Schedule-Aware Loop Parallelization for Embedded MPSoCs by Exploiting Parallel Slack

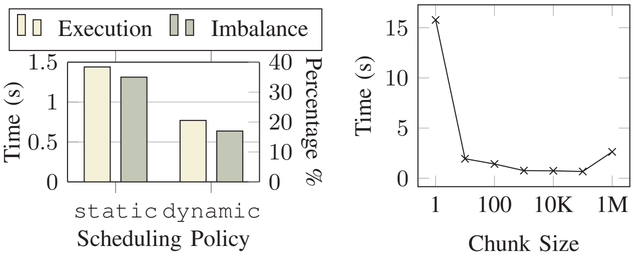
Miguel Angel Aguilar*, Rainer Leupers*, Gerd Ascheid*, Nikolaos Kavvadias†, Liam Fitzpatrick†
*Institute for Communication Technologies and Embedded Systems, RWTH Aachen University, Germany
{aguilar, leupers, ascheid}@ice.rwth-aachen.de
†Silexica Software Solutions GmbH, Germany
{kavvadias, fitzpatrick}@silexica.com

***Abstract*—MPSoC programming is still a challenging task, where several aspects have to be taken into account to achieve a profitable parallel execution. Selecting a proper scheduling policy is an aspect that has a major impact on the performance. OpenMP is an example of a programming paradigm that allows to specify the scheduling policy on a per loop basis. However, choosing the best scheduling policy and the corresponding parameters is not a trivial task. In fact, there is already a large amount of software parallelized with OpenMP, where the scheduling policy is not explicitly specified. Then, the scheduling decision is left to the default runtime, which in most of the cases does not yield the best performance. In this paper, we present a schedule-aware optimization approach enabled by exploiting the parallel slack existing in loops parallelized with OpenMP. Results on an embedded multicore device, show that the performance achieved by OpenMP loops optimized with our approach outperform by up to 93%, the performance achieved by the original OpenMP loops, where the scheduling policy is not specified.**

I. Introduction

The use of *Multiprocessor Systems on Chip* (MPSoCs) is a widely accepted solution to design modern embedded devices, as they provide a good trade-off between conflicting requirements, such as performance, energy and cost. Unfortunately, MPSoC programming is still a cumbersome task. The current practice involves several manual steps: finding computationally intensive code sections, exploiting the most profitable parallelization opportunities, selecting a proper scheduling policy to balance the workload across the available cores and finally implementing the parallel code. Parallelizing compiler technologies have the potential to overcome the challenge of MPSoC programming by automating the previously described steps. In the embedded domain, multiple parallelization frameworks have been proposed [1], [2], [3]. However, these frameworks have focused mainly on identifying parallel loops without considering the scheduling policy required to properly balance the workload. Choosing the proper scheduling policy is critical to achieve significant performance improvements in loop parallelization [4].

OpenMP is a compiler directive based approach for loop parallelization that allows to define the scheduling policy as indicated by the `schedule` clause, e.g. `static` or `dynamic`. The `static` scheduling policy is typically the default one in OpenMP runtime systems [4]. In this policy, the loop iteration space is split into as many chunks as cores are available. This is a simple approach, but is not necessarily the most efficient one, especially for loops that present an uneven workload across iterations. In contrast, in the `dynamic` scheduling policy, the iteration space is split into smaller chunks to allow more flexibility for distributing iterations across cores, thus achieving a better load balancing. Moreover, the `dynamic` scheduling policy presents a good performance even in presence of external system load [5]. To show the impact of the scheduling policy on the performance, we use as a motivating example the *PathFinder* benchmark [6] running on a Nexus tablet [7]. Figure 1a shows the execution time and the load imbalance of *PathFinder* with both scheduling policies. The default `static` policy results in an execution time of 1.44s and a load imbalance of 35%, while with the `dynamic` policy and using the proper chunk size the execution time is reduced to 0.76s and the load imbalance to 17%. However, defining a proper chunk size is not a trivial task and could negatively impact the performance. As Figure 1b shows, while choosing small chunk sizes increases the overhead, choosing large chunk sizes serializes the execution.



(a) Execution Time and Imbalance

(b) Impact of Chunk Size

Fig. 1: Motivating Example: *PathFinder*

In this paper, we propose a schedule-aware optimization approach for parallel loops based on the standard OpenMP `dynamic` scheduling policy. To compute the chunk size we exploit the concept of parallel slack, which refers to the extra potential parallelism within a code region above the actual parallelism available on the target platform [8]. The main contributions of this paper are:

1) An approach for automatic schedule-aware OpenMP loop optimization for embedded MPSoCs, by exploiting the parallel slack to improve load balancing.
2) Evaluation of the approach on a commercial Android device based on a quad-core MPSoC.
3) Comparison of the optimized OpenMP pragmas by our approach with handwritten pragmas by expert programmers, where the scheduling policy is not specified.

978-3-9815370-8-6/17/$31.00 ©2017 IEEE

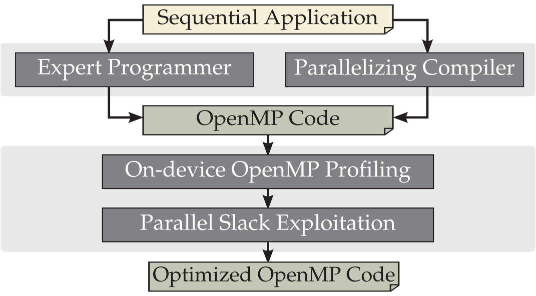


Fig. 2: Proposed Toolflow

II. Schedule Optimization Approach

Figure 2 shows the proposed toolflow. The aim is to improve the load balancing of loops parallelized with OpenMP, either by expert programmers or by parallelizing compilers. Specifying potential parallelism in applications higher than the actual parallelism available in the target MPSoC, gives the schedulers more flexibility to achieve a better load balancing. This extra parallelism is known as *parallel slack* [8]. We achieve this by *overdecomposing* their iteration space into chunks smaller than the ones created by the default `static` scheduling policy, therefore we create more parallel slack that can be exploited by the `dynamic` scheduling policy.

In order to compute the parallel slack and thus the chunk size, we use the *work-span* model [8]. In the work-span model tasks are arranged in a directed acyclic graph, where a task is ready to execute when all its predecessors are done with their work. It is also assumed that the scheduling is *greedy*, which implies that a core is never left idle if there is a task ready to be executed. The model works based on two values: the total work t_{work} and the span t_{span}. Time t_{work} refers to the total time that it would take to serially execute all tasks. Time t_{span} is the longest chain of tasks that must be executed in a sequence, which is equivalent to the *critical path*. In the work-span model super-linear speedup it is not possible, as it defines a limit shown in Equation 1:

$$Speedup = \frac{t_{work}}{t_{par}} \leq \frac{t_{work}}{t_{span}} \tag{1}$$

Figure 3 shows an example, where each box represents a task that takes one unit to be executed. Therefore, the total work (t_{work}) is 18 and the span (t_{span}) is 6, which results in an upper bound for the speedup of 3× (t_{work}/t_{span}). The span can be also used to define a lower bound to the speedup, by using Brent's Lemma [9], which bounds t_{par} in terms of the work and the span:

$$t_{par} \leq \frac{t_{work} - t_{span}}{N} + t_{span} \tag{2}$$

Equation 2 shows that the total work t_{work} can be divided into two components: *imperfectly parallelizable work* that takes t_{span} independently of the number of cores (N), and *perfectly parallelizable work* $t_{work} - t_{span}$ that can be speeded up by N. From Equation 2 we can estimate t_{par} by assuming that t_{work} is much higher than t_{span}, thus $t_{work} - t_{span} \approx t_{work}$, which is the case in computationally intensive parallel loops:

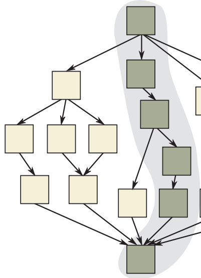


Fig. 3: Work-Span model example. Each task takes one unit of time: $t_{work} = 18$, $t_{span} = 6$

$$t_{par} \approx \frac{t_{work}}{N} + t_{span} \quad \text{if} \;\; t_{span} \ll t_{work} \tag{3}$$

From this approximation we can see that increasing the total work t_{work} impacts negatively the parallel execution, as well as increasing the span t_{span}, no matter how many cores are available. Moreover, Brent's Lemma provides a formal motivation for *overdecomposition* as follows:

$$Speedup = \frac{t_{work}}{t_{par}} \approx N \quad \text{if} \;\; \frac{t_{work}}{t_{span}} \gg N \tag{4}$$

Equation 4 says that it is possible to achieve linear speedup if a problem is overdecomposed to create more potential parallelism, which can be exploited by schedulers. This extra parallelism is the *parallel slack*, which is defined as follows:

$$Parallel\ Slack = \frac{t_{work}}{N \cdot t_{span}} \tag{5}$$

As discussed earlier, the default `static` scheduling policy in OpenMP takes a parallel loop and divides its number of iterations by the number of threads, which is typically equal to the number of cores on the platform. However, as it was shown in the motivating example in Figure 1, this simple default scheduling policy does not always provides the best performance. The reason for this is that quite often there are iterations that perform more work than others, which leads to load imbalance. Figure 4 exemplifies this, where a loop with 8 independent iterations is divided into two threads with 4 iterations each. In this example, Thread 0 has 4 iterations that take 2 time units to finish each, while Thread 1 has 4 iterations that take 1 time unit to finish each. Therefore, Thread 0 takes in total 8 time units to complete, while Thread 1 takes 4 time units. This results in a load imbalance of 25%.

To address the problem previously described, we propose to exploit the parallel slack by overdecomposing the iteration

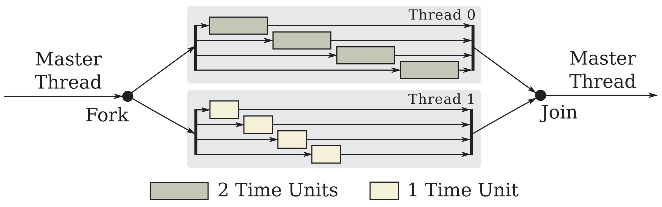


Fig. 4: Work-Span in OpenMP. $t_{work} = 12$, $t_{span} = 2$

```
R00001  file.c (50-60) PARALLEL LOOP
TID  execT  execC  bodyT  exitBarT  startupT  shutdwnT
  0   8.0     1     8.0     0.0       0.0       0.0
  1   8.0     1     4.0     4.0       0.0       0.0
SUM  16.0     3    12.0     4.0       0.0       0.0
```

Fig. 6: Simplified `ompP` Profile Example

space into smaller chunks, which are then distributed to threads by the `dynamic` scheduling policy. We estimate the chunk size in terms of the parallel slack and the load imbalance observed with the default `static` policy:

$$Chunk = \left\lceil \frac{t_{work}}{N \cdot t_{span}} \cdot (1 - Imbalance) \right\rceil \qquad (6)$$

On one hand, this estimation says that the higher the load imbalance is, the smaller is the chunk size and then the schedule is more dynamic. On the other hand, the lower the load imbalance is, the bigger is the chunk size and then the schedule is more static. For the example in Figure 4, the resulting chunk size is 3, which reduces the execution time of the slowest thread from 8 to 6 time units and the load imbalance from 25% to 0%, as now each thread has the same workload.

In order to get the values to compute the chunk size, we perform an OpenMP profiling run on the target embedded platform, as Figure 2 shows. For this purpose we use `ompP`, which is a profiling tool for OpenMP applications [10]. Applications are linked to the `ompP` library and a profiling report is generated on program termination. Figure 6 shows an example of the profile corresponding to the example in Figure 4. This report shows the source level location of the loop. Also, each line displays data for a particular thread (`TID`) and the last line shows the totals (`SUM`). Column names ending with `C` represent counts, while columns ending with `T` represent execution times. `bodyT` is the actual work of the threads. From this value we derive the total work t_{work} and the span t_{span} by dividing the execution time of the slowest thread over the iteration count assigned to that thread. `exitBarT` is the time that the threads wait at the exit barrier when they are done with their work. From this value we derive the percentage of load imbalance. Additional values are provided, such as the `startupT` and `shutdwnT`, which are the times to create and to destroy the threads of a parallel region, respectively. Finally, we generate OpenMP pragmas optimized with the `schedule` clause (last step in Figure 2). If the load imbalance is higher than a fixed threshold, the `dynamic` policy is used, otherwise the `static` policy is used.

III. Experimental Evaluation

This section describes the experimental evaluation of the proposed optimization approach.

A. Experimental Environment

The Nexus 7 Android-based tablet was selected as the target platform [7]. It is based on a Krait quad-core Snapdragon MPSoC from Qualcomm running at 1.5GHz. On this platform the default OpenMP scheduling policy is `static`. The schedule optimization and OpenMP pragma transformation were implemented in the Clang/LLVM compiler framework.

TABLE I: Characteristics of the Rodinia Benchmarks

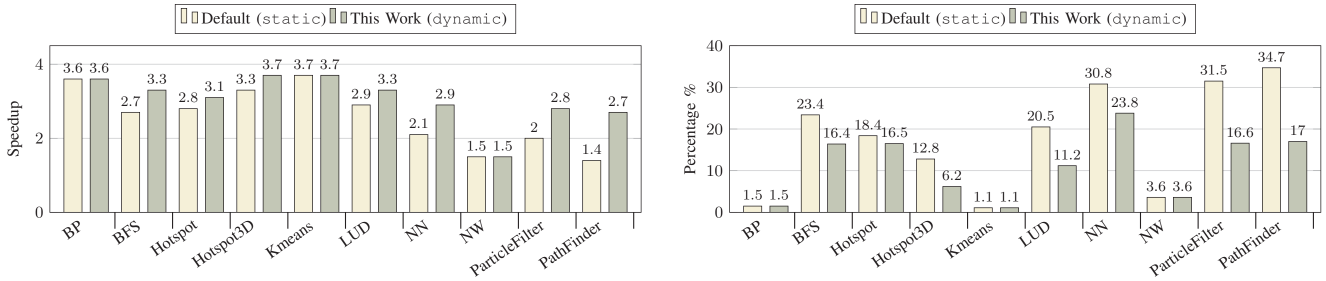
Benchmarks	OpenMP Loops	Scheduling Policy	Chunk Size
BP	2	`static` `static`	default default
BFS	2	`dynamic` `dynamic`	133224 192188
Hotspot	1	`dynamic`	8
Hotspot3D	1	`dynamic`	2
Kmeans	1	`static`	default
LUD	2	`dynamic` `dynamic`	7 4
NN	1	`dynamic`	764
NW	2	`static` `static`	default default
ParticleFilter	2	`dynamic` `dynamic`	2250 846
PathFinder	1	`dynamic`	155435

B. Case Studies

The case studies considered in this work are taken from the Rodinia benchmark suite [6]. This suite is a collection of benchmarks manually parallelized using multiple paradigms including OpenMP. However, the handwritten pragmas do not specify a scheduling policy. Therefore, the default `static` scheduling policy is used, as it the case typically in OpenMP runtime systems. Rodinia is an excellent choice to evaluate the effectiveness of our approach, since it allows us to compare benchmarks automatically optimized by our framework, with their corresponding original versions manually parallelized by expert programmers.

Table I shows the characteristics of the benchmarks. First of all, the number of loops parallelized with OpenMP for each benchmark are presented. Then, the table shows for each loop the selected scheduling policy by our approach, and where `dynamic` scheduling is used, the chunk size is presented as well. For the analysis we set the minimum threshold for the load imbalance to 5%, thus any loop with an imbalance lower than this threshold is scheduled with the default `static` scheduling policy, which in these cases is a better choice as it presents less overhead. From the benchmarks considered only three cases presented loops with a load imbalance lower than 5%: *BP*, *Kmeans* and *NW*. The effect of low per-loop imbalance is reflected in the overall imbalance per benchmark in Figure 5b.

Figure 5a shows the speedup results, using as a baseline the sequential execution time of each benchmark. We evaluated two scenarios: *i*) initial manually parallelized versions in which the scheduling policy is not specified, thus the `static` policy is used by default, and *ii*) automatically optimized versions with our framework using the proper scheduling policy, as Table I shows. The average speedup in the initial scenario was 2.6×, while in the optimized scenario the average speedup was 3.1×. Therefore, our approach performs on average around 20% better than the original manually annotated code, where the default `static` scheduling policy is used. The most dramatic improvement is in *PathFinder* with a performance increase of 93%. Figure 5b shows the



(a) Speedup

(b) Overall Load Imbalance

Fig. 5: Experimental Results

overall load imbalance results per benchmark. In this case we considered the same scenarios described in the speedup results. The average load imbalance in the initial scenario was 18%, while in the automatic optimized scenario was 11%. Therefore, our approach reduces the load imbalance presented in the initial scenario on average by 36%. Here the most dramatic improvement is also in *PathFinder*, where the load imbalance is reduced by 51%.

IV. Related Work

Parallelizing Compilers: Multiple works have addressed automatic parallelization in the embedded domain. Castrillon [1] proposed a parallelization toolflow for homogeneous MPSoCs that relies on a set of heuristics to discover multiple forms of parallelism from sequential applications. We extended this toolflow targeting multicore Android devices in [2] and heterogeneous MPSoCs in [3]. However, none of the previous works address loop scheduling.

Parallel Slack: This concept has been exploited mainly in the context of parallel programming paradigms from Intel, for example in *Threading Building Blocks* (TBB) [11]. In TBB a work-stealing scheduler is used, which takes advantage of abundant parallel slack to efficiently balance the workload of applications. Instead, in this work, we exploit this concept to improve the load balancing of OpenMP parallel loops.

Loop Scheduling: Most of the research efforts to improve the OpenMP loop scheduling have focused on the high performance computing (HPC) domain, and none of them have addressed the problem of defining a proper chunk size for the standard OpenMP `dynamic` scheduler. Thoman et al. [5] proposed an OpenMP scheduler that relies on the polyhedral analysis. However, the polyhedral model imposes multiple restrictions on the code that it can analyze, which limits its applicability. Wang et al. [12] proposed a compiler-based machine learning approach to parallelize loops with OpenMP. However, it does not address how to define the chunk size for the `dynamic` policy, then the default value of 1 is used, which could lead to high overhead, as shown in Figure 1b. Moreover, in contrast to our work where only a single-pass analysis is required, machine learning approaches require extensive training.

V. Conclusion

In this paper, we presented a schedule-aware loop optimization approach for embedded MPSoCs. The approach optimizes existing OpenMP loops by choosing the proper scheduling policy on a per loop basis to improve load balancing by exploiting the parallel slack. This approach is enabled by on-device profiling on the target embedded platform. The effectiveness of our framework was demonstrated by optimizing benchmarks from the Rodinia benchmark suite on an Android tablet. Results show that the speedups achieved by our approach, outperformed the ones achieved by the initial handwritten OpenMP pragmas on average by around 20% and in the best case by 93%. Moreover, our approach was able to reduce the load imbalance on average by 36% and in the best case by 51%. In future work, we plan to perform further evaluations on other platforms and to apply our approach to heterogeneous MPSoCs.

References

[1] J. Castrillon and R. Leupers, *Programming Heterogeneous MPSoCs: Tool Flows to Close the Software Productivity Gap*. Springer, 2014.

[2] M. A. Aguilar, J. F. Eusse, P. Ray, R. Leupers, G. Ascheid, W. Sheng, and P. Sharma, "Parallelism extraction in embedded software for Android devices," in *SAMOS XV*, July 2015.

[3] M. A. Aguilar, R. Leupers, G. Ascheid, and L. G. Murillo, "Automatic parallelization and accelerator offloading for embedded applications on heterogeneous MPSoCs," in *DAC '16*, pp. 49:1–49:6.

[4] E. Ayguadé, B. Blainey, A. Duran, J. Labarta, F. Martínez, X. Martorell, and R. Silvera, "Is the schedule clause really necessary in OpenMP?" in *WOMPAT'03*, Berlin, Heidelberg.

[5] P. Thoman, H. Jordan, S. Pellegrini, and T. Fahringer, "Automatic OpenMP loop scheduling: A combined compiler and runtime approach," in *IWOMP'12*, Berlin, Heidelberg, 2012.

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC 2009*, Oct 2009, pp. 44–54.

[7] "Nexus 7 (2013)," [Online] Available http://www.asus.com/Tablets_Mobile/Nexus_7_2013/ (accessed 09/2016).

[8] M. McCool, J. Reinders, and A. Robison, *Structured Parallel Programming: Patterns for Efficient Computation*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann, 2012.

[9] R. P. Brent, "The parallel evaluation of general arithmetic expressions," *J. ACM*, vol. 21, no. 2, pp. 201–206, Apr. 1974.

[10] K. Fürlinger and M. Gerndt, "ompP: A profiling tool for OpenMP," in *IWOMP'05/IWOMP'06*, Berlin, Heidelberg, 2008.

[11] "Intel Threading Building Blocks," [Online] Available https://software.intel.com/en-us/intel-tbb (accessed 09/2016).

[12] Z. Wang and M. F. O'Boyle, "Mapping parallelism to multi-cores: a machine learning based approach," in *PPoPP '09*, pp. 75–84.