Tolerant Value Speculation in Coarse-Grain Streaming Computations

Nathaniel Azuelos, Idit Keidar
Technion - Israel Institute of Technology
Electrical Engineering Faculty
Technion City, Haifa, Israel
Email: nazuel@tx.technion.ac.il, idish@ee.technion.ac.il

Ayal Zaks
IBM Research Haifa, and
Technion - Israel Institute of Technology
Haifa, Israel
Email: zaks@il.ibm.com

***Abstract*—Streaming applications are the subject of growing interest, as the need for fast access to data continues to grow. In this work, we present the design requirements and implementation of coarse-grain value speculation in streaming applications. We explain how this technique can be useful in cases where serial parts of applications constitute bottlenecks, and when slower I/O favors using available prefixes of the data. Contrary to previous work, we show how allowing some tolerance can justify early predictions on a scale of a large window of values. We suggest a methodology for runtime support of speculation, along with the mechanisms required for rollback. We present resource management issues consequent to our technique. We study how validation and speculation frequencies impact the performance of the program. Finally, we present our implementation in the context of the Huffman encoder benchmark, running it in different configurations and on different architectures.**

I. Introduction

The current computing landscape has changed a good deal of late. On one hand, many-core CPUs are now commonplace, allowing faster computation of data, particularly on server installations. On the other hand, the set of applications produced by developers is growing in scale and complexity, including an ever-increasing amount of audio and video content, and in ever-greater definition. As a consequence of these developments, the streaming model is an increasingly popular programming paradigm. Several languages, compilers and platforms were developed for this paradigm, such as StreaMIT [1] and Harmony [2] among others. More generic platforms such as CUDA [3] and OpenCL[4] also offer support for the model.

In this paper, we introduce the concept of *value speculation* in the streaming model. In a streaming framework, tasks are often defined in advance (e.g., using a data flow representation), before the data they process is known or available. Speculation allows us to reduce latency by eliminating the need to wait for it all, allowing tasks to optimistically start processing as early as possible based on estimated values of some of their inputs. Such speculative computations offer an opportunity to circumvent Amdahl's Law by optimistically executing tasks that ought to follow sequential bottlenecks.

For example, consider a program that operates in two passes over its input; in the first sequential pass, it computes some statistics over the data, and uses them to define certain parameters for processing the data in the second parallel pass. The parameter computation is a serial bottleneck that cannot be executed before the entire input stream is available. A *speculative* version of the program might guess the parameters based on partial statistics obtained from a subset of the input, and begin executing the second pass optimistically with these parameters on available data.

Nevertheless, it might be difficult or even impossible to accurately predict the parameter values based on a subset of the input. Fortunately, most computations of this nature are not overly sensitive to their parameter values, and can work well with values that are "accurate enough" for the application. To this end, we introduce the notion of *tolerance*, which allows for a margin of error in the predicted value. Of course, if the prediction is off by more than this tolerance margin, the speculative execution must be aborted.

Once we introduce speculative value prediction into a streaming framework, we open up many opportunities for value guessing and speculative processing. In designing a given application using this framework, the programmer can choose how early and how often to speculate as well as how often to verify the validity of the speculation. Another consideration is the relative priority to attribute to speculative tasks as opposed to non-speculative ones during runtime scheduling.

We further discuss speculative value prediction, the degrees of freedom it allows and their implications in Section II.

In Section III, we present a sample implementation of speculative value prediction in the context of the Streaming Runtime Environment (SRE) [5]. Introducing speculation on a software-only platform requires some modifications. First, a mechanism must be introduced to generate speculative tasks. Second, a check condition must be implemented to compare one output to the next. Finally, a rollback mechanism must be added using proper garbage collection.

In Section IV we elaborate on the cases where our technique is useful. We then illustrate in detail how speculative value prediction may be exploited in the context of a practical, real-world benchmark. Specifically, we present a speculative Huffman encoder that predicts the encoding tree based on a subset of the input stream, and uses it to optimistically begin encoding. This application implements a common data compression mechanism when streaming long files; it is also sufficiently complex so as to demonstrate many interesting aspects of speculation.

In Section V we experiment with our Huffman encoder implemented in the SRE framework on two different architectures – x86 and Cell. We show how different parameters impact the latency and execution time. We experiment with two different I/O scenarios: reading from the hard disk that simulates very low I/O latency, and over a tunneled socket connection between two servers. Our results show that speculation can improve average latency by a whopping 51%.

In summary, we present the following contributions:

- we introduce the idea of task-level speculation in a streaming environment;
- we introduce the use in this context of programmer-directed tolerance that trades accuracy for performance;
- we suggest a new interface for programmers to realize speculation on streams semi-automatically;
- we present a proof-of-concept implementation of our speculation framework on SRE;
- we design and implement a speculative Huffman encoder in our framework; and
- we experiment with the Huffman encoder implemented in our framework on two different architectures, and obtain up to 28% speedup in execution time and a 51% reduction in average latency in certain scenarios.

II. Introducing Speculation

Let us provide a quick overview of the streaming model, then elaborate on how speculation is introduced as well as when and how to use it.

A. Speculation in Stream Programs

Streaming applications define a set of computation elements, called *tasks* that process a flow of data. Many tasks are free of side-effects and produce data that is then fed into other tasks farther down the execution path. As a whole, a streaming program defines a *Data Flow Graph* (DFG) that represents how data provided I/O input or produced by tasks propagates from one task to another, according to the dependencies between tasks. We view the DFG as a snapshot of the application's dynamic execution, rather than a static description of the application's code.

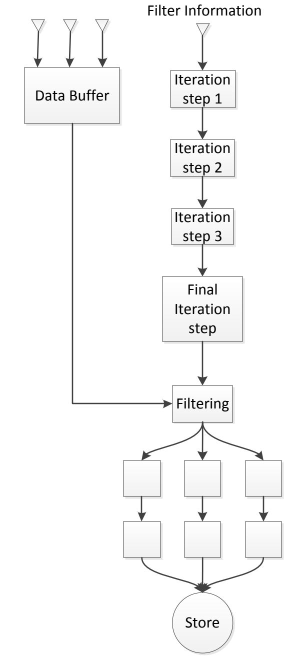
Iterative algorithms such as *k-means* and random-based optimization heurisitcs such as *simulated annealing* are commonly used in large computations, notably in image processing. Figure 1a shows the DFG of an iterative solver that is used to compute the coefficients of a filter, which is then used to operate on a stream of data. The program uses basic information to compute initial values for filter coefficients. These coefficients are then refined iteratively, as they are sent from one stage to the next, making their values more precise in the process. Concurrently, raw data becomes available for filtering. After a given number of iterations, the coefficients are sent to the filter, where they are used to process the data received. The arrows in the figure show the movement of data, and the boxes represent side-effect free tasks. New tasks are scheduled as the required values for their execution arrives. They thus feed one another with respect to their dependencies.

Value Speculation: We introduce a mechanism that allows the scheduling of a task *before* all the data it requires is available. Value speculation can shorten computation time in streaming applications by predicting and providing input values to a task ahead of time. Downstream dependent tasks can then start executing earlier than they would have otherwise. This technique can increase the amount of parallelism available in the application, and mitigate the impact of slow I/O. All predictions are verified; if they fail verification, the speculative execution associated will be rolled back to the original state and recomputed therefrom. To ensure that the original state is recoverable, only tasks that are free of side-effects may execute speculatively. When speculative data arrives at a state-modifying task such as writing to disk or network I/O, it is buffered until the validity of the speculation is confirmed. Keeping speculative tasks free of side effects simplifies rollback; no interim steps need to be saved — the speculative tasks are simply stopped and their data reclaimed. Note that our framework can be extended to support user-defined rollback routines, to enable more tasks to execute speculatively.

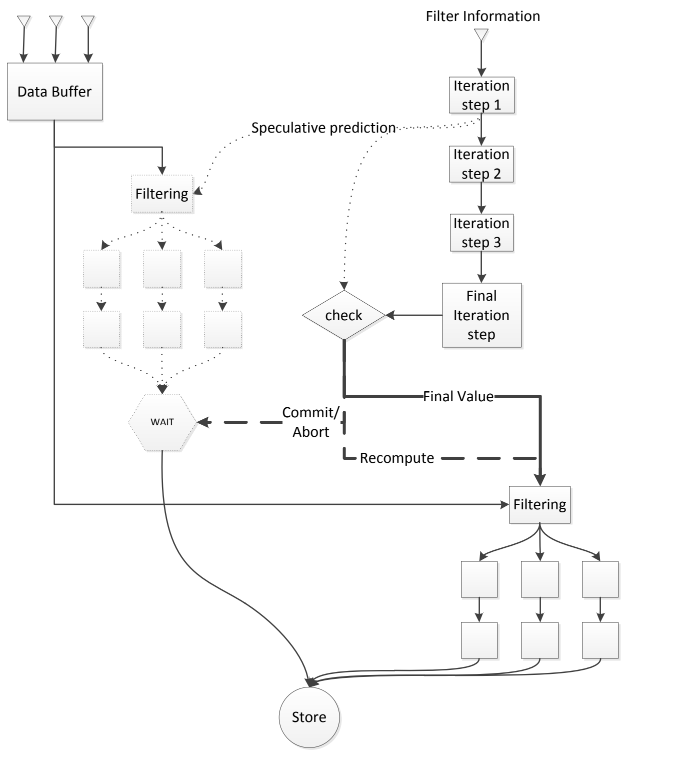
Value prediction can generate new parallelism opportunities, especially in cases where parallel portions of an algorithm depend upon narrow serial portions, thereby addressing the slowdown formulated by Amdahl's law. In the previous example, the computation of the filter coefficients depends on a number of tasks feeding one another serially and cyclically. Predicting an early value of the coefficients can allow the program to reach the parallel filtering phase earlier.

Figure 1b depicts the example of Figure 1a augmented with speculation. An early stage of the filter calculation phase triggers early speculative execution of the filtering phase. The dotted shapes and arrows in the graph mark speculative work. Following filtering and several side-effect free tasks, computed data is buffered (hexagon). Meanwhile, the natural execution path of the application follows its course until the final iteration stage produces its outputs. The predicted data and its true value are now compared (diamond): a positive comparison commits the buffered data; a negative comparison generates a new filtering task that uses the new coefficients to (re)process the data, instead of the speculative processing which is aborted along with any speculative data created.

Our technique is useful in particular in cases where data is



(a) Filtering without speculation.



(b) Filtering with speculation (speculative tasks shown in dashed lines). Upon completion, the speculative task's output is held at the Wait task (hexagon), waiting for validation. The Check task (diamond) verifies whether the speculative value guessed based on Iteration step 1 is close enough to the outcome of the final iteration step. If it is, the speculative computation is committed and there is no need to execute the non-speculative Filtering task. Otherwise, the speculative value is discarded, and Filtering is re-executed in the normal path.

Figure 1. Speculating on an iterative filter coefficient computation.

accessed through slow I/O. If a critical computation requires all data to arrive, it could be beneficial to approximate the result of the computation using available subsets (typically prefixes) of the data, and operate speculatively based on such approximations. This can effectively allow an application to hide some of the latency delaying critical computations.

Tolerance: Typically, value prediction requires anticipating precise values. We argue that it is often sufficient to give a "good enough" approximation of the value. For example, when the mean value of a set is needed, estimates can be obtained faster using only part of the data; these estimates may come sufficiently close to the mean, with respect to the requirements of the application. This introduces the concept of *tolerance*. In our approach, the programmer defines comparison criteria to validate speculated values. For the case of the above-mentioned example, the true value of the filters will be known eventually, and a user-defined criteria can be set to justify the early predictions made.

Interface: In order to introduce value speculation to a streaming application, the programmer provides the following four details to our programming environment:

1) what to speculate: which data elements (typically flowing along an edge of the DFG) to speculate;
2) how to speculate: the source providing approximate data, substituting the data in (1) above (typically a node of the DFG);
3) where (not) to speculate: the point where speculative data need to wait due to side-effects,
4) how to validate speculations: a comparison method, generally a task, applied to both original and speculated values (from (1) and (2) above), controlling the commits of speculative computations (according to (3) above) and rollback operations.

This interface can be supported by a compiler through the introduction of keywords in high-level languages, or simply through the addition of API functions. A compiler can also assist in analyzing tasks to detect potential side-effects, recommending they should not run speculatively (thereby assisting and potentially automating (3) above).

Using the details listed above, the DFG can be modified to support value speculationas follows. Tasks marked to operate on speculative data following (1) (referred heretofore as *speculative tasks*) are connected to receive data from the speculation sources (2). New *checking tasks* are introduced, to check if speculated values are satisfactory ((4) above). These checking tasks also receive data from the speculation sources, and from the original computations as well. Following speculative tasks, results scheduled to flow into other speculative tasks do so freely. However, results scheduled to flow into non-speculative tasks (3) are redirected instead to reach the checking task, where they are potentially buffered. The checking task determines whether to commit pending speculative results, or whether to discard them and initiate re-processing based on the original data.

Granularity: Our approach of employing semantic checks and custom evaluations of approximation tolerance is especially applicable to coarse grain task-level parallelism prevalent in streaming applications. In order to avoid and hide overheads, tasks are generally coarse-grained and represent code with execution time in the millisecond range [6]. As a consequence, each task processes a window of data as opposed to individual values. This granularity of tasks makes the semantic nature of tasks more accessible to programmers, thus facilitating aggressive optimization decisions, including value speculation. These modifications certainly add more work to system design, but can potentially exploit extra, idle resources in order to shorten the critical path of our solution.

B. Managing Speculation

Speculation and verification frequency: In order to manage speculative execution effectively, two distinct parameters need to be handled: speculation frequency — the rate at which we calculate new speculative values, and verification frequency — the rate at which we check if our speculations are not stale. Speculation frequency can affect the *quality* of the speculation, as updating possibly premature speculations based on additional data can improve their accuracy. On the other hand the frequency of verification can impact the *efficiency* of speculation. Verifying only the final value when available may result in potentially large amounts of wasted work and time. Verifying very often may incur overheads. These two options are extreme, and an appropriate compromise must be struck between them.

Resource allocation: The way resources are allocated can also have a profound impact on speculation success and overall performance. Assigning higher preference to speculative tasks may slow down the execution along the natural path, while assigning them with lower preference may limit the benefit of speculation. Setting preferences to speculative or natural paths can be done in several ways, such as a priority scheme, limiting the amount of speculative tasks allowed to run concurrently, favoring a given speculative to non-speculative ratio, or simply allowing speculation only when idle resources are available.

III. Implementation

We implemented our method for speculation on our own streaming framework, and ran it on two different architecture sets.

A. Streaming Runtime Environment

We quickly describe our implementation of a streaming runtime environment (SRE) [5] for task scheduling on a many-core machine. The SRE consists of an API interacting with a task scheduler. The SRE makes the key requirement, common in streaming models, that computation be divided into tasks that are free of side-effects. The SRE is designed to run on different multicore platforms such as x86 systems and the Cell Broadband Engine.

The SRE runs two auxilliary threads in addition to any number of worker threads. The first receives data from a parent application, feeding it into the system. The second is in charge of managing scheduling and directing data. The rest are responsible for executing computational code.

An API defines two high-level C++ classes, the Task and SuperTask. Contrary to classic streaming environment models, our SRE defines a hierarchy of node SuperTasks whose sole purpose is to direct the flow of data between its child Tasks and SuperTasks, and eventually to its parent as it completes. Tasks, on the other hand, represent coarse-grain elements of computation with clearly defined inputs and outputs. As soon as all the data required for a task is propagated from a parent SuperTask, the task is sent to a queue.

The way in which the dispatcher chooses ready tasks for execution is very consequential on the behaviour of the program. If it is to use a first-come first-serve (FCFS) approach, it would tend to focus resources to the beginning of the pipeline at the expense of the end. This breadth-first approach certainly extends latency and tends to be toxic to memory locality. In the opposite case, accessing the end of the pipeline is preferred, but in cases initial conditions are important, the first ready task may be computed last, delaying the whole system.

Our platform uses a priority-based scheduling policy where depth is favored, but uses FCFS for tasks of equal priority. We change somewhat this policy when adding speculation. Value predicting and verification tasks are given highest priority, no matter where they are located in the pipeline; we try to optimize for latency, and these tasks should have a high impact thereupon.

On the x86 version of our platform, a single thread runs on each computation CPU. A simple polling mechanism waits for tasks to be assigned to the thread, and processes them as soon as notified. The CBE is slightly more complex, due to the machine's use of local stores rather than cache. The 256KB local stores are very fast access memory attached to processing units that impose software-controlled memory management. A well-known technique called multiple buffering [7] promotes the overlay computation with communication. It assigns a number of memory transfers to a local store while the CPU works on unrelated data. Our platform uses that technique at the level of tasks, and attempts to overlay four tasks' worth per local store. This limits the size of task memory to 32KB.

B. Speculating and Rolling Back

Supertasks are responsible for associating freshly arrived data with its corresponding task. We append a flag to tasks that produce data that can be a basis for speculation. When this flag is asserted, the SRE understands that it must notify the parent SuperTask of two things: the expected data has arrived and should advance normal program execution, and to trigger a speculative task. Speculative tasks are marked as dashed boxes in Figure 1b.

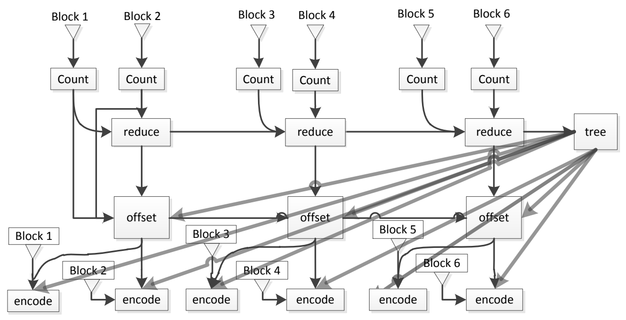
The speculative task is a task that runs like all others, and is treated in much the same way. Its reception triggers new tasks dependent upon it. However, if the same task was launched previously, it indicates that a confirmation is in place, and a special checking mechanism should be used. In the Figure 1 example, the comparison is not trivial, so a checking task is generated. This task will compare the speculative values and will render a verdict, valid or invalid. If the old value is valid, the new value will not trigger anything new and will simply be destroyed.

When speculation fails, several things must occur. First, all the data produced from the speculation point onwards must be discarded. Second, ready tasks must be deleted along with the memory allocated for results. Launched tasks cannot be deleted; the system marks them with an *abort* flag, and deletes them with their content when they complete.

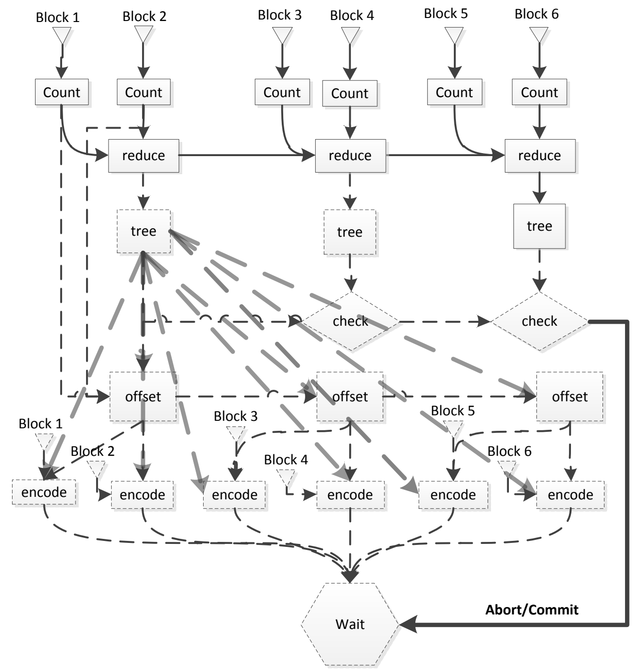
To abort the chain of tasks dependent on the discredited value, the system leverages the description of the relationships between tasks to propagate a destroy signal down the chain of dependences. The absence of task side-effects is clearly beneficial; the dependences to a given point remain the same no matter what happens in the future. We are thus certain that only the correct tasks and their values are destroyed.

IV. The Huffman Encoder Benchmark

Our technique is particulaly relevant to two classes of problems. The first is explained in the previous example, where an early result is extracted from an iterative computation. The second relates to cases where data tricles into the system slowly, and a prefix (subset) of the data can be speculated upon. We focus our experimentation on this second case, as described next.



(a) Non-Speculative Huffman.

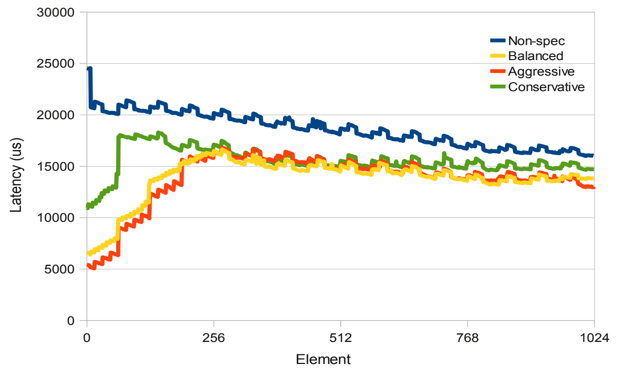


(b) Speculative Huffman.

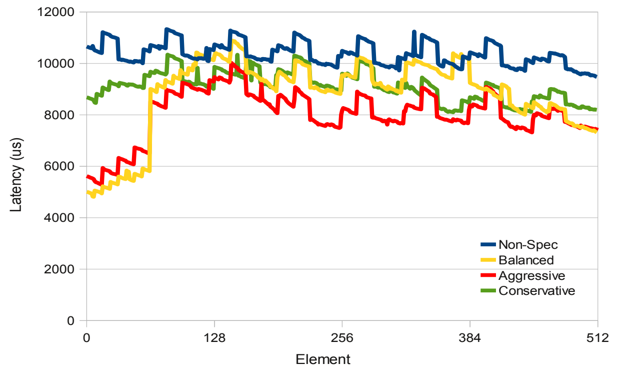
Figure 2. Data flow diagram of the parallel speculative Huffman encoding algorithm (speculative tasks shown in dashed lines). The algorithm makes two passes over its input data. In the first pass, a Count task per input data block creates a frequency histogram of the characters in that block, and the Reduce tasks merge these histograms into one. The outcome of the final Reduce is used to create an encoding tree; speculative encoding trees are created from partial Reduce outcomes. In the second pass, an offset is computed for each data block, and the block is encoded using the encoding tree. Speculative encoding may begin before the first pass completes on the entire data stream. In the figure, trees are created with every new histogram that in turn generate checing tasks. In this example, the first check passes, causing no change in program behaviour. The second check makes the final decision.

A. The Huffman Algorithm

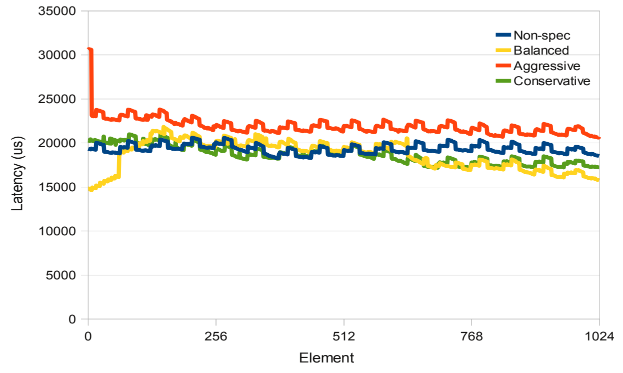
We illustrate the potential benefits of our value speculation approach in the context of data compression. Specifically,



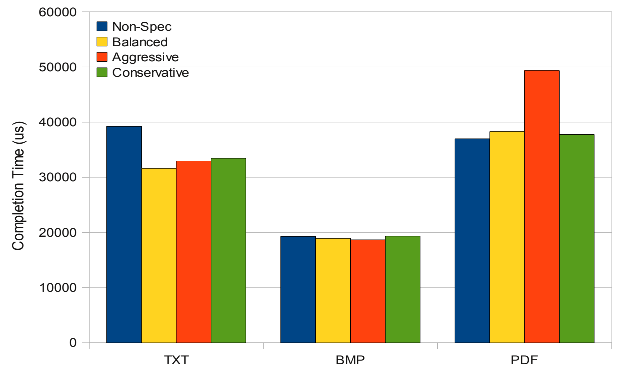
(a) Text file.



(b) BMP file.



(c) PDF file.



(d) Run times.

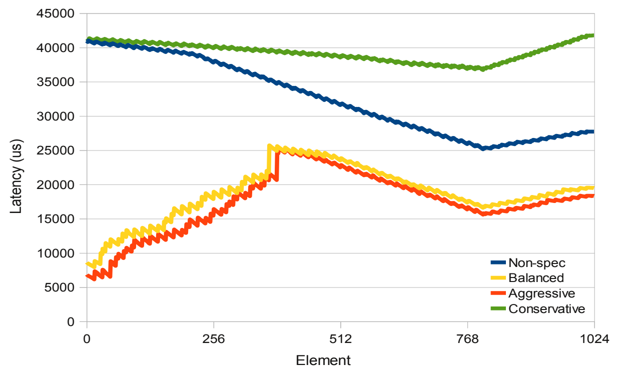
Figure 3. Latency and runtime for different files encoded using *balanced*, *aggressive* and *conservative* dispatching policies on the x86 platform. The *balanced* policy is resilient in the presence of rollbacks (PDF) and performs well in the other cases.

we implement a parallel, speculative, Huffman encoder [8]. A Huffman encoder makes two passes over its input data, (which can be provided as a stream or a file). In the first pass, it calculates a histogram of character frequencies and uses it to construct an encoding tree. The second pass encodes the data using the tree. The tree computation step is a serial bottleneck; once it completes, any number of data blocks may be encoded in parallel.

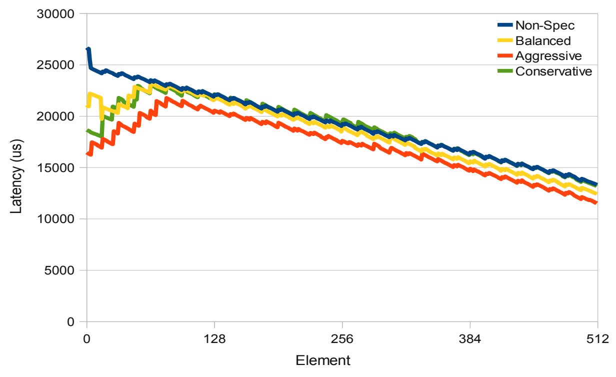
We chose this benchmark for a number of reasons. First, it is an application frequently used in practice– it is a common data compression mechanism when streaming long files. Second, it has a serial bottleneck, which can be potentially circumvented via speculation. Specifically, we can use a speculative histogram (based on part of the input) to create a speculative encoding tree. Third, Huffman encoding is amenable to value speculation with some degree of tolerance– even if the tree is based on a biased histogram, the encoding is still valid, albeit less optimal. The tolerance level thus introduces an interesting tradeoff between compression efficiency and speed. Finally, this benchmark is sufficiently complex to demonstrate the various optimization variables of interest.

We now explain our implementation in more detail. The Huffman compression algorithm requires a few tasks, as illustrated in Figure 2. First, the input data set is parsed in order to count the frequency of each of 256 possible characters encountered. This step is performed by the data-parallel *count* task, and generates a 256-entry histogram. The frequency of each character encountered is used to determine the binary code representing it. All the histograms are summed up into a single one that represents the entire data set. Multiple iterations of a *reduce* task creates this histogram, based on the commutative and associative properties of addition. On the basis of this global histogram, a single *tree* building task creates a binary tree that dictates the encoding pattern. Frequent characters are to be allocated fewer bits. For instance, text files use only around 70 characters (uppercase and lowercase letters, as well as punctuation and digits), allowing at minimum a nearly 3.5x compression ratio.

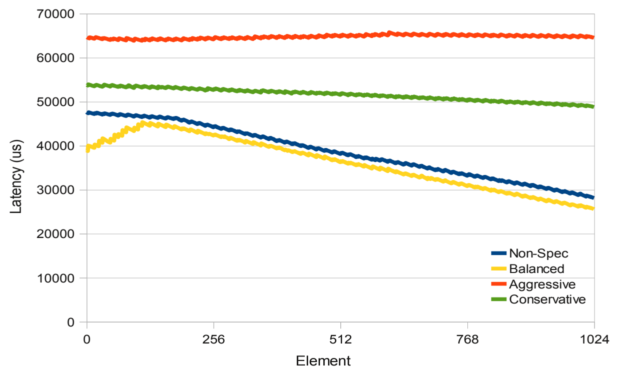
The global histogram offers a basis upon which to build a binary tree where leaves are individual characters. The more nodes are traversed to reach the character, the more bits alloted. Finally, the data set is encoded by *encode* tasks according to the tree, byte by byte. The encoding is variable-length. Hence, the position of an encoded block can only be



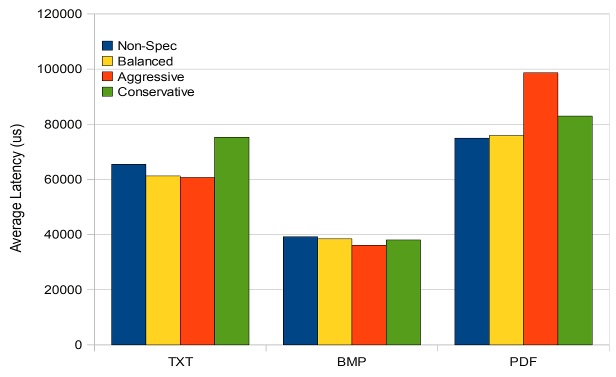
(a) Text file.



(b) BMP file.



(c) PDF file.



(d) Run times.

Figure 4. Latency and runtime for different files encoded using *balanced*, *aggressive* and *conservative* dispatching policies on the Cell platform. *Conservative* speculation yields poor results, whereas the *balanced* policy remains efficient.

known once the previous one's encoding is decided. This makes the use of parallelism in the encoding phase more difficult.

The encoding phase is parallelized by adding an extra phase to the algorithm. Its sole purpose is to compute the offset of each data blocks. It computes the offsets based on the block-specific histogram computed first, the Huffman tree, and the final offset of the previous block. Offset computations feed many encoding tasks.

In order to build a global frequency histogram, the whole data set must be traversed. This certainly causes a problem with respect to data locality, and forces an I/O limited operation to complete before the global tree may be built. Obtaining the global histogram in order to build the Huffman tree is a reduce operation ripe for speculation. The more data comes in, the more solid the foundations of the histogram and thus the tree, and the less likely it is to change. This situation is well suited to the tolerance-based speculation technique we propose.

B. Speculation with the Huffman Encoder

The Huffman encoder is *not* well suited for *exact* value speculation; as time progresses, the common characters will find a more important position on the histogram, but less common ones (z and w in text files, for example), will tend to shift with respect to one another. This has little effect on the quality of the approximation, but can have a large effect on its suitability to speculation.

Our *check* task checks if the difference in compression size is within a certain percentage of the compressed file. It does so by using the current global histogram to sum the product of the frequency of each character with the number of bits associated to it by each tree. When the difference in compression size is larger than a certain percentage of the new compression rate, the verification yields a negative result, and rollback ensues. *Check* tasks are simple and run very quickly.

V. Results

A. Experimental Setup

We ran our experiments using the SRE on two different architectures. The first is a 8xQuad-Core AMD Opteron(tm) Processor 8356 shared-memory CMP multiprocessor system running at 2.3GHz with 132GB RAM. The second is a Cell Broadband Engine blade running at 3.2GHz with 1GB XRAM. In both cases, we use 16 worker threads. We tested two modes of input processing:

1) reading from a hard disk cache, and

2) where data is streamed via a tunneled SSH socket connection over a long distance.

We encoded three types of files: an e-book text, a Windows bitmap (BMP) file and a PDF file. In order to simplify the comparison between benchmarks, the encoder parses 4MB of both the text and PDF files, while parsing only 2MB of the BMP file.

The text file demonstrates the advantages of speculation in no-rollback scenarios. BMPs and PDFs generally have a high entropy resulting in frequent rollbacks. Our main evaluation criterion is per block *latency*. We measure it by subtracting the time a data block arrives from the time we complete its processing. In this manner, we discount data transfer time. When testing the socket connection however, we observe data arrival time in addition to latency. We include this extra criterion in order to show the dramatic impact of rollbacks in slow I/O situations.

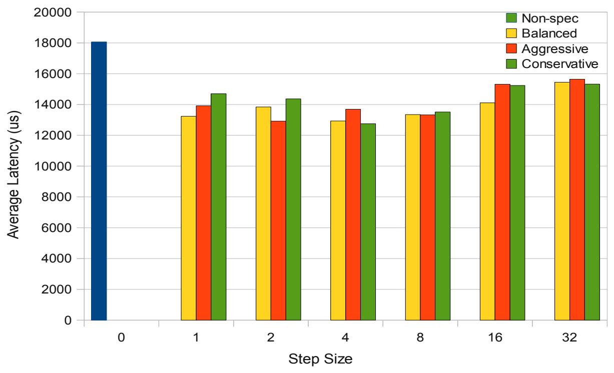
Parametrization: Our implementation of the Huffman algorithm uses different configurations in the disk and socket cases. When reading from disk, the low input latency favors large reduce operations, in opposition to the socket reading case.

The source data is first broken into 4KB blocks, each processed by a seperate *count* task. When reading from disk on the x86 platform, each *reduce* task merges the histograms produced by 16 *count* tasks, and each *offset* task feeds 64 encode tasks which conclude the process. Due to the limited amount of local store on the Cell platform, 16:1 ratios are used there in both cases. When reading from a socket, both *reduce* and *offset* ratios go down to 8:1 in order to reduce average latency. The baseline configuration verifies speculation upon reception of every eighth result of a *reduce* task histogram, a choice explained later. A tolerance margin of 1% of the compression ratio is used.

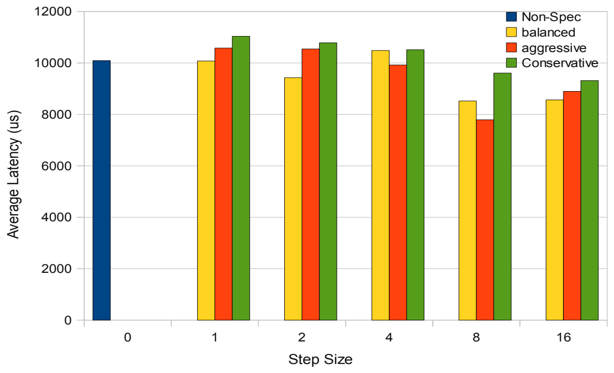
B. Reading from Disk

Scheduling Policies: We integrated three resource allocation policies in our framework. In the first *conservative* policy, we give priority to the natural execution of the algorithm. Speculative tasks are dispatched only when no non-speculative ones are available. The second *aggressive* algorithm actively prefers any speculative task over non-speculative tasks. Finally, the third favors dispatching an equal number of speculative and non-speculative tasks. We denote this policy as *balanced*. Being aggressive is likely to help scenarios where no rollbacks occur, but is likely to be more costly when rollbacks do occur. Conversely, a conservative policy will only exploit free resources and miss opportunities to execute later pipeline stages early.

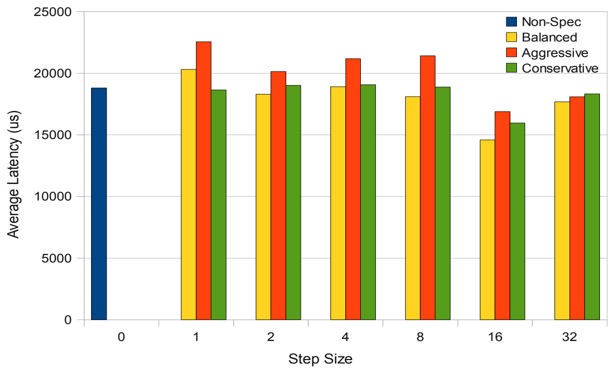
Figure 3 presents sample execution of these three policies and compares them with a typical non-speculative run. It can be seen that the *conservative* and *balanced* policies generally perform better in the PDF case, as frequent rollbacks have a smaller impact on the use of machine resources. However,



(a) Text file.



(b) BMP file.



(c) PDF file.

Figure 5. Choosing the speculation interval determines when speculative execution starts. The earliest it does, the more benefit (TXT) with some price in overhead. A larger interval can jump ahead of transient states and avoid rollbacks altogether. In these plots, a step size of 8 for BMP and 16 for PDF represent this threshold.

being aggressive can be a good choice when no rollbacks occur, as in the case of the text file. In any case, it can be observed that the policy of *balanced* dispatching is generally the best in all cases, as it combines the benefits of being aggressive when no rollbacks occur with the resiliency of the *conservative* policy when rollbacks do occur.

Figure 6d shows the dramatic impact on total runtime speculating properly can bring by bypassing the serial bottleneck of the application. When processing the text file, speculating early and correctly brings as much as 19.5% speedup in runtime. This is not the case where rollbacks do occur, but *conservative* and *balanced* policies manage to offer a runtime similar to the non-speculative case.

Figure 4 shows the same examples but on the Cell platform. We observe the same phenomena as for the x86 platform, with the exception of a rather poor performance by the *conservative* policy. This is probably due to the longer dispatch queue required by the multiple buffering technique we use. It seems this deep pipeline always offers some non-speculative task, and little speculation is done overall.

Speculation Frequency: We then study the impact of changing the speculation frequency in different scenarios. Figure 5 shows the impact of different step sizes for each dispatching policy and benchmark. For the text file, small step sizes perform similarly, but there is a drop in efficiency as they get larger. This is due to the fact that successful speculation starts later in time, delaying data processing. The BMP and PDF files show an interesting pheonomenon. For small step sizes rollbacks occur and performance is poor, insomuch as it is similar to the non-speculative case. However, when the step size grows beyond a certain threshold, rollbacks do not occur any more and the average latency drops significantly.

Figure 5 also demonstrates the impact of choosing well the speculation interval and method. For the PDF and BMP cases, average latency can be reduced by as much as 22%, and by 28% for the text file.

Given the observation that launching speculative tasks early is an important factor for reducing latency, we compare two additional extreme cases in verification frequency. First, an overly *optimistic* approach which speculates based on the first tree available (from the first *reduce* task) and applies but a single comparison, verifying the speculation only when the final tree is available. In the other extreme case, we verify at every opportunity and re-start speculative execution immediately when failure is detected. We term this policy *full* speculation.

Figure 6 shows sample runs for these two cases, along with our baseline and non-speculative policies. When no rollbacks occur, such as for the text and BMP files, being optimistic allows speculation to start the earliest, with virtually no overhead caused by *checking* tasks. *Full* speculation allows speculation to start at the same time as the *optimistic* case, but includes the maximal amount of overhead. The small difference between the curves shown in Figure 6 indicates that checking has a relatively low impact on performance.

The PDF example is interesting as it shows that when rollbacks do occur, the impact on performance is significant in both the *optimistic* and *full* cases. In the optimistic case, a large amount of computation has to be re-started. In the *fully* speculative case, rollbacks occur frequently along with

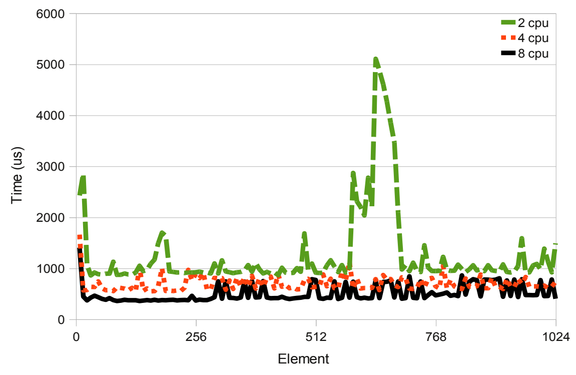


Figure 8. Even with large communication delays, latencies are still reduced significantly with an increased number of CPUs.

the penalty they incur. Nevertheless, in situations with no rollback, optimistic runs can reduce average latency by as much as 51% for the text file on the Cell platform.

C. *Reading From a Socket*

Figure 7 shows the latency and arrival time in the extreme case where long I/O delays clearly show the benefit of speculation. In Figure 7a, where no rollback occurs, latency is essentially negligible with respect to the transfer time. Where a rollback does occur as in Figure 7b, its impact is very clear. It is shown by the flat portion of the curve, where all the data blocks already available are almost instantly encoded. A new and more correct version of the tree is then found, and blocks are encoded as they arrive.

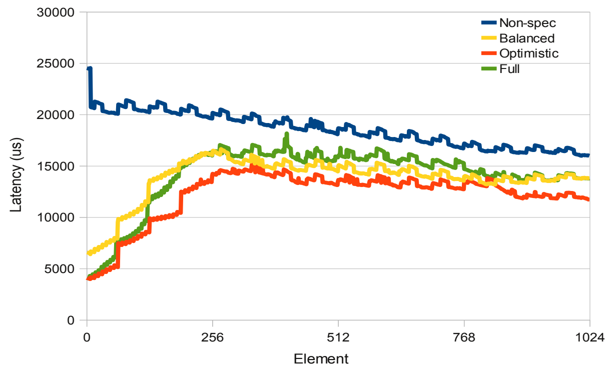
The traces shown in Figure 7 use disproportionately slow I/O, leading one to believe there is little interest in using multiple cores to solve our problem. However, Figure 8 shows that adding CPUs does indeed reduce latency.

Tolerance: Finally, Figure 9 shows how modifying the tolerance margin in our benchmark affected results. Somewhat surprisingly, *increasing* it produced *poorer* results both in aggressive and conservative cases. When the margin was raised from 1% to 2%, performance decreased dramatically. This further emphasizes the importance of detecting an error early to get better results. When raised to 5%, no rollbacks occur any more, and results are as optimal as can be hoped forin this experiment.

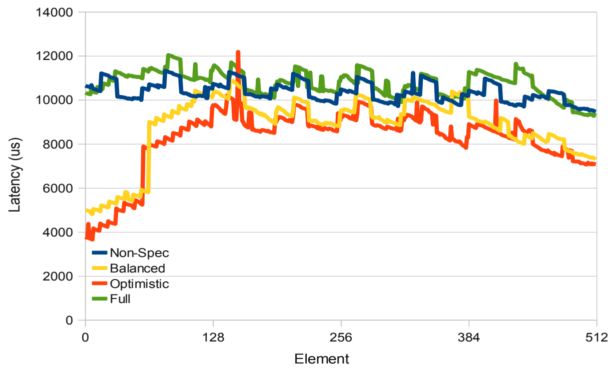
VI. Related work

Our work relates to two well-established topics: value prediction and thread-level speculation (TLS).

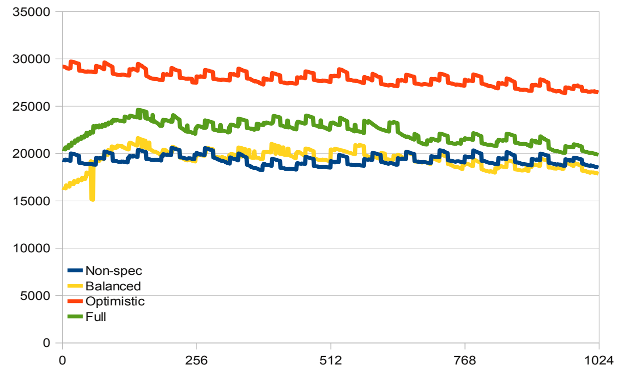
Data speculation: The desire not to wait for certain data to be available, but instead predict its value and initiate early speculative computations based on predicted values, is the subject of extensive work on Value Prediction, initiated by Lipasti [9] and implemented in various forms [10], [11], [12]. Value prediction concentrates on fine grain parallelism,



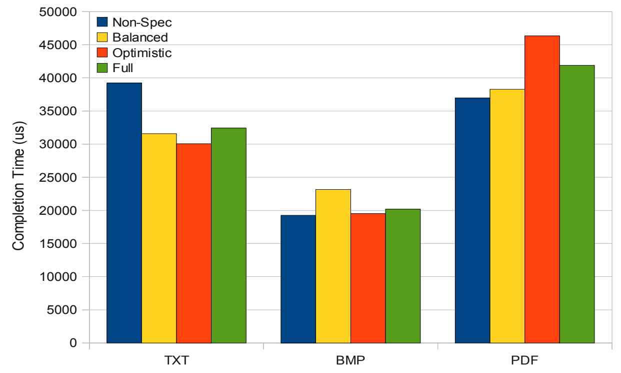
(a) Text file.



(b) BMP file.

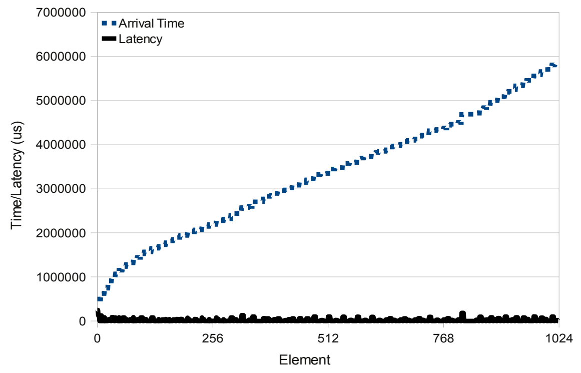


(c) PDF file.

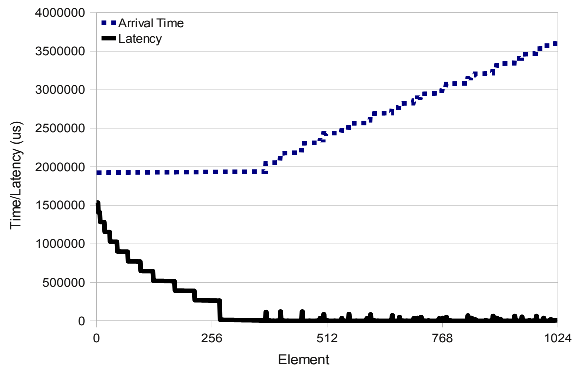


(d) Run times.

Figure 6. Varying verification and speculation frequency on the x86 platform. Optimism pays off in the absence of rollbacks but is very expensive in the opposite case. The small difference between *fully* speculative and *optimistic* policies indicates that *check* tasks cause low overhead.



(a) Text file.

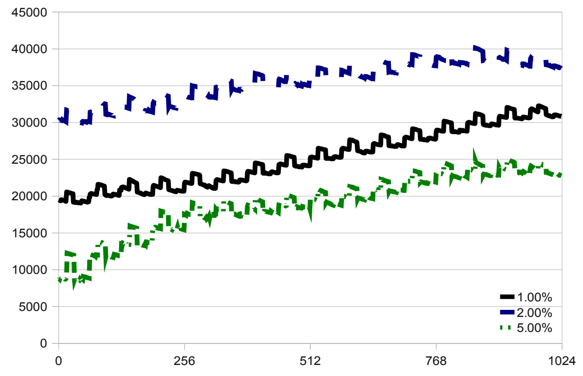


(b) PDF file.

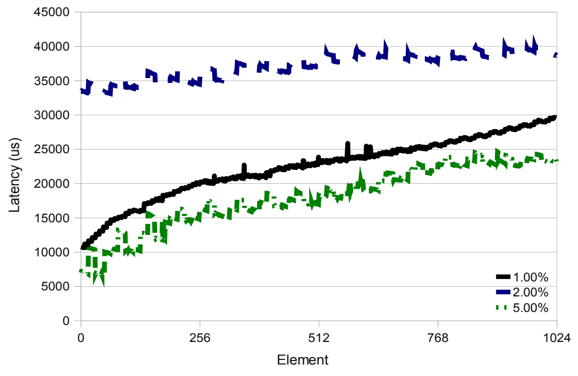
Figure 7. Encoding over a socket I/O connection

attempting to automatically predict register values based on past values. This is akin to branch prediction, and indeed relies on hardware support to detect misspeculation and perform rollback. Our approach predicts values based on approximation computations at a coarser, semantic grain, leveraging structural properties of streaming computations, without requiring hardware support.

Thread-Level Speculation: TLS attempts to execute multiple, potentially dependent computations speculatively in parallel, hoping no data or control dependencies occur in practice, rolling back their execution whenever they do. The TLS concept has been an active area of research, from being proposed to leverage chip multiprocessor capabilities [13], to underlying the increasingly popular transactional



(a) Text file.



(b) PDF file.

Figure 9. The impact of different tolerance percentages on one PDF file in different configurations

memory paradigm. It has been implemented in the context of decoupled software pipelining [14], where large dependence distances are broken speculatively to increase pipeline parallelism, relying on transactional memory support to abort mis-speculation. A similar notion of Kernel Level Speculation was proposed in the context of accelerators and GPUs [15], using speculation to overcome potential control dependencies among kernels thereby increasing concurrency. Our approach is complementary, complying with true data dependencies by trying to guess their values, rather than hoping they will not occur.

Several mechanisms for efficient roll-back in case of misspeculation have been devised, such as *Copy or discard* [16] and that of *Program demultiplexing* [17]. We employ a similar roll-back mechanism, and our framework could alternatively use other solutions.

Recently, some work has demonstrated how the use of extra cores can be used to speculatively run alternative implementations of code segments. Two versions of program code are created, one faster than the other. [18] creates faster versions by using unsafe optimizations, whereas [19] creates slower ones by integrating runtime checks. These approaches are different than ours in several ways. They improve the speed and reliability of sequential as opposed to parallel workloads. This reduces the intensity of the competition between natural and speculative work. Further, we do not artificially modify the speed of any single serial code segment. Finally, they do not include mechanisms that account for tolerance.

The use of approximation to reduce latency has been explored in database systems for some time [20], but to the best of our knowledge it has not yet been used in general compilation settings. Our approach also uses approximate answers when computing exact values is time consuming, and involves semantic checks to evaluate the approximation quality. Applied with tolerance, this opens new opportunities to accelerate applications by revealing coarser-level concurrency.

Rinard *et. al* [21], [22] has demonstrated how a compiler framework can reduce on purpose the accuracy of the applications we target by skipping refining loop iterations. They further demonstrate how using a margin of tolerance can significantly reduce the power consumption of an application, negating the main argument against speculation. However, their measure of accuracy remains fixed at compile-time and does not take into account poperties of the dataset.

Finally, Prabhu [23] defines the main requirements needed for coarse-grain value speculation and integrates them into a formal language. However, there is no implicit margin of tolerance involved, and misspecualtions are binary.

VII. Conclusion

In conclusion, we presented a promising approach to reduce latency through value-based speculation in the context of dynamic streaming frameworks. We further introduced the notion of application-defined tolerance, allowing slack in value prediction. Our technique can rely on the use of prefixes as well as on early results of iterative computations. A proof-of-concept prototype of our idea was implemented using the SRE framework. We illustrate the significant potential benefit of speculative value prediction by showing how a Huffman encoder can take advantage of this idea to circumvent a sequential bottleneck. We experimented with the Huffman benchmark running with our prototype implementation on two different architectures: x86 and Cell. From our experiments, we learn the following conclusions:

- Speculative value prediction can help address Amdahl's Law by bypassing sequential bottlenecks. This may reduce latency — our experiments show improvements of up to 51% in some cases, and shorten execution times by up to 28%.
- Speculation should not be applied too aggressively so as to significantly delay the normal (non-speculative) execution path.
- On the other hand, it is typically worthwhile to begin speculating early; giving speculative tasks a head start

maximizes the opportunities for parallelism. This offsets the additional overhead induced by verification of such early speculations.
- Perhaps counter-intuitively, higher tolerance does not always lead to better performance.

We believe that coarse-grain tolerant value speculation can reveal additional vital parallelism opportunities for more applications and platforms to come, subject of ongoing and future work.

Acknowledgments

We would like to thank the Barcelona Supercomputing Center for granting us access to Cell Blades. This work is partially supported by the Intel Corporation and the Semiconductor Research Corporation. We would also like to thank Prof. Eran Yahav for his help.

References

[1] W. Thies and S. Amarasinghe, "An empirical characterization of stream programs and its implications for language and compiler design," in *PACT '10: Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. New York, NY, USA: ACM, 2010, pp. 365–376.

[2] G. Diamos and S. Yalamanchili, "Harmony: An execution model and runtime for heterogeneous many core systems," in *Proceedings of the 17th international symposium on High performance distributed computing*. ACM, 2008, pp. 197–200.

[3] "CUDA: Compute Unified Device Architecture," `http://www.nvidia.com/object/cuda.html`.

[4] A. Munshi, "The OpenCL Specification," *Khronos OpenCL Working Group*, pp. 11–15, 2009.

[5] N. Azuelos, "An Integrated Functional Solution to Multi-Core Programming on the Cell Broadband Engine," Master's thesis, 2009.

[6] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta, "Hierarchical task-based programming with starss," *Int. J. High Perform. Comput. Appl.*, vol. 23, no. 3, pp. 284–299, 2009.

[7] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, "The potential of the cell processor for scientific computing," in *Proceedings of the 3rd Conference on Computing Frontiers*. ACM, 2006, pp. 9–20.

[8] D. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the I.R.E.*, pp. 1098–1102, September 1952.

[9] M. Lipasti and J. Shen, "Exceeding the dataflow limit via value prediction," in *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 1996, pp. 226–237.

[10] L. Hammond, M. Willey, and K. Olukotun, "Data speculation support for a chip multiprocessor," *SIGOPS Oper. Syst. Rev.*, vol. 32, no. 5, pp. 58–69, 1998.

[11] J. Steffan and T. Mowry, "The potential for using thread-level data speculation to facilitate automatic parallelization," in *HPCA '98: Proceedings of the 4th International Symposium on High-Performance Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 1998, pp. 2–13.

[12] P. Marcuello, J. Tubella, and A. González, "Value prediction for speculative multithreaded architectures," in *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 1999, pp. 230–236.

[13] J. Oplinger, D. Heine, S. Liao, B. A. Nayfeh, M. S. Lam, and K. Olukotun, "Software and hardware for exploiting speculative parallelism with a multiprocessor," Stanford, CA, USA, Tech. Rep., 1997.

[14] N. Vachharajani, "Intelligent Speculation for Pipelined Multithreading," Ph.D. dissertation, 2008.

[15] G. Diamos and S. Yalamanchili, "Speculative execution on multi-GPU systems," in *24th IEEE International Parallel & Distributed Processing Symposium, Atlanta, Georgia, USA*, vol. 4. Citeseer, 2010.

[16] C. Tian, M. Feng, V. Nagarajan, and R. Gupta, "Copy or discard execution model for speculative parallelization on multicores," in *Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2008, pp. 330–341.

[17] S. Balakrishnan and G. S. Sohi, "Program demultiplexing: Data-flow based speculative parallelization of methods in sequential programs," *SIGARCH Comput. Archit. News*, vol. 34, no. 2, pp. 302–313, 2006.

[18] K. Kelsey, T. Bai, C. Ding, and C. Zhang, "Fast track: A software system for speculative program optimization," in *Code Generation and Optimization, 2009. CGO 2009. International Symposium on*. IEEE, 2009, pp. 157–168.

[19] M. Süßkraut, T. Knauth, S. Weigert, U. Schiffel, M. Meinhold, and C. Fetzer, "Prospect: A compiler framework for speculative parallelization," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 2010, pp. 131–140.

[20] S. Vrbsky and J. Liu, "Producing approximate answers to set- and single-valued queries," *Journal of Systems and Software*, vol. 27, no. 3, pp. 243–251, 1994.

[21] M. Rinard, S. Misailovic, A. Agarwal, M. Carbin, S. Sidiroglou, H. Hoffmann, and M. Rinard, "Power-Aware Computing with Dynamic Knobs," *MIT-CSAIL-TR-2010-027*, 2010.

[22] M. Rinard, A. Agarwal, M. Rinard, S. Sidiroglou, S. Misailovic, and H. Hoffmann, "Using Code Perforation to Improve Performance, Reduce Energy Consumption, and Respond to Failures," *MIT-CSAIL-TR-2009-042*, 2009.

[23] P. Prabhu, G. Ramalingam, and K. Vaswani, "Safe programmable speculative parallelism," in *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*. ACM, 2010, pp. 50–61.