# Dynamic Detection and Mitigation of DMA Races in MPSoCs

Selma Saidi, Yliès Falcone

# Dynamic Detection and Mitigation of DMA Races in MPSoCs

Selma Saidi*, Yliès Falcone†
*Institute of Computer and Network Engineering, Technische Universität Braunschweig, Germany
†Univ. Grenoble Alpes, Inria, LIG, F-38000 Grenoble, France

***Abstract*—Explicitly managed memories have emerged as a good alternative for multicore processors design in order to reduce energy and performance costs. Memory transfers then rely on Direct Memory Access (DMA) engines which provide a hardware support for accelerating data. However, programming explicit data transfers is very challenging for developers who must manually orchestrate data movements through the memory hierarchy. This is in practice very error-prone and can easily lead to memory inconsistency. In this paper, we propose a runtime approach for monitoring DMA races. The monitor acts as a safeguard for programmers and is able to enforce at runtime a correct behavior w.r.t the semantics of the program execution. We validate the approach using traces extracted from industrial benchmarks and executed on the multiprocessor system-on-chip platform STHORM. Our experiments demonstrate that the monitoring algorithm has a low overhead (less than 1.5 KB) of on-chip memory consumption and an overhead of less than 2% of additional execution time.**

## I. Introduction

In recent Multiprocessor Systems-on-Chip (MPSoCs) design, a combination of Scratchpad Memories (SPM) [1] and Direct Memory Access (DMA) engines have been proposed as an alternative to traditional caches, where data (and sometimes code) transfers through the memory hierarchy are *explicitly* managed by the software. This is promising in terms of performance, energy, and silicon area. However the price to pay is clearly programming complexity, since the programmer/software has a disjoint view of the different levels of memories and must manually orchestrate data movements using explicit DMA operations. In this context, *DMA races* emerge as one of the regular issues programmers have to face.

In concurrent programming, *Data racing* refers to the situation where several threads perform *conflicting* accesses (i.e., at least one of them is a write) on a *shared* (global) variable with no proper synchronization between these accesses [2]. In MPSoCs, the notion of *DMA races* is an extension of the classical notion of data races, where two or more DMA requests access the same *memory region* simultaneously and at least one request is a write operation. While data races result from a natural need of parallel programs to access shared data, DMA races usually result from a misuse of DMA operations in a program (e.g., a misplaced DMA-synchronization instruction or a misuse of DMA commands identifying tags). These errors are very common since the programmer is solely responsible of the correct management of DMA operations, which quickly becomes tricky when asynchronous DMA calls are involved.

Detecting data races in multithreaded, lock-based programs, is an extensive area of research, and a lot of techniques have been proposed to detect and prove the absence of races in a program. Some of these tools, such as [3], [4], [5], [6] or more recently [7], rely on formal methods like static analysis, theorem proving and model checking. These techniques suffer from the state-space explosion problem which makes them hardly applicable for complex real size (industrial) applications. In contrast, even though they are not exhaustive, dynamic approaches are lightweight methods that have more precise information available at runtime and can thereby be used to react to errors. Approaches for dynamic data race detection such as Eraser [2], Goldilocks [8] and [9] are proposed to keep track of active locks when accessed.

In this paper, we also propose a dynamic approach, for the detection and correction of DMA races. In the embedded multi-core domain, only few work address the problem of DMA races (cf. [10] and [11]). These approaches statically verify DMA operations, by automatically instrumenting the program using assertions. The instrumented program can then be analyzed using software model-checkers. One of the main drawbacks of these methods is their restriction to the verification of sequential programs (as opposed to concurrent ones). The work in [12] proposes a dynamic approach for DMA race detection as part of a full system simulation, however the overhead of their approach is not described and nothing is performed at runtime for data race correction.

We propose a monitoring algorithm which observes in an *online* fashion the arrival of a sequence of DMA events and emits a verdict each time a DMA race is detected. Moreover, it can *enforce* a correct and a DMA race-free execution of the program. The algorithm acts as a filter that does not allow the execution of a new DMA command that would create a race. The execution of this command is deferred to another point in time when no race is possible. The algorithm therefore changes the order of execution of DMA commands in a program and leads to a rescheduling of DMA requests. The enforced order of execution maintains the causality between *dependent* DMA requests in order to preserve the *memory consistency* of the program. The monitor is implemented in a simulation tool. We use traces extracted from industrial benchmarks and executed on the MPSoC platform STHORM [13]. Our experiments demonstrate that the proposed approach induces a cheap overhead, both in terms of required on-chip memory and execution time delay due to the rescheduling of events.

## II. Runtime Monitoring of DMA Races

### A. Formalizing the Problem of DMA Races

In case of SPMs, the on-chip and off-chip memories are referred to as *one* global address space. A *contiguous memory region* in the global address space is defined by a pair $(\alpha, s)$

where $\alpha$ is the address of the beginning of the region and $s$ is the size of the region. The memory addresses denoted by this region are included in the interval $[\alpha, \alpha + s]$.

When a processor needs data, it issues a transfer command to the DMA. Typically, a DMA command consists of a *source* address, a *destination* address, a block size[1] and a *tag* to identify the command, the DMA then takes charge of the transfer. When the data transfer terminates, the processor is notified of its completion.

*Definition 1 (DMA command):* A DMA command for transferring a *contiguous* block of data is defined as a 4-tuple $(tag, src\alpha, dst\alpha, s)$ where, $tag$ is the identifier of the command, $src\alpha$ is the *source* address, $dst\alpha$ is the *destination* address and $s$ is the *size* of the block to transfer. Note that a DMA command involves two memory regions: $(src\alpha, s)$ in read mode, and $(dst\alpha, s)$ in write mode.

*Definition 2 (DMA event):* At runtime, a (contiguous) DMA command is associated with a pair $(e_s, e_d)$ of *DMA events*: a *start* event $e_s$ that marks the beginning of the execution of the command at some time $t$, and an *end* event $e_d$ that marks the end of the execution of the command at some time $t'$, with $t' > t$. Both $e_s$ and $e_d$ are identified with the *same* tag of the corresponding DMA command.

*Remark 1:* Note that the execution time $t' - t$ of the command depends mainly on the size of the block to transfer along with some architectural hardware parameters (bandwidth, contentions, etc), and can be approximated statically using the DMA performance model defined in [14], [15].

*Definition 3 (DMA Trace):* A *DMA trace* is an *ordered* sequence of DMA events with the following constraints:

a) For each DMA command $i$, the start event precedes the end event, that is: $e_s(i) \leq_p e_d(i)$, where $\leq_p$ denotes the precedence relation between events.
b) A trace can contain two DMA commands with the same tag, however they cannot be executed concurrently. That is, if the trace contains the events $(e_{s1}, e_{d1})$ and $(e_{s2}, e_{d2})$ of two DMA commands with the same tag $i$ then, the termination of one should precede the start of the other, i.e., $e_{d1}(i) \leq_p e_{s2}(i)$ or $e_{d2}(i) \leq_p e_{s1}(i)$.

A DMA race occurs at runtime when at least two DMA commands operate on the *same* memory region at the *same* time, and at least one of the commands operates in write mode on the memory region. In the following, we assume a DMA trace as per Definition 3.

*Definition 4 (Overlap in space):* Consider two memory regions $r = (\alpha, s)$ and $r' = (\alpha', s')$. Regions $r$ and $r'$ are said to *overlap in space*, noted $\text{overlap}(r, r')$, when:

$$(\alpha \leq \alpha' < \alpha + s) \vee (\alpha' \leq \alpha < \alpha' + s').$$

Their overlapping is another memory region, noted $\text{overlapping}(r, r')$, and is defined as:

$$\begin{cases} (\alpha', \alpha + s - \alpha') & \text{if } \alpha \leq \alpha', \\ (\alpha, \alpha' + s' - \alpha) & \text{otherwise.} \end{cases}$$

[1]The block size refers to the quantity of data (in words/bytes) to be transferred.

*Definition 5 (Overlap in time):* Two DMA commands identified by $tag$ and $tag'$ are said to be *executed concurrently*, noted $\text{concur}(tag, tag')$ if:

$$e_s(tag) \leq_p e_s(tag') \leq_p e_d(tag) \vee e_s(tag') \leq_p e_d(tag) \leq_p e_d(tag').$$

*Definition 6 (DMA race):* Two DMA commands $(tag, src\alpha, dst\alpha, s)$ and $(tag', src\alpha', dst\alpha', s')$ are *racing* if they are executed concurrently in the sense of Definition 5 and if two distinct memory regions involved in these commands overlap in space:

$$\text{concur}(tag, tag') \wedge \begin{pmatrix} & \text{overlap}\left((src\alpha, s), (dst\alpha', s')\right) \\ \vee & \text{overlap}\left((dst\alpha, s), (src\alpha', s')\right) \\ \vee & \text{overlap}\left((dst\alpha, s), (dst\alpha', s')\right) \end{pmatrix}.$$

### B. Algorithm for Monitoring DMA Races

The proposed algorithm observes the arrival of DMA events and acts as a filter. When a new DMA command arrives and it is racing with another active command, the new command is not executed and is stored, to be released later when it is not racing anymore with any active command. Subsequent DMA commands that have a dependency with a currently suspended DMA command must also be suspended (even if their execution does not induce a race with active commands).

Consider the scenario depicted in Fig. 1 that illustrates the behavior of the proposed monitor over the execution of 3 DMA requests. When the DMA request $r_1$ is issued, a race is detected with another request (for instance, from another processor) that is currently being executed. Request $r_1$ is therefore suspended. Assuming request $r_3$ depends on $r_1$, when $r_3$ is issued, it must also be suspended in order to preserve the causality between DMA commands defined by the program. When a DMA request $r_2$ is issued, it is not suspended and is executed since it is not racing with any active command nor depends on a suspended command.

We consider 2 forms of dependencies,

- *Address* Dependency: All DMA requests involving the same or overlapped address space of a currently-executing DMA request depend on it.
- *Processor* Dependency: All DMA requests issued from a processor are considered to be dependent on a currently-racing DMA request from the *same* processor. In this case, the dependency order follows the processor sequential order in which the requests are issued.

The Processor Dependency constraint applies in addition to the Address Dependency constraint in order to guarantee a consistent execution of the program. Address Dependency serializes the DMA read/write order operations. Processor Dependency serializes the execution of DMA commands issued from the same processor to follow a sequential execution. More generally, the constraint induced by DMA Dependency can be formalized as follows. When a DMA request $r_i$ depends on a request $r_j$, written $r_i \rhd r_j$, $r_i$ must be executed before $r_j$. Therefore, our algorithm must ensure that if $r_i \rhd r_j$, then $e_d(r_i) \leq_p e_s(r_j)$ should hold in the output trace.

The proposed monitoring algorithm is described in the sequel. It is triggered each time a DMA start or end event is observed.

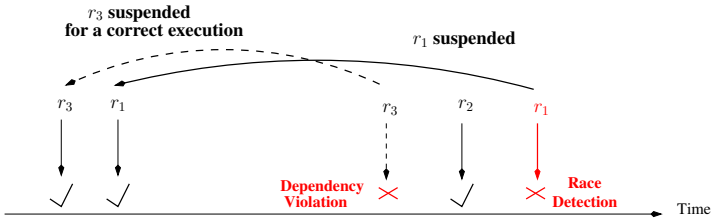



Fig. 1: $r_1$ is suspended due to a DMA race detection, $r_3$ is also suspended due to a dependency violation.

*a) In case of a start event (lines 6 to 17).*

The algorithm first determines the possible races with active DMA commands through the check_race function (line 6). It checks whether the arriving DMA command is racing (in the sense of Definition 6) with any active DMA command and returns the list $rl$ of racing DMA commands tags. If a race is detected (line 15), the execution of this command is deferred and it is added to the list of *suspended* DMA commands along with its associated list of racing commands. If there is no race with any active commands (line 8), the algorithm checks through the check_dependency function if the current/arriving DMA command $r_c$ depends on a currently suspended DMA command $r_s$ and returns the list $depl$ of depending DMA commands tags. If such a dependency exists, then the current command is also suspended to be released later after the execution of the DMA requests in $depl$. Otherwise, the new DMA command is added to the list of active commands and is executed (lines 10 and 11).

*b) In case of an end event (lines 19 to 37).*

- The terminating command is removed from the list of active commands. The lists $rl$ and $depl$ of suspended DMA commands are updated to reflect the termination of a racing or a depending DMA command (lines 21 and 22).
- Suspended commands with an empty racing list and an empty dependency list can now be released provided that they do not race with any active command. Therefore the algorithm first computes the set $candidates$ of possible DMA commands to be released (i.e., with an empty racing list and an empty dependency list, line 24). These commands are then removed from the list of suspended DMA commands.
- The algorithm then computes the set $to\_release$ (lines 26–31) of DMA commands. It checks for each DMA command in $candidates$ if it is race-conflicting with any active DMA commands. Commands with no race are added to the set $to\_release$ and can be actually executed. The rest of the commands are stored in the set $tmp\_suspended$.
- DMA commands in the set $tmp\_suspended$ are once again stored in the set of suspended DMA commands, but with an updated/recomputed racing list taking into account the DMA commands to be released. These steps are done at lines 33 to 35.
- DMA commands to be released are finally validated and become active (line 38).

## III. Implementation and Experimental Validation

The monitoring algorithm described in Section II-B has been implemented in a prototype tool. The tool mimics the behavior of a DMA controller. It takes as input an execution trace of DMA commands and applies the algorithm according to the arrival of each DMA start or end event.

In order to provide input traces to the tool, we use some industrial benchmarks of array and image processing, namely increment, the difference between 2 images and FAST, provided by STMicroelectronics as part of STHORM SDK. In order to produce DMA races, we run a mutated version of these benchmarks where errors, such as synchronization or addressing, on DMA operations have been introduced. These errors are commonly reproduced by programmers during software development. Then, we run the applications on the STHORM platform [13] using the cycle-accurate simulation mode. After execution, we extract the DMA events information required as input for the monitoring tool.

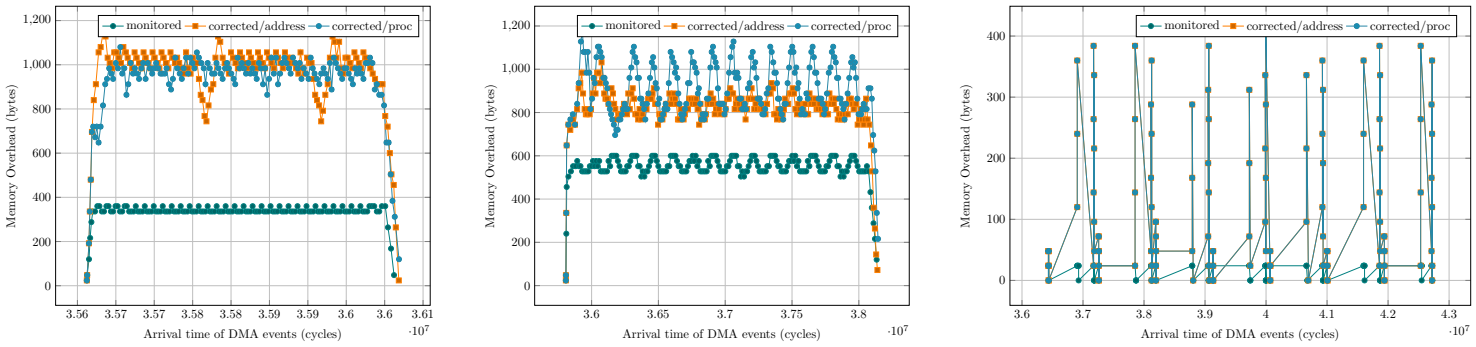
**Monitoring Algorithm for DMA Races**

**Input:** a trace of DMA commands that are possibly racing.
**Output:** a correct and race-free execution trace of DMA commands.

```
1:  active := ∅                                   (* The set of active DMA commands *)
2:  suspended := ∅                                (* The set of suspended DMA commands *)
3:  while not end of trace do
4:      wait e(tag, src, dst, size)               (* Wait for a new DMA event *)
5:      if e is a start event then
6:          rl := check_race((tag, src, dst, size), active)
7:          if rl = ∅ then
8:              depl := check_dependency((tag, src, dst, size), suspended)
9:              if depl = ∅ then
10:                 execute(tag)
11:                 add (active, (tag, src, dst, size))
12:             else                              (* a dependency detected *)
13:                 add (suspended, ((tag, src, dst, size), depl))
14:             end if
15:         else                                  (* a race detected *)
16:             add (suspended, ((tag, src, dst, size), rl))
17:         end if
18:     else                                      (* The DMA event is of kind END *)
19:         remove (active, (tag, src, dst, size))
20:         for all (_, rl) ∈ suspended do (_, depl) ∈ suspended
21:             remove(rl, tag)
22:             remove(depl, tag)
23:         end for     (* Trying now to release some suspended DMA commands *)
24:         candidates := {dma | (dma, ∅) ∈ suspended}
25:         remove(suspended, candidates)
26:         to_release := ∅
27:         for all dma ∈ candidates do
28:             if check_race(dma, to_release ∪ active) = ∅ then
29:                 add(to_release, dma)
30:             end if
31:         end for
32:         tmp_suspended := candidates \ to_release
33:         for all dma ∈ tmp_suspended do
34:             add (suspended, (dma, check_race(dma, to_release)))
35:         end for
36:         output(to_release)
37:         add(active, to_release)
38:     end if
39: end while
```

Figure 2 shows how the size of the memory space required by the tool for storing active and suspended DMA requests evolve with the arrival of start and end DMA events. The arrival of events is clearly seen with raising and falling edges in the graphs. Note that the arrival of events in the input trace follows some regular pattern where DMA events start (ascending edge) and terminate (falling edge) at regular intervals. This is due to the data parallel regular feature of the considered applications. We plot the results for the following,

- *monitored trace*: where the monitor is applied to the input trace only for the detection of errors, without any correction.
- *corrected trace with Address Dependencies*: it is the monitored trace where correction is applied by deferring Address dependent DMA requests.
- *monitored trace with Processor Dependencies*: it is the monitored trace where correction is applied by deferring Processor dependent DMA requests, in addition to Address dependent requests.




(a) Increment: 1024 DMA commands (b) Image difference: 1500 DMA commands (c) FAST: 672 DMA commands.

Fig. 2: Memory and execution time overheads of the monitor when applied to benchmark traces.

*a) Memory Overhead.*

In all benchmarks, the memory overhead, i.e the required on-chip memory to store active and suspended DMA commands, of both detection and correction mode does not exceed 1.5 KB. This is very important to assess because of the very limited available on-chip memory (in STHORM, 16 KB of TCDM are allocated to the DMA controller and a total of 256 KB is shared between 16 cores). Memory overhead does not depend on the *total* number of DMA events during the whole execution, but rather on the number of active (concurrent) DMA requests combined with suspended DMA requests. This number varies during execution and the maximal value is negligible (less than 5%) compared to the trace size. Recall that the number of suspended DMA requests when correction is not considered in null. Therefore, in all benchmarks the memory overhead is larger when correction is considered.

*b) Execution Time Overhead*

The last DMA event marks the total execution of the trace. In the monitored trace, this time is the same total execution time as in the input trace. However, when correction is considered, it is not the case anymore. This is observed on the x-axis where the last event occurs later in the corrected trace than in the monitored one. The difference between the two values represents the execution time overhead induced by rescheduling in the correction. We can clearly see that overhead is very small (less than 2%) with both types of rescheduling (with Address or Processor dependency).

## IV. Conclusion

In this paper, we proposed a monitoring algorithm for the detection and the prevention of DMA races. This allows to build more robust DMA controllers with a better and an automatic control of DMA requests. The algorithm is efficient, and the results obtained experimentally present a good proof of concept that the implemented prototype monitoring tool can be part of a real debugging tool-chain. Indeed, the proposed algorithm has a very cheap memory overhead which mainly depends on the number of active DMA requests because the algorithm runs in an online fashion. The execution time overhead behaves differently according to the traces, however we observe that it is in most cases negligible. This opens an optimistic perspective in being able to avoid dynamically DMA races with almost no delay on the execution time of programs.

*Acknowledgement*

We thank the AST-computing division team at STMicroelectronics, for providing us with the STHORM SDK and the benchmarks for our experiments.

## References

[1] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: design alternative for cache on-chip memory in embedded systems," in *Proc. of the 10th int. symp. on Hardware/software codesign*. ACM, 2002, pp. 73–78.

[2] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: a dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, 1997.

[3] K. Havelund and T. Pressburger, "Model checking Java programs using Java PathFinder," *Inter. Journal on Software Tools for Technology Transfer (STTT)*, vol. 2, no. 4, pp. 366–381, 2000.

[4] D. Aspinall and Jaroslav, "Formalising Java's data race free guarantee," in *In 20th int. conf. on Theorem Proving in Higher Order Logics (TPHOLs)*, vol. 4732, 2007, pp. 22–37.

[5] F. Dabrowski and D. Pichardie, "A certified data race analysis for a Java-like language," in *Proc. of the 22nd int. conf. on Theorem Proving in Higher Order Logics*, ser. TPHOLs '09. Springer-Verlag, 2009, pp. 212–227.

[6] M. Rinard, "Analysis of multithreaded programs," in *Static Analysis*, 2001, pp. 1–19.

[7] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-C: a software analysis perspective," in *Proc. of the 10th int. conf. on Software Engineering and Formal Methods*, ser. SEFM'12. Springer-Verlag, 2012, pp. 233–247.

[8] T. Elmas, S. Qadeer, and S. Tasiran, "Goldilocks: a race and transaction-aware Java runtime." in *PLDI*, J. Ferrante and K. S. McKinley, Eds. ACM, 2007, pp. 245–255.

[9] J. Huang, P. O. Meredith, and G. Rosu, "Maximal sound predictive race detection with control flow abstraction," *SIGPLAN Not.*, vol. 49, no. 6, pp. 337–348, 2014.

[10] A. F. Donaldson, D. Kroening, and P. Rümmer, "Automatic analysis of dma races using model checking and *k*-induction," *Formal Methods in System Design*, vol. 39, no. 1, pp. 83–113, 2011.

[11] A. F. Donaldson, D. Kroening, and P. Rümmer, "Scratch: a tool for automatic analysis of dma races." in *PPOPP*, C. Cascaval and P.-C. Yew, Eds. ACM, 2011, pp. 311–312.

[12] M. Kistler and D. Brokenshire, "Detecting race conditions in asynchronous dma operations with full system simulation," in *Proceedings of the IEEE Inter. Symposium on Performance Analysis of Systems and Software*, ser. ISPASS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 207–215. [Online]. Available: http://dx.doi.org/10.1109/ISPASS.2011.5762737

[13] D. Melpignano, L. Benini, E. Flamand, B. Jego, T. Lepley, G. Haugou, F. Clermidy, and D. Dutoit, "Platform 2012, a many-core computing accelerator for embedded socs: performance evaluation of visual analytics applications." in *DAC*. ACM, 2012, pp. 1137–1142.

[14] S. Saidi, P. Tendulkar, T. Lepley, and O. Maler, "Optimizing explicit data transfers for data parallel applications on the CELL architecture," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 37:1–37:20, 2012.

[15] S. Saidi, P. Tendulkar, T. Lepley, and O. Maler, "Optimal 2d data partitioning for DMA transfers on MPSoCs," in *DSD*. IEEE, 2012, pp. 584–591.