Int J Parallel Prog (2011) 39:533–552
DOI 10.1007/s10766-010-0155-0

Value Prediction and Speculative Execution on GPU

Shaoshan Liu · Christine Eisenbeis · Jean-Luc Gaudiot

Received: 10 September 2010 / Accepted: 8 November 2010 / Published online: 1 December 2010

Abstract GPUs and CPUs have fundamentally different architectures. It is conventional wisdom that GPUs can accelerate only those applications that exhibit very high parallelism, especially vector parallelism such as image processing. In this paper, we explore the possibility of using GPUs for value prediction and speculative execution: we implement software value prediction techniques to accelerate programs with limited parallelism, and software speculation techniques to accelerate programs that contain runtime parallelism, which are hard to parallelize statically. Our experiment results show that due to the relatively high overhead, mapping software value prediction techniques on existing GPUs may not bring any immediate performance gain. On the other hand, although software speculation techniques introduce some overhead as well, mapping these techniques to existing GPUs can already bring some performance gain over CPU. Based on these observations, we explore the hardware implementation of speculative execution operations on GPU architectures to reduce the software performance overheads. The results indicate that the hardware extensions result in almost tenfold reduction of the control divergent sequential operations with only moderate hardware (5–8%) and power consumption (1–5%) overheads.

S. Liu (✉)
Microsoft, Redmond, WA, USA
e-mail: shaoliu@microsoft.com

C. Eisenbeis
Alchemy team, INRIA Saclay - Île-de-France & Univ Paris-Sud 11 (LRI, UMR CNRS 8623), Orsay 91405, France
e-mail: christine.eisenbeis@inria.fr

J.-L. Gaudiot
University of California, Irvine, CA, USA
e-mail: gaudiot@uci.edu

Keywords Value prediction · Speculative execution · GPU

1 Introduction

With the introduction of such processors as the IBM Cell Broadband Engine [1] and the NVIDIA Tesla [2], the computer industry has shifted to many-core designs. However, the problem of utilizing the enormous computing power delivered by tens or even hundreds of cores is yet to be solved. There are two major barriers to solving this problem: first, due to strong data dependencies, application programs may not contain enough parallelism to fully utilize the computing resources. Second, although some programs do contain high parallelism, the data dependencies exist in these programs may be irregular or complex, making the programs hard to parallelize.

Several techniques have been proposed to mitigate these barriers: to exploit more parallelism, value prediction techniques predict the live-in value to allow the consumer thread to continue instead of idling to wait for the producer thread to finish. On the other hand, to overcome the parallelization problem, Thread-Level Speculation (TLS) techniques are used to expose most parallelism by dynamically resolving data dependencies. While these techniques bring performance gains, the implementations of these techniques on Chip-Multi-Processors (CMP) suffer from several limitations. First, communication and synchronization between different cores incur a high overhead ($\sim$100 cycles); to amortize this overhead, the granularity of the thread need to be at least 100s of instructions. Second, if the thread granularity becomes too big, it requires excessive storage of the speculative states.

Unlike the CPU architecture, most of a GPU's on-chip area is dedicated to datapaths, thus minimizing the size of the caches and associated control logics [2]. It is conventional wisdom that GPUs can accelerate only those applications that exhibit very high parallelism, especially vector parallelism such as image processing. Nevertheless, the GPU architecture provides several advantages over the CPU architecture on speculative execution. First, the GPU architecture provides abundant closely coupled hardware threads, a large amount of registers for each thread, and fast synchronization and communication operations. These tightly coupled threads can be utilized for speculative execution with very low communication and performance overhead. Second, GPUs are much more power efficient than CPUs. In this paper we explore the possibility of using GPUs for speculative executions to accelerate programs that contain limited parallelism and those that contain irregular and dynamic parallelism, which are hard to parallelize. Specifically, the main contributions of this paper are as follows:

1. *Develop software methods of mapping speculative execution on GPU architectures*
2. *Identify the performance bottlenecks of software speculative execution on GPU architectures*
3. *Explore hardware solutions to speculative execution on GPU architectures*
4. *Evaluate the chip area and power consumption overheads introduced by value predictors on GPU architectures*

The rest of this paper is organized as follows: in Sect. 2, we review various value prediction and speculation techniques and explain why the GPU architecture is suitable for speculative executions. In Sect. 3, we present the programming framework that maps value prediction and speculation techniques on GPU architectures. In Sect. 4, we study the design tradeoffs in mapping software speculative execution on GPU architectures and propose hardware extensions to accelerate the speculative execution. In Sect. 5, we evaluate the software speculative execution techniques on a NVIDIA GeForce 8800 GPU and evaluate the hardware extensions on a Virtex-4 FPGA board. Finally, in Sect. 6, we conclude and discuss our future work.

2 Background

In this section, we review the background of value prediction and speculation techniques, as well as GPU architectures.

2.1 Value Prediction

The degree of Instruction-Level Parallelism (ILP) is directly related to the amount of data dependency in the program. To overcome this problem, several studies have proposed data value prediction techniques to speculatively break data dependency and improve and exploit ILP: Lipasti et al. [3] introduced the basic design of value predictors in the context of superscalar architectures. Sazeides et al. [4] demonstrated that different value prediction mechanisms can achieve prediction accuracies ranging from 56 to 92% and result in performance gains ranging from 7 to 23%. Sodani et al. [5] identified the key difference between Instruction Reuse and Value Prediction and their interactions with the microarchitecture.

Due to the emergence of many-core designs, Thread-Level Speculation (TLS) has gained much recent attention as several studies have proposed incorporating value prediction in TLS: Marcuello et al. [6] evaluated the potential for value prediction in a Clustered Speculative Multithreaded Architecture when speculating at the thread-level on the innermost loops. Oplinger et al. [7] evaluated the potential benefits to TLS in terms of memory value prediction, register value prediction, and procedure return value prediction. In addition, a recent study [8] indicated that modern application programs may not contain enough parallelism and value prediction techniques can significantly improve ILP and TLP.

2.2 Speculation

Applications with irregular or complex runtime data dependencies are hard to parallelize. For instance, in a simple loop that calculates $A[i] = A[B[i]]$, it is difficult to statically resolve the address $B[i]$, in part because these indirect memory accesses often depend on runtime inputs and behaviors. Thread-Level Speculation (TLS) techniques allow the automatic parallelization of code in the presence of statically ambiguous data dependences, thus dynamically extracting parallelism by resolving data dependencies at runtime. Under TLS, threads execute out of order and use spec-

ulative storage to record the necessary information to track inter-thread dependencies and to revert to a safe point and restart the computation upon the occurrence of a dependency violation [15].

Knight was the first to propose hardware support for TLS [9]; his work was in the context of functional languages. The Multiscalar architecture [10,11] was the first complete design and evaluation of an architecture for TLS; it breaks a program into a sequence of arbitrary tasks to be executed, and then allocates tasks in-order around a ring of processors with direct register-to-register interconnections. While the division of a program into tasks is done at compile time, all dynamic control of the threads is performed by ring management hardware at runtime.

One of the problems of existing TLS designs is that they are only able to improve performance when there is a substantial amount of medium-grained loop-level parallelism in the application. When the granularity of parallelism is too small or there is limited parallelism in the program, the software overhead overwhelms any potential performance benefits from speculative thread parallelism. Empirical results [12] have shown that communication and synchronization operations can take $\sim$100 cycles for most speculation operations. Thus, in order to amortize these overheads, the size of threads should be in the range of $\sim$1000 instructions.

2.3 The GPU Architecture

We feel that the GPU architecture is suitable for fine-grained speculative executions: first, the GPU architecture is tightly coupled such that it can simultaneously support hundreds of active threads. The excessive amount of hardware threads allows the implementation of aggressive speculative executions. For instance, to perform an *n-thread* speculation on CPU, it requires at least $n/2$ CPUs (with hyper-threading). On the other hand, a single GPU chip can simultaneously support a large amount of speculative threads, thus allowing the implementation of aggressive speculative execution. Second, the GPU architecture provides highly efficient threading supports: the synchronization operations only cost a few cycles. This facilitates the coordination and communication between speculative and non-speculative threads. Third, the GPU architecture provides a large amount of fast-access registers for each thread and shared memory for each thread block. The on-chip storage can be utilized to hold the speculative states. Fourth, GPUs are much more power efficient than CPUs. The Tesla C1060 GPU provides almost 1 TFlops/s of compute power while consuming only 190 W of power (0.19 W/GFlop) [2]. On the other hand, the Intel I7 Core CPU provides only 6.4 GFlops/s while consuming 340 W of power (53 W/GFlop) [18]. Thus, speculative execution would incur a much lower power overhead on GPUs compared to CPUs.

For instance, the GeForce 8800 GPU [13] contains 16 streaming multiprocessors (SM). each SM contains 8 streaming-processor (SP) cores. In order to efficiently execute hundreds of threads in parallel, the SM is hardware multithreaded such that it manages and executes up to 768 concurrent threads in hardware with very low scheduling overhead. However, one of the major performance bottlenecks for GPU is control divergence. The scheduling unit of each SM creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps. Individual threads can be

inactive due to independent branching or predication. If threads of a warp diverge via a data dependent conditional branch, the warp serially executes each branch path taken, and when all paths complete, the threads re-converge to the original execution path. In this paper, we construct the speculative execution framework using CUDA [14] and perform experiments on the NVIDIA GeForce 8800 GPU.

3 Speculative Execution on GPU

We are now ready to discuss our implementation of value prediction and speculation schemes on GPU.

3.1 Value Prediction

To perform value prediction on GPU, we first exploit the intrinsic parallelism by partitioning the program into independent thread blocks. For example, if each iteration calculates $A[k][i] = A[k][i-1] + 1$, then the outer loop, which is indexed by k, can be parallelized such that each thread block handles the calculation with a unique k index. Within each block, we can start one non-speculative thread (the master thread) and one or more speculative threads (the slave threads). The non-speculative and speculative threads can utilize the fast shared memory to store the value prediction table and the speculative execution results; also we can use barriers to synchronize these threads. Using the previous example, within each thread block, the non-speculative thread calculates the value indexed by i whereas the speculative thread calculates the value indexed by $i + 1$ using a predicted live-in value.

As shown in Fig. 1, computation with value prediction is carried out in three stages: the first stage is sequential; the master thread prepares the non-speculative live-in values. In the second stage, both the non-speculative and the speculative threads start execution simultaneously, and all threads synchronize at the barrier when they finish execution. The last stage is again sequential: the master thread verifies the correctness of the speculative execution, commits the results, and updates the value predictors.

Rollback can be easily handled by re-executing the speculative iteration with a correct live-in value. Assume that in the current round, the non-speculative thread is calculating iteration i, and the speculative thread is calculating iteration $i + 1$; if value prediction were successful, the loop counter is incre-

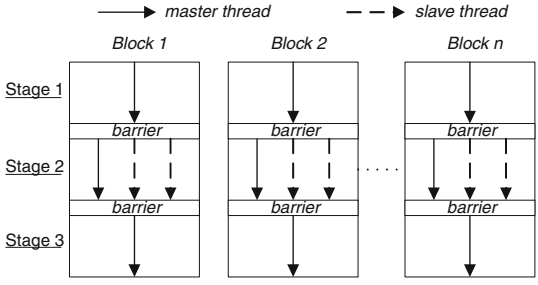

Fig. 1 Value prediction on GPU

mented by two, so that in the next round, the non-speculative thread calculates iteration $i + 2$ and the speculative thread calculates iteration$i + 3$. Otherwise, rollback occurs, so that the loop counter is only incremented by one, in the next round, the non-speculative thread re-calculates iteration $i + 1$ with the correct live-in value and the speculative thread calculates iteration $i + 2$. In this fine-grained value prediction approach, each speculative iteration only generates a limited number of speculative states which can be stored in the on-chip registers and fast shared memory. Thus when rollback occurs, the speculative storage is discarded without affecting the global memory space.

3.1.1 Data Dependencies

Various regular data dependency patterns exist in program loops. In Fig. 2, we categorize these data dependency patterns as simple data dependency, in which there is no communication between computation streams; and complex data dependency, in which thread blocks communicate with neighbor computation streams.

Simple Data Dependency: as shown in Fig. 2a, each column represents a computation stream. To map loops with simple data dependencies to GPUs, each computation stream is assigned to a thread block in an interleaving fashion, and there is no communication between thread blocks.

In the simplest case, each thread block contains one non-speculative thread and one speculative thread. The non-speculative thread calculates iteration i, and the speculative thread calculates iteration $i + 1$. The calculation of iteration $i + 1$ depends on the result of iteration i, thus the speculative thread uses a predicted live-in value in order to run in parallel with the non-speculative thread. The calculation results are temporarily stored in the fast shared memory to be visible to all threads in the same block.

When the threads finish the current computation, they reach a synchronization barrier. Then, the result of the non-speculative thread is used to verify the predicted live-in value of the speculative thread. If they match, the master thread commits the speculative and non-speculative results, as well as increment the iteration counter by two. Otherwise, rollback occurs, and the loop counter is only incremented by one.

Complex Data Dependency: for loops with complex data dependency (Fig. 2b), we also assign each computation stream to a thread block. However, the computation streams are no longer independent such that the calculation depends on the values produced by neighbor computation streams. In addition, these loops take multiple

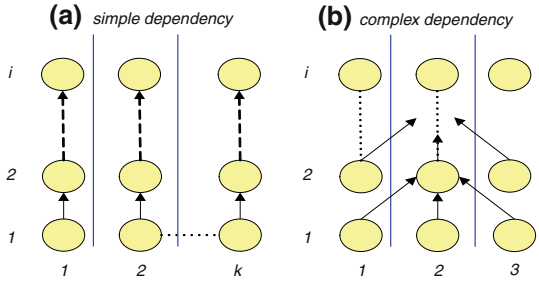

Fig. 2 Data dependency patterns

live-in values from the current and neighbor computation streams. Thus each computation stream needs to maintain multiple value predictors, one for each live-in value. This has two implications: first, the progress of different computation streams may be uneven, such that the leading thread may have to stall to wait for the lagging thread. Second, since the GPU architecture does not provide a fast shared memory between thread blocks, for synchronization of cross-computation streams, we have introduced a *global_tag* structure to record the data productions and consumptions of all streams.

In stage 1, the master thread consults the *global_tag* to compare the progresses of the current and the neighboring computation streams:

Case 1: the block contains a non-speculative and a speculative thread. The non-speculative thread fetches the live-in value from memory and the speculative thread fetches a predicted live-in value.

Case 2: both threads are speculative and both speculative threads obtain live-in values from the value predictors.

In stage two, both threads execute in parallel, and they reach a barrier upon completion. In stage three, the master thread first checks whether the live-in value for the speculative thread can be verified. This depends on the progress of the neighbor computation stream: the speculative execution can be verified only if the live-in values from the neighbor computation streams have been produced. If all live-in values have been produced, then we can compare the live-in values and the predicted live-in values. The verification is successful only if all live-in values are correctly predicted.

3.1.2 Aggressive Value Prediction Techniques

In this subsection, we discuss more aggressive value prediction techniques that utilize multiple speculative threads per block to improve performance.

N-Predictors There exist several basic value predictor designs: last value predictor (LVP), which predicts the current value the same as the previous one; stride value predictor (SVP), which predicts that the difference between the current value and the first previous value is the same as the difference between the first previous value and the second previous value; and finite context predictor (FCP), which maintains multiple contexts of past values, and when the current context matches one in the context table, we predict the next value the same as the next value maintained in that context.

In conventional designs, these predictors are combined to form a hybrid predictor, and runtime decision logic is required to select one component predictor to predict the next value. This introduces two problems: first, the decision logic introduces performance and power overheads. Second, the decision logic may not always select the correct predictor. In Sect. 5, we study the hardware and power overheads introduced by these predictor designs.

Since the GPU architecture provides excessive hardware threads [13], instead of selecting one predicted value to use, we can spawn multiple speculative threads, each associated with a value predictor. During the verification stage, as long as one speculative thread has used the correct predicted live-in value, then the speculative execution is successful. In this way, no decision logic is required. To implement this scheme,

each computation stream maintains N predictors, thus each thread block contains one non-speculative thread and N speculative threads.

K-Steps Another way to perform aggressive value prediction is to predict K steps ahead. In this case, each thread block contains a non-speculative thread, which computes iteration i, and K speculative threads, each uses a predicted live-in value to compute an iteration between $i + 1$ to $i + K$. In the verification stage, the result of iteration i is compared to the predicted live-in value of iteration $i + 1$; if they match, then the first speculative thread, which calculates iteration $i + 1$, is successful, and its result is compared to the predicted live-in value of iteration $i + 2$, and so on. In the ideal case, if all speculative threads are successful, we can increment the loop counter by $K + 1$, such that in the next round the non-speculative thread computes iteration $i + K + 1$.

3.2 Speculation

In programs such as finite-element computations, some loops contain data dependencies that are hard to parallelize statically: if each iteration calculates $A[i] = A[B[i]] + 1$, then we do not know whether there is a data dependency until we resolve the value of $B[i]$, which will be generated at runtime. To parallelize this type of loops, we can speculatively execute multiple iterations assuming no data dependency. Then, when a data dependency is detected, we re-execute those iterations with violations. Note that since all threads are speculative, the program is no longer divided into thread blocks; instead, it is divided into n speculative threads.

As shown in Fig. 3, the computation is carried out in two stages: in the first stage, the thread resolves the data dependency and determines whether there is a violation. If a violation is detected, then the thread enters the polling state until the data dependency has been satisfied. Since the target hardware provides no fast shared memory for threads across different blocks, to determine whether there is a dependency violation, we implement a *global_tag* data structure in global memory to keep track of data productions and consumptions. In the second stage, after the data dependency has been resolved, the thread fetches the live-in value from global memory and continues with execution. After execution completes, the thread commits the computation result to the global memory as well as updates the *global_tag*.

Dependency Checking In order to guarantee the correctness of the computation, we used a *global_tag* structure to keep track of all data productions and consumptions during execution. This is similar to the LPD test approach [15]. If each iteration com-

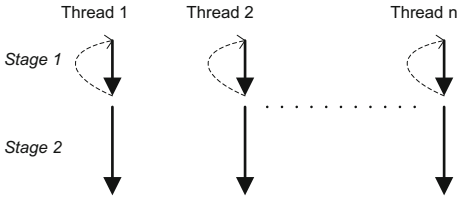

Fig. 3 Speculation on GPU

putes $A[i] = Op(A[B[i]])$, we assume no concurrent modification of $B[i]$ during the execution of the loop.

To determine whether a data item has been produced, all bits in the *global_tag* are set to zero when the program starts, and when a data, e.g., $A[i]$, has been produced, the corresponding bit, here the ith bit, is set to one. A true data dependency violation occurs if $B[i] < i$, and $A[B[i]]$ has not been produced ($global\_tag[B[i]]$ is not set). This means that the computation depends on a value that should have been produced but has not yet been produced.

An anti-dependency violation occurs if the thread is attempting to write a data that is supposed to be read but has not yet been read. If we are trying to produce $A[i]$, we need to know whether $A[i]$ is a live-in value to other computation, $A[j]$. In the indirection relationship defined in this loop, $i = B[j]$, in order to discover j, we have generated an inverted array B' such that $j = B'[i]$. Hence, the condition for an anti-dependency violation is if $i > B'[i]$ and $A[B'[i]]$ has not been produced. Here, $i > B'[i]$ indicates that the computation of $A[B'[i]]$ takes the original $A[i]$ as its live-in value; and $A[B'[i]]$ has not been produced indicates that if $A[i]$ is overwritten by the new value now, then the computation of $A[B'[i]]$ would be using an incorrect live-in value.

Optimization In the basic scheme, when a thread detects a dependency violation, it enters the polling state until the data dependency has been resolved. This may cause a high overhead if a thread is blocked for a long time. We have implemented an optimized scheme such that when a violation is detected, the current computation state is stored in a buffer and the thread moves on with the next round of computation. Assume that the current thread computes $A[i]$ and in stage 1, detects a dependency violation, then the index i is stored in a buffer and the thread moves on to the computation of $A[i + n]$. Then in the next round, the thread checks whether the buffer is empty and attempts to handle the buffered computations before moving on to its normal execution flow.

4 Design Space Exploration

We perform experiments to identify the design tradeoffs of the speculative execution schemes. As shown in Fig. 4, the program we use is the discrete transport application. It contains two loops: the iterations of the outer loop are independent whereas the inner loop contains data dependencies between the current and the previous iterations.

We parallelize the outer loop such that the program can be divided into n independent computation streams. In each stream, we compute two inner loop iterations at a time with one non-speculative and one speculative thread. Our test platform consists of an NVIDIA GeForce 8800 GTX GPU which contains 128 cores, each runs at 575 MHz; and the host CPU is an Intel Core 2 Duo E6300 running at 1.83 GHz.

Fig. 4 Discrete transport application

```
1:  for (l=1 ; l<=m ; l++ ) {
2:    for( k=0 ; k<n ; k++ ) {
3:      xx[l][k+1] = ( x[l][k] - xx[l][k] )* dn + xx[l][k];
4:    }
5:  }
```

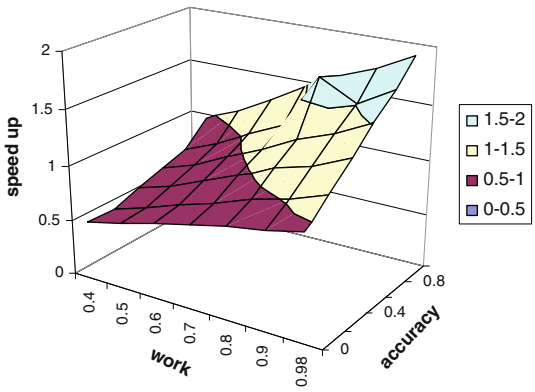

Fig. 5 Conditions for performance gain

We execute the application with three different schemes: the CPU scheme, in which we execute the program on the host CPU; the GPU scheme, in which we parallelize only the outer loop; the GPU with prediction scheme, in which we parallelize the outer loop into independent computation streams such that each stream contains a non-speculative and a speculative thread.

4.1 Conditions for Performance Gain

First, we identify the conditions under which the GPU with prediction scheme brings performance gain compared to the GPU scheme. To achieve this, we manipulate two parameters: the fraction of useful work in the value prediction framework, and the prediction accuracy. Recall that when we perform value prediction in three stages, stages 1 and 3 introduce pure overheads whereas stage 2 performs useful computation work in parallel. Thus, the fraction of useful work represents the workload of stage 2 relative to the overall program workload. To control this parameter, for testing purposes, we artificially inject work into the program loop body to increase the relative size of stage 2. The second parameter, prediction accuracy, is the fraction of live-in values that are correctly predicted. Recall that in each round of computation a block contains a non-speculative thread and a speculative thread, the speculative thread uses a predicted live-in value. If the prediction is correct, then the speculative execution is successful. We control this parameter by artificially injecting correct live-in values.

The result is shown in Fig. 5: the x-axis shows the fraction of useful work, the y-axis shows the prediction accuracy, and the z-axis shows the speedup of the GPU with prediction scheme over the GPU scheme. When the fraction of useful work is 90%, a speedup of 1 is reached when the prediction accuracy is 20%; when the prediction accuracy reaches 80%, the speedup becomes 1.5. Since in this case we are only looking one step ahead, when the fraction of useful work is 98% and the prediction accuracy is 100%, the speedup can reach 1.9. Hence, in order to achieve high performance, we need to design value predictors with high prediction accuracy; meanwhile, the predictors should cause as little overhead as possible.

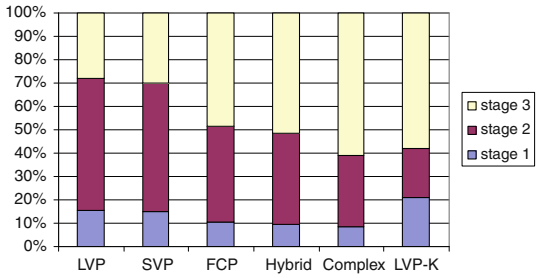

Fig. 6 Overheads of value prediction schemes

4.2 Overheads of the Software Approach

Different value prediction schemes vary in their complexity, as well as the performance overheads they introduce. We implement six software value prediction schemes to execute the discrete transport application. These schemes include the component predictors including last value predictor (*LVP*), stride value predictor (*SVP*), finite context predictor (*FCP*), and the more sophisticated predictors including *complex, hybrid*, and *LVP-k*. The component predictors only utilize one speculative thread in each block. *Complex* combines the three component predictors, but it supports only one speculative thread thus it requires some decision logic to select a predictor to use. *Hybrid* supports three speculative threads, each associated with a component predictor. *LVP-k* supports k speculative threads to execute k steps ahead.

The results are shown in Fig. 6: in the *LVP* and *SVP* schemes, the overhead introduced by stages 1 and 3 is around 40%. The *FCP* scheme is more complex than *LVP* and *SVP* since it requires a context table, thus the overhead can reach 60%. The *complex* scheme combines the three component predictors and one decision module; the overhead can reach 70%. The *hybrid* scheme is similar to the *complex* scheme but it does not have a decision module, the overhead is 62%. The *LVP-k* scheme involves long verification and predictor update, thus the overhead can reach 78%.

4.3 Performance Comparisons

To understand how GPU can be used to accelerate computations, we execute the application on a CPU, on a GPU without value prediction, and with value prediction. In this experiment, value prediction is ideal such that all predictions are correct and the fraction of useful work is 90%. The results are shown in Fig. 7: the x-axis shows the number of computation streams and the y-axis shows the execution time in microseconds. The line with triangle blocks shows the CPU performance. The line with square blocks shows the performance on GPU without value prediction, and its performance matches the CPU performance when the number of computation streams reaches 130. The line with tilted square blocks shows the performance on GPU with value prediction: its performance is almost double of that of GPU without prediction due to high prediction accuracy and high fraction of useful work.

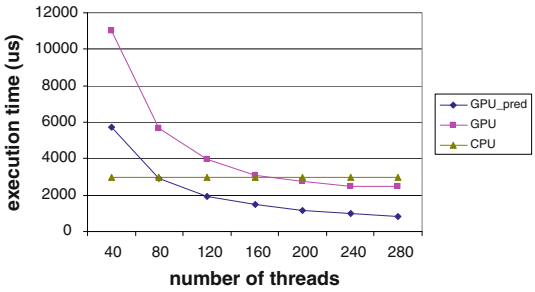

Fig. 7 Performance comparisons

4.4 Analytical Model

We summarize the empirical findings from the previous two subsections in Eqs. 1 and 2, which describe an analytical model capturing the relationship between the performance of speculative execution and the key design parameters. $Speedup_{GPU}$ represents the performance improvement of GPU with speculative execution over that of GPU without speculative execution; oh represents the overhead incurred by the particular speculative execution scheme; p represents the prediction or speculation accuracy; and k represents the look-forward steps used in our execution: for example, if we have two speculative threads, one calculates $i+1$ and the other $i+2$, then $k=2$.

$$Speedup_{GPU} = \frac{1}{(1+oh)(1-p)+\frac{(1+oh)p}{k+1}} \tag{1}$$

$$oh = oh_{mem} + oh_{seq} + oh_{comp} \tag{2}$$

Equation 1 explains the behavior of the proposed value prediction and speculation techniques. Simple value predictors such as *LVP* and *SVP* incur a fairly low overhead, oh, but the prediction accuracy, p, is also low. Techniques such as *FCP*, *Hybrid*, and*Complex* may be used to improve p but they incur higher oh. Similarly, *LVP-k* aims to improve k but it also results in a high oh due to a long verification stage. Equation 2 breaks down the overhead incurred by the proposed speculative schemes. Note that these overheads are not orthogonal and they may overlap with each other.

1. Memory access overhead oh_{mem}: the predictors need to consult and update the prediction table.
2. Sequentialization overhead oh_{seq}: control divergence causes the execution to be serialized.
3. Computation overhead oh_{comp}: the value predictors require extra computations to generate the predicted values.

The analytical model indicates that the key to generate performance gain is to effectively reduce or mask the overhead. The current target hardware is able to hide the memory access latency effectively: when a long latency memory operation is issued, it is capable to switch to a new *warp* of threads with very low overhead. However, the main bottleneck is the sequentialization (control divergence) problem.

Fig. 8 Sequential execution of stage 3

```
1:    ld.param.u32      $r75, [__cudaparm_xx];
2:    ld.shared.s32     $r76, [__cuda_spec8+0];
3:    mul.lo.u32  $r77, $r18, 4;
4:    add.u32           $r78, $r75, $r77;
5:    st.global.s32     [$r78+0], $r76;
6:    ld.shared.s32     $r81, [step];
7:    add.s32           $r82, $r81, 1;
8:    st.shared.s32     [step], $r82;
9:    ld.shared.s32     $r79, [hit];
10:   mul.lo.u32  $r83, $r79, 4;
11:   add.u32           $r84, $r83, $r20;
12:   ld.shared.s32     $r85, [$r84+0];
13:   add.s32           $r86, $r18, $r16;
14:   mul.lo.u32  $r87, $r86, 4;
15:   add.u32           $r88, $r75, $r87;
16:   st.global.s32     [$r88+0], $r85;
17:   add.s32           $r89, $r85, $r85;
18:   ld.shared.s32     $r76, [__cuda_spec8+0];
19:   sub.s32           $r90, $r89, $r76;
20:   st.shared.s32     [__cuda_pred16+4], $r90;
```

4.5 Hardware Acceleration

One effective way to reduce the sequential section is to move some of the control divergent operations into hardware. In stages 1 (prediction stage) and 3 (verification stage), most of the instructions are additions, comparisons, and table updates. Specifically, one effective way of acceleration is instruction collapsing, in which we collapse a sequence of n instructions (usually the whole basic block) into a single specialized instruction. As a result, instead of fetching and executing n instructions, we only have to fetch and execute one instruction. In the ideal case, it results in a speedup of n; or in other words, it reduces the sequential section by n-fold.

For instance, Fig. 8 shows the assembly code of the verification stage of our speculative execution framework. Lines 1–5 commit the non-speculative result to memory; lines 9–16 commit the speculative result to memory; and lines 17–20 update the SVP prediction table. If we could extend the ISA to include two special instructions *commit* and *table_update*, then we could significantly reduce the size of the sequential section. We explore the impact of this technique in Sect. 5.

5 Experiments and Results

First, we perform a number of experiments on the GPU test platform described in Sect. 4 to evaluate the software speculative execution schemes. Since most benchmark programs are written for CPUs, thus it is very difficult to port the whole programs onto GPUs. Instead, we extract kernel loops from the SPEC CPU 2006 [16] and the PARSEC [17] benchmark suites and ported these kernel loops on GPU. Second, we implement a GPU-like architecture on Virtex-4 FPGA to evaluate the performance, hardware overheads, and power consumption of the proposed scheme.

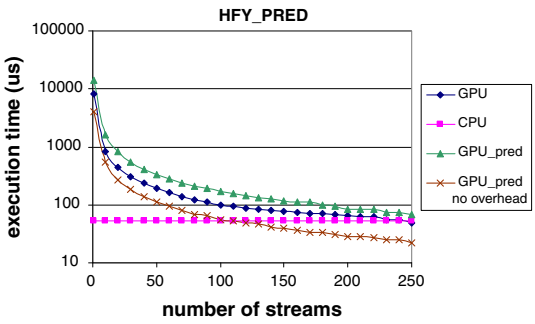

Fig. 9 HFY_PRED performance

5.1 Software Value Prediction on GPU

For software value prediction, we use stride value predictors and focus on the non-aggressive techniques since the aggressive techniques introduce overwhelming overhead, which in the previous section have been identified as the main performance bottleneck.

HFY_PRED: this loop is taken from *swaptions* of PARSEC, in the *HJM_Forward_to_Yield* routine. It is a financial application that computes yield rates from the forward rates supplied. This loop contains simple data dependencies such as discussed in Sect. 3.1.1. Thus, we map each outer loop iteration to a block; and one non-speculative iteration and one speculative iteration of the inner loop to the threads within a block.

Figure 9 shows the experiment results: the x-axis shows the number of independent execution streams. Note that with value prediction, each stream contains one non-speculative and one speculative thread. The y-axis shows the execution time. We show the CPU performance (the line with square blocks) for reference. In the baseline GPU approach (the line with tilted square blocks), each independent stream is assigned to one thread. In the GPU with value prediction approach (the line with triangle blocks), the performance is always higher than the baseline GPU approach due to the long sequential section in the program. To understand the overhead incurred by the sequential section, we also record the execution of only the loop body (the line with cross blocks), which shows that the overhead of the sequential section reaches 60% in this case. On the other hand, it implies that if we only consider the parallel loop body and compare to the GPU baseline, a 30% improvements can be achieved. Note that in the current GPU architecture, there is no hardware support for value prediction, thus all operations are done in software, resulting in a high overhead. Nevertheless, this result implies that if we could add value prediction hardware supports in the GPU architecture, the sequential overhead could be minimized and this technique can indeed introduces a high performance gain.

Transform_PRED: this loop is in *fp_tree* from *freqmine* of PARSEC. It transforms a tree structure into an array structure. This loop contains simple data dependencies such as discussed in Sect. 3.1.1. The mapping of this loop to GPU is similar to the previous case. Figure 10 shows the experiment results: the *GPU_pred* scheme introduces a very high overhead compared to the baseline GPU scheme. This

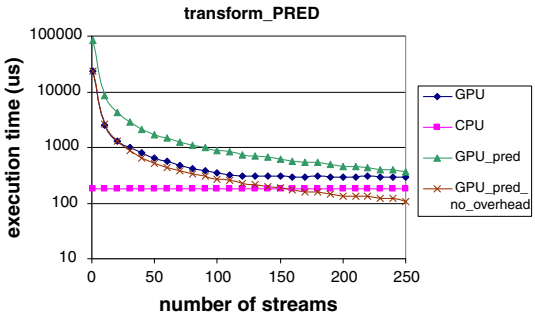

Fig. 10 Transform_PRED performance

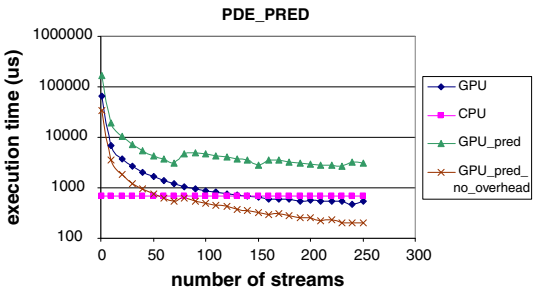

Fig. 11 PDE_PRED performance

is because this loop contains a very low degree of data redundancy, resulting in low prediction accuracy, and thus an overwhelming overhead. This is confirmed after we strip the sequential section overhead in *GPU_pred_no_overhead*: even without overhead, the GPU with prediction case has similar performance as the GPU baseline case.

PDE_PRED: we code and port to GPU this wavefront PDE solver. This loop contains complex data dependencies discussed in Sect. 3.1.1: it requires input values from the left and right neighbor streams, thus cross-block communication and synchronization is required. Figure 11 shows the experiment results: the GPU with prediction scheme (the line with triangle blocks) shows an irregular behavior. This is due to the communication between computation streams. As illustrated in Fig. 3, during execution, the progresses of the different execution streams cannot be controlled, thus resulting in fluctuations of the performance behavior. Again, the pure computation time of the GPU with prediction scheme is improved by about 30% when compared to the baseline GPU scheme.

5.2 Software Speculation on GPU

For software speculation, we parallelize the loops assuming no data dependencies. When data dependencies are detected, we re-execute those loops with dependency violations using the correct live-in values.

Cont_SPEC: this is the *concenter* routine taken from *streamcluster* of PARSEC. It computes the means for the k clusters as well as the relative weight of points in the

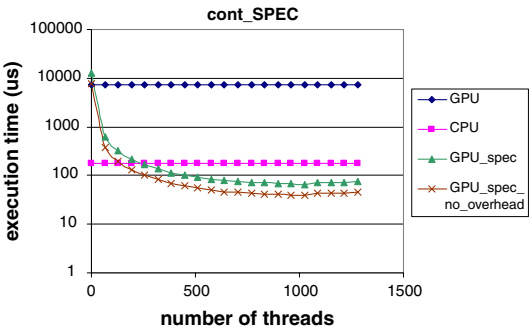

Fig. 12 Cont_SPEC performance

clusters. This loop is hard to parallelize because the data production in the loop body goes through a level of indirection such that data consumption can not be performed until the source data address has been resolved. We implement this loop with the speculation framework discussed in Sect. 3.2.

The experiment results are shown in Fig. 12: the x-axis shows the number of threads used in the program whereas the y-axis shows the execution time. Because the loop is hard to parallelize, we also show the single thread performance of GPU (line with tilted square blocks) and CPU (line with square blocks) for reference. The speculative execution on GPU (line with triangle blocks) scales well with the increasing number of threads. This is due to the limited amount of data dependencies, allowing this technique to expose a large amount of parallelism. We are also interested in the overhead introduced by this scheme. Thus, we capture the execution time of only the loop body (line with cross blocks) as well. This result shows that the speculation overhead is about 40%.

Decompress_SPEC: this is a kernel loop in the *decompress* routine inside *bzip2* of SPEC CPU 2006. It traverses an array and un-compresses the data. During data un-compression, it traverses an array that contains the addresses of the source data. Then, it fetches the data through pointer indirection, performs some mathematical operations on the piece of data, and then stores it to a target array. Again, since data dependency depends on runtime behavior, this program is hard to parallelize. Figure 13 shows the experiment results: compared to *cont_SPEC*, this loop is not as scalable due to the existence of strong data dependencies in the loop body. In this case, many speculative threads would fail due to dependency violations. Under this situation, these threads would enter a polling state until the dependency violations have been resolved. Note that when the number of threads exceeds 1000, the execution time starts to increase due to an increasing thread scheduling overhead. Also, in this case, the overhead introduced by this scheme is about 40%.

Rebuild_SPEC: this is a kernel loop in the *rebuild* routine inside *fluidanimate* of PARSEC. It rebuilds the grid to simulate the incompressible fluid for interactive animation purposes. This program is different from the previous two programs in that its data production, instead of data consumption, goes through a level of indirection. Specifically, it fetches a piece of data from a source array, performs some operations, then stores the result to the location which's address is stored in the target array. Sim-

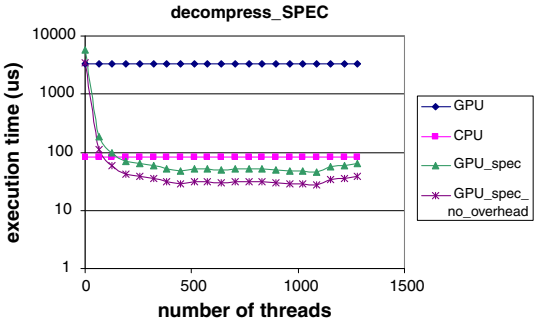

Fig. 13 Decompress_SPEC performance

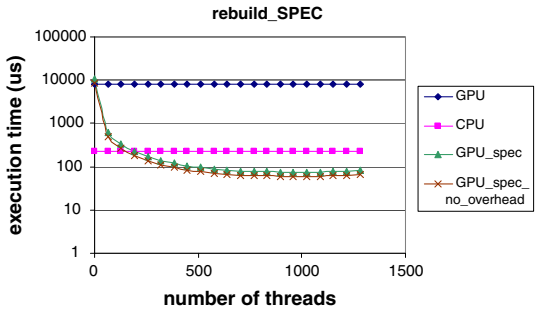

Fig. 14 Rebuild_SPEC performance

ilar to the previous two cases, the indirection in data production makes this program hard to parallelize. In addition, at each iteration, it involves the calculation of nine grid values and thus contains significantly more work than the previous cases. As shown in Fig. 14, in this program the speculation operation overhead is only about 10% compared to 40% in the previous cases. This is because the large loop body amortizes the overhead of speculation.

In Sect. 3.2, we have proposed an optimization scheme such that when a dependency violation is detected, it stores the current iteration information in a buffer and moves on with the next iteration. Our evaluation shows that this technique is capable of improving performance by 3–5%. Performance gain comes from the hiding of data dependencies. However, the performance gain brought by this technique is fairly low because it is offset by the constant overhead: no matter whether data dependencies are present or not, this scheme always checks buffer for pending computations.

5.3 Exploration of Hardware Implementation

In this subsection we explore the hardware implementation of speculative execution operations in order to identify the hardware and power consumption overheads. To achieve this, we implement a GPU-like architecture onto the ML401 FPGA board [19]. Our GPU-like architecture emulates a streaming multiprocessor (SM) as in the NVIDIA GeForce 8800 GPU (please refer to Sect. 2.3). Our implementation consists of a block RAM to store instructions. In the main datapath, it contains eight ALU

Table 1 Hardware and power overheads of different value predictors

	# FF	# LUTs	Power (mW)	Performance (cycles)
LVP	40	40	3.64	2
SVP	90	70	7.27	2
FCP	190	70	14.55	2
Hybrid	400	220	32.73	4
Overhead (%)	20.00	18.18	22.22	50.00

engines to perform parallel computation of basic arithmetic operations (add, subtract, multiply, divide, and shift). Also, each ALU engine contains 16 32-bit registers. During execution, a common instruction fetch engine fetches an instruction from the block RAM and distributes this instruction to all ALUs, which then execute in parallel. In order to measure the chip power consumption, we place a 0.033 Ohm shunt resistor on the power supply rail and measure the voltage drop across the shunt resistor. Then from the voltage drop we derive the current that goes into the Virtex-4 FPGA chip, and then we multiply this current by the 1.2 V supply voltage to calculate the chip power consumption under different configurations.

In the first step, we evaluate the hardware and power overhead of standalone hardware value predictors. As summarized in Table 1, we implement standalone LVP, SVP, FCP, and Hybrid predictors on the FPGA board and measure the usage of hardware resources including the flip-flops (FF) and the look-up tables (LUT), also we measure the chip power consumption when the predictors are active. Note that the hybrid predictor design consists of a LVP, a SVP, a FCP, and a decision circuit. The last row of Table 1 shows the overheads introduced by the decision circuit, which contribute to 20% of the total hardware cost, 22% of the power consumption, and 50% of the total latency. The high overheads introduced by the decision circuit imply that it may be beneficial to perform aggressive value prediction on GPUs, as proposed in Sect. 3.1.2.

Then we implement the GPU-like architecture on the FPGA board and measure the hardware and power consumption overheads. We consider several cases: *baseline* represents existing GPU design such that no hardware value predictor is implemented; *baseline+LVP, baseline+SVP,* and *baseline+FCP* represent the implementation of one value predictor into the *baseline*; and *baseline+3 VP* represent the implementation of the three value predictors into the *baseline*, which allows the execution of

Table 2 Incorporation of hardware value predictors onto GPU architectures

	# FF	# LUTs	Hardware overhead (%)	Power (mW)	Power overhead (%)
Baseline	17055	26106	0.00	356.36	0.00
Baseline+LVP	17153	28226	5.48	360.00	1.02
Baseline+SVP	17185	28355	5.77	363.64	2.04
Baseline+FCP	17283	28396	6.19	367.27	3.06
Baseline+3 VP	17511	28762	7.75	374.55	5.10

aggressive value prediction. These designs introduce moderate hardware (5–8%) and power consumption (1–5%) overheads compared to the *baseline* (Table 2).

Each hardware value predictor keeps a value history table. These value predictors run in parallel with the datapath components. When these predictors are triggered, they generate the predicted values from the value history table; and when the actual values are generated, the value history tables are updated automatically. In the software value prediction implementation, stages 1 and 3 respectively consist of more than ten and twenty instructions, and they are sequential. As shown previously, this long sequential section is the main performance bottleneck. In our prototype implementation, these sequential instructions are collapsed into two instructions *predict*, with a latency of two cycles, and *verify*, with a latency of two cycles. Specifically, *predict* triggers the predictors to generate a prediction; and *verify* compares the actual value and the predicted value, and updates the value history table. This simple hardware implementation results in almost tenfold reduction of the sequential section.

6 Conclusions

In this paper we have studied the possibility of using GPUs for speculative execution. The key idea is that with speculative execution, we can accelerate applications with limited parallelism and those that are hard-to-parallelize. We evaluated the software speculative execution schemes on a NVIDIA GeForce 8800 GPU and performed a space exploration study to identify the performance bottlenecks. In addition, we proposed hardware solutions to minimize the performance bottlenecks and explored these hardware solutions on a ML401 FPGA board.

The results of this study have demonstrated the potential of speculative execution on GPU architectures: in the software value prediction case, although it introduces a fairly high overhead, we observed a great improvement of the pure computation time (without overhead) of the speculative scheme over the non-speculative scheme. In the software speculation case, we managed to parallelize loops with complex and dynamic data dependencies, and the performance of such scheme already surpasses that of CPU. On the other hand, we also identified several problems in using existing GPU architectures to execute software value prediction and speculation operations.

1. *Sequential execution overhead*: control divergence of the target hardware results in a large sequential section, which is a major performance bottleneck.
2. *Lack of cross-block synchronization*: without cross-block synchronization support, it is very hard to manage the speculative execution of programs with cross-block communication.
3. *Memory access overhead*: many synchronization and communication operations have to go through the global memory, which incurs a high latency.

In addition, our exploration on hardware solutions showed that the hardware extensions (hardware value predictors) resulted in almost tenfold reduction of the control divergent sequential operations with only moderate hardware (5–8%) and power consumption (1–5%) overheads. Thus this result confirmed that it may be beneficial to add hardware support to existing GPU architectures to support speculative execution.

This exploratory work naturally leads to two directions of our future work: first, we will perform a more detailed design and optimization of specialized instructions in order to enhance speculative execution performance and to reduce power consumption. Second, we will design and implement more sophisticated and efficient dependency checking techniques to guarantee the correctness of speculative execution. The current dependency checking scheme requires a *global_tag* data structure to keep track of data dependencies thus resulting in a very high overhead due to excessive memory access. We plan to perform optimization on the current technique as well as propose new methods for dependency checking.

Acknowledgments This work is partly supported by the National Science Foundation under Grant No. CCF-0541403 and by the French Agence Nationale pour la Recherche (ANR) PetaQCD project. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or of the ANR.

References

1. IBM Cell Broadband Engine, http://www.ibm.com/developerworks/power/library/pa-cellperf/
2. NVIDIA Tesla Computing Solutions, http://www.nvidia.com/object/tesla_computing_solutions.html
3. Lipasti, M.H., Shen,J.P.: Exceeding the dataflow limit via value prediction. In: Proceedings of the 29th International Symposium on Microarchitecture, December 1996
4. Sazeides, Y., Smith, J.E.: The predictability of data values. In: Proceedings of the 30th Annual International Symposium on Microarchitecture, December 1997
5. Sodani, A., Sohi, G.S.: Understanding the differences between value prediction and instruction reuse. In: Proceedings of the 31st Annual International Symposium on Microarchitecture, December 1998
6. Marcuello, P., Tubella, J., González, A.: Value prediction for speculative multithreaded architectures. In: Proceedings of the 32nd Annual international Symposium on Microarchitecture (Micro'99), November 1999
7. Oplinger, J., Heine, D., Lam, M.S.: In search of speculative thread-level parallelism. In: Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT'99), October 1999
8. Liu, S., Gaudiot, J-L.: Potential impact of value prediction on communication in many-core architectures. IEEE Trans. Comput. **58**, 6 (2009)
9. Knight, T.: An architecture for mostly functional languages. In: Proceedings of the ACM Lisp and Functional Programming Conference, August, 1986
10. Franklin, M., Sohi, G.S.: APB: a hardware mechanism for dynamic reordering of memory references. IEEE Trans. Comput. **45**, 5 (1996)
11. Sohi, G.S., Breach, S., Vijaykumar, T.N.: Multiscalar Processors. In: Proceedings of the 22nd International Symposium on Computer Architecture (ISCA'95), June, 1995
12. Hammond, L., Hubbert, B.A., Siu, M., Prabhu, M.K., Chen, M., Olukotun, K.: The stanford hydra CMP. IEEE Micro **22**, 2 (2000)
13. NVIDIA GeForce 8800, http://www.nvidia.com/page/geforce_8800.html
14. CUDA Zone—the resource for CUDA developers, http://www.nvidia.com/object/cuda_home.html#
15. Rauchwerger, L., Padua, D.: The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. ACM SIGPLAN Notices **30**, 6 (1995)
16. SPEC CPU2006, http://www.spec.org/cpu2006/
17. Bienia, C., Kumar, S., Singh, J.P., Li, K.; The PARSEC benchmark suite: characterization and architectural implications, Princeton University Technical Report TR-811-08, January 2008
18. Intel Core i7 Processor, http://www.intel.com/products/processor/corei7/index.htm
19. Xilinx ML401 Overview, http://www.xilinx.com/products/boards/ml401/index.htm