The Potential of Exploiting Coarse-Grain Task Parallelism from Sequential Programs

Jeroen Hordijk and Henk Corporaal

Delft University of Technology
Department of Electrical Engineering
Mekelweg 4, 2628 CD Delft, The Netherlands
e-mail: {J.Hordijk,H.Corporaal}@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 reasonable potential 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 [1]; 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 [2]). 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.

Numerous studies have proven the existence of a large potential of coarse-grain task parallelism in sequential programs, see for instance [3, 4]. 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 [5, 6, 7].

Most of the research 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 [8] 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 from each other [6] [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.

[1] Note that, Girkar *e.a.* only considers control independent function calls, so that the percentage of both data and control independent task is even smaller.

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 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 [9, 7], 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 investigate the solution space for the problems of exploiting coarse-grain task parallelism and analyze the performance of these theoretic solutions using an experimental framework.

The different machine models used in the studies are described in Sect. 2. In Sect. 3, we discuss the experimental framework used to evaluate the limits of parallelism under the different machine models. The performed experiments themselves are discussed and analyzed in Sect. 4. Finally, in Sect. 5 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 Machine Models

In order to analyze the limits of task parallelism, we introduce a number of techniques which solve the problems of coarse-grain task parallelism. The combinations of these techniques 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 briefly analyzed below. For this analysis 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 all dependences are obeyed. Two main approaches can be distinguished: dependent tasks can be sequentialized or the dependences are obeyed by explicit synchronization and communication. Sequentializing 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 synchronization and communication schemes. Data communication between tasks can be viewed from two sides: the definition of data which resulted in the dependence; tasks should be small to reduce the number of inter-task dependences. 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 a table which for each task holds the task it is dependent on and when a task is finished the occurrences 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

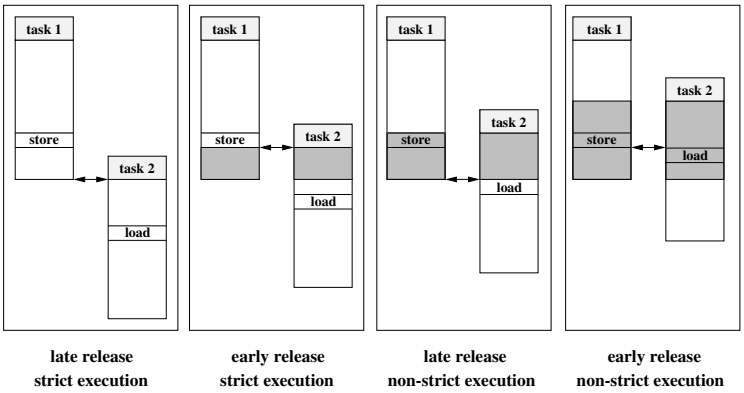

Fig. 1. The four combinations of data communication schemes.

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 too costly. The most obvious 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 usage of 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* or data dependences can 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*. Combining both data communication schemes four different implementations are possible; they are shown in Fig. 1.

Non-strict execution of tasks requires for each load and store an explicit test whether the dependences on this data value are satisfied yet. For strict execution the scheduler satisfies these dependences by spawning only tasks for which all data is ready yet.

In addition to these tests non-strict execution also requires some fundamental optimizations 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 [6, 10]. This requires smaller tasks in order to achieve the same amount of parallelism and smaller tasks result 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 [4]. 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 [3]. 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.

In Fig. 2 the control synchronization for an example code assuming different control flow models is given. The dashed border of a task means that the execution of the task is speculated so that the task needs to be squashed when the prediction is false. The dashed lines indicate the predicted direction of a branch. Task 1 contains an if-then-else statement in which the if part calls task 2 and the else part task 3 (Note that, the granularity of a task is a procedure). In the speculative model tasks 2 and 4 start directly at the beginning of task 1, because the left hand path is predicted. When the multiple path model is assumed, all tasks can start at the beginning of task 1. The multiple flow model uses the control independence of task 4 to start it at the beginning of task 1.

3 Experimental Framework

An experimental framework, called *PARTASK*, has been set up to measure parallelism upper bounds under the different abstract machine models. 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 (using the MOVE framework [11]). 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

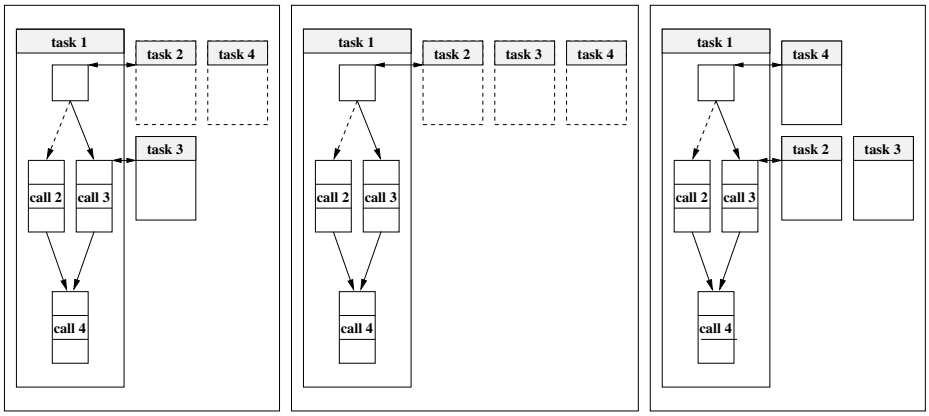

Fig. 2. The control synchronization for the speculative model, multiple path model and multiple flow model.

Table 1. The main parameters for the experimental framework and their default values.

parameter	range	default value
function unit latencies	> 0	1
branch prediction	static or dynamic	dynamic
number history bits branch table	≥ 0	10
number branch table entries	≥ 0	64
intra-task control speculation	none, speculative or oracle	spec
exploitation of task parallelism	yes or no	yes
data communication latency	≥ 0 clock cycles	1 cycle
strictness	strict or non-strict	strict
release time	late or early	late
ignore false memory dependences	yes or no	no
multiple control flows	yes or no	no
inter-task control speculation	none, speculative, multiple path, oracle	none
length main path of multiple path	≥ 0	infinite
number subpaths of multiple path	≥ 0	3
length subpaths of multiple path	≥ 0	infinite

the critical path can be found. Dividing the total number of instructions by the critical path length equals the maximum available parallelism.

We will use the procedures of the program as individual tasks. As shown in [11] 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 [6]. 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.

The assumptions made for the generation of instruction-level parallelism are briefly discussed below. We do not consider any resource constraints, which means that the number of functional units is unlimited 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 most of these spurious data dependences. For data dependences on memory locations anti-dependences and output dependences are obeyed. Anti-dependences and output 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 techniques can never achieve this upper bound.

The speculative execution schemes use a global history branch prediction buffer. The global history register is 10 bits wide, and the table contains 64 entries.

For the exploitation of task parallelism, an infinite number of processors are assumed, discarting the need for a task scheduling algorithm. 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.

The experimental framework supports a wide range of machine models, because many machine characteristics can be adjusted by the user. In Table 1 these parameters and their default values are summarized. Almost all of these options have already been described in this or the previous section. Note that, the control model for the scheduling of instructions within a task, *i.e.* the *intra-task* control model, and the control model used to schedule tasks on processors, *i.e.* the *inter-task* control model, can be varied separately.

4 Experimental Results and Evaluation

This section describes the experiments used to analyze the parallelism for the abstract machine models. In order to keep a clear overview, only a small subset of the experiments will be presented. We will only discuss the most important characteristics of the techniques discussed in Sect. 2. For more detailed performance results see the table in the appendix.

The experiments are performed on the ten benchmarks summerized in Table 2. 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 five 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 Sect. 2. The DATA machine model combines non-strict execution with early releases of data values, so that data dependent 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).

Table 2. The different benchmarks and their characteristics.

benchmark	lines of source code	operations	number of tasks	task size (in ops)	branch accuracy
a68	13565	2805K	31266	90	97%
bison	18962	6630K	31104	213	97%
compress	5596	18K	6866	211	90%
cpp	7657	1983K	8614	230	94%
diff	11424	602K	3699	317	96%
flex	10454	12214K	34068	125	96%
grep	19329	525K	2758	152	97%
od	3132	22000K	161136	128	98%
queen	3407	20K	227	227	76%
wc	1313	283K	126	126	94%
average	9484	4708K	27986	182	94%

In Fig. 3 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 conclusion of Beckmann's study which reported the tightly coupled behavior of procedures [8]. The DATA and CTL models naturally extract more task parallelism than the BASE model, and the DATA model performs somewhat 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 notices that there is a large variation in the difference between exploited parallelism under the *ALL* and *ILP* machine model. In benchmarks like *diff* and *wc* there is little benefit from task parallelism. In contrast to benchmarks like *a68* and *od* where task parallelism highly increases the total exploited parallelism. The benchmark *wc* does not benefit from task parallelism, because all work is done in one procedure.

Table 2 also shows the task size in operations and the branch prediction accuracy. The large average task size of 182 operations guarantees that enough instruction-level parallelism can be exploited within the tasks. The average branch prediction accuracy is suppressed by the low prediction rate of *queen*, without this benchmark the average prediction rate increases to 96%.

The performance of the different data communication techniques are summerized in Fig. 4. All four possible combinations of the data schemes are shown for both a machine without speculative execution and a machine which combines multiple flow techniques with the multiple path scheme. 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 use of both complex control synchronization and complex data communication techniques results in large speedups. The differences between the various data models are for complex control synchronization schemes much larger than for restricted control models. In other words, complex data models are only relevant in combination with complex control models.

In Fig. 5 the performance of the different control synchronization techniques are analyzed by comparing the available parallelism for all six control schemes. All these con-

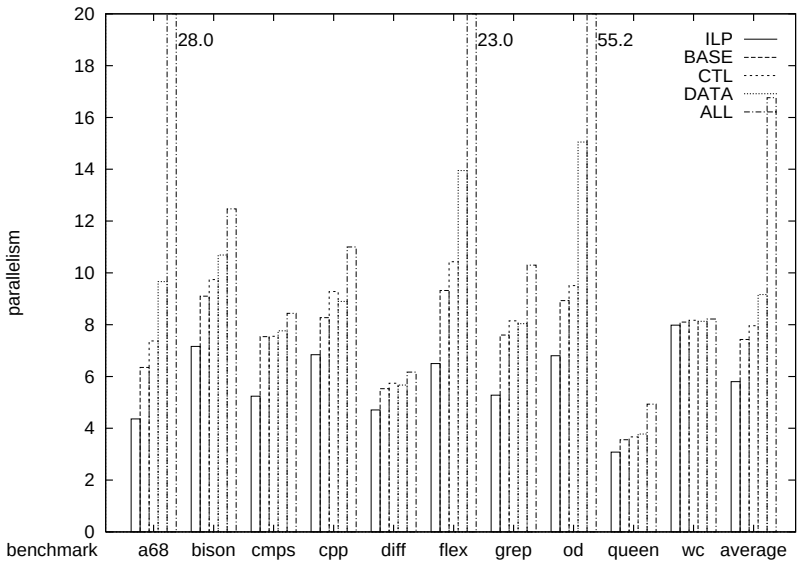

Fig. 3. The exploited parallelism for the four basic machine models.

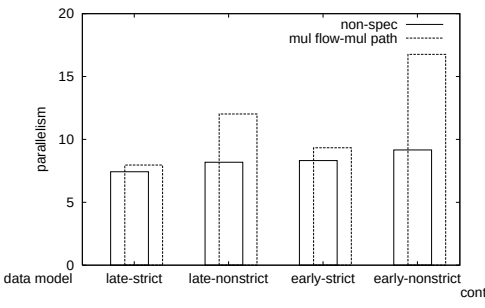

Fig. 4. The parallelism for different data schemes.

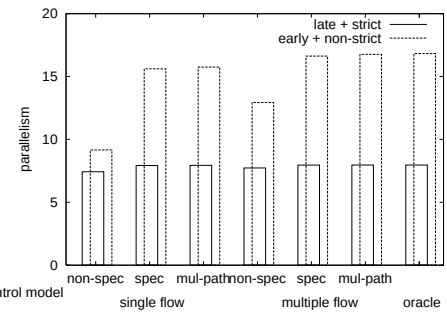

Fig. 5. The parallelism for different control schemes.

trol schemes are combined with both the most restricted data model, *i.e.* the combination of strict execution and late release of data values, and the most complex data model, *i.e.* the combination of non-strict execution and early releases of data values. As a reference also the oracle model is given, *i.e.* speculative execution with perfect prediction.

The figure shows that complex data models result only in high performance when also complex data synchronization schemes are used. This phenomenon was also found in Fig. 4. Speculative execution results in large speedups, because of the high branch prediction accuracy. Therefore the multiple path technique does not result in much speed-up over speculative execution. The multiple path technique also executes side-paths of the predicted main path. When the prediction rate is low, the side-paths have a higher probability and only then the multiple path technique significantly increases the performance. The combination of the multiple flow and multiple path schemes equals almost that of the oracle model, so that little performance loss is expected due to control synchronization.

So far we only varied parameters which influenced the behavior on task level. All ex-

periments used the speculative control model for the scheduling of instructions within tasks. We will now study the effects on the performance when the intra-task control model is changed without varying the inter-task data and control model. In Fig. 6 the task parallelism is plotted for different intra-task control models. The task parallelism is defined as the total parallelism, *i.e.* parallelism when both instruction-level and task parallelism is exploited, divided by the instruction-level parallelism. In the experiments the inter-task data model combines early releases and non-strict execution and the inter-task control model allows the execution of multiple control flows and multiple alternative paths. We observe that the intra-task control model has enormous influence on the task parallelism. A less restricted intra-task control model increases not only the instruction-level parallelism but also the task parallelism. The production of data values consumed by successor tasks takes more time in more restricted intra-task control models, therefore the task parallelism will also decrease. This experiment showed that the exploitation of instruction-level parallelism and task parallelism are closely related; more efficient instruction-level parallelism also enchances task parallelism.

5 Conclusions

In this paper we analyzed the upper bound of available coarse-grain task parallelism. We discussed several data communication and control synchronization models and combined them in various abstract machines. Through a study of these machine models we evaluated the importance of each technique.

The experiments showed that the available coarse-grain task parallelism of sequential programs offers a potential performance increase for execution on multithreaded processors or multiprocessors. However, there is a large variation in the amount of exploitable task parallelism for the different benchmarks. This is partly caused by the fact that some benchmarks spend almost all execution time in one procedure. Therefore a task parallelism exploiting system will need a more complex partitioning algorithm than just scheduling procedures concurrently. On the other hand, decreasing the granularity of the partitioning scheme can possibly decrease the exploitable instruction-level parallelism. Since efficient techniques for the exploitation of instruction-level parallelism have already been developed, preserving the instruction-level parallelism within tasks is very important.

Two major conditions for the efficient exploitation of task parallelism follow directly from the presented empirical study. First, the processor needs both effective data communication techniques and control synchronization techniques in order to exploit a substantial amount of coarse-grain task parallelism. The tasks are tightly coupled; sequentializing data or control dependent tasks will not result in significant speedups.

Now the question is: what are effective data communication and control synchronization techniques ? Comparing the data models non-strict execution and early release of data values, the performance of the first is far better. The combination of both data communication schemes results in even far better speedups, making this an attractive data communication model. However, in Sect. 2 we already indicated that the implementation costs of both schemes are also quite large.

The high accuracy of global history branch prediction results in large performance increase of speculative execution over none speculative execution. Extending the control model with execution of multiple alternative control paths or the multiple flow technique does not result in large speedups. So that, only speculative execution is already effective enough as task control model.

Second, the available task parallelism is highly dependent on the techniques to exploit instruction-level parallelism. A restricted model for intra-task parallelism results also in limited amount of task parallelism even if the inter-task model is quite complex. Therefore, an effective task parallelism exploiting multithreaded processor or multiprocessor needs besides fast synchronization and communication between the processing units also powerful support for instruction-level parallelism.

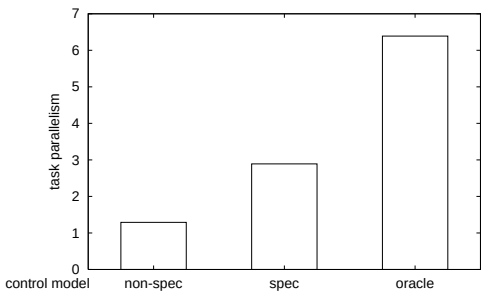

Fig. 6. The speedup of task parallelism exploitation for different control models within tasks.

References

1. B. Ramakrishna Rau and Joseph A. Fisher. Instruction-level parallel processing: History, overview and perspective. *The journal of Supercomputing*, pages 9–50, May 1993.
2. Ken Kennedy. Compiler technology for machine-independent parallel programming. *International Journal of Parallel Programming*, 22(1):79–98, 1994.
3. 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.
4. Augustus K. Uht and Vijay Sindagi. Disjoint eager execution: An optimal form of speculative execution. In *Proc. of the 28th Annual International Symposium on Microarchitecture*, December 1995.
5. 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.
6. 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.
7. 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.
8. 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.
9. 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.
10. Walid A. Najjar, Lucas Roh, and A. P. Wim Bohm. An evaluation of medium-grain dataflow code. *International Journal of Parallel Programming*, 22(3):209–242, 1994.
11. H. Corporaal. *Transport Triggered Architectures: Design and Evaluation*. PhD thesis, Delft University of Technology, Department of Electrical Engineering, 1995.