Trace-based Analysis Methodology of Program Flash Contention in Embedded Multicore Systems

Lin Li, Albrecht Mayer
Infineon Technologies AG, Germany
Email: lin.li@infineon.com, albrecht.mayer@infineon.com

***Abstract*—Contention for shared resources is a major performance issue in multicore systems. In embedded multicore microcontrollers, contentions of program flash accesses have a significant performance impact, because the flash access has a large latency compared to a core clock cycle. Therefore, the detection and analysis of program flash contentions are necessary to remedy this situation. With a lack of existing tools being able to fulfill this task, a novel post-processing analysis methodology is proposed in this paper to acquire the information of program flash contentions in detail based on the non-intrusive trace. This information can be utilized to improve the overall performance and particularly to enhance the real-time performance of specific threads or functions for hard real-time multicore systems.**

I. Introduction

Contention for shared resources degrades the overall performance and real-time performance of embedded multicore systems. When the program flash is accessed simultaneously by several cores, a read operation from one core will be delayed by another core that is currently occupying the program flash. The performance impact of program flash contentions is severe owing to the fact that the read latency of program flash [1] is much larger than the clock cycle of a fast core. For microcontrollers with on-chip flash, such as AURIX TC29 from Infineon [2], the ratio between the flash access latency and the core clock cycle is approximately ten, which is much larger than that for RAM. If two cores access the same program flash simultaneously, one must wait ten additional clocks in the worst case. The waiting time is much longer under the circumstances with more than two cores. This can be even worse for consumer devices with external flash. Although in most cases the average number of flash contentions is largely reduced by a program cache, the contention number could be high for certain functions e.g. functions that are not cached. This is particularly a problem for hard real-time systems when such a function is in a critical timing path. The problem is how the flash contentions and performance impact can be estimated or measured, which is currently not solved with existing tools. Only when the contention details and the performance loss are analyzed in detail, can it be efficiently reduced by solutions such as program code relocation or a change in the scheduling of program execution on the different cores.

In this paper, a novel analysis methodology is proposed to detect program flash contentions and estimate the performance impact in detail. Prior to the analysis, two fundamental requirements must be fulfilled. 1) The methodology is based on the non-intrusive trace in embedded multicore systems; i.e., the trace HW is capable of tracing without any impact on execution and timing. 2) The trace data must incorporate fine grained time stamps for instructions and bus transfers related to the program flash. Based on the trace data and several HW parameters, program flash contentions degrading the performance can be detected by our program flash contention analyzer, as shown in Figure 1. The basic process of the contention analysis is explained as follows.

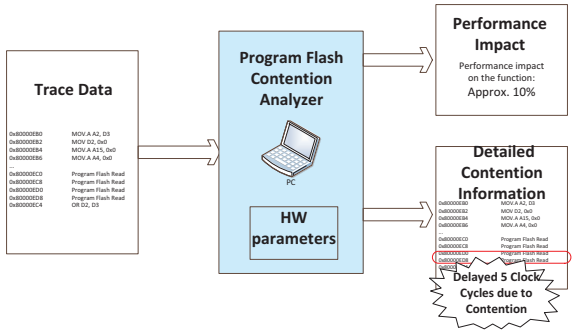

Fig. 1. Overview of the flash contention analysis

After execution, the collected trace data is read into the program flash contention analyzer to find all the spots where the core is idle and waiting for the next instruction, resulting in performance loss. Referring to the trace data and several HW parameters available in data sheets, the analyzer decides for each spot whether the delayed instruction fetch is caused by a contention or other factors. The program flash contention spots are then sorted out. The output information is on the instruction level. This shows the instructions that are fetched late because of contentions. It also presents how many cycles are added for each contention.

The performance loss of specific functions or threads can be easily derived from this detailed contention information. This is important in the real-time field especially for specific real-time critical functions. When known, the program flash contentions can be reduced. For instance, if a system has more than one program flash bank, most of the program flash contentions can be avoided by instruction relocation to a different flash bank. Furthermore, the contention information can also be used as an input to contention-aware schedulers [3] which ensure that conflicting functions are executed at different points in time.

The research leading to these results receives funding from the ARTEMIS Joint Undertaking under grant agreement n° 621429 and from the German BMBF.

A. Contributions

This paper has three contributions:

- To the best of our knowledge, the detection and analysis of program flash contentions are currently missing in the existing tools. This is the first approach to detect program flash contentions based on the non-intrusive trace and also estimate the performance impact of program flash contentions. The analysis results facilitate the performance optimization for multicore embedded systems.
- The proposed approach is demonstrated to have high contention detection rate and good performance impact estimation as shown in section III.
- It provides an example of how to make use of non-intrusive trace to analyze problems automatically for commercial off-the-shelf embedded systems.

B. Related work

Significant research has been done for shared resource contention [3] [4] [5] [6] [7] [8] [9] [10]. The article [4] addresses the shared resource conflicts in SystemC models to accelerate the simulation speeds and provides accurate timing information. Lampka et al. [5] proposed an approach for bounding worst case response time (WCRT) considering share resource contentions especially memory contentions. The research [6] distinguishes and measures the impacts due to sharing of different individual resources e.g. cache, bus and memory, by comparing the runs of different process patterns. Pellizzoni et al. [7] introduced an analysis methodology to compute the upper bounds of the task delay due to data memory contentions. Liu et al. [8] proposed the notion of memory access intensity to facilitate quantitative analysis of program's memory behavior in multicores. In the articles [3] [9] [10] contention-aware schedulers are used to minimize the impact on the performance from shared resource contentions. The research [11] is focused on the performance optimization by considering both data allocation and data cache contentions for Non-Uniform Memory Access (NUMA) systems. The method in the paper [12] copes with the contentions for shared cache and bus with modeling. Those works perform the analysis of the contention problem via simulation [4] [7] [5], static source code analysis, statistical analysis [6] or software instrumentation [3] [8] [9]. However, none of them has used hardware trace, which is non-intrusive and suitable for real-time systems. Most research focuses on the data memory contentions while this paper is addressing program flash contentions, which are difficult to measure but could cause significant performance degradation for hard real-time systems. Program flash contentions are often ignored by SW developers and are complicated by branch predictions. Thus, the measurement is more complex than data contentions. Furthermore, this proposed methodology has an advantage that the analysis can be done with the real hardware operating in its real system. It provides also immediate feedback during on-board debugging, which is convenient for users.

In the following, Section II explains the basic approach and Section III shows the experiment and the assessments. Section IV draws conclusions.

II. Analysis Methodology

A. Basic idea

Ideally, to achieve the maximum throughput, a core should always have instructions running in its pipeline and never wait for the instruction fetch from flash. However, it has to wait in reality due to the flash read latency, flash contentions and other factors. In this paper, this phenomenon is named ***Delayed Instruction Fetch (DIF)***. The basic idea of the proposed methodology is that even though the contentions cannot be traced directly with the existing commercial trace hardware, they have special symptoms and these symptoms can be detected by the designated method based on trace data.

B. Input: trace data acquired by hardware trace

This methodology is based on trace. Two different kinds of traces including an instruction trace and a bus transfer trace are needed as inputs. Trace hardware should be available in embedded systems, in order to obtain trace data non-intrusively. An example of a commercial SoC Infineon AURIX TC29 [2] is given to explain both contentions and the trace, as shown in Figure 2. The decoded trace data consists of information including time stamps (TimeR), observation points (Opoint), origins, addresses and instructions as shown in TABLE I and TABLE II. A time stamp records the time when the operation is finished and traced. The unit of a time stamp is the clock cycle of the trace hardware. In this example, the clock frequency of the trace hardware equals to the core clock frequency. Time stamps facilitate the reconstruction of the traces from different modules in a system. The location where the message is traced is defined as an observation point. For example, the Opoint is program flash bank 0 (PF0) where the accesses from CPU0 to PF0 are observed in TABLE II. The origin column displays the origin of the operation, for instance the master in a bus transfer. Moreover, the address in TABLE I indicates the instruction address in the program flash. In the instruction column in TABLE I, the detailed instructions are displayed.

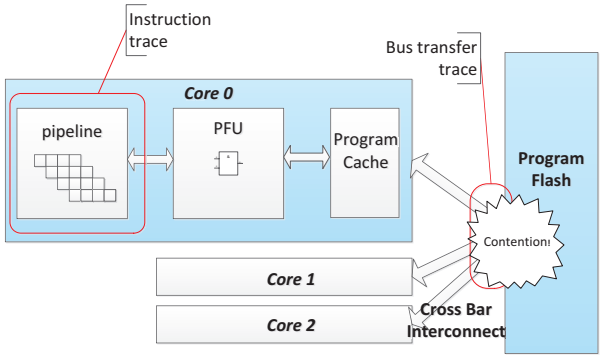

Fig. 2. Example: An instruction trace and a bus transfer trace in AURIX TC29x [2]

TABLE I. AN INSTRUCTION TRACE

Index	*TimeR*	*Opoint*	*Origin*	*Address*	*Instruction*
0	19	CPU0	CPU0	80000A48	MOVH.A A15,xF0030000
1	20	CPU0	CPU0	80000A4C	LEA A15, [A15], 0x6100
2	20	CPU0	CPU0	80000A50	LD.W D15, [A15], 0x0
3	25	CPU0	CPU0	80000A52	MOVH D2, 0x70020000

TABLE II. A BUS TRANSFER TRACE

Index	*TimeR*	*Opoint*	*Origin*	*Address*
0	59516	PF0	CPU0.PMI	800006A0
1	59517	PF0	CPU0.PMI	800006A8
2	59518	PF0	CPU0.PMI	800006B0
3	59519	PF0	CPU0.PMI	800006B8
4	59526	PF0	CPU1.PMI	80002AE8

C. Analysis: contention symptoms

1) Causal Delayed Instruction Fetch: The trace includes executed instructions and bus transfers. It is arranged according to time stamps. The trace provides the analysis basis for the program flash contention analysis.

Not all the contentions on the flash interface have an influence on performance. Normally PreFetch Unit (PFU) is implemented to ensure a core's continuous execution. As the name indicates, PFU inside a core prefetches the instructions from either program flash or RAM. Instruction prefetch operations from PFU could be earlier in advance than needed. In this case, the system has a large tolerance and is not very sensitive to contentions. In spite of a slight contention, an instruction still arrives before it is needed by the core pipeline shown in Fig. 2. This type of contentions is not considered, because the goal of this methodology is to estimate the performance related contentions.

For potentially performance related *DIFs*, we introduce a term ***Causal DIF (CDIF)***. *CDIF* means that a core finishes the execution of current instruction and waits for the next instruction fetch. Ideally, this *CDIF* should not exist because the PFU in principle guarantees that the code is always prefetched and available before its execution. In the real trace, *CDIF* is detectable and is indicated by a certain pattern, which is an instruction fetch closely followed by the execution of the fetched instruction. For example, an instruction fetch operation happens just before an instruction execution. The bus transfer address and the instruction address are identical, indicating a *CDIF*. The interesting set caused only by ***Program Flash Contentions (PFC)*** is a subset of *CDIF*. In order to calculate *PFC*, all *CDIFs* are measured and then a set of ***Other Delay Factors (ODF)*** is subtracted.

2) Other delay factors: There are various factors that can cause *CDIFs* including wrong branch predictions by PFU, reaching the maximum throughput limit of the program flash and instructions that cannot be predicted by PFU. The PFU in a core is responsible for queuing the instructions in the program order to assure the continuous operation of the core. In order to fetch the instruction needed in advance, a branch prediction is required in PFU. There are two possibilities, correct branch predictions and wrong branch predictions. For a correct prediction, a core is supposed to run without waiting for an instruction fetch. On the contrary, a wrong branch prediction will result in a wrong instruction fetch and a core idle state waiting for the instruction fetch.

Wrong branch predictions can be analyzed in two different ways. Firstly, a wrong branch prediction can be detected by checking the branch instruction for a PFU with a static branch prediction algorithm. The static branch prediction algorithm is the simplest branch prediction algorithm and it does only rely on the branch instruction. In this way, the static branch prediction algorithm is an input for the methodology. The second way is to check whether the fetched instructions are executed in the next cycles by looking into the following trace data. For example, if an instruction fetch is observed in the bus transfer trace, but the fetched instruction is not executed in the next 20 instructions, then this fetch can be marked as wrongly fetched. By either of two ways, the wrong branch prediction cases can be collected in the set indicated by ***Wrong Branch Prediction (WBP)***.

The designed throughput of program flash usually satisfies the throughput requirement. However, instructions are read from flash with a fixed sized block, e.g. AURIX TC29x reads 32 bytes in one flash read. Therefore, the maximum throughput limit of program flash can be reached when many jump instructions occur and cause several flash reads in a short period of time. In this case, *CDIFs* can happen even without program flash contentions. These cases can be recognized by calculating the throughput of the program flash in a specific period. In the bus transfer trace, it can be calculated by measuring the time interval between two instruction fetches. The interval reaches the minimum interval defined by HW, meaning maximum throughput reached. This set is denoted by ***OverLoad(OL)***.

Another limitation of the PFU is that for some branch instructions it is not predictable, e.g. indirect branch instructions, in which the jump destination address depends on the register value and they cause *CDIFs* similar to the wrong branch prediction. The list of unpredictable branch instructions depends on the design of the specific PFU. The consequence of the unpredictable instructions is that the next instructions may have to be fetched after the execution of the unpredictable instructions, resulting in *CDIFs*. In this paper, the set of *CDIFs* caused by the unpredictable instructions is indicated by ***UnPredictable (UP)***.

The set *ODF* includes all the other delay factors including *WBP*, *OL* and *UP* as shown in 1.

$$ODF = WBP \cup OL \cup UP \tag{1}$$

3) Contentions: There also exists a possibility that a contention happens together with the one of the other delay factors. These cases are able to be analyzed by comparing to the contention-free situation in the systems. For example, an instruction fetch after an unpredictable branch takes much longer than normal, which is a contention indicator. The set ***Contention and Others (CO)*** is defined for the cases with other delay factors together with program flash contentions. When the contention-free response time for specific HW is known, this set can be derived by comparison between the traced delay and response time.

Finally the contention set *PFC* is the intersection between set *CDIF* and complement set *ODF*, and then union the set *CO*, as shown in 2.

$$PFC = (CDIF \cap \overline{ODF}) \cup CO \tag{2}$$

D. Output: contention instructions and performance impact

1) Contention instructions: The output of the proposed methodology includes contention instructions and performance impact. After the analysis, *PFCs* are calculated and marked on the trace. The affected instructions are defined as contention instructions. A contention instruction is an instruction that is delayed by *PFC*. Based on the known contention instructions, functions with contentions are also available given the elf file, which contains symbol and range information.

2) Performance impact: The performance impact is defined as extra clock cycles added by program flash contentions. This is approximately estimated by comparing the contention case to the ideal contention free situation. The extra clock cycles are estimated for each contention instruction after the analysis.

With the extra cycle information of the contention instructions, the performance impact on a specific function or thread can be assessed, which provides the basis to optimize the performance.

III. Experiment

In order to validate the feasibility and the effectiveness of the analysis approach, experiments with third-party test programs were conducted using an Infineon TC29x microcontroller. As discussed above, program flash contentions and performance impact are not able to be measured by the existing tools, thus a controlled experiment is designed. The experiment consists of a control group and several experimental groups. The control group is a contention-free reference test while the experimental groups are with contentions. In this way, the contentions and the performance impact in the experimental groups can be measured by comparing the control group and the experimental groups.

A. Setup

AURIX TC29x Emulation Device (ED) provides an additional trace functionality as a complement to Production Device (PD). It has MultiCore Debug Solution (MCDS) [13] and a 1 MB Emulation MEMory (EMEM) as a trace buffer. It consists of 3 performance cores running at 80 MHz (max. 300MHz) and 4 different flash banks, supporting concurrent operations. In order to observe the flash contentions more frequently, the program cache is disabled for the tested core so that all instructions are fetched from the same program flash bank. Moreover, the cores and program flash are connected via a crossbar with direct connections between all masters and slaves. Thus, the contentions are only at the program flash interface. The AURIX TC29x board is connected to the PC via Device Access Port (DAP) and Device Access Server (DAS) [14]. The operations in the system are selectively traced and compressed by MCDS and stored temporally in EMEM. Then the trace data is transferred to the PC and loaded into the contention analyzer, which is implemented in Java.

In the experimental groups, program flash contentions can be generated in two scenarios. (a) A real multicore program runs on several cores and program flash contentions are also introduced. (b) Artificial program flash load is added to the flash by Direct Memory Access (DMA) in a controlled way as shown in Fig. 3. Then the program flash contentions occur

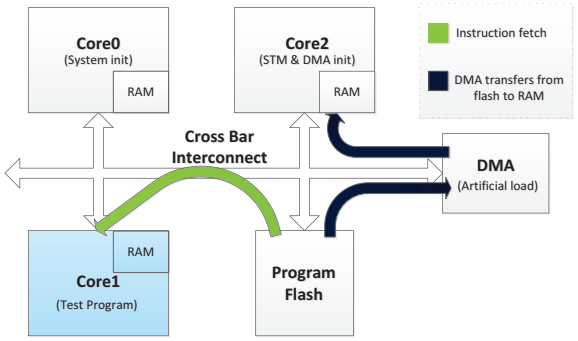

Fig. 3. Experiment setup: The control group has only instruction fetches while the experimental group has both instruction fetches and DMA transfers.

when the DMA accesses are conflicting with the normal instruction fetches by the tested core. Scenario (a) is closer to reality but it has the disadvantages that the generated contentions are not under control and the number of contentions is also difficult to tune compared to scenario (b). In order to estimate the detection rate of this contention analysis methodology with different contention frequencies, scenario (b) is chosen in this experiment.

The added artificial load is generated by a time-triggered DMA that transfers data from program flash to Data Scratch Pad RAM (DSPR) in CPU2. The DMA transfers are requested by System Timer (STM), which sends out a DMA request when the timer reaches a pre-defined value. The pre-defined value is configurable in STM by adjusting the lower compare bits. STM persistently counts the system clock cycles. Each STM request triggers one DMA transfer and a 32-bit-word is read from program flash by DMA. The read address increments one word after each DMA transfer to avoid the "cache" effect in program flash since a small buffer exists in program flash and it buffers the previous read value. With this setup, the program flash is accessed by DMA periodically and the period length is also controlled. This artificial load generation is considered to have no impact on normal operations of the control group except for the program flash contentions as the control group does not use DMA and STM.

Single-core test programs including an internal benchmark and Dhrystone [15] for AURIX are tested in this experiment. The Dhrystone applied here is C 2.1 version. A non-intrusive instruction trace is applied to measure the time information of each instruction, artificial flash accesses and instruction fetches. The extra time consumed in the experimental group compared to the control group is then only caused by the program flash contentions. Then the instructions in the experimental group consuming more cycles than those in the control group are treated as contention instructions.

In this experiment, core 1 is used to run the test program while core 2 is dedicated to initialize STM and DMA. All the initializations are done before the execution of the test program. The frequency of the DMA accesses is decided by a global variable and set during the STM initialization. This variable value can be modified from the PC via On-Chip Debug System (OCDS) after target reset. Therefore different load scenarios can be generated without modifying the binary. It is noted that the artificial load is not generated strictly

periodically because the DMA access could be also disturbed by flash contentions.

To evaluate the methodology accurately and comprehensively, several terms are used and several types of data are measured.

Measured means the value is measured by comparing the experimental groups to the contention-free control group. It shows the actual performance impact and the contention instructions.

Estimated means the value is estimated by the contention analyzer based only on the experimental group.

Load is defined as the DMA read frequency. The average wait cycles (access intervals) are measured and then the load is calculated, which is presented in read/cycle.

Correctly Estimated Contentions (CEC) are defined as the intersection set between measured contention instructions and estimated contention instructions. To show the correctness of the method, detection rate defined the ratio of correct contention estimations to measured contentions, is introduced.

False Positives In this experiment, the overestimated contentions that actually are not contentions.

False Negatives In this paper, the underestimated instructions that indicate no contentions in the estimation but actually have contentions.

Performance Impact is defined as the ratio of additional clock cycles over the measured total clock cycles. For each contention, the extra clock cycles are estimated by the methodology and then added up to have an overall performance impact in the experiment.

B. Results

In order to make the comparison fair, the trace scope is adjusted to the same sequence of the instructions in all tests. The experiments were repeated 5 times and the variances are only several clock cycles. This is because a device reset is performed for each measurement and the identical trace scope is also guaranteed by the trace HW. In the experiment, the measured contention numbers and the estimated contention numbers are recorded. The detection rate is also calculated.

Two experiments were conducted. Experiment 1 is with the internal benchmark and Experiment 2 is with Dhrystone. Experiment 1 result is shown in Table III and Experiment 2 result is shown in Table IV.

The measured trace starts from the reset and ends until the trace buffer is full. As shown in Table III, 14486 instructions in the identical sequence are considered as the measurement scope. For each measurement, the only difference is the artificial load. In Table III, the correctly estimated contentions and wrongly estimated contentions are shown.

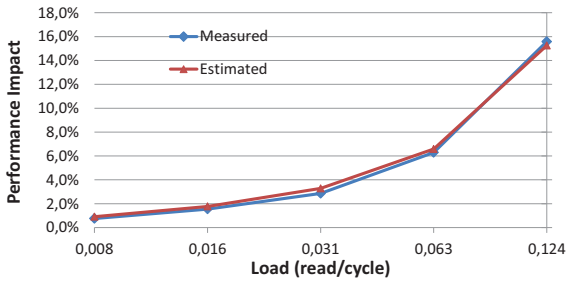

Fig. 4. Performance impact in the internal benchmark experiment

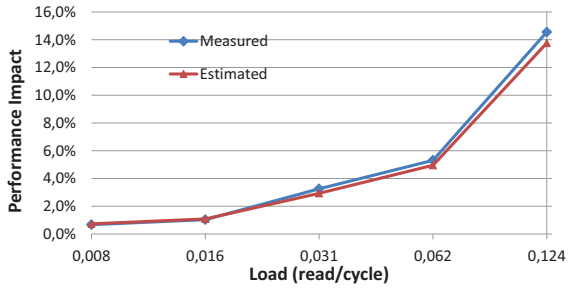

Fig. 5. Performance impact in the Dhrystone experiment

In order to show the effect of the program flash contentions better, the performance impact is also analyzed compared to the ideal scenario, that is without program flash contentions. Both the measured performance impact and the estimated performance impact are shown in Fig. 4.

Experiment 2 is conducted on Dhrystone, which is a commonly used benchmark. The measured results Table IV and Fig. 5 are expressed in the same way as Experiment 1 to facilitate the comparison.

C. Assessments

The load shown in Table III is approximately doubled in every row by using fewer comparison bits for STM. As shown in Table III , the number of contentions increases according to the load. The false negatives are about 10% compared to the measured contentions. The missed contentions by this method are mainly the slight contentions, which are ignored by the analysis and only have a slight impact on the performance. The estimated performance impact has similar trends as the measured performance impact and the impact estimation accuracy is high. This also proves that the performance impact of the contentions is mainly contributed by the contentions

TABLE III. EXPERIMENT RESULTS OF THE INTERNAL BENCHMARK

Load(read/cycle)	Instructions	Measured contentions	Estimated contentions	CEC	False negatives	False positives	Detection rate
0,008	14486	90	89	81	9	8	90,00%
0,016	14486	189	182	170	19	12	89,95%
0,031	14486	366	365	343	23	22	93,72%
0,063	14486	761	751	724	37	27	95,14%
0,124	14486	1708	1675	1645	63	30	96,31%

TABLE IV. EXPERIMENT RESULTS OF DHRYSTONE

Load(read/cycle)	Instructions	Measured contentions	Estimated contentions	CEC	False negatives	False positives	Detection rate
0,008	16367	95	86	74	21	12	77,89%
0,016	16367	158	158	133	25	25	84,18%
0,031	16367	342	330	304	38	26	88,89%
0,062	16367	615	646	592	23	54	96,26%
0,124	16367	1429	1427	1373	56	54	96,08%

except for the slight contentions. The performance hazard is up to 16% in the worst case.

The performance impact error is located in the estimation of the ideal scenario. The ideal scenario can only be coarsely estimated without simulations. The real scenario is usually more complicated than the ideal case. The flash-contention-free scenario could be influenced by other factors such as other contentions, extremely long instruction execution time and pipeline hazard. All these other non-ideal cases could cause errors in the estimation. Even though, in this experiment, the performance impact estimation matches the measured performance impact quite well.

The detection rates in both experiments indicate that the detection rate increases when load is higher. This is also caused by the contentions that impact the performance but do not belong to *CDIF*. This kind of contentions delays the instructions fetch and has hazard to core pipeline, but the instruction arrives before the end of the previous instruction execution. Therefore, it does not belong to *CDIF*. These contentions are more often with lower load and introduce more estimation errors in lower load area. Since the performance impact of this type is low, they can be ignored.

The flash contentions are not only limited to the instruction fetch contentions but also happen to const data fetch. Usually const data is stored together with instructions. This allocation potentially has contentions between instruction and const data or between const data and const data. For the contentions that have impact on the instruction fetch, they could also be analyzed and estimated by this method. The contentions that have no impact on the instruction fetch but have influence on the const data fetch, are not detectable with the proposed method. For example, a const data fetch is delayed due to a simultaneous instruction fetch by another core.

One limitation of this estimation method is that the length of analysis scope is limited by the size of trace buffer in the current implementation. The solutions could be continuous traces or a trace integration strategy which merges several limited traces together to acquire a larger trace scope. Another limitation compared to the SW instrumentation is that this method requires extra HW, which provides non-intrusive traces of several modules.

IV. CONCLUSIONS AND FUTURE WORK

In this paper, we have proposed a general trace-based methodology that allows measurement of program flash contentions directly in a real system without run time impact and artificial reference measurements. The proposed approach helps to find hard-to-detect program flash contentions and achieves reasonable contention analysis results. The performance impact by the contentions is also estimated, which gives a reference to improve the program allocation. This output can be used to accomplish better performance or scheduling especially for hard real-time systems. In the future, the approach could also be implemented in other platforms with enabled program cache to verify this approach in general.

REFERENCES

[1] C. Park, J. Seo, D. Seo, S. Kim, and B. Kim, "Cost-efficient memory architecture design of nand flash memory embedded systems," *Computer Design, 2003. Proceedings. 21st International Conference on*, pp. 474–480, 2003.

[2] Infineon, "AURIX Family-TC29xT," http://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-tm-microcontroller/aurix-tm-family/aurix-tm-family-%E2%80%93-tc29xt/channel.html?channel=db3a304342c787030142dc92c9aa1674, [Online; accessed 20-August-2015].

[3] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1, pp. 129–142, 2010.

[4] K. Lu, D. Muller-Gritschneder, and U. Schlichtmann, "Analytical timing estimation for temporally decoupled tlms considering resource conflicts," *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013*, pp. 1161–1166, 2013.

[5] K. Lampka, G. Giannopoulou, R. Pellizzoni, Z. Wu, and N. Stoimenov, "A formal approach to the WCRT analysis of multicore systems with memory contention under phase-structured task sets," *Real-Time Systems*, vol. 50, no. 5-6, pp. 736–773, 2014.

[6] R. Hood, H. Jin, P. Mehrotra, J. Chang, J. Djomehri, S. Gavali, D. Jespersen, K. Taylor, and R. Biswas, "Performance impact of resource contention in multicore systems," *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1–12, 2010.

[7] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele, "Worst case delay analysis for memory interference in multicore systems," *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*, pp. 741–746, 2010.

[8] L. Liu, Z. Li, and A. H. Sameh, "Analyzing memory access intensity in parallel programs on multicore," *Proceedings of the 22nd annual international conference on Supercomputing*, pp. 359–367, 2008.

[9] S. Blagodurov, S. Zhuravlev, and A. Fedorova, "Contention-aware scheduling on multicore systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 28, no. 4, p. 8, 2010.

[10] F. Pinel, J. E. Pecero, P. Bouvry, and S. U. Khan, "Memory-aware green scheduling on multi-core processors," *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pp. 485–488, 2010.

[11] Z. Majo and T. R. Gross, "Memory management in numa multicore systems: trapped between cache contention and interconnect overhead," *ACM SIGPLAN Notices*, vol. 46, no. 11, pp. 11–20, 2011.

[12] S. Chattopadhyay, A. Roychoudhury, and T. Mitra, "Modeling shared cache and bus in multi-cores for timing analysis," *Proceedings of the 13th International Workshop on Software & Compilers for Embedded Systems*, p. 6, 2010.

[13] A. Mayer, H. Siebert, and K. D. McDonald-Maier, "Debug support, calibration and emulation for multiple processor and powertrain control socs," *Proceedings of the conference on Design, Automation and Test in Europe-Volume 3*, pp. 148–152, 2005.

[14] A. Mayer, "A seamless tool access architecture from esl to end product," *System, Software, SoC and Silicon Debug Conference (S4D)*, 2009.

[15] Wikipedia, "Dhrystone," https://en.wikipedia.org/wiki/Dhrystone, [Online; accessed 20-August-2015].