Data Mining MPSoC Simulation Traces to Identify Concurrent Memory Access Patterns

Sofiane Lagraa[1,2], Alexandre Termier[1] and Frédéric Pétrot[2]
[1]LIG, [2]TIMA, CNRS/Grenoble-INP/UJF

***Abstract*—Due to a growing need for flexibility, massively parallel Multiprocessor SoC (MPSoC) architectures are currently being developed. This leads to the need for parallel software, but poses the problem of the efficient deployment of the software on these architectures. To address this problem, the execution of the parallel program with software traces enabled on the platform and the visualization of these traces to detect irregular timing behavior is the rule. This is error prone as it relies on software logs and human analysis, and requires an existing platform. To overcome these issues and automate the process, we propose the conjoint use of a virtual platform logging at hardware level the memory accesses and of a data-mining approach to automatically report unexpected instructions timings, and the context of occurrence of these instructions. We demonstrate the approach on a multiprocessor platform running a video decoding application.**

I. Introduction

Most integrated devices today include several if not many, processors, and even though they raise predictability issues, come equipped with caches. These MultiProcessor System on Chip (MPSoC) are fully programmable elements, thus, even when confined to a given application domain, they target potentially quite different applications. This trend is likely to continue and increase due to the technological and economical issues of VLSI integration, and several industrial initiatives have started in that direction [7]. Due to either the hardware complexity, which makes the lack of observability an obstacle, or the lack of availability of the hardware, as it is being developed concurrently with the software [12], virtual prototyping is nowadays commonly used to validate the software integration on the hardware before it is actually there. This hardware/software integration consists in writing parallel software and deploying it on the complex and often heterogeneous architectures of the integrated device in an optimized way. However, because of the system final requirements (power, latency, throughput, area, flexibility, time to market), computing and interconnect architectures are very complex, software is organized in stack that can be partly legacy, and includes operating systems or virtual machines. Hardware/software interaction is therefore very complex and often not analysable at design time because of the dynamicity of the current applications and architectures. Given this context, there is a dramatic need for tools that will ease this integration and optimization process. Assuming the use of standard APIs for parallel programming, functional validation can be done on a multiprocessor machine, but spotting the parallel code inefficiencies can only be done with an appropriate virtual platform. In this paper, we focus on the detection of one such inefficiency: concurrent accesses to memory segments leading to high latencies and low throughputs. The memory accesses of a given execution are dumped, producing a huge amount of traces. We then rely on advanced data-mining techniques which allow to automatically identify and report access patterns whose latency deviate significantly from the average behavior of the traces. This method allows to help developers to extract automatically the parts of the traces exhibiting contention and mining them in order to discover both frequent interactions between several processors and the patterns that create this contention.

978-3-9815370-0-0/DATE13/©2013 EDAA

The rest of the paper is organized as follows. Section II briefly presents the background on MPSoC simulation and related works on using data mining techniques for mining traces. Section III formally defines the traces we consider and states our problem of mining "contention patterns". We detail our approach in Section IV. In Section V, we show through an experimental study that our approach helps to discover a complex contention pattern in a video decoding application. Section VI concludes and gives some future research directions.

II. Context and related works

MPSoC become more and more complex according to the evolution of: number of processors, memory architecture, interconnect system. Due to the increase in hardware complexity, monitoring, debugging and optimizing get more difficult. Thus, simulation and tracing tools for MPSoC software development have become a necessity. For simulating single or multi processor architectures running multi-threaded programs, the use of trace simulators offers some significant advantages over execution driven tools. By employing traces, researchers get more flexibility to perform simulation experiments and capture an intrinsic behavior. In [9] the authors present a trace system that consists in tracing hardware events that are produced by instrumented models of multiprocessor platform components. The component models are instrumented in a non-intrusive way so that their behavior in simulation is not modified. Using this trace results allow to run precise analysis of the software that is executed on the platform.

The use of data mining to analyze execution traces is relatively recent, especially in the SoC community. In [5], the authors proposed an approach that can automatically extract

relationship among several signals from simulation traces. The authors experimented it on Advanced Micro-controller Bus Architecture (AMBA). In [11], the authors developed a framework for mining kernel trace data in order to isolate processes responsible for systemic problems that are elusive to standard tools such as `top`. In [20], the authors developed an efficient sequence mining algorithm that can discover short sequences of instruction responsible for inefficiencies. The data mining technique they propose is less robust to noise in the trace, coming from example from different schedulings. In [17], [3], [15], the authors detect congestion patterns, with goals similar to ours. However, their method is limited to detect congestion only in NoC routers. Their work is based on metrics and does not consider execution traces.

The originality of our approach compared to the related works is that first, it targets explicitly MPSoCs and/or multi-core processors and addresses the delicate problem of memory contention. Second, it relies on a completely automatic data mining approach that both finds contention points across multiple cores and presents explicitly what happens frequently at these points. Third, we provide a framework adapted to instruction-level traces (instead of process level, in [11]).

III. Trace definitions

The traces considered in this paper are execution traces of applications running in a simulated MPSoC environment. These traces are collected with a tool presented in [9]. We first give definitions and notations used throughout this paper, and then give more specific details about our traces.

We trace the CPU memory access (fetch, load/store, load-link/store-conditional) that we call events. For each event corresponds a *trace event* $e = (ts, cpuid, latency, s)$. A *trace event* consists of a timestamp $ts \in [0, t_{max}]$, a CPU identifier $cpuid \in \{1, ..., k\}$, a latency $latency \in \mathbb{N}^+$ and $s \subseteq TS$ where TS is a set of trace symbols containing instruction address, data address and memory access type that has been performed by CPU $cpuid$ at time ts with a latency $latency$. The *execution trace* $ET = \{e_1, ..., e_n\}$ is the ordered set of events produced by all the CPUs of the MPSoC. The ordering is on timestamps: for all $i, j \in [1, n]$ with $i < j$ we have $e_i.ts \leq e_j.ts$. We also note $\forall i \in [1, n] \;\; ET[i] = e_i$.

In our case, a trace event has the fixed form represented by the following table:

CPU ID	Cycle Number	Program Counter	Instruction Type	Data Address	Access Latency
1	212305	0x10009d60	fetch	0x10009d60	28

It consists of, in order of occurrence, the global date at which the event occurred in cycles since the power-up of the system, which CPU initiated the transaction, the program counter of the instruction that produced the access, the transaction type which can be instruction fetch, load/store, load-link/store-conditional pairs, and finally the memory access latency by the CPU.

Our goal in this paper is to discover in the trace *contention* between several cores during execution. There is a contention where either a memory unit or one or more links of the MPSoC platform are accessed simultaneously by more processors than they have been designed to satisfy, leading to delays in response time and increasing the memory access latency from the processor to the memory [13]. Due to the exhaustive nature of our traces, the accesses leading to contention are captured in the trace. A *contention pattern* is a set of co-occurring events i.e. whose timestamps differ by at most ω cycles, that are each on a different CPUs that access a similar resource (a memory address), and where at least one of the events exhibits an unusually long latency, indicating contention on the resource. A contention pattern is *frequent* if it occurs in the trace more than a predefined *minimum support threshold* ε times.

Problem statement: Given an execution trace ET and two thresholds ε and ω, our goal is to discover automatically the frequent contention patterns. If the ε threshold is set sufficiently high, the frequency of a contention pattern indicates that it is not a rare and difficult to predict situation, but a misuse of the resources that come from the application design, and that should be fixed to improve overall performances.

IV. Contention pattern discovery

Discovering contention patterns is a multi-step process. As we detect contention through instruction latency value, the first step is to determine what is an *unusually* high latency. This information can of course be given in input, however we show a simple method to determine it semi-automatically. Then the execution trace must be filtered to keep only events that are around high latency events, and this filtered execution trace requires further preprocessing in order to be fed to a pattern mining algorithm that will discover the contention patterns.

A. Long latencies determinations

The latencies will the analysed through simple statistical techniques. Let L denote the list of all non-trivial (i.e. not cache hits) latencies found in the execution trace, whatever the CPU: $L = \{e.latency \mid e \in ET\}$. Without domain knowledge, the basic assumption that we make is that most memory accesses are done without contention. The median of the latency values in L is thus supposed to be representative of the normal access latency. In order to allow for some variations in the latency value, we only consider as unusual latency values that are in the *upper quartile*, i.e. the highest 25% of the latency values. $Q_3(L)$ denotes the lowest latency of the upper quartile. This is a standard statistical way to identify high values in a dataset [14]. Hence for latencies, the set L_H of high latency values contains all latencies above $Q_3(L)$, $L_H = \{l \mid l \in L \wedge l \geq Q_3(L)\}$,

B. Slicing the execution traces into contention windows

By having identified high latencies, the execution trace can be filtered to focus on events having these latencies and their immediate surroundings. The output of this filtering step is a sequence of *contention windows*. A contention window is a slice of an execution trace having a duration ω, and that

contains one or more high latency events. It thus give us the context of occurrence of high latency events.

For constructing the windowed events trace, our solution is presented in Algorithm 1. It receives in input a contention window duration ω, the execution trace ET, and a set of high latency events HL defined by: $HL = \{e \mid e \in ET \wedge e.latency \in L_H\}$. HL is sorted on increasing timestamp in order of events, as well as is ET. The algorithm outputs the set WT of contention windows. It's principle is as follows: for each high latency event $e_H \in HL$ increasing according to the timestamp order (line 3), all the events from the execution trace ET that surround it (at most $-\omega/2$ cycles before or $\omega/2$ cycles after) are inserted into the current window (lines 4-5). If one of these events is a high latency event, it is removed from HL (lines 6-8) to avoid making a near-duplicate window in the next iteration of the for all (line 2). Our algorithm authorizes some overlap between windows, but it will be limited to at most $\omega/2$ cycles for a couple of overlapping windows.

Algorithm 1 Windowed events trace

Require: duration ω, execution trace ET, high latency events HL
Ensure: Windowed trace WT
1: $n \leftarrow 0$
2: **for all** $e_H \in HL$ **do**
3: $WT[n] \leftarrow \emptyset$
4: **while** $ET[i].ts \geq e_H.ts - \omega/2$ AND $ET[i].ts \leq e_H.ts + \omega/2$ **do**
5: $WT[n] \leftarrow WT[n] \cup \{ET[i]\}$
6: **if** $ET[i] \in HL$ **then**
7: $HL \leftarrow HL \setminus \{ET[i]\}$ {*Avoid some overlapping windows*}
8: **end if**
9: $i \leftarrow i + 1$
10: **end while**
11: $n \leftarrow n + 1$
12: **end for**
13: **return** WT

Fig.1 shows an example of the windowed execution trace on 4 CPUs. The window 1 is constructed according to the first high latency event encountered in any of the CPUs. Here its C on CPU_0. The window 1 contains the events of all CPUs occurring from $(-\frac{\omega}{2})$ before C to $(+\frac{\omega}{2})$ after C. Here a second high latency event occurs in this window (event A on CPU_1). The window 2 is constructed according to the high latency event encountered in any of the CPUs after exiting window 1. Here for example lets consider its D on CPU_1. The window 2 contains the events of all CPUs occurring from $(-\frac{\omega}{2})$ before D of CPU_1 to $(+\frac{\omega}{2})$ after D of CPU_1. We also see that window 2 overlaps partially window 1.

C. Mining the frequent contention patterns

There exists many different algorithms for mining patterns in data. In our case, the most important information is the frequent co-occurrence of set of instructions, memory address,

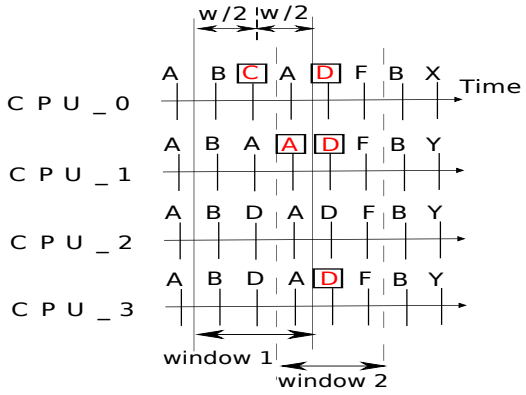

Fig. 1. The windowed events trace

memory access types represented by the repetition of the set of trace symbols over several CPUs. Our assumption is that due to the way contention windows were selected, any set of events appearing frequently in these windows is suspicious and have a high chance to be involved, directly or indirectly, in the contention that drives up the latencies. The data mining technique used to discover such sets of frequently occurring events is called *frequent itemset mining algorithm* [2], [18]. In this technique, the first input is a multiset of *transactions* $D = \{t_1, .., t_p\}$ defined over an alphabet of *items* $\Sigma = \{i_1, .., i_q\}$, where $\forall t_i \in D \quad t_i \subseteq \Sigma$. The second input is a *minimum support threshold* $\varepsilon \in [0, p]$. Frequent itemset mining algorithms then extract all the *frequent itemsets*, i.e. all the *itemsets* $is \subseteq \Sigma$ that appear in more than ε transactions of D. More formally, is must satisfy $support(is) \geq \varepsilon$, where $support(is) = |\{t_i \mid t_i \in D \wedge is \subseteq t_i\}|$.

In order to exploit this technique, we transform the set of windows WT into a set of transactions D. This is presented in Algorithm 2. Each window $w \in WT$ becomes a transaction (lines 2-5), i.e. a set of items. We thus lose the sequencing of events inside a window, for the data mining algorithm all the events of a single window are considered simultaneous. This is not a problem, as inside a window we are interested in the co-occurrence of different events and not in their precise sequencing at the cycle scale.

Algorithm 2 Windowed events transactions

Require: Windowed trace WT
Ensure: Transactions dataset D
1: **for all** $w \in WT$ **do**
2: $D[i] \leftarrow \emptyset$
3: **for all** $e \in w$ **do**
4: $D[i] \leftarrow D[i] \cup \{e.cpuid\_e, e\}$
5: **end for**
6: **end for**
7: **return** D

Pattern mining algorithms are complex algorithms that explore a large combinatorial space i.e the algorithms have

exponential time complexity according to the number of items with the exacts solutions. In order to do so efficiently, they exploit several properties of sets over the alphabet Σ, and of the frequency definition. Changing any of these properties prevents from using the most efficient algorithms. Thus, in order to mine complex data while keeping good scale up properties, a delicate problem is to find an alphabet Σ that allows to find informative patterns while fitting with the pattern mining framework defined above. In our case, the alphabet Σ of transaction items should at least contains all the possible trace symbols TS. But consider the case where two CPUs CPU_1 and CPU_2 make the same memory access, represented by $a \in TS$, in a single contention window. The associated transaction is a set and not a multiset, so it will only be the singleton $\{a\}$. This will not allow to discover any contention: a single CPU issuing access a would have given the same transaction. Our solution is to prefix each trace event with the CPU that issued it: here this will give $\{CPU_1\_a, CPU_2\_a\}$ such as all CPUs of the platform here CPU_1, CPU_2 are in transaction, and it gets possible to find contention patterns. Now consider the case where we have two transactions $t_1 = \{CPU_1\_a, CPU_2\_b\}$ and $t_2 = \{CPU_1\_b, CPU_2\_a\}$. There is no itemset common to both t_1 and t_2, as they contain completely different items: the algorithm cannot see their similarity and does not extract any frequent pattern. Our solution is to keep also the original event with its CPU prefix (line 4). This gives: $t_1 = \{CPU_1\_a, CPU_2\_b, a, b\}$ and $t_2 = \{CPU_1\_b, CPU_2\_a, b, a\}$, with a common pattern being to have simultaneously the events a and b. We thus have $\Sigma = TS \cup (\{CPU_1, ..., CPU_k\} \times TS)$.

Once we have the transactions, we can use a state of the art frequent itemset mining algorithm. We use LCM [18], the most efficient one according to the FIMI contest [1]. The resulting frequent itemsets are the contention patterns that we are looking for.

V. Experimental Results

This section presents the experimentations and important results of our proposed method to extract contentions patterns from traces using the approach defined in Section IV. First, we present the simulation environment and architecture of the simulation platform. Second, we present how our approach helps the developer to discover important contention patterns in a video decoding application.

A. Simulation environment and Hardware architecture

Our experimentations are done on a simulated MPSoC architecture implemented using the SoCLib [16] infrastructure, which is a set of interoperable, VCI/OCP compliant, hardware component models in SystemC. We use the CABA (Cycle Accurate, Bit Accurate) simulation models, that include processors, caches, memories, and so on. The traces were generated using a non intrusive simulation-based trace system for MPSoC software proposed in [9]. The processor simulation is done using Instruction Set Simulators (ISS). The software that runs on this platform for our experiments is a parallel Motion-JPEG decoder on top of a operating system for embedded system that includes a Pthread library. The hardware platform is a shared memory multiprocessor that contains *n* MIPS32 processors such as $n = \{1, 4, 8\}$, interfaced with one data cache and one instruction cache. It also contains one memory and others peripherals components: a timer, an interrupt controller, a frame buffer, a block device, a tty. We perform three simulations on different platforms: platform 1, platform 2 and platform 3 contains 1, 4 and 8 processors, respectively. These three platforms should exhibit different contention levels, and thus help us validate our approach of contention pattern discovery. The simulation generates trace files for each CPUs of the simulated platform. These trace files are very large, often hundreds of gigabytes depending on the video duration decoded and the number of processors in the simulated platform.

B. Results

The first criterion taken into consideration when the performances of the parallel systems are analysed is the speed up used to express how many times a parallel program works faster than a sequential one. The speed up of the video decoding application results are 3.3 and 4.5 corresponding for 4, and 8 cores in a platform, respectively. It is just acceptable for 4 cores and bad for 8 cores, hence the video decoding application considered does not scale well with the number of cores. We verified that this bad scalability is not due to a lack of work or a load unbalance issue. Having eliminated these reasons for lacking of parallel scalability, the remaining reason is likely to be contention that slows down memory accesses and thus prevents the application to reach the desired speed up. It is thus justified to apply our approach in order to automatically detect the parts of the trace where contention occurs, and to understand through contention patterns the reasons for this contention. First, we explain what preprocessing was necessary on trace in order to apply our approach.

Trace preprocessing: The raw traces, as output by the simulator, contain for each trace event information that are not useful for our analysis, so we dispose of them. There are no function names, but only the PC address of the executed instruction: using the symbols table of the executable, we determine using well-known techniques [4] the function to which this instruction belongs and replace the PC by the function name. To be able to use the pattern mining algorithm, we discretized continuous numeric attributes into bins of numeric intervals in order to regroup events exhibiting similar values. As an example, the memory access latencies are discretized by bins of 10 from 0 to 250 *i.e.* all latencies between 0 and 10 are represented by the bin lat_0_10.

The first step of our approach compute the high latency values for each trace (trace for 1, 4 and 8 CPUs). The high latency thresholds in different platforms using 1, 4 and 8 CPUs are 9.65, 12 and 15 respectively. However, as the number of CPUs increases the number of the high latency thresholds also increases.

The second step of our approach exploits these thresholds in order to compute the windows identifying the parts of the traces exhibiting contention. We set the window duration to $\omega = 200$ cycles.

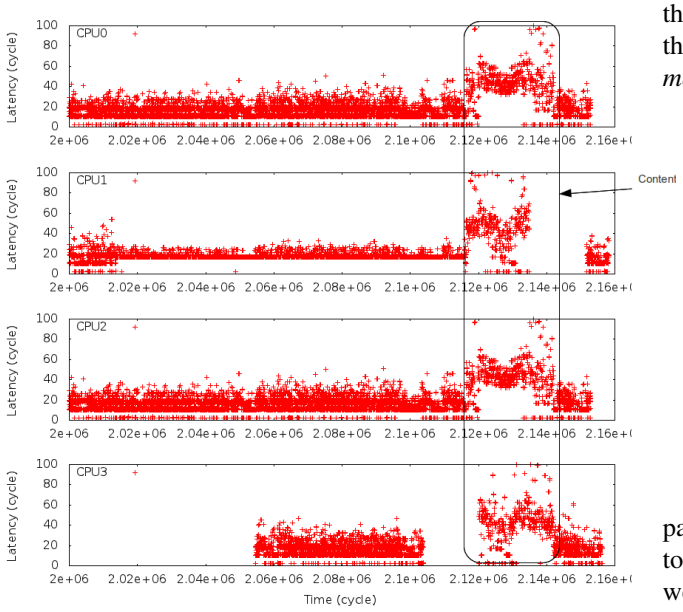

Fig. 2. Latency versus Time

We graphically illustrate our notion of high latency contention windows with a small portion of the trace for 4 CPUs, where we plotted for each event its latency value, and differentiated the CPUs. The resulting graph is shown in Figure 2. We can observe in the region highlighted by a rectangle that latencies get much higher than in the other regions of the graph: there must be contention in this period, and it covers around a hundred of contention windows.

Over all the traces, the number of contention windows found and the percentage of the trace they cover is summarized in Table I.

TABLE I
CONTENTION WINDOWS

Platform	Nb of windows	Coverage of trace
1 CPU	86 843	1.20%
4 CPUs	925 951	16.97%
8 CPUs	1 686 785	36.79%

As expected, there are few windows with high latencies for the platform with 1 CPU. However, as the number of CPUs increases the number of these windows also increases, and it covers a significant fraction of the execution time. This is in line with the speed up results, and confirms that contention is a problem for our experiments with 4 and 8 CPUs.

In order to better understand the reasons for contention, we plot in Figure 3 the frequency of apparition of instructions in the contention windows for 4 CPUs platform. More precisely, x-axis is the program counter identifying instructions, and y-axis is the frequency of apparition of each instruction over all contention windows. As instructions can be related to functions, this figure indicates which functions are the most responsible for contention. Three highly frequent groups arise that we identified with the corresponding function names: there are *memset*, *idct* and *memcpy* functions. The figure shows that contention is mostly due to *idct* and *memcpy*. However, this figure does not indicate the interactions between *idct* and *memcpy* in contention windows.

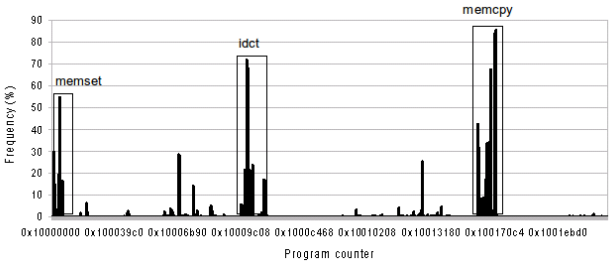

Fig. 3. Memory access frequency

In order to understand such interactions, we apply the pattern mining algorithm to the contention windows converted to transactions with a minimum support threshold of $\varepsilon = 65\%$: we are interested in interactions between functions, memory locations and CPUs that occur in more than 65% of contention windows, *i.e.* very frequently in potential the parts of the traces exhibiting contention. We focus on platform with 4 and 8 processors, which exhibit high levels of contention. The most interesting contention patterns discovered for these simulation platforms are presented in Table II.

TABLE II
FREQUENT PATTERNS

Platform	Frequent Pattern	Support
4 CPUs	CPU[0,3] [0x10009ee4, 0x10009f78] idct [0x10016b50, 0x10016f2c] memcpy lat_10_20 lat_20_30	72 %
8 CPUs	CPU[0,7] [0x10009b10, 0x1000a224] idct [0x10016ab0, 0x10016e8c] memcpy lat_10_20 lat_20_30	88 %

The pattern of the platform using 4 CPUs shows a concurrent memory access pattern that creates a contention implying all 4 CPUs and occurring in 72 % of contention windows. This pattern shows a frequent interaction between the functions *idct* and *memcpy*, and more specifically between the loops of *idct* located in address interval [0x10009ee4, 0x10009f78] and the loops of *memcpy* located in address interval [0x10016b50, 0x10016f2c]. The pattern also shows that the usual latencies around these interactions are between 10 and 30 cycles (lat_10_20, lat_20_30). Having in mind that the high latency threshold is $Q_3 = 12$ for the 4 CPUs trace, this corresponds well to contention latencies.

The pattern for the 8 CPUs platforms is the same as on the 4 CPUs platform, with different addresses due to a different executable. However these addresses correspond to the same assembler instructions than previously: this enforces

the importance of the *idct*/*memcpy* interaction. In the 8 CPUs platform the pattern has an even higher support of 88 %, whereas there are more contention windows in this case: this pattern is clearly the main responsible for most of the contention and thus the lack of scalability when the number of cores increase. This pattern thus helps the application developer to know that the *idct* function, which performs the inverse discrete cosine transformation, has negative interactions with *memcpy*, a function for copying data from one address to another. It even pinpoints the specific assembler instructions of both functions that are the most impacted: the developer, which is more likely to work on *idct* than *memcpy*, will know immediately which loop of *idct* he/she has to work on.

Discussion:

Our approach is more accurate than the existing current works [17], [3], [15] because it identifies the frequent concurrent memory access patterns, extracts from the execution traces the patterns that create a contention and generates a compact and readable output that can be analyzed by the software developers. Our tool helps the developers to highlight concurrent memory access patterns and the impact on the parallel scalability. In the experimentations, we saw, firstly, the high frequency interactions between *idct* and *memcpy* functions in a parallel platform. These interactions lead to contention in different platforms. However, it's difficult to find such interactions with the existing profiling tools [8], [10]. Secondly, *memcpy* is having a major impact on the parallel scalability of a video decoding application. Thus, the developer can optimize his program with the following possible solutions:

- Using software (or hardware) based prefetching caches or non-blocking caches techniques. These effective techniques allow to decrease memory access latency [6].
- In [19], a dedicated hardware accelerator was proposed that works in conjunction with caches found next to modern-day microprocessors, to speed up the commonly utilized *memcpy* operation.
- The *memcpy* function can be put between a lock/unlock pair to serialise the memory access and ensures that when one CPU is executing *memcpy*, no contention will be created. As such, serialization can be detrimental to performance, it can be activated only when *idct* is executing the loops identified in the previous frequent pattern and deactivated the rest of the time.
- Refactor the communication scheme of the whole application.

VI. Conclusion and future work

The automatic identification of parallel application contentions is a major issue for the optimization of application deployed in MPSoCs, as it is one of the key to enable good scalability. Using the trace generation capabilities of nowadays well accepted virtual platforms, we have presented an automatic approach based on data mining that when given only two thresholds, can automatically discover contention patterns. We have shown by experiments on a video decoding application that the patterns extracted are interesting and can provide information that will help the application developer to understand the reasons of the contention.

To the best of our knowledge, this is the first work reporting the use of data mining on MPSoC traces to identify the patterns that create contentions in multithreaded applications. Our results advocate that such approaches are of interest and that they can allow developers to save a lot of time during the optimization of their applications, a task that is very difficult to do manually or with pure visualization tools.

Our future plan consists on mining other characteristics than memory access latencies, i.e. interrupt latencies or spinlock stalls to identify other costly application behaviours.

References

[1] Workshop on frequent itemset mining implementations (fimi'04). 2004. http://fimi.ua.ac.be/fimi04/.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB*, pages 487–499, 1994.

[3] N. Alfaraj, J. Zhang, Y. Xu, and H. J. Chao. Hope: Hotspot congestion control for clos network on chip. In *NOCS*, pages 17–24, 2011.

[4] R. M. Balzer. Exdams: extendable debugging and monitoring system. In *Proceedings of the 1969 Spring Joint Computer Conference*, AFIPS '69 (Spring), pages 567–580. ACM, May 1969.

[5] P.-H. Chang and L.-C. Wang. Automatic assertion extraction via sequential data mining of simulation traces. In *ASP-DAC*, pages 607–612, 2010.

[6] T.-F. Chen and J.-L. Baer. Reducing memory latency via non-blocking and prefetching caches. In *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, ASPLOS-V, pages 51–61, New York, NY, USA, 1992. ACM.

[7] F. Flamand. Strategic directions towards multicore application specific computing. In *Proceedings of the Design, Automation and Test in Europe Conference*, page 1266, Nice, France, Apr. 2009. Keynote speach.

[8] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler. *SIGPLAN Not.*, 39(4):49–57, Apr. 2004.

[9] D. Hedde and F. Pétrot. A non intrusive simulation-based trace system to analyse multiprocessor systems-on-chip software. In *International Symposium on Rapid System Prototyping*, pages 106–112, 2011.

[10] Intel Corporation. Using intel vtune's counter monitor. January 2005.

[11] C. LaRosa, L. Xiong, and K. Mandelberg. Frequent pattern mining for kernel trace data. In *ACM Symposium on Applied Computing*, pages 880–885, 2008.

[12] J.-J. Lim, J. Menon, and D. Palmer. A distributed development environment for embedded software. *Softw., Pract. Exper.*, 23(11):1235–1248, 1993.

[13] P. Malani, Y. Tan, and Q. Qiu. Resource-aware high performance scheduling for embedded mpsocs with the application of mpeg decoding. In *ICME*, pages 715–718, 2007.

[14] K. Potter. Methods for presenting statistical information: The box plot. *Hans Hagen, Andreas Kerren, and Peter Dannenmann (Eds.), Visualization of Large and Unstructured Data Sets, GI-Edition Lecture Notes in Informatics (LNI)*, pages 97–106, 2006.

[15] R. S. Ramanujam and B. Lin. Destination-based adaptive routing on 2d mesh networks. In *ANCS*, page 19, 2010.

[16] SoCLib Consortium. A library of cycle accurate system simulation models. http://www.soclib.fr, 2010.

[17] L. Tedesco, T. R. da Rosa, F. Clermidy, N. Calazans, and F. G. Moraes. Implementation and evaluation of a congestion aware routing algorithm for networks-on-chip. In *SBCCI*, pages 91–96, 2010.

[18] T. Uno, M. Kiyomi, and H. Arimura. Lcm ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets. In *FIMI*, 2004.

[19] S. Wong, F. Duarte, and S. Vassiliadis. A hardware cache memcpy accelerator. In *In Proc. IEEE International Conference in Field Programmable Technology*, pages 141–147, 2006.

[20] J. Zou, J. Xiao, R. Hou, and Y. Wang. Frequent instruction sequential pattern mining in hardware sample data. In *ICDM*, pages 1205–1210, 2010.