Sequential PLEX, and its Potential for Parallel Execution

Johan Lindhult[1] and Björn Lisper[1]

Dept. of Computer Science and Electronics, Mälardalen University
P.O. Box 883, SE-721 23 Västerås, SWEDEN,
{Johan.Lindhult, Bjorn.Lisper}@mdh.se

Abstract. Some computer systems have been designed under the assumption that activities in the system are executed non-preemptively. Exclusive access to any shared data in such a system is automatically guaranteed as long as the system is executed on a single-processor architecture. However, if the activities are executed on a multiprocessor, exclusive access to the data must be guaranteed when memory conflicts are possible. An analysis of the potential memory conflicts can be used to estimate the possibility for parallel execution.
Central parts of the AXE telephone exchange system from Ericsson is programmed in the language PLEX. The current software is executed on a single-processor architecture, and assumes non-preemptive execution.
In this paper, we investigate some existing PLEX code with respect to the number of possible shared-memory conflicts that could arise if the existing code, without modifications, would be executed on a parallel architecture. Our initial results are promising; only by examining the data that actually can be shared, we manage to reduce the number of conflicts from the assumed 100% to figures between 25-75% for the observed programs. Simple optimizations decrease the numbers even further.

1 Introduction

Over the years, many computer systems have been designed under the (sometimes implicit) assumption that activities in the system are executed non-preemptively. Examples of such systems are small embedded systems that are quite static to their nature, or priority-based systems where activities on the highest priority are assumed to be non-interruptible. Non-preemptive execution gives exclusive access to shared data, which guarantees that the consistency of such data is maintained.

However, new machines will increasingly be parallel [14]. On a parallel architecture, activities executed on different processors may access and update the same data concurrently, and non-preemptive execution does not protect the shared data any longer. On the other hand, the very idea of parallel architectures is to increase performance by parallel execution. The question is: *how utilize the power of a parallel processor for a system designed for non-preemptive execution?*

Our subject of study is the language PLEX, used to program the AXE telephone exchange system from Ericsson. The AXE system, and the PLEX language, have roots that go back to the late 1970's. The language is event-based in the sense that only events, encoded as signals, can trigger code execution. Signals trigger independent activities (denoted jobs), which may access shared data stored in different shared data areas. Jobs are executed in a priority-based, non-interruptable (at the same priority level), fashion on a single-processor architecture, and the language lacks constructs for synchronization. Due to the atomic nature of jobs (further discussed in the following section), they can be seen as a kind of transactions. Thus, when executing jobs in parallel, one will face problems that are similar to maintaining the ACID[1] properties when multiple transactions, in a parallel database, are allowed to execute concurrently.

Our primary motivation for this study is the fact that multicore architectures will become a de-facto standard in a near future. There are millions of lines of legacy event-based code in industry. Rewriting this code into explicitly parallel code would be extremely expensive. Thus, there is a need to investigate methods to migrate such code to parallel architectures with a minimum of rewriting.

In order to estimate the possibility for parallel execution of the existing PLEX code, we have performed a static program analysis of the potential memory conflicts that actually can arise. The number of conflicts are measured as the relative numbers of different jobs that can interfere with each other through the shared data areas. Our initial results show that compared to a straightforward parallel implementation, where each shared data area is protected by a lock, we can by a simple static analysis of the data usage reduce the potential conflicts between jobs to be in the range 25-75% for the observed programs. Furthermore, we also show that simple, static, optimizations are likely to reduce the number of potential conflicts even further, thereby reducing the amount of manual work that probably still needs to be performed in order to adapt the code for parallel processing.

The rest of this paper is organized as follows: the main characteristics of the language PLEX is covered in Section 2. Section 3 contains a brief summary of the assumed parallel architecture as well as a closer examination of the shared data. Section 4 contains a description of our static approach, whereas the examination of the code is covered in Section 5. Related work in Section 6, before the paper is wrapped up with conclusions and further research in Section 7.

[1] Atomicity, Consistency, Isolation and Durability

2 Programming Language for EXchanges

PLEX is a pseudo-parallel and event-driven real-time language developed by Ericsson. The language is used in the AXE telephone exchange system, and it was developed in conjunction with the first AXE versions in the 1970's. Apart from an asynchronous communication paradigm, PLEX is an imperative language, with assignments, conditionals, goto's, and a restricted iteration construct. It lacks some common statements from other programming languages such as while loops, negative numeric values and real numbers.

A PLEX program is structured in *blocks*. Each block contains several, independent sub-programs together with block-wise scooped data, see Fig. 1. As we will see in the following section, this data (variables) can be classified into different categories depending on whether or not the value of a variable 'survives' termination of the software. Blocks can be thought of as objects, and the subprograms are somewhat reminiscent of methods. However, there is no class system in PLEX, and it is more appropriate to view a block as a kind of software component whose interface is provided by the entry points to its sub-programs. Data within blocks is strictly hidden, and there is no other way to access it than through the sub-programs.

The sub-programs in a block can be executed in any order: execution of a sub-program is triggered by a certain kind of event called *signal* arriving to the block. Signals may be *external*: arriving from the outside or *internal*: arriving from other sub-programs, possibly executing in other blocks. The execution of one, or several, sub-programs constitutes a *job*; a job begins with a signal receiving statement, and is terminated by the execution of an `EXIT` statement. Due to the 'atomic' execution of a job, i.e., once a job is started it will run to completion, we may also view them as a kind of transactions.

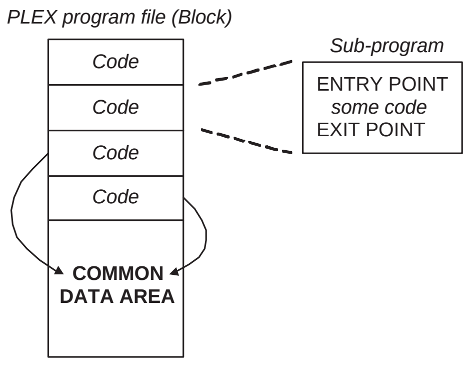

Fig. 1. *A PLEX program file (a* block*) consists of several sub-programs.*

Since sub-programs can be independently triggered, it is accurate to consider jobs as "parallel". However, the jobs are not executed truly in parallel: rather, when spawned, they are buffered (queued), and non-preemptively executed in FIFO order, see Figs. 2 (b) and 3 (a). Because of the sequential FIFO order imposed, we term the language as "pseudo-parallel" since externally triggered jobs could be processed in any order (due to the order of the external signals). We also note that different types of jobs are executed on different levels of priority, and that jobs of the same priority are executed non-preemptively. User jobs (or call processing jobs), i.e., handling of telephone calls, are always executed with high priority, whereas administrative jobs (e.g„ charging) always are executed with low priority (and never when there are user jobs to execute).

A key aspect, which distinguishes PLEX from an "ordinary" imperative language, is the asynchronous communication paradigm: jobs communicate and control other jobs through signals. Signals are classified through combinations of different properties, where the main distinction is between *direct* and *buffered* signals, see Fig. 2. The difference is that a direct signal continues an ongoing job, whereas a buffered signal spawns off a new job. We refer to [5] for a more thorough description on signals as well as the asynchronous communication paradigm.

Finally, we denote the set of jobs originating from the same external signal a *job-tree*. See also Fig. 3 (b), where the corresponding job-tree for the execution in Fig. 3 (a) is shown.

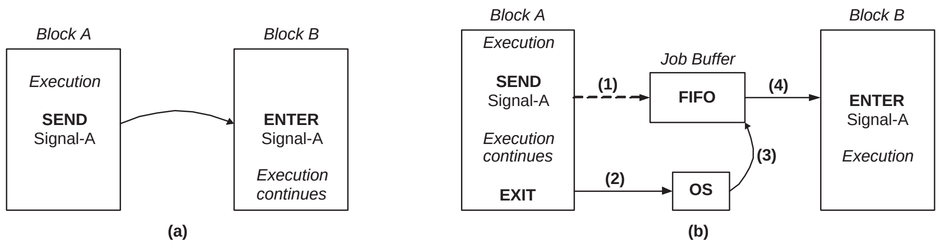

Fig. 2. (a): a direct signal, "similar" to a jump. (b): buffered signals: a buffered signal is sent from Block A which is inserted at the end of the job buffer (1). When the job in Block A terminates, the control is transferred to the OS (2), which fetches a new signal from the buffer (3). This signal then triggers the execution in Block B (4).

3 A Parallel Architecture, and the Shared Data

We consider a hypothetical, parallel architecture which is a conventional shared-memory architecture. It is equipped with a run-time system, which

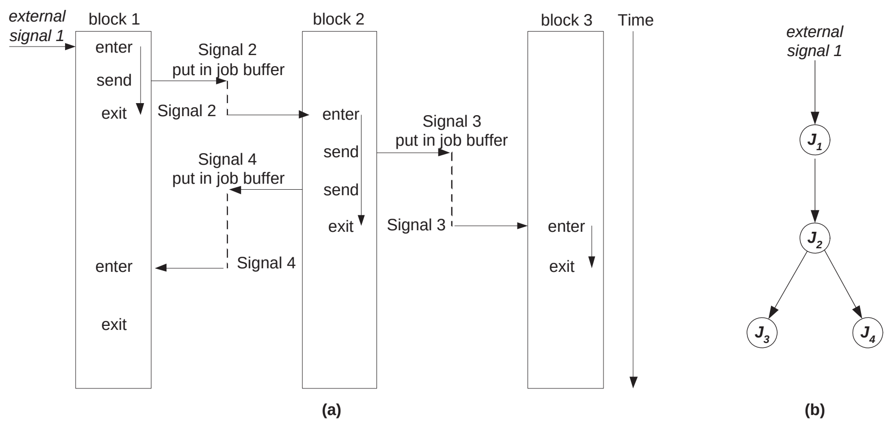

Fig. 3. The "pseudo-parallel" execution model of PLEX (a), and a corresponding job-tree (b).

is designed to execute PLEX programs as they are. The execution of jobs is done by a static number of threads, which may or may not equal the number of processors. Each thread has its own local state. Threads are executing user jobs in parallel. However, to make the system functionally equivalent with the original, sequential system, jobs from the same job-tree execute in the same sequential order as in the single-processor case, and a block is locked as soon as a job is executing in it in order to protect its data from being concurrently accessed. We assume that the run-time system has a method to resolve deadlocks at runtime.

Locking blocks will guarantee consistency of data, since data in a block can never be accessed by a job executing outside that block. However, it may be overly conservative, since two parallel jobs accessing the same block may well never touch the same data. Our analysis of the potential memory conflicts aims at allowing a more loose locking scheme, where a block need not be locked if we know for sure that the jobs executing in it cannot have any memory conflicts.

We only consider parallel execution of user-level jobs in this study. It is assumed that jobs executing on other levels are handled by some other means.

Since the data in a block is shared between all its sub-programs (as shown in the previous section), it might seem as **all** variables may be potentially shared. However, as we indicated in the previous section, the variables belong to different categories: basically, the variables can be divided into the following two main categories; *stored (DS)* or *temporary*.

– The value of a temporary variable exists only in the internal processor registers, and only while its corresponding software is being executed.

Variables are by default temporary, and thus cannot be shared between different jobs.
- DS variables are persistent: they are loaded into a processor register from the memory when needed, and then written back to the memory. These variables can be further divided into[2]:
  1. *Files*
  2. *Common variables*

Common variables are “scalar” variables, whereas files essentially are arrays of *records* (similar to “structs” in C). Elements of records are called *individual variables*. A special kind of variable, called *pointer*, holds integer indices identifying elements of files. Fig. 4 shows an example file with n records and a pointer, whereas Table 1 tries to relate the above PLEX concepts to its closest counterparts in C.

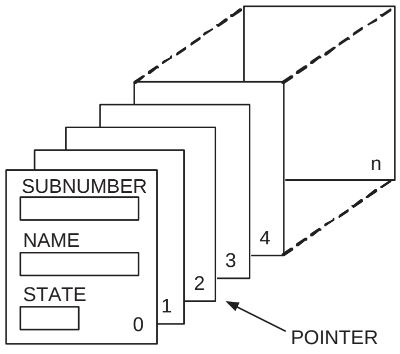

Fig. 4. *An example file with n records and a pointer with the current value 2.*

PLEX	**C**
record	struct
file	array of structs
pointer	array index
individual variable	struct member
common variable	global variable

Table 1. *Some PLEX concepts, and their ”counterparts” in C.*

[2] See also [8] where this distinction is discussed more thoroughly.

4 Analysis of Conflicts

We say that two signals in the same block are in *conflict* if they might access the same variable in such a way that the consistency of data is threatened if the code triggered by the signals is executed concurrently. This is the case if both signals might access the variable and least one may write to it. If two signals are *not* in conflict, they may safely be executed concurrently with no protection at all. A run-time system may use this information to lock a block selectively only for signals that are in conflict. This improves on the assumed parallel architecture in Section 3, which locks a block as soon as one of its signals is executed.

To determine whether or not two signals might be in conflict with each other, the usage of each variable in each signal is classified in the following way:

$\perp$**-** The variable is **never** used by the signal in question.
R **-** Read Only, i.e., the only way the signal is accessing the variable is in read operations.
W **-** If the signal accesses the variable, the first access will **always** be a write operation.
$\top$**-** It is not possible to (statically) classify the variable according to the previous cases, i.e., the usage of the variable might be input dependent, or there might be different paths through the code that use the variable in different ways. It might also be the case that the signal performs a read operation as a first access to the variable.

$$\begin{tikzcd}
 & \top \arrow[dl, no head] \arrow[dr, no head] & \\
R \arrow[dr, no head] & & W \arrow[dl, no head] \\
 & \perp &
\end{tikzcd}$$

Fig. 5. *The hierarchical ordering of variable usage.*

Based on our knowledge on how the variables are used, we can order them in a hierarchical way as in Fig. 5, where we go from absolute knowledge ($\perp$- never used) to actually no knowledge at all ($\top$- can't be determined). We also make the following observations:

– A variable that is never used ($\perp$) can never cause the signal to be in conflict with other signals.

- The value of a Read Only variable is only used (read from), and similar to $\perp$ does not cause the signal to be in conflict with other signals unless some other signal writes to the variable.
- For a variable classified as W, we notice that if every signal that accesses the variable always performs a write as a first possible access, it will be safe to perform the following transformation; let each signal work on a local copy of the variable. This does not change the semantics of the program since no signal will ever use a value written by another signal, regardless of whether or not this transformation is performed.
- Since an unambiguous use of a variable classified as $\top$ can not be determined, we must **always** assume a potential conflict between signals that use this variable.

A conflict matrix for each block would then be straight forward to deduce based on the classification of variables. We give a small example; consider three signals $Sig_{1,\ 2,\ 3}$, and three variables $Var_{1,\ 2,\ 3}$. Table 2 shows how the signals use the variables, as well as the corresponding conflict matrix. The conflict matrix indicates potential conflicts between Sig_1 and Sig_2, and between Sig_1 and Sig_3. Sig_2 and Sig_3 can however execute concurrently.

	Var_1	Var_2	Var_3
Sig_1	R	W	$\perp$
Sig_2	R	R	R
Sig_3	$\perp$	R	R

	Sig_1	Sig_2	Sig_3
Sig_1		X	X
Sig_2	X		
Sig_3	X		

Table 2. *Variable usage in three example signals, together with a corresponding conflict matrix.*

Once the conflict matrix has been constructed, it could be used by the run-time system to perform a table look-up before allowing a signal to start executing.

5 Examining the Code

Our studies are performed on existing PLEX code, and a total of four blocks have been examined. Common for these blocks is that their fraction of the execution time is high compared with other blocks. Each examined block contains a number of signals that are executed more frequently than other signals in each respective block. We call these "HF-signals" (High Frequency Signals). For every *common variable* that are read from, or written to, by such a HF signal, we have examined the usage of this variable in every other signal in that block in order to find out which signals that may possibly be in conflict with these HF-signals. Table 3 summarizes the characteristics of each examined block: type of block, fraction of execution time, as well as the number examined signals, and variables.

The code has been inspected manually, and the reason for not trying to automate the process was that we believed that manual inspection also would increase our general knowledge on how the language is used, in "reality", i.e., it would be *"possible to "see" the semantics of the program"* [8].

Block	**Type**	**Execution time (%)**	**HF**	**Examined signals**	**Examined variables**
CHVIEW	*Middleware*	6.70 %	6	45 of 92	20
LAD	*OS*	0.64 %	2	8 of 28	77
MFM	*OS*	3.40 %	3	26 of 75	51
MSCCO	*Application*	1.76 %	2	59 of 75	14

Table 3. *The examined blocks.*

As said above, only the common variables are considered in this study. Thus, the presented figures will in general underestimate the actual number of conflicts since we have omitted conflicts caused by simultaneous access to the same file. However, we believe that conflicts through files tend to be rare. A conflict takes place through a file only when the same individual variable, in the same record, is accessed simultaneously. For a file of size n the probability of two accesses going to the same record is $1/n$, if the accesses are random, independent, and equally distributed. Files in PLEX are usually used to hold subscriber data or similar. The index of an access thus usually depends on externally supplied data, like a subscriber number, which should be quite random under normal circumstances.

The assumed parallel architecture in Section 3 corresponds to a conflict ratio of 100%, since it always locks a block when used. However, when we examined the common variables and excluded simultaneous 'Read-only accesses' the actual figures were found to be between 25-75% as shown in Table 4, column 3. The last column (of Table 4) also shows that the figures can be further reduced.

Some comments about the figures in Table 4:

CHVIEW: As can be seen in Table 4, we do not get any improvements on the number of conflicts in this block when we try to optimize the code according to the discussion in the previous section. However, worth noting is the fact that all remaining variables are either used as counters, or are holding different pointer values needed for file access.

Block	Observed signals	Nr of conflicts	$\backslash w$
CHVIEW	45 of 92	25.89%	**25.80%**
LAD	8 of 28	47.22%	**8.33%**
MFM	26 of 75	64.67%	64.67%
MSCCO	59 of 75	73.50%	73.50%

Table 4. *Summary of the (relative) number of possible conflicts, between the observed signals, with (and without) optimization applied.*

LAD: Here, the W-optimization almost manages to remove all conflicts. The remaining conflicts are caused by variables that are used in the same way as in the CHVIEW block.

MFM and **MSCCO:** A first examination of the variables shows conflict ratios of 64.67%, and 73.50% respectively. Further improvements (by the W-optimization) can not be achieved in any of these blocks. This is due to that several signals share not only one, but several variables. A closer examination of the variables does not improve the result either (as opposite to the blocks CHVIEW and LAD), since many of the variables in this block is used for communication, e.g., ”the current state of the system is X”.

6 Related Work

Due to its event-based execution model, it may seem natural to relate the possibility of parallel execution of existing PLEX programs to other event-based systems, and especially to Rational Rose RT-models since PLEX and Rose have a similar asynchronous communication paradigm with events encoded as signals [13]. However, the few works that we are aware of in the event-based domain, [10,11] and [12], are all concerned with optimizing performance on a single-processor architecture. Since different modeling languages such as UML and Rose are basically used in the OO domain, the lack of literature might be caused by known difficulties to parallelize an OO program (inheritance, late binding, encapsulation and reusability) [7].

As seen in previous sections, 1 and 2, we have related the execution of a job to the execution of a transaction, and we will therefore review relevant works in the field of parallel databases. First of all we note that there are two architectural “extremes”; the *shared-nothing* (SN) and the *shared-memory* (SM) architectures [4,17,15]. The only way two processors communicate in the SN-architecture is by message passing, and hence transactions can not interfere with each other. The SM-architectures resolve the problem with interfering transactions by locking schemes [16]. Due to better scaling and non-interference between transactions the SN-architecture has been considered superior to the SM-architecture. However, the emerge of multi core architectures will most likely force the database community to revisit the SM-architecture [2]. The latter work explores a parallel database implemented on a Cray MTA-2. This architecture provides hardware primitives for locking of single words of memory, and hashes the physical address space to distribute memory references.

The current approach to keep the system consistent is the coarse locking scheme (lock an entire block) which was described in Section 3. The static analysis described in the previous section is able to safely state that some of the potential conflicts never occur, which implies that the current locking scheme is unnecessarily conservative. However, potential conflicts that we can't resolve still need to be handled dynamically. An alternative to the current "mutual exclusion" approach are reactive concurrent data structures: shared data with non-blocking synchronization, with an ability to adapt their algorithmic complexity to contention variation. Examples of such structures include spin-locks [1], reactive diffracting trees [3], and software transactional memory [9]. The known drawbacks with such structures are that the reactive schemes (i.e., the algorithms) rely on either some experimentally tuned thresholds or know probability distribution of inputs. However, as shown by Ha [6] it is possible to implement the algorithms in a "self-tuning" way.

7 Conclusions and Further Research

As stated in the beginning of this paper, the primary goal with this study was to get an opinion on whether or not the existing PLEX code is suitable for parallel processing. So what conclusions can be drawn based on the results in the previous section?

As a starting point, we had to assume the worst case scenario; i.e., that the number of conflicts in the examined programs were close to 100%. However, a simple static analysis of the data usage reduce the potential conflicts between jobs to be in the range 25-75% for the observed programs. Simple static optimizations are, in some cases, able to reduce the figures even further.

Our initial results are an underestimation of the actual number of conflicts since we have omitted conflicts caused by simultaneous access to the same file. We have chosen this approximation as a starting point for our studies since the probability for two jobs to simultaneously access the same part of a file normally is $1/n$, where n is the size of the file. In the continuation of our research, files will be regarded from the other extreme, i.e., we will consider every simultaneous access as a potential conflict. This will provide us with a safe upper bound on the number of potential conflicts.

To maintain consistency in the case the static analysis fails to resolve a conflict, as well as for allowing simultaneous access to a file, a dynamic solution is required. We have so far compared our static analysis with a dynamic approach, where each shared data area is protected by a lock. However, we have seen that such a "mutual exclusion" approach is too conservative since two jobs accessing the same block may never touch the same data. (Another drawback is the risk of deadlocks). As an alternative to the assumed locking scheme we plan to evaluate some of the reactive data structures as implemented by Ha [6].

To summarize, our initial results are encouraging. We have shown that simple static methods are sufficient to resolve many of the potential conflicts, and we believe that the combination of static analysis and lock-free synchronization might be sufficient to migrate the code to a parallel architecture (or at least minimizing the amount of rewriting).

8 Acknowledgements

This work has been supported by Ericsson AB, and Vinnova through the ASTEC competence center. We want to thank Janet Wennersten and Ole Kjøller at Ericsson AB for technical support and discussions regarding PLEX and its implementations. We are also grateful to the anonymous reviewers for valuable feedback on earlier drafts of this paper.

References

1. T. E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.
2. J. Cieslewicz, J. Berry, B. Hendrickson, and K. A. Ross. Realizing parallelism in database operations: insights from a massively multithreaded architecture. In *DaMoN '06: Proceedings of the 2nd international workshop on Data management on new hardware*, page 4, New York, NY, USA, 2006. ACM Press.
3. G. Della-Libera and N. Shavit. Reactive diffracting trees. *Journal of Parallel and Distributed Computing*, 60(7):853–890, 2000.
4. D. J. DeWitt and J. Gray. Parallel database systems: the future of database processing or a passing fad? *SIGMOD Rec.*, 19(4):104–112, 1990.
5. J. Erikson and B. Lindell. The Execution Model of the APZ/PLEX - An Informal Description. Technical report, Mälardalen University, 2002.
6. P. Ha. *Reactive Concurrent Data Structures and Algorithms for Synchronization*. PhD thesis, Chalmers University of Technology, 2006.
7. S. Kumar. Issues in parallelizing object-oriented programs. In *Proceedings of the 1995 ICPP Workshop on Challenges for Parallel Processing*, pages 64–71, 1995.
8. B. Lindell. Analysis of reentrancy and problems of data interference in the parallel execution of a multi processor AXE-APZ system. Master's thesis, Mälardalen University, 2003.
9. V. Marathe, W. Scherer, and M. Scott. Adaptive software transactional memory. In *Distributed Computing, Proceedings Lecture Notes in Computer Science 3724*, pages 354–368. Springer-Verlag Berlin, 2005.
10. A. Marburger and D. Herzberg. E-CARES Research Project: Understanding Complex Legacy Telecommunication Systems. In *Fifth European Conference on Software Maintenance and Reengineering*, pages 139 – 147, 2001.

11. C. Mosler. E-CARES Project: Reengineering of PLEX Systems. *Softwaretechnik-Trends*, 26(2):59–60, 5 2006.
12. M. Rajagopalan, S. K. Debray, M. A. Hiltunen, and R. D. Schlichting. Profile-directed optimization of event-based programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation PLDI '02*, pages 106 – 116, 2002.
13. Rational. *Modeling Language Guide - Rational Rose Realtime*, 2002.
14. H. Sutter and J. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, Sept. 2005.
15. A. S. Talwadker. Survey of performance issues in parallel database systems. *J. Comput. Small Coll.*, 18(6):5–9, 2003.
16. P. Watson and G. Catlow. Architecture of the icl goldrush megaserver. In *BNCOD 13: Proceedings of the 13th British National Conference on Databases*, pages 249–262, London, UK, 1995. Springer-Verlag.
17. M. T. Özsu and P. Valduriez. Distributed and parallel database systems. *ACM Comput. Surv.*, 28(1):125–128, 1996.