Harmony

Runtime Techniques for Dynamic Concurrency Inference, Resource Constrained Hierarchical Scheduling, and Online Optimization in Heterogeneous Multiprocessor Systems

Gregory Diamos and Sudhakar Yalamanchili
Georgia Institute of Technology
Computer Architecture and Systems Lab
Atlanta, Georgia, USA
{gregory.diamos@gatech.edu, sudha@ece.gatech.edu}

Abstract

The emergence of heterogeneous multiprocessor architectures presents a unique opportunity for delivering order of magnitude performance increases for high performance applications by matching certain classes of algorithms to specifically tailored architectures. Their widespread adoption, however, has been limited by: (1) complex programming models that can vary dramatically between architectures, (2) difficulty in determining an efficient parallel partitioning of an application into compute kernels across all architectures in a system, and, in general, (3) a lack of scalability with system resources.

This paper presents Harmony, a collection of techniques that provide runtime services for dynamically inferring concurrency between compute-kernels, hierarchical variable-granularity scheduling, and performance monitoring and online optimization in the context of heterogeneous multiprocessor systems. Experimental results presented in this paper using a prototype implementation of Harmony demonstrate the usefulness of these techniques for enabling the development, migration, and scalability of parallel matrix multiplication, ray tracing, and software-pipelined audio frequency analysis across several heterogeneous multiprocessor system configurations.

1. Introduction

WHEN asked for his perspective on the challenges of integrating heterogeneous accelerators with future High Performance Computing (HPC) systems, the senior director of Intel's HPC Unit responded that

> **The greater challenge is not so much the connectivity, but the software that allows heterogeneous computing to have continuous benefit. It's important for a platform using an accelerator-of-choice one year to work with next-generation and prior-generation processors. That's a major software challenge. We [Intel] believe the market will welcome accelerators ... that can support software continuity from one generation to the next [1].**

This statement captures the heart of the problem preventing systems combining general purpose computation with heterogeneous accelerators from gaining mainstream adoption: **introducing heterogeneity trades complexity for performance**, and for all but a few select applications, the complexity cost is too high to justify the performance gains. It has been shown over and over again that specific HPC applications like N-Body Simulations [2], Molecular Dynamics [3], Numerical Weather Modeling [4], and Terrain Rendering [5] along with a growing number of end user applications like Game Servers [6], AES Cryptography [7], and Real-Time Ray Tracing [8] can experience order of magnitude or greater speedups when paired with architectures that are specifically tailored to their needs. This trend has been noticed and gained some momentum in the HPC community headed by the Los Alamos National Labs (LANL) Roadrunner project [9]. However, a consensus seems to have formed that the amount of effort required to port or rewrite applications to take advantage of these heterogeneous architectures without middleware support is non-trivial at best and herculean at worst [10].

An application that attempts to use two or more heterogeneous processors cooperatively must be partitioned into segments (hence forth referred to as kernels) to run in parallel on some set of cores in the system in order to fully utilize the resources offered by the system – this is an open problem even in the case of systems with homogeneous multiprocessors that is compounded by the introduction of heterogeneity. In the heterogeneous case, parallel partitioning must match application kernels to the most efficient architecture while taking care to balance the computational load on all processors in the system and guarantying correct execution by considering communication and synchronization between kernels. If partitioning is done statically for only one system configuration with a specific set of processors, then the application is tied to that configuration and will re-

quire porting or rewriting to be run on a system with a different set of processors. Of course, this limits the scalability of applications written for such systems: users concerned with high performance can not get seedups simply by buying bigger and better systems with more accelerators, and budget constrained users cannot run the same applications on more modest systems. **These limitations motivate the need for middleware applications and runtime environments that allow applications to harmoniously utilize the heterogeneous resources offered by a system in a way that enables portability and scalability across a range of configurations** and reduces the relatively large initial investment in designing such applications due to their inherently high complexity.

Previous work on application development and execution for heterogeneous multiprocessor systems has focused on case studies where the performance of particularly demanding applications is evaluated on relatively exotic architectures like FPGAs, GPUs, IBM Cells [11], Intel IXPs [12], Clearspeed SIMD Processors [13], and others. Relatively few have examined the more general problems of compatibility, scalability, design complexity, and high performance on heterogeneous platforms for generic applications, being content instead to addressing them for specific high profile applications. Notable exceptions include an examination of scheduling multi-grained parallelism on IBM Cell [14], a new programming language for parallel systems with heterogeneous memory hierarchies [15], and compiler techniques for parallelism on the IBM Cell [16].

This paper presents Harmony, a collection of techniques for efficient, low overhead, runtime partitioning, mapping, and optimization of sequentially specified imperative programs to HMPs and clusters of HMP based systems. Harmony provides novel functionality that improves or compliments existing techniques for developing and dynamically mapping applications to HMPs. To start, Harmony provides a new programming model for specifying an application as a sequence of control structures and compute kernels with associated meta-data. The addition of meta-data, which can be inserted by the programmer or an advanced compiler, allows for low overhead parallel partitioning at runtime that is bounded in complexity by $O(N^2)$ where N is the number of kernels partitioned in parallel. Additionally, compute kernels can be compiled separately, or linked against pre-compiled binaries, for different architectures – allowing the application to be run on any HMP system with a minimal set of processor architectures. Second, it extends a methodology for resource constrained scheduling in the presence of data dependent kernel latencies [17] to schedule kernels that could potentially run on one or more heterogeneous processors, each with different performance characteristics. The dynamic mapping from kernels to processor architectures allows applications to potentially improve performance by adding additional heterogeneous processors or moving kernels to less efficient architectures to reduce contention for highly efficient architectures. Third, a multivariate regression model is proposed for predicting the execution time of a kernel on a specific architecture based on statically computed meta-data and runtime dependent data. These predictions can be back annotated to the scheduler to improve the quality of the scheduling decisions. Fourth, it introduces a technique for combining or dividing compute kernels based on their predicted execution time to balance scheduling quality with overheads. Fifth, it addresses the concept of speculation across control decisions to improve concurrency and across free processor resources to reduce kernel execution times when predictions have low confidence. Finally, we show how these techniques can be recursively applied to hierarchically schedule and map applications to clusters of HMP systems.

Our research uses Harmony to evaluate implementations of embarrassingly parallel scientific applications and media applications with complex dependencies. In the context of scientific applications, of which a large number depend on compute intensive linear algebra operations, dense matrix multiplication benefits from Harmony's ability dynamically infer concurrency and balance load across system resources. A more complicated example of parallel ray tracing makes use of dependency resolution techniques present in Harmony to avoid loop-carry dependencies. For multimedia applications, Harmony is shown to improve thread-level parallelism in the presence of non-trivial inter-kernel dependencies. This example demonstrates the ability of the performance monitoring and scheduling facets of Harmony to bias the execution of certain kernels on specific architectures that have highly efficient implementations.

The results presented in Section 8 show the usefulness of Harmony's dynamic mapping from compute kernels to heterogeneous architectures. More specifically, they show that for an application where each kernel has an implementation for at least one architecture (which could be commonplace like an x86 processor), the the application will be portable across all systems that have at least one processor with that architecture. As an extension, we show how performance can scale nearly linearly by adding more accelerators as long as there is sufficient kernel-level parallelism within an application. An evaluation of concurrency inference overheads demonstrates that for a scheduling window size of N kernels, the complexity of determining the dependencies among all kernels in the window is $1/2 * (N^2 - N)$ comparisons. This process can produce comparable results to explicit partitioning via MPI (as shown in the results section) and requires much less overhead than compiler based techniques for concurrency inference like loop-level evaluations [18] [19] or unroll-and-pack operations [20] and is not limited by constraints such as loop-carry-dependencies for extracting vector-level parallelism, allowing the parallel partitioning to be done at runtime without introducing excessive overheads. An evaluation of the overhead of this parallel partitioning combined with a greedy scheduler shows that the total partitioning, scheduling, and synchronization overhead is only 5.3% of the total execution time of a sequential

64x64 dense matrix multiply when scheduling up to 128 matrix multiply kernels in parallel. Increasing the granularity of the kernels being partitioned or reducing the size of the scheduling window can be used to reduce this overhead even further. Compared to our multivariate regression model, other approaches to dynamically determining the efficiency of a particular kernel when executed on a specific architecture [14] assume that the execution time of a specific kernel remain constant for all executions of that kernel. Unfortunately, this is only valid if the computational cost of a compute kernel is independent of the data upon which it executes. The end result is a set of techniques that perform well on a certain class of applications, but make ill informed decisions in the more general case.

Figure 1 depicts an example using the middleware services provided by the Harmony runtime techniques to determine a parallel partitioning of compute kernels, map kernels to heterogeneous processor resources in the system, use predicted execution times of of kernel on each architecture to determine a schedule that minimizes the total execution time, and gather performance statistics while kernels are running to help make more informed scheduling decisions in the future. Once an application has been statically partitioned into control decisions and compute kernels, the application is executed sequentially, adding kernels to the scheduler without blocking as they are encountered in the normal execution flow of the program and only stopping at control decisions that require a result from a kernel that has not yet been computed. The runtime determines data dependencies between kernels, remaps memory resources to eliminate false dependencies, estimates the execution time of kernels on architectures available in the system, and determines a schedule that attempts to minimize the overall runtime of the application. Missing from this figure is the extension of the Harmony techniques to clusters of HMP systems and speculative execution of kernels which are discussed in depth in Section 7.

The principle technical advantages of the Harmony runtime techniques include

- *Minimal dependence on system configuration*: because binary implementations for compute kernels are provided for at least one, but possibly multiple, heterogeneous architectures, compute kernels can be mapped to different implementations based on what types of architectures are actually available in a system. This allows applications to run on any system for which every compute kernel has an implementation for at least one architecture in the system. Providing an implementation for all kernels for a general purpose architecture like x86 or IBM Power allows an application to run on the majority of platforms and take advantage of more exotic architectures if they are available.

- *No requirement for explicitly specified concurrency*: the addition of meta-data into the description of compute kernels allows for low overhead inference of concurrency among kernels at runtime. The programming and execution model advocated by Harmony meshes well with sequential and imperative programming languages like C and Fortran without the need to explicitly declare parallelism as with MPI and OpenMP.

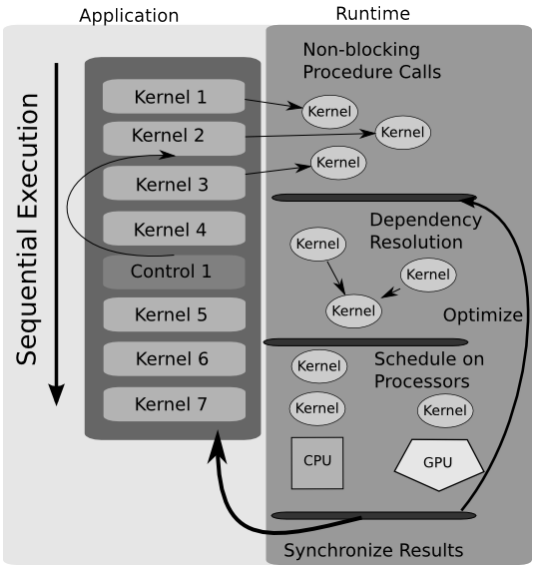


Figure 1. Harmony Execution Model

- *Runtime Self-Optimization*: the automatic determination of the suitability of a specific kernel to a specific architecture precludes the need for application developers to profile algorithms on different architectures and statically assign them to the most efficient architectures while simultaneously load balancing work on all available processors.

- *Low Overheads*: starting with low partitioning and scheduling overheads, Harmony continuously monitors the overheads of partitioning, scheduling, and mapping operations compared to kernel execution time. If the overheads are too high, it can fall back to less complex algorithms or combine kernels together to reduce balance overheads against the quality of decisions being made.

The remainder of the paper is organized as follows. The next section compares Harmony with related work to highlight its technical contributions to the field. An in depth discussion of the Harmony runtime techniques follows, beginning with the programming and execution model, followed by mapping and scheduling, performance monitoring, and ending with a discussion on speculation. Experimental evaluations are presented for sample HMP systems before the paper ends with conclusions and future work.

2. Related Work

The potential for dramatic speedups for high performance applications has drawn a considerable amount of attention to heterogeneous architectures, primarily focusing those like GPGPUs, IBM's Cell Broadband Engine, and FPGAs that have already strong market and industry support for other applications like computer graphics, game consoles, and hardware controllers respectively. Hagen et al. [21] have implemented a Euler equation solver to simulate the dynamics of ideal gases in multiple dimensions, showing speedups from 10-20x over an Intel 2.8Ghz Xeon on an NVIDIA 6800 Ultra for two different benchmarks. This also typical of cryptographic applications as demonstrated by Kaur et al. [22], who give an implementation of AES on a Xilinx Virtex 2 Pro that can sustain a throughput rate of 1.18 Gbps compared to a software implementation on a Pentium 4 that can sustain a throughput of only 245 Mbps [7]. Extensions to the IBM Cell can be observed in Minor et al.'s analysis of parallel ray casting which experiences a 36x speedup on Cell compared to a similarly clocked IBM G5. In this paper, we present three example applications showing significant performance improvements for dense matrix multiplication, ray tracing, and audio frequency analysis when run cooperatively on a system with an x86 processor and NVDIA G80 series GPU, showing speedups of up to 26.2, 99.1, and 12.24 compared to an x86 only implementation respectively. However, unlike the previously mentioned examples, runtime partitioning and scheduling handled by the Harmony runtime services allows our applications to run on a system with any combination of x86 processors and NVIDIA G80 GPUs without recompilation.

Eichenberger et al. [16] present several compilation techniques for inferring loop-level, SIMD style parallelism and partitioning instructions and data across the SPEs in a Cell processor. These techniques borrow concepts from OpenMP style pragmas to map kernels to SPEs and the PPE available on the Cell. These static, compile-time, techniques benefit from the ability to perform more sophisticated analysis without impacting performance, but are limited to inferring concurrency that is not data dependent and require recompilation to target different system configurations. Runtime techniques like those advocated by Harmony trade management overheads to leverage the extra information available when an application is being executed.

Our basic approach of determining the optimal granularity at which to infer parallelism is shown feasible in [14], which exhaustively searches the configuration space of thread-level and loop-level parallelism for MPI tasks on IBM Cell processors. The exhaustive search is made feasible by an over subscription of MPI tasks to SPEs available on the Cell fabric and the relatively small number of configurations of SPEs (8). Their method of moving from thread-level to loop-level parallelism based on the availability of one versus the other is similar to our method of combining or dividing kernels to trade concurrency for scheduling overheads. The incorporation of kernel dependency information in the form of meta-data and predictions about the execution time of kernels based on input data allows Harmony to a priori make more informed decisions about the quality and cost of creating schedules.

An important component of Harmony's runtime services is its specification of dependencies between kernels. This concept has parallels to instruction formats that explicitly name the hardware registers upon which they operate. The IBM ALF [23] runtime for IBM Cell also uses this abstraction as a method to infer concurrency between PPE and SPE kernels. We extend the abstraction beyond that presented in ALF with: (1) additions for memory page remapping for eliminating resource constrained false-dependencies, (2) the introduction of data-dependent control decisions enabling software branch prediction and speculative kernel execution, (3) multiple implementations of the same kernel for different processor architectures, and (4) extra information for estimating the runtime of compute kernels on specific processor architectures.

Other approaches to runtime of management of heterogeneity focus on the memory hierarchy, taking note of the fact that most HMP systems communicate through explicit DMA transfers rather than shared memory among processors. The Sequoia [15] programming language expresses the memory hierarchy in a HMP as a tree of distinct memory modules and use the concept of tasks to associate computation with a local address space and explicitly declare parallelism. The semantics of Sequoia force the programmer to explicitly manage vertical transfer of data up and down the memory hierarchy in a way that is portable across different configurations and runtime mapping is used to match a program's specification to the system architecture. Harmony considers only the highest level of the memory hierarchy that is accessible by all processors in the system and allows compute kernels to explicitly manage the memory modules local to a specific processor in order to maintain forward compatibility with new processors with potentially novel memory configurations.

The following sections of this paper provide overviews of and justifications for the Harmony runtime techniques. They focus on presenting high level abstractions and techniques that could be useful for managing a HMP system rather than those used in a specific implementation. For specific implementation details, we refer the reader to the experimental evaluations in Section 8.

3. Program Execution Model

This section describes the format of a compute kernel and control decision respectively; these are the only two primitives required by the Harmony programming model. A compute kernel consists of: (1) a list of memory locations that can be read from by the kernel, (2) a list of memory

locations that an be written to by the compute kernel, (3) binary implementations of the kernel for at least one but possibly many heterogeneous architectures, (4) potentially a user defined function to calculate the complexity of the kernel based on the input data to the kernel, and (5) the next kernel or control decision to be executed in the sequential flow of the application. Note that the list of memory locations to be read from can overlap with the list to be written to, and that these lists can be generated at any time during the execution of the program before the kernel is called – they can even be generated by the outputs of other kernels. Control decisions are like kernels in that they require a list of memory locations that can be read from, but differ in that they only have a single binary implementation for the host processor and have at least two possibilities for the next kernel to be executed. In this programming model, kernels are directly analogous to arithmetic or memory instructions and control decisions are directly analogous to branch instructions.

It should be noted that the specification for a compute kernel requires all implementations of the same kernel for all heterogeneous architectures to be identical. In this context, identical means that for every possible combination of input data, each kernel will produce the same output data. This constraint potentially complicates the verification of heterogeneous applications where different algorithms are used for each architecture or where different architectures provide slightly different implementations of certain standards. For example, some architectures only support single-precision floating point operations compared to the double-precision supported by most general purpose processors. This could lead to situations where results could vary from one run of an application to another depending on which implementation was used. We acknowledge that this is a problem that should be taken into account when designing any application for a HMP system, regardless of whether kernels are mapped to architectures at runtime or statically by hand.

The execution of an application implementing this model is nearly identical to that of a sequential application executing on a modern super-scalar, out-of-order (OOO), processor with instructions replaced by kernels and functional units replaced by processors with potentially heterogeneous architectures. The caveats being that instructions might be able to execute on more than one functional unit with different performance characteristics, the performance of specific instructions might be very dependent on their input operands, and that any number of applications can simultaneously share the processor core without affecting the correctness of any other. Additionally, the difference in time scales for instruction compared to kernel execution allows for much more complicated scheduling algorithms and wider dependency checking windows to be evaluated in software by a runtime than in hardware for a processor.

The application executes as though following a single thread of execution. Kernels are added to the runtime as they are encountered in the normal sequential flow of the application. As kernels are being inserted into the runtime, the list of memory addresses that they read from and write is calculated and used to infer concurrency between kernels. This process is explained in detail in Section 4. When the thread of execution adds more kernels to the runtime than can fit in the current scheduling window, or the thread encounters a control decision, it passes control to the runtime which begins launching kernels on available processors until there is enough information to resolve the control decision or the scheduling window has enough space for another kernel. The factors used to make scheduling decisions are detailed in Section 5.

The explicit specification of the memory locations that a kernel will manipulate effectively creates a division between the global memory space of the program and the local memory space of a kernel. A kernel is able to allocate as much memory as it needs in its local address space for local computations, but this memory cannot affect the state of any other kernel that is executing – this local memory could even be on some external memory module accessible only by a specific processor in the system. Once a kernel has finished executing, all of its local memory that it used for internal computation must be freed and only the locations that it explicitly declared as inputs or outputs are allowed to be persistent. Furthermore, only the location specified as outputs can be modified at any point during the kernel's execution: Inputs cannot be written to, even if they are restored to their original state before the kernel completes. This allows for the runtime to dynamically determine whether or not the program correctness would be violated by executing two kernels in parallel – the isolation of their local memory spaces ensures that the execution of one will not affect any of the results produced by the other as long as they do not share any memory locations in the global space of the application.

The sequential nature of programs written using these primitives and this execution model lends itself to traditional imperative programming languages like C/Java/Fortran where kernels have direct analogs in terms of syntactical constructs like function/procedure calls or compiler analysis primitives like super-blocks . Simply adding lists of input and output data to C functions, calling functions by handles that could possibly refer to multiple implementations of the same function, and adding synchronization points before reading from a variable that could be modified by a function are enough to integrate this programming model with regular C programs. More complicated extensions would be needed to support speculation across control decisions, but these could be inferred by a compiler. Future work will look into the possibilities of extending traditional imperative language compilers to target this runtime supported execution model without modification to the language being compiled.

4. Dynamic Concurrency Inference

Once an application has been statically partitioned into a sequence of compute kernels like the one shown in Figure 1, and the memory locations that each kernel will read and write to have been determined, it is possible to determine the data dependencies between kernels and create a parallel schedule. It has been a prevalent problem in compilers to determine data dependencies between program sections from static program analysis due to runtime dependent program behavior. Syntactical constructs like pointer arithmetic, dynamic memory allocation, and variable length iterations over arrays make the memory locations that a program segment manipulates vary between successive executions of the application. Harmony addresses this problem by forcing all kernels to provide functions for calculating their data dependencies before they are partitioned by the runtime.

Once the read and write memory addresses have been determined for a kernel, it can be added to a dependency graph that expresses the possible concurrency among all kernels in the graph. When a kernel is added to the runtime by the main application thread, it has to check all of its input memory locations with the output memory locations of all other kernels that have not finished executing. If there are any matches found, this represents a true dependency from the current kernel on the previous kernel. In this case, the previous kernel must complete execution before this kernel can being. It must then check its output memory addresses against the input memory addresses of any previously added kernel. These represent anti dependencies from the current kernel to the previous kernel. Finally, the runtime must check the output memory addresses of the current kernel to the output memory addresses of all previously added kernels. This checks for output dependencies from the current kernel to the previous kernel with matching output memory addresses. Kernels that complete execution are removed from the dependency graph and remove all dependencies associated with them. All kernels that depend on no other kernels are eligible for immediate execution, and the degree of parallelism in an application at any point is determined by the number of kernels in the dependency graph with all of their dependencies satisfied.

The complexity of this operation is quite reasonable for even relatively large numbers of kernels. Every time a new kernel is added, three checks are needed against every other kernel that has not yet completed. This operation requires three checks for the second kernel, six checks for the third kernel, and, in general, $3 * (N - 1)$ checks for the Nth kernel. Taken as a series from one to a maximum dependency graph size of N, **completely filling the dependency graph will require** $3/2 * (N^2 - N)$ **checks**. Doing this evaluation in software rather than hardware allows for a potentially larger dependency graph, and consequently a larger scheduling window, without excessive overheads. In a real implementation, it is possible to compare the computation required to add a new kernel to the dependency graph to the potential amount of time that could be saved by executing the kernel in parallel to decide whether it is worthwhile to add the kernel to the dependency graph.

It should be noted that only true dependencies are fundamental limitations on the parallelism of an application. Anti dependencies and output dependencies exist only because two kernels share the same underlying memory addresses as scratch memory for computation. Like register renaming in OOO processors, these memory locations can be remapped before a kernel executes to eliminate these dependencies. This remapping process increases the overall memory footprint of the application while potentially allowing more kernels to execute in parallel. Increasing in memory footprint can have adverse affects in terms of exhausting cache and memory bandwidth resources in the system. The specific application and system configuration will determine whether the potential for increased parallelism offered by this technique is worth the overheads. A first order approximation can be made by the scheduler to determine if allow a specific kernel with an anti or output dependency to execute in parallel would result in an overall faster scheduler and only remapping memory if the speedup was beyond a threshold. A better analysis would require estimating the performance overheads of increasing the memory footprint and compare them to the reduction in execution time due to increased parallelism.

Parallelism between kernels can be increased even further by dividing kernel execution into two distinct phases. In the first phase, the kernel reads from its input memory addresses and then signals to the runtime that it will not longer use them. After this point, the kernel is allowed only to write to its output memory addresses. This technique would allow the relaxation of anti dependencies that would only have to wait until a kernel completed the first phase of execution to begin. This technique also has benefits for speculatively executed kernels which can begin execution up to the point when they need to write their results back to their output memory locations and then wait on the runtime for permission to continue. This process is described in more detail in Section 7.

It is possible that even the function used to determine read and write memory locations for a kernel might depend on results generated by a previous kernel. For these cases, it is possible to add the kernel to the scheduler without adding it to the dependency graph and wait until its memory addresses become available before adding any new kernels. If only the outputs of the kernel are available, memory remapping can be used to ensure that no kernel that is added to the runtime at a later time has an anti dependency on that kernel, and the thread of execution of the main application can continue. The converse is not true for input memory addresses because the unknown output memory addresses could produce a true dependency that could not be eliminated by memory remapping.

5. Scheduling

The first set of runtime services outlined in Section 4 allow Harmony to determine parallelism between kernels in a sequential application and express it in the form of a dependency graph. This, along with the number of type of heterogeneous processors in the system and the implementations of each kernel for potentially different architectures is enough information to construct a simplistic mapping from kernels with satisfied dependencies to available processors in the system. A simple heuristic to accomplish this without considering the execution times of individual kernels or their efficiency when running on a specific architecture would be to simply assign kernels with all of their dependencies satisfied to processors for which there exists a binary implementation. We refer to this technique as a greedy scheduler because it attempts to start kernels as fast as possible without considering more than the minimum amount of required information. Results from Section 8 show that the greedy scheduler has extremely low overheads and works well for embarrassingly parallel applications where the execution time of a kernel on a specific architecture is relatively constant across all architectures or the runtime of the application is much greater than that of a specific kernel.

This scheduler runs into problems when it is given applications with kernels whose execution times vary dramatically with the architecture they are being run on – the greedy scheduler is just as likely to schedule a kernel on a very inefficient architecture as an efficient one. This is actually the typical case for many implementations including all of the examples given in Section 8, where the difference in performance was 10x or greater for all applications. A simple addition to the greedy scheduler to help with this case would be to rank the implementations for each architecture by efficiency. This ranking could be collected from runtime statistics or specified in the kernel meta-data when it was compiled. Making this simple addition would allow the scheduler to try to pair a kernel with the architecture it runs the fastest on. If the best architecture was not in the system, it could move on to the next best, and if the best architecture was busy, it could move on to another ready kernel and attempt to match it to its best architecture. A fundamental observation about this scheduler is that it attempts to ensure that all processors in the system are always executing a kernel which, again, works well for applications with few dependencies and similar execution times for all kernels.

The modified greedy scheduler performs poorly in situations where it assigns kernels to inefficient cores in an attempt to reduce the idle time of that core. In cases where the difference in execution time for a kernel on two architectures is significant, it might be more advantageous to leave a less efficient processor idle and wait for the more efficient processor to finish working. In a somewhat contrived example where there is only one type of kernel, two processor types, one with 20x the performance of the other, and no dependencies between kernels, it makes sense to schedule 20 kernels on the first processor before scheduling a single kernel on the less efficient processor. A second addition to the greedy scheduler to resolve this problem would be to allow kernels to be queued up behind specific processors in the system even if they are busy. If there was a mechanism to calculate the runtime of a kernel on a specific processor, it would be possible to determine the execution time of all kernels in all queues as a sum on all of the projected execution times of the kernels in the longest queue. Adding new kernels to this scheduler could be accomplished by testing each queue and adding the kernel to the one that minimized the overall execution time of the system.

The scheduling community has produced a wealth of knowledge for algorithms for resource constrained scheduling algorithms, any of which could be applied to this problem with slight modifications to account for the ability to execute certain kernels on different processor architectures with different performance characteristics. However, while techniques like those presented by Hannig et al. [17] can create globally optimal schedules by taking into account complexities introduced by dependencies between kernels, projected kernel execution times on specific processors, and even speculatively launching kernels past control decisions that have not yet been resolved for a given scheduling window, they were designed for static analysis and potentially have high overheads for finding an optimal solution.

We address the overheads introduced by dependency graph generation and resource constrained scheduling with three methods that trade scheduling overheads for quality. The first, and most obvious choice, is to reduce the complexity of scheduling and dependency resolution is to restrict the maximum number of kernels that can be managed by the runtime at one time. This reduces the dependency resolution complexity quadratically and also the complexity of the scheduler by reducing the amount of kernels that it has to schedule at a single time. The second is to move to a less complicated scheduling algorithm: it is possible to implement any number of scheduling algorithms and switch between them based on the dynamic behavior of the program in order to keep overheads to an acceptable level. The third method is to change the granularity of scheduling decisions of kernels based on their estimated execution times. This techniques is based on the observation that certain kernels will inherently require more computation than others and that the overall performance of the system will be most significantly impacted by scheduling decisions made about these kernels. When a kernel is added to the dependency graph, if its estimated execution time is less than a threshold, it can be combined with the previous or next kernel encountered in program order. When this happens, the two kernels are assumed to be executed in program order and the memory inputs and outputs of the new kernel are taken as the union of the inputs and outputs of the two smaller kernels. This effectively reduces the total number of kernels in the dependency graph, allowing more kernels to be scheduled with the same overheads.

6. Performance Monitoring and Optimization

Many of the scheduling decisions detailed in the previous section rely on the ability to predict the runtime of specific compute kernels when paired with specific heterogeneous architectures before it actually executes. Harmony addresses this problem with a combination of performance statistics gathered as kernels are executing, regression models constructed over successive runs of a kernel, and meta-data supplied statically. The most simplistic technique for estimating kernel execution time is to keep a separate running average for each kernel on each architecture – every time a kernel executes on a specific architecture, the runtime records the latency, computes a running average over all such measurements, and uses this average to predict the latency of any future executions of the same kernel. This model works well if the latency of a kernel on a specific architecture is constant for all executions of that kernel, but can produce wildly inaccurate results if the latency is data-dependent.

Take for example the case of a kernel that performs a dot-product between two arrays. The computation required to compute the dot-product is largely dependent on the size of the arrays being processed. Examples like this lead to situations where a more efficient architecture might actually be predicted to be slower than a less efficient architecture if the more efficient architecture was given a harder problem. It also creates problems for complicated schedulers that require accurate projections of kernel latencies to create effective schedules. We attempt to solve this problem by adding meta-data to kernels when they are compiled or written to indicate which input variables affect the complexity of a kernel. During kernel execution, the values of these variables can be recorded along with the latency of the kernel and a multivariate regression can be used to fit a curve to the measured values. This method allows for a relatively accurate model of the latency of a kernel on a specific architecture to be built over time, where accuracy can be traded off against runtime overheads by moving from linear multivariate regressions to higher order polynomials and visa versa. However, it runs into problems when attempting to evaluate the performance of kernels whose behavior is not well characterized by a few input variables. Examples of this type of kernel might be iterative methods for PDE solvers where the time to convergence might change dramatically and non-uniformly with small changes in the input data.

It is possible to improve the results obtained from this method by augmenting them with user supplied estimates of the complexity of a kernel. The complexity of certain algorithms such as matrix multiplication, Fouier Transforms, various sorting algorithms, and many others have well known formulations that depend on only a few variables. It would be fairly simple to add an interface for the user to specify a function that would calculate the complexity of a kernel based on a subset of its inputs. To account for differing speeds of processors that support the same architectures (differing clock rates between two implementations of the same processor for example), the running average of a kernel latency could be used as an offset in the estimate produced by the user defined function. The inclusion of this function for each kernel would preclude the need to perform a regression after each kernel finished executing and would significantly reduced the overheads incurred by performance monitoring.

It would also be possible to increase the reliability of these regression models over time by persistently keeping track of latency measurements for kernels across successive runs of the application. Assuming that the system configuration does not change from one execution of an application to another, the regression models obtained from a complete execution of an application could be saved to non-volatile storage by the runtime and loaded if the same program was ever executed again. This would allow statistically sound models of the latency characteristics of a kernel to be built up over time, or even benchmarked with test values as in [24] to create initial performance estimates before an application was run with real data. Similarly, it would be possible to use idle processors in the system to launch multiple copies of the same kernel in parallel on different architectures to obtain more latency measurements that could be used to make more accurate scheduling decisions the next time that the kernel is executed. Additional uses for this type of speculative execution of kernels is covered in the next section.

7. Speculation and Recursive Application

Speculation has been examined with great success in the context of low level program-instruction analysis and the similarities between the Harmony execution model and that of programs composed of a sequence of instructions enables the direct application of many of these techniques to HMP systems. In this paper, we focus on two classes of speculation that we believe show the greatest potential for performance improvements: (1) speculation across control decisions, and (2) speculation as to the most efficient architecture upon which to execute a kernel. At a very high level, speculation across control decisions is motivated by the observation that the sequential thread of execution of a Harmony application has to stop at control decisions that use input data that has yet to be produced by a previously encountered kernel. Similarly, speculation across available architectures is motivated by the difficulty to predict the efficiency of some kernels on different architectures.

In the case of speculation across control decisions, the Harmony runtime requires knowledge of the entire set of compute kernels and control decisions that compose an application. Rather than just executing the application sequentially and breaking into the runtime at set synchronization points and kernel calls, the possibility of moving down multiple execution paths and having to backtrack in case

of a mis-predicted control decision necessitates the entire sequence of compute kernels and control decisions to be internalized by the runtime. This can be accomplished by scanning through an application before it is executed and loading all kernels and control decisions into internal data structures. For very large programs, a kernel-cache (similar to instruction-caches in virtual machine monitors) can be created in runtime memory that refers to the application binary in the case of a cache miss thereby reducing the amount of information that needs to be stored as an application is running.

Once the runtime has internalized the sequential structure of an application in terms of kernels and control decisions, it can add kernels to the dependency graph as they are encountered. When it reaches a control decision that has not yet been resolved and the dependency graph is not yet full, it can guess at the outcome of the control decision and continue adding kernels from one of the possible execution paths. Any kernels added from this speculated path must be prevented from modifying the state of the system until the runtime has determined that the control decision was correct. This can be accomplished by separating the start and finish stages of a kernel or remapping the output memory addresses of the speculated kernels as detailed in Section 4.

When the control decision is actually resolved, the runtime must check if the predicted outcome of the decision matches the real value. If it does, then the program can continue without interference. If it does not match, then the runtime must clear the results generated by any kernel that was launched as a result of making the wrong decision, remove them from the dependency graph and scheduler, and return the flow of the application to the correct execution path. Methods for estimating the outcome of a control decision can be borrowed directly from branch predictors for modern multiprocessors like bimodal predictors, global history based predictors, and combinations of the two [25]. An interesting future direction for research would be to examine more complicated predictors that could be implemented in software with much larger time-scales than are required for hardware based solutions.

For the case of speculation across available architectures, it is sometimes not possible to determine which architecture is best suited for a specific kernel. This could happen because there is not yet enough information to create an accurate model of the performance characteristics of a kernel, or if the best mapping varies between two or more different architectures based on input data values in a way that is difficult to predict using a multivariate regression. For these cases, it might make sense to launch the same kernel multiple times on different processors and take the results from which ever one finishes first. This type of speculation is easy to accommodate in that it only requires remapping the output memory of the two speculatively launched kernels. The runtime would keep the outputs of the kernel that finishes first and discard the results from the higher-latency kernels.

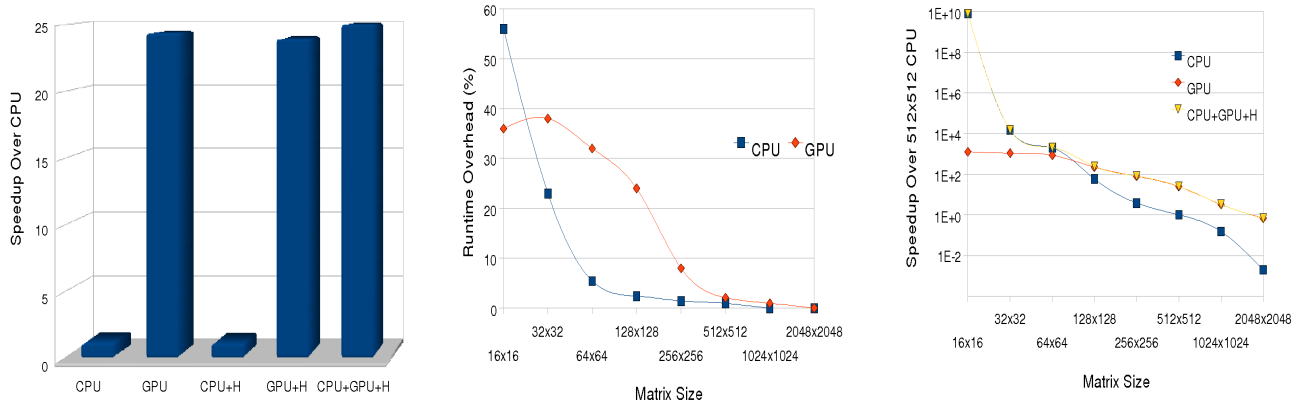
7.1. Hierarchical Runtime Scheduling

Though the majority of this paper has focused on the application of the Harmony execution model and runtime optimizations to single HMP machines, nearly all of these techniques can be applied recursively to hierarchically schedule kernels on a cluster of HMP machines, or even clusters of clusters. In such a system, a single master node begins sequential execution of an application. As kernels are encountered, they are packed together into higher level kernels as described in Section 5 and added to the runtime. Once enough kernels have been combined based on their estimated aggregate latency, they are added to the dependency graph. The difference from the single node case occurs in the scheduling stage where instead of mapping a kernel to a specific architecture, it is mapped to a specific HMP machine in the cluster. The aggregated kernel is sent as a message along with its input data to the HMP machine, which unpacks the lower-level kernels, adds them to its local dependency graph, schedules them on local processors, and sends the results back to the master node when they have completed.

This model adds a few slight complications in terms of communication overheads for sending data to a remote node and ensuring that a remote node has the correct types of heterogeneous processors to be able to execute every low-level kernel that it receives. These can be taken care of by adding a model for communication overheads into the computation of the execution time of a high-level kernel on a remote node. A simple check is needed to make sure that remote nodes have all of the required architectures before sending a high-level kernel to them. **The recursive application of the Harmony execution model effectively transforms sequentially specified applications into distributed diffusing computations**.

8. Experimental Evaluation

To evaluate both the feasibility and benefits of applying the Harmony execution model and runtime techniques to real high performance and multimedia applications, we provide sample implementations of dense matrix multiplication, ray tracing, and audio frequency analysis using an **AMD Athlon64 2.4Ghz CPU** and an **NVIDIA 8800GT GPU** cooperatively. The test configuration had **2GB DDR2 800 RAM, Linux 2.6.22-14, GCC 4.1.3**, and was used for generating all of the results in this paper. For this configuration we show that the runtime overheads from dependency graph generation and queue based scheduling are less than 5% for 64x64 matrix multiplication on the CPU, less than 4% for rendering 32x32 pixel blocks with the ray tracer, and less than 1% analyzing 65536 segments of an audio waveform. It should be noted that not all of the techniques described above are used for these applications for the sake of simplicity, and that these results are only meant to pro-



(a) 512x512 performance on various platforms (b) Runtime overheads (c) Performance scaling with matrix size

Figure 2. Matrix Multiplication

vide sample points for what is possible when applying Harmony to specific applications rather than highly optimized or comprehensive implementations. Unless specified otherwise, each implementation of the Harmony runtime used a dependency graph a maximum of 128 entries, a queue based scheduler as described in Section 5, and pthreads to launch kernels in parallel.

8.1. Dense Matrix Multiplication

The first example application is a dense matrix multiplier that multiplies two variable sized matrices using a panel-dot approach where each value in the result matrix is computed during one kernel call by computing the dot product of a row in the first matrix and a column in the second matrix. In this example, there are no inter-kernel dependencies, there is only one type of kernel, and the complexity of each kernel is constant for the duration of the application. Figure 2a, compares the performance of a single run of the application using 512x512 matrices run entirely on the CPU, entirely on the GPU, entirely on the GPU using the Harmony runtime to schedule kernels, entirely on the CPU using the Harmony runtime, and, finally, using both the GPU and CPU using Harmony to schedule kernels. It is worthwhile to note that **all of the implementations using Harmony use the same binary** and the only modification is to change the number and type of processors that Harmony thinks are in the system.

Compared to the reference implementations, the examples using Harmony incur less than a 5% overhead. Figure 2b shows the average Harmony overheads compared to matrix size, while Figure 2c shows the performance scaling of the three configurations (CPU, GPU, CPU+GPU+Harmony) with matrix size. Note that as the size of the matrix increases, the GPU performs relatively better compared to the CPU. This is due to the overheads required to move data to and from the GPU which can be amortized for larger data sizes. Figure 2c shows that the queue based scheduler used by Harmony is able to observe this behavior and allocate more work to the CPU for small matrices and more to the GPU for large matrices, causing the performance of the combined system to follow that of the best performing of the two standalone programs. This behavior would be difficult to exploit by statically partitioning some of the work to the CPU and the rest to the GPU.

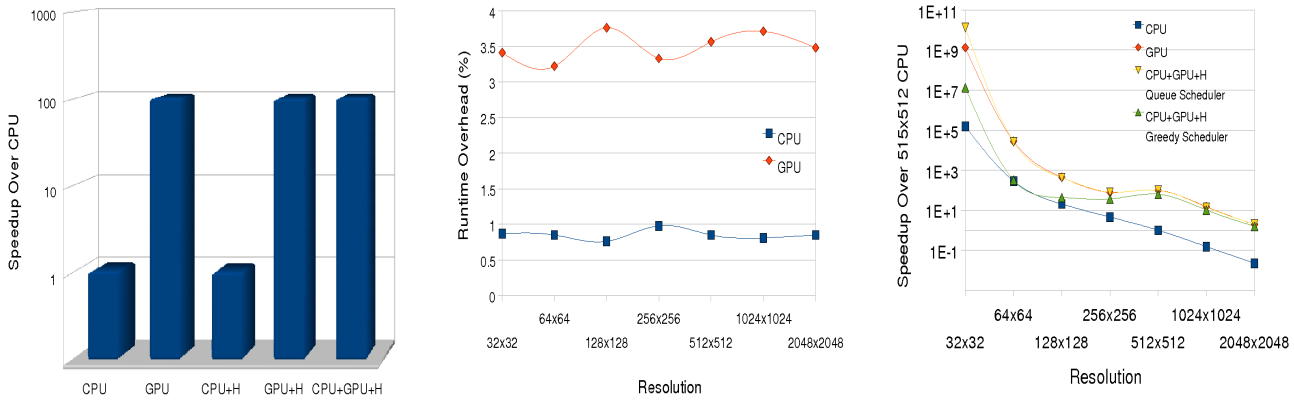
8.2. Ray Tracing

The second example application is a parallel ray tracer that renders a scene by following the paths from every pixel in a camera to objects and light sources within a scene, coloring each pixel based on the color of the light or object in the scene that it intersected. This specific example moves the camera around the scene, rendering the entire scene in terms of 32x32 blocks of pixels before moving the camera to another location and repeating the process. Here, each of the 32x32 blocks are given GPU and CPU implementations and have no inter-kernel dependencies. However, another kernel is added to move the camera after a scene has been rendered which is only given a CPU implementation. The structure of the program is shown in pseudo below:

```
while(true)
    for( all 32x32 blocks )
        renderBlock();
    moveCamera();
```

All of the render operations shown in the for loop can execute in parallel. However, there is a loop carry dependency that prevents the next set of render operations from beginning until the camera has been moved.

Again, the performance of this application is compared to that of a standalone CPU and GPU version and is depicted in Figure 3a along with the Harmony overheads compared



(a) Peformance for 512x512 pixels

(b) Runtime overheads

(c) Performance scaling with resolution

Figure 3. Ray Tracing

to image resolution shown in Figure 3b. For this operation, the GPU is much more efficient than the CPU, experiencing a 99.2 times speedup. Such a dramatic difference in performance between the two versions of the kernel benefits greatly from a more intelligent scheduler. As figure 3c demonstrates, simply assigning kernels to cores as they become available as with the greedy scheduler performs as much as 80 times worse than the queue scheduler.

8.3. Audio Frequency Analysis

The final example application performs analysis and noise removal on an audio signal. The application first takes an arbitrary number of samples of an audio signal. The samples are broken into partially overlapping segments of samples which are then passed through a Fourier transform to determine their frequency components. The frequency content of the signal is analyzed and statistics are collected. The frequency representation of the signal is then multiplied with a filter function, and an inverse Fourier transform is used to return the signal to a time domain representation. Finally, the signal is reassembled using a windowing function. The application is written with nine distinct kernels to process a segment of the audio signal, many of which have data dependencies. Additionally, there is a loop-carry dependency between generating successive segments of the output signal.

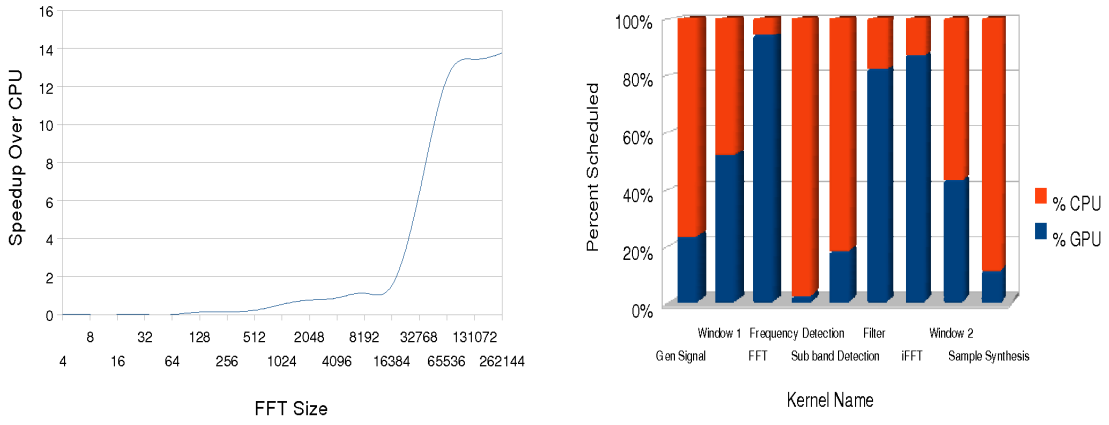
The processing time for this application is dominated by the 65536 point FFTs, for which there is a very efficient implementation available for the GPU. The graph in Figure 4a reflects this behavior, where the GPU implementation is nearly 12x faster than the CPU using FFTW3 [24]. However, the GPU performs relatively poorly for the signal analysis component which is reflected in the comparison of the percentage of time a kernel was scheduled on the GPU versus the CPU shown in Figure 4b. Even though the CPU was so much faster than the GPU for this kernel, its computation time was insignificant compared to that of the FFT and thus did not influence the overall results.

9. Conclusions and Future Work

Harmony is comprised of a set of runtime techniques that collectively reduce the complexity of designing applications for HMP systems while maintaining a high level of performance that is competitive with hand-built applications. Harmony uses meta-data to augment a sequential programming model, enabling concurrency to be inferred dynamically at runtime. In addition, Harmony presents techniques for mapping compute kernels to available processor architectures and creating a schedule that minimizes the overall execution time of an application. These techniques are enhanced by extensions for speculation, performance monitoring and optimization, and false dependency resolution.

The ability to dynamically bind kernels to implementations for specific heterogeneous architectures enables applications based on the Harmony execution model to be portable across systems with minimal hardware configurations without sacrificing performance on higher end systems. The statistical regression-based model used by Harmony allows the runtime to dynamically determine the most efficient architecture for executing a specific kernel even if its computational complexity is data-dependent. As a result, Harmony is able to address problems where the best architecture for a specific part of a program depends on the data that is being processed.

This paper uses the Harmony runtime techniques to implement three applications on HMP systems without sacrificing performance or portability: (1) parallel multiplication of dense matrices, (2) real time ray tracing, and (3) audio frequency analysis. The Harmony runtime techniques are integrated with these applications in a non-intrusive manner that does not require the restructuring of the the sequential, standalone, versions of the application and only requires the



(a) FFT scaling on GPU vs CPU

(b) Time spent on GPU and CPU

Figure 4. Audio Analysis

replacement of regular function calls with calls to kernels that are registered with the runtime.

In the future, we plan to pursue several interesting extensions to the Harmony runtime techniques. We would like to develop a complete implementation of Harmony using all of the techniques described in Sections 3-7 (particularly speculation and memory remapping were lacking from our experimental implementation) and evaluate the performance of specific techniques compared to others. We would also like to apply the Harmony execution model to other example applications from different sectors like computational finance, medical imaging, and others. Similarly, we would like to port our existing implementations of Harmony to support new types of accelerators like IBM Cell, FPGAs, or Intel IXPs. Another interesting topic would be to integrate the Harmony runtime services with an operating system or hypervisor to allow for runtime management of several applications concurrently.

There is a large amount of effort behind integrating compilation tools for different architectures. It would be very beneficial to be able to use one compilation tool to generate binaries for several different architectures from the same source code instead of using a separate tool for each architecture and then linking the results together at runtime. Along the same lines, we would like to explore techniques for dynamic recompilation of existing binaries to support new architectures, and evaluate compiler techniques that allow meta-data like memory accesses or computational complexity to be inferred from standard programming languages like C/Java/Fortran.

References

[1] S. Wheat, "Intel outlines its hpc strategy," *hpcwire*, February 2007.

[2] E. Elsen, M. Houston, V. Vishal, E. Darve, P. Hanrahan, and V. Pande, "N-body simulation on gpus," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, (New York, NY, USA), p. 188, ACM, 2006.

[3] J. Yang, Y. Wang, and Y. Chen, "Gpu accelerated molecular dynamics simulation of thermal conductivities," *J. Comput. Phys.*, vol. 221, no. 2, pp. 799–804, 2007.

[4] J. Michalakes and M. Vachharajani, "Gpu acceleration of numerical weather prediction," *Workshop on Large Scale Parallel Processing, IPDPS*, 2008.

[5] V. T. Barry Minor, Gordon Fossum, "Terrain rendering engine (tre)," *IBM Technical Report*, 2005.

[6] B. D'Amora, K. Magerlein, A. Binstock, A. Nanda, and B. Yee, "High-performance server systems and the next generation of online games," *IBM Syst. J.*, vol. 45, no. 1, pp. 103–118, 2006.

[7] P. Parikh, Chirag; Patel, "Performance evaluation of aes algorithm on various development platforms," *Consumer Electronics, 2007. ISCE 2007. IEEE International Symposium on*, pp. 1–6, 20-23 June 2007.

[8] D. R. Horn, J. Sugerman, M. Houston, and P. Hanrahan, "Interactive k-d tree gpu raytracing," in *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, (New York, NY, USA), pp. 167–174, ACM, 2007.

[9] K. Koch, "Roadrunner system overview," tech. rep., LANL, 2007.

[10] J. A. Turner, "Roadrunner applications team: Cell and hybrid results to date," tech. rep., LANL, 2007.

[11] J. Kahle, "The cell processor architecture," *Microarchitecture, 2005. MICRO-38. Proceedings. 38th Annual IEEE/ACM International Symposium on*, pp. 3–3, 12-16 Nov. 2005.

[12] R. Govind, S.; Govindarajan, "Performance modeling and architecture exploration of network processors," *Quantitative Evaluation of Systems, 2005. Second International Conference on the*, pp. 189–198, 19-22 Sept. 2005.

[13] M. Meissner, S. Grimm, W. Strasser, J. Packer, and D. Latimer, "Parallel volume rendering on a single-chip simd architecture," in *PVG '01: Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphics*, (Piscataway, NJ, USA), pp. 107–113, IEEE Press, 2001.

[14] F. Blagojevic, D. S. Nikolopoulos, A. Stamatakis, C. D. Antonopoulos, and M. Curtis-Maury, "Runtime scheduling of dynamic parallelism on accelerator-based multi-core systems," *Parallel Comput.*, vol. 33, no. 10-11, pp. 700–719, 2007.

[15] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: programming the memory hierarchy," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, (New York, NY, USA), p. 83, ACM, 2006.

[16] A. E. Eichenberger, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind, "Optimizing compiler for the cell processor," in *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, (Washington, DC, USA), pp. 161–172, IEEE Computer Society, 2005.

[17] J. Hannig, F.; Teich, "Resource constrained and speculative scheduling of an algorithm class with run-time dependent conditionals," *Application-Specific Systems, Architectures and Processors, 2004. Proceedings. 15th IEEE International Conference on*, pp. 17–27, 27-29 Sept. 2004.

[18] S. Larsen and S. Amarasinghe, "Exploiting superword level parallelism with multimedia instruction sets," *SIGPLAN Not.*, vol. 35, no. 5, pp. 145–156, 2000.

[19] C. B. Software, "Vast-f/altivec: Automatic fortran vectorizer for powerpc vector unit," tech. rep., Crecent Bay Software, 2001.

[20] J. Shin, M. Hall, and J. Chame, "Superword-level parallelism in the presence of control flow," in *CGO '05: Proceedings of the international symposium on Code generation and optimization*, (Washington, DC, USA), pp. 165–175, IEEE Computer Society, 2005.

[21] T. R. Hagen, K.-A. Lie, and J. R. Natvig, "Solving the euler equations on graphics processing units," in *V. Alexandrov, G. van Albada, P. Sloot, and J. Dongarra, editors, Computational Science ICCS 2006*, (Springer, USA), pp. 220– 227, 2006.

[22] R. Kaur, Swinder; Vig, "Efficient implementation of aes algorithm in fpga device," *Conference on Computational Intelligence and Multimedia Applications, 2007. International Conference on*, vol. 2, pp. 179–187, 13-15 Dec. 2007.

[23] I. Research, "Alf programming guide api v30," *IBM Technical Report*, 2005.

[24] M. Frigo and S. G. Johnson, "FFTW: An adaptive software architecture for the FFT," in *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, vol. 3, (Seattle, WA), pp. 1381–1384, May 1998.

[25] K. Thangarajan, W. Mahmoud, E. Ososanya, and P. Balaji, "Survey of branch prediction schemes for pipelined processors," *System Theory, 2002. Proceedings of the Thirty-Fourth Southeastern Symposium on*, pp. 324–328, 2002.