Scalability Bottlenecks Discovery in MPSoC Platforms Using Data Mining on Simulation Traces

Sofiane Lagraa[1,2], Alexandre Termier[1] and Frédéric Pétrot[2]
Université Grenoble Alpes. CNRS, [1]LIG, [2]TIMA, F-38031 Grenoble

Abstract—**Nowadays, a challenge faced by many developers is the profiling of parallel applications so that they can scale over more and more cores. This is especially critical for embedded systems powered by Multi-Processor System-on-Chip (MPSoC), where ever demanding applications have to run smoothly on numerous cores, each with modest power budget. The reasons for the lack of scalability of parallel applications are numerous, and it can be time consuming for a developer to pinpoint the correct one. In this paper, we propose a fully automatic method which detects the instructions of the code which lead to a lack of scalability. The method is based on data mining techniques exploiting low level execution traces produced by MPSoC simulators. Our experiments show the accuracy of the proposed technique on five different kinds of applications, and how the information reported can be exploited by application developers.**

I. Introduction

Embedded devices have been powered for some time by Multiprocessor System on chip (MPSoC), which integrate in a single die general purpose and specialized computation cores as well as memory and external interfaces, allowing to reduce costs and power consumption while increasing performances. In an extremely competitive market, application developers are hard-pressed to provide demanding applications (multimedia, computer vision, software-defined radio, ...) that run smoothly without draining too much power. They thus require good profiling tools. However, such profiling tools are still an active research topic and actual tools are scarce, even though more and more dedicated hardware is added to support this task [11].

In this paper, we focus on one critical point of parallel applications, their *scalability*. Intuitively, a parallel program is *scalable* if it runs n times faster on n cores than on 1 core. In this case, it is said that there is a *linear speedup*. In practice such scaling cannot be obtained by all programs, and the well known Amdahl's law gives a more precise bound on the maximal speedup that can be reached by a given application. There are many reasons that can prevent the scaling of a parallel application: the program can spend too much time doing synchronization, it can suffer from congestion on memory accesses or accesses to other external resources, or there can be load unbalance, or cache trashing, etc. It is tedious for an application developer to find the correct reason for a lack of scalability among all those.

Our contribution is to propose a fully automatic method that discovers the main reasons for lack of scalability of an application, and reports the exact code lines involved. The developer can thus directly concentrate on understanding and solving the problem found, gaining a lot of time in the profiling process. Our method is based on the analysis by data mining techniques of low-level execution traces produced by running the application on a MPSoC simulator. Using such simulators is already part of the workflow of MPSoC application development. Indeed, due to the fast evolution rate of these chips, applications often start to be developed before the chip physically exists. Because of the complex execution of these applications on MPSoC, collecting traces and analyzing them *a posteriori* has emerged as the best way to understand the complex interactions between the components of the MPSoC. Our method thus integrates in the existing workflow of MPSoC application development, bringing further benefits for profiling scalability.

The rest of the paper is organized as follows. Section II states the problem and gives an outline of the proposal. We detail our approach in Section III. Then, we present the experimental results on MPSoC platforms running five software applications in Section IV. Section V briefly reviews related works, and Section VI concludes and gives some future research directions.

II. Problem statement

The approach we present in this paper being based on executions traces, we formally define the traces that we consider. These execution traces, obtained by running applications in a simulated MPSoC environment, consists of fine grained information about the execution. These traces, viewed as a set of events, are collected with the non intrusive simulation-based trace system to analyse Multiprocessor Systems-on-Chip software presented in [8]. We first give definitions and notations used throughout this paper, and then give more specific details about our traces.

Definition 1 (Trace event): Let TS be a set of trace symbols. A *trace event* or an *event* $e = (ts, cpuid, latency, s)$ consists of a timestamp $ts \in [0, t_{max}]$, a CPU identifier $cpuid \in \{1, ..., m\}$, a latency $latency \in \mathbb{N}^+$ and a set of symbols $s \subseteq TS$ representing the event s that has been performed by CPU $cpuid$ at time ts with a latency $latency$.
Practically, the trace event has the fixed form represented by Table I. It consists of, in order of occurrence, which CPU initiated the transaction, the global date at which the event occurred in cycles since the power-up of the system, the program counter of the instruction that produced the access or the data address accessed, the transaction type which can be instruction fetch, load/store, load-linked/store-conditional

TABLE I
RAW TRACE FORMAT

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

pairs, and finally the memory access latency as seen by the CPU. Formally, the instruction address or data address are noted $@_i = e_i.program\_counter \vee e_i.data\_address$. Trace events are collected in an execution trace:

Definition 2 (Execution trace): 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 $\forall i \in [1, n] \;\; ET[i] = e_i$.

We can now define some metrics on the trace:

Definition 3 (% Time spent): Given an execution trace ET and an address $@_i$, $\%\_time\_spent(@_i, ET)$ is the percentage of the total execution time of the program spent in this adress. Let $ET(@_i) = \{e \mid e \in ET \wedge e.s \supseteq \{@_i\}\}$ be the events of ET that are accesses to $@_i$, we have:

$$\%\_time\_spent(@_i, ET) = \frac{\sum_{e \in ET(@_i)} e.latency}{\sum_{e \in ET} e.latency} \times 100$$

Definition 4 (% accesses): Given an execution trace ET and an address $@_i$, $\%\_accesses(@_i, ET)$ is the percentage of the total number of accesses that were done to $@_i$.

$$\%\_accesses = \frac{|ET(@_i)|}{|ET|} \times 100$$

From these metrics, it is possible to evaluate how detrimental to performance an access is likely to be:

Definition 5 (Hot predicate, hot access): Given an execution trace ET and an address $@_i$, a predicate $isHot(@_i, ET)$ is called *hot predicate* if it answers $true$ when both $\%\_time\_spent(@_i, ET)$ and $\%\_accesses(@_i, ET)$ are significantly higher for $@_i$ than for the other addresses, and $false$ otherwise.
An $@_i$ for which $isHot(@_i, ET) = true$ is called a *hot access*.
The definition of hot predicate exhibits the two main characteristics of problematic regions of the code: first, the time spent and number of accesses are unusually high for a set of accesses, and second this problem occurs several times in the execution, further degrading performance. However, such hot predicate, even if detrimental for performances, may have no impact at all on the parallel scalability of the application considered. We thus propose a definition of hot predicates having an impact on scalability: the *scalability hotspots*.

Definition 6 (Hotspot): A *hotspot* HA is a set of hot accesses appearing consecutively in the execution trace ET.

Definition 7 (Scalability hotspot): Let $P_1, .., P_k$ be k homogeneous MPSoC platforms only differing in their number of cores, with for all $i < j \in [1..k]$ P_i has less cores than P_j. Let $ET_1, ..., ET_k$ be execution traces of an application, where ET_i has been produced on platform P_i using all its cores. Let min_p be a user given threshold, with $min\_p \in [1..k]$.

A set of accesses HS is a *scalability hotspot* if:

- For the accesses in HS, the metrics $\%\_time\_spent$ and $\%\_accesses$ increase with the number of cores of the platforms where HS is a hotspot
- HS is a hotspot in at least min_p execution traces of $ET_1, .., ET_k$

Problem statement: Given a set of execution traces $ET_1, ..., ET_k$ produced by platforms $P_1, .., P_k$ as defined above and user threshold min_p, our goal is to discover the scalability hotspots of the traces. Such scalability hotspots are the parts of the code that are most likely to impact parallel scalability, and should be investigated in priority by the application developers. Thus, our objective is to quantify and pinpoint the bottlenecks in multi-threaded application.

III. Scalability Bottlenecks discovery method

The five major scalability bottlenecks of multi-threaded workloads on multi-core hardware are: resource sharing, cache coherency, synchronization, load imbalance, and parallelization overhead [5]. For discovering such scalability bottlenecks, our proposed approach uses *data mining* techniques in order to analyze automatically large quantities of execution traces and discover such bottlenecks of software. This approach takes as input a set of traces resulting from the simulated execution of the same program, with the same parameters, on a simulated MPSoC with a scalable number of cores. We call each of these MPSoC instances a *platform*. Our approach outputs addresses that are the more impacted when running on more cores, forming what we call scalability hotspots (Def 7). These addresses can either be addresses of instructions in the code, pinpointing parts of the code that are responsible of the scalability issues, or addresses of data, indicating where the memory of the MPSoC is not used efficiently. Our approach is described in the following steps.

A. Trace collection

The trace files are obtained by running the same multi-threaded program on different instances of the multi-processor platform, each instance differing from the other by its number of processors (either by instantiating more processors, or simply by having only a subset of all processors actually active). From each platform instance, the execution traces are saved into trace files. These trace files are used for discovering scalability hotspots across the platform instances. Each trace file corresponding to a MPSoC platform, the following steps are applied independently of the other trace files: the feature extraction and feature-based clustering.

B. Feature extraction

Trace processing may include transformation of the original traces into simplified ones, along with reduction of dimensionality by extraction of only the most informative features from a huge amount of execution traces. Extracting the features may improve the recognition process and make easier the extraction of the critical zones through the consideration of only the most important traces representation. For a trace of a given platform, each trace line gives information on an access to an address. For each such address, we compute the following statistics, called *features*: $\%\_time\_spent$ and $\%\_accesses$ according to definitions 3 and 4. The features are represented by the following address-feature vector of the platform $j \in [1, p]$ where the platforms are numbered in $[1, p]$ and platform j has by convention 2^j cores and has corresponding raw trace ET_j:

$$X_{p_j} = [X_1, X_2, ..., X_{n_j}] \qquad (1)$$

where $\forall i \in [1, n_j]$ $X_i = v(@_i, x, y, z)$, with n_j the number of different addresses in the trace ET_j, the feature vector v with $x = \%\_time\_spent(@_i, ET_j)$ and $y = \%\_accesses(@_i, ET_j)$.

In the traces, we have only the address of the executed instruction and information about the source code (*i.e.* instruction or line number in the source code). In order to have a higher level of granularity in the traces and facilitate the interpretation of the results, we use the symbol table of the executable to determine using well-known techniques [2] the function to which this instruction belongs. In the feature vector, $z = e_i.function$ is the function name. These features give first performance metrics at the level of granularity of the address, similar to those provided by `gprof` [6] at the level of granularity of functions. Transforming each trace ET_j as a feature vector X_{p_j} first allows an important compression of the volume of data mining algorithms will have to process. We also show in the following sections that the features of vector X_{p_j} are sufficient for discovering scalability hotspots.

C. Feature-based clustering

This step allows to discover automatically groups of a set of accesses using *clustering* algorithm. Clustering is a data mining technique for organizing data elements into similarity groups, called *clusters* such that the data elements in the same cluster are similar to each other and data element in different clusters are very different from each other. A classical clustering algorithm is *k-means* [7]. The k-means algorithm is the best known partitional clustering algorithm. It is also widely used among all clustering algorithms due to its simplicity and efficiency. Given a set of data points and the required number of *k* clusters (*k* is specified by the user), this algorithm iteratively partitions the data into *k* clusters based on a distance function such as Euclidean distance. Each cluster has a cluster center called the **centroid**. The centroid, usually used to represent the cluster, is simply the mean of all the data points in the cluster, which gives the name to the algorithm, since there are *k* clusters. The clustering is the basis of our hot predicate definition. The clusters are obtained by applying *k-means* clustering algorithm on X_{p_j} with the number of clusters *k* as an input parameter. The result of *k-means* algorithm is the cluster feature vector (2) of the platform *j* which is the extension of the address-feature vector (1) with the cluster identifier assigned to each address performed by processors.

$$X_{p_j} clusterVector = [X'_1, X'_2, ..., X'_{n_j}] \qquad (2)$$

Where $X'_i = v(@_i, x, y, z, C_{@_i})$, and $C_{@_i} \in [1, k]$ is the identifier of the cluster for address $@_i$ such as k is the maximum number of clusters.

In this work, we set the number of clusters $k = 2$ as we are interested to distinguish two types of accesses: *hot accesses* within a *hot cluster* and other accesses in the second cluster called *normal cluster*. Formally, the hot cluster is based on its centroid that satisfies the following definition:

Definition 8 (Hot Cluster): Let two centroids $c_{p_0}(x_{p_0}, y_{p_0})$ and $c_{p_1}(x_{p_1}, y_{p_1})$ of the platform having p processors such as $c_{p_0} \in C_0$ and $c_{p_1} \in C_1$ where C_0 and C_1 are two clusters and x is the percentage of the time spent $\%\_time\_spent$ and y is the percentage of accesses $\%\_accesses$. C_1 is a hot cluster if $c_{p_1} > c_{p_0}$ which is true if $(x_{p_1} - x_{p_0}) + (y_{p_1} - y_{p_0}) > 0$, and normal cluster otherwise.

By definition, we consider that the hot cluster will always have the label C_1. The hot predicate $isHot$ for an address $@_i$ simply consists in testing if $@_i$ is in the hot cluster or not. The set of hot accesses for platform P_j is thus $Hot_j = \{@_i \mid @_i \in [1, n_j] \wedge isHot(@_i, ET_j) = true\}$. Hot cluster give, for each platform, the set of hot accesses of that cluster. Now when considering all platforms together, it becomes interesting to check if there are set of hot accesses that are found in several hot clusters, indicating that they are problematic for several platforms. Furthermore, for these sets of hot accesses found in several platforms, if their performance statistics decrease with the number of cores, it is a high indication that these hot accesses are scalability hotspots. We present in the next two sections these two last steps for discovering scalability hotspots.

D. Growth rate of hot cluster

The hot clusters are computed independently for each platform. The next step of the analysis, presented in this section, is to determine if there is a correlation between the increase of number of cores in platforms and the statistics determining the hot clusters. This way we can determine if the hot clusters can help to determine a scalability problem.

Definition 9 (Performance loss): Given the hot clusters extracted from each platform, we say that an application loses its performance if the centroid of the hot cluster evolves across platform instances, *i.e.* both $\%\_time\_spent$ and $\%\_accesses$ of the centroid grow with the number of cores.
Such a performance loss is illustrated in Fig. 1.

Discovering the impact of the loss of performance of scalability hotspots is not an easy task. When the number of processors grows in the platform instances, the distance between the centroids of the two clusters grows too. Thus, we define a metric based on the euclidean distance between the centroids of the clusters. It measures the evolution of the distance in a multi-core platform relative to the distance in the one core platform. This principle is inspired from the speed-up metric. This metric is necessary to evaluate the impact

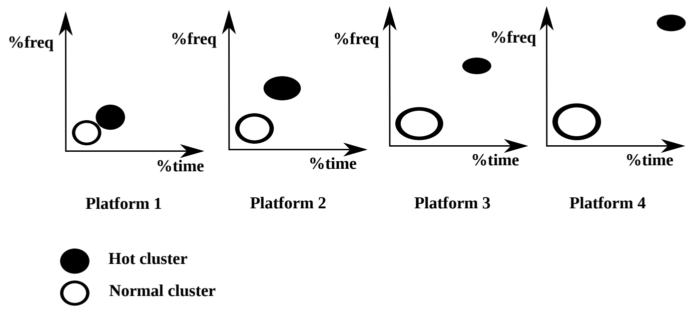

Fig. 1. Hot cluster evolution

of scalability hotspots on application performance. Therefore, this metric is called the *growth rate* metric.

Definition 10 (Growth rate): The *Growth rate (Gr_p)* refers to how much the distance between the two centroids $c_{p_1}(x_{p_1}, y_{p_1})$ and $c_{p_2}(x_{p_2}, y_{p_2})$ of the platform having p processors grows relative to the corresponding distance between the two centroids $c_1(x_1, y_1)$ and $c_2(x_2, y_2)$ of the platform having 1 processor.

$$Gr_p = \frac{\sqrt{(x_{p_1} - x_{p_2})^2 + (y_{p_1} - y_{p_2})^2}}{\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}} \tag{3}$$

Where p is the is the number of processors in a platform. Gr_p is a value, typically between 1 and 100, estimating how many times the hot clusters grows over platforms, compared to how much the hot cluster containing the scalability bottlenecks patterns decrease the performance of the platform, *i.e.* how much effort is wasted in communication, synchronization or waiting state. Thus, the centroid of the cluster provides a scale for measuring the cluster evolution over different platforms.

The interest of computing the growth rate is not only to measure the impact of the scalability bottlenecks but also to aid the developer to make a decision if a given program needs to be optimized. If the growth rate for two platforms with increasing number of cores is close to zero, it is likely that the program has no parallel scalability problem, or if one exists, it is not possible to detect it with our metrics. Otherwise, large positive values of growth rate indicate that the instructions contained in the hot clusters are likely to cause parallel scalability bottlenecks. The next section will focus on pinpointing the instructions of the hot clusters that the developer should investigate in priority.

E. Frequent scalability bottlenecks mining

It is vital to understand bottlenecks in platforms and their impact over the scalability for optimizing application performance and design future hardware. From the hot accesses, the developer wants to discover the frequent hot accesses common in each platform in order to focus on the parts of the code to improve. For this, we describe the frequent scalability hotspot mining method which discovers the set of hot accesses on multi-threaded application across multi-core platforms instances. A delicate problem is to find the frequent patterns that decrease the performance when the number of processors increases. We thus need to discover the frequent scalability hotspots, and do so using *frequent itemset mining algorithm* existing in *data mining* for mining instructions memory addresses that belong to the hot clusters through all platform instances. The extracted patterns are the most likely to be responsible of scalability issues. In the frequent itemset mining algorithm, the first input is a multiset of *transactions* $D = \{t_1, .., t_p\}$ defined over *items* in our case, the items are the hot accesses of all platforms, i.e. $\Sigma = \cup_{i \in [1,p]} Hot_j$, where $\forall t_i \in D \ \ t_i \subseteq \Sigma$. To do this, we transform the set of hot accesses into a set of transactions D by merging all hot clusters in a same transaction table. Each hot cluster becomes a transaction, *i.e.* a set of hot accesses. The second input is a *minimum support threshold* $min\_p \in [1, p]$ where p is the number of platform instances. Frequent itemset mining algorithms then extract all the *frequent hot accesses*, *i.e.* all the *hot accesses is* $\subseteq \Sigma$ that appear in more than min_p transactions of D. Once we have the transactions, we can use a state of the art frequent itemset mining algorithm. We use LCM [15], the most efficient one according to the FIMI contest [1], to which we provide the minimum support threshold min_p and the transactions of hot accesses.

Example: Let a number of platform instances $p = 3$, a minimum support threshold $min\_p = 2$ and set of hot accesses and the functions contained in their hot cluster respectively. We assume the following transactions:
Platform 1 itemset is `{0x01,0x02,0x03,0x9,0x10}` $\in Hot_1$, where `{0x01,0x02,0x03}` $\in$ function f_1 and `{0x9,0x10}` $\in$ function f_2.
Platform 2 itemset is `{0x11,0x12,0x13,0x19,0x20}` $\in Hot_2$, where `{0x11,0x12,0x13}` $\in$ function f_3 and `{0x19,0x20}` $\in$ function f_4.
Platform 3 itemset is `{0x31,0x32,0x33,0x19,0x20}` $\in Hot_3$, where `{0x31,0x32,0x33}` $\in$ function f_5 and `{0x19,0x20}` $\in$ function f_4.
The frequent pattern is thus `{0x19,0x20}` $\in$ function f_4, occurring in platforms 2 and 3.

The discovered frequent hot accesses with their frequencies pinpoint the scalability bottlenecks in the source code, and should be investigated by the developer.

IV. Experimentations and results

This section presents the experimentations and important results of our proposed method for detecting the scalability problem in multi-processors platforms.

A. Simulation environment and Hardware architecture

Our experimentations are done on a parameterizable simulated MPSoC architecture in which we can vary the number of processors. It is implemented using the SoCLib [13] 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, which include processors, caches, memories, and so on. The traces are generated using a non intrusive simulation-based trace system for MPSoC software [8]. The processor simulation is done using interpretive Instruction Set Simulators (ISS). We performed our experiments on five applications that run on this platform: Matrix multiplication, parallel Motion-JPEG decoder, SPLASH-2 [3] parallel benchmarks suite: Ocean (that simulate large scale ocean movements), Fast Fourier transform (FFT) and LU decomposition factors a matrix as the product of a lower triangular matrix and an upper triangular matrix. The hardware platform architecture is a coherent shared memory multiprocessor that contains n MIPS32 processors such as $n = \{1, 4, 8, 16\}$, 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.

B. Results

In this section, we show the different results found in experimented applications. The scalability hotspots in multi-threaded applications that undergo evolution across scalable platforms are discovered and the hot accesses are distinguished from other accesses. Table II shows the frequent hot patterns discovered having a minimum support threshold $min\_p = 25\%$. The frequent patterns discovered from hot clusters in each platform P_i running the same multi-threaded applications highlight the common critical zones.

TABLE II
SCALABILITY HOTSPOTS

Software	scalability hotspot pattern	% Occurrence
Matrix Multiplication	[1000825c:10008268] cpu_mp_wait	75 %
Ocean	[100235b0:100235dc] lock_acquire	75 %
Motion-JPEG	[100023b4:100023d0] soclib_fb_write	75 %
	[100171a4:100171d4] __malloc_lock [100171e0:10017200] __malloc_unlock	50 %
FFT	10044650 (data address)	75 %
	[1000b0a0:1000b0a8] cpu_mp_wait	
LU	[10013c60:10013c80] __muldf3 [100147b8:10014830] __unpack_d	100 %

In matrix multiplication application, we see that the scalability hotspot contains synchronisation addresses in the *cpu_mp_wait* function, decreasing the performance by its evolution in each platform instance as shown in Fig. 3. The pattern represents a sequence of instructions in a loop as shown in Fig. 2. The particular loop means that the processors are in a 'wait' state waiting to be 'notified' of work to be done. Therefore, it induces load imbalance of the tasks in cores, with a high impact on scalability as shown by the growth rate in Fig. 3. In Splash's Ocean application, like the first experiment the growth rate increases with the number of cores of the platform. The discovered scalability hotspot contains

Fig. 2. Scalability hotspot in assembly code for the matrix multiplication application.

```
10008250:    <cpu_mp_wait>
                          …
1000825c:    sync
10008260:    lw v0,-17120(v1)
10008264:    bnez v0,10008260 <cpu_mp_wait+0x10>
10008268:    nop
```

the address range grouped into *lock_acquire* function. The *lock_acquire* function acquires access to a specific lock being represented by a given lock set. If the lock is already controlled by another thread then the calling thread will spin. It means that the high number of accesses and their high percentage of execution time are grouped in synchronization operations into number of barriers (locks) encountered per processors when they access to the critical section. In Motion-JPEG application, some addresses in the hot cluster are present in each platform: the frequent pattern is a loop of *soclib_fb_write* function responsible for displaying the decoded image. The evolution of this loop is stable in both 4, 8 and 16 CPUs platforms. Other results are detected by the approach, the frequent accesses to the data addresses are called by set of instruction belonging to *memcpy*, and specially to the *__malloc_lock* and *__malloc_unlock* functions that copies the values from one memory block to another, and protect/release that memory blocks from corruption during simultaneous allocations, respectively. In LU application, the scalability hotspots is the software functions: **__unpack_d** and **__muldf3**. These functions are associated with the manipulation of floating-point numbers and contained in **fp-bit** and **libgcc** libraries of GCC complier, respectively. Regarding the impact of discovered scalability hotspots in applications, there are more important and significant growth rate increases across platform instances in matrix multiplication and Ocean applications which is not the case for Motion-JPEG and LU and a little evolution of patterns in FFT (Fig. 3). Thus, these results indicate that Motion-JPEG, FFT and LU applications are sufficiently optimized unlike Matrix multiplication and Ocean applications. In [4], the author analyzes the performance of Ocean application and notes that the increased communication cost is the reason for speed-up degradation. With our tool, we confirm that by the detection of the increased accesses to the lock. These results confirm the interest of our approach, and its ability to provide both visual clues on the sources of scalability issues as well as precise code location that the developer should examine.

V. Related Works and Discussion

Profiling critical jobs in parallel platforms and identifying bottlenecks inside applications is hard [10]. Recently, new solution based on data mining has been applied in execution traces for discovering contention in MPSoC platform [12] but this solution is for contention detection. In [14] the authors propose Scal-Tool. Scal-Tool is a tool that isolates and quantifies scalability bottlenecks such as insufficient caching space, load imbalance, and synchronization in parallel applications running on distributed shared memory machine. Scal-Tool is

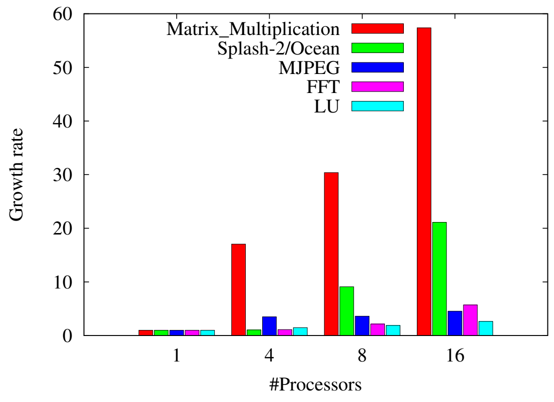

Fig. 3. Growth rate evolution over platform instances running five multi-threaded applications

based on an empirical model that uses Cycles Per Instruction (CPI) equations, and uses as inputs the measurements from hardware event counters in the processor. In [5], the authors propose *speedup stacks* which is a representation that quantifies the impact of individual components of scalability bottlenecks. It isolates and quantitatively estimates the cycle count impact of different scalability bottlenecks. A cycle stack (a.k.a CPI stack) is used in [9] to analyze multi-threaded programs and understand performance bottlenecks for multi-core environments as there could be other factors that were not seen in a single core environment. They simulate the analyzed program and capture cycle stacks for each individual thread and employ a statistical data analysis technique that extracts important trends from data.

The originality of our approach compared to the non-exhaustive list of works above is that first, it targets explicitly MPSoCs and/or multi-core processors and addresses the delicate problem of scalability. Second, it not only quantifies the bottlenecks but also it pinpoints them in source code. Third, we provide a framework with exact measures adapted to instruction-level traces unlike the related work based on estimations. Hence, using our framework, the scalability bottleneck patterns discovered help the user to improve his/her multi-threaded applications. Once the hotspots discovered, the developer focuses only on pinpointed section for improving the performance of the multi-threaded applications.

VI. Conclusion and Future Work

In this paper, we have proposed a new approach for discovering the scalability hotspots that reduce the parallel performance in MPSoC platforms running embedded software. The proposed approach uses data mining techniques on simulation traces, thus offering several advantages. It can profile the parallel embedded software in one (intra platform) or multiple platforms (inter platforms), and it is applicable on embedded systems as well as on parallel machines as long as detailed information on the memory accesses are available. It can be performed on homogeneous or heterogeneous architectures with different type of processors in order to select the most appropriate processors type running a given multi-threaded program. The approach uses data mining techniques which help in the discovery process of the unknown scalability bottlenecks of a given application from traces. The approach only requires one parameter, and then is completely automatic. It can thus be very helpful in reducing the amount of work a programmer has to do.

To the best of our knowledge, this is the first work reporting the use of data mining on traces to identify the scalability problem in multi-processors platforms. Our experimental results indicate that our approach based on data mining techniques discovers unknown specific problems of a given application and specific instructions decreasing the performance. The quantified and pinpointed scalability bottlenecks help the developers to understand better the scalability problems of their parallel applications. Our future plan consists on varying the number of clusters k in order to distinguish more finely several types of performance problems.

References

[1] Workshop on frequent itemset mining implementations, 2004. "http://fimi.ua.ac.be/fimi04/".

[2] R. M. Balzer. Exdams: extendable debugging and monitoring system. In *AFIPS*, pages 567–580, New York, NY, USA, 1969. ACM.

[3] W. S. Cameron, O. Moriyoshi, T. Evan, S. J. Pal, and G. Anoop. The splash-2 programs: characterization and methodological considerations. In *ISCA*, pages 24–36, 1995.

[4] M. Dipperstein. Splash-2 ocean performance analysis. michael.dipperstein.com/ocean/speedup.ps. Technical report, 2003.

[5] S. Eyerman, K. D. Bois, and L. Eeckhout. Speedup stacks: Identifying scaling bottlenecks in multi-threaded applications. In *ISPASS*, 2012.

[6] J. Fenlason and R. Stallman. Gnu gprof. 2003. [Accessed April 6th 2008].

[7] J. A. Hartigan and M. A. Wong. A K-means clustering algorithm. *Applied Statistics*, 28:100–108, 1979.

[8] D. Hedde and F. Pétrot. A non intrusive simulation-based trace system to analyse multiprocessor systems-on-chip software. In *RSP*, pages 106–112, 2011.

[9] W. Heirman, T. E. Carlson, S. Che, K. Skadron, and L. Eeckhout. Using cycle stacks to understand scaling bottlenecks in multi-threaded workloads. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*, IISWC '11, pages 38–49, 2011.

[10] R. Hoffmann and T. Rauber. Profiling of task-based applications on shared memory machines: scalability and bottlenecks. In *Euro-Par*, pages 118–128, Berlin, Heidelberg, 2007. Springer-Verlag.

[11] A. B. T. Hopkins and K. McDonald-Maier. Debug support strategy for systems-on-chips with multiple processor cores. *IEEE Transactions on Computers*, 55(2):174–184, 2006.

[12] S. Lagraa, A. Termier, and F. Pétrot. Data mining mpsoc simulation traces to identify concurrent memory access patterns. In *DATE*, pages 755–760, 2013.

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

[14] Y. Solihin, V. Lam, and J. Torrellas. Scal-tool: pinpointing and quantifying scalability bottlenecks in dsm multiprocessors. In *Supercomputing*, New York, NY, USA, 1999. ACM.

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