Towards a compiler/runtime synergy to predict the scalability of parallel loops

Georgios Chatzopoulos*, Kornilios Kourtis†, Nectarios Koziris*, and Georgios Goumas*

* School of Electrical and Computer Engineering
National Technical University of Athens, Greece
E-mail: {gchatz,nkoziris,goumas}@cslab.ece.ntua.gr

† Department of Computer Science
ETH, Zurich, Switzerland
E-mail: kkourt@inf.ethz.ch

***Abstract*—Large classes of applications fail to scale well in CMPs due to contention in the memory subsystem. Assigning full core capacity to such applications is a clear resource waste. To support efficient and power-aware resource allocation policies, we need a prediction mechanism to provide information about the potential scalability of an application. In this paper we take an initial step towards building a scalability predictor, based on the utilization of information collected both during compile and runtime. Our approach is applied separately to each `parallel-for` region in the program and calculates an on-chip to off-chip activity ratio S_r, which then is associated to the scalability of the region (maximum speedup) with linear regression. Experimental results on two architectures using the Polybench suite demonstrate that our prediction model exhibits a good accuracy in predicting the scalability of various `parallel-for` regions.**

I. Introduction

Multithreaded applications can execute on multiple cores, albeit they do not necessarily scale perfectly for a variety of reasons: synchronization overheads, hardware contention, OS scalability problems, load imbalance. Memory contention, in particular, has been shown to be a major hindrance to scalability for parallel applications [5], [6], [11], [22], [33].

Limited scalability introduces a challenging problem: determining the maximum parallel speedup an application can achieve to enable for efficient utilization of cores. For example, many applications exhibit diminishing returns in performance after using a portion of the cores of the system, at which point the rest of the cores should be suspended to save power or used to execute other applications. Solving the core allocation problem, however, requires a mechanism to predict the scalability of a parallel application — i.e. its ability to utilize cores. Adequately accurate solutions could substantially facilitate the task of contention-aware schedulers [5], [7], [25], [28], or automatic parallelization frameworks like PLUTO [8], where the decision whether a loop is parallel would be accompanied by the appropriate concurrency level (number of threads).

Predicting scalability is difficult because it highly depends on both the code being executed and the underlying hardware. One approach would be to treat the application as a black box and try to use runtime measurements (e.g. via performance counters) to predict its scalability, as is the case of the ACTOR runtime system [11], [30]. Application-agnostic approaches, however, ignore useful information that can be extracted from the code. For example, an application will typically exhibit different behavior in different parallel regions. Extracting the boundaries of these regions using code analysis, a trivial task for many parallel programming languages, can eliminate the run-time overhead (e.g., due to sampling or stability concerns) of application-agnostic methods asserting a behavioral change.

In this paper, we advocate the utilization of application information collected both during compile and runtime, in order to build a synergistic scalability predictor. In particular, we consider a synergy between the compilation framework and the parallel runtime system, where program information collected statically remains alive during program execution, and conveyed to the runtime system in order to support its decision on the concurrency level for each parallel region. We apply our mechanism to each separate parallel region of the original application, which in our case refers to parallel loops that operate on regular data structures (arrays). Our approach sacrifices generality for simplicity. For example, although it cannot be applied to arbitrary parallel programs (e.g. those written using the `pthread` library), it makes it trivial to identify the parallel regions of the program and provide a different prediction for each region. As we show in our evaluation, different regions may indeed have substantial differences in their scaling. We need to mention, however, that the restrictions induced by our scheme preclude a minority of parallel regions, as the large majority of parallel constructs facing scalability problems due to memory contention can be successfully analyzed.

Our static analysis examines the loop body and classifies operations as off-chip or on-chip and assigns a specific score to each one of them. The key observation inspired by the roofline performance prediction model [33], is that parallel constructs with heavy off-chip traffic tend to saturate the memory bus and thus fail to scale well with increasing number of cores. To accurately characterize off-chip and on-chip activity we need runtime information on the actual size of the data structures involved in the execution of the parallel construct. This is the only input brought in by the runtime system which finalizes the prediction task by calculating the scalability level of the parallel construct. We experiment with a simple prediction model, which is the ratio of the on-chip to off-chip traffic scores and use a small set of parallel loops to train a linear regression model as our predictor. Although simple, our experimental results on two CMP machines using loops from the Polybench suite [29], verify that the on-chip to off-chip scores ratio shows a very high correlation to the scalability level of the application. Regarding prediction accuracy, our method shows acceptable accuracy being able to capture the scalability behavior of parallel regions in the great majority of the cases.

The rest of the paper is organized as follows: In Section II we provide details about our scalability prediction framework, accompanied by information about the architectural and application model. Section III presents our model evaluation and Section IV discusses related work. Section V discusses limitations of our model and ideas for relaxing those limitations, while Section VI concludes the paper.

II. A SYNERGISTIC SCALABILITY PREDICTOR

In prediction models there is an inherent tension between precision, completeness and simplicity. The major goal of the synergistic scalability predictor we present in this paper is to facilitate an initial experimental evaluation. Hence, we often choose to sacrifice precision and completeness to keep our model as simple as possible.

A. Algorithmic model

A key characteristic of our approach is that, instead of predicting the scalability of the whole program, it considers each parallel region separately. For programs written in parallel languages like OpenMP [26], our approach can be applied naturally since parallel regions are constructs included in the program's syntax tree. In this paper specifically, we deal with parallel regions that are built using parallel `for` loops, such as the ones that can be constructed using a `#pragma omp parallel for` directive in OpenMP. Although our model is not conceptually tied to OpenMP, for the sake of brevity in the rest of the paper we assume programs written in C using OpenMP.

Algorithm 1: Algorithmic model

```
for ($i_0$ = $L_0$; $i_0$ < $U_0$; $i_0$ += $c_0$) do
    $CS_1$
    for ($i_1$ = $L_1$; $i_1$ < $U_1$; $i_1$ += $c_1$) do
        $CS_2$
        for ($i_n$ = $L_n$; $i_n$ < $U_n$; $i_n$ += $c_n$) do
            $CS_n$
        ...
    ...
```

To make the application of our static analysis straightforward, we place a number of restrictions on the parallel loops we consider. Algorithm 1 presents the application model subject to our analysis. The following hold:

- Any loop in the nest is potentially parallel and thus can be parallelized by the appropriate directive. Legality of parallelization and insertion of the directive is left to the programmer. A single loop in the nest is parallelized (no nesting parallelism is supported).
- Each loop body is a compound statement CS_i which can include either a `for` loop (implying recursive definition of our model), or a set of instructions that do not affect the control flow, including arithmetic operations, loads and stores, and excluding conditional or unconditional jumps, function calls, `return` statements, etc.
- The loops are in canonical form as required by the OpenMP standard [26]. This restriction implies that it is possible to determine the iteration space of each loop before entering it. Hence, the runtime system is able to calculate the number of iterations for each loop before its execution.
- Loop bodies are free of critical sections and synchronization operations in general.
- Loop bodies contain operations on regular data structures (e.g. multidimensional arrays). We assume that the size of the arrays is known at runtime.

Although our model might seem quite restrictive, large classes of parallel applications especially in scientific computing involve code segments that follow the aforementioned model. In particular, the model can embrace the large majority of loop constructs in the Polybench [29] and NAS [3] suites. Moreover, such parallel regions are typically prone to scalability limitations due to memory bandwidth saturation. Hence, despite the simplifications, we were able to evaluate our approach on a large number of parallel loops from the Polybench suite (see Section III).

B. On-chip to off-chip ratio

We use a simple hardware model: we assume an UMA architecture where all cores access a single memory node, and a single cache between each core and the memory. In par with most hardware implementations, we assume a write-back cache policy. Finally we set the cache size to the size of the last level cache (LLC).

Since we are concerned with scalability problems caused by multiple cores accessing the shared memory node, we distinguish between two operation classes: *on-chip* operations, i.e. ALU operations and data accesses serviced by the memory hierarchy up to (and including) the LLC, and *off-chip* operations, i.e. operations which include memory accesses that need to be serviced by main memory. A key observation inspired by the roofline model [33] is that parallel regions dominated by on-chip activity exhibit good scalability, while the opposite holds for regions dominated by off-chip activity. In an attempt to quantify this observation, we calculate a score for each class: a "goodness" score (S_g) for scalable operations, and a "badness" score (S_b) for non-scalable operations:

$$S_g = \sum_{o \in O_{\text{on}}} n_o \cdot w_o \qquad S_b = \sum_{o \in O_{\text{off}}} n_o \cdot w_o$$

In the above formulas, O_{on} (O_{off}) is the set of on-chip (off-chip) operations, n_o is the number of operations o in the loop, and w_o is the weight of operation o. The weights represent the effect of each operation on the score. For example, expensive arithmetic operations (e.g. division) have a greater impact on S_g than fast operations (e.g. addition). O_{on} consists of arithmetic operations (addition, multiplication, division, etc) and memory accesses that result to cache hits, while O_{off} consists of memory accesses that result to cache misses. To account for their different effect in scalability, we differentiate between streaming and latency misses, as well as read and write misses. Streaming misses, especially with prefetchers employed by modern hardware, tend to create ample memory traffic restraining scalability. On the other hand, misses with increased latency, although having cost at execution time, lead to less memory traffic and disrupt scalability to a lesser degree. Tables I and II show the weights for both machines (Dunnington and Sandy Bridge) used in our evaluation (see Section III). At the moment, the weights are specified based on empirical observations, being inline however with the manufacturer's specifications. Future work involves the calculation of these weights using a set of artificial microbenchmarks.

operation	weight
addition/subtraction	1
multiplication	2
division	4
cache hit	3

TABLE I. ON-CHIP OPERATIONS WEIGHTS FOR DUNNINGTON AND SANDY BRIDGE

	streaming	latency
read	2	1
write	3	1

TABLE II. OFF-CHIP OPERATIONS WEIGHTS FOR DUNNINGTON AND SANDY BRIDGE

For each parallel region in the application under analysis we calculate the on-chip to off-chip ratio $S_r = \frac{S_g}{S_b}$. To simplify our approach, compound statements at levels higher than the innermost one are considered to make a negligible contribution to the region's activity, and thus, are ignored. This could lead to mispredictions for cases where substantial computation load is located in outer loops. We need to point out, however, that we faced no such case in our experiments so far, so were not motivated to be more elaborate in this aspect. Yet, we can easily extend our approach to analyze computations everywhere in the nest. Moreover, the total S_r of a compound statement is calculated by considering contributions across all statements. Next we discuss our approach for classifying memory accesses as on-chip or off-chip.

C. Classifying memory accesses

Our main challenge is to classify memory accesses. To this direction, we follow the strategy described below:

- For each array reference in the compound statement we calculate its reuse distance [12], i.e. the number of unique memory locations accessed between a pair of accesses to a particular data item. If the reuse distance is smaller than the LLC size, the reference is considered a hit.
- If the reuse distance is larger than the LLC size we consider the accesses to be a potential miss with a probability p which is:

$$p = \begin{cases} 0, & \text{ws} \le \text{cs} \\ \frac{ws-cs}{ws}, & \text{ws} > \text{cs} \end{cases} \quad (1)$$

where `ws` is the working set size and `cs` the LLC size. The working set size is calculated as the sum of the sizes of the arrays. In this case we calculate the expected value of the weight, i.e. we multiply the probability with the weight.
- If the fastest changing dimension of the array is indexed using the innermost loop index, we characterize a miss as streaming, otherwise we characterize it as a latency miss. Accesses to scalar variables are ignored.

Calculating the S_r ratio is performed with a compile and runtime synergy. Static analysis is responsible for calculating S_r as a function of the ws. The runtime system only feeds the actual value of the ws parameter for the running instantiation of the program, to calculate the final value of S_r.

Example 1: We illustrate our classification using Listing 1 as an example, which is the nested loop at line 72 of the adi benchmark taken from Polybench. The second loop is parallelized. Arrays `A`, `B` and `X` are accessed more than once and there is a probability that some parts of them will remain in the LLC, thus we use Equation (1). Assuming execution on the Dunnington machine (see Section III for more details), cs $= 64$MiB. If arrays store `double` types and have dimensions 4000×4000, then ws $= 3 \cdot 8 \cdot 4000^2$ and $p = 0.825$.

```
for (t = 0; t < _PB_TSTEPS; t++) {
  #pragma omp parallel for
  for (i1 = 0; i1 < _PB_N; i1++)
    for (i2 = 1; i2 < _PB_N; i2++) {
      X[i1][i2] = X[i1][i2] - X[i1][i2-1]
                  * A[i1][i2] / B[i1][i2-1];
      B[i1][i2] = B[i1][i2] - A[i1][i2]
                  * A[i1][i2] / B[i1][i2-1];
    }
}
```

Listing 1. adi line 72

We calculate the "good" and "bad" scores for each statement separately and sum them to calculate the final score. The first statement includes 3 streaming read misses since `X[i1][i2]`, `A[i1][i2]` and `B[i1][i2-1]` are indexed by the innermost index variable, and one write miss when writing `X[i1][i2]`. The access to `X[i1][i2-1]` is considered a hit, as its reuse distance is smaller than the LLC cache size, and thus it will be served by the cache. Using the operation weights above, with *Misses* = 4, *Hits* = 1, and the formulas:

$$\begin{aligned} S_g =& Hit\_Weight * ((1-p) * Misses + Hits) + \\ & \sum_{i \in Ar\_Inst} Weight_i \cdot Occurrences_i \end{aligned}$$

$$\begin{aligned} S_b =& W_{Stream\_Read\_Miss} * p * Stream\_Read\_Miss + \\ & W_{Stream\_Write\_Miss} * p * Stream\_Write\_Miss + \\ & W_{Latency\_Read\_Miss} * p * Latency\_Read\_Miss + \\ & W_{Latency\_Write\_Miss} * p * Latency\_Write\_Miss \end{aligned}$$

we have a good score of 12.1, a bad score of 7.43. Using the same procedure for the second statement, we have 1 streaming write miss and 4 hits, since we reuse data from the previous statement. This way, the good score for this statement is 19.52 and the bad is 2.48. Thus, the total score of the loop is $S_r = \frac{12.1+19.52}{7.43+2.48} = 3.19$.

Example 2: The example of Listing 2, is taken from the 2mm benchmark (line 83) of Polybench. The outer loop is parallelized. In this case, arrays `tmp` and `A` are considered to be found in the cache, since each element of `tmp` is reused *_PB_NK* times and each row of `A` is reused *_PB_NJ* times (we assume that a row can fit in the biggest cache available). This way, since they are not being referenced using only the 2 innermost indices of the nested loops, their reuse distance is 0. Meanwhile, array `B` is considered to be a *latency* miss, since it is being referenced using the 2 innermost indices of the nested loops, which means that the whole of the array is referenced before any reuse takes place and also it is referenced by columns, rather than by rows. This miss also is under a probability, since the array is referenced multiple times in total. Statement `tmp[i][j] = 0;` is ignored, since it is not included in the innermost loop. Using the above formulas, we have a good score of 14.52, a bad score of 0.89 and a ratio of 17.60 for the loop.

```
#pragma omp parallel for
for (i = 0; i < _PB_NI; i++)
  for (j = 0; j < _PB_NJ; j++) {
    tmp[i][j] = 0;
    for (k = 0; k < _PB_NK; ++k)
     tmp[i][j] += alpha * A[i][k] * B[k][j];
  }
```

Listing 2. 2mm line 83

D. Scalability prediction

We base our prediction on the $S_r = S_g/S_b$ ratio. Intuitively, the higher the ratio, the better the scalability of the parallel loop. To map S_r to the maximum speedup of a loop σ, we assume that σ is a piecewise function of S_r as follows:

$$\sigma(S_r) = \begin{cases} \alpha \cdot S_r + \beta & S_r \leq \text{sp} \\ \sigma_{max} & S_r > \text{sp} \end{cases}$$

That is, we assume that σ is a linear function of S_r up to a split point (sp) where maximum speedup σ_{max} is reached (typically equal, or almost equal to the number of available cores). The parameters α, β and sp of σ are hardware specific, and are calculated using linear regression over the maximum speedup of a set of training loops (see Section III for more details).

III. Evaluation

In this section we provide evaluation results for our prediction mechanism. We need to test the viability of our predictor by checking the correlation of S_r with the actual speedup, and, more importantly, the ability of our scheme to predict the speedup with some acceptable accuracy.

1) Data set: We performed our evaluation using the 60 loops of the Polybench [29] suite. We had to discard 5 loops because they violated the restrictions of our model regarding the structure of the loop nest. Yet, these loops could be easily embraced by our model with straightforward extensions (left for future work). We had to also discard 6 loops that could not be legally parallelized. Finally, we also excluded 9 loops with trivial loads (basically initialization loops) that would bias our experimental analysis, and thus worked on 40 loops in total. We experimented with two data set sizes, one much larger than the LLC size and one almost equal to the LLC size. We parallelized the loops manually.

We split the loops into two sets: a *training* set consisting of 11 loops, used to calculate parameters α, β and sp (see Table III), and a *testing* set containing the remainder 29 loops used to evaluate the approach (see Table IV). We selected a training set covering a wide range of score ratios S_r.

Benchmark	Line	S_r (Dunnington)
3mm	99	15.2
atax	73	12.0
doitgen	68	20.0
fdtd-apml	130	3.7
fdtd-apml	139	2.4
gemm	77	17.6
gemver	92	3.6
gemver	96	11.0
gemver	103	7.0
reg_detect	79	0.8
syr2k	79	8.8

TABLE III. Polybench Training Set Loops

Benchmark	Line	S_r (Dunnington)
2mm	83	17.6
2mm	92	15.2
3mm	79	15.2
3mm	89	15.2
adi	72	3.2
adi	80	0.6
adi	84	1.1
adi	96	0.6
atax	70	6.0
covariance	68	7.0
covariance	78	0.8
covariance	84	17.3
dynprog	69	2.0
fdtd-2d	88	2.3
fdtd-2d	93	1.0
fdtd-apml	153	20.0
fdtd-apml	166	8.8
floyd-warshall	63	1.4
gemver	102	3.6
gesummv	77	6.0
jacobi-2d-imper	71	0.8
lu	66	1.4
lu	71	1.8
mvt	76	6.0
mvt	81	12.0
reg_detect	80	0.8
syr2k	77	1.7
syrk	72	1.7
syrk	77	10.0

TABLE IV. Polybench Testing Set Loops

2) Hardware Platforms: For the evaluation of our prediction model, we used two hardware platforms. We took measurements using high precision timer, based on cpu ticks, which dictated that measurements should be taken on the same cpu before and after a parallel region.

a) Dunnington: The first platform is a 24-core Dunnington-based node with the following characteristics (see also Figure 1): 4 physical packages, 6 cores per package, 32 KB L1 cache per core, 3 MB L2 cache shared by 2 cores, 16 MB L3 cache per package and 8 GB RAM. This platform was running an x86_64 version of Debian Squeeze (6.0.7), with version 3.7.10-1 of the Linux kernel. Programs were compiled with gcc version 4.6.3 (implementing version 3.0 of the OpenMP standard), with the -O3 optimization flags. Averages of three runs are presented.

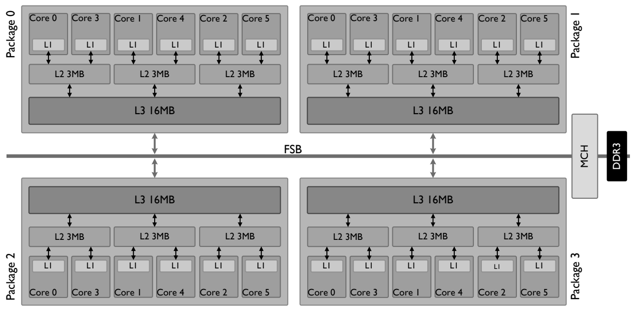

Fig. 1. Dunnington Layout

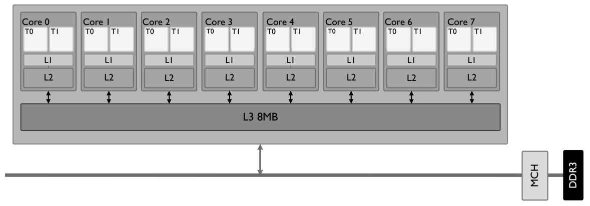

Fig. 2. Sandy Bridge Layout

b) Sandy Bridge: The second platform is a 8-core NUMA Sandy Bridge node with the following characteristics (see also Figure 2): 1 physical package, 8 cores per package, 16 KB L1 cache per core, 256 KB L2 cache per core, 16 MB L3 cache per package and 256 GB RAM. Operating system and compiler version/flags are the same with Dunnington.

3) Configuration: A specific configuration was used for the OpenMP runtime, using environmental variables and pre-processor directives to control the assignment of threads in available cores, the affinities of threads and the scheduling of parallel regions. More specifically:

- OpenMP threads were assigned to cores in a way that they use different L2 caches when possible (when not all cores are used) while staying in the same package, since we want to apply our model and measure its success in predicting utilization in a per-package level. Also, this way we benefit from cross-processor reuse of data, where this is possible.
- Wherever possible, static scheduling was used to avoid thread management and scheduling overheads. In any other case, scheduling was set to dynamic.

A. Correlation of S_r score with Speedup

In this set of experiments, our intention is to verify whether the core of our prediction scheme, i.e. the S_r score ratio has a good correlation with the actual region speedup. Figures 3 and 4 plot the speedup achieved by all benchmarking loops

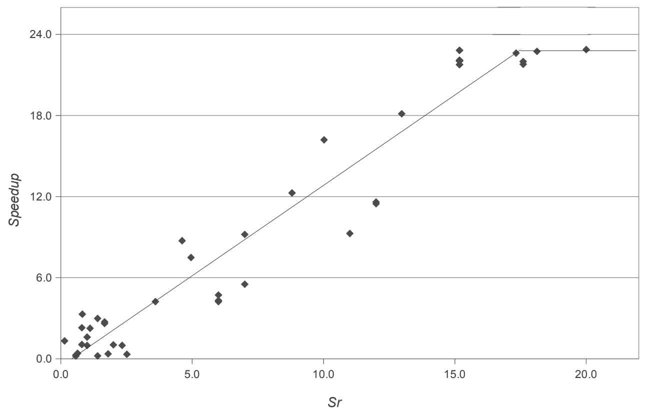

Fig. 3. Speedup as a function of S_r score for Dunnington for large data sets not fitting the LLC

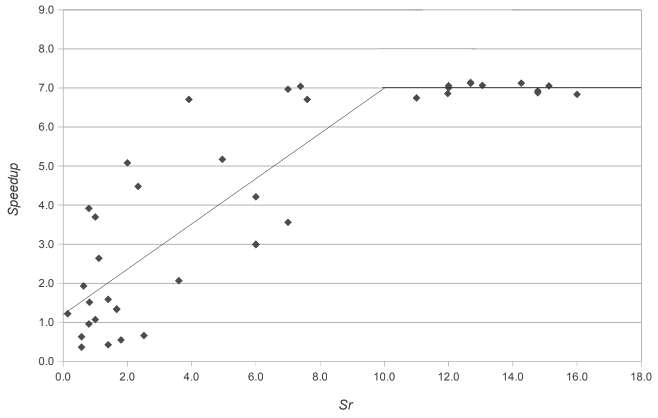

Fig. 4. Speedup as a function of S_r score for Sandy Bridge for large data sets not fitting the LLC

(both training and testing) as a function of S_r for the large data sets. We can make the following remarks: a) The trend of the speedup is successfully captured by S_r, as it succeeds a 95.7% correlation in Dunnington and a 84.1% correlation in Sandy Bridge, b) The two regions in the piecewise function discussed in Section II are visible as indicated by the trend lines in the graphs and c) in the two extremes of very low or very high values for S_r, the landscape seems more clear, especially in Dunnington.

B. Scalability prediction

In this set of experiments we compare the predicted speedup provided by our mechanism with the maximum speedup and the speedup achieved by the full utilization of the machine cores. Figures 5 – 8 provide this information for the loop regions of the testing set in the Dunnington and Sandy Bridge machines respectively. Apart from the relative error, we also employ as prediction error the metric

$$\frac{|\text{predicted speedup} - \text{maximum speedup}|}{\text{maximum number of cores}}$$

expressed in $\%$ which normalizes the absolute error in speedup prediction to the range of possible speedup values (no super-linear speedup is considered). Table V summarizes relative and

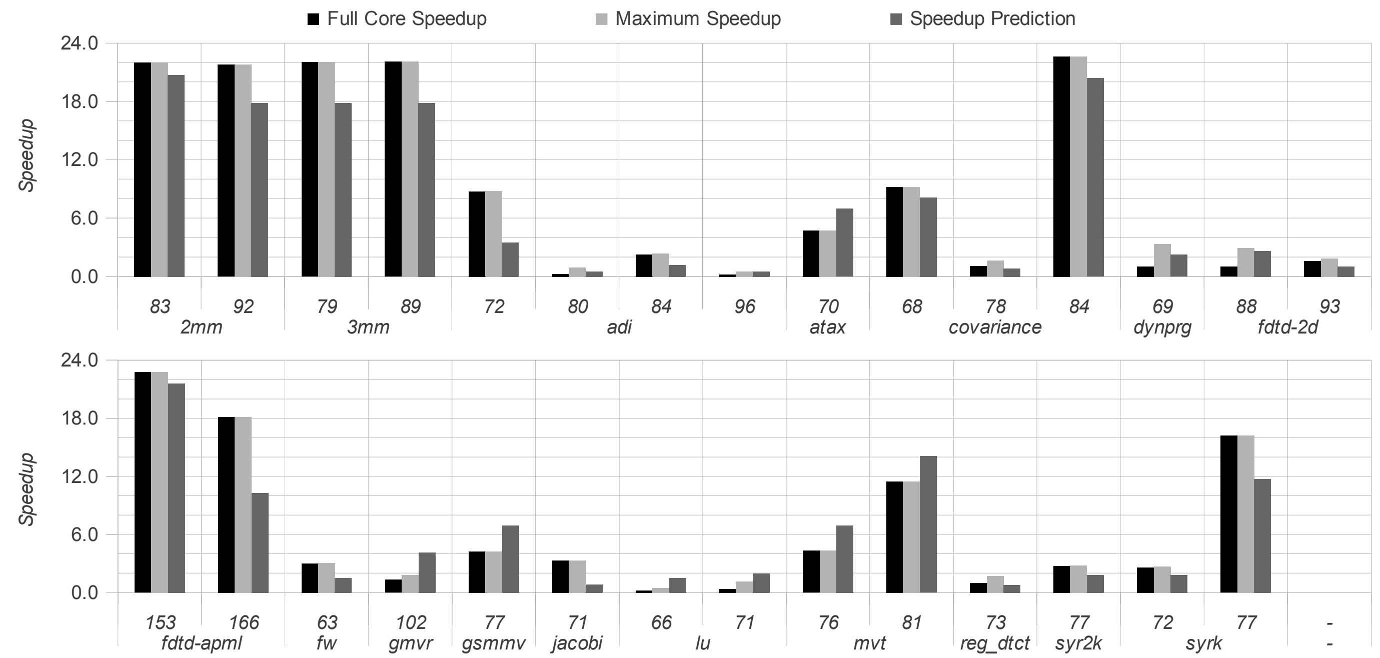

Fig. 5. Dunnington speedups for large data sets not fitting the LLC

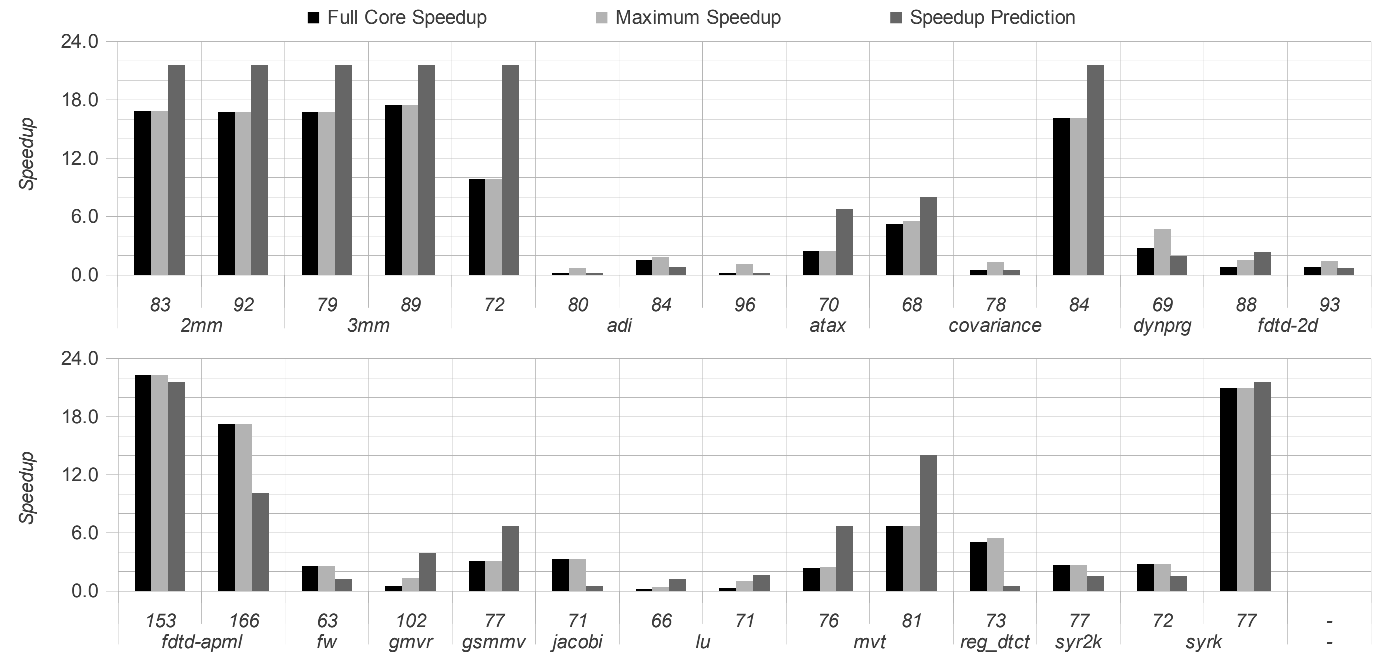

Fig. 6. Dunnington speedups for small data sets fitting the LLC

prediction errors for the four experimental scenarios. Although relative errors are quite high, we argue that the prediction error is a better metric to assess the accuracy for a scalability prediction model, as for example a prediction of speedup 3 in a 24-core machine compared to an actual speedup of 1, would give a relative error of 200%, although the predictor was able to capture the problematic scalability of the code.

In Dunnington (where the prediction ranges between 0 and 24) with the large data set (Figure 5) our scalability predictor had an average (maximum) prediction error of 8.8% (32.7%, for fdtd-apml, line 166 loop, where our scheme predicted a speedup of 10.3, while the maximum speedup was 18.1). Regarding the small data set, our predictor had an average (maximum) prediction error of 12.9% (48.9%, for adi, line 72 loop, where our scheme predicted a speedup of 21.6, while the maximum speedup was 9.8, actually failing to provide an acceptable prediction for this case).

In Sandy Bridge (where the prediction ranges between 0 and 8) with the large data set, the average (maximum) prediction error was 15.3% (46.7%, for adi, line 72 loop,

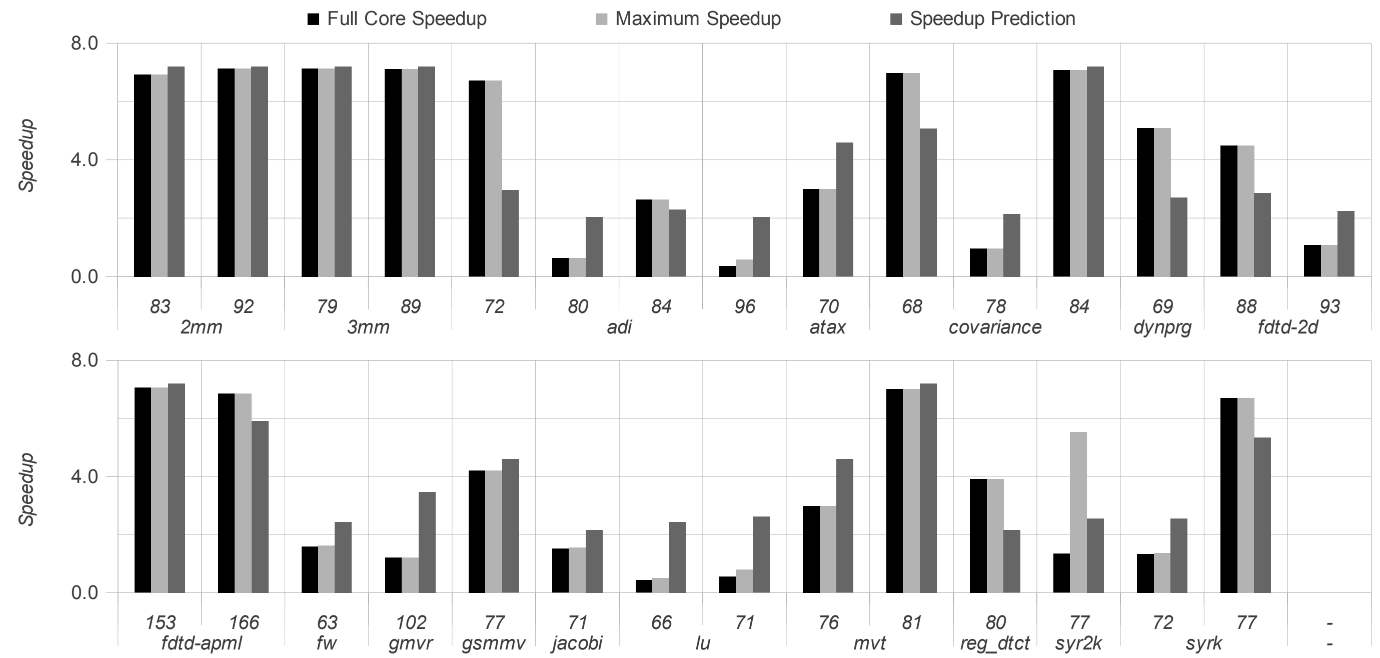

Fig. 7. Sandy Bridge speedups for large data sets not fitting the LLC

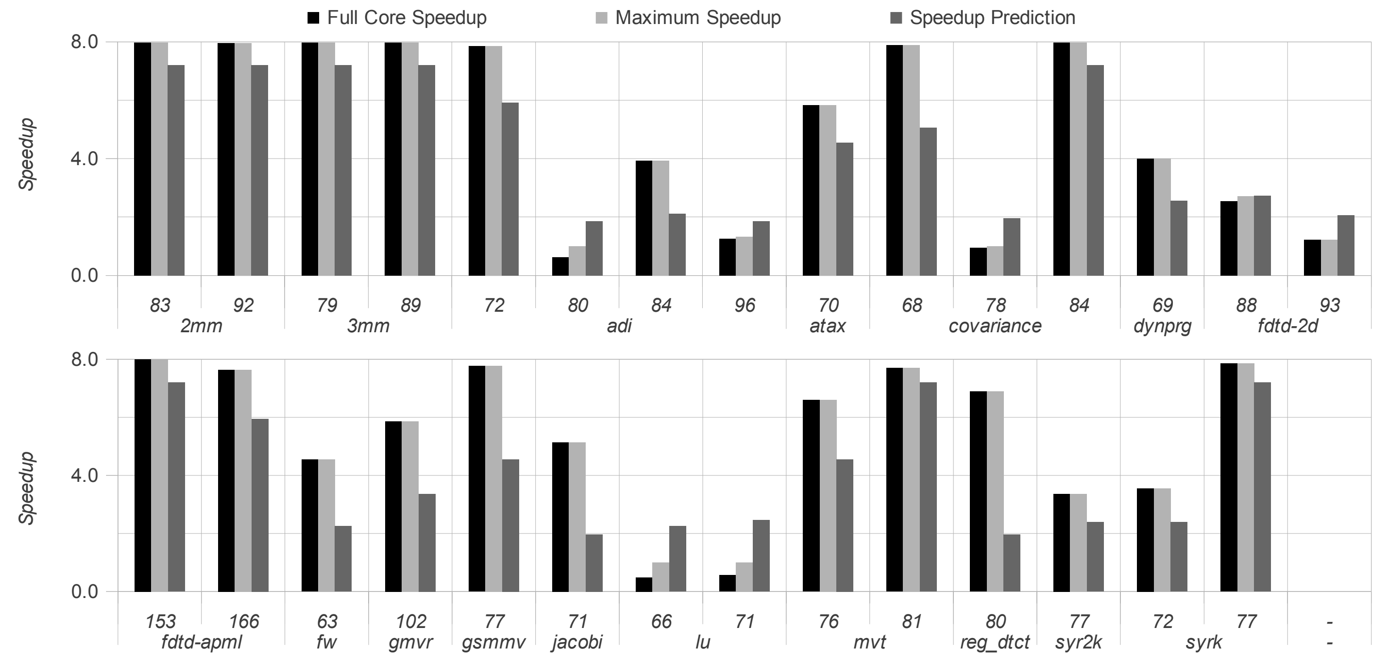

Fig. 8. Sandy Bridge speedups for small data sets fitting the LLC

where our scheme predicted a speedup of 3.0 while the maximum speedup was 6.7). For the small data set, the average (maximum) prediction error was 18.5% (61.6%, for reg_detect, line 80 loop, where our scheme predicted a speedup of 2.0 while the maximum speedup was 6.9). Note that in the large majority of the cases the full core speedup and maximum speedup are either the same or very close together. In a few cases there is a significant difference between them, as for example in Sandy Bridge, syr2k, line 77 loop. Our model is not trained to capture such behavior and thus presents this large absolute error in this case. When compared to the speedup with full core utilization, the prediction error drops to 16.0%.

Platform	Data set	prediction error (%)			relative error (%)		
		min	max	avg	min	max	avg
Dunnington	large	0.1	32.7	8.8	4.3	232.5	45.6
	small	2.0	48.9	12.9	3.1	193.5	72.8
Sandy Bridge	large	3.4	46.7	15.3	1.0	389.2	73.8
	small	0.3	61.6	18.5	1.0	145.9	40.7

TABLE V. PREDICTION ERRORS

We believe that the accuracy of our model is acceptable for the purpose that it was designed, i.e. to facilitate the decisions of a resource-aware allocation framework. Our model is capable of capturing the "big picture" of scalability of a parallel region, as it successfully follows the trend in all loops. Qualitatively, if utilized to classify a parallel loop region as "bad performing" (e.g. achieves a speedup close to or less than 25% of the number of the available cores), "fair performing" (e.g. achieves a speedup roughly between 25% and 75% of the number of the available cores), or "good performing" (e.g. achieves a speedup close to or more than 75% of the number of the available cores), it will provide a reliable prediction. In that perspective, in 116 experiments involving two architectures and two data set sizes, 101 predictions had an error smaller than 25%. Yet, in the intermediate class of fair performing loops, a better accuracy could be anticipated. Fine-tuning our model to better predict scalability behavior in this case is left for future work.

Finally, one interesting remark is that our model makes better predictions when the system and data set size lead to memory bound execution scenarios, as is the case of the configuration in Dunnington with the large data set, while the prediction error increases when we use data set fitting the LLC and switching to Sandy Bridge that has a memory interconnect with higher bandwidth. This is rather expected as our model is designed to work under the assumption of memory bandwidth limitations.

IV. Related work

Performance prediction, especially in the context of parallel computing, has traditionally been an interesting topic, due to its wide applicability. From the traditional performance prediction models like Amdahl's law [2], Gustavson's law [17] and the LogP model [10], to the more recent ones discussed next, researchers have tried to capture the behavior of parallel programs to meet a variety of objectives, including the efficient use of resources, the design of new architectures, system procurement, the decision of whether it would pay off to go parallel or optimize a specific application, etc. Depending on their objective, prediction models may trade accuracy for speed or vice versa, or provide lower and/or upper bounds to guide the relevant decisions. When the prediction accuracy is not satisfactory, or the size of the parameter space is too large, autotuning is a viable alternative [16], [32].

Performance prediction was a significant component of automatic parallelizing compilers for massively parallel distributed-memory machines back in the 90's [15], [27]. The same holds for MPI programs where several extensions of the LogP model have been proposed [1], [9], [18]. The key performance bottleneck that needs to be captured in the above cases is the communication overhead, a challenge that still remains open in today's supercomputers consisting of multicore nodes [19]. Our approach targets inter-node performance bottlenecks and can be combined with prediction models focusing on the communication overhead.

With a goal to predict scalability of applications in large-scale, HPC systems, Barnes et al. [4] present a regression-based approach that correlates scalability to input variables. Their method is applied offline, and targets large-scale systems. Lee et al. [23] compare the effectiveness of piecewise polynomial regression and artificial neural networks (ANNs) to predict performance in the context of varying input. To facilitate system architecture decisions, the Phantom framework uses execution traces from a single node to predict application performance in large scale environments.

A number of research papers focus on predicting the behavior of applications that suffer from contention in the memory hierarchy. Within the context of co-existing applications in the same multicore platform, researchers work on the critical properties of contentiousness (how much an application's execution affects other applications) and sensitivity (how much an application is affected by the execution of other applications [31], [34]. Cache Pirate [13] and Bandwidth Bandit [14] provide information on the slowdown an application may suffer when the LLC and memory bus respectively operate under contention.

Our work is inspired by the roofline model [33]. Roofline incorporates the significant performance gap between the CPU and memory with a goal to capture the performance of memory-bound applications as well. The model utilizes two architectural characteristics, peak CPU performance and memory bandwidth, and one application characteristic, the flop/byte ratio, to predict application performance. However, although memory bandwidth can be rather accurately measured (e.g. with a benchmark like STREAM [24]), peak CPU power depends on the mixture of computations performed by the application, and thus to some extent is application-dependent, while the flop/byte ratio cannot be easily determined since one needs to decide whether memory accesses are serviced by DRAM or the cache hierarchy. For these reasons, the roofline model cannot be applied directly to calculate multithreaded scalability, especially at compile time where important runtime information about the overall working set is absent. Our work refines upon the roofline model, and bases its prediction on an "on-chip to off-chip activity" ratio, analogous to the flop/byte ratio. We utilize information on the size of the LLC and the runtime size of the working set to classify memory accesses as hits or misses and do not use architectural parameters like peak CPU performance or memory bandwidth, which are implicitly incorporated in our model through the training process.

In this paper we focus on prediction models targeting multithreaded applications in CMP machines. Within this context, Kismet [20] provides speedup estimates of unmodified serial programs based on extensive offline analysis. Its goal is to provide upper bounds for scalability to support decisions regarding the benefits of a software parallelization process. The model does not account for memory bandwidth saturation, and thus fails to predict scalability limitations due to this critical factor. Parallel Prophet [21] reduces the analysis overheads and takes into account memory bandwidth saturation, but again the approach is applicable to offline decisions.

Our work is directly relevant to previous approaches elaborating on online scalability prediction to facilitate concurrency throttling. Similar to the goal of our work, The ACTOR runtime system [11], [30] seeks the optimal operating point of concurrency in multithreaded programs at the granularity of program phases dynamically, by feeding information collected by the machine's performance counters to a predictor based on regression or ANN. The phases handled by ACTOR are more generic than ours as presented in Section II-A, but its approach requires an online sampling period where different configurations need to be tested. Apart from the additional overhead of the sampling phase, a straightforward implementation of this scheme would require parallel phases to be included inside an outer sequential loop, this being a restriction for ACTOR not present in our approach, which is able to decide upon the concurrency levels prior to the execution of a parallel phase. In any case, our method can work in a complementary way to similar purely dynamic schemes, by drastically restricting the search space within their sampling period.

V. Limitations and future work

Our initial evaluation shows that our approach holds some promise since it reasonably predicts the scalability of a large class of parallel loops. Our current model, however, has a limited scope since it was developed to facilitate an initial experimental evaluation to investigate its potential. In the next paragraphs, we discuss what are the main limitations of our current model, and how we plan to address them in the future.

a) Algorithmic model: Although our algorithmic model is able to capture a large number of parallel loops, it enforces restrictions which potentially limit its applicability. One of these restrictions is that it does not allow function calls inside the body of loops. Although addressing the general case of this problem —especially if recursion is involved— is a very challenging task, we argue that a significant number of practical cases can be solved by applying interprocedural techniques already within the capabilities of modern compilers such as gcc and clang/LLVM.

Another important programming construct that is not currently supported is synchronization primitives such as the OpenMP critical construct (`#pragma omp critical`). Our current model is not able to deal with these constructs because it only considers memory contention as the source of scalability problems. We plan to address this issue by extending our model to support multiple classes of "badness" and "goodness" scores, one per each potential scalability bottleneck. Using the same methodology, we will end up with multiple ratios that will need to be considered for our prediction. A simple approach would be to select the ratio that causes the greatest scalability obstruction. Furthermore, future work involves relaxing some of our restrictions to include nested parallelism, operations on irregular data structures, affine functions of the enclosing loop iterators as loop bounds, and generally increase the coverage of our predictor.

b) Hardware model: Modern multicore hardware exhibits high diversity. Multicore systems include complex cache hierarchies where caches are shared between some cores, support multiple hardware contexts in the same core (SMT), and employ NUMA techniques to allow for scalable memory access. On one hand, a detailed hardware model can lead to more precise predictions, but on the other hand it significantly complicates the parameter extraction using training and the overall prediction. Hence, the main challenge is to find the minimal model that can lead to adequate predictions. We plan to tackle this problem using a workload driven approach. That is, rather than fully modeling the current hardware, use experimental analysis to determine which are the factors that have the greater impact and focus only on them.

c) Improving prediction accuracy: Experimental results demonstrated that our model is able to capture the general scalability trends but may face problems in cases where its scoring S_r falls between the range of low and high score. Prediction accuracy in this case could be improved by elaborating more on the model and investigating the actual reasons for mispredictions.

d) Predicting core assignment: Our current model predicts the maximum speedup of parallel loops. To maximize its utility, however, we also need to find the minimum number of cores that maximize the parallel speedup. This information can be communicated from the parallel runtime system to the OS scheduler, so that the latter can make appropriate scheduling decisions regarding the number of cores assigned to each parallel loop. A simple approach would be to assume linear scalability until the loop reaches the "roof", where it remains constant. In other words, if σ is the maximum speedup of a loop, assume that assigning $\lceil \sigma \rceil$ cores will result in a speedup of σ. As it is expected, however, parallel loops rarely exhibit linear scalability and according to our initial experimental results this approach underestimates the number of cores needed to reach the maximum speedup. Hence, as future work we aim to exploit additional information from the training set execution, i.e. not just the maximum speedup but also the behaviour of each loop for different core assignments, for predicting the minimum number of cores required to achieve maximum speedup.

e) Implementation: Finally, our current model is applied manually and some of its parameters (i.e. operation weights) are determined empirically. Our goal is to build a static compiler analysis (e.g. using a gcc plugin) which, along with the appropriate runtime functionality, will be able to apply our model automatically in parallel OpenMP loops. We plan to start with a simple implementation that supports what we describe here and gradually extend it to support additional parallel loops taken from benchmarks and real-world applications.

VI. Conclusions

In this paper we worked towards the implementation of a synergistic scalability predictor for parallel regions. We performed an initial evaluation of a mechanism that utilizes both compile and runtime information to predict the maximum speedup of a parallel loop construct. Our approach is based on the formulation of an on-chip to off-chip activity ratio that is associated with the application's speedup on a specific machine through the use of linear regression. Our model is trained with a limited set of loop constructs and evaluated on a wider set of loops over two CMP machines. Evaluation results show a promising performance of our prediction approach that could be further utilized to assist a resource-aware allocation policy.

Acknowledgment

This research was carried out within project I-PARTS: "Integrating Parallel Run-Time Systems for Efficient Resource Allocation in Multicore Systems" (code 2504) of Action ARISTEIA, co-financed by the European Union (European Social Fund) and Hellenic national funds through the 'Operational Program Education and Lifelong Learning' (NSRF 2007-2013).

References

[1] Albert Alexandrov, Mihai F Ionescu, Klaus E Schauser, and Chris Scheiman. LogGP: Incorporating long messages into the logP model one step closer towards a realistic model for parallel computation. In *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pages 95–105. ACM, 1995.

[2] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks - Summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, pages 158–165, New York, NY, USA, 1991. ACM.

[4] Bradley J. Barnes, Barry Rountree, David K. Lowenthal, Jaxk Reeves, Bronis de Supinski, and Martin Schulz. A regression-based approach to scalability prediction. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 368–377, New York, NY, USA, 2008. ACM.

[5] Major Bhadauria and Sally A. McKee. An approach to resource-aware co-scheduling for CMPs. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 189–199, New York, NY, USA, 2010. ACM.

[6] Major Bhadauria, Vincent M. Weaver, and Sally A. McKee. Understanding PARSEC performance on contemporary CMPs. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 98–107, Washington, DC, USA, 2009. IEEE Computer Society.

[7] Sergey Blagodurov, Sergey Zhuravlev, and Alexandra Fedorova. Contention-aware scheduling on multicore systems. *ACM Trans. Comput. Syst.*, 28(4):8:1–8:45, December 2010.

[8] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[9] WenGuang Chen, JiDong Zhai, Jin Zhang, and WeiMin Zheng. LogGPO: An accurate communication model for performance prediction of mpi programs. *Science in China Series F: Information Sciences*, 52(10):1785–1791, 2009.

[10] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: towards a realistic model of parallel computation. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '93, pages 1–12, New York, NY, USA, 1993. ACM.

[11] Matthew Curtis-Maury, Filip Blagojevic, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Prediction-based power-performance adaptation of multithreaded scientific codes. *IEEE Trans. Parallel Distrib. Syst.*, 19(10):1396–1410, October 2008.

[12] Chen Ding and Y Zhong. Reuse distance analysis. Technical report, Rochester, NY, USA, 2001.

[13] David Eklov, Nikos Nikoleris, David Black-Schaffer, and Erik Hagersten. Cache pirating: Measuring the curse of the shared cache. In *ICPP*, pages 165–175, 2011.

[14] David Eklov, Nikos Nikoleris, David Black-Schaffer, and Erik Hagersten. Bandwidth bandit: Quantitative characterization of memory contention. In *CGO*, pages 1–10, 2013.

[15] Thomas Fahringer, Roman Blasko, and Hans P. Zima. Automatic performance prediction to support parallelization of Fortran programs for massively parallel systems. In *ICS*, pages 347–356, 1992.

[16] Basilio B. Fraguela, Yevgen Voronenko, and Markus Püschel. Automatic tuning of discrete fourier transforms driven by analytical modeling. In *PACT*, pages 271–280, 2009.

[17] John L. Gustafson. Reevaluating Amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988.

[18] Roger W Hockney. The communication challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel computing*, 20(3):389–398, 1994.

[19] Torsten Hoefler, William Gropp, Rajeev Thakur, and Jesper Larsson Träff. Toward performance models of MPI implementations for understanding application scaling issues. In *Recent Advances in the Message Passing Interface*, pages 21–30. Springer, 2010.

[20] Donghwan Jeon, Saturnino Garcia, Chris Louie, and Michael Bedford Taylor. Kismet: parallel speedup estimates for serial programs. *SIGPLAN Not.*, 46(10):519–536, October 2011.

[21] Minjang Kim, P. Kumar, Hyesoon Kim, and B. Brett. Predicting potential speedup of serial code via lightweight profiling and emulations with memory performance model. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 1318–1329, 2012.

[22] Kornilios Kourtis, Vasileios Karakasis, Georgios Goumas, and Nectarios Koziris. CSX: an extended compression format for spmv on shared memory systems. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 247–256, New York, NY, USA, 2011. ACM.

[23] Benjamin C. Lee, David M. Brooks, Bronis R. de Supinski, Martin Schulz, Karan Singh, and Sally A. McKee. Methods of inference and learning for performance modeling of parallel applications. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '07, pages 249–258, New York, NY, USA, 2007. ACM.

[24] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[25] Andreas Merkel, Jan Stoess, and Frank Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 153–166, New York, NY, USA, 2010. ACM.

[26] OpenMP application programming interface (ver. 3.1). http://www.openmp.org/mp-documents/OpenMP3.1.pdf, 2011.

[27] Manish Parashar and Salim Hariri. Compile-time performance prediction of HPF/Fortran 90D. *IEEE Parallel Distrib. Technol.*, 4(1):57–73, March 1996.

[28] Simon Peter, Adrian Schüpbach, Paul Barham, Andrew Baumann, Rebecca Isaacs, Tim Harris, and Timothy Roscoe. Design principles for end-to-end multicore schedulers. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, HotPar'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[29] PolyBench/C: the polyhedral benchmark suite. http://www.cse.ohio-state.edu/ pouchet/software/polybench/, 2012.

[30] Karan Singh, Matthew Curtis-Maury, Sally A. McKee, Filip Blagojević, Dimitrios S. Nikolopoulos, Bronis R. de Supinski, and Martin Schulz. Comparing scalability prediction strategies on an SMP of CMPs. In *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part I*, EuroPar'10, pages 143–155, Berlin, Heidelberg, 2010. Springer-Verlag.

[31] Lingjia Tang, Jason Mars, and Mary Lou Soffa. Contentiousness vs. sensitivity: improving contention aware runtime systems on multicore architectures. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 12–21, New York, NY, USA, 2011. ACM.

[32] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '98, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.

[33] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[34] Yuejian Xie and Gabriel Loh. Dynamic classification of program memory behaviors in CMPs. In *Proceedings of the 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects*, 2008.