# Hints based Speculative Execution for Exploiting Probabilistic Parallelism

András Vajda
Ericsson SW Research &
Chalmers University of Technology
andras.vajda@ericsson.com

Per Stenström
Chalmers University of Technology
per.stenstrom@chalmers.se

## ABSTRACT

Performance growth of processors will depend on the amount of parallelism exposed by software; at the same time, many algorithms are inherently sequential: they have dependencies that prevent parallelism from being automatically exploited. In this paper we describe a new type of parallelism called *probabilistic parallelism* that can be unlocked in inherently sequential programs using speculative execution based on programmer-provided semantic information. Our preliminary results show promising gains without the need of re-writing applications.

## 1. INTRODUCTION

The computing industry faces a bleak reality: while the transistor count keeps following Moore's prediction, we do not have a reliable method for using these transistors to run software with limited parallelism any faster. Furthermore, clock frequency increases are disappearing as processor cores will have to be more power efficient [1, 2], thus magnifying the challenges of improving the performance of applications with limited parallelism. We are clearly in need of novel methods – and new types of parallelism – that can enable applications with limited amount of inherent parallelism to take advantage of a large number, yet individually simple cores.

Thread-level speculation (TLS) [3, 4, 5] has been explored as a possible path for unlocking additional parallelism. Unfortunately, so far TLS has not been very successful because of the high percentage of squashes, leading to wasted power and limited gains. Consequently, the research community has started to exploit the usage of programmer-provided additional *code annotations* [6, 7, 8, 9, 12], with the aim of removing dependencies that do not matter for the correctness of the program. In addition, the concept of *pre-execution* [10, 11], was earlier applied on the architecture level primarily in order to hide memory access latencies and thus improve the execution speed of single threaded applications.

We build on these concepts, but we apply the techniques at a higher abstraction level; our approach is based on the premise that high-level semantic hints from the programmer, coupled with new run-time speculative techniques can unlock new types of parallelism in applications with limited inherent parallelism. Consequently, the contribution of this paper is twofold. First, we introduce the concept of *probabilistic parallelism:* parallelism that cannot be guaranteed, but may be available with a certain level of certainty; in order to exploit it however, the execution system will need to deploy speculative execution on multiple paths. Second, we show how programmer provided hints can be used to identify probabilistic parallelism and exploit it using thread-level hints based on speculative execution.

## 2. Probabilistic Parallelism

To illustrate the concept of probabilistic parallelism, let's start with the example of Huffman decompression [15]. Huffman compression relies on assigning shorter codes to more frequent symbols, building a binary tree called the code tree where leafs represent symbols from the data stream that is being compressed. Symbols with higher frequency are closer to the root, while those with lower frequency are deeper down and the code assigned to a symbol is defined by the path to the corresponding leaf from the root node. Decoding of Huffman-encoded streams is inherently sequential: there is no reliable way to detect where in the compressed stream the boundaries between symbols are.

However, as the compression algorithm assigns shorter codes for more frequent symbols, it is possible to calculate, based on the code tree, the probability *P* of having a symbol boundary within *N* consecutive bits. Using this information, if we divide up the compressed bit stream into *M* chunks, then with probability *P*, there will be a code boundary within *N* bits distance from the beginning of each chunk (except the first chunk, of course). This is what we call *probabilistic data parallelism*: Huffman decompression can be performed in a data parallel manner, with the constraint that the boundaries of data chunks are only known as a *spatial interval* with an associated *probability*.

Generalizing the example above, we define probabilistic parallelism as parallelism that *may be* available, depending on specific conditions. Probabilistic parallelism is characterized by associated uncertainty factors in terms of

- *probability*: the probability that the parallelism will actually be achieved, within the frame of an
- *uncertainty range*, defined as a set of alternative execution possibilities of which all have to be explored to reach the specified probability level. In the Huffman decoding case, the uncertainty range is defined by *N*, the number of consecutive bits with a probable symbol boundary.

Beside probabilistic data parallelism, we define a second class: *probabilistic pre-execution based parallelism*. Let's consider the case of gzip compressor, included in the SPEC CINT 2000 benchmark [14]. The high-level pseudo-code of the compressor is the following:

```
deflate () {
  while (lookahead != 0) {
        //insert 3-char string into dictionary
        INSERT_STRING();
        match = longest_match();
        if (match) {
          reset_needed = ct_tally_match(match);
          while (matchProcessed) INSERT_STRING();
```

```
else
  reset_needed = ct_tally_match(currentSym);
if (reset_needed)
  FLUSH_BLOCK();
while (lookahead < MIN_LOOKAHEAD)
  fill_window(); } }
```

The functions INSERT_STRING and longest_match can, from a program semantics point of view, be executed ahead of time – *pre-executed* – in each iteration, and the result can be reused at the right moment. However, the second invocation of INSERT_STRING may not be needed at all; the probability of needing its result is dependent on the actual input data. It will be executed speculatively and in case its result is needed, the pre-execution will lead to a speedup of the main program. We will call such a pre-execution probabilistic pre-*execution*; the performance gain is due to the exploited *pre-execution based parallelism*.

Probabilistic pre-execution based parallelism is characterized by the same parameters as probabilistic data parallelism: the *probability* of the parallelism being achieved and, optionally, an *uncertainty range* for the pre-executed code. In our example, the uncertainty range is implicit (the "if" condition) and disregarded: pre-execution will take place, but its result may never be used. In general, a set of potential alternative execution possibilities may be defined for the uncertainty range (e.g. possible values of a variable).

The common characteristic for both types of probabilistic parallelism is that the probability of achieving true parallelism depends on the *nature of the input data*, rather than the algorithm alone. This is a subtle, yet important difference: traditionally, parallelism was extracted based on the nature of the algorithm alone; however, as exemplified above, the likelihood of materializing probabilistic parallelism will not be dependent on the algorithm alone, but also on the input data.

It is also intuitively clear that exploiting probabilistic parallelism requires *speculative execution*, but on a coarser-grained, application level, rather than on the level of the instruction-level architecture. For example, in case of Huffman decoding, the natural way to exploit probabilistic data parallelism is to execute *N* speculative versions for each chunk and retain the result of the correct one (if it was among the alternatives – which should be the case with probability *P*). A detailed description of the concrete implementation for Huffman decompression can be found in [16].

# 3. EXPRESSING PROBABILISTIC PARALLELISM

Probabilistic parallelism can be implemented explicitly, with programmer-controlled management of speculation. However such an approach puts a significant burden on the shoulders of the programmer that may act as a serious show-stopper for practical purposes.

As an alternative, we have developed a hints-based system [16].We use the term *semantic hints* to denote indications which the programmer can use to guide the run-time system during the execution of the application. In practice, a hint is a #pragma like construct inserted into the source code of the application and converted into run-time system primitives during compilation. Such hints don't require a change in the original code; ignoring them will result in sequential execution.

For expressing probabilistic parallelism, we use two types of hints: application-level pre-execution hints and coarse-grained speculative execution hints for probabilistic data parallelism. These hints can be associated with functions, function calls, sets of instructions or individual instructions.

## 3.1 Pre-Execution

From the programmer's perspective, the hint for pre-execution must contain the following information:

- which parts of a program can be executed ahead of time and how to group these into one or several pre-execution groups, expressing the extent to which semantic ordering must be preserved, and
- the probability and optional uncertainty range associated with each pre-executed piece of code

The probability associated with the hint is needed in this case to help the run-time system to prioritize pre-execution in case of execution resource constraints.

Conceptually, a pre-execution hint should result in an execution flow, with the specific probability, where the program still appears to be executed sequentially, but the instructions 'hinted' for pre-execution require virtually zero time, under the condition that the result of the probabilistic pre-execution turns out to be the correct one.

## 3.2 Coarse-Grained Speculation

*Coarse-grained speculation* of parts of the application is a hint that can create probabilistic data parallelism in an application. For example, in the case of Huffman decoding, the hint the programmer introduces is that the application can be run in a data-parallel fashion, but the parallelism is probabilistic: the boundaries of the data chunks are only known as an interval rather than a precise position (the uncertainty range), with a certain probability. The run-time system must speculate, executing several variants of the program as defined by the uncertainty range and then choose the winner of the execution.

The coarse-grained speculation hint provided by the programmer must therefore include the following elements:

- which part of the application exposes probabilistic parallelism,
- what is the uncertainty range, and
- how to choose / decide on the correct result of the speculation (which variant within the uncertainty range was the correct one, if any).

# 4. RUN-TIME SYSTEM

The run-time system to exploit probabilistic parallelism is described in reference [16]; in this section we only provide a brief overview of the key concepts.

The basic primitive exposed in order to support probabilistic parallelism is that of an *execution fiber*, used for modeling both pre-execution and coarse-grained speculative execution. A *fiber* is an isolated thread of execution that executes certain functionality on private data and whose result can be used to help the execution of the main thread. A fiber may be *speculative:* it is executed in isolation and its result is made visible globally or not based on input from the programmer.

Speculative fibers may form *fiber groups.* A fiber within a group must be executed for each member of the uncertainty range

associated with probabilistic parallelism. At the end of the execution of all speculative fibers within a group, an *election* of the winner fiber will take place. The run-time system allows for exactly one or no fiber to be selected as a winner; if there is a winner fiber, the result of that fiber is the successful outcome of the speculative execution, otherwise the speculative execution is considered unsuccessful and the main thread will have to execute the correct version of the code on the correct input data.

## 5. PRELIMINARY RESULTS

We evaluated the applicability of probabilistic parallelism on a TilePro64 processor [13]. The maximum geometric mean speedup for gzip was up to 1.7-fold [16], which can further be improved through the exploitation of data parallelism; for Huffman coding, the mean speedup was 8-fold [16]. Except for one input data type in the gzip case, the speedup was obtained within the same or lower energy budget as for the sequential application. It's also important to stress that the speedup did not required rewriting of the algorithms, other than the insertion of the hints to express probabilistic parallelism.

## 6. CONCLUSIONS

In this paper we introduce the concept of probabilistic parallelism, a new type of parallelism that we show can be used to extract parallelism from applications with limited inherent parallelism. We also show how probabilistic parallelism can be expressed through a simple set of hints and how a speculative execution based run-time system can take advantage of these hints in order to speed up the execution of applications.

Our preliminary evaluation indicates that significant speedup can be obtained within the same or reduced power budget, largely due to the efficiency of speculation driven by hints provided by the programmer rather than extracted by the hardware.

## 7. REFERENCES


[1] Azizi, O., Mahesri, A., Lee, B.C., Patel, J. and Horowitz, M. Energy-Performance Tradeoffs in Processor Architecture and Circuit Design: A Marginal Cost Analysis. *Proceedings of the 37th International Symposium on Computer Architecture, 2010*

[2] Falsafi, B. Dark Silicon and Its Implications on Server Chip Design. *Available at http://parsa.epfl.ch/~falsafi/talks/MSR-2010.pdf* (checked July 2011)

[3] Hammond, L., Willey, M. and Olukotun, K. Data speculation support for a chip multiprocessor. *Proceedings. of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1998

[4] Ohsawa, T., Takagi, M., Kawahara, S. and Matsushita, S. Pinot: Speculative multi-threading processor architecture exploiting parallelism over a wide range of granularities. *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, 2005.

[5] Warg, F, Stenström, P. Improving speculative thread-level parallelism through module run-length prediction. *Proceedings of the Int. Parallel and Distributed Processing Symposium*, 2003.

[6] Prabhu, P., Ghosh, S., Zhang, Y., Johnson, N.P. and August, D.L. Commutative Set: A Language Extension for Implicit Parallel Programming. *Proceedings of the 32nd ACM Conference on Programming Language Design and Implementation*, 2011.

[7] Bridges, M. J. The VELOCITY compiler: Extracting Efficient Multicore Execution from Legacy Sequential Codes. *PhD thesis*, 2008.

[8] Kulkarni, M., Pingali, K., Walter B., Ramanarayanan, G., Bala, K. and Chew, L. P. Optimistic Parallelism Requires Abstractions. *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[9] Vandierendonck, H., Rul, S. and De Bosschere, K. The Paralax Infrastructure: Automatic Parallelization with a Helping Hand. *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010.

[10] Dundas, J. and Mudge, T. Improving Data Cache Performance by Pre-Executing Instructions under a Cache Miss. *Proceedings of the 11th International Conference on Supercomputing*, 1997.

[11] Mutlu, O., Stark, J., Wilkerson, C., Patt, Y.N. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors. *Proceedings of the 9th Symposium on High Performance Computer Architecture*, 2003.

[12] Bridges, M.J., Vachharajani, N., Zhang, Y., Jablin, T. and August, D.I. Revisiting the Sequential Programming Model for Multi-Core. *Proceedings of the 40th IEEE/ACM International Symposium on Microarchitecture (MICRO-40)*, 2007.

[13] Tilera Corporation.TILEPro64 Processor. *Available at* http://www.tilera.com/sites/default/files/productbriefs/TILEPro64_Processor_PB019_v4.pdf (checked July 2011)

[14] Standard Performance Evaluation Corporation. 164.gzip SPEC CPU2000 Benchmark Description File. *Available at* http://www.spec.org/cpu2000/CINT2000/164.gzip/docs/164.gzip.html (checked July 2011)

[15] Huffman, D., A method for the construction of minimum redundancy codes. In *Proc. IRE*, vol. 40, 1952

[16] Vajda, A., Själander, M., Stenström, P. Parallel Execution of Inherently Sequential Applications on Many-Core Processors using Semantic Hints. *Chalmers University of Technology Technical report* 2011:16, ISSN 1652-926X, 2011