The Impact of Data Communication and Control Synchronization on Coarse-Grain Task Parallelism

Jeroen Hordijk Henk Corporaal

Delft University of Technology
Department of Electrical Engineering
Section Computer Architecture and Digital Systems
P.O. Box 5031, 2600 GA Delft, The Netherlands
e-mail: {jeroen,heco}@duteca.et.tudelft.nl

Abstract

Research into automatic extraction of instruction-level parallelism and data parallelism from sequential languages by compilers has been going on for many years. However, task parallelism has been almost unexploited by parallelizing compilers. It has been shown that coarse-grain task parallelism is a useful additional resource of parallelism for multiprocessors, but the simple and restricted execution models of the automatic compilers have resulted into poor performance figures. This paper presents experimental results used to evaluate the available coarse-grain (procedure based) task parallelism in a set of *C* benchmarks assuming different machine models, ranging from very basic to extreme complex. The experiments show large amounts of available parallelism for machines which support both fast data communication and complex control synchronization.

1 Introduction

The automatic exploitation of parallelism from sequential programs has been focused on instruction-level parallelism and data parallelism for many years; this in contrast to task parallelism which is mostly extracted from parallel languages (either with explicit parallel constructs or using implicit parallelism like in functional languages). Instruction-level parallelism is limited to regions or traces and data parallelism is restricted to loops; both usually containing a few basic blocks only. Therefore coarse-grain task parallelism is an additional parallel resource almost unexploited by parallelizing compilers up till now.

The choice of explicit parallel languages or sequential languages depends on the fundamental question whether the programmer or the compiler is responsible for detecting and exploiting the parallelism in the program. The use of special explicit parallel languages may result in high executing performance, but commonly used imperative languages allow better standardization and better reusability of (existing) code. The motivation of using *C* (or *C*++) as source language, is that we try to develop compiler techniques which are capable of extracting task parallelism with a minor interference of the programmer. Although some imperative language elements, like pointers in *C*, make parallelization difficult, they increase the expressive power useful for inherently sequential code pieces.

Numerous studies have proven the existence of a large potential of coarse-grain task parallelism in sequential program, see for instance [1, 2]. On the other hand only a few actual system designs have been developed and these exploit only a limited amount of coarse-grain task parallelism [3, 4]. The most important goal of our study is to discover which abstract machine characteristics give rise to these large differences in parallelism. The most effective machine models presented in the paper also suggest the future direction of the research in automatic exploitation of coarse-grain task parallelism.

In order to explore the solution space, the specific properties and problems of task parallelism are analyzed in section 2. The different machine models used in the studies are described in section 3. In section 4, we discuss the experimental framework used to evaluate the limits of parallelism under the different machine models. The performed experiments itself are discussed and analyzed in section 5. Finally, in section 6 we summarize the presented work and discuss the implications of the study for further research in exploitation of coarse-grain task parallelism in sequential programs.

2 Problem Statement

The efficient exploitation of existing multiprocessors requires programs to be partitioned into coarse-grain tasks, because smaller grain sizes usually result into too large synchronization overhead. In addition, the coarse task granularity will preserve the fine-grain instruction-level parallelism to be exploited by the individual processors. The automatic exploitation of coarse-grain parallelism is difficult because of the limitations of compile-time analysis techniques. Most notably, the state-of-art data dependence analysis, which must conservatively assume a lot of non-existing dependences, inherently decreases the parallelism [5]. When only exploiting parallelism from data and control independent tasks, parallelization already fails if only one dependence is mispredicted. Larger tasks contain more instructions, which results in more opportunities for dependence misprediction.

Most of the research into detection and exploitation of task parallelism from imperative languages has been focussed on generation of data independent tasks. The performance of these exploitation techniques is still poor, especially for tasks with larger granularity; *e.g.* Beckmann [6] reports a parallelism of about 1.5 for coarse-grain tasks. The explanation of these disappointing results is given by the fact that function calls are only in 18 percent data independent [4] [1]. For basic blocks and individual statements this is 45% and 76% respectively. It must be concluded that fine-grain (instruction-level) parallelism can be exploited more easily. And for the extraction of coarse-grain task parallelism a solution for the extreme data dependence problem is needed.

In order to exploit parallelism with larger granularity, one must develop partitioning algorithms to generate parallel tasks, which assume more complex synchronization schemes. This requires multiprocessors which allow fast data communication and control synchronization between the processing units. Some research is already done on hardware [7, 8], which supports fast interprocessor forwarding of (register) data. Code generation and optimization techniques for these kind of architectures are rare or non-existent. In this article we propose basic solutions for the problems of exploiting coarse-grain task parallelism and analyze the performance of these techniques using an experimental framework.

3 Machine Models

In order to analyze the limits of task parallelism, we will introduce a number of techniques which solve the fundamental problems of coarse-grain task parallelism. As shown in the previous section these solutions have to concentrate on efficient data communication and control synchronization. The combinations of the proposed techniques in this section result into a wide set of different machine models, ranging from very basic to extreme complex. The consequences for the actual hardware implementations and compiler construction of the proposed techniques are also analyzed below. For this analusis we assume that the partitioning into tasks is performed by the compiler and that the scheduling of these tasks is done dynamically by a central scheduling unit.

Data Communication

Partitioning a program may result into data and control dependences between the sub-tasks. Correct execution of the program requires that the tasks are scheduled so that all dependences are obeyed. Two main approaches can be distinguished: dependent task may or may not execute concurrently; of course dependences must always be resolved. Sequentialize dependent tasks is a simple technique but also very restricted and will only perform well for small tasks. Larger and more tightly coupled tasks will need more complex schemes. Data communication between tasks can be viewed from the two sides: the definition of data which resulted in the dependence and the use of this data. Both are explained in detail below.

There are two possible times to forward the data values on which other tasks are dependent. One is directly after the data is produced, which we will call *early release* and the other is when the task is finished, named *late release*. Naturally, the first scheme results in the best performance, but at the cost of more complex hardware synchronization support.

The scheduling unit of the late release machine model only has to keep track of the tasks on which each task depends on. A task can be scheduled when all of the parent tasks are finished. The scheduling unit needs

[1] Note that, they only consider control independent function calls, so that the percentage of both data and control independent task is even smaller.

a table which for each task holds the task it is dependent on and when a task is finished the occurences of the task are marked in the table. A task is ready to execute when all its parent tasks are marked.

Releasing data values as early as possible, *i.e.* directly after they are produced, requires that for each task it is registered on which data values it depends. Because tasks are likely to consume multiple results of one task, the needed table size will increase compared to the late release technique. For instance, consider arrays which are large collections of data values and storing these dependences separately is probably to costly. The most forward solution is to determine the dependences for regions of arrays, instead of single values.

Inside the source code of each task release statements must be scheduled, which generate messages to the central scheduling unit to indicate that particular data is generated. Possible interaction of control flow complicates the job of the compiler, because for some control flow paths the values may not be generated at all.

Where the release time is related to the definition of data values, the *strictness* of the execution model relates to the use of the data values. Just like the release time there are two variants of strictness. Either the scheduling algorithm requires that all input dependences are satisfied before a task can start, this is called *strict execution*. Data dependences can also be resolved during the execution of the task, *i.e.* tasks are allowed to start executing before all dependent data values are available, which is called *non-strict execution*.

In addition to the extra hardware and compiler support needed for early releases of data values, non-strict execution also requires fundamental adaptations to the task scheduling algorithm. Consider the example where two dependent, equal sized, tasks are scheduled simultaneously. When the first task produces data about the same time the other task is ready to consume this data, the performance doubles. However, in the case where one task produces information at the end which is needed by the other task in the beginning the performance equals the sequential model. Therefore, an effective scheduling algorithm will also need the information about definition and use times of data values inside tasks. This significantly increases the size of the scheduling table and the complexity of the scheduling algorithm.

Most of the developed designs do not allow non-strict execution and partition tasks so that results are produced at the end of a task and used in the beginning. This requires smaller tasks and therefore results in higher overall costs for the spawning of tasks.

Control Synchronization

Control dependences are generated by control flow operations, like the branch statement. Correct execution of a program requires that all control dependences are satisfied. Nevertheless, two techniques are well-known to speedup control synchronization: *speculative execution* and *executing multiple flows of control*.
Speculative execution of tasks allows execution of control dependent tasks in parallel. The technique predicts the probability of initiating a new task. Whenever this probability is high the initiation is done speculatively. In case a mis-prediction occurs, *i.e.* the control path did not reach the actual spawning statement, the speculative task must be squashed. The efficiency of the hardware usage under the speculative task model is very dependent on the accuracy of the prediction scheme. On the other hand the performance is dependent on the aggressiveness of the prediction scheme.

Instead of following only one execution path in the control flow, *multiple alternative paths* can be executed in parallel. The multiple path execution model however spills hardware usage, regardless of the precision of the prediction scheme. The experimental framework discussed in the next section implements the Disjoint Eager Execution (DEE) technique, which has a better performance and better utilization of resources than Eager Execution (EE) [9]. In the experiments the number of alternative paths followed by the DEE scheme is three.
Multiple flows of control is a technique that uses control dependence analysis to find control independent flow paths from different parts of a program [2]. Consider for instance the following simple *C* code

```
for (i = 0; i < 100; i++)
     bar(i);
foo();
```

The function call `bar` is control dependent on the loop exit condition `i<100`. The `foo` function is not dependent on any part of the loop and can execute in parallel with the loop, assuming there is no data dependence between `bar` and `foo`. In general multiprocessors are needed for following multiple flows of control. Exploiting multiple control flows on a uniprocessor is only possible if the control flows of the functions `bar` and `foo` are simple, otherwise the code size explodes.

4 Experimental Framework

An experimental framework has been set up to measure parallelism upper bounds under the different abstract machine models. The techniques considered essential were discussed in the previous section, applying them in various combinations results in a set of abstract machine models. This section focuses on the implementation and limitations of the experimental framework.

In order to produce upper bounds for the available task parallelism, we will only enforce the essential machine characteristics and relax all other constraints. The basic approach for this is the use of trace analysis. First the program is executed to produce a trace of performed instructions. Then these instructions are examined for data and control dependences. Analyzing the dependences is relatively easy because the trace includes the outcome of branches and the actual memory locations. Using the dependences the critical path can be found. Dividing the total number of instructions by the critical path length equals the maximum available parallelism.

The trace simulator is implemented using the MOVE software framework, which supplied the instruction simulator, gathering of profile information and analysis of the control flow [10]. The MOVE software framework uses the *GNU gcc* compiler as frontend.

We will use the procedures of the program as individual tasks. As shown in [1] concurrent execution of procedures is a large sources of parallelism. Programs can roughly be divided into four levels of granularity: *i.e.* statement, basic block, loop-iteration and procedure [4]. Using large tasks *i.e.* procedures, minimizes the overhead of spawning tasks on processors and preserves most of the instruction-level parallelism exploitable for the individual processors. More complex partitioning algorithms will be researched in the future.

For the generation of instruction-level parallelism the following assumptions were made. We do not consider any resource constraints, which means that the number of functional units is infinite and the same holds for registers, bypass links and other hardware resources. This also eliminates the need for an instruction scheduling algorithm. All functional units have a latency of one clock cycle. Therefore, the distribution of the operations over the different operations classes has no effect on the execution time of the application nor its parallelism.

We only consider true data dependences on registers, *i.e.* all anti-dependences and output dependences on registers are removed. In practice, various renaming techniques can be used to remove some of these spurious data dependences. For data dependences on memory locations anti-dependences and output dependences are obeyed. Untrue data dependences are far more complex to eliminate for memory locations than for registers. The memory disambiguation is perfect, *i.e.* memory references are never conservatively assumed dependent because their actual address is not known yet. The compiler memory analysis and hardware speculation techniques can never achieve this upper bound.

For the exploitation of task parallelism, an infinite number of processors are assumed, discarting the need for a scheduling technique. The processors can communicate through a bus with unlimited bandwidth, but the latency for the data communication can be varied (the default value is one clock cycle). All control synchronization schemes assume that tasks can never start earlier than their calling task did.

5 Experimental Results and Evaluation

This section describes the experiments used to analyze the parallelism for the abstract machine models. The experimental framework allows a very wide range of machine models. Therefore, only a small subset of the experiments will be presented. And, we will only discuss the most important characteristics of the techniques discussed in section 3.

The experiments are performed on the ten benchmarks summerized in table 1. The benchmarks are coded in the sequential imperative language *C* and are well known within the 'UNIX' world. The benchmark are integer type of code, also called non-numeric programs. These benchmarks are usually quite sequential, *i.e.* they contain relatively a large number of data and especially control dependences. This requires fast communication and complex synchronization techniques in order to exploit the available parallelism.

We will use four basic machine models as basis for the analysis of the experimental results. The BASE model does not have any special technique for fast communication of data values nor control synchronization, which means that data and/or control dependent tasks can not execute concurrently. This model uses the strict execution and late release schemes and none of the control techniques discussed in section 3. The DATA machine model combines non-strict execution with early releases of data values, so that data depend-

benchmark	lines	operations	ILP	BASE	DATA	CTL	ALL
a68	13565	2805K	4.8	6.8	10.9	7.5	44.1
cpp	7657	1983K	9.7	16.3	21.1	18.7	26.2
dhrystone	3915	7378K	4.4	12.5	38.7	12.5	73.6
diff	11424	602K	6.6	7.9	8.2	10.9	13.7
flex	10454	12214K	7.4	13.4	23.4	15.4	57.2
grep	19329	525K	7.3	11.9	13.8	12.6	17.6
od	3132	22000K	7.1	12.3	34.7	12.4	200.7
queen	3407	20K	6.3	8.7	10.0	9.1	86.9
sieve	3361	3627K	109.3	110.1	110.5	110.2	110.5
wc	1313	283K	12.3	12.8	12.8	12.8	12.9
average	7756	5144K	17.5	21.3	28.4	22.2	64.3
average[a]	9110	5941K	6.7	11.2	20.1	12.4	65.0

[a] Average of all benchmark excluding *sieve* and *wc*.

Table 1: Benchmark characteristics and parallelism under four basic machine models.

ent tasks are perfectly overlapped. Basic model CTL implements the multiple path and multiple flow techniques, so that control synchronization between tasks is minimized. As last, the ALL machine combines the techniques of the DATA and CTL machine, allowing execution of both data and control dependent task concurrently. As reference we use a machine model which does not exploit task parallelism, called ILP (abbreviation for Instruction-Level Parallelism).

In table 1 the total available parallelism, *i.e.* both instruction-level parallelism and task parallelism, is shown for the basic machine models. The BASE model hardly exploits more parallelism than the available instruction-level parallelism. This directly supports conclusions of earlier studies which reported the tightly coupled behavior of procedures. The DATA model and CTL naturally extracts more task parallelism than the BASE model, but the DATA model performs quite better than the CTL model. However, the ALL model has superior speedups. This simply shows that most coarse-grain tasks are both control and data dependent on each other.

Looking to the benchmarks individually, one notice that two of them, namely *sieve* and *wc*, do not benefit from task parallelism at all. Apparently, these program perform most of the work in one procedure. Table 1 also shows the average parallelism ignoring these two benchmarks. The rest of the performed measurements exclude them from the benchmark set, because they are irrelevant for our research.

In table 2 the performance of the different control synchronization techniques are analyzed by comparing the available parallelism for different control-oriented machine models. The data communication scheme for these experiments combines non-strict execution and early release data values. Speculative execution does not result in large speedups over non-speculative execution. There are two main reasons for this. Firstly, the procedure calls are normally quite far apart in the sequential code. Secondly, the misprediction rate is relatively high for static prediction schemes. If we assume a prediction rate of 80%, then the rate has already dropped to 35% after four branches! The multiple path technique almost increases the parallelism with a factor of three. Tasks normally initiate only a few new tasks. Therefore, the disjoint eager execution technique can almost always speculatively execute all these child tasks directly at the beginning of the parent task. The control synchronization technique multiple flows of control does not yield significant speedups. The oracle model can perfectly resolve all control dependences as early as needed, *i.e.* speculative execution with perfect prediction. The speedup of the oracle equals almost that of the combination of the multiple flow and multiple path schemes.

The performance of the different data communication techniques are summerized in table 2. All four possible combination of the data schemes are shown for both non-speculative execution and a machine with oracle control synchronization. The machine models which allow non-strict execution perform better than those which release data values as early as possible. Apparently, tasks tend to produce data values at the end and on the other hand, data values are normally not consumed directly at the beginning of a task. The combination of both data communication techniques results in better speedups than the product of the separate speedups. The combination of control synchronization and data communication techniques results in large speedups. In table 2 this phenomenon can be also found: the differences between the various data models are for the oracle synchronization much larger.

control schemes		parallelism
non-speculative	single flow	20.1
speculative	single flow	22.3
multiple path	single flow	63.0
non-speculative	multiple flow	26.3
speculative	multiple flow	26.3
multiple path	multiple flow	65.0
oracle		66.7

data schemes		control scheme	parallelism
strict	late	non-speculative	11.2
non-strict	late	non-speculative	15.0
strict	early	non-speculative	13.1
non-strict	early	non-speculative	20.1
strict	late	oracle	12.5
non-strict	late	oracle	27.7
strict	early	oracle	15.6
non-strict	early	oracle	66.7

Table 2: Average parallelism under different control schemes (left) and data schemes (right).

6 Conclusions

This paper shows that the available coarse-grain task parallelism of sequential programs offers a large potential performance increase for execution on multiprocessors. However, the efficient exploitation of the coarse-grain parallelism requires fast data communication and control synchronization techniques. Because of the large granularity of the tasks, the instruction-level parallelism is completely preserved. In this paper we have discussed the basic data communication and control synchronization techniques and combined them in various machine models. Through a study of these abstract machine models we analyzed the importance of each technique.

The experiments showed that the application of only data communication techniques or control synchronization not nearly resulted in the performance of the machines which combine effective data and control schemes. The difference between simple speculative execution techniques and the speculative execution of multiple alternative paths, like disjoint eager execution, is huge. This behavior is due to the fact that the multiple path technique can normally start all child task at the beginning of the parent task. Comparing the data communication schemes non-strict execution and early releases of data values, the performance of the first is far better. But section 3 we already concluded that the complexity of the non-strict scheme is quite large. The combination of both data communication schemes results in large speedups.

References

[1] H. Corporaal. *Transport Triggered Architectures: Design and Evaluation*. PhD thesis, Delft University of Technology, Department of Electrical Engineering, 1995.

[2] M.S. Lam and R.P. Wilson. Limits of control flow on parallelism. In *Proceedings 19th Annual International Symposium on Computer Architecture*, pages 46–57, 1992.

[3] V. Sarkar. PTRAN - the IBM parallel translation system. In B. K. Szymanski, editor, *Parallel Functional Languages and Compilers*, pages 309–392. Addison-Wesley, Reading, MA, 1991.

[4] Milind Girkar and Constantine D. Polychronopoulos. The hierarchical task graph as a universal intermediate representati on. *International Journal of Parallel Programming*, 22(5):519–551, 1994.

[5] D.E. Maydan, J.L. Hennessy, and M.S. Lam. Effectiveness of data dependence analysis. In *Proceedings of the NSF-NCRD Workshop on Advanced Compilation Techniques for Novel Architectures*, 1992.

[6] Carl Josef Beckmann. *Hardware and Software for Functional and Fine Grain Parallellism*. PhD thesis, University of Illinois at Urbana-Champaign, Centre of Supercompter Research and Development, 1994.

[7] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, 1995.

[8] Pradeep K. Dubey, Kevin O'Brien, Kathryn O'Brien, and Charles Barton. Single-program speculative multithreading (SPSM) architecture: Compiler-assisted fine-grained multithreading. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 109–121, 1995.

[9] Augustus K. Uht and Vijay Sindagi. Disjoint eager execution: An optimal form of speculative execution. In *Proc. of the 28th Annual International Symposium on Microarchite cture*, December 1995.

[10] J. Hoogerbrugge. *Code Generation for Transport Triggered Architectures*. PhD thesis, Delft University of Technology, Department of Electrical Engineering, 1996.