Speculative Execution on Multi-GPU Systems

Gregory Diamos and Sudhakar Yalamanchili
School of Electrical and
Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332–0250
gregory.diamos@gatech.edu sudha@ece.gatech.edu

***Abstract*—The lag of parallel programming models and languages behind the advance of heterogeneous many-core processors has left a gap between the computational capability of modern systems and the ability of applications to exploit them. Emerging programming models, such as CUDA and OpenCL, force developers to explicitly partition applications into components (kernels) and assign them to accelerators in order to utilize them effectively. An accelerator is a processor with a different ISA and micro-architecture than the main CPU. These static partitioning schemes are effective when targeting a system with only a single accelerator. However, they are not robust to changes in the number of accelerators or the performance characteristics of future generations of accelerators.**

In previous work, we presented the Harmony execution model for computing on heterogeneous systems with several CPUs and accelerators. In this paper, we extend Harmony to target systems with multiple accelerators using control speculation to expose parallelism. We refer to this technique as Kernel Level Speculation (KLS). We argue that dynamic parallelization techniques such as KLS are sufficient to scale applications across several accelerators based on the intuition that there will be fewer distinct accelerators than cores within each accelerator. In this paper, we use a complete prototype of the Harmony runtime that we developed to explore the design decisions and trade-offs in the implementation of KLS. We show that KLS improves parallelism to a sufficient degree while retaining a sequential programming model. We accomplish this by demonstrating good scaling of KLS on a highly heterogeneous system with three distinct accelerator types and ten processors.

I. Introduction

The pursuit of performance through parallelism and efficiency through specialization is gradually shifting the focus of the computing industry from general purpose single-core to heterogeneous many-core processors. Traditionally, accelerators have been used to improve power efficiency and performance of domain specific applications without the overheads required to support generic computation. As rising power and thermal constraints of future technology nodes limit the use of highly complex ILP centric architectures and PVT variations [1] reduce worst case margins, domain-specific accelerators traditionally used for graphics [2], media processing [3], and security [4] have become increasingly attractive. The problem is the additional complexity introduced by these accelerators – complexity that is typically exposed directly to the programmer.

Several research and industry efforts have identified the complexity of writing scalable high performance applications as a major challenge to the proliferation of many-core systems, and have focused on reducing this complexity through programming model abstractions that explicitly address modularity, data sharing, and encapsulation of code running on homogeneous cores combined with runtime execution models that map these abstractions onto diverse hardware resources.

The origins of these models can be traced to research efforts into asynchronous function calls in *Cilk* [5], runtime management of load balancing, synchronization, and communication latency in *Charm++* [6], and streaming data parallel operations in *Brook* [7] and *StreamIt* [8]. These initial explorations have been solidified in industrial implementations such as Brook++ for ATI GPUs [9], CUDA for NVIDIA GPUs [10], and OpenCL [11] for generic architectures where concepts like asynchronous encapsulated kernel calls, explicit communication channels, and runtime resource mapping reflect their foundations in earlier efforts.

However, as the original efforts typically identified parallel programming as the primary source of complexity, the problem was again revisited to address architecture heterogeneity in efforts such as *Merge* [12], *Harmony* [13], and *Qilin* [14] as well as memory hierarchy heterogeneity in *Sequoia* [15]. In the context of these new efforts, applications are typically expressed as a set of encapsulated function calls whose execution is constrained by explicit data flow and control flow. (Hereafter, we refer to encapsulated function calls as kernels). The goal of these models is to exploit coarse-grained kernel-level-parallelism (KLP) to partition work among multiple accelerators in a system and fine-grained data-parallelism to partition work across cores within an accelerator.

Fine-grained parallelism within an accelerator is typically handled via partitioning into data-parallel threads as in CUDA and OpenCL or into streams as in StreamIt or Brook. These partitioning schemes implicitly rely on the fact that cores within an accelerator are homogeneous to simplify the programming model. Additionally, they must deal with a large amount of parellelism in hardware. For example, NVIDIA's GT200 GPUs support 23040 threads in hardware [10]. This extreme degree of parallelism benefits from the use of explicitly parallel programming models to fully utilize all of the resources in a given accelerator.

Unfortunately, most of these programming models apply the same explicitly parallel programming models to target multi-accelerator systems. CUDA and OpenCL for instance, require

U.S. Government work not protected by U.S. copyright

```
3 int* p_hmaxs = h_maxs; // initial input data
4 float* p_hvars = h_vars; // initial input data
5
6 // Launch the device computation threads
7 for (i = 0; i<t-block_num; i+=block_num)
8 {
9   // Launches a kernel
10  PetrinetKernel<<< grid, threads>>>(g_places, g_vars, g_maxs,
11    N, s, 5489*(i+1));
12  // Copies results from the kernel to the host
13  CopyFromDeviceMemory(p_hmaxs, g_maxs, block_num*sizeof(int));
14  CopyFromDeviceMemory(p_hvars, g_vars, block_num*sizeof(float));
15
16  // Computes the next dataset
17  p_hmaxs += block_num;
18  p_hvars += block_num;
19 }
```

Fig. 1. CUDA Source Code For The Inner Loop of PNS

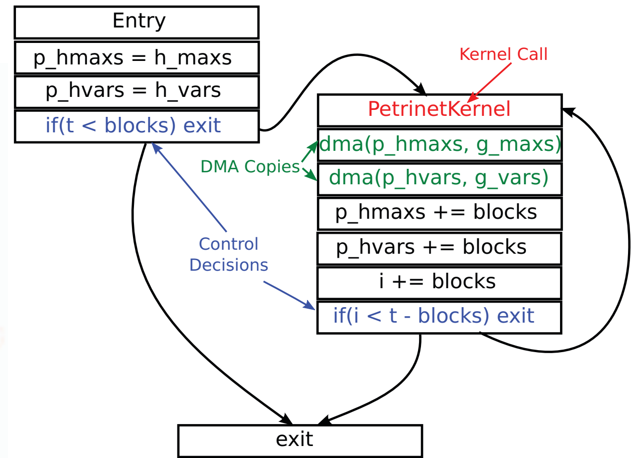

Fig. 2. PNS Control Flow Graph

the programmer to determine the number and type of accelerators in the system and statically assign work to them. These represent explicitly parallel programming models and static partitioning schemes; they force the developer to deal directly with parallelism and heterogeneity. In this paper, we offer an alternative approach. We use speculation to expose parallelism within an application and then dynamically map kernels to accelerators with potentially heterogeneous architectures.

This paper explores the use of a technique traditionally used to extract parallelism from sequential applications, thread level speculation, to extract parallelism from applications to target multi-accelerator systems. We use the term Kernel Level Speculation (KLS) for our approach to indicate that the basic unit of work that we launch speculatively is a kernel rather than a thread. This paper does not introduce a new form of speculation that is significantly different from those proposed in the past [16]–[18]. Instead we apply optimizations at the kernel level and seek to answer the question of whether or not speculation can extract enough parallelism from workloads for heterogeneous systems. We expect this work to be supplemental to fine-grained parallel programming models for individual accelerators such as OpenCL, CUDA, and OpenMP, which can be also be augmented to support KLS in future work.

This paper makes the following contributions:

- We show how existing programming models for heterogeneous systems such as Harmony, CUDA, and OpenCL can be automatically parallelized by using speculation to exploit multi-accelerator systems.
- We develop a complete prototype of a runtime for the Harmony execution model described in [13], and then extend it to support speculation.
- We derive a metric to quantify the amount of Kernel Level Parallelism (KLP) within a given application. We calculate the upper limit on KLP assuming that all dependencies can be removed via perfect speculation and compare this against the speedups achieved by our implementation.

Organization. This paper is organized as follows: Section II gives background on execution models for programming heterogeneous systems; Section III presents the extensions required to support speculation; Section IV explores the amount of kernel level parallelism in Harmony and CUDA applications; Section V describes experimental results; and Section VI covers related work.

II. Execution Model Overview

OpenCL and CUDA define an execution model where code segments are encapsulated in compute kernels which are launched asynchronously and guaranteed to be side-effect-free. Kernels can be compiled for several possible processor architectures or an intermediate code representation[1] that can be dynamically recompiled to target different accelerators. Though neither CUDA nor OpenCL currently exploit this property, the side-effect-free constraint allows kernels without data dependencies to be launched in parallel on different accelerators. Harmony [13] extends these models by annotating each kernel with explicit input and output parameters to simplify dependence analysis and actively launches kernels without data dependencies in parallel, relieving the programmer from performing the same task manually.

A. The Kernel Control Flow Graph

These execution models are purely imperative at the inter-accelerator level: the order of execution is constrained by the programmer defined sequence of kernels and control decisions. In this paper, we use the term control decisions to refer to classical if-then-else blocks, loops, etc that typically map to branch instructions. Figure 1 contains an example of a for loop structure in CUDA on line 7. The complete figure shows the inner loop of the PNS application from the UIUC Parboil benchmark suite [20]. We will continue to refer to this example application in this section.

Recall that in programming models such as CUDA and OpenCL, kernels are guaranteed to be side-effect-free. In our

[1]When implemented on NVIDIA hardware, both standards use NVIDIA's PTX [19] virtual ISA.

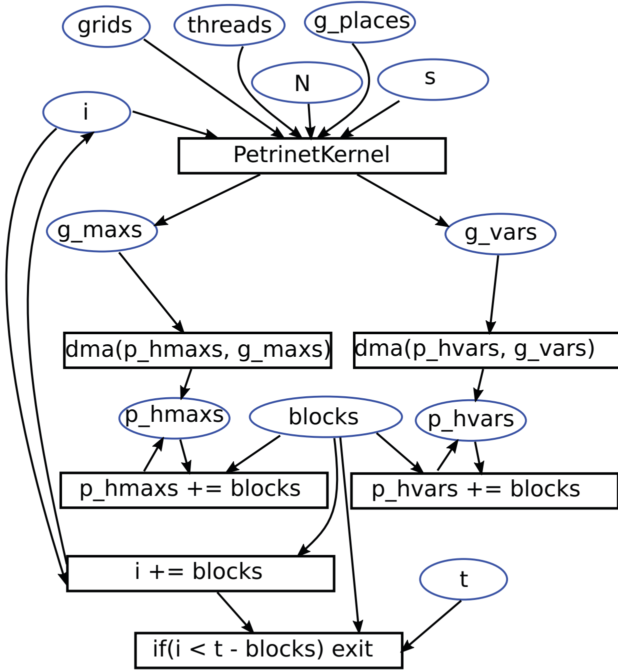

Fig. 3. Data Flow Graph for PNS Inner Loop

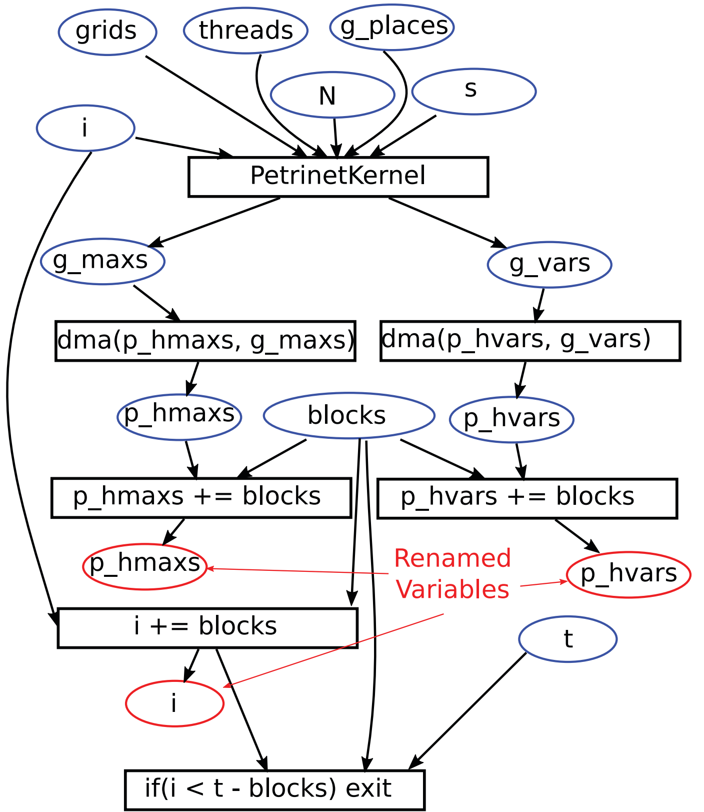

Fig. 4. PNS Inner Loop Data Flow Graph After Renaming

previous work, we use this property to assert that kernels that modify variables are atomic units that can be treated like instructions that modify registers. In this paper, we extend this concept further by applying basic compiler analysis to Harmony programs. We assert that it is possible express an entire program as a control flow graph (CFG) of interleaved kernels and instructions in the same way that a traditional imperative program can be expressed as a CFG of instructions.

More formally, we can express a program as a directed cyclic graph where nodes are a series of interleaved kernel calls and native instructions terminated by a control decision. Figure 2 shows a possible control flow graph for the PNS inner loop. Edges originate at control decisions and end at possible targets of a given control decision. In this representation, we use the common term basic block (BB) to refer to a node. The only difference between our notion of a BB and the classical notion is that our BBs contain kernel calls as well as native instructions. In the figure, the kernel call and dma operations map to kernels and the other statements map to native instructions.

Using a CFG representation of a program makes it easier to visualize parallelism within a program. Parallelism within kernels in the same BB is limited only data dependencies among kernels. However, for the Harmony applications evaluated in this paper, there are only an average of 5.78 kernels per BB (PNS has 3), and, of course, all kernels are not independent. In order to increase parallelism to an acceptable level to exploit systems with several accelerators, it is necessary to search across basic block boundaries to discover kernels that can be executed in parallel. In this paper, we use speculation to address this problem.

Recall that kernels in Harmony are annotated with explicit input and output variables which allows for easy determination of data dependencies. This problem is more difficult in OpenCL and CUDA because the memory access patterns of kernels are typically not known at compile time. For the PNS example, it happens that all instances of the main Petrinet kernel are completely independent and can be executed in parallel. We have proven this using our CUDA emulator, Ocelot [21], to instrument all of the memory access from each kernel and ensure that there are no conflicting accesses. This represents the best case, where all kernels in the same BB are independent and can be executed in parallel at runtime. Unfortunately, it is not currently possible to assert this property automatically for CUDA applications and this potential parallelism is wasted. Automatically converting CUDA or OpenCL applications to the form used by Harmony could possibly address this problem, but it is beyond the scope of this paper. For our implementation of KLS presented here, we assume that kernels are annotated with explicit inputs and outputs, a process which we see eventually being assisted by a high-level compiler.

We would like to point out that some applications such as PNS could be re-written to increase the degree of par-

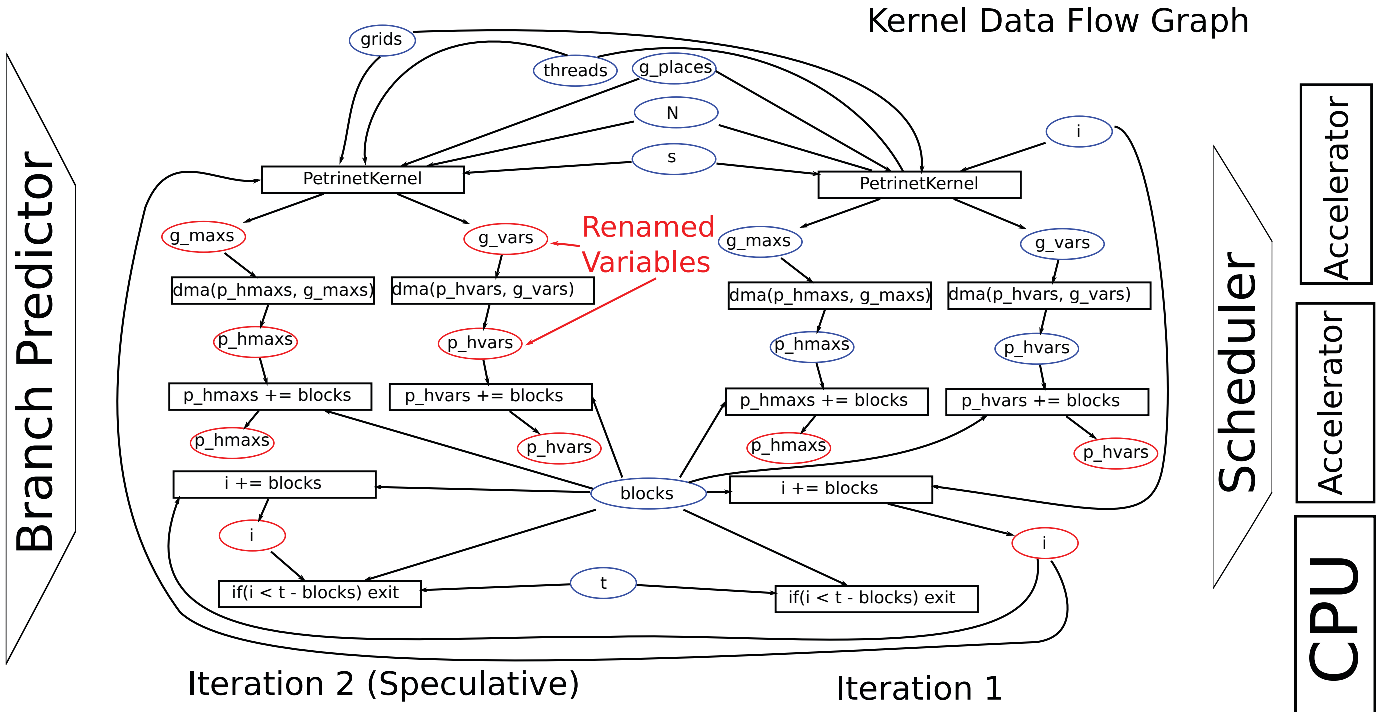

Fig. 5. Execution of a Harmony Application

allelism within a basic block via manual loop unrolling or function in-lining. However, this would defeat the purpose of automatically extracting parallelism from programs that utilize accelerators, which already have to deal with fine-grained thread-level parallelism within kernels. Speculation offers the potential to scale these applications to additional accelerators without requiring the programmer to specify parallelism explicitly. Our goal is to eventually enable existing CUDA applications like PNS to scale across systems with different numbers and types of accelerators. We focus on enabling this property for Harmony applications in this paper and leave the problem of expressing CUDA programs using the Harmony execution model for future work.

B. Program Execution

A high level program such as the PNS example in Figure 1 is compiled to a Harmony control flow graph[2] as in Figure 2. Figure 5 shows the major steps involved in executing a program stored in control flow graph form. The CFG is passed to the runtime; it walks the CFG in program order, examining BBs to determine data dependencies among kernels.

Determining Data Dependencies. In many programming models for heterogeneous systems including CUDA and OpenCL, processors are assumed to reside in disjoint memory spaces. In order to execute a kernel on another processor, it is necessary to allocate and copy memory segments from one address space to another before launching the kernel. In most cases, memory segments are managed at the level of variably sized contiguous blocks to facilitate DMA operations (lines 13-14 in Figure 1). In the Harmony execution model, these memory segments are referred to as variables and are tracked explicitly by the runtime. Before launching a kernel on a given processor, the runtime checks to make sure that all variables used by the kernel are allocated and copied into the address space of that processor. Intuitively, kernels can be executed in parallel as long as they do not make conflicting accesses to the same variables, ie. they do not have data dependencies.

The Data Dependency Graph. As kernels are fetched from the CFG, data dependencies are expressed in a directed acyclic graph. Nodes in this graph represent kernels and edges represent conflicting reads/writes to variables. Figure 3 shows the data flow graph for a the PNS loop body. During execution, the runtime computes a parallel schedule for fetched kernels subject the to dependency constraints and assigns them to available accelerators. Analytical models similar to those used in Qilin [14] are used to predict the execution time of the kernel on each type of core in the system with 6.53% average error for the applications in Section V. These predictions coupled with the constraints from the data dependency graph are used to compute a schedule of all available kernels on all available cores that minimizes their total execution time.

Speculation At Control Decisions. Without speculation, the runtime must block on control decisions until they have finished execution before fetching the next basic block of kernels from the CFG. In cases where the inputs to control decisions are generated by long running kernels, the parallelism within an application is limited by the runtime's

[2]Note that we currently do not have a compiler to the Harmony execution model from any high level language. All of the applications used in this study were compiled manually.

inability to fetch more kernels from the next BB. Speculating the outcome of a given control decision can alleviate this problem by allowing the runtime to fetch from the next several BBs. In literature, this technique is traditionally referred to as control speculation and has been explored exhaustively as an approach for automatic parallelization of sequential programs. For traditional imperative programs such as the SPEC2000 benchmarks, speculation has been shown to improve performance from a modest 1.8x speedup on a simulated 6-core machine using program demultiplexing [16] to an almost linear 7.8x on an 8-core Intel machine using copy-or-discard [17]. We use a technique inspired by copy-or-discard to implement kernel level speculation.

III. Supporting Speculation

In this section, we outline the high level modifications required to extend the Harmony execution model presented in [13] to support kernel level speculation.

A. Variable Renaming

As variables are tracked by the runtime, it is possible to eliminate false write-after-read and write-after-write dependencies by renaming and reallocating kernel outputs. Renaming intuitively trades additional parallelism for increased memory footprint and allocation latency. During execution the runtime decides to rename kernel output variables if there is already a reader or writer on that variable and there are enough spare accelerator cycles to execute the kernel in parallel. Figures 3 and 4 show the PNS dataflow graph before and after renaming.

Though renaming is used primarily for removing data dependencies, it also provides a mechanism for distinguishing between speculative and non-speculative state. Assuming that the outputs of speculative kernels are renamed, they will not affect the variables accessed by previous non-speculative kernels. In our implementation of speculation, we use this property to color variables based on which speculative kernels produce them. This allows variables that were modified by misspeculated kernels to be easily identified and discarded.

B. Keeping Speculative Variables Separate

In order to distinguish between speculative and non-speculative state, we use a coloring scheme similar to Tian et al.'s approach [17] that tags all kernels and variables with a sequence number, or color, that determines the system state that they belong to.

Coloring Speculative Variables. Every kernel within a non-speculative BB must be executed. Therefore, the execution of a BB changes the state of the variables written to by kernels within the BB. If we assign a color to the complete system state after each BB has executed, the process of executing a BB will change the color of modified variables from that of the previous BB to that of the executed block. For a BB that is executed speculatively, it is necessary to be able to roll back to the color of the previous BB in case of a misspeculation.

Renaming On Speculative Writes. In this model, speculations are made at control decisions which correspond to edges in the CFG. In order to determine the next BB to fetch from, we use a software implementation of branch prediction using combined global and local history as in Pierre et al. [22]. Once the next BB has been determined by the software branch predictor, the runtime assigns a speculative color to all kernels in that BB. The renaming mechanism is extended to always rename variables that are written to by a speculative kernel. As variables are written to by speculative kernels, they inherit the kernel's color. In the case of a series of speculations, a different speculative color will be assigned to each basic block. Variables are renamed whenever they are written to by a kernel with a different speculative color than the variable's color. Additionally, the runtime does not discard the copy of a variable with the most recent non-speculative color. As control decisions are resolved and speculative colors are confirmed to be correct, all kernels with that color are reassigned non-speculative colors, old copies of renamed variables are discarded, and speculative variables with that color are assigned non-speculative colors.

C. Handling Misspeculation

If a control decision resolves to a different target BB than it was predicted to, it is considered to be misspeculated and the system state must be reverted back to the color of the immediately preceding BB.

Reseting Control Flow. In case of a misspeculation, the runtime must flush all kernels and variables with a speculative color greater than the color of the misspeculated control decision. This does not mean that all speculative state is rolled back, only the state modified by BBs after the misspeculated control decision. Since control decisions can be resolved out of order, it is possible that a correctly speculated control decision precedes an incorrectly speculated control decision in program order that resolves earlier, resulting in a subset of all outstanding speculative colors being rolled back. After the state has been reverted, the runtime front-end can resume fetching BBs from the correct path.

Deallocating Resources. Once a control decision is determined to have been misspeculated, all kernels with subsequent speculative colors are flushed. It would be useful to support interrupting misspeculated kernels as soon as possible. However, the underlying libraries that we use to launch kernels (pthreads on x86 and cudart on NVIDIA GPUs) do not support reliable asynchronous kill operations [3].

Errors In Speculative Kernels. Speculative kernels can possibly be given incorrect input data resulting in undefined behavior such as making out of bounds memory accesses, executing invalid instructions, or looping indefinitely. This is a problem of any speculative system and numerous solutions have been presented in the past to deal with it [16], [17]. In our implementation, we leverage the side-effect-free property

[3]We do not consider pthread_cancel to be reliable since it does not free dynamically allocated memory

Application	Description	Problem Size	Control Flow	Model
AES	Encrypts and decrypts a large document using 256-bit AES	3.2 MB Text File	For Loops	Harmony
MonteCarlo	Gaussian Quadrature estimates the area under a normal function	1 Precision value	While Loops	Harmony
MatrixMultiply	Dense matrix multiplication using subblocks	4096x4096 matrices	Nested Loops	Harmony
CapModel3	Risk analysis for adding a new asset to an existing loan portfolio	1000000 Samples	Nested Loops	Harmony
Random	A regression test for Harmony that constructs a CFG of simple kernels with random edges subject to a completion constraint	10 Variables 100 Average Iterations	Random Structure	Harmony
MRI-Q	Computation of a matrix Q, representing the scanner configuration, used in a 3D MRI reconstruction algorithm in non-Cartesian space.	450KB Image	For Loops	CUDA
MRI-FHD	Computation of an image-specific matrix FHd, used in a 3D MRI reconstruction algorithm in non-Cartesian space.	450KB Image	For Loops	CUDA
CP	Computes the coulombic potential at each grid point over on plane in a 3D grid in which point charges have been randomly distributed.	40000 Atoms in a 512x512 grid	For Loops	CUDA
SAD	Sum of absolute differences kernel, used in MPEG video encoders.	50KB image	None	CUDA
TPACF	Measures the probability of finding an astronomical body at a given angular distance from another astronomical body.	100 Random Numbers 4096 Points	None	CUDA
PNS	Implements a generic algorithm for Petri net simulation.	2000x2000 matrix	For Loops	CUDA
RPES	Calculates 2-electron repulsion integrals which represent the Coulomb interaction between electrons in molecules.	20000 molecules	For Loops	CUDA

TABLE I
APPLICATION CHARACTERISTICS

of kernels to determine that all memory accesses that do not correspond to registered inputs or outputs of the kernel are memory errors. These can be handled with appropriate MMU support. Invalid instructions can similarly be handled by trapping faults until the kernel that generated them becomes nonspeculative. Infinite loops are currently not handled in our implementation, though they could be easily supported if we could asynchronously kill running kernels. For our benchmark applications, we did not encounter any of these errors.

D. Deciding When to Speculate

Balancing the benefits against the overheads of speculation for a given kernel requires a decision model to determine whether or not to launch a speculative kernel. A simple expression of the form of equation (1) relates the probability of correctly predicting a control decision P_C to the expected benefit of launching a kernel speculatively $E(T_B)$. Intuitively, the product of the probability of correctly speculating a kernel and the reduction in execution time T_E plus the reduction in memory copy time T_M represents the expected benefit of a given speculation. Similarly, the product of the execution time of a misspeculated kernel T_{RB} and the probability of incorrectly speculating a kernel $1-P_C$ expresses the expected overhead of a given speculation. Our runtime uses a decision model that chooses to launch a given kernel speculative only if the expected benefit is greater than the expected overhead.

$$E(T_B) = P_C * (T_E + T_M) - (1 - P_C) * (T_{RB}) \quad (1)$$

E. Scheduling Optimizations - Control Decision Criticality

Our first implementation of a scheduler for Harmony was based on the widely used list scheduling algorithm. We used a ranking function that assigned the predicted execution time of each kernel in the data dependency graph as a weight in the scheduling algorithm. Based on the performance of this initial implementation, we made several noteworthy modifications to address the criticality of control decisions.

Even in the presence of speculation, we found that it was still advantageous to resolve control decisions as soon as possible so that misspeculated kernels could be flushed before they were launched, previous copies of renamed variables could be discarded, and control flow could continue at the next correct basic block. This suggested that control decisions were more critical to the execution of a program than generic kernels. More generally, data dependency chains in the program data-flow graph that contain a control decision are critical to the execution of the program because they determine the latency required to resolve a control decision. In our initial implementation of speculation, the scheduler was agnostic to the criticality of control decisions, resulting in several cases where control decisions would be scheduled behind particularly long running kernels, artificially increasing the latency of resolving a given control decision.

To address this problem, we implemented an improved scheduling algorithm that gave preference to control decisions. This algorithm improved average performance by up to 20% over the base implementation. However, it still led to cases where kernels that produced values used as inputs to control decisions could be scheduled behind long running independent kernels. The scheduling algorithm was again revised to identify data-dependency chains of kernels ending in control decisions. These kernels were assigned high priority status and scheduled before lower priority (any other) kernels.

IV. KERNEL LEVEL PARALLELISM

In this section, we develop a theoretical formulation of the upper bound of KLP in Harmony and CUDA applications. We show how this upper bound is impacted by speculation and renaming for the benchmark applications in Table I. The Harmony applications were written from scratch by us and the CUDA applications were taken from the UIUC Parboil benchmark suite [20].

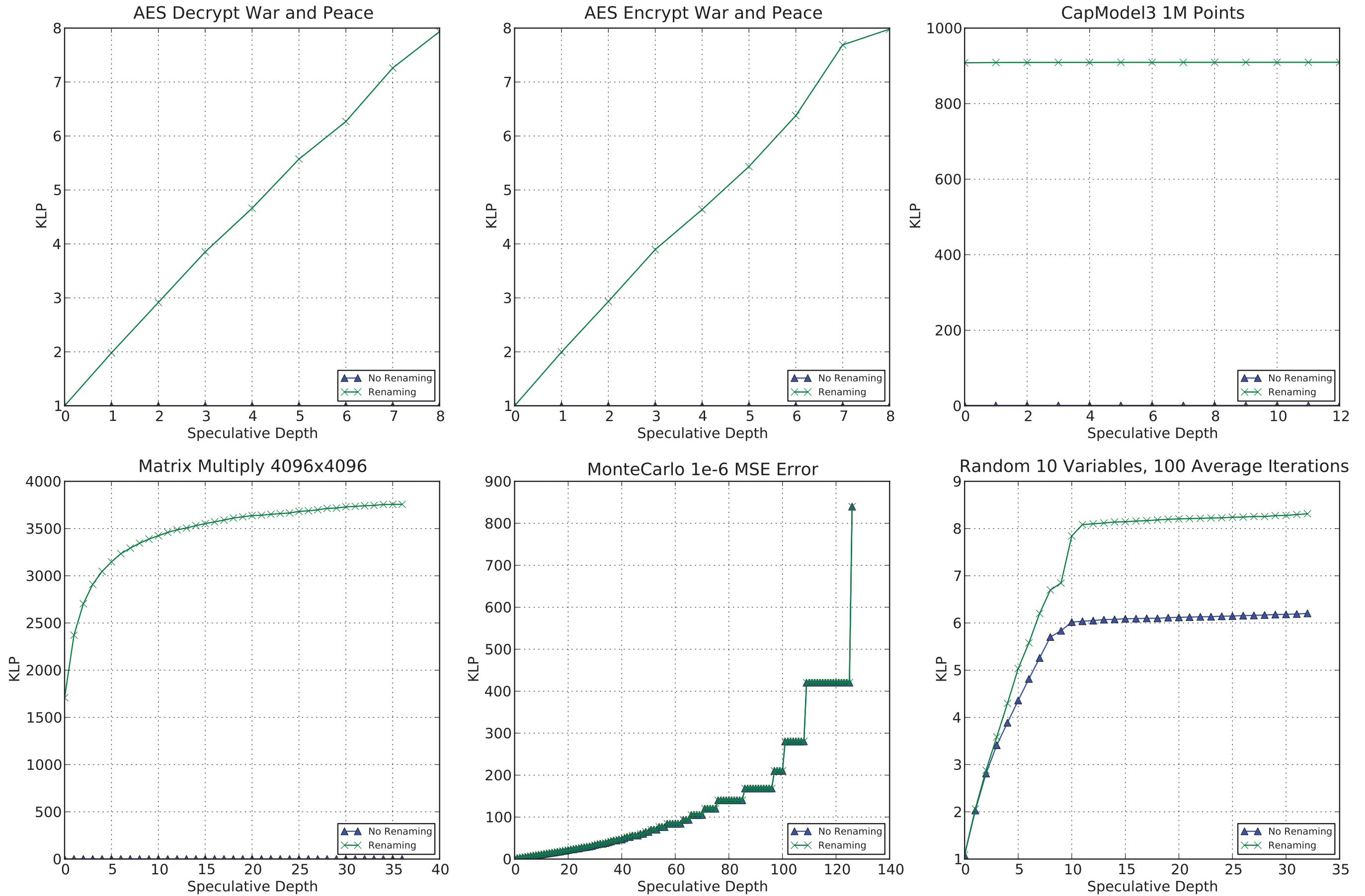

Fig. 6. Kernel Level Parallelism in Harmony Applications

Application	Kernels	KLP	MIMD	SIMD
CP	10	9.85	256	128
MRI-Q	4	3.91	97.5	320
MRI-FHD	7	6.96	110.57	292.57
SAD	3	2.6	594	70.28
TPACF	1	1.0	156.63	206.11
PNS	112	111.03	17.99	248.88
RPES	71	70.42	64757	40.5799

TABLE II
PARALLELISM IN CUDA APPLICATIONS

A. KLP Definition

At a high level, we would like a metric that expresses the amount of parallelism within a Harmony or CUDA application in the same way that ILP expresses the amount of instruction level parallelism within a single threaded application. KLP is difficult to formulate exactly as different kernels typically have different, data-dependent execution times as shown in Diamos et al. [13] and Luk et al. [14]. With these concerns in mind, *we define kernel level parallelism for an application on a heterogeneous system as the speedup of a parallel execution on a system with an infinite number of accelerators over a sequential execution on the same system where each kernel is run on the accelerator that gives the lowest execution time*. In order to account for possible non-determinism in the execution time of a kernel, we use the average execution time from the accelerator with the lowest such average time.

For Harmony applications, we computed KLP by analyzing traces of the kernels and control decisions launched by an application as it executed. These traces expressed the average kernel execution time and the input and output variables of each kernel. For CUDA applications, we used Ocelot [21] to instrument all load and store instructions from every kernel in an application. We maintained a set of all memory locations written by kernels along with the id of the last kernel to write to that location. If a kernel ever loaded a value that was stored by a previous kernel, we created a dependency between the two kernels. This information was used to create a dependency graph for the entire application. We did not have the ability to identify control decisions in CUDA applications, so we report on the best case KLP assuming that all control decisions could be removed via perfect speculation in Table II. We also present the average MIMD and SIMD parallelism as defined in Kerr et al. [21] within each kernel for comparison.

B. Results

Figure 6 shows the computed upper bound on KLP for all of the Harmony applications in our test suite. In this figure, speculative depth refers to the maximum number of control decisions that can be outstanding at a time. These results show a significant amount of KLP within all applications tested. For all of the applications except Monte Carlo, renaming provides the most significant boost to KLP and indeed most applications without renaming do not show any improvement from speculation. Monte Carlo stands out because it explicitly uses different variables for the input seed and output result of each Monte Carlo simulation; this demonstares that it is possible for the programmer to do the equivalent of renaming. However, once renaming has been enabled, being able to remove control decisions via speculation greatly improves KLP. Over all of the applications, it extends the upper bound of KLP by an average of 3.6x over renaming alone. The KLP saturates around a speculative depth of about 10 except for Monte Carlo which continues to scale up to a speculative depth of 133.

For the CUDA applications, KLP is comparable to that of the Harmony applications in all cases except for SAD, TPACF and MRI-Q, which simply do not launch a significant number of kernels. It is possible that increasing the data set size for these benchmarks would improve their KLP. It is also possible that the kernels in these applications would have to be split to expose additional KLP, which would limit the potential benefits of speculation. Kerr et al. [21] show that this is not the common case for CUDA applications, which typically have tens to hundreds of kernels, but it still represents a problem that will have to be addressed in future work in order to apply kernel level speculation to certain CUDA applications.

For the remainder of the paper, we focus on the Harmony benchmarks exclusively as our runtime prototype does not yet support CUDA. However, from these KLP results, we hypothesize that many CUDA applications have enough KLP to benefit from speculation as well. This hypothesis may or may not be correct, but we believe that these results warrant deeper investigation in future work.

C. Branch Prediction Accuracy

The KLP measurements in Figure 6 give an upper bound on parallelism for Harmony applications assuming that all branches can be predicted with perfect accuracy. Branch predictors used for speculation in out-of-order processor can typically achieve over 95% accuracy [22]. Our runtime uses a combined branch predictor as in [22] with 16 bits of global history and a history table that maintains a unique entry for each control decision and global history value. This is possible because our implementation is done in software, and table entries are lazily allocated upon their first use. We instrumented the branch predictor in our runtime to report the average accuracy for each Harmony application using several history sizes. Figure 7 shows that the accuracy of our predictor on Harmony applications is comparable to that of state of the art hardware predictors and that control decisions in Harmony applications are about as easy to predict as branches in SPEC2000 benchmarks. For the subsequent analysis our branch predictor is set to use tables of 1024 entries each.

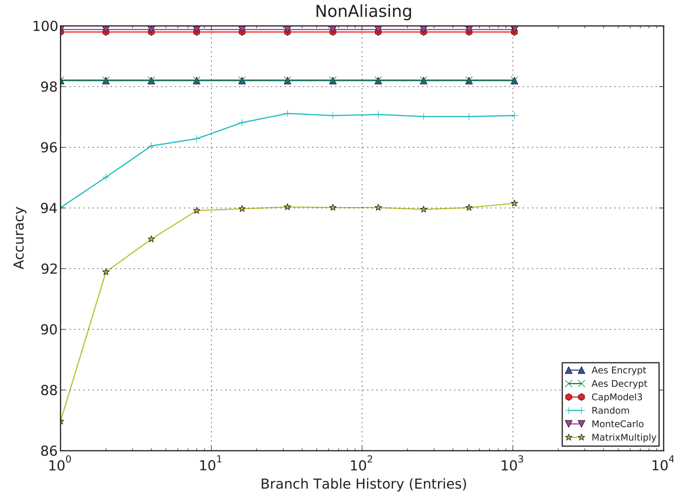

Fig. 7. Branch Predictor Accuracy

CPU	Intel Core2 Quad-Core 2.33Ghz
Accelerator 1	NVIDIA 9800GX2 (2-GPUs)
Accelerator 2	NVIDIA Tesla s870 (4-GPUs)
CPU Compiler	GCC-4.3.2
GPU Compiler	NVCC-2.1
OS	64-bit Ubuntu 9.04

TABLE III
TEST SYSTEM

V. Experiments

This section covers an empircal evaluation of our implementation of KLS running on a highly heterogeneous system. The characteristics of our test system are given in Table III. We begin by comparing measured scaling to our KLP model, then present the base case execution time of each Harmony application using multiple system configurations, and conclude with the execution time of the complete system with and without speculation.

A. KLP Comparison

The KLP metric presented in Section IV represents an upper bound on the parallel scaling of a given application. In order to evaluate the effectiveness of our implementation of speculation in relation to the KLP ideal, we first measure the execution time of each application without speculation. The KLP metric for each speculative depth is then normalized to the measured non-speculative time, eg. an application with an execution time of 10s and a KLP of 2 at depth 1 would be predicted to finish in 5s on a machine with at least two cores. For this experiment, we use only CPU cores for both the KLP metric as well as the measured execution time so that scaling trends are easily visible.

Figure 8 shows the measured and KLP predicted execution times. The MonteCalo, CapModel3, and MatrixMultiply

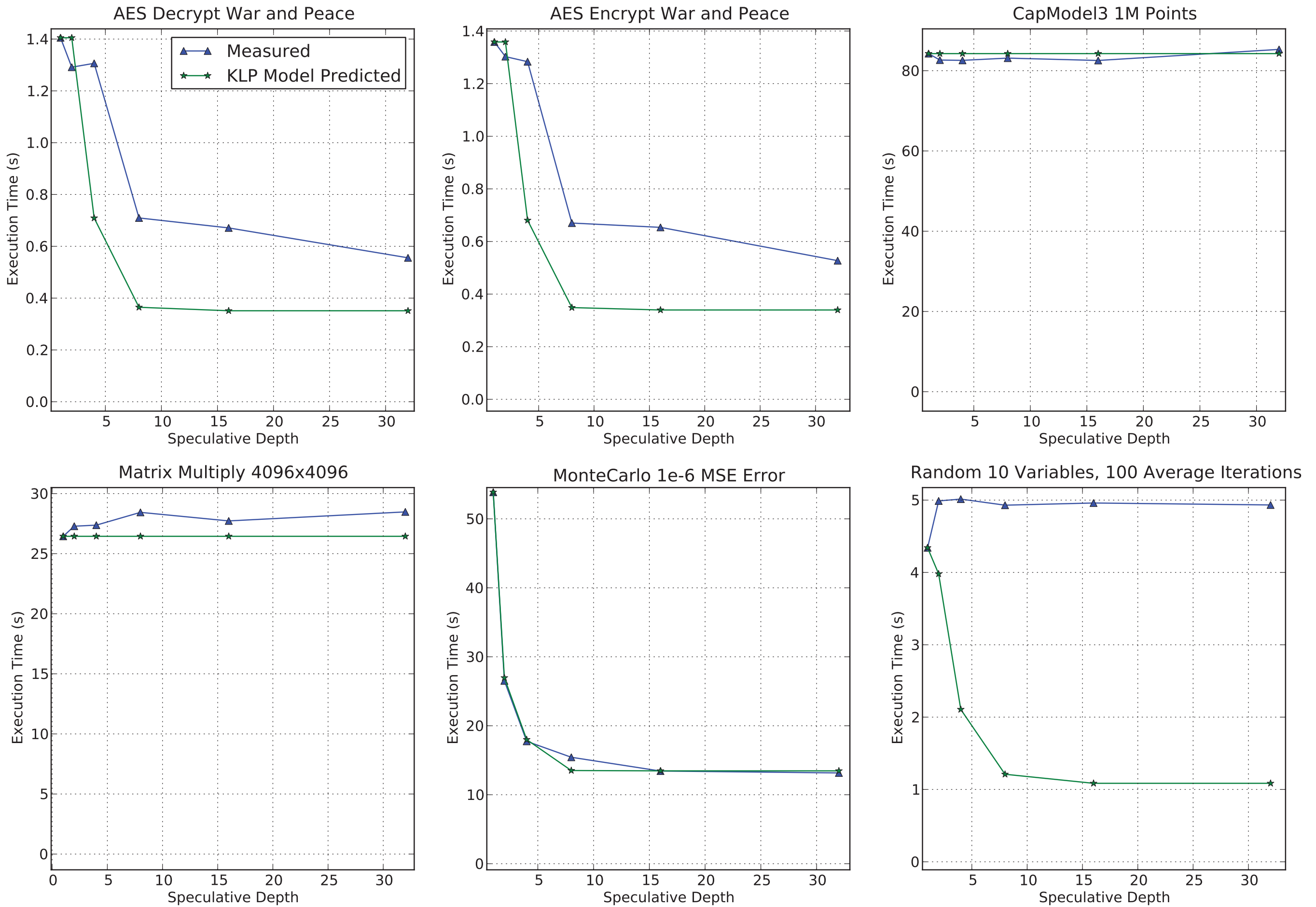

Fig. 8. KLP Predicted vs Measured Scaling (4 CPUs)

follow the prediction closely. Random deviates the most, experiencing no speedup even though the KLP metric predicts a speedup of up to 4x. This can be explained by examining the structure of the application, where kernels only contain several instructions each; the execution time is dominated by runtime overheads, which are serialized. The AES application suffers from inaccuracy as well. Profiling the application shows that it spends most of its time in functions doing memory and disk accesses suggesting that it is IO rather than compute bound, limiting its scalability on a multicore system.

B. *Heterogeneous Scaling*

This experiment establishes a base case for the speedup of each application using Harmony without KLS[4]. As can be seen in Figure 9, on average, the use of all 10 cores in the system provides a 14.8x increase in performance over a single CPU with the MatrixMultiply example seeing the largest improvement at 57.6x. For MatrixMultiply, the complete system achieves 824 Gflops compared to 201 Gflops achieved by a single 8800GTX GPU in prior work [23].

[4]Random is omitted from these results since it does not use GPU kernels.

Only the AES application experiences a slow-down moving from a single CPU to the entire system. In our GPU implementation, the GPU encrypt and decrypt are at least an order of magnitude slower than their CPU equivalent. This is not a typical result for AES, which others have shown to perform well on GPUs [24], and is likely due to an inefficiency in our implementation. However, it presents an interesting case where a GPU kernel is much slower than a CPU kernel. As these applications only have ten encrypt or decrypt kernels each, running even a single kernel on a GPU core will degrade the performance of the application. Examples like this motivate the refinement of the performance predictor to either try to estimate the execution time on each architecture before actually launching a kernel, or kill extremely long running kernels and restart them on faster architectures.

C. *Additional Scaling Using Speculation*

The next experiment focuses on the benefits from adding speculation to the base implementation. Figure 10 shows the performance improvement of the entire system moving from the non-speculative implementation to the speculative implementation using various speculative depths. Of the applications

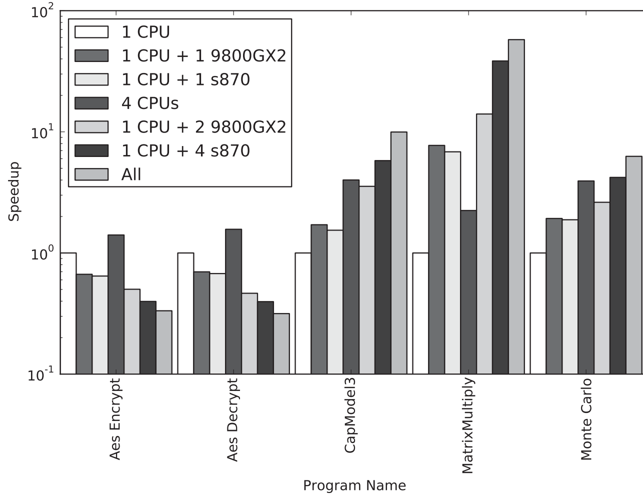

Fig. 9. Scaling Without Speculation

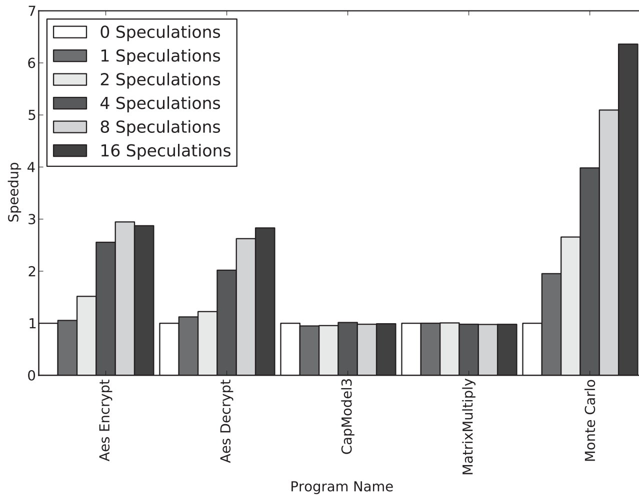

Fig. 10. Performance Improvement With Varying Speculative Depth

that were predicted by the KLP model to benefit from speculation, performance improves by an average of 3.98x. The other two applications, CapModel3 and MatrixMultiply, are not affected at all by the overheads of speculation, experiencing a $\pm 3\%$ change in execution time. These two applications already have enough KLP within each basic block to fully utilize the system, and thus do not require speculation to expose any more. Across all of these applications, adding kernel level speculation either improves performance or does not impact it at all.

VI. Related Work

A. Heterogeneous Many-Core Programming

BrookGPU [7] proposes the use of stream extensions to the C programming language where compute kernels are defined to be functions applied to every element in a stream. The definition of a kernel explicitly declares stream parameters as inputs and outputs enabling the determination of data dependencies between kernels. Similarly, *StreamIt* [8] applies uniform operations (kernels although they do not use this terminology) to each element in several input data streams to produce an output data stream. Complete programs are composed of a data-flow-like graph of kernels. In each of these languages, a runtime component maps kernels onto processing elements. *BrookGPU* passes kernels to the runtime as they are encountered during the execution of the application, whereas in *StreamIt*, the entire program is directly made visible to the runtime.

CUDA [10] and OpenCL [11] begin with the C programming language and again introduce the concept of kernels which are executed on GPUs in the case of *CUDA* or generic accelerators in the case of *OpenCL*. Kernels in this context are different in that they can operate on any data structure, not just streams, and assume a Single-Program Multiple-Data execution model within kernels where the number of threads launched per kernel is explicitly stated by the programmer. They also drop the requirements for specifying parameters as read-only inputs or write-only outputs, but they keep the restriction that kernels are side-effect-free and can only update local variables in accelerator memory.

Sequoia [15] models heterogeneous systems as an arbitrarily structured tree of distinct memory modules with the leaves containing processors. The programmer must orchestrate transfers up and down the tree as well as mapping kernels to leaf nodes. *Sequoia* kernels maintain the explicit input/output semantics for kernel parameters and the side-effect-free restriction that they can only operate on local data. Imperative control flow is permitted, but cannot be easily decomposed into parallel code. *Merge* [12] uses a similar map-reduce tree style programming model, but without programmer orchestrated data movement up and down the memory hierarchy. It supports heterogeneity by including multiple implementations of each kernel and its runtime uses simple sampling to bias specific kernels to faster cores.

Qilin [14] Is by far the most similar execution model to Harmony. Qilin allows programs to be specified either in terms of Intel TBB [25] for CPU kernels or NVIDIA CUDA [10] for GPU kernels. Like the approach used in our prior work [13], a directed acyclic dependency graph of kernels is created by a runtime component as the program executes. Qilin uses the term adaptive mapping to refer to the process by which the runtime determines which kernels can execute in parallel and maps them to available accelerators, dynamically choosing either the CPU or GPU implementation, similar to the scheduling process in Harmony. Additionally, Qilin uses an analytical performance model to determine the execution time of individual kernels on specific accelerators, which is very similar to our approach. Qilin retains a sequential programming model, but like the original formulation of Harmony, it can only exploit parallelism within a single basic block.

The *Harmony* [13] execution model draws from these high level languages the concepts of side-effect-free kernels, explicit input and output parameters, and kernels that can operate on generic data structures. It extends these execution models to make the entire program control flow graph visible to the

runtime in the same way that the program data flow graph is made visible to the *StreamIt* runtime and the map/reduce tree is made visible to *Merge*. These abstractions allow us to extend the techniques described in these prior works to support executing kernels speculatively. All of these prior works either impose an explicitly parallel programming model at the inter-accelerator level, or employ a sequential programming model without the ability to search beyond basic block boundaries for additional parallelism.

B. Speculation

Tian et al. [17] present a copy-or-discard mechanism for unrolling generic imperative loops via speculation by running speculative threads for each loop iteration. Variables modified by each speculative thread are stored locally and upon completion are either copied back into the main thread or discarded if they were modified by the main thread. Similarly, Program-Demultiplexing [16] uses compiler analysis to identify functions in imperative programs that are side-effect-free and can be executed speculatively. Their implementation requires hardware-support to buffer speculative memory operations, but their concept of a side-effect-free function is very similar to our concept of a kernel.

Several proposed schemes [18], [26], [27] exist for thread level speculation (TLS) where threads are spawned and allowed to proceed ahead of a main thread. Hardware support is required to detect memory dependency violations between the speculative thread and the main thread. Typically, writes from speculative threads are buffered in hardware and only committed after the thread becomes non-speculative. Also, the points at which to launch speculative threads are added by the compiler [18].

Finally, Eric Petit and Francois Bodin propose a software only approach for thread level speculation in systems with attached accelerators [28]. In this study, threads with the potential for acceleration are identified using a combination of static compiler analysis and profiling before being speculatively assigned to discrete accelerators without shared memory. Only those regions of memory that are expected to be accessed by the speculative thread are copied into the accelerator's memory space and, in the case the thread accesses an unmapped region, the program faults and falls back on execution on the main processor. Their study focuses mainly on the identification and partitioning of a sequential application into threads, whereas our approach relies on the concept of encapsulated kernels embedded in the targeted programming models.

We drew upon these previous implementations of speculation to guide our implementation of KLS. The techniques that we describe in this paper are not fundamentally different from these previous works. Instead, we show that the techniques described in these prior works and implemented in our prototype of the Harmony runtime enable the retention of a sequential programming model for heterogeneous systems with several accelerators that can efficiently utilize all of the accelerators in a large system.

VII. Conclusions

We have leveraged the abstractions offered by kernel programming languages to design a software implementation of speculation for heterogeneous many-core systems, augmenting the our prior work on the Harmony execution model [13]. Our implementation uses speculation to break control dependencies between kernels and increase program concurrency using software branch prediction coupled with memory renaming to distinguish between speculative and nonspeculative state.

Compared to a theoretical upper bound on the performance improvement from speculation assuming oracle predictors and infinite accelerator resources, we show that our implementation achieves $41.2\% - 98.6\%$ of the theoretical ideal across 6 full applications running on a system with 10 cores and 3 different architectures, resulting in a 1.02x-6.13x speedup. Additionally, we show that many CUDA applications have a similar degree of kernel level parallelism as the applications evaluated in this paper. In the future, we plan to explore expressing CUDA program as Harmony programs such that they can benefit from optimizations like Kernel Level Speculation.

Acknowledgement

The authors gratefully acknowledge the generous support of this work by LogicBlox Inc. and NVIDIA Corp. both through research grants, fellowships, and technical interactions, and equipment grants from Intel Corp. and NVIDIA Corp.

References

[1] B. Shekhar, K. Tanay, N. Siva, T. Jim, K. Ali, and D. Vivek, "Parameter variations and impact on circuits and microarchitecture," in *DAC '03: Proceedings of the 40th conference on Design automation*. New York, NY, USA: ACM, 2003, pp. 338–342.

[2] D. Kirk, "Nvidia cuda software and gpu parallel computing architecture," in *ISMM '07: Proceedings of the 6th international symposium on Memory management*. New York, NY, USA: ACM, 2007, pp. 103–104.

[3] W. Mark, L. Yuan, S. Sangwon, M. Scott, M. Trevor, C. Chaitali, B. Richard, K. Danny, R. Alastair, W. Mladen, and F. Krisztian, "From soda to scotch: The evolution of a wireless baseband processor," in *MICRO '08: Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 152–163.

[4] A. Elbirt and C. Paar, "An fpga implementation and performance evaluation of the serpent block cipher," in *FPGA '00: Proceedings of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays*. New York, NY, USA: ACM, 2000, pp. 33–40.

[5] K. H. Randall, "Cilk: Efficient multithreaded computing," Ph.D. dissertation, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1998.

[6] L. V. Kale and S. Krishnan, "Charm++: A portable concurrent object oriented system based on c++," Champaign, IL, USA, Tech. Rep., 1993.

[7] J. Gummaraju and M. Rosenblum, "Stream Programming on General-Purpose Processors," in *MICRO 38: Proceedings of the 38th annual ACM/IEEE international symposium on Microarchitecture*, Barcelona, Spain, November 2005.

[8] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "Streamit: A language for streaming applications," in *CC '02: Proceedings of the 11th International Conference on Compiler Construction*. London, UK: Springer-Verlag, 2002, pp. 179–196.

[9] AMD, "Ati stream computing - technical overview," One AMD Place, Sunnyvale CA, 94088, Tech. Rep. [Online]. Available: http://developer.amd.com/gpu_assets/Stream_Computing_Overview.pdf

[10] NVIDIA, *NVIDIA CUDA Compute Unified Device Architecture*, 2nd ed., NVIDIA Corporation, Santa Clara, California, October 2008.

[11] K. O. W. Group, *The OpenCL Specification*, December 2008. [Online]. Available: http://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf

[12] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, "Merge: a programming model for heterogeneous multi-core systems," in *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 2008, pp. 287–296.

[13] G. Diamos and S. Yalamanchili, "Harmony: An execution model and runtime for heterogeneous many core systems," in *HPDC'08*. Boston, Massachusetts, USA: ACM, june 2008.

[14] C. Luk, S. Hong, and H. Kim, "Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *MICRO'09*. New York, USA: IEEE, devember 2009.

[15] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: Programming the memory hierarchy," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[16] 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.

[17] C. Tian, M. Feng, Nagarajan, Vijay, and R. Gupta, "Copy or discard execution model for speculative parallelization on multicores," in *MICRO '08: Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 330–341.

[18] L. Hammond, M. Willey, and K. Olukotun, "Data speculation support for a chip multiprocessor," in *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 1998, pp. 58–69.

[19] NVIDIA, *NVIDIA Compute PTX: Parallel Thread Execution*, 1st ed., NVIDIA Corporation, Santa Clara, California, October 2008.

[20] IMPACT, "The parboil benchmark suite," 2007. [Online]. Available: http://www.crhc.uiuc.edu/IMPACT/parboil.php

[21] A. Kerr, G. Diamos, and S. Yalamanchili, "A characterization and analysis of ptx kernels," in *IISWC09: IEEE International Symposium on Workload Characterization*, Austin, TX, USA, October 2009.

[22] M. Pierre, S. André, and U. Richard, "Trading conflict and capacity aliasing in conditional branch predictors," in *ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 1997, pp. 292–303.

[23] V. Volkov and J. W. Demmel, "Benchmarking gpus to tune dense linear algebra," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–11.

[24] O. Harrison and J. Waldron, "Practical symmetric key cryptography on modern graphics hardware," in *SS'08: Proceedings of the 17th conference on Security symposium*. Berkeley, CA, USA: USENIX Association, 2008, pp. 195–209.

[25] G. Contreras and M. Martonosi, "Characterizing and improving the performance of the intel threading building blocks runtime system," in *International Symposium on Workload Characterization (IISWC 2008)*, September 2008. [Online]. Available: http://www.gigascale.org/pubs/1350.html

[26] M. Cintra, J. F. Martínez, and J. Torrellas, "Architectural support for scalable speculative parallelization in shared-memory multiprocessors," *SIGARCH Comput. Archit. News*, vol. 28, no. 2, pp. 13–24, 2000.

[27] B. Anasua and F. Manoj, "A general compiler framework for speculative multithreading," in *SPAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA: ACM, 2002, pp. 99–108.

[28] E. Petit and F. Bodin, "Extracting threads using traces for system on a chip," in *12th International Workshop on Compilers for Parallel Computers (CPC)*, A Coruna, Spain, January 2006.