms for Query Evaluation and Processing", Proceedings of the Australasian Workshop on Parallel and Real-Time Systems PART'94, Melbourne, 1994, pp 1 -10

35. Leung, C.H.C., and Ghogomu, H.T., " A High-Performance Database Architechture", Proceedings of the 7th ACM International Conference on Supercomputing, Tokyo, 1993, pp 377-386

36. Leung, C.H.C. and S. Kaspi, "A Flexible Paradigm for Semantic Integration in Cooperative Heterogeneous Databases" *Proceedings of FGCS '94, ICOT*, Tokyo, December 1994

37. Menasce, D.A., Nakanishi, T., "Optimistic Versus Pessimistic Concurrency Control Mechanisms in Database Management Systems", Information Systems, Vol. 7 No. 1, 1982, pp 13 - 27

38. Molesky, L.D. and Ramairtham, K., " Recovery Protocols for Shared Memory Database Systems", Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, Vol. 24 No. 2, San Jose California, May 1995, pp 11-22

39. Ng,W.T. and Peter M. Chen, P.M.," Integrating Reliable Memory in Databases", Proceedings of 23rd International Conference on Very Large Data Bases, , Athens, Greece, August 25-29, 1997, pp 76-85

40. O'neil, P. " The Escrow Transactional Mechanism", ACM Transactions on Database Systems, Vol. 11, No. 4, December, 1986, pp 405-430

41. PolyHydera, "Product Overview, PolyHydera Real Time Embedded Database" retrieved February 16 2001, <http:// www.polyhedra.com/ product.htm >

42. Ram, P., Do, L. and Drew, P, "Distributed Transactions in Practice", Sigmod Record, Vol. 28, No. 3, September 1999, pp 49-55

43. Reuter,A., "Concurrency on High-Traffic Data Elements", Proceedings of the 1st ACM PODS, Los Angeles, California, March 1982, pp 83 – 92

44. Ryu, I.K., and Thomasian, A., "Analysis of performance with Dynamic Locking", Journal of the ACM, Vol. 37, No.3, September 1990, pp 491 - 523

45. Shasha, D., "Efficient and Correct Execution of Parallel Programs that Share Memory", ACM Transactions on Programming Languages and Systems, Vol. 10, No. 2, April 1988, pp 282-312

46. Shasha, D., Llirbat, F., Simon, E. and Valduriez, P., “Transaction Chopping: Algorithms and Performance Studies”, ACM Transactions on Database Systems, Vol. 20, No. 3, September, 1995, pp 325-363

47. Thomasian, A., “Performance Limits of Two Phase Locking “, 7th IEEE International Conference on Data Engineering, Kobe, Japan, 1991, pp 426-435.

48. Thomasian, A., and Ryu, K., “Performance Analysis of Two-phase Locking”, IEEE Transactions on Software Engineering, Vol. 17 No. 5, May 1991, pp 386-402

49. Thomasian, A., “Checkpointing for Optimistic Concurrency Control Methods”, IEEE Transactions on Software Engineering, Vol. 17 No. 5, May 1995, pp 386-402

50. Thomasian, A., “ A performance Comparison of Locking Methods with Limited Wait Depth”, IEEE Transactions on Knowledge and Data Engineering, Vol. 9 No. 3, May/June 1997, pp 421-434

51. Thomasian, A., “Distributed Optimistic Concurrency Control Methods For High-Performance Transaction Processing”, IEEE Transactions on Knowledge and Data Engineering, Vol. 10 No. 1, May, 1998, pp 173-187

52. TimesTen Performance Software, “A Performance Brief, 2001”, retrieved April 26 2001 <http://www.timesten.com/library/index.html#whitepapers>

53. TimesTen Performance Software, “TimesTen White Paper, 2001”, retrieved April 26 2001 <http://www.timesten.com/library/index.html# whitepapers>

54. Ulusoy, , O., and Buchmann, A., “ Exploiting Main-Memory DBMS Features to Improve Real-Time Concurrency Control Protocols”, SIGMOD Record, Vol.25 No. 1, March 1996, pp 13 - 17

55. Valduriez, P., “Parallel Database Systems: Open Problems and New Issues”, Distributed and Parallel Databases, Vol. 1, No. 2, April 1993, pp 137-165

56. WEB PROFORUM TUTORIALS, “Commercial In-Memory Databases: A New Alternative”, retrieved April 26 2001 <http:// www. iec. org/tutorials/in_ memory/topic04.html >

57. Zaharioudakis, M. and Carey, M.J., “Highly concurrent cache consistency for indices in client-server database systems”, Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data, Vol. 26 No. 2, Tucson, Arizona, June 1997,pp50-61

# *Appendix A*

## *Simulation Implementation*

All the evaluations and analyses in this thesis were based on data supplied by the running of simulation programs constructed entirely by the author.

The hardware used was an IBM-compatible machine with 256Mb of RAM operating at 400 MHz initially and 850 MHz after an upgrade. The operating systems used were Windows NT4 and Windows 2000. All the programs were written using Borland C++ (Versions 4 and 5) with output going out to Microsoft Exel files. Figure A.1 shows a UML class diagram of the classes used in our programs and the associations between them. It should be noted that the same basic structure is used for all our programs regardless of the concurrency control used. Thus the classes Lock and Locktable exist even where optimistic concurrency is used. Thus under 2PL the Locktable class uses lock, unlock and wakeup methods whereas if optimistic concurrency is used, then the Locktable class calls

the lock, kill and die methods. The lock method under optimistic concurrency is non-blocking and is merely used to indicate over which objects a conflict occurred and which transactions to kill (or which should die).

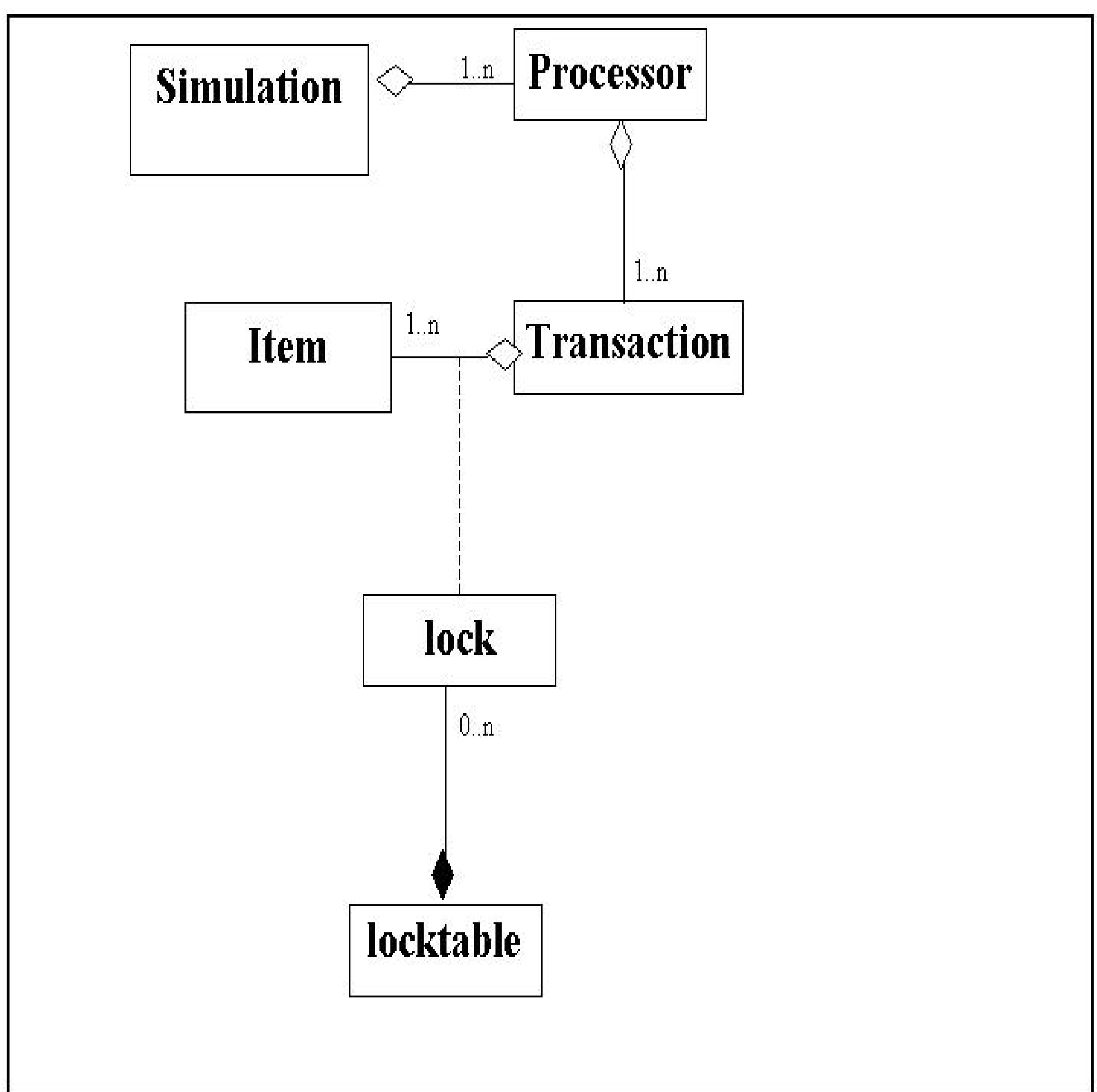



**Figure A.1. A UML class diagram of our simulation programs**

Parameters which are variable include the length of the simulation, the number of times a particular simulation was run, the number of processors, the speed of the

processors, disk access speed, the arrival rate of transactions, the mix of transactions arriving at the system and their composition, the number and size of each database.

Thus the programs could be used to test for a very large number of system configurations. Of the very large number of configurations possible, only a relatively small number were tested and of these, an even smaller number was actually used in the thesis. The reasons for these restrictions are as follow –

1. To have conducted all the tests possible with these programs would have taken several lifetimes and thus a limitation had to be set because of time constraints.
2. Even with a relatively few configurations the thesis is quite sizeable. To have reported all of the systems actually tested would have made the thesis mammoth without making any additional significant points while risking being excessively repetitive.

The reasons particular subsystems or parameters were finally chosen, if not explicitly detailed in the body of my thesis, is largely because they had been used in tests reported in reputable journals and proceedings. For example, the main transaction and database subsystems used in this thesis are essentially the same as used in [17] where it was thought appropriate for representing a high contention environment. This Database subsystem and many of the transaction parameters (number of instructions for initialization, access, committal) was also used in [51].

As indicated above, the results of each run were written directly to an EXEL file by the program itself. A screen print of part of one such file is shown in Figure A..2 below.

Microsoft Excel - 2pl 1 bil mips all mem with deadlock breaking.xls

File Edit View Insert Format Tools Data Window Help

Arial 10

A1 = 5

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| 25 | 5 | 11409 | 2394 | 2278 | 4069 | 2668 |
| 26 | 5 | 11263 | 2342 | 2213 | 3924 | 2784 |
| 27 | 5 | 11344 | 2421 | 2256 | 3956 | 2711 |
| 28 | 5 | 11458 | 2408 | 2316 | 4078 | 2656 |
| 29 | 5 | 11336 | 2442 | 2264 | 3872 | 2758 |
| 30 | 5 | 11364 | 2416 | 2243 | 3954 | 2751 |
| 31 | 5 | 11304 | 2402 | 2186 | 3955 | 2761 |
| 32 | 5 | 11409 | 2400 | 2277 | 4039 | 2693 |
| 33 | 5 | 11308 | 2324 | 2283 | 3941 | 2760 |
| 34 | 5 | 11407 | 2401 | 2286 | 4031 | 2689 |
| 35 | 5 | 11357 | 2421 | 2274 | 3889 | 2773 |
| 36 | 5 | 11374 | 2359 | 2338 | 3986 | 2691 |
| 37 | 5 | 11368 | 2420 | 2213 | 4026 | 2709 |
| 38 | 5 | 11413 | 2446 | 2298 | 4004 | 2665 |
| 39 | 5 | 11453 | 2458 | 2333 | 3951 | 2711 |
| 40 | 5 | 11385 | 2407 | 2356 | 3873 | 2749 |
| 41 | 5 | 11286 | 2349 | 2204 | 3965 | 2768 |
| 42 | | 11365.66 | 2400.024 | 2270.512 | 3973.927 | 2721.195 |
| 43 | | | | | | |
| 44 | 10 | 21661 | 4708 | 4268 | 7515 | 5170 |
| 45 | 10 | 21435 | 4495 | 4292 | 7426 | 5222 |
| 46 | 10 | 21540 | 4571 | 4329 | 7422 | 5218 |
| 47 | 10 | 21552 | 4544 | 4357 | 7502 | 5149 |
| 48 | 10 | 21423 | 4484 | 4271 | 7456 | 5212 |

2pl 1 bil mips all mem with d

After the program is complete, the EXEL file is opened and the mean result for each concurrency is calculated by using EXEL's inbuilt

**Figure A..2. in-memory 2000 MIPS.**

For comparing different systems, the mean results for each system to be compared was copied to a separate EXEL file and the results were then charted using EXEl's inbuilt features.

# *Appendix B*

## *Thrashing Control Strategies*

To determine the relative costs and benefits of each of the thrashing control strategies outlined in chapter 4, we test each of these strategies on the 96 by 200 MIPS per processor in-memory system at a concurrency of 100. The reason for the choice of this hardware configuration is that it completes a very large number of cycles at a reasonably high concurrency thus giving us a large number instances where thrashing may occur and enabling us to have a large sample of the relative costs and benefits of each strategy. Two thresholds are tested – 0.378 and 0.756 - the latter is arbitrarily chosen by doubling the 0.378 threshold. We also implement a mechanism to test for a circular deadlock to a depth of 3. If a deadlock is not detected at this level, then the thrashing control mechanism reverts to a simple proportion of transactions blocked

strategy. The implementation of a limited depth check is used to indicate whether such a mechanism significantly reduces the number of restarted transactions. The result of this series of tests is shown in table B.1 below.

All the systems tested are successful in preventing thrashing. However, apart from their prevention of thrashing, they display a variety of behaviors. As indicated by table B.1, the implementation of a partial depth check does not seem to affect either the number of restarts or throughput. Since as indicated in chapter 4, implementing a very deep search would be excessively costly and a shallow search does not yield meaningful results, this strategy does not seem viable. The results also indicate that staggering restarts by time stamping transactions tends to reduce the number of restarts. However, this strategy also tends to lower throughput. Similarly, increasing the allowed proportion of blocked transactions reduces the number of restarts but also tends to lower throughput.

Because the cost of restarts is higher in disk- based systems than it is in in-memory systems, we also test each of the above strategies (but discarding any depth checking) in our disk-based systems. Here, despite the disparity in system configurations, the relative results are consistent. Figures B.1 to A.5 show the results for each of our disk-based systems at a concurrency of 100. As the results show, for all hardware configurations, all of the thrashing control strategies yield better results on equivalent hardware than no thrashing control. In all systems the lower threshold yields a better result than the higher threshold indicating that the cost of allowing blocked transactions to reside in memory for

an extended period exceeds the cost of unnecessarily restarting transactions that would clear if the system were allowed to continue without interference.

| Proportion of Blocked Transactions Allowed | Depth Checking | Restart Strategy | Restart Frequency | Throughput |
|---|---|---|---|---|
| 0.378 | no | staggered restart | 9900 | 15053 |
| 0.378 | no | immediate | 221983 | 22547 |
| 0.378 | yes | staggered restart | 9822 | 15167 |
| 0.756 | no | staggered restart | 788 | 2022 |
| 0.756 | no | immediate | 15836 | 17420 |
| 0.756 | yes | staggered restart | 675 | 1900 |

**Table B.1. Cost benefit of diverse thrashing**

However, unlike the in-memory system, in the disk-based systems a staggered restart of transactions (by using the time stamp mechanism outlined above) actually increases throughput with each of the thresholds used. This is because as indicated above, a staggered re-start changes the sequence in which locks are acquired thus reducing the probability that the same wait chain will recur and thus reducing the probability that transactions will have to be continually restarted. In disk-based systems, the benefit of this reduction in the number of restarts exceeds the cost of temporarily keeping transactions inactive.

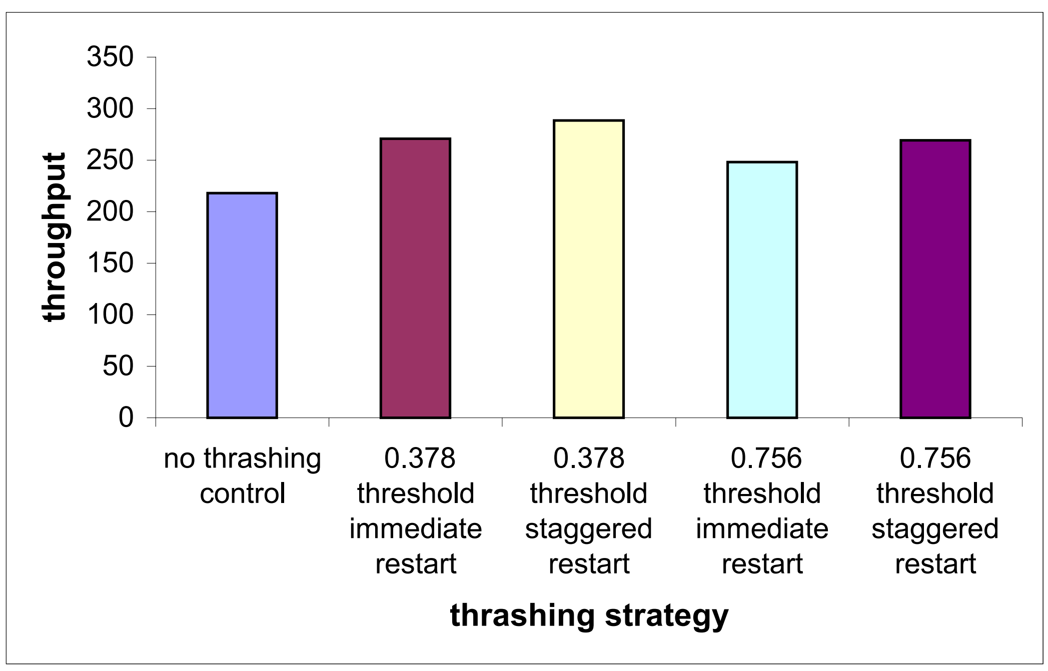



**Figure B.1. 1248 processors by 4 MIPS disk based system**

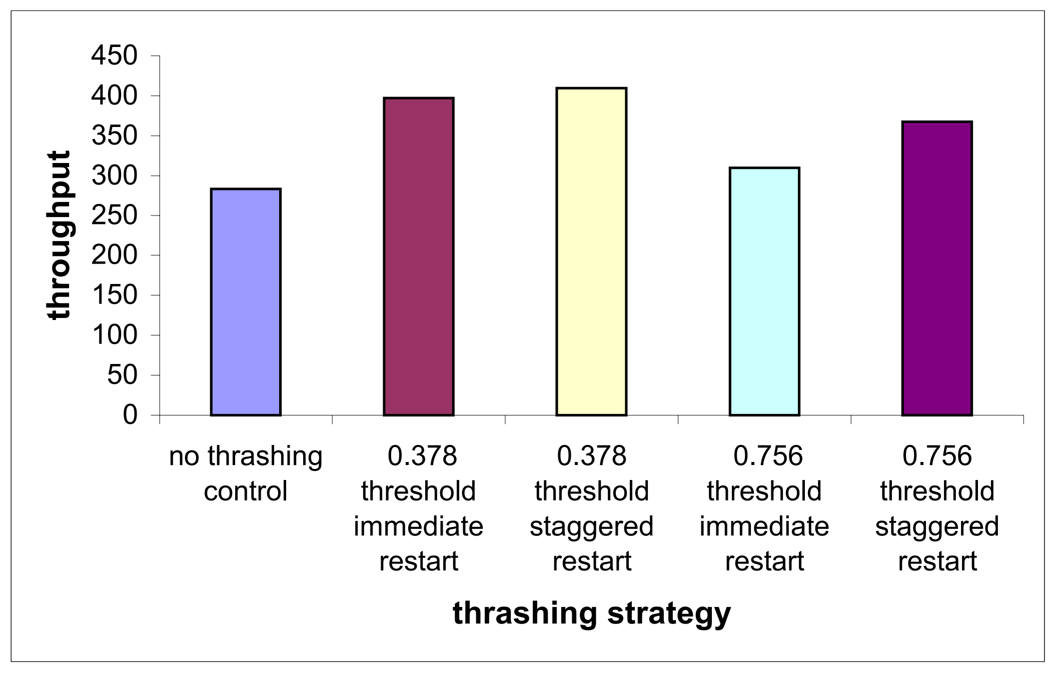



**Figure B.2. 96 processors by 100 MIPS disk based system**

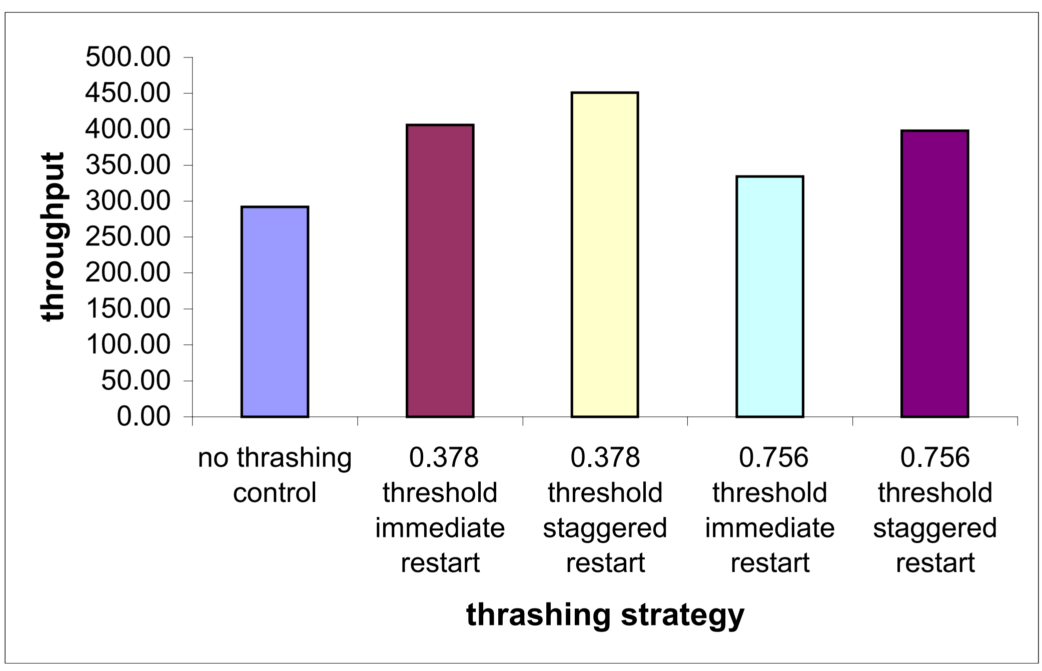



**Figure B.3. 96 processors by 200 MIPS disk based system**

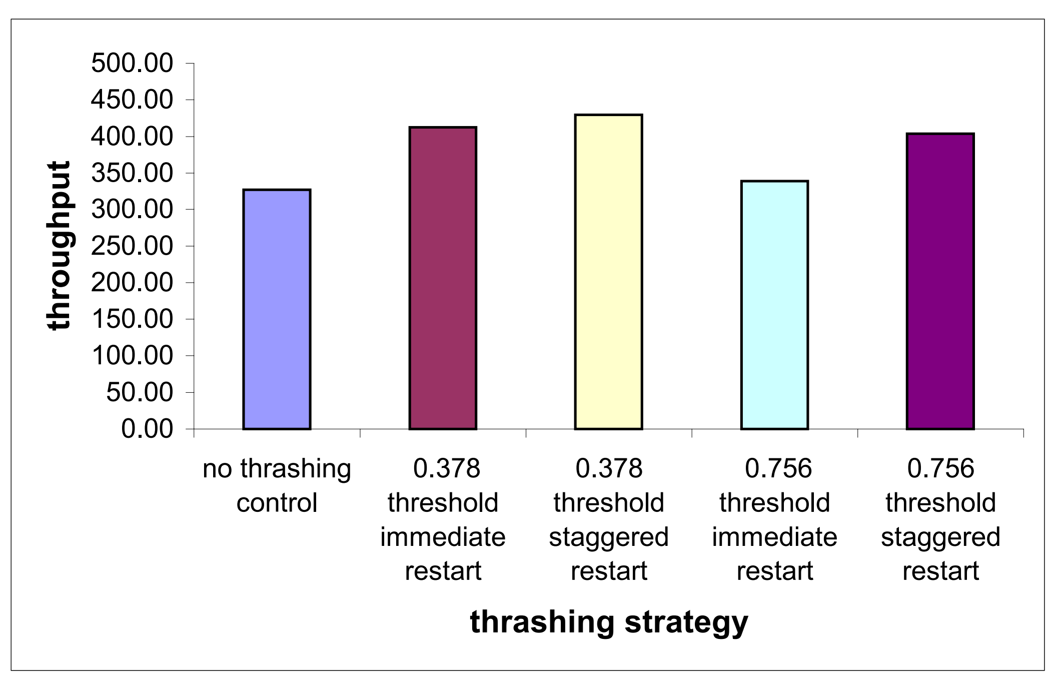



**Figure B.4. 20 processors by 1000 MIPS disk based system**

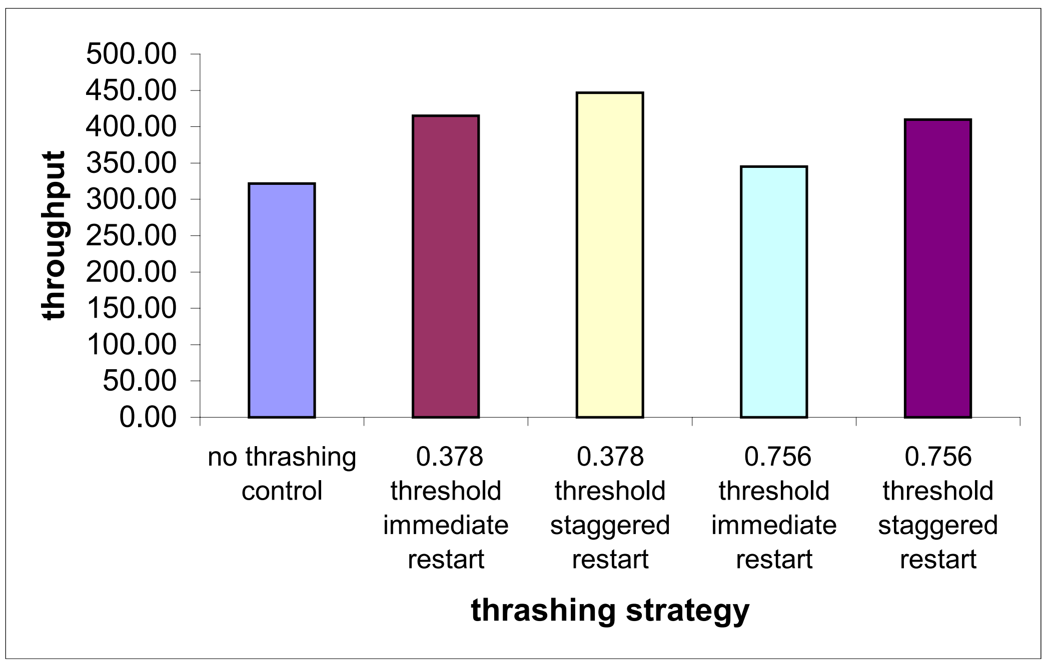



**Figure B.5. processors by 2000 MIPS disk based system**

These results indicate that the two most successful disk-based thrashing control strategies are a 0.356 threshold with staggered restarts and a 0.356 threshold without staggered restarts.