Existing PLEX Code, and its Suitability for Parallel Execution - A Case Study

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

April 19, 2008

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 results show that only by examining the data that actually can be shared, we derive a safe upper bound on the number of conflicts to figures between 33-91% for the observed programs (in comparison with the assumed 100%). A more fine-grained analysis, based on certain run-time properties, can decrease the numbers to figures between 0-75%.

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 [SL05]. 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 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 0-76% for the observed programs, 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: Section 2 gives a brief introduction to the language PLEX and some related concepts. Section 3 describes the experimental shared-memory architecture (which in the remaining of this paper will be called the "prototype") that executes the observed PLEX code. Section 4 defines the conflicts we are investigating, whereas Section 5 covers the examination of the code. Related work is covered in Section 6 before the paper is summarized in Section 7.

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

[1] Atomicity, Consistency, Isolation and Durability

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.

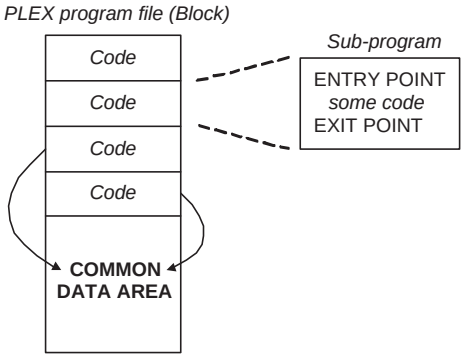

Figure 1: *A PLEX program file (a* block*) consists of several sub-programs.*

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 [EL02] 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.

3 A Parallel Architecture, and the Shared Data

The parallel architecture we consider in this paper is a conventional shared-memory architecture. It is equipped with a run-time system, which is designed to execute PLEX programs as they are. Logically, the execution is done by a static number of threads, which may or may

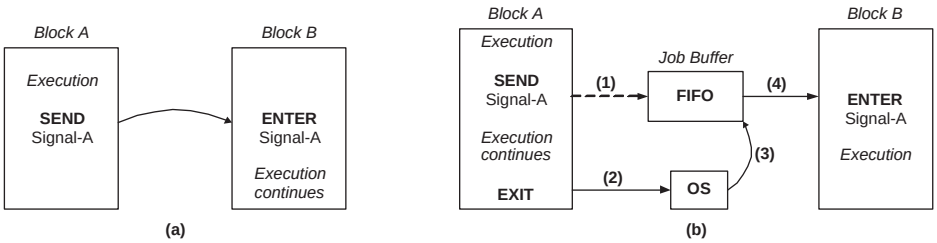

Figure 2: (a): a direct signal, ”similar” to a jump. (b): buffered signals: a buffered signal is sent from Block A which is inserted at the end of the job buffer (1). When the job in Block A terminates, the control is transferred to the OS (2), which fetches a new signal from the buffer (3). This signal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

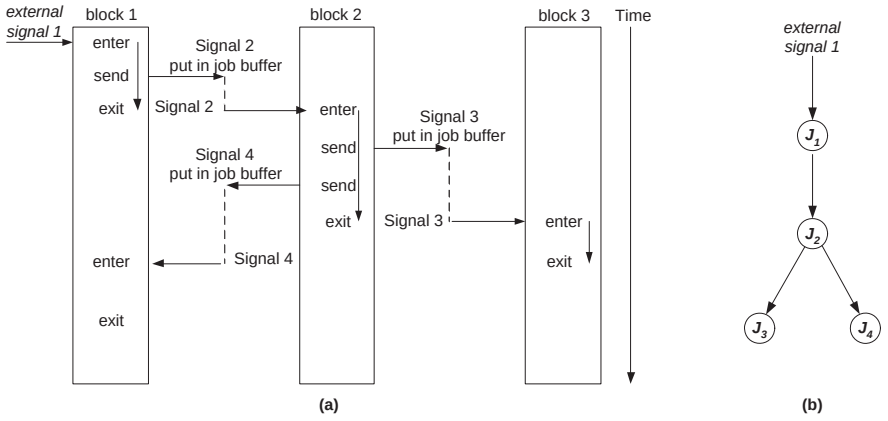

Figure 3: The ”pseudo-parallel” execution model of PLEX (a), and a corresponding job-tree (b).

not equal the number of processors. Each thread has its own local state, and a number of pre-allocated blocks, which are only executed by the thread they have been allocated to. The "remaining" blocks can be executed by any of the threads, and we say that these blocks execute in "parallel mode". See also [Lin08] where the execution paradigm is given a more detailed coverage.

The old (sequential) software is, without modifications, executed on the parallel architecture. The run-time system of this architecture is designed to preserve functional equivalence with the original, sequential system. The approach taken to achieve this equivalence is to (1) let jobs from the same job-tree execute in the same sequential order as in the single-processor case, and (2) lock a block as soon as a job is executing in it in order to protect its data from being concurrently accessed.

Although the use of a locking scheme introduces the risk of deadlocks, we will not consider this further since the run-time system has a mechanism to resolve this.

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. This is because administrative jobs are not allowed to be executed while there are user-level jobs to process.

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 (mostly) "scalar" variables[3], whereas files essentially are arrays of *records* (similar to "structs" in C). Elements of records are called *individual variables*. *Pointers* address the relevant record in a file. The records in a file are numbered, and the value of the pointer is the number of the current record. Fig. 4 shows an example file with its records and a pointer, whereas Table 1 tries to relate the above PLEX concepts to its closest counterpart in C.

Notable is that a pointer "behaves" like a temporary variable in that it will lose its value when the job that uses the pointer terminates. Thus, common variables are used to store the "current value" of a pointer between the execution of different jobs.

[2]See also [Lin03] where this distinction is discussed more thoroughly.
[3]A common variable may also be an array.

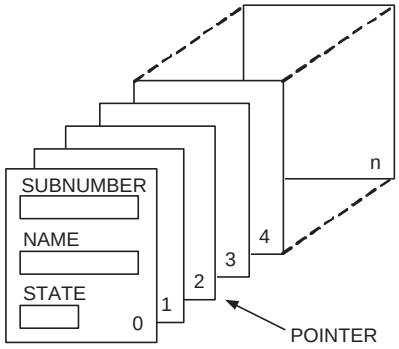

Figure 4: *An example file with n records and a pointer with the current value 2.*

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.

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 at 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 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.

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 always be determined). We also make the following observations:

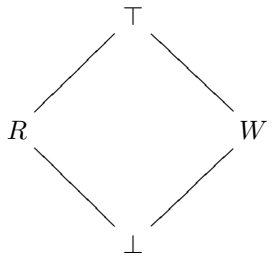

Figure 5: *The hierarchical ordering of variable usage.*

- 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. We denote this transformation as 'w-optimization' in the following sections.
- 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

As we said in the previous section, only a subset of the blocks are executed in "parallel mode" which means that they may be executed by any thread, and consequently are candidates

for parallel execution. Other blocks are not considered in this study. Furthermore, every block might also, besides ”call processing code” (i.e., handling of telephone calls), contain some administrative code (e.g., charging for a call). Since administrative code is not allowed to execute while there is call processing work to be done, the variables that are considered in this study are DS variables accessed by call processing code (and where the block executes in ”parallel mode”).

Our studies are performed on existing PLEX code, executed on the shared-memory architecture described in Section 3. Our figures on execution time for different signals are based on traces from these executions. The software consists of 1045 blocks; 34 of those are executed in ”parallel mode” whereas the remaining blocks are allocated to different threads. 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 *DS 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*” [Lin03].

Block	**Type**	**Execution time (%)**	**HF**	**Examined signals**	**Examined variables**
CHVIEW	*Middleware*	6.70 %	6	70 of 92	26
LAD	*OS*	0.64 %	2	8 of 28	88
MFM	*OS*	3.40 %	3	26 of 75	53
MSCCO	*Application*	1.76 %	2	68 of 75	16

Table 3: *The examined blocks.*

As we pointed out in Section 3, the persistent data are divided into Files and Common variables, where Files are arrays of records, and Common variables in most cases ”scalar” variables. For Files, we make the following observation:

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 and/or data generated during a telephone call [Lin03]. The index of an access thus usually depends on externally supplied data, like a subscriber number, which should be quite random under normal circumstances.

Based on the above, we believe that conflicts through Files tend to be rare. As a starting point we therefore make the following approximation: conflicts caused by simultaneous access to the same file does not occur! This is of course an underestimation of the actual number of conflicts. The usage of the common variables in each examined block is shown in Table 7 (CHVIEW), Table 10 (LAD), Table 12 (MFM), and Table 14 (MSCCO). The conflict matrixes

derived from the above tables are found in Table 17 (CHVIEW), Table 22 (LAD), Table 25 (MFM), and Table 28 (MSCCO). Applying the 'w-optimization' described in the previous section on the above conflict matrixes result in Table 18 (CHVIEW), and Table 23 (LAD). For the matrixes in Table 25 and Table 28 no improvement is achieved. This is due to that several signals share not only one, but several variables that are used for communication, e.g., "the current state of the system is X".

The results so far is summarized in Table 4.

Block	**Initial approx.**	$\backslash w$
CHVIEW	10.78%	10.74%
LAD	47.22%	8.33%
MFM	64.67%	64.67%
MSCCO	55.46%	55.46%

Table 4: *Summary of the (relative) number of possible conflicts, between the observed signals, with (and without) 'w-optimization' applied. The figures approximates that conflicts due to simultaneous access to the same file does not occur.*

On the other hand, by regarding Files from the other extreme, i.e., by considering **every** simultaneous access as a potential conflict, we achieve a *safe upper bound* on the number of conflicts. The usage of files in each block is shown in Table 8 (CHVIEW), Table 11 (LAD), Table 13 (MFM), and Table 15 (MSCCO). The corresponding conflict matrixes is shown Table 19 (CHVIEW), Table 24 (LAD), Table 26 (MFM), and Table 29 (MSCCO). The safe upper bound is achieved by combining Table 18 and Table 19 into Table 20 (CHVIEW), Table 23 and Table 24 into Table 24[4] (LAD), Table 25 and Table 26 into Table 27 (MFM), and Table 28 and Table 29 into Table 30 (MSCCO). Adding the upper bounds to Table 4 gives us Table 5.

Block	**Initial approx.**	$\backslash w$	**Upper bound**
CHVIEW	10.78%	10.74%	72.56%
LAD	47.22%	8.33%	33.33%
MFM	64.67%	64.67%	75.78%
MSCCO	55.46%	55.46%	90.96%

Table 5: *Adding the upper bound of the possible number of conflicts (column 4) to the figures from Table 4.*

Coming this far, the question is whether or not we can tighten the derived upper bounds. Earlier in this section we said that records in a file are used to hold subscriber data and/or data generated during a telephone call. But to prevent arbitrary accesses to a record, an instance variable (the 'state') indicates whether or not the record is currently used.

[4]The combination of Table 23 and Table 24 is identical with Table 24 since the remaining conflicts in the former are already captured in the latter.

- To `SEIZE` a record is the operation of changing the state of a record from `IDLE` to `BUSY`, where `IDLE` means "not currently used by any job", and `BUSY` "currently used to hold data".

Further on, we also know (from [Lin03]) that files (and their records) can be divided into different ”sub-classes”. But before we look into these sub-classes, we need to cover the concept of ”Forlopps” introduced in [Lin03].

In Section 2, we introduced the notions of jobs, and job-trees (the set of jobs originating from the same external signal);

- *a Forlopp is the set of one, or more, related job-trees* [Lin03].

i.e., a set of job-trees that co-operate to establish, and carry out, a telephone call. Fig. 6 - 8 (all[5] from [Lin03]) illustrate the concepts.

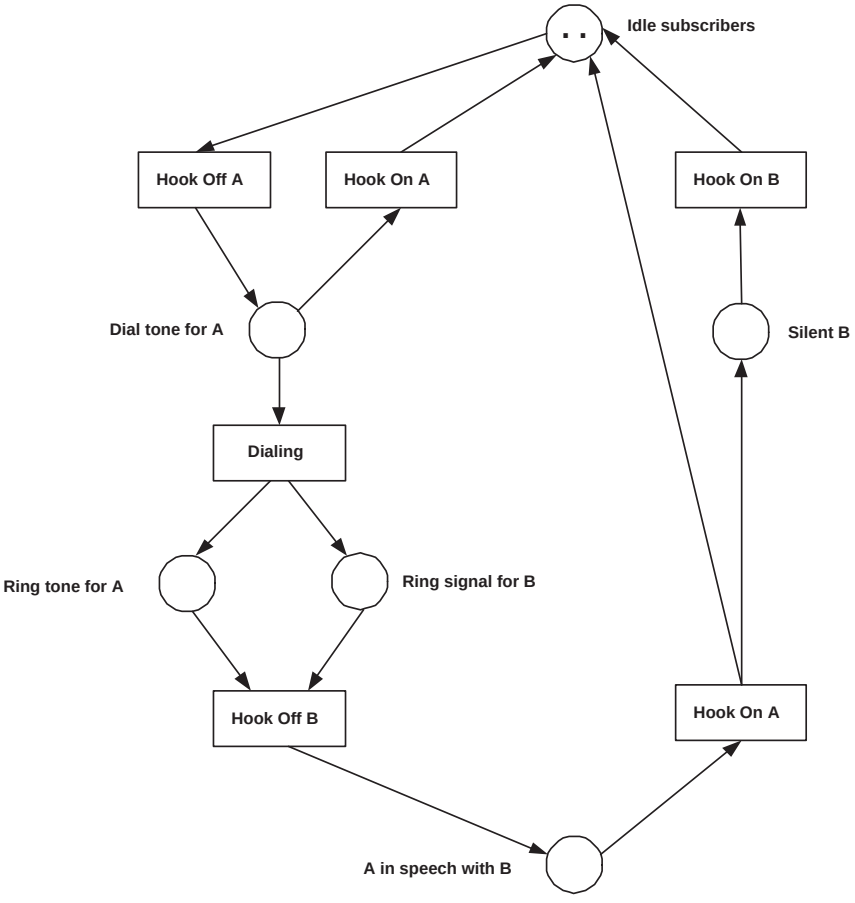

Figure 6: *A Petri-Net inspired representation of a Forlopp.*

Having covered the Forlopp-concept, we return to the discussion on files/records[6], and start dividing the files into the following classes:

[5] Fig. 6 is originally from [FW00].

[6] In the following, we will sometimes use the terms files, and records interchangeably

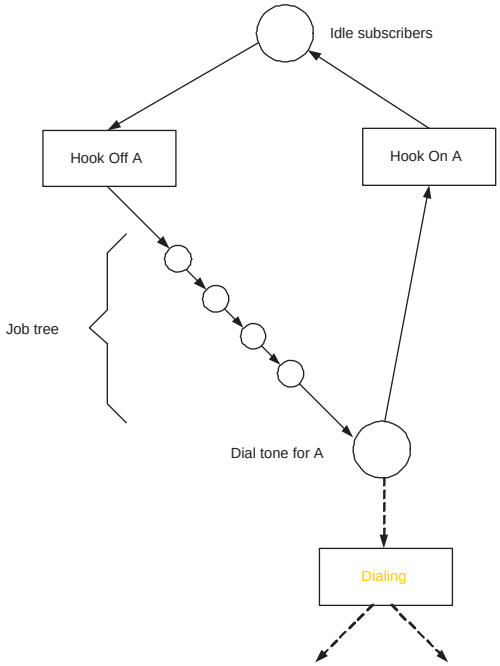

Figure 7: *Showing one of the Job-trees in the Forlopp from Fig. 6.*

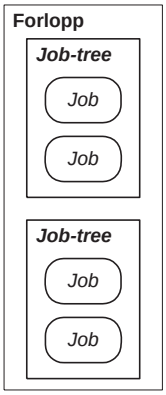

Figure 8: *The relation between jobs, job-trees, and forlopps.*

Forlopp unique: *a Forlopp unique record is a communication channel with its liveness limited to the boundary of the currently executed Forlopp. The communcation channel is not live when entering or exiting the Forlopp, and the pointer value addressing the record is solely used by the Forlopp that has seized the record*, Definition 10.2 in [Lin03]. Since the jobs in the job-trees, as well as the job-trees in the Forlopp, are sequentially executed, the only kind of conflict that can possibly occur in a Forlopp unique record, is if two different Forlopps simultaneously `SEIZE` a new record for future use. This implies that if we can classify a file as Forlopp unique, we can be sure that conflicts in the corresponding records can never occur **as long as** the SEIZE operation is protected!

Forlopp shared: similar to the Forlopp unique file, records in a "Forlopp shared" file is only used inside a Forlopp. But unlike the previous case the records are used for communication between different Forlopps which means that conflicts might occur in these records even if the SEIZE operation is secured.

Shared: a record used for communication between an arbitrary number of jobs not part of a Forlopp. It might also be the case that uses of the record rely on sequential execution, e.g., by using a constant value as a pointer value. Finally, whenever we safely can't determine any of the other two cases (Forlopp unique/shared), the record must be regarded as shared. Simultaneous access to this file type must always be treated as a potential conflict!

Examining the usage of the files leads us to Table 16 where each file is classified according to the above division. Notable is that we actually can classify 18 of 21 files as Forlopp unique, which means that **if** different signals are prevented from executing the `SEIZE` operation concurrently, there would be no conflicts in the 18 files! The `SEIZE` operation itself is a quit small piece of code, usually written in one of the following two cases

$$pointer \ = \ Commonvar; \ \ Commonvar \ = \ pointer : next$$

`FOR FIRST` *pointer* `FROM` *expression* `WHERE STATE = IDLE`
`DO STATE = BUSY`

Moreover, the `SEIZE` operation is not seldom placed in a sub-routine since many signals may perform the same operation. This means that protecting the `SEIZE` operation is a question of ensuring exclusive access to the sub-routines in question! An idea that has been discussed with our partners at Ericsson is to use the (for PLEX programmers) well known `DISABLE/ENABLE` construct in the following way:

```
DISABLE PARALLEL
 seize operation
ENABLE PARALLEL
```

The `DISABLE/ENABLE` construct is today used by code of lower priority to prevent interference from higher prioritized code[7] like in the following code snippet (and we refer to [EL02] for further information on the `DISABLE/ENABLE` construct).

```
DISABLE INTERRUPT
 low prioritized code accessing shared data
ENABLE INTERRUPT
```

[7] We recall from Section 2 that different jobs execute on different levels of priority, and that jobs of higher priority normally are allowed to interrupt a job of lower priority.

Now, if we assume that the SEIZE operations in each block is protected, not only can we safely say that potential conflicts in 18 of the 21 files are eliminated, but for 2 of the examined 4 blocks we also manage to remove **all** the remaining conflicts in the common variables! This is due to the fact that these variables are (1) only used in SEIZE operations, or (2) only accessed in the sub-routines that performs the SEIZE operations!

Based on the above assumption and discussion, as well as on our earlier examination of the common variables, we conclude our study by noting that in block CHVIEW, we only need to consider the potential conflicts in Table 9. All potential conflicts in the remaining files, as well as in the common variables, are removed under the above assumption. For block LAD we achieve even better results; **all** remaining conflicts are removed since all files are Forlopp unique, and the remaining common variables are used in SEIZE operations or will be protected if the SEIZE operations are protected. For block MFM, no file conflicts can be removed since all files are potentially shared. This means that the previous derived upper bound is our final result for the block. In block MSCCO, all considered files are classified as Forlopp unique which means, opposite to MFM, that our initial approximation is the final result for this block (since no conflicts in the common variables can be removed).

Our final figures on the possible number of shared-memory conflicts are as in Table 6, and the new conflict matrix for block CHVIEW is shown in Table 21.

Block	**Initial approx.**	$\backslash w$	**Upper bound**	**Protected SEIZE**
CHVIEW	10.78%	10.74%	72.56%	16.30%
LAD	47.22%	8.33%	33.33%	0%
MFM	64.67%	64.67%	75.78%	75.78%
MSCCO	55.46%	55.46%	90.96%	55.46%

Table 6: *The final results for the examined blocks.*

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 [Rat02]. However, the few works that we are aware of in the event-based domain, [MH01, Mos06] and [RDHS02], 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) [Kum95].

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 [DG90, zV96, Tal03]. 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 [WC95]. 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 [CBHR06]. 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 coarse grained 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, and algorithms, include

spin-locks: a busy-waiting approach that may be preferable if the waiting time is low, which we believe that it generally is. If contention on the shared is considered low, the backoff-delay version in [And90] might be an alternative to protect the same data.

diffracting trees (DT): which is a software solution to implement shared counters in a multiprocessor system [SZ96]. The Reactive DT (RDT) by [DLS00] has the ability to grow and shrink depending on work load. Since a significant part of the variables in some of the examined blocks are used as counters, RDT's might be considered for these variables.

software transactional memory (STM): in which operations on the shared data are seen as transactions (which corresponds to what we said about a job earlier in this section) that are either committed or aborted [ST95]. The Adaptive STM proposed in [MSS05] also adjusts to different workloads.

However, common for the above structures are that their reactive schemes (i.e., the algorithms) rely on either some experimentally tuned thresholds or know probability distribution of inputs. However, as shown by Ha [Ha06] it is possible to implement the algorithms in a "self-tuning" way.

7 Conclusions

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 reduced the potential conflicts between jobs to be in the range 11-65% for the observed programs. Simple static optimizations were, in some cases, able to reduce the figures even further.

These initial results were an underestimation of the actual number of conflicts since we omitted conflicts caused by simultaneous access to the same file. We chose 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.

We continued our study by regarding files from the other extreme, i.e., by considering every simultaneous access as a potential conflict. This provided us with a safe upper bound on the number of potential conflicts. These figures were found to be in the range 33-91%.

Under the assumption that the operation of fetching a record from a file is prevented from parallel execution, we showed that it is possible to tighten the upper bounds to figures in the range 0-76%.

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 this coarse grained locking scheme, we have sketched on an 'Atomic Section' solution. Another approach are the reactive data structures as implemented by Ha [Ha06].

To summarize, our 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 atomic sections/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

[And90] Thomas E Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[CBHR06] John Cieslewicz, Jonathan Berry, Bruce Hendrickson, and Kenneth 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.

[DG90] David J. DeWitt and Jim Gray. Parallel database systems: the future of database processing or a passing fad? *SIGMOD Rec.*, 19(4):104–112, 1990.

[DLS00] Giovanni Della-Libera and Nir Shavit. Reactive diffracting trees. *Journal of Parallel and Distributed Computing*, 60(7):853–890, 2000.

[EL02] J. Erikson and B. Lindell. The Execution Model of the APZ/PLEX - An Informal Description. Technical report, Mälardalen University, 2002.

[FW00] Peter Funk and Janet Wennersten. Asynchronous signal paradigm and AI for soft real time systems. Technical report, Mälardalen University, March 2000.

[Ha06] Phuong Ha. *Reactive Concurrent Data Structures and Algorithms for Synchronization*. PhD thesis, Chalmers University of Technology, 2006.

[Kum95] S. Kumar. Issues in parallelizing object-oriented programs. In *Proceedings of the 1995 ICPP Workshop on Challenges for Parallel Processing*, pages 64–71, 1995.

[Lin03] 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.

[Lin08] J. Lindhult. An Operational Semantics for the Execution of PLEX in a Shared Memory Architecture. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-227/2008-1-SE, Mälardalen University, 2008.

[MH01] 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.

[Mos06] Christof Mosler. E-CARES Project: Reengineering of PLEX Systems. *Softwaretechnik-Trends*, 26(2):59–60, 5 2006.

[MSS05] VJ Marathe, WN Scherer, and ML Scott. Adaptive software transactional memory. In *Distributed Computing, Proceedings Lecture Notes in Computer Science 3724*, pages 354–368. Springer-Verlag Berlin, 2005.

[Rat02] Rational. *Modeling Language Guide - Rational Rose Realtime*, 2002.

[RDHS02] 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.

[SL05] Herb Sutter and James Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, September 2005.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM Press.

[SZ96] Nir Shavit and Asaph Zemach. Diffracting trees. *ACM Trans. Comput. Syst.*, 14(4):385–428, 1996.

[Tal03] Ameet S. Talwadker. Survey of performance issues in parallel database systems. *J. Comput. Small Coll.*, 18(6):5–9, 2003.

[WC95] Paul Watson and George 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.

[zV96] M. Tamer Özsu and Patrick Valduriez. Distributed and parallel database systems. *ACM Comput. Surv.*, 28(1):125–128, 1996.

A Blocks, Variables, and Conflicts

The background material, which we refer to in Section 5, and from which the figures in Table 4-6 is derived, is collected in this section. Table 7-15 shows how the variables in each examined block are used, and from Section 4, we repeat our classification of the variables;

$\perp$**-** 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.

Table 16 contains the classification of the files, whereas Table 17-30 contains the conflict matrixes for the different blocks.

When the figures in Table 4-6 was calculated, it was only necessary to consider one half in each respective conflict matrix. This is (of course) due to symmetry; if $Signal_A$ may be in conflict with $Signal_B$, then obviously $Signal_B$ may be in conflict with $Signal_A$. However, $Signal_A$ may also, in some situations, be in conflict with itself. This has been included in our figures.

Finally, note that some of the signals in the tables of Section A are marked bold, and italic; these signals are the ”high-frequent” signals mentioned in the beginning of Section 5.

Block: CHVIEW (Middleware) *variables →* *signals ↓*	CPUBDATAID	CPUBDATAVALUE	CPUB2DATAID	CPUB2DATAVALUE	CGLOBALSUBIUA	CGLOBALSUBSREF	CSTARTIDLELISTLOGPRT	CSTARTIDLELISTVIEW	CSTARTIDLELISTLOGHDR	CSETMASK	CGLOBALSUBPCDI	CTMZSUPPORTED	CAUTOSIZEALTSUPPORT	COWNREF	CSTOPIDLELISTLOGPRT	CPREPNUMLOGPART	CSTOPIDLELISTVIEW	CSTOPIDLELISTLOGHDR	CPUBLICDEFAULTS	CCLEARSUBSCR
WRITEPUBLIC	W	W	⊤	⊤	R	R	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
READPUBLIC	W	W	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
READLOG	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
CREGENVIEW	⊥	⊥	⊥	⊥	⊥	R	⊤	⊤	⊤	R	R	R	R	R	W	⊥	W	W	R	R
WRITELOGEND	⊥	⊥	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	R	R	⊤	R	⊥	⊥	⊥	⊥
CREATEVIEW	⊥	⊥	⊥	⊥	⊥	⊥	⊤	⊤	⊤	R	⊥	R	R	R	W	⊥	W	W	⊥	⊥
INITVIEW	⊥	⊥	⊥	⊥	⊥	⊥	⊤	W	W	R	⊥	⊥	R	R	⊤	R	⊤	⊤	⊥	⊥
OPENVIEWCON	⊥	⊥	⊥	⊥	⊥	⊥	⊤	⊥	⊤	R	⊥	⊥	R	R	W	⊥	⊥	W	⊥	⊥
WRITEPUBEND	⊥	⊥	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	R	R	⊤	R	⊥	⊥	⊥	⊥
WRITELOG	⊥	⊥	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	R	R	⊤	R	⊥	⊥	⊥	⊥
RESERVEVIEW	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊤	⊤	⊥	⊥	R	R	R	⊥	⊥	W	W	⊥	⊥
STOREFEP	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
OPENSESSION1R	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥
OPENSESSION1REJ	⊥	⊥	⊥	⊥	⊥	⊥	W	W	W	R	⊥	⊥	⊥	R	⊤	R	⊤	⊤	⊥	⊥
INITOUTPUT	⊥	⊥	⊥	⊥	⊥	⊥	W	W	W	R	⊥	⊥	⊥	R	⊤	R	⊤	⊤	⊥	⊥
CONTINUEB	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥
REQRESKEY	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
STARTCHOUTPUT	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	R	R	R	⊥	⊥	⊥	⊥	⊥	⊥
PARTOUTBREAK	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
FOAMTRACE	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
RELVIEWCON	⊥	⊥	⊥	⊥	⊥	⊥	W	W	W	⊥	⊥	⊥	⊥	⊥	⊤	R	⊤	⊤	⊥	⊥
CANLOCPUBLIC1	⊥	⊥	⊥	⊥	⊥	⊥	W	W	⊥	⊥	⊥	⊥	⊥	⊥	⊤	R	⊤	⊥	⊥	⊥
UNDOWRITELOG	⊥	⊥	⊥	⊥	⊥	⊥	W	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊤	R	⊥	⊥	⊥	⊥
CLEARLOG	⊥	⊥	⊥	⊥	⊥	⊥	W	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊤	R	⊥	⊥	⊥	⊥
PARTOUTREQ	⊥	⊥	⊥	⊥	⊥	⊥	W	⊤	⊤	⊥	⊥	⊥	⊥	R	⊤	R	⊤	⊤	⊥	⊥
OUTPUTVAREQR	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊤	⊤	⊥	⊥	⊥	⊥	R	⊥	⊥	W	W	⊥	⊥
DISASSOCVIEW	⊥	⊥	⊥	⊥	⊥	⊥	W	W	W	⊥	⊥	⊥	⊥	⊥	⊤	R	⊤	⊤	⊥	⊥
DELAYDISASSOC	⊥	⊥	⊥	⊥	⊥	⊥	W	W	W	⊥	⊥	⊥	⊥	⊥	⊤	R	⊤	⊤	⊥	⊥
SPLITVA	⊥	⊥	⊥	⊥	⊥	⊥	W	W	W	⊥	⊥	⊥	⊥	⊥	⊤	R	⊤	⊤	⊥	⊥
DELAYSPLITVA	⊥	⊥	⊥	⊥	⊥	⊥	W	W	W	⊥	⊥	⊥	⊥	⊥	⊤	R	⊤	⊤	⊥	⊥
NOOUTPUTR	⊥	⊥	⊥	⊥	⊥	⊥	W	W	W	⊥	⊥	⊥	⊥	R	⊤	R	⊤	⊤	⊥	⊥
CHARGRESULTSR	⊥	⊥	⊥	⊥	⊥	⊥	W	W	W	⊥	⊥	⊥	⊥	⊥	⊤	R	⊤	⊤	⊥	⊥
PARTOUTREADYR	⊥	⊥	⊥	⊥	⊥	⊥	W	W	W	⊥	⊥	⊥	⊥	⊥	⊤	R	⊤	⊤	⊥	⊥
SETPRECONGFEP	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥
PRECONGFEP	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥
CLEARPRECONGFEP	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥
PRECONGFEPCLRD	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥
OUTPUTVAREQREJ	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥
OUTPUTFEPR	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥
OUTPUTFEPREJ	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥
CLOSESESSION1R	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥
CLEARFULLCONG	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥
GETSTRUCTOT	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥
SWITCHFEPS	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥
FEPSWITCHEDR	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥

Table 7: CHVIEW - usage of common variables.

Block: CHVIEW (Middleware) *Files →* *signals ↓*	VIEWFILE	LOGHEADER	LOGPART	PUBLICDATA	SESSION	SAECNTRECORD	*Files →* *signals ↓*	VIEWFILE	LOGHEADER	LOGPART	PUBLICDATA	SESSION	SAECNTRECORD
WRITEPUBLIC	R	⊥	⊥	W	⊥	⊥	OUTPUTFEPR	⊤	⊤	⊤	⊥	⊤	⊤
READPUBLIC	R	⊥	⊥	R	⊥	⊥	OUTPUTFEPREJ	⊤	⊤	⊤	⊥	⊤	⊤
READLOG	⊥	R	R	⊥	W	⊥	CLOSESESSION1R	⊤	⊤	⊤	⊥	⊤	⊤
CREGENVIEW	⊤	⊤	⊤	W	⊥	⊤	CLEARFULLCONG	⊤	⊤	⊤	⊥	⊤	⊤
WRITELOGEND	⊤	W	⊤	⊥	⊥	⊤	GETSTRUCTOT	⊤	R	R	⊥	W	⊥
CREATEVIEW	⊤	⊤	⊤	⊥	⊥	⊤	CHVIEWCONTRACE	⊤	⊤	⊥	⊥	⊥	⊥
INITVIEW	W	W	⊤	⊥	⊥	⊤	CHVIEWCONCLEAR	⊤	W	⊥	⊥	⊥	⊥
OPENVIEWCON	⊤	⊤	⊤	⊥	⊥	⊤	RVIEWSTRUC	W	⊥	⊥	⊥	⊥	⊥
WRITEPUBEND	R	⊥	⊤	⊤	⊥	⊤	STORERESKEY	W	⊥	⊥	⊥	⊥	⊥
WRITELOG	⊤	W	⊤	⊥	⊥	⊤	SUBLOCPUBCDT	R	⊥	⊥	⊤	⊥	⊥
RESERVEVIEW	⊤	⊤	⊥	⊥	⊥	⊤	VIEWSTATUS	W	⊥	⊥	⊥	⊥	⊥
STOREFEP	⊤	W	⊥	⊥	⊥	⊥	VIEWSERVICE	W	⊥	⊥	⊥	⊥	⊥
OPENSESSION1R	⊤	⊤	⊤	⊥	⊤	⊤	VIEWSPEC	W	⊥	⊥	⊥	⊥	⊥
OPENSESSION1REJ	⊤	⊤	⊤	⊥	⊤	⊤	UPDATEEVENT	W	⊥	⊥	⊥	⊥	⊥
INITOUTPUT	⊤	⊤	⊤	⊥	⊤	⊤	READVABSTIME	W	⊥	⊥	⊥	⊥	⊥
CONTINUEB	⊤	⊤	⊤	⊥	⊤	⊤	ASSOCVIEW	⊤	R	⊥	⊥	⊥	⊥
STARTCHOUTPUT	⊤	⊤	⊥	⊥	⊥	R	DELAYASSOC1	⊤	R	⊥	⊥	⊥	⊥
PARTOUTBREAK	⊤	⊤	⊥	⊥	⊥	⊥	SETVAVIEWMIS	⊤	⊥	⊥	⊥	⊥	⊥
FOAMTRACE	⊤	⊥	⊥	⊥	⊤	⊥	FEPSTREATED	⊤	⊥	⊥	⊥	W	⊥
RELVIEWCON	⊤	⊤	⊤	⊤	⊥	⊤	GETNEXTVIEW	⊤	⊥	⊥	⊥	W	⊥
CANLOCPUBLIC1	⊤	⊥	W	W	⊥	⊤	GETVABSTIME	R	⊥	⊥	⊥	W	⊥
UNDOWRITELOG	⊤	W	⊤	⊥	⊥	⊤	GETFEPPONUM	R	⊥	⊥	⊥	W	⊥
CLEARLOG	⊤	W	⊤	⊥	⊥	⊤	GETVIEWINFO	R	R	R	⊥	W	⊥
PARTOUTREQ	W	⊤	⊤	⊥	⊥	⊤	TRACEAM	R	⊥	⊥	⊥	⊥	⊥
OUTPUTVAREQR	⊤	⊤	⊥	⊥	⊥	⊤	SUBLOCPUBLIC	R	⊥	⊥	⊤	⊥	⊥
DISASSOCVIEW	⊤	⊤	⊤	⊥	⊥	⊤	CANLOCPUBLIC	R	⊥	⊥	⊤	⊥	⊥
SPLITVA	⊤	⊤	⊤	⊥	⊥	⊤	COPYPUBLIC	R	⊥	⊥	⊤	⊥	⊥
NOOUTPUTR	⊤	⊤	⊤	⊥	⊥	⊤	GETNEXTLOG	⊥	R	R	⊥	W	⊥
CHARGRESULTSR	W	⊤	⊤	⊥	⊥	⊤	STOREAC	⊥	W	⊥	⊥	⊥	⊥
PARTOUTREADYR	⊤	⊤	⊤	⊥	⊥	⊤	FOAMCLEAR	⊥	⊥	⊥	⊥	W	⊥
OUTPUTVAREQREJ	⊤	R	⊥	⊥	⊥	⊥							

Table 8: CHVIEW - usage of File variables

Block: CHVIEW (Middleware) *Files* → *signals* ↓							SAECNTRECORD	*Files* → *signals* ↓						SAECNTRECORD
WRITEPUBLIC							⊥	OUTPUTFEPR						⊤
READPUBLIC							⊥	OUTPUTFEPREJ						⊤
READLOG							⊥	CLOSESESSION1R						⊤
CREGENVIEW							⊤	CLEARFULLCONG						⊤
WRITELOGEND							⊤	GETSTRUCTOT						⊥
CREATEVIEW							⊤	CHVIEWCONTRACE						⊥
INITVIEW							⊤	CHVIEWCONCLEAR						⊥
OPENVIEWCON							⊤	RVIEWSTRUC						⊥
WRITEPUBEND							⊤	STORERESKEY						⊥
WRITELOG							⊤	SUBLOCPUBCDT						⊥
RESERVEVIEW							⊤	VIEWSTATUS						⊥
STOREFEP							⊥	VIEWSERVICE						⊥
OPENSESSION1R							⊤	VIEWSPEC						⊥
OPENSESSION1REJ							⊤	UPDATEEVENT						⊥
INITOUTPUT							⊤	READVABSTIME						⊥
CONTINUEB							⊤	ASSOCVIEW						⊥
STARTCHOUTPUT							R	DELAYASSOC1						⊥
PARTOUTBREAK							⊥	SETVAVIEWMIS						⊥
FOAMTRACE							⊥	FEPSTREATED						⊥
RELVIEWCON							⊤	GETNEXTVIEW						⊥
CANLOCPUBLIC1							⊤	GETVABSTIME						⊥
UNDOWRITELOG							⊤	GETFEPPONUM						⊥
CLEARLOG							⊤	GETVIEWINFO						⊥
PARTOUTREQ							⊤	TRACEAM						⊥
OUTPUTVAREQR							⊤	SUBLOCPUBLIC						⊥
DISASSOCVIEW							⊤	CANLOCPUBLIC						⊥
SPLITVA							⊤	COPYPUBLIC						⊥
NOOUTPUTR							⊤	GETNEXTLOG						⊥
CHARGRESULTSR							⊤	STOREAC						⊥
PARTOUTREADYR							⊤	FOAMCLEAR						⊥
OUTPUTVAREQREJ							⊥							

Table 9: CHVIEW - file variables that need to be considered under the assumption that SEIZE is not executed in parallel.

Block: LAD (OS) *variables* → *signals* ↓	ALLBUF	GETBUFABSADR	READBSIZE	***ALLCOMBUF***	CBFLCONNECT	GETCOMBUFREF	***RELCOMBUF***	READCBSTATE
CCOMBANK512FREELISTEXIT	⊣	⊣	⊣	W	⊣	⊣	W	⊣
CCOMBANK512MAXNUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK512NUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK512NUMBOFCONGESTIONS	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK512FREELISTENTRY	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK256FREELISTEXIT	⊣	⊣	⊣	W	⊣	⊣	W	⊣
CCOMBANK256MAXNUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK256NUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK256NUMBOFCONGESTIONS	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK256FREELISTENTRY	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK128FREELISTEXIT	⊣	⊣	⊣	W	⊣	⊣	W	⊣
CCOMBANK128MAXNUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK128NUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK128NUMBOFCONGESTIONS	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK128FREELISTENTRY	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK64FREELISTEXIT	⊣	⊣	⊣	W	⊣	⊣	W	⊣
CCOMBANK64MAXNUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK64NUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK64NUMBOFCONGESTIONS	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK64FREELISTENTRY	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK32FREELISTEXIT	⊣	⊣	⊣	W	⊣	⊣	W	⊣
CCOMBANK32MAXNUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK32NUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK32NUMBOFCONGESTIONS	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK32FREELISTENTRY	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK32KPREPNUM	⊣	⊣	⊣	⊣	⊣	⊣	R	R
CCOMBANK16KPREPNUM	⊣	⊣	⊣	⊣	⊣	⊣	R	R
CCOMBANK8KPREPNUM	⊣	⊣	⊣	⊣	⊣	⊣	R	R
CCOMBANK4KPREPNUM	⊣	⊣	⊣	⊣	⊣	⊣	R	R
CCOMBANK2KPREPNUM	⊣	⊣	⊣	⊣	⊣	⊣	R	R
CCOMBANK1KPREPNUM	⊣	⊣	⊣	⊣	⊣	⊣	R	R
CCOMBANK512PREPNUM	⊣	⊣	⊣	⊣	⊣	⊣	R	R
CCOMBANK256PREPNUM	⊣	⊣	⊣	⊣	⊣	⊣	R	R
CCOMBANK128PREPNUM	⊣	⊣	⊣	⊣	⊣	⊣	R	R
CCOMBANK64PREPNUM	⊣	⊣	⊣	⊣	⊣	⊣	R	R
CCOMBANK32PREPNUM	⊣	⊣	⊣	⊣	⊣	⊣	R	R
COWNREF	R	⊣	⊣	R	⊣	⊣	⊣	⊣
CAUTOSAACTIVE	R	⊣	⊣	R	⊣	⊣	⊣	⊣
CCOMBANKRESULTCODETEMP	⊣	⊣	⊣	W	⊣	⊣	⊣	⊣
CCOMBANKUSERPOINTERTEMP	⊣	⊣	⊣	W	W	W	W	⊣
CCLOCKDATE	R	⊣	⊣	R	⊣	⊣	⊣	⊣
CPROGERRORBUFFERTYPE	W	W	W	W	W	⊣	⊣	⊣
CPROGERRORCODE	W	W	W	W	W	⊣	⊣	⊣

	ALLBUF	GETBUFABSADR	READBSIZE	***ALLCOMBUF***	CBFLCONNECT	GETCOMBUFREF	***RELCOMBUF***	READCBSTATE
CCOMBANKNIL	⊣	⊣	⊣	R	⊣	⊣	R	⊣
CCOMBANKTIMEOUTQUEUECARDINALITY	⊣	⊣	⊣	⊣	⊣	⊣	⊢	⊣
CCOMBANKTIMEOUTQUEUEENTRY	⊣	⊣	⊣	⊣	⊣	⊣	W	⊣
CCOMBANKTIMEOUTQUEUEEXIT	⊣	⊣	⊣	⊣	⊣	⊣	W	⊣
CCOMBANK32KFREELISTEXIT	⊣	⊣	⊣	W	⊣	⊣	W	⊣
CCOMBANK32KMAXNUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK32KNUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK32KNUMBOFCONGESTIONS	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK32KFREELISTENTRY	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK16KFREELISTEXIT	⊣	⊣	⊣	W	⊣	⊣	W	⊣
CCOMBANK16KMAXNUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK16KNUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK16KNUMBOFCONGESTIONS	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK16KFREELISTENTRY	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK8KFREELISTEXIT	⊣	⊣	⊣	W	⊣	⊣	W	⊣
CCOMBANK8KMAXNUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK8KNUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK8KNUMBOFCONGESTIONS	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK8KFREELISTENTRY	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK4KFREELISTEXIT	⊣	⊣	⊣	W	⊣	⊣	W	⊣
CCOMBANK4KMAXNUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK4KNUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK4KNUMBOFCONGESTIONS	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK4KFREELISTENTRY	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK2KFREELISTEXIT	⊣	⊣	⊣	W	⊣	⊣	W	⊣
CCOMBANK2KMAXNUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK2KNUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK2KNUMBOFCONGESTIONS	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK2KFREELISTENTRY	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK1KFREELISTEXIT	⊣	⊣	⊣	W	⊣	⊣	W	⊣
CCOMBANK1KMAXNUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK1KNUMBOFUSEDIND	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣
CCOMBANK1KNUMBOFCONGESTIONS	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CCOMBANK1KFREELISTENTRY	⊣	⊣	⊣	⊢	⊣	⊣	⊢	⊣

Table 10: LAD - usage of common variables.

Block: LAD (OS) *Files* → *signals* ↓	COMBANK32	COMBANK64	COMBANK128	COMBANK256	COMBANK512	COMBANK1K	COMBANK2K	COMBANK4K	COMBANK8K	COMBANK16K	COMBANK32K
ALLCOMBUF	W	W	W	W	W	W	W	W	W	W	W
CBFLCONNECT	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
GETCOMBUFREF	R	R	R	R	R	R	R	R	R	R	R
RELCOMBUF	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
READCBSTATE	R	R	R	R	R	R	R	R	R	R	R

Table 11: LAD - usage File variables.

Block: MFM (OS) *variables* → *signals* ↓	FLSTARTUNRQ	***FLSTARTCORQ***	FLPARTRQ	FLPARTEXTRQ	***FLLEAVERQ***	FLJOINRQ	FLUNJOINRQ	FLSAVERQ	FLUNSAVERQ	FLABORTRQ	FLRELEASERQ	INQRESTFLRQ	RESTARTFLRQ	MFMOPERATER	CPBREAKR	INITFLERROR	FLPROTECTRQ	CHECKJOINFID	INQSTATFLRQ	FLERRORCREATE	***FLAUDITRQ***	EVENTMESSAGEA	MFMREADR	FLERRORINFR	FLUPDATEDUMP	FLERRORRQ
COWNBLOCKREF	R	R	R	R	R	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	R	R	⊣	⊣	⊣	⊣	⊣	⊣	⊣	R	⊣	⊣	⊣
CSYFSCACTIVEDELAY	⊢	⊢	⊢	⊢	⊢	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	W	⊣	⊣	⊣
CECFONDELAY06	W	W	W	W	W	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣
CECFACTIONAFTERCONG06	R	R	R	R	R	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣
CTQSECSTATUSUPDATE	W	W	W	W	W	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣
CTQSECLOADSCAN	W	W	W	W	W	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣
CREQUESTSUBMODE06	W	W	W	W	W	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣
CUPDATEINPROGRESS06	W	W	W	W	W	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	W	⊣	⊣	⊣	⊣
CFLMODE	R	R	R	R	R	⊣	⊣	⊣	⊣	⊣	⊣	⊣	R	R	R	R	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣
CTQMINCONGIND	⊢	⊢	⊢	⊢	⊢	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣
CREBUILDIDLELISTS	W	W	W	W	W	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	W	W	⊣	⊣
CTQMINCONGFL	⊢	⊢	⊢	⊢	⊢	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	W	⊣	⊣	⊣
CTQSECCONGFLREL	W	W	W	W	W	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	R	R	⊣	⊣	⊣	⊣	⊣	⊣	⊣	R	⊣	⊣	⊣
CNUMSIZEALTDECRFL	⊢	⊢	⊢	⊢	⊢	⊢	⊢	⊢	⊢	⊣	⊣	⊣	⊢	⊣	⊢	⊣	⊣	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣	⊣
CPREPNUMFL	R	R	R	R	R	R	R	R	R	⊣	⊣	⊣	R	R	R	⊣	⊣	⊣	⊣	⊣	R	R	⊣	⊣	⊣	⊣
CSIZEALTDECRFL	R	R	R	R	R	R	R	R	R	⊣	⊣	R	R	⊣	R	⊣	⊣	⊣	⊣	⊣	⊢	R	⊣	⊣	⊣	⊣
CNUMOFUSEDFLREC	⊢	⊢	⊢	⊢	⊢	⊢	⊢	⊢	⊢	⊣	⊢	⊣	⊢	⊣	⊢	⊣	⊣	⊣	⊣	⊣	⊢	W	⊣	⊣	⊣	⊣
CNUMSIZEALTDECRIND	⊣	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣
CPREPNUMIND	R	R	R	R	R	R	R	R	R	⊣	⊣	⊣	R	⊣	R	⊣	⊣	⊣	⊣	⊣	R	⊢	⊣	⊣	⊣	⊣
CNUMOFUSEDINDREC	⊢	⊢	⊢	⊢	⊢	⊢	⊢	⊢	⊢	⊣	⊣	⊣	⊢	⊣	⊢	⊣	⊣	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣	⊣
CIXTOLASTIDLEINDREC	⊣	⊣	⊣	⊣	W	W	W	W	⊣	⊣	⊣	⊣	W	W	W	⊣	⊣	⊣	⊣	⊣	W	W	⊣	⊣	⊣	⊣
CPTRTOLASTIDLEINDREC	⊣	⊣	⊣	⊣	W	W	W	W	⊣	⊣	⊣	⊣	W	W	W	⊣	⊣	⊣	⊣	⊣	W	W	⊣	⊣	⊣	⊣
CSIZEALTDECRIND	R	R	R	R	R	R	R	R	⊣	⊣	⊣	⊣	R	⊣	R	⊣	⊣	⊣	⊣	⊣	R	W	⊣	⊣	⊣	⊣
CFLECFSTATUS	R	R	R	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	W	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣
CMAXNUMPASSIVEIND02	⊣	R	R	R	R	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣
CSIZEALTINCRIND	⊢	⊢	⊢	⊢	⊢	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	W	⊣	⊣	⊣
CINDINDNUM	R	R	R	R	R	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	R	⊣	⊣	⊣
CLARGERESTARTINDIX	⊢	⊢	⊢	⊢	⊢	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	W	⊣	⊣	⊣
CLARGERESTARTINDPTR	⊢	⊢	⊢	⊢	⊢	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	W	⊣	⊣	⊣
CRESTARTINDTYPE	R	R	R	R	R	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	W	⊣	⊣	⊣
CRESTARTINDMARK	⊢	⊢	⊢	⊢	⊢	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	W	⊣	⊣	⊣
CPTRTOFIRSTIDLEINDREC	⊢	⊢	⊢	⊢	⊢	⊢	⊢	⊢	⊣	⊣	⊣	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣
CIXTOFIRSTIDLEINDREC	⊢	⊢	⊢	⊢	⊢	⊢	⊢	⊢	⊣	⊣	⊣	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣
CSTATICLINKINCL	W	W	W	⊣	⊣	⊣	⊣	W	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣
CFLSUBMODE	⊢	⊢	⊢	⊢	⊢	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊢	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣
CTIMESTAMP	R	R	R	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	R	R	R	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣
CSIZEALTINCRFL	⊢	⊢	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	R	⊣	⊣	⊣	⊣	⊣	⊣	R	⊣	⊣	⊣
CPTRTOLASTIDLEFLREC	⊢	⊢	W	W	W	⊢	⊢	⊣	W	W	W	W	⊣	⊢	⊣	⊣	⊣	⊣	⊣	⊣	⊢	⊣	W	⊣	⊣	⊣
CCOPYPTRTOFIRSTIDLEFLREC	W	W	W	W	W	W	W	⊣	W	W	W	W	⊣	⊣	W	⊣	⊣	⊣	⊣	⊣	⊣	W	W	⊣	⊣	⊣
CPTRTOFIRSTIDLEFLREC	⊢	⊢	W	W	W	W	W	⊣	W	W	W	W	⊣	⊣	W	⊣	⊣	⊣	⊣	⊣	W	W	W	⊣	⊣	⊣
CTQMINSUPERVCONG	W	⊢	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	W	⊣	⊣	⊣
CPTSFLTRACEINPROGR	R	R	R	R	R	⊣	R	⊣	⊣	⊣	R	⊣	⊣	R	R	R	⊣	⊣	⊣	⊣	⊣	⊣	R	⊣	⊣	⊣
CSAVEDOUBLE02	W	W	W	W	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	R	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣
CLARGERESTARTFLPTR	⊢	⊢	R	R	R	⊣	⊣	R	R	R	⊣	⊣	⊣	R	R	⊣	⊣	⊣	⊣	⊣	⊣	⊣	W	⊣	⊣	⊣
CRESTARTFLTYPE	R	R	R	R	R	⊣	⊣	R	R	R	⊣	⊣	⊣	R	R	⊣	⊣	⊣	⊣	⊣	⊣	⊣	W	⊣	⊣	⊣
CRESTARTFLMARK	⊢	⊢	R	R	R	⊣	⊣	⊣	R	R	⊣	⊣	⊣	R	R	⊣	⊣	⊣	⊣	⊣	⊣	⊣	W	⊣	⊣	⊣
CFCDATINPROGR	R	R	R	R	R	⊣	⊣	⊣	R	R	⊣	R	⊣	R	R	⊣	R	⊣	⊣	⊣	W	⊣	W	⊣	⊣	⊣
CRESTARTID	R	R	R	R	R	⊣	R	R	R	R	⊣	⊣	⊣	R	R	⊣	⊣	R	⊣	⊣	R	⊢	⊣	R	⊣	⊣
CFLINDNUM	R	R	R	R	R	R	⊣	R	R	R	R	⊣	⊣	R	R	W	⊣	⊣	R	⊣	R	⊣	⊣	R	⊣	⊣
CFORLOPPID	W	W	W	W	⊢	⊢	W	W	W	⊢	W	⊣	⊣	⊢	W	⊢	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	W
CPROTECTMFMDATA	W	W	W	W	W	W	W	W	W	⊢	⊢	⊢	W	⊣	W	⊣	⊢	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣	⊣

Table 12: MFM - usage of common variables.

Block: MFM (OS) *Files →* *signals ↓*	FLREC	INDREC
FLSTARTUNRQ	⊤	⊤
FLSTARTCORQ	⊤	⊤
FLPARTRQ	⊤	⊤
FLPARTEXTRQ	⊤	⊤
FLLEAVERQ	⊤	⊤
FLJOINRQ	⊤	⊤
FLUNJOINRQ	⊤	⊤
FLSAVERQ	⊤	⊥
FLUNSAVERQ	⊤	⊥
FLABORTRQ	⊤	⊥
FLRELEASERQ	⊤	⊥
INQRESTFLRQ	R	R
RESTARTFLRQ	⊤	⊤
CPBREAKR	⊤	⊤
FLPROTECTRQ	⊤	⊥
CHECKJOINFID	R	⊥
INQSTATFLRQ	R	⊥
FLERRORCREATE	R	R
FLAUDITRQ	⊤	⊤
MFMREADR	⊤	⊤

Table 13: MFM - usage File variables.

Block: MSCCO (Application) *variables →* *signals ↓*	CMESSAGEBUFR	CDATAREADFROMTEMP	CMESSAGEBUFRA	CANTERALENGTH	CUNEXPECTEDEXIT	CDATALENGTH	CSSN	CINDEX	CEXTSPC1	CEXTSPC2	CPRIORITYANSI	COWNSPMBM	COWNSPNET	COWNSPCLU
SCCONNIND	W	⊥	⊥	⊥	⊥	⊥	⊥	W	⊥	W	⊥	⊥	⊥	⊥
SCCONNINDE	W	⊥	⊥	⊥	⊥	⊥	⊥	⊤	R	⊤	⊥	⊥	⊥	⊥
RNOCOMREQ	W	⊥	⊥	⊥	⊤	⊤	W	W	W	W	R	R	R	R
RNOCOMREQL	W	⊥	⊥	⊥	⊤	⊤	W	W	W	W	R	R	R	R
C7CREFIND2	W	⊥	⊥	⊥	⊤	⊥	⊥	W	⊥	⊥	⊥	⊥	⊥	⊥
SCCONNCONF	W	⊥	⊥	⊥	⊤	⊥	⊥	W	⊥	⊥	⊥	⊥	⊥	⊥
SCCREFIND	W	⊥	⊥	⊥	⊤	⊥	⊥	W	⊥	⊥	⊥	⊥	⊥	⊥
C7DATAIND2	W	W	W	⊤	⊤	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
SCDATAIND	W	W	W	⊤	⊤	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
SCCONNCONFE	W	⊥	⊥	⊥	⊤	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥
C7CREFIND2E	W	⊥	⊥	⊥	⊤	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥
SCCREFINDE	W	⊥	⊥	⊥	⊤	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥
C7DATAIND2E	W	W	W	⊤	⊤	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥
SCDATAINDE	W	W	W	⊤	⊤	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥
C7DATARET2	W	⊥	W	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
C7DATARET2E	W	⊥	W	⊥	⊤	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥
SCDATARET	W	⊥	W	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
SCDATARETE	W	⊥	W	⊥	⊤	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥
CBSEIZER	⊥	⊤	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
C7CONNCONF2	W	⊥	⊥	⊥	⊤	⊥	⊥	W	⊥	⊥	⊥	⊥	⊥	⊥
C7CONNCONF2E	W	⊥	⊥	⊥	⊤	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥
C7DISCIND2	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
C7DISCIND2E	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
C7SCSEIZED	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
C7SCCONG	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
SCDISCIND	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
SCDISCINDE	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
BSCPTODPCR1	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	W	⊥	⊥	⊥
OPCTOBSCPR	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
RRLAYCONG	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
RRLAYCONRESP	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
RRLAYDISC	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
RRLAYLINKED	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
RRSEIZEFAIL	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
CBSEIZEF	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
C7CONNIND2I	W	⊥	⊥	⊥	⊥	⊥	⊥	W	⊥	⊥	⊥	⊥	⊥	⊥
C7CONNIND2S	W	⊥	⊥	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥
SCCONNINDI	W	⊥	⊥	⊥	⊥	⊥	⊥	W	W	W	⊥	⊥	⊥	⊥
SCCONNINDS	W	⊥	⊥	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥
C7CONNCONF2I	W	⊥	⊥	⊥	⊥	⊥	⊥	W	⊥	⊥	⊥	⊥	⊥	⊥
C7CREFIND2I	W	⊥	⊥	⊥	⊥	⊥	⊥	W	⊥	⊥	⊥	⊥	⊥	⊥
SCCONNCONFI	W	⊥	⊥	⊥	⊥	⊥	⊥	W	⊥	⊥	⊥	⊥	⊥	⊥
SCCREFINDI	W	⊥	⊥	⊥	⊥	⊥	⊥	W	⊥	⊥	⊥	⊥	⊥	⊥
C7DATAIND2I	W	⊥	⊥	⊥	⊥	⊥	⊥	W	⊥	⊥	⊥	⊥	⊥	⊥
SCDATAINDI	W	⊥	⊥	⊥	⊥	⊥	⊥	W	⊥	⊥	⊥	⊥	⊥	⊥
C7CONNCONF2S	W	⊥	⊥	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥
SCCONNCONFS	W	⊥	⊥	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥
C7CREFIND2S	W	⊥	⊥	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥
SCCREFINDS	W	⊥	⊥	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥
C7DATAIND2S	W	⊥	⊥	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥
SCDATAINDS	W	⊥	⊥	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊥	⊥	⊥	⊥
C7CONNIND2E	W	⊥	⊥	⊥	⊥	⊥	⊥	⊤	R	⊤	⊥	⊥	⊥	⊥
SCDATARETI	W	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
SCOWNSPINFO	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	W	W	W
C7CONNIND2	W	⊥	⊥	⊥	⊥	⊥	⊥	W	⊥	W	⊥	⊥	⊥	⊥
SEIZECOO	R	⊥	⊥	⊥	⊥	⊥	⊥	W	⊥	W	⊥	⊥	⊥	⊥
C7DATARET2I	W	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
C7DATARET2S	W	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥
SCDATARETS	W	⊥	⊥	⊥	⊥	⊥	⊥	R	⊥	⊥	⊥	⊥	⊥	⊥

Table 14: MSCCO - usage of common variables.

Block: MSCCO (Application) *Files* → *signals* ↓	MSCCODATADATA	TEMPRECORD	*Files* → *signals* ↓	MSCCODATADATA	TEMPRECORD
SCCONNIND	⊤	⊥	RRLAYLINKED	⊤	⊥
SCCONNINDE	⊤	⊥	RRSEIZEFAIL	⊤	⊥
RNOCOMREQ	⊤	⊥	CBSEIZEF	⊤	⊤
RNOCOMREQL	⊤	⊥	C7CONNCONF2I	⊤	⊥
C7CREFIND2	⊤	⊥	C7CREFIND2I	⊤	⊥
SCCONNCONF	⊤	⊥	SCCONNCONFI	⊤	⊥
SCCREFIND	⊤	⊥	SCCREFINDI	⊤	⊥
C7DATAIND2	⊤	⊤	C7DATAIND2I	⊤	⊥
SCDATAIND	⊤	⊤	SCDATAINDI	⊤	⊥
SCCONNCONFE	⊤	⊥	C7CONNCONF2S	R	⊥
C7CREFIND2E	⊤	⊥	SCCONNCONFS	R	⊥
SCCREFINDE	⊤	⊥	C7CREFIND2S	R	⊥
C7DATAIND2E	⊤	⊤	SCCREFINDS	R	⊥
SCDATAINDE	⊤	⊤	C7DATAIND2S	R	⊥
C7DATARET2	R	⊥	SCDATAINDS	R	⊥
C7DATARET2E	R	⊥	C7CONNIND2E	⊤	⊥
SCDATARET	⊤	⊥	SCDATARETI	R	⊥
SCDATARETE	⊤	⊥	C7CONNIND2	⊤	⊥
CBSEIZER	⊤	⊤	SEIZECOO	⊤	⊥
C7CONNCONF2	⊤	⊥	C7DATARET2I	R	⊥
C7CONNCONF2E	⊤	⊥	C7DATARET2S	R	⊥
C7DISCIND2	⊤	⊤	SCDATARETS	R	⊥
C7DISCIND2E	⊤	⊤	C7DISCIND2I	⊤	⊥
C7SCSEIZED	⊤	⊥	C7DISCIND2S	R	⊥
C7SCCONG	⊤	⊥	CBRELEASER	⊤	⊥
SCDISCIND	⊤	⊤	RSCLEAR	⊤	⊥
SCDISCINDE	⊤	⊤	RSTRACE	⊤	⊥
BSCPTODPCR1	⊤	⊥	SCDISCINDI	⊤	⊥
OPCTOBSCPR	⊤	⊥	SCDISCINDS	R	⊥
RRLAYCONG	⊤	⊥	TRACE	R	⊥
RRLAYCONRESP	⊤	⊥	FLSETSTORD	W	⊤
RRLAYDISC	⊤	⊤			

Table 15: MSCCO - usage File variables.

Block	*File*	*Shared*	*Forlopp: shared*	*Forlopp: unique*
CHVIEW				
	VIEWFILE			X
	LOGHEADER			X
	LOGPART			X
	PUBLICDATA			X
	SESSION			X
	SAECNTRECORD	X		
LAD				
	COMBANK32			X
	COMBANK64			X
	COMBANK128			X
	COMBANK256			X
	COMBANK512			X
	COMBANK1K			X
	COMBANK2K			X
	COMBANK4K			X
	COMBANK8K			X
	COMBANK16K			X
	COMBANK32K			X
MFM				
	FLREC	X		
	INDREC	X		
MSCCO				
	MSCCODATADATA			X
	TEMPRECORD			X

Table 16: The different types of files in the examined blocks.

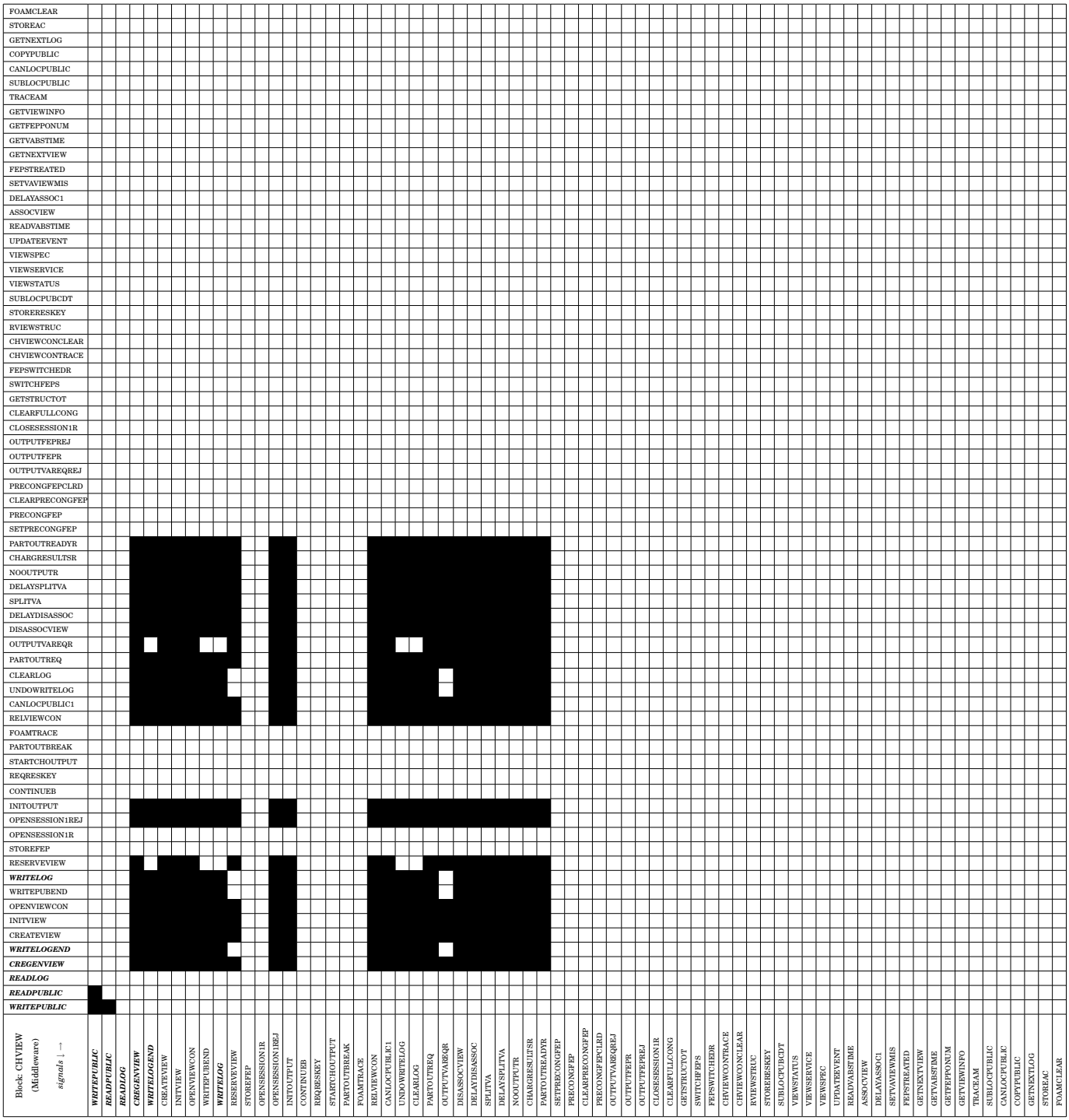

Table 17: CHVIEW - Potential conflicts (marked with a black box) caused by possible simultaneous access to common variables. Derived from Table 7.

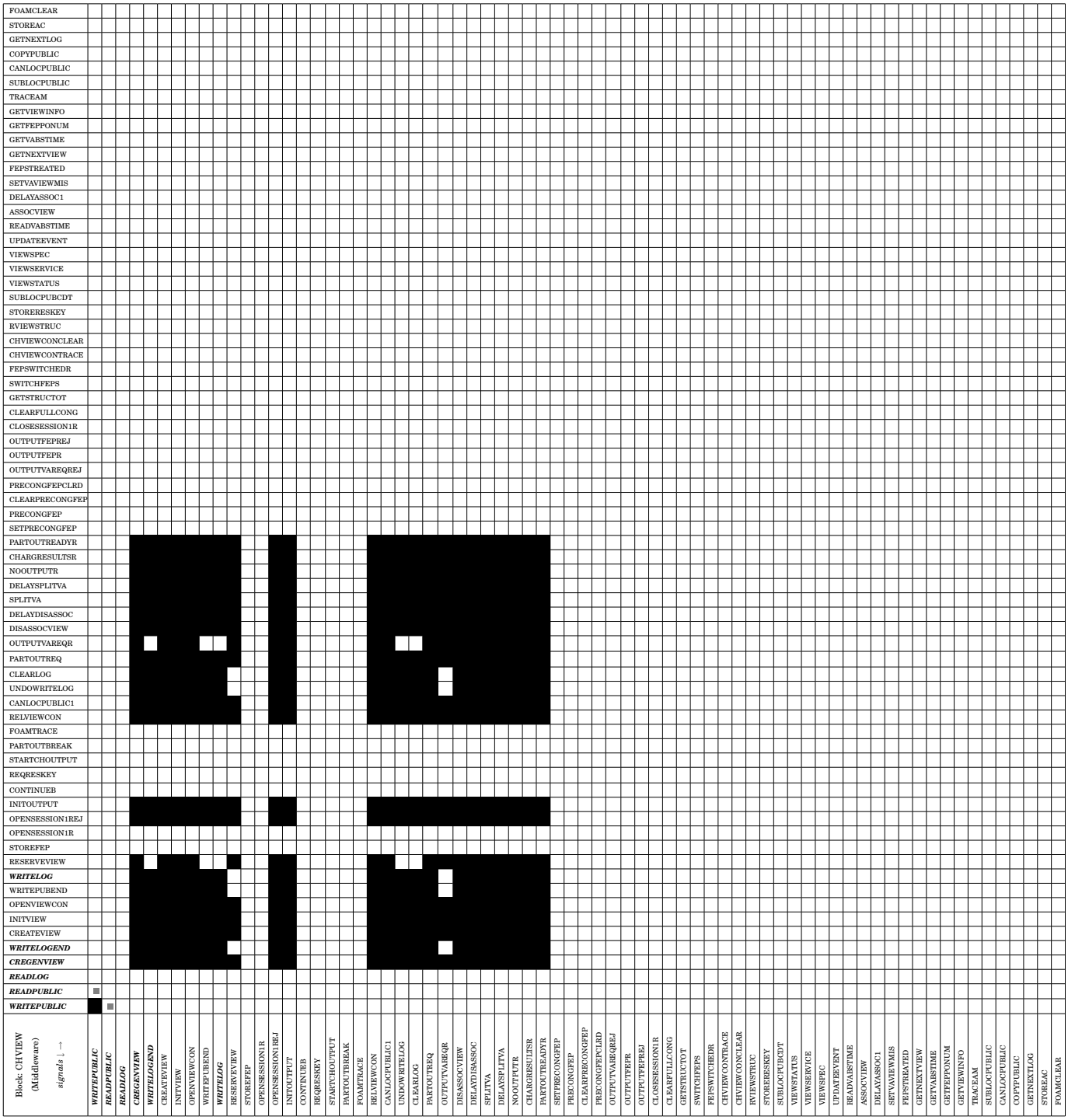

Table 18: CHVIEW - 'w-optimization' applied on Table 17. Eliminated conflicts are marked with a gray box.

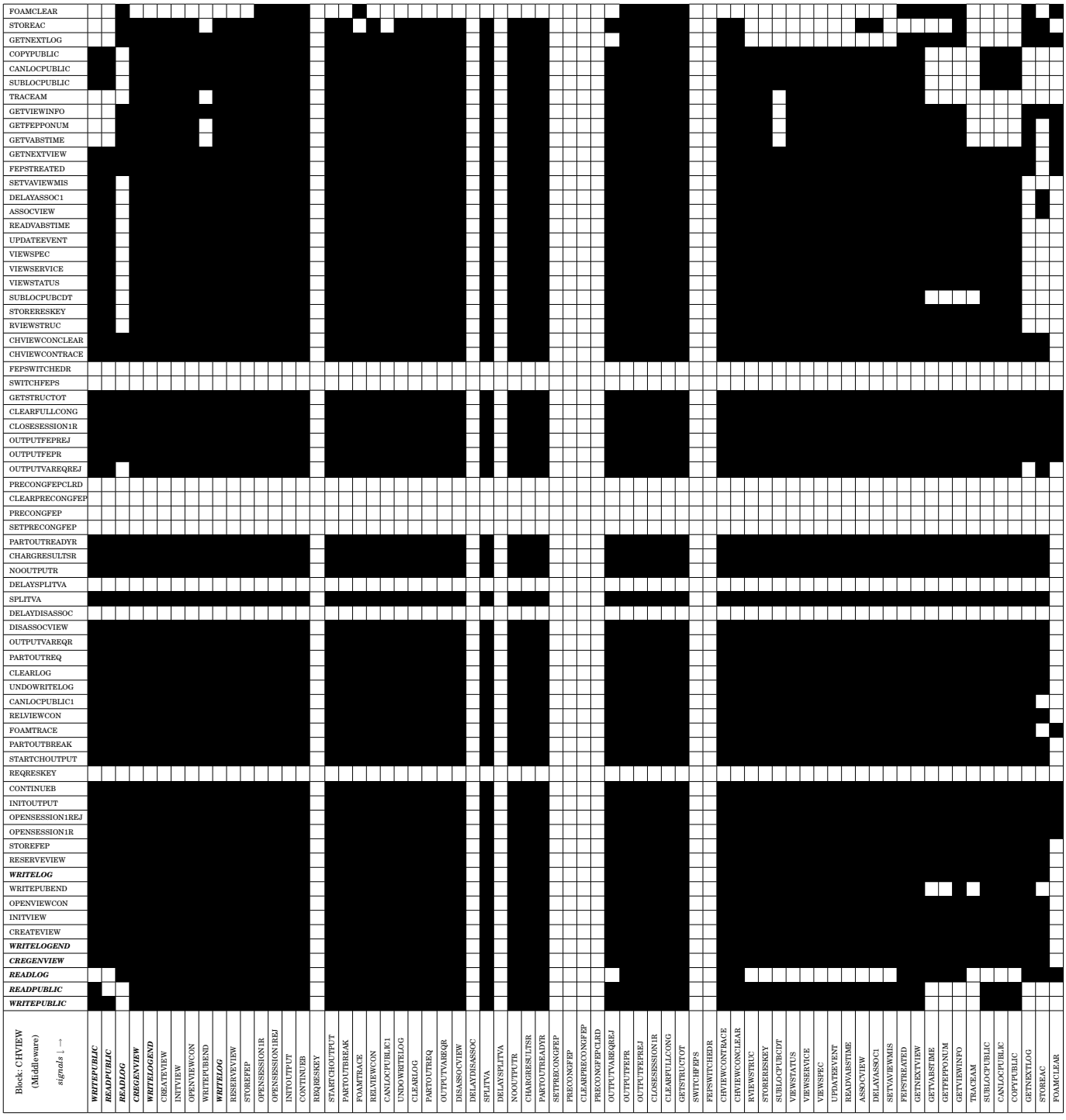

Table 19: CHVIEW - Potential conflicts in the file variables.

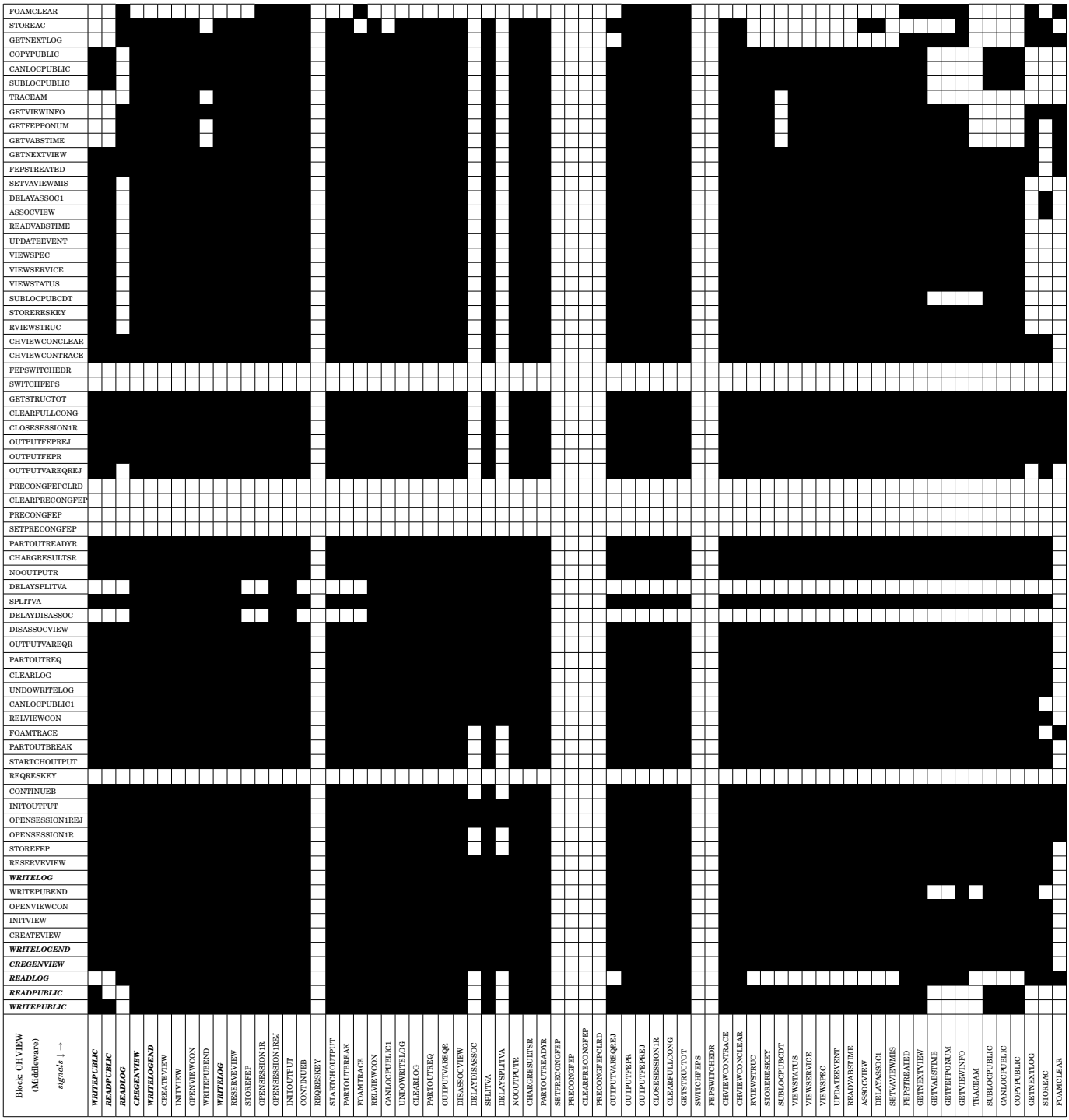

Table 20: CHVIEW - A safe upper bound of the number of conflicts achieved by combining Table 18 and 19.

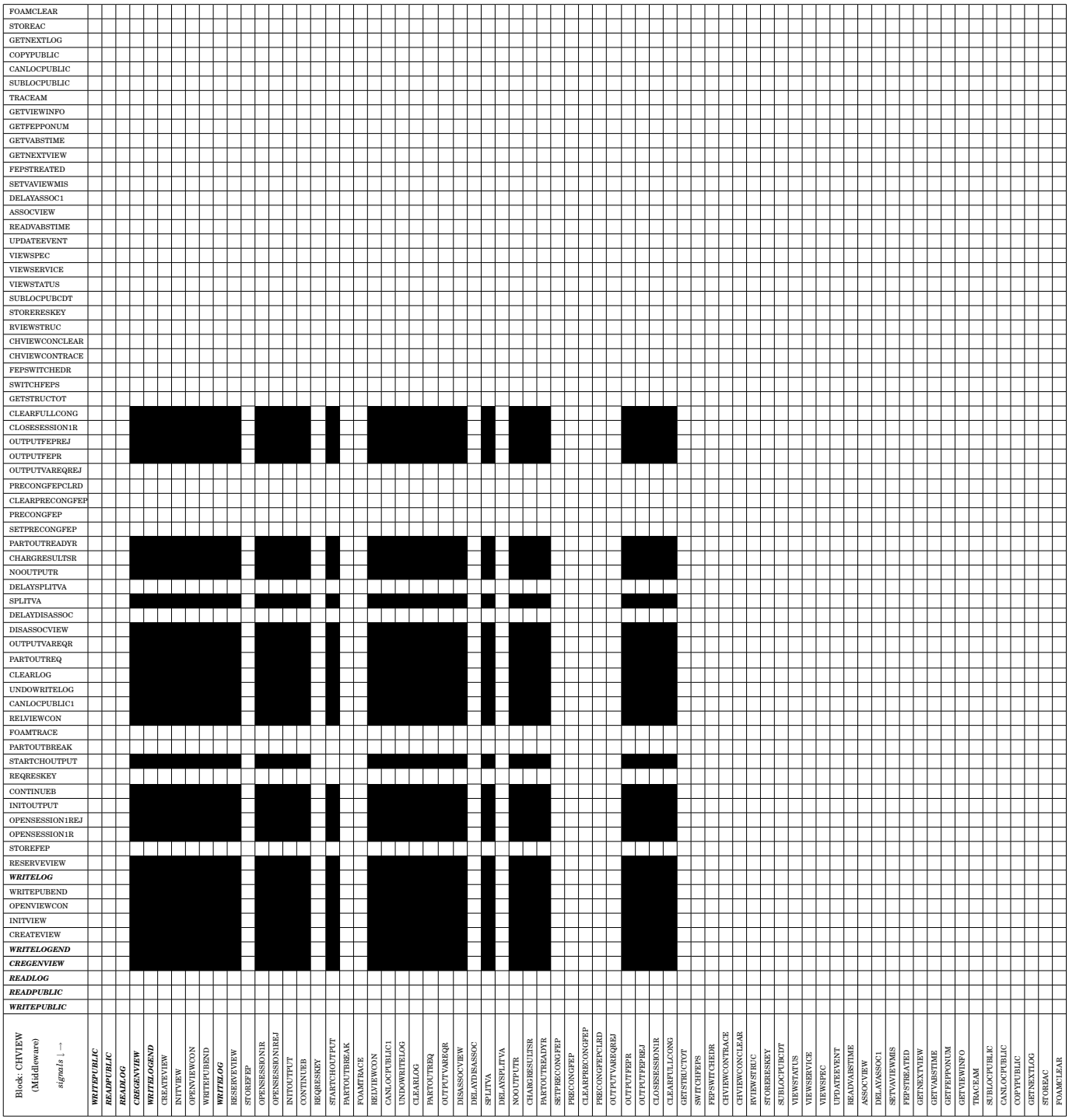

Table 21: CHVIEW - A tightened upper bound is achieved by assuming a protected SEIZE operation.

Block: LAD (OS) *signals* ↓ →	ALLBUF	GETBUFABSADR	READBSIZE	***ALLCOMBUF***	CBFLCONNECT	GETCOMBUFREF	***RELCOMBUF***	READCBSTATE
ALLBUF		■	■	■	■			
GETBUFABSADR	■		■	■	■			
READBSIZE	■	■		■	■			
ALLCOMBUF	■	■	■		■	■	■	
CBFLCONNECT	■	■	■	■		■	■	
GETCOMBUFREF				■	■		■	
RELCOMBUF				■	■	■		
READCBSTATE								

Table 22: LAD - Potential conflicts in the common variables.

Block: LAD (OS) *signals* ↓ →	ALLBUF	GETBUFABSADR	READBSIZE	***ALLCOMBUF***	CBFLCONNECT	GETCOMBUFREF	***RELCOMBUF***	READCBSTATE
ALLBUF		■	■	■	■			
GETBUFABSADR	■		■	■	■			
READBSIZE	■	■		■	■			
ALLCOMBUF	■	■	■		■	■	■	
CBFLCONNECT	■	■	■	■		■	■	
GETCOMBUFREF				■	■		■	
RELCOMBUF				■	■	■		
READCBSTATE								

Table 23: $WriteBeforeRead$-conflicts removed from Table 22.

Block: LAD (OS) *signals* ↓ →	ALLBUF	GETBUFABSADR	READBSIZE	***ALLCOMBUF***	CBFLCONNECT	GETCOMBUFREF	***RELCOMBUF***	READCBSTATE
ALLBUF								
GETBUFABSADR								
READBSIZE								
ALLCOMBUF				■	■	■	■	■
CBFLCONNECT				■	■		■	■
GETCOMBUFREF				■	■	■		■
RELCOMBUF				■	■	■	■	
READCBSTATE				■	■		■	

Table 24: LAD - Potential conflicts in the file variables.

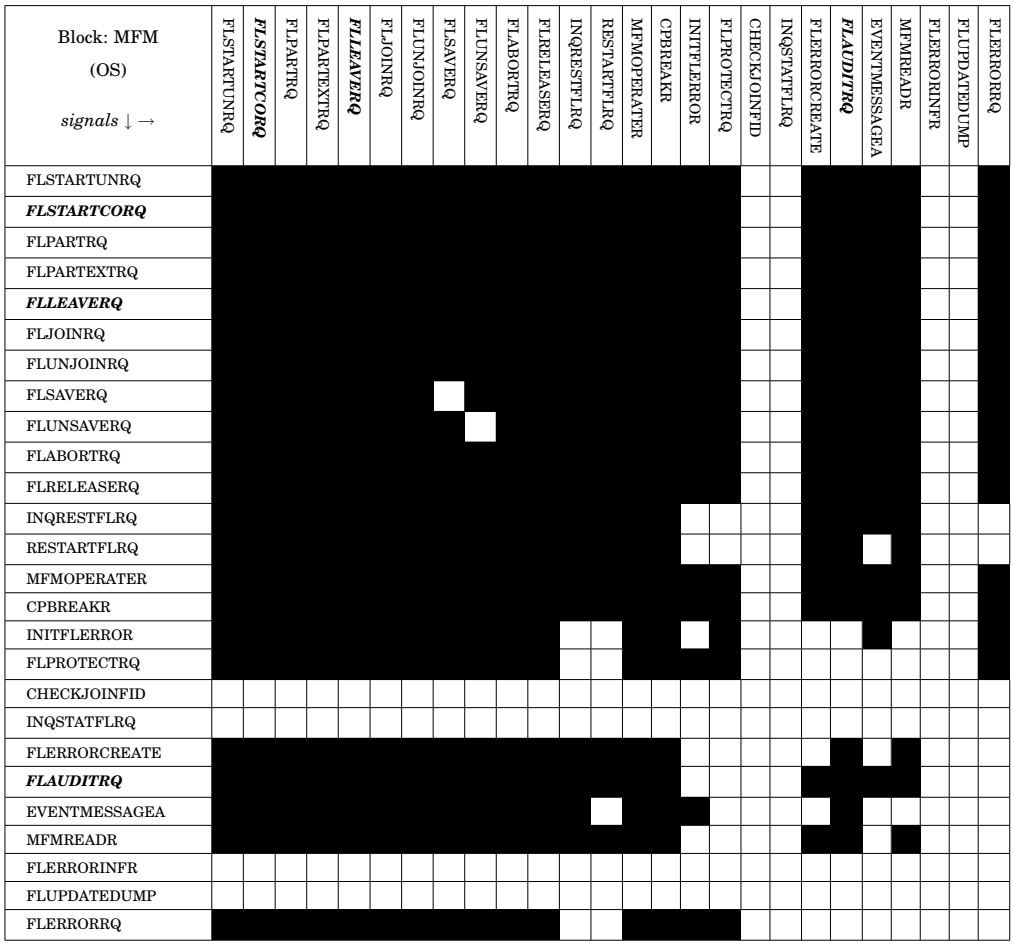

Table 25: MFM - Possible conflicts in the common variables, with/without optimization.

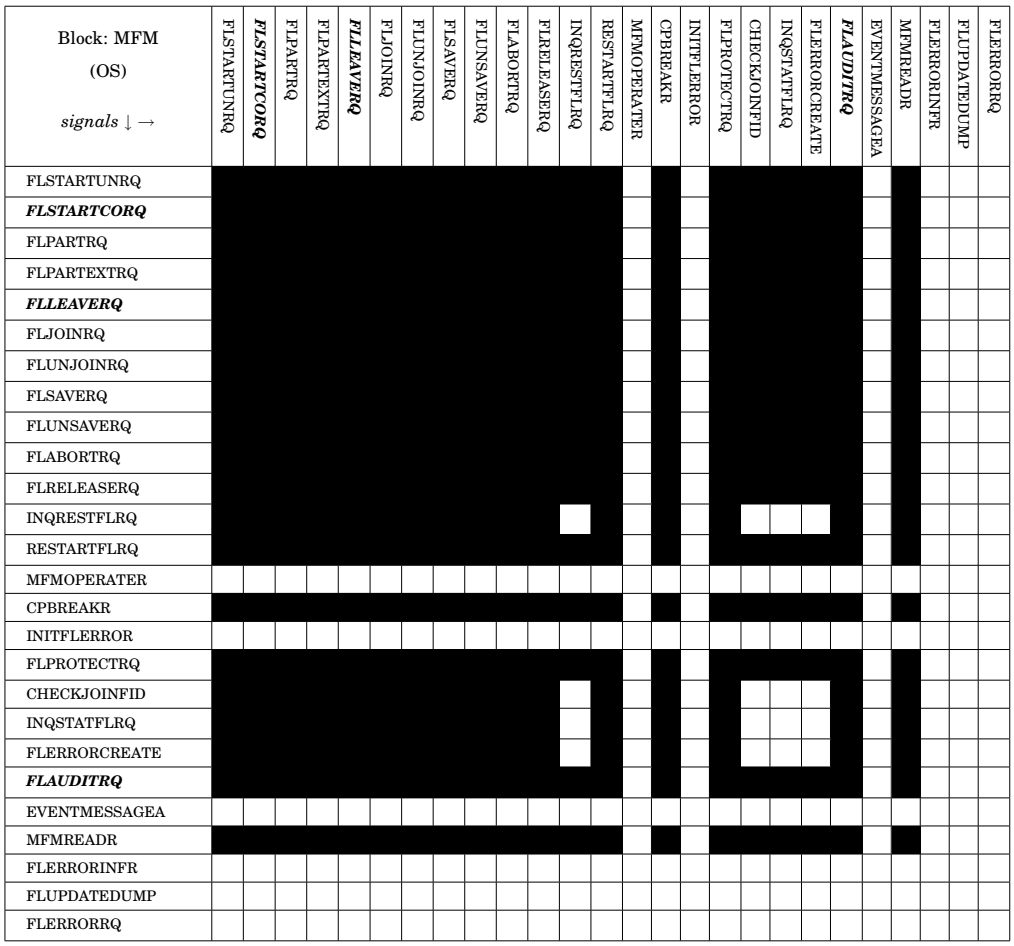

Table 26: MFM - Possible conflicts in the file variables.

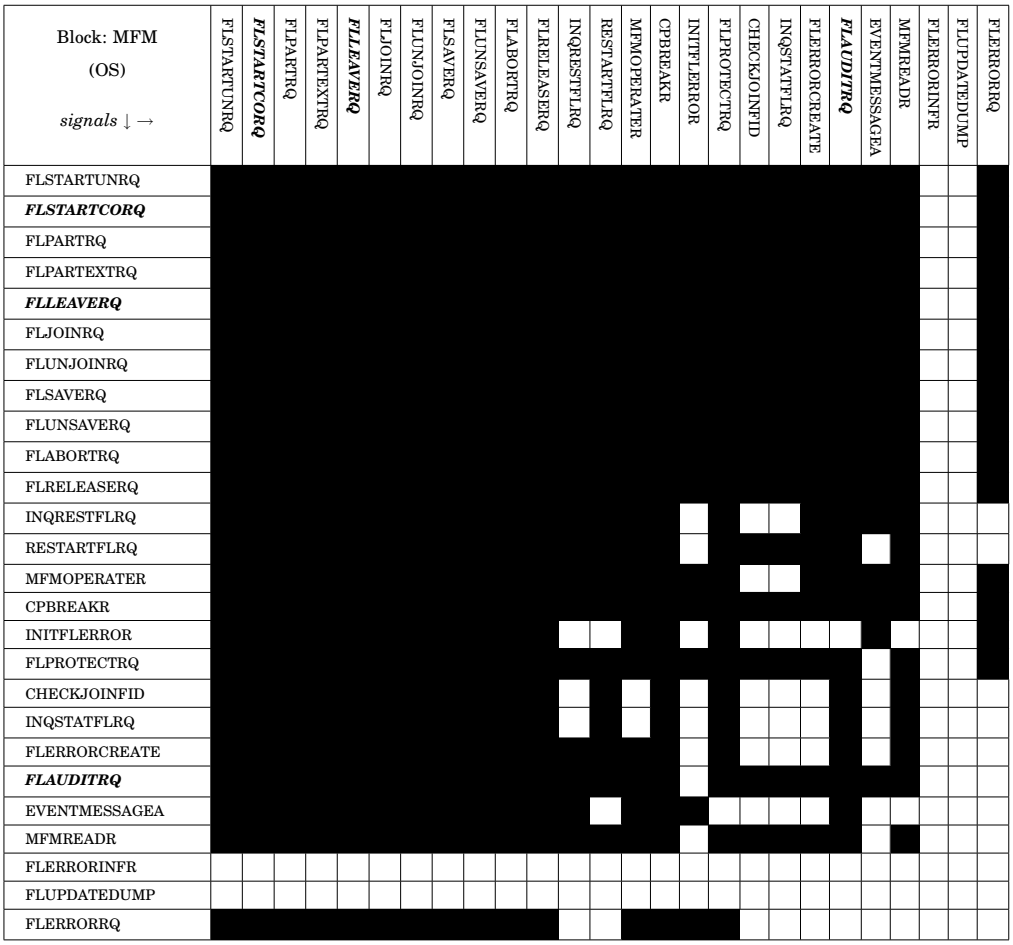

Table 27: MFM - A safe upper bound of the number of conflicts achieved by combining Table 25 and 26.

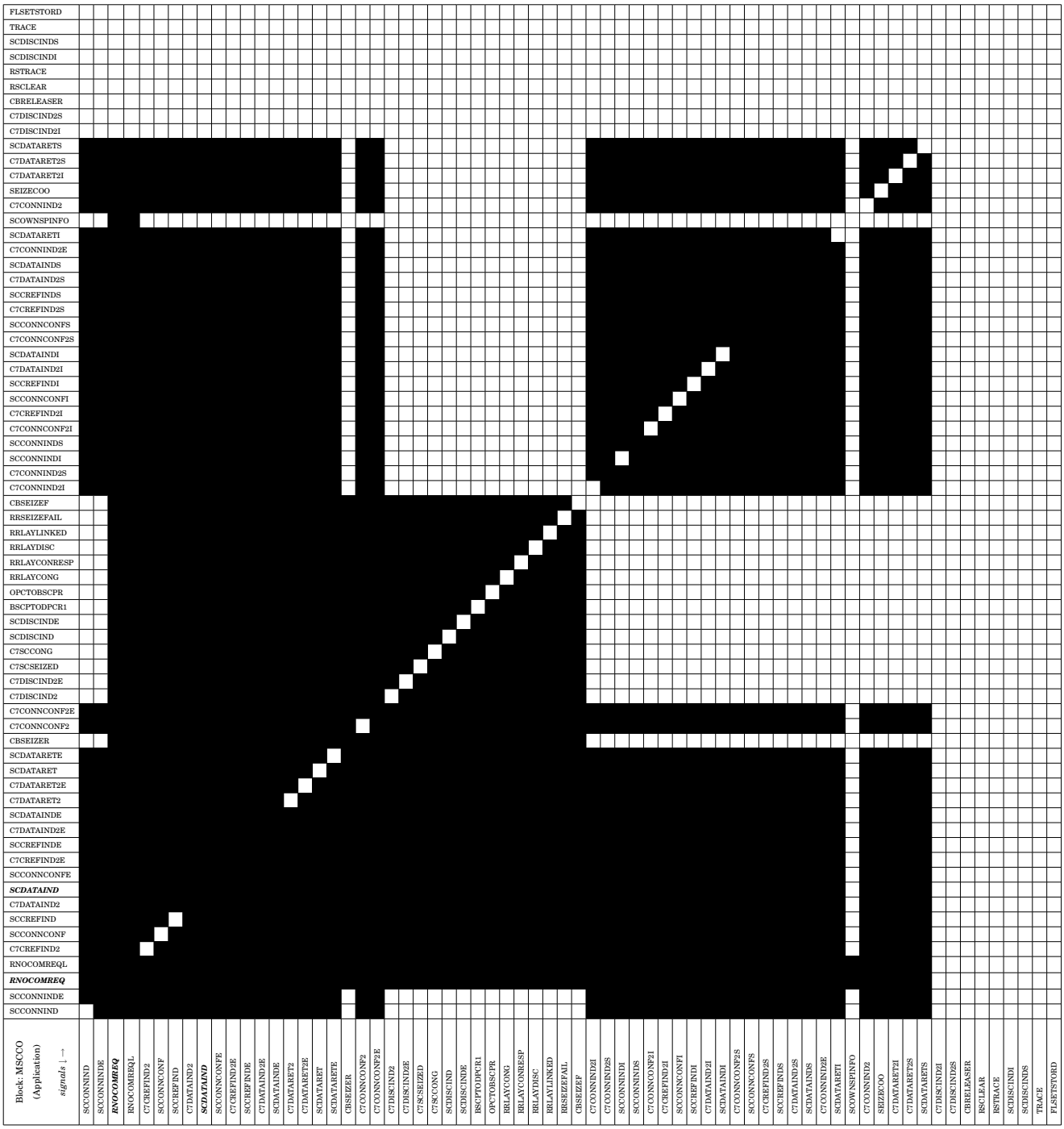

Table 28: MSCCO - Possible conflicts in common variables, with/without $WriteBeforeRead$-elimination applied.

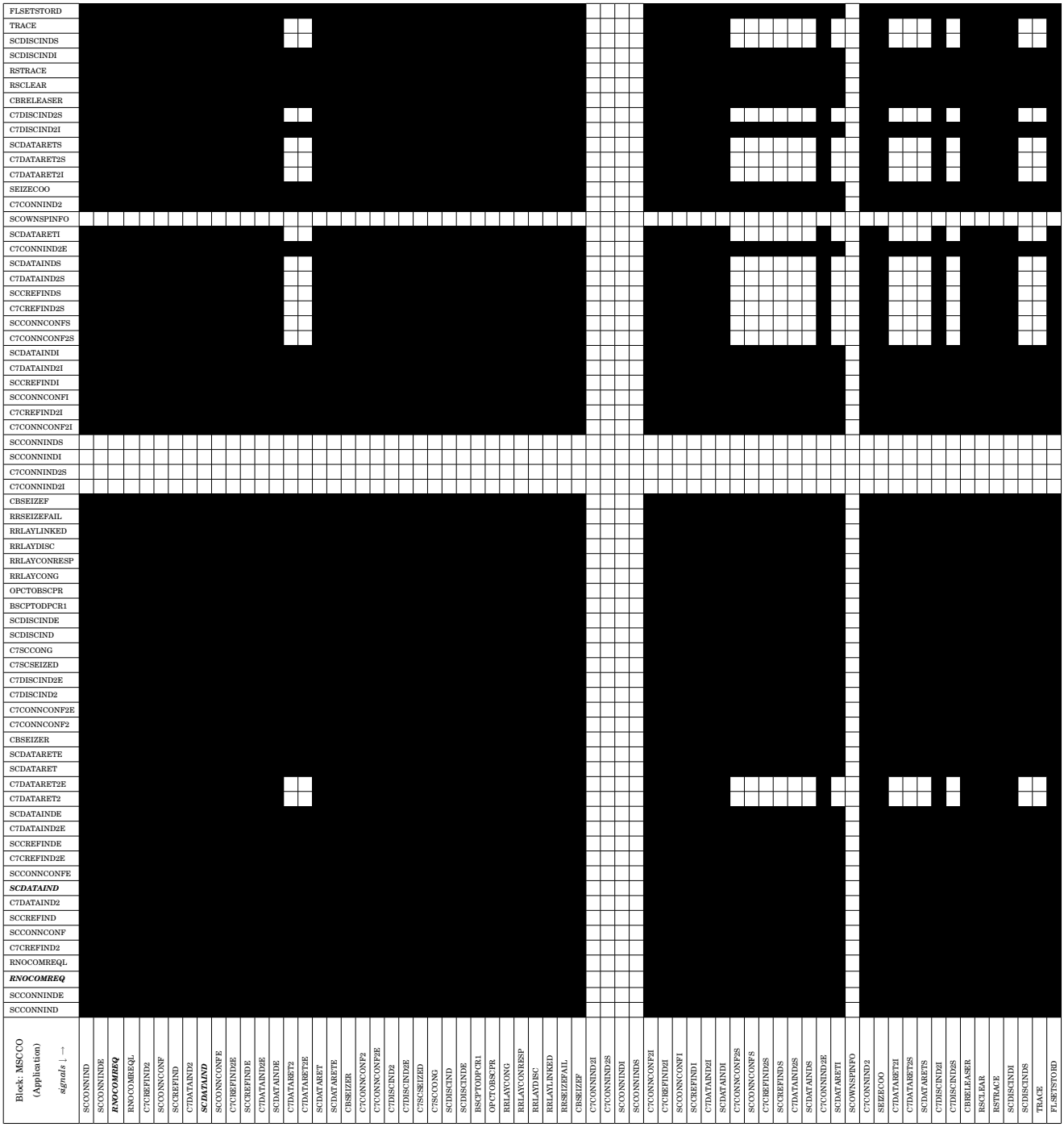

Table 29: MSCCO - Possible conflicts in the file variables.

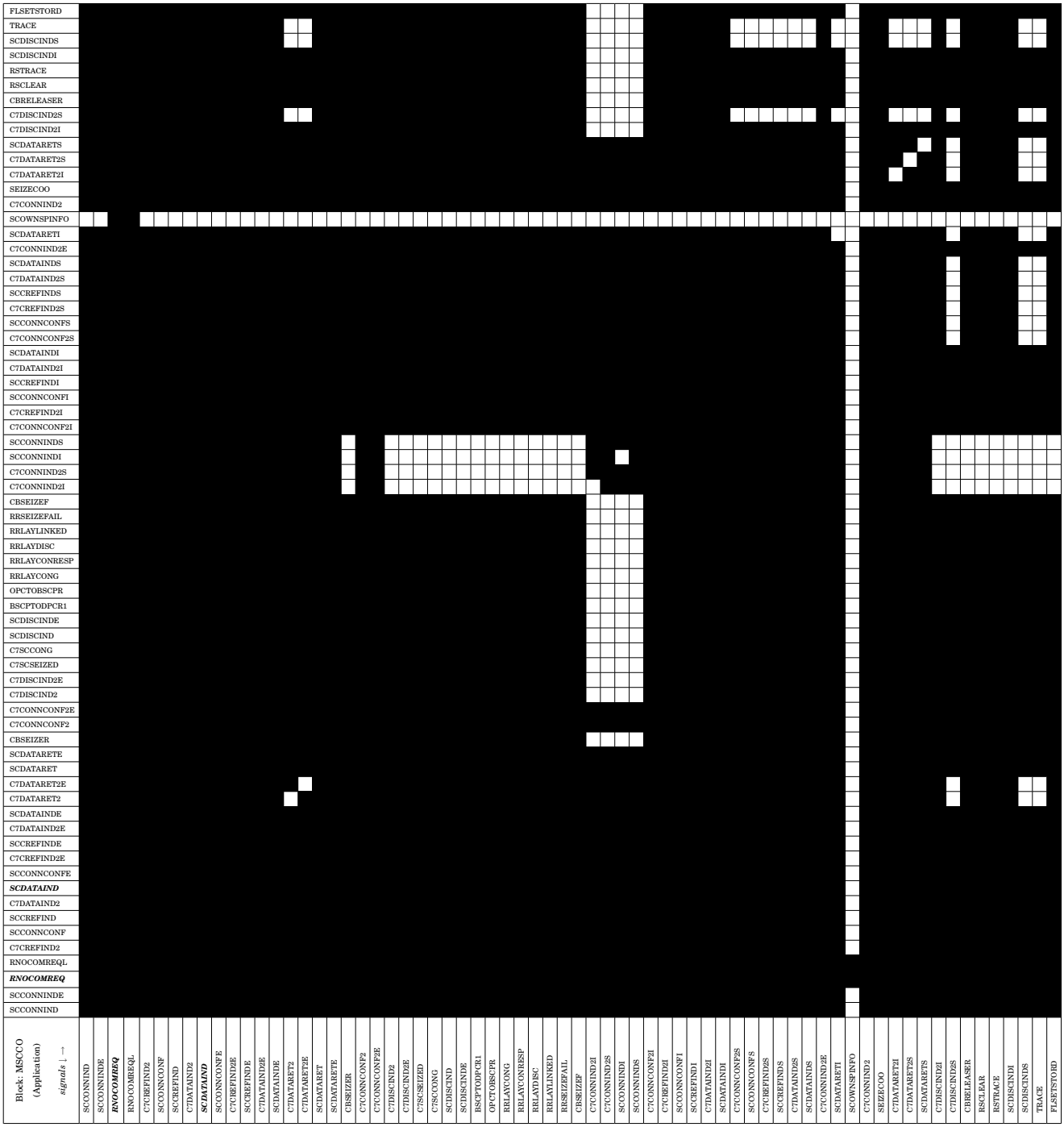

Table 30: MSCCO - A safe upper bound of the number of conflicts achieved by combining Table 28 and 29.