Semantic information based speculative parallel execution

András Vajda, Per Stenstrom

Semantic information based speculative parallel execution

András Vajda
Ericsson Software Research
Hirsalantie 11
FI-02420, Jorvas, Finland
andras.vajda@ericsson.com

Per Stenström
Department of Computer Science and Engineering, Chalmers University of Technology
-412 96 Goteborg, Sweden
pers@chalmers.se

ABSTRACT

As the number of cores within one processor will continue to increase, while per-core frequency is likely to stagnate, speculative execution in general and data value prediction in particular, will once again be a prime candidate for speeding up execution of applications with limited inherent parallelism. In order to succeed however, we need to improve the semantic information the applications will be able to provide to the hardware; in this paper we present initial results based on using formal contracts to convey such information to the run-time environment in order to guide the type and scope of speculative execution. Experiments with an application that is traditionally considered hard to parallelize – Huffman decoding – show the potential of such an approach, without the need to re-write the actual application. One of the strengths of this method is that it can make efficient use of large number of cores, opening a possible path for scaling applications with limited amount of parallelism on massively multi-core processors.

1. INTRODUCTION

Multi-core architectures has established itself as the main-stream approach for building processors delivering improved performance according to Moore's law. There are however two major issues that the industry and the research community have to tackle: (1) performance of applications with limited inherent parallelism and (2) increasing unreliability of HW as the industry progresses to newer process technologies.

In fact, in the absence of a universal method for turning software with limited parallelism into software with unlimited parallelism, the computing industry is in the front of a stark reality: while we can keep increasing the transistor count, we simply cannot use these transistors to run software with limited parallelism any faster. Thread-level speculation is an approach that has been well studied [3, 4, 7, 11]; while intuitively appealing, it has not yet been very successful. A major reason for that is that it is very difficult to decide at run-time when speculation will be successful. Relying on run-time information has not offered much gain. On the other hand, limit studies have shown that there is indeed a wealth of parallelism to be exploited [5, 9], therefore speculation as an approach should not be abandoned. Provided that we can apply speculation where it pays off, the upside is that we can use the higher amount of transistors to perform speculative execution of sequential software at radically larger scale than attempted before.

The pre-requisite however, in our opinion, is to strengthen the interface between high level software and hardware, so that hardware can utilize detailed semantic knowledge at large scale to support execution of single threaded applications and improve HW reliability. In this position paper we argue for using the possibilities offered by the design by contract paradigm – in terms of possible and/or realistic execution paths – coupled with speculative, run-ahead execution of sequential programs. We believe that such an approach can exploit chips with thousands of simple cores, running at lower speed for improving the performance of applications with limited amount of parallelism, without the need for rewriting the actual software. We will illustrate the potential of this technique by applying it to decoding Huffman-encoded streams,, a problem usually considered embarrassingly sequential. We will show that significant speedup can be achieved using thread-level speculation within a fixed power budget and without actually re-writing the sequential version of the algorithm.

Relevant previous work includes the Galois system [16], which provides a mechanism for expressing and exploiting semantic commutativity; the software behavior oriented parallelization method [18] using programmer-defined or automatically detected possibly parallel regions as well as the *copy or discard* execution model [17] that provides a mechanism for low cost isolation of speculative, uncommitted data. These methods all fall short however of an approach to providing pervasive and varied semantic information to the run-time system.

2. . THE BUILDING BLOCKS

We build our proposal on two existing technologies that we will briefly introduce below: design by contract and speculative execution.

2.1 Design by contract

The concept of design by contract was introduced in the context of the Eiffel programming language (see [8]). Design by contract is a method for specifying logical constraints on software entities (e.g. functions) in terms of dependencies between variables, side effects, invariants etc. A *contract* is the formalization of obligations and benefits of a given software component, in terms of what does it *expect, guarantee* and *maintain* when it's executed. The original goal of the design by contract paradigm was to improve the reliability of software systems by providing means to verify that the execution is still consistent with a predefined set of conditions (or contracts).

Contracts may cover a wide range of conditions, such as

- Acceptable or unacceptable input values or types (e.g. to a function)
- Return value types, meaning and ranges
- Side effects
- Pre-conditions and post-conditions
- Invariants

The key property of the design by contract paradigm is that it offers formal ways to pass semantic information on software components to the run-time system and HW.

2.2 Speculative Execution

Speculative execution has been explored extensively as a way to improve parallelism, both instruction-level parallelism (ILP) and thread-level parallelism (TLP) (e.g. [3, 4, 7, 11]). One sub-branch of the research focused on data value prediction (e.g., [2]), but with very limited results due to the large overhead in terms of logic, power and the high percentage of squashed execution branches.

We believe the main reason for limited success was that *all* previous approaches relied on methods – such as static code analysis, observation of dynamic behavior – that *did not take into account semantic information* that only the programmer could have provided. There has been only a very limited amount of research focusing on provision of semantic information to the HW (such as [12]).

3. SEMANTIC INFORMATION DRIVEN SPECULATIVE EXECUTION

3.1 Overview

As the number of cores will continue to increase, while per-core frequency is likely to stagnate, speculative execution, especially data value prediction, will in our opinion, re-emerge as a candidate for speeding up execution of single-threaded applications. However, the key to success will lie in *improving the semantic information the applications will be able to provide to the hardware*; we believe this is an interesting new direction worth exploring. While there are certainly several ways of achieving this, we believe one of the promising approaches to consider is to exploit the possibilities offered by the design by contract paradigm. The main reason is that design by contract is a well established method that offers a suitable level of abstraction and formalism that can be translated into definitions that can be interpreted by the hardware.

3.2 Programming Model

The fundamental principle of our proposal is to associate with any software – even not parallelized applications – deployed on a multi-core chip, wherever it's needed, a set of *contracts* that capture semantic information in a formalized and HW readable manner. We believe the following types of contracts are the most relevant:

- Value ranges on input parameters, variables
- Relationships between parameters and variables (for example, "if $x < 0$ then y is always even and less than 20")
- Accessed global, shared memory areas
- Constraints on valid pointer values, valid types for dynamically typed parameters
- Expected loop and array sizes, as well potential splitting options
- Constraints on branch conditions
- Helper computations that can provide, based on input data, useful insights to the expected behavior of the system

An important and useful enhancement (from speculative execution perspective) is the usage of probabilistic contracts (e.g. "with 95% probability, x is either 0, 1 or 2"). The goal is to provide as much as possible *formalized semantic information* to the run-time system and hardware. Instead of trying to parallelize the execution of the software, we aim to *reduce the lead-time* of the execution. We aim to achieve this by limiting the possible – or probable –

scope of the next step in the execution of the software to a subset of possible branches and values that makes it feasible to preemptively execute *all* (or most) of these alternatives. This way, at the moment the respective piece of code comes to execution in the logical flow of the program the execution would have actually already taken place – and hence the software is sped up *without being written in a parallel style* or have been automatically parallelized.

For example, consider the code fragment written in C below. The #CONTRACT statements contain the semantic information the programmer provided, based on the actual problem that is being solved. If f1 is called with e.g. a = 5, the two contracts above the "for" statement will indicate that b will be 0, 1, 2, 3, 4 or 5, with a 90% probability and the *for* loop will have less than 10 iterations. Using this information, it's possible to spawn 60 speculative threads of execution, with a 16% success rate at 90% probability level. For inherently sequential applications this may lead to significant performance gains.

Much of the information provided in this simple example could be obtained through profiling; however, our contracts based approach provides the means to express more subtle, even data dependent relationships and constraints that cannot be derived through profiling, as we'll illustrate with the case study on Huffman decoding. One important such class of contracts is related to the commutability of operations (such as memory allocations): the fact that e.g. two memory allocations can be interchanged cannot be detected through profiling.

```
unsigned f1(unsigned a) {
        unsigned y[50];
        unsigned x, b;
        // some code
        #CONTRACT x < 2*a
        #CONTRACT b <= a PROBABILITY 90%
        for (i = 1;  i <= x; i++) y[i] = f2( b * i);
        // more code  };
```

3.3 Compiler, Run-time System and HW

The first step in turning the semantic information encoded in the contracts into run-time information is compilation. The compiler shall turn the contracts into a separate chunk of code; depending on the nature of the contracts, the compiler may perform a static analysis of the constraints and turn those into a new set of contracts more suitable for execution. In any event, the outcome is a special program that will be allocated to one of the cores that we will call the *contract analyzer core*.

During run-time, the *contract analyzer core* will execute in loose synch with the actual code and will evaluate in advance the contracts relevant for the upcoming branch, loop or function call events and will identify a set of potential – or probable, in case of probabilistic contracts - execution paths that all fulfill the conditions defined in the contracts. Contracts may also instruct the HW how to split up computations, what could and what should not be executed in parallel.

The next step is to select a subset (ideally, all) of alternatives that can be executed in advance – this selection will depend on the number of available cores and can involve previous knowledge (such as previous executions of the same code segment). Once this selection is done, the selected alternatives are dispatched for execution on the available cores – we will call these *speculative fibers*. For each speculative fiber, a new contract analyzer core can be allocated and the same procedure performed in the limit of available cores. In the example from the previous chapter, if f1() is called with *a = 4*, the *contract analyzer core* can decide to execute 50 instances of f2 () (for b = 0, 1, 2, 3, 4 and x = 8), assuming there are so many cores available.

In order to perform speculative execution, two issues need to be addressed: permanent side effects (such as disk access and I/O operations) and duplication of memory areas. These are issues in any speculative approach such as transactional memory (e.g., [1, 4]). However, in order to unlock significant amounts of parallelism, solutions must be scalable and energy efficient.

First, user interaction cannot be performed using speculative execution, hence functions performing user interactions – primarily input operations – cannot be subject to run-ahead execution. For output operations (terminal or disk) buffering followed by commit can be used.

Second, regarding memory, a method for quickly creating private memory contexts with same virtual address space needs to be used. This can be done through e.g. creating a private copy in the local cache of the core executing a speculative fiber of all memory areas accessed during speculative execution. The private memory context can be discarded and need not be committed to memory.

As the main thread of execution progresses, the contract analyzer core will refine the evaluation of

contracts and may squash some of the speculative fibers if the pre-conditions are not valid anymore. Eventually, only one speculative fiber's result will be committed and the same process of run-ahead execution of potential alternatives based on contract evaluation will resume.

3.3.1 HW Architecture Considerations

This method clearly favors usage of simple, low speed cores for three main reasons:

- Speculative fibers are usually relatively simple pieces of code (one program-level function)
- The larger number of cores are available, the higher the number of speculative fibers that can be executed in parallel
- Simple, low frequency cores yield better power efficiency

An asymmetric multi-core architecture [5] with few powerful cores (running the main thread(s) of execution) coupled with several simpler cores (for running the speculative fibers) would be a suitable platform for the method we propose in this paper. The main issue that needs to be addressed is the replication of certain parts of the data memory for each speculative fiber. This will clearly require more on-chip memory per core.

3.4 Potential speedup

Traditionally the issue related to speculative execution was the large number of squashed speculative fibers.

A theoretical model of the possible speedup that can be achieved through speculative execution – Amdahl's law for speculative hardware – provides the following formula:

$$Speedup = \frac{\sum FTime(i)}{M \times Max(FTime(i)) * (1 + SpecMiss)} \times N$$

Where:

- *FTime(i):* is the execution time of one function run (one execution instance)
- *M:* total number of function runs
- *SpecMiss:* amount of speculative fibers that were squash (as % relative to single threaded execution)
- *N:* total number of cores

Applying a simplification (FTime(i)==Max(FTime(i)), the formula becomes

$$Speedup = \frac{1}{(1 + SpecMiss)} \times N$$

In practice, if we can keep *SpecMiss* constant – not dependent on *N* – we can achieve linear speedup of sequential applications *without the need of re-writing.*

The key insight we promote in this paper is the expansion of speculative execution to application level functions and pervasive usage of programmer provided, fine-grained, formalized semantic information – conveyed through contracts – in order to dramatically improve the accuracy of the selection of speculative fibers.

4. IMPROVING HW RELIABILITY

It is probable that HW will be increasingly unreliable. We argue that contracts based approaches can be used to mitigate this unreliability as well. We propose a statistical voting scheme, whereas software would always be executed on multiple cores and the result most often generated would be chosen as the correct answer. This fits clearly into the context of design by contract driven speculative execution, by defining the contract as the 'the result must be the dominant result from multiple executions'. Applications can provide additional semantic information in forms of contracts to steer the HW for electing the correct result.

5. CASE STUDY: PARALLEL HUFFMAN DECODING

Huffman coding [13] is one of the oldest, but still widely used lossless compression algorithms, e.g. in image and video compression (as part of the JPEG and MPEG formats). It relies on building a binary tree where leafs represent symbols from the data that is being compressed; characters with higher frequency are closer to the root, while those with lower frequency are deeper down. Each symbol is assigned a code based on the path to the corresponding leaf from the root: left path means '0', right one means '1'. In short the algorithm relies on assigning shorter codes for more frequent symbols.

Traditionally, decoding of Huffman-encoded streams is considered hard to parallelize. Indeed, it's impossible to split up the compressed stream into chunks, as there are no reliable ways to detect where a new code starts. Previous proposals ([14], [15]) relied on re-designed algorithms, either exploiting that the decompression will re-synchronize or on carefully designed coding schemes that partitioned the code tree into independent structures. All these methods required however re-design of the basic algorithm.

Prior work closest to our proposal is in [19]. The method described therein relies on programmer provided prediction functions that predict the likely outcome of the execution of a subset of the program in

order to guide the speculative execution of the second part. When applied to the Huffman decoding algorithm, this method is similar to [14] and requires modification of the code. In addition, it only provides a prediction mechanism and no other semantic input to speculative execution.

We propose an approach where the basic decoding algorithm is not modified at all; instead, we annotate it with semantic information in form of contracts that the run-time system can exploit for parallel execution.

5.1 Identifying semantic information and contracts

The fundamental question we need to answer is what kind of semantic information could the programmer provide to the run-time system and the hardware that would help speeding up the algorithm?

Even if it's difficult to detect the boundary between two consecutive codes, the programmer *knows* that it certainly occurs within a bit-range equal to the length of the longest code (corresponding to the least frequent symbol). For example, if the largest code is 12 bits long, within a range of 12 bits of the compressed stream, there will certainly be at least one inter-code boundary.

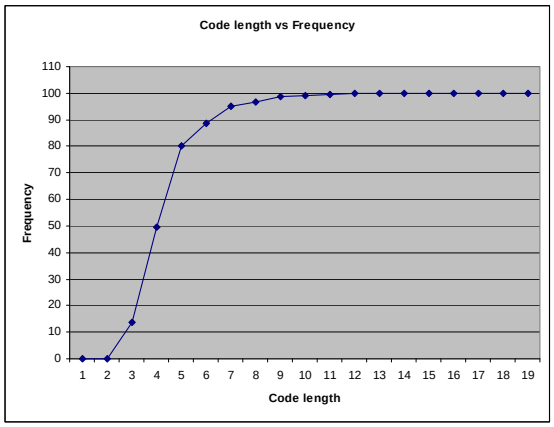

Figure 1 *Code length versus frequency of symbols for the complete Bible*

Using a probabilistic approach, the bit-range for different frequency levels of input symbols can also be calculated; for example, our experiments with different types of texts (scientific, novel and the complete Bible) show that even for maximum code lengths of 19 bits, code lengths of up to 7 already cover 95% of all symbols, while code lengths of up to 9 cover 99% of all symbols (see Figure 1 for the statistics obtained from compressing the complete Bible)

Based on this observation, the task of decoding a Huffman encoded stream may be parallelized by splitting the stream up into several (equal-sized) chunks of compressed code (into several streams); however, to address the issue of unknown code boundaries, for each chunk (except the first one), a number of speculative fibers would be spawned (depending on the desired probability level: in the example above, 7 for a probability level of 95%), each starting at a specific bit position. At least one of these will start at the right position (with e.g. 95% probability), and, since it's executed in parallel with the rest will speed up of decoding.

Thus, a contract expressing the above approach will look like this (using an intuitive pseudo-code description):

#CONTRACT #1

- Calculate tree depth (*TD)* with given probability level (code provided by the programmer)
- Instruct runtime :
    - Execute application on *N x TD* chunks of input data (*N xTD fibers)*

#CONTRACT #2

- Check return values (stop bit) from each chunk
- Instruct runtime which fiber's result to keep

Speculative fibers may be squashed earlier by exploiting the resynchronization characteristic of the Huffman code. The programmer may provide a third contract that checks the results of fibers acting on the same chunk and if two or more of these produce exactly the same result, all but one may be terminated in advance.

5.2 Performance evaluation

Our experiments with various input data sets have shown that, even at only about 95% probability level, there was always at least one fiber (and, in many cases, several fibers, due to frequent short – 3 or 4 bit – codes) that was started at the right position. We ran experiments where we split the compressed stream into 2, 4 and 8 chunks respectively, each executed on 7 speculative fibers; the result was the same for even the largest data set (at close to 4MB of input data), hence a speedup of 100% (two chunks), 300% (four chunks) and 700% (eight chunks) can be obtained.

However, this comes at a cost: the method requires an increasing number of additional cores to achieve this speedup. (for *N* chunks and seven fibers, the method needs (N-1)*7 + 1 cores, thus 8 cores for 2 chunks, 22 cores for 4 chunks, 50 cores for 8 chunks). This obviously highlights one of the drawbacks generally ascribed to speculative execution: waste of (computational and electric) power.

To run the fibers, there is no need for complicated cores. The fibers will act on relatively small pieces of data that could even fit into the core-local cache, thus some features – such as HW threads, out-of-order execution, ILP support etc – can be left out. Therefore, we assume that cores on which the fibers execute can be significantly more power efficient than the original core, at the same frequency and voltage level. In addition, we can apply frequency and voltage scaling to further reduce the power consumption. *Table 1* shows the estimated speedup that can be obtained while keeping the power constant and assuming a factor of 8x better power efficiency for simpler cores.

Table 1*Estimated speed-up factors at different power levels*

Speculative fibers per chunk:		**7**		
Number of chunks	**Required number of cores**	**Speedup without power saving**	**Freq. reduction multiplier**	**Speedup at same power level**
2	8	2	1	2
4	22	4	0,71	2,84
8	50	8	0,54	4,32
10	64	16	0,5	8

Assuming two chunks and eight simple cores, we note that a speedup of two can be obtained without increasing power consumption. If we consider ten chunks and 64 simple cores, a speedup of 16 is possible but that would increase the power consumption. By applying frequency-voltage scaling with a factor of 0.5 we can keep the power consumption at the same level but enjoy a speedup of eight.

These estimations indicate that our method can lead to significant speedup, in case sufficient amount of simpler cores is available, without re-writing the application. These calculations do not take into account the potential savings due to early squashing of speculative fibers.

6. SUMMARY AND FUTURE RESEARCH

In this position paper we argue for research into speculative execution coupled with fine-grained, formalized and pervasive semantic information provided by the application in terms of contracts. By reducing the number of squashed speculative fibers, we believe we can open up for a new approach to parallel programming that does not require parallelization, but enables scaling of performance with the number of cores. Our preliminary experiments indicate that this is possible, showing good speedup on a problem traditionally considered hard to parallelize.

There are several new directions to pursue: exploit semantic information provided by applications, make the speculative execution much more coarse grained (application level functions) and deploy machine learning techniques for improving accuracy.

7. REFERENCES

[1] J. Bobba, K. E. Moore, L. Yen, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood, "Performance Pathologies in Hardware Transactional Memory", *Proceedings of the 34th International Symposium on Computer Architecture*, 2007

[2] M. Burtscher and B. G. Zorn, "Hybrid Load-Value Predictors," *IEEE Transactions on Computers*, 51, 2002.

[3] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *Proc. of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS-VIII '98), pages 58–69. ACM Press, Oct. 1998

[4] L. Hammond, V. Wong, M. Chen, B. Hertzberg, B. Carlstrom, J. Davis, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional Memory Coherence and Consistency", *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pp. 102-113, München, Germany, June 19-23, 2004.

[5] M. Hill and M. Marty. Amdahl's law in the multicore era. IEEE Computer, 41(7):33–38, July 2008.

[6] Md. Mafijul Islam, Busck, A., Engbom, M., Lee, S., Dubois M., Stenstrom, P., Loop-Level Speculative Parallelism in Embedded Applications. In *Proc. of the ICPP 2007*.

[7] T. Ohsawa, M. Takagi, S. Kawahara, and S. Matsushita. Pinot: Speculative multi-threading processor architecture exploiting parallelism over a wide range of granularities. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture (MICRO '05)*, pages 81–92, Washington, DC, USA, 2005. IEEE Computer Society.

[8] Eiffel: The Language, Prentice Hall, 1991.

[9] J. G. Steffan and T. C. Mowry. The potential for using threadlevel data speculation to facilitate automatic parallelization. In *Proc. of the Fourth International Symposium on High- Performance Computer Architecture (HPCA '98)*, pages 2–13. IEEE Computer Society, 1998.

[10] F. Warg and P. Stenström: Limits on Speculative Module-Level Parallelism in Imperative and Object-Oriented Programs on CMP Platforms. In *Proc. of Int. Conf. on Parallel Architectures and Compiler Techniques (PACT'2001)*, pages 221-230, Sept. 2001.

[11] F.Warg and P. Stenstrom. Improving speculative thread-level parallelism through module run-length prediction. In *Proc. of the International Parallel and Distributed Processing Symposium (IPDPS '03)*. IEEE Computer Society, 2003.

[12] Shepelow, D., Fedorova, A., Scheduling on Heterogeneous Multicore Processors Using Architectural Signatures. In *Proceedings of WIOSCA, at ISCA-35,2008*

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

[14] Klein, S.T., Wiseman, Y., Parallel Huffman Decoding with Applications to JPEG Files, *The Computer Journal*, vol.46.

[15] Tsai, T-H., Liu, Ch-N., A Low-Latency Multi-Layer Prefix Grouping Technique for Parallel Huffman Decoding of Multimedia Standards, in *Journal of Signal Processing Systems*, vol. 53, 2008.

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

[17] Tian, Ch., Feng, M., Nagarajan, V., Gupta, R., Copy or Discard Execution Model for Speculative Parallelization on Multicores, in *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture (*MICRO-41), 2008

[18] Ding, Ch., Shen, X., Kelsey, K., Tice, Ch., Huang, R., Zhang, Ch. Software Behavior Oriented Parallelization, in *ACM SIGPLAN Notices*, vol. 42, issue 6, 2007.

[19] Prabhu, P., Ramalingam, G., Vaswani, K., Safe Programmable Speculative Parallelism, to appear in *Proceedings of the 2010 SIGPLAN Conference on Programming Language Design and Implementation (*PLDI 2010), 2010