Introducing Speculative Optimizations in Task Dataflow with Language Extensions and Runtime Support

Nathaniel Azuelos*, Yoav Etsion*, Idit Keidar*, Ayal Zaks*† and Eduard Ayguadé‡
*Technion - Israel Institute of Technology
contact author: nazuel@tx.technion.ac.il
†Intel Corp., Haifa, Israel
‡Barcelona Supercomputing Center and Technical University of Catalunya (UPC)

***Abstract*—We argue that speculation leads to increased parallelism in the coarse-grain dataflow paradigm. To do so, we present a framework for adding speculation in a popular and well-established framework. We specify a limited set of additions to the OmpSs language and changes required in its supporting runtime environment. These modifications enable speculation across the system in a flexible way. We evaluate our implementation using a simple benchmark leading to a promising 10% speedup.**

I. Introduction

The dataflow model is emerging as a main contender in the challenge of parallelism in multi-core and many-core systems. Indeed, the dataflow model relieves the programmer from the burden of managing synchronization explicitly, and promotes the use of automatic runtime environments to schedule execution efficiently. Originally, dataflow paradigms focused on individual data elements, emulating hardware design. Recently, the low cost of on-chip memory coupled with the capacity of high core density lead the design of dataflow paradigms at coarser grain [1], [2], [3].

The OmpSs (OpenMP SuperScalar) language and related infrastructure [2] is one prominent example of a coarse-grain dataflow programming model. OmpSs offers programmers a compact set of non-intrusive language annotations in the form of pragmas to embed in standard serial C/C++ code, similar to OpenMP. Imperative languages such as C/C++ are ubiquitous, but they dictate the use of control flow which does not fit naturally within the dataflow paradigm. Indeed, the interface between control flow and dataflow requires synchronization, limiting graph exploration and dependence resolution. We are thus interested in techniques to remove the synchronization required at control flow points.

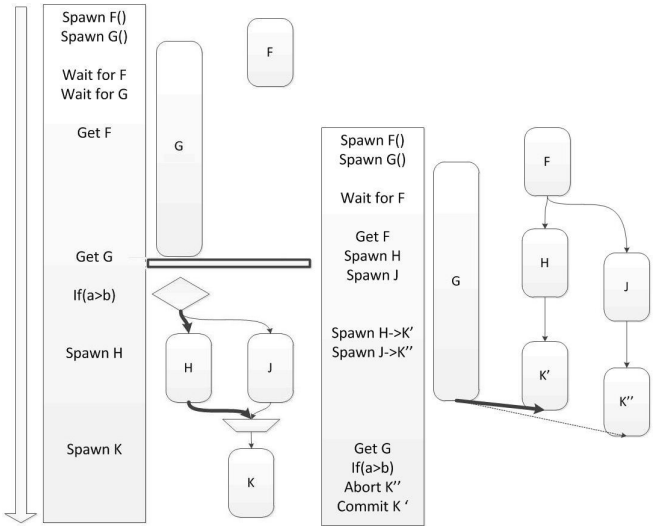
Speculation is widely proven to be a profitable approach for breaking dependences and achieving higher performance. It allows to jump over synchronization points and removing serial bottlenecks, thereby increasing parallelism and performance by Amdahl's law. Branch prediction and value speculation are two techniques that are used to jump over synchronization points in branches and loops. iValue prediction has also been briefly studied [4] in the context of individual variables whose values rarely change.

```
#task output(a)
  a = F();
#task output(b)
  b = G();

#task_wait on(a,b)
if(a>b)
  #task inout(a)
    H(a);
else
  #task inout(a)
    J(a);
}
#task input(a)
  K(a);
```

(a) Code.

```
#task output(a)
  a = F();
#task output(b)
  b = G();

#task spec_on(b) ino ut
     (a) hint(true,
     false)
if(a>b)
  #task inout(a)
    H(a);
else
  #task inout(a)
    J(a);
}
#task input(a)
  K(a);
```

(b) Adding a hint at the top of the branch condition.



(c) Branch condition arriving late.

(d) Both speculative paths are executed. Both branch sucessors' codes are executed. When the condition arrives, one task is committed and the other aborted.

Figure 1. Adding hints to the condition produces two versions. When the parameters to the condition are available, it is evaluated and the data and computation each option generated is either committed or aborted.

These techniques target individual data elements and enable improvements at the level of single instructions, promoting instruction-level parallelism. In contrast, we confront the challenge of promoting task-level parallelization by applying speculation at a coarse grain in the context of a dynamic dataflow environment.

In this extend abstract we outline how speculation can be introduced into a dynamic dataflow environment. We then present our language additions, compiler adaptations and runtime environment modifications to the well-established OmpSs framework of dataflow execution. We explain how input-output relationships can be useful not only for enforcing parallelism and ordering, but also for supporting coarse-grain speculation.

To this end, we argue that resorting to dynamic redirection and multiversionning can fit the dataflow model and allow for simple commit and abort operations. The crux of our idea lies in leveraging the existing dataflow graph to create several speculative versions of the data. Figure 1 exemplifies our approach. On the occurence of a branch, both paths are executed. The output of each path then passes on its output data, to different versions of the *same* symbolic task. We demonstrate our approach running the Huffman algorithm, which shows a 10% speedup in performance.

Our main contribution in this paper lies in presenting how to introduce speculation in a coarse grain dataflow model. We list as further contributions the choices and design decisions made implementing our infrastructure, namely:

- Intuitive language extensions to properly and correctly offer speculation opportunities to the programmer.
- A compiler framework to translate speculation information provided by the programmer and pass them on to the runtime environment.
- A set of runtime environment modifications and additions to implement efficient speculative execution.
- A flexible and extendable runtime environment design so as to allow for ruther speculation techniques than those currently implemented.

II. Background

A. The OmpSs Language

The OmpSs programming standard defines additions to the OpenMP standard to enable a dataflow representation in C and C++ programs. It makes use of pragmas that define tasks with a set of input, output and inout parameters. Typically, a task is described by its input and output and inout sensitivity lists as such:

```
#pragma omp task input(a1, a2, ...) inout(b)
    output(c1, c2, ...)
```

Although variable names are given as arguments to these lists, the dependence information is evaluated at task creation time. In other words, dependences between objects created or modified at runtime are inferred at runtime as well.

The OmpSs language fundamentally supports an accelerator model of computation. A main thread runs through the code until it discovers a graph node description. It builds the graph node before passing it on to the runtime environment. When the task's inputs are available, the runtime environment passes it on to a scheduler that in turn assigns it to an available core.

At every nesting hierarchy level, a small table is kept that keeps in store the list of variables active in the system. Variables are represented in the table according to the address of their data, whether in the stack or heap. To each variable is associated the last graph node to write to it. When the code represented by a graph node completes, it removes itself as las writer. A graph node with no associated last writers found is considered free of the described dependence.

Graph nodes typically go through a lifecycle of

- *birth*, when dependence relations are established
- *wait state*, where each predecessor graph element notifies it of completion until
- *dispatch* where the code block is passed on to a scheduler, and
- *completion*, when the code completes and successor graph nodes are notified.

III. Adding Speculation

Let us quickly go over the two speculation techniques we present in this paper before describing how to best implement it in the system.

A. Branch Prediction

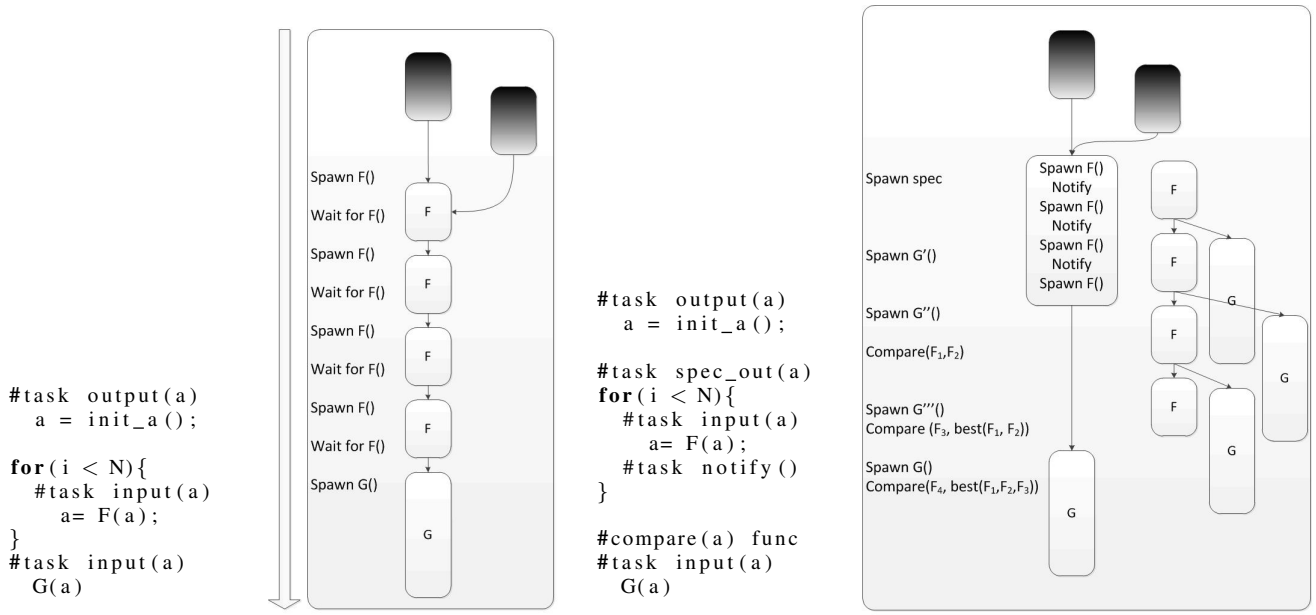
Branch prediction is a well-known technique to imporve performance by speculatively computing the most likely outcome of a branch. In case of misprediction, the system must revert to its original state. The main advantages of this technique is the overlap between the computation of the condition and the code that follows it.

In Figure 1b, we show how the programmer passes on information to the system about the likely outcome of the branch using the 'hint' keyword. We rely on the fact that tasks are self-contained and do not influence variables other than those of their output lists. As such, keeping different copies of the output permits us to run more than one branch option, and possibly several (as in switch-case statements).

Figure 1d shows how in the example, both possible paths are executed. Further, the task that *follows* the branch is also executed using the output of each speculative version. When task G completes, it triggers the system to verify which branch path is the valid one. The valid path is committed and the invalid one aborted, deleting the data it created.

B. Value Prediction

We implement value prediction, following an approach we presented in earlier work [3]. One use of value prediction occurs when some value is iteratively refined; in such



(a) Code. (b) Filtering without speculation. (c) Code. (d) Filtering without speculation.

Figure 2. Speculating on an iterative filter coefficient computation.

cases, it allows programmers to 'perforate' the loop [5] speculatively and compute subsequent stages based on the result of early iterations. A tolerance measure is provided to verify the reliability of the approximation.

Another use of value prediction is one where some information is retrieved from a large data set. In these cases, communication latencies force the system to synchronize on the last read operation to compute this information and pass it to subsequent stages of the algorithm.

Figure 2a shows an iterative computation, where successive iterations of the loop feed one another, updating the same variable `a`. The final version of `a` is passed to task `G`. However, passing earlier versions of `a` to outside the loop allows for G to be computed earlier, as shown in Figure 2c. These earlier versions can be compared between them at the output of the loop variable `F` or `G`, or any successive task. In our example, all speculation attempts *fail*, showing that even in this case, the final result is obtained at the same time as in the original version.

IV. Design decisions and Challenges

Several modifications must be made to the NANOS runtime environment to realize the ideas we propose in this position paper. We propose to augment the NANOS runtime environment with the following features:

- **dynamic pointer redirection** for easy commit of specative tasks;
- **plural speculation** to allow multiple instances of the same task to coexist;
- **versionning of data elements** for concurrency and rollback; and
- **pipeline forwarding** of speculative data through series of tasks.

Dynamic redirection simplifies the commit-abort mechanisms required by speculation. Every data element created on a speculative basis is allocated from the heap. A commit operation then simply reduces to assigning a value to a pointer. An abort becomes a deallocation operation on the data elements speculatively created. In certain cases, we also allow a copy-on-commit mechanism where the result is copied from its temporary location to its originally intended one. The redirection process requires that the actual address of an input be pulled from the dependency graph rather than used as is. This makes sure that a redirected variable's address is used, as opposed to the original one. However, special precautions must be taken if several variables point to the same data. For the time being, we leave this hazard as a warning to the programmer.

We implement redirection using the table of active variables described previously. We leverage the existance of this table to turn it into an address translation table. Every speculative instsance of the variable is stored in the table. There is no fear that a normally variable written to in different places will pass along an incorrect version along its use-def chain, since each task outputing speculative data uniquely identifies the speculative computation. It is this identification mechanism that is used to look up the translation table.

Speculation can often occur along several paths. We choose to allow plural speculation, where the same symbolic task runs several concurrent instances of itself based on different inputs. The outputs of these tasks represent different *versions* of the same semantic data. Multi-versionning has several advantages in a speculative context. In the context of branch prediction, it allows the possibility of computing both options rather than just one. For value prediction, it allows different predictions to be alive at the same time. This can be particularly useful in solution-space exploration benchmarks, where iterative searches could be performed speculatively, thus searching concurrently several solutions in parallel.

An important question in speculation is the extent to which speculative data is propagated. We argue that one important purpose of speculation is not only to avoid waiting on a control-flow condition, but also to bypass sequential bottlenecks in order to reach a more parallel region of the application. Therefore, the output of speculative tasks must continue to feed tasks further down the pipeline. In fact, we rely on this idea to justify value speculation. On the other hand, in order to protect code areas with side-effects, it is desirable to allow the programmer to specify tasks that are not allowed to be speculative. We can thus avoid writing speculative data to disk by placing barriers to the propagation of speculative data.

Memory must be allocated from the heap every time a speculative task is run. This can be a concern as the programmer might define an output on the stack, and deallocating it will cause an error. The compiler should identify such cases and replace all dependencies subject to speculation with heap elements.

There is no need to re-create the dependency graph even if parts of it will be executed several times by different speculation threads. To all extents and purposes, the same graph will be executed no matter if parts of it are speculative or run several times. Speculative instances must run several different and independent code blocks that produce and possibly consume different data elements.

A tracking mechanism of speculative instances is necessary in order to track speculative elements, such as data, graph nodes and active code blocks. We therefore add speculation tracking objects that follow the different elements consequent to a speculation instance. In case of a commit or abort, these objects take care of deleting or approving all the elements they track.

V. Preliminary Results

We have implemented a speculative version of the parallel Huffman algorithm. The algorithmm originally requires building a histogram measuring the frequency of every bytein a data file to build from it a tree representing the optimal compression code. Once the tree is built, the data is compressed on its basis starting from the first element. We use value specualtion to evaluate the graph based on a prefix of the data, enabling compression at an earlier stage. We ran experiments on a simple machine using only four threads. We see a performance increase of about 10% using this technique when reading files from a cold cache. This clearly shows the benefit of overlapping early communication with computation.

VI. Conclusion

In conclusion, we have presented how enabling speculation in coarse-grain dataflow leads to a big potential incrase in performance. We have shown how simple language additions can allow the programmer to make use of speculation. On the back-end, we have outlined the methods required to implement the rutime support for specuative execution. Finally, we presented early results from benchmarks showing a good performance speedup using our technique and infrastructure.

References

[1] C. Kyriacou, P. Evripidou, and P. Trancoso, "Data-driven multithreading using conventional microprocessors," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 17, no. 10, pp. 1176–1188, 2006.

[2] A. DURAN, E. AYGUADÉ, R. BADIA, J. LABARTA, L. MARTINELL, X. MARTORELL, and J. PLANAS, "Ompss: A proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 2, pp. 173–193, 2011.

[3] N. Azuelos, I. Keidar, and A. Zaks, "Tolerant value speculation in coarse-grain streaming computations," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 2011, pp. 490–501.

[4] F. Gabbay and A. Mendelson, "Can program profiling support value prediction?" in *Thirtieth Annual IEEE/ACM International Symposium on Microarchitecture, 1997. Proceedings.*, 1997, pp. 270–280.

[5] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 124–134.