Adaptive Encoding of Multimedia Streams on MPSoC

Julien Bernard[1], Jean-Louis Roch[1], Serge De Paoli[2], and Miguel Santana[2]

[1] INRIA/MOAIS
Laboratoire Informatique et Distribution
51, avenue Jean Kuntzmann
38330 Montbonnot Saint Martin, France
[2] STMicroelectronics
850, rue Jean-Monnet
F-38926 Crolles Cedex, France

Abstract. This paper describes a dynamic scheduling technique based on work-stealing that is proved to be efficient on SMP and clusters. We apply this technique in the MPSoC field, using a simulation in SystemC. We experiment on a MPEG-4 encoding application and we demonstrate that the work-stealing scheduling is more efficient than a static placement scheduling in terms of time and use of resources.

Keywords: MPSoC, scheduling, work-stealing.

1 Introduction

In order to improve the performance of current embedded systems, Multiprocessor System-on-Chip (MPSoC) offers many advantages, especially in terms of flexibility and low cost.

Applications require more and more intensive computations, especially multimedia applications such as video encoding. The system should be able to exploit the resources as much as possible in order to save power and time. This challenge may be addressed by a technique based on parallel computing coupled with performant scheduling.

In this paper, we present a dynamic scheduling technique based on work-stealing. It is proved to be efficient in the SMP and cluster area and we make a proof-of-concept adaptation for an MPSoC platform based on SystemC. We use an MPEG-4 encoding algorithm to compare the work-stealing scheduling with a static placement scheduling.

This paper is organized as follows. Section 2 gives an overview of different scheduling techniques used on MPSoC and explains the principles of work-stealing. Section 3 presents the key points in the implementation of work-stealing and our choices for the implementation on top of SystemC. Section 4 describes various MPEG-4 implementations including our implementation using work-stealing. Section 5 exposes the results we obtained with our implementation on our platform, compared to a static sheduling. Finally, section 6 concludes the paper.

V.N. Alexandrov et al. (Eds.): ICCS 2006, Part IV, LNCS 3994, pp. 999–1006, 2006.

2 Related Work and Scheduling on MPSoC

In this section, we discuss about the state of the art related to scheduling, and in particular scheduling on MPSoC.

2.1 Scheduling by Mapping and Pipelining

Mapping and pipelining are two static scheduling methods to improve the performances of MPSoC.

Mapping consists in sharing data by making a static placement on the available resources. The way the placement is done is closely linked to the application. Moreover, the time of computation highly depends on the input data. So the performance are irregular and impredictible.

Pipelining consists in sharing computation by cascading several processors, each one making a part of the whole function. By nature, the number of processors is limited and fixed by the application. And the global computation rate is limited by the rate of the slowest processor.

So, all these approaches are neither scalable nor efficient. First, the number of processors is fixed and depends on the application itself. And more generally, the program and the hardware are closely linked. Second, the program does not adapt to the input data. This results in poor performances for the worst-case data.

2.2 Dynamic Work-Stealing

Scheduling constraints such as architecture independence and input data independence are close to the ones considered for fine grain multithreaded computations [15]. To schedule such computations, many works focus on *work-stealing*, from both a theoretical point of view [1] [8] and a practical point of view [9] [3] [10].

A work-stealing scheduling is based on a classical greedy scheme. It consists in mapping to an idle processor a task that is ready to be executed. Following [1], we note T_∞ the execution time of an algorithm on an infinite number of processors and T_1 the sequential time of this algorithm. Then, neglecting the cost of the interpretation, R.L. Graham [13] proved that the time T_p required for execution on p processors verifies:

$$T_p \leq \frac{T_1}{p} + T_\infty \tag{1}$$

This time appears asymptotically optimal in the case of very parallel applications where $T_\infty \ll T_1$. However, realizing this scheduling also has a cost that must be taken into account. It is *a priori* bounded by the number n of tasks. Since $n > T_1/T_\infty$, this overhead can be very important for a fine-grained algorithm.

Work-stealing schedulers try to minimize this overhead by generating parallelism only when required, i.e. when a processor becomes idle. Efficient work-stealing is based on the *work-first principle* [3]: move the cost of parallelism to the critical path. Indeed, the number of idle tops is bounded by T_∞ on each processor; thus, if the processors are able to easily find some tasks that are ready

to be executed, the scheduling overhead, bounded by $O(p.T_\infty)$, will be negligible for algorithms that have a high level of parallelism.

Initially developped for SMP architectures [3], the principle has been extended to processors with different speeds [8] and then to distributed architecture [7], SMP clusters and heterogeneous grids [2].

2.3 Conclusion

So, on the one hand, usual scheduling techniques on MPSoC such as mapping and pipelining seems to have many drawbacks, especially in terms of adaptability. On the other hand, a proven efficient scheduling technique based on work-stealing exists for distributed systems. Our approach is to import work-stealing in the MPSoC field.

3 Implementation of Work-Stealing on MPSoC

This section describes our technical choices for the implementation of work-stealing on MPSoC.

In our experiments, we use a platform to simulate a MPSoC. It is implemented over SystemC, using a Transaction Level Modeling. It is composed of severals nodes linked together by a component called network. A node has a processor, a ROM, a RAM, an interrupt controller and a timer. The network is simply a shared memory in which we added extra-functionnalities. Figure 1 shows this platform.

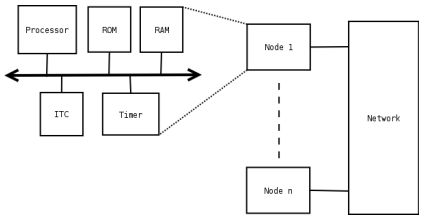

Fig. 1. The MPSoC platform of our experiments

To implement the schedule while reducing contention, work-stealing is often based on a randomized distributed algorithm. A task is locally managed on the processor that creates it and the default sequential (depth-first) execution is optimized. Each processor then locally handles its own list of tasks. When it is required (synchronization between some tasks on different processors, or idle time of one processor), a processor can access to a part of the list owned by another processor, in mutual exclusion, to steal a task (theft operation).

3.1 Choice of the Victim

First, when a processor becomes idle, it steals the oldest ready task on a chosen processor. Here, two ways are possible. A deterministic approach [2]: the victim

processor is chosen cyclicly (round-robin). A probabilistic approach: the victim processor is chosen randomly [3]. Then, for parallel computations on p identical processors, it is proved that $T_p < \frac{T_1}{p} + O(p).T_\infty$ with high probability.

To choose a victim processor, we applied the deterministic approach for two main reasons. First it is simpler to implement, there's no need for a pseudo-random number generator, a simple counter is enough. Moreover, a processor can easily determine the end of the computation: if none of the other processors has work left then, it's finished.

3.2 Mutual Exclusion of the Stacks

The synchronization between processors being rare ($O(p.T_\infty)$ for parallel computations), most exclusive accesses are local. Then, an arithmetic lock based on an atomic instruction (such as CompareAndSwap) may be used to implement mutual exclusion. The synchronization may even be implemented basically through very light counters if both process access to distinct parts of the list, such as the head and the tail typically (THE protocol [14] [3]).

Working with SystemC, we could not use an architecture-dependent instruction like CompareAndSwap. In fact, we implemented a very trivial method that may be improved a lot: we added a hardware lock in the network component. It is a special address in memory that, when read, activates a lock. There is one lock for each processor attached to the network. Each one is used to protect each processor's stack.

More generally speaking, embedded processors may have or not such an atomic instruction. It is necessary to have this operation when dealing with efficient distributed applications. The technique we used is far from efficient due to the contention it implies: it may be improved in the future.

3.3 Local Stack Management

We implemented the local lists of tasks of the processors in the shared memory. The main advantage is that processors can steal work themselves to other processors. So, a processor is never interrupted and is always in activity until there is no more work anywhere.

The memory is shared between all the processors. Each list of task is simply organized as a stack of equal size chunks. This is possible because we know the application and then, we can make some optimizations. More precisely, a chunk consists in 32 bytes that stores structure information (mainly to reproduce the calling stack) and the parameters of the functions.

We make this choice because we want to keep things simple ; having only a single level of memory to manage is easier in this first attempt. We don't want to make a fully functional portable system but rather a proof of concept.

In the future, it will be possible to implement the local stacks in the local memories of the processors. It will probably be more efficient as an access in a local memory is faster than an access in a shared memory.

4 Case sudy: MPEG-4 Encoder

In this section, we will first quickly analyze an existing parallel implementations and then, explain how we did an implementation using work-stealing. We assume that the reader has a knowledge of the MPEG-4 standard.

4.1 Analysis of an Existing Implementation

An approach for a parallel implementation is to use fine grain parallelism. This method is fully described in [5]. The idea is to search for the data dependencies in the algorithm at a very fine grain level. Then, this model is transformed to add a proper mapping and scheduling. Finally, with all the added meta-data, the compiler is able to generate a parallel code for an SMP machine.

This approach of parallelism is very application-dependent. Moreover, it does not allow to adapt the computations to the actual picture. It is close to the static scheduling problems we talked about previously (see 2.1).

4.2 Adaptive Parallel MPEG-4 Encoding

Our approach is to use the work-stealing scheduler that we implemented (see 3).

We first make some simplifications to the MPEG-4 encoder, based on the data dependencies analysis: we only consider the encoding of one frame, as each frame has to be processed after the previous one. Then, we also consider to use a motion compensation algorithm that only depends on the previous picture and not the current one. This allows to compute all the macroblocks in parallel.

Then, we introduce a little overhead in the algorithm to improve the performance of work-stealing. We make a recursive cut of the image i.e. we make a function that simply cut the image in several parts and apply itself recursively on each part until there is only one macroblock in the part. Then, the normal function effectively treats the macroblock.

This overhead allows to create tasks of different weight at each step of the recursion so that big task will be stolen first. As a consequence, the number of steals decreases as each processor computes big tasks before idling.

This configuration given, we can make some theoretical analysis relative to our application.

The total work T_1 is the sum of the computation of all the macroblocks (which is in fact the sequential work T_s) plus the overhead implied by the recursive cut and by the work-stealing mechanisms.

T_∞ is, in our simple case, the largest computation time among the computation times of all the macroblocks. It should not be too far from the average computation time of one macroblock. In fact, to be in the condition of the greedy scheduling theorem [4] (see 2.2), the average parallelism T_1/T_∞ should be close to the number p of processors.

In our case, we make our tests with CIF pictures. That represents roughly 400 macroblocks[1]. And we consider that p is not higher than 20. So, that allows the

[1] In fact, 22*18=396 macroblocks.

worse macroblock to be computed 20 times slower than an average macroblock, which should be largely enough.

5 Experiments on an MPSoC Platform and Results

This section presents the results we obtained in our experiments.

For our experiments, we used usual test sequences in CIF format: coastguard, football, foreman, news and stefan. We encoded them with the algorithm presented in 4.2.

We made a first series of experiments on each sequence. We encoded 30 consecutive frames of each sequence. Then, we calculated the average T_∞ and the average *average parallelism* (T_1/T_∞) on the overall frames. Finally, we chose a picture whose caracteristics were close to this average for the second series of experiments.

The reason for doing this is that we wanted to make our experiments on real pictures and not on average unreal pictures, as the work-stealing scheduler takes advantage of the non-uniformity of input data. So we adopted this compromise to have an average real picture for our test.

In the second series of experiments, we compare our work-stealing scheduling with a static placement scheduling. The static placement scheduling consisted in sharing the macroblocks in strips of 18 macroblocks (the height of the picture). Each processor receives the same number of strips with a maximum difference of 1.

We tested both on the same frame (the one chosen previously) of each sequence and we calculated the parallel efficiency $(T_1/(p * T_p))$.

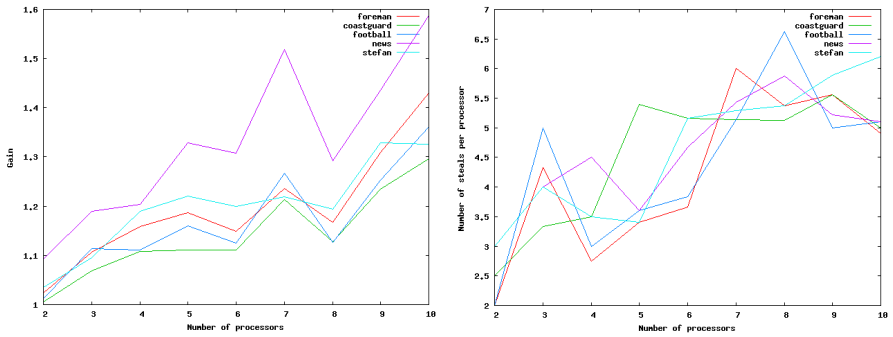

Fig. 2. (a) Gain of the work-stealing sheduling over the static placement scheduling ; (b) Number of steals per processors for the work-stealing scheduling

Figure 2(a) shows the gain in term of time of the work-stealing method over the static method (i.e. $T_{p_{sp}}/T_{p_{ws}}$) for p processors, p varying from 2 to 10 (included).

With four or more processors, the static placement scheduling is at least 10% slower. In the worse case, it can even be 50% slower than the work-stealing scheduling.

We can notice that there is a big improvement with 7 processors. This can be explained: in the case of the static placement, a total of 22 strips are shared among 7 processors, which means that all processors receives 3 strips except the last one which receives 4 strips. While the last processor computes its fourth strip, the other one are simply waiting. That's where the dynamic scheduling is far more efficient, allowing the idling processors to help the last processor.

Another static placement could have been chosen. More efficient static placement will be used and compared to in the future. But this shows that in a real case, the static placement can be penalizing.

In addition, for the work-stealing scheduling, we calculated the number of steals per processor. Figure 2(b) shows the number of steals per processor for p processors, p varying from 2 to 10 (included).

Figure 2(b) proves that the number of steals per processor does not grow too much and then remains constant. Further measures with a higher number of processors confirm this. This totally sticks to the theory (see 2.2): the amount of communications between processors remains quite low, whatever the number of processors.

These results must take into account that the overhead of the work-stealing scheduling does not have much impact as we have been able to make a very optimized version. A more general implementation would have more weight. Moreover, this results are based on simulations and the simulations must be improved in order to have more precise results.

6 Conclusion and Perspectives

In this paper, we demonstrated that the work-stealing scheduling, that was proved efficient for distributed systems, is worth being considered for MPSoC. We made some experiments on a MPSoC simulation platform based on SystemC with a MPEG-4 encoding algorithm that gave us a gain of 10% at least for four processors or more.

This is currently a proof of concept. We aim at improving the implementation of the work-stealing scheduler, and to use better MPSoC simulations so that we can have even more reliable results.

References

1. Blumofe, R.D., Leiserson, C.E.: Space-efficient scheduling of multithreaded computations. SIAM Journal on Computing **27**(1) (1998) 202–229
2. Revire, R.: Ordonnancement de graphe dynamique de tâches sur architecture de grande taille. Régulation par dégénaration séquentielle et distribuée. PhD thesis, Institut National Polytechnique de Grenoble (2004)
3. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98). (1998)
4. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. In: Proceedings of the 35th Symposium on Foundations of Computer Science, Santa-Fe, New Mexico (1994) 356–368

5. Assayad, I., Gerner, P., Yovine, S., Bertin, V.: Modelling, analysis and parallel implementation of an on-line video encoder. In: 1st International Conference on Distributed Frameworks for Multimedia Applications. (2005) 295–302
6. Galilée, F., Roch, J.L., Cavalheiro, G., Doreille, M.: Athapascan-1: On-line Building Data Flow Graph in a Parallel Language. In IEEE, ed.: International Conference on Parallel Architectures and Compilation Techniques, PACT'98, Paris, France (1998) 88–95
7. Roch, J.L., Gautier, T., Revire, R.: Athapascan: API for Asynchronous Parallel Programming. Technical Report RT-0276, INRIA Rhône-Alpes, projet APACHE (2003)
8. Bender, M.A., Rabin, M.O.: Scheduling Cilk multithreaded parallel programs on processors of different speeds. In: ACM Symposium on Parallel Algorithms and Architectures. (2000) 13–21
9. Mohr, E., Kranz, D.A., Robert H. Halstead, J.: Lazy task creation: A technique for increasing the granularity of parallel programs. IEEE Transactions on Parallel and Distributed Systems **2**(3) (1991) 263–280
10. Blumofe, R.D., Papadopoulos, D.: HOOD: A user-level threads library for multiprogrammed multiprocessors. Technical report, The University of Texas at Austin (1998)
11. Willebeek-Le-Mair, M., Reeves, P.: Strategies for dynamic load-balancing on highly parallel computers. IEEE Transactions on Parallel and Distributed Systems **4**(9) (1993) 979–993
12. Chretienne, P., Coffman, E.J., Lenstra, J.K., Liu, Z.: Scheduling Theory and its Applications. John Wiley and Sons, England (1995)
13. Graham, R.: Bounds on multiprocessing timing anomalies. SIAM J. Appl. Math. **17**(2) (1969) 416–426
14. Dijkstra, E.W.: Solution of a problem in concurrent programming control. Communications of the ACM **8**(9) (1965) 569
15. Roch, J.L.: Ordonnancement de programmes parallèles sur grappes : théorie versus pratique. In: Actes du Congrès International ALA 2001, Université Mohammmed V, Rabat, Maroc (2001) 131–144
16. Pazos, N., Maxiaguine, A., Ienne, P., Leblebici, Y.: Parallel modelling paradigm in multimedia applications: Mapping and scheduling onto a multi-processor system-on-chip platform. In: Proceedings of the International Global Signal Processing Conference, Santa Clara, California (2004)