A Scalable Performance Prediction Heuristic for Implementation Planning on Heterogeneous Systems

John R. Wernsing, Dr. Greg Stitt
Department of Electrical & Computer Engineering
University of Florida
Gainesville, FL 32611
wernsing@ufl.edu, gstitt@ece.ufl.edu

***Abstract*—Despite speedups of 10x to 1000x, effective usage of multi-core and heterogeneous systems has largely been limited to experts due to increased application design complexity resulting from the requirement for significantly different algorithms for different device types and amounts. Compiler and high-level synthesis research has attempted to address this problem but is fundamentally limited to the algorithm specified by the high-level code. Thus, future compilers will need to choose from numerous implementations/algorithms for a given function when optimizing for a multi-core heterogeneous system. This emerging problem, which we refer to as the implementation planning problem, requires compilers and similar tools to rapidly determine performance of a particular implementation on different devices for all possible input parameters. To help solve the implementation planning problem, we introduce a heuristic that repeatedly selects statistically significant input values, measures actual execution time, and then statistically analyzes the results to predict the execution time for all inputs within requested accuracy and confidence levels. We evaluated the heuristic using twelve examples on three different platforms with up to 16 microprocessor cores and a field-programmable gate array, achieving an average prediction error of 6.2% and a root-mean-squared error of 7.4%, which required an average of only 463 samples and 51 seconds to complete.**

***Keywords*—elastic computing; heterogeneous computing; implementation planning; performance prediction**

I. INTRODUCTION

Embedded architectures have started on a clear trend towards increased parallelism, as evidenced by the widespread usage of multi-core microprocessors, and also increased heterogeneity, with graphics processing units (GPUs) and field-programmable-gate-arrays (FPGAs) being increasingly used to achieve speedup ranging from 10x to more than 1000x [5][11][27]. Despite significant advantages, usage of multi-core heterogeneous systems for real-time multimedia and other embedded domains has been limited due to increased design complexity compared to software design.

To enable more widespread usage, numerous studies on tools for hardware/software co-design, compilation, high-level synthesis, and hardware/software partitioning (which for simplicity we collectively refer to as *compilation tools*) have focused on optimizing a single high-level implementation (e.g., a multimedia kernel) for heterogeneous systems [3][12][15][24][28]. Although such previous work has reduced design complexity, a fundamental problem that has limited the effectiveness of existing compilation tools is that different devices in heterogeneous systems often require significantly different algorithms for efficient execution [9][24]. For example, a software sorting implementation would likely use a Quick Sort algorithm, whereas an FPGA implementation would likely use a Merge Sort or Bitonic Sort algorithm. If a compiler optimized a Quick Sort-based implementation for an FPGA, the performance could be orders of magnitude slower than a Bitonic Sort implementation [24]. This problem is not limited to heterogeneous devices; multi-core devices with different numbers of cores may also exhibit widely differing performances for different implementations [21]. The problem is further complicated by a common dependence on input parameters, where no single implementation is optimal for all inputs. For example, the optimal algorithm for sorting 10 elements on a particular device is likely different than the optimal algorithm for sorting 1,000,000 elements.

To effectively optimize applications for multi-core heterogeneous systems, future compilation tools will need to rapidly select the best implementation for a given function call. Due to potentially large numbers of function calls, implementations, and input possibilities, it is impractical for a compiler to evaluate all implementation possibilities at compile-time. Therefore, compilation tools must *plan* ahead what implementations to use for different resource combinations and input parameters, which we define as the *implementation planning* problem. Compilers can then use planning results to quickly select an efficient implementation for each function call at compile-time, which is critical for meeting constraints in real-time multimedia systems.

The main challenge of implementation planning is dealing with scalability issues that result from numerous combinations of implementations, resources, and input parameters. In this paper, we introduce a heuristic that repeatedly selects statistically significant invocation parameters, measures actual execution time, and then statistically analyzes results to predict execution time for all combinations of input parameters. The goal of the heuristic is to minimize implementation planning execution time by executing as few input combinations as possible, while attempting to meet specified accuracy and confidence levels. For twelve case studies, the presented heuristic achieved an average error of 6.2% and a root-mean-squared error of 7.4% on three significantly different systems that included 16 microprocessor cores and an FPGA, while executing an average of only 463 input combinations out of a parameter space with millions of combinations. Total planning time averaged 51 seconds.

This research was supported by the National Science Foundation (CNS-0914474).

The paper is formatted as follows. Section II discusses related work. Section III describes the heuristic. Section IV discusses an interface abstraction used by the heuristic. Section V discusses limitations. Section VI presents experimental results.

II. Related Work

Performance prediction, estimation, and analysis are widely studied topics with challenges similar to implementation planning challenges. Performance prediction is often used to evaluate the amenability of a particular architecture for certain applications [1][13][22], to assist design space exploration [18] and verification [26], to help identify performance bottlenecks [19][25], among others. Although the majority of previous work focuses on microprocessors, several previous approaches have focused on performance prediction for FPGAs using analytical [13] and simulation [10] methods. Although existing performance prediction techniques are related to implementation planning in that they must predict performance of an application for a particular device, implementation planning must also predict performances for different devices and for all possible input combinations.

Numerous compiler studies have focused on automatic parallelizing transformations [6][8][14] and adaptive optimization techniques [4][17] to optimize applications for different multi-core architectures. For FPGAs, high-level synthesis tools [3][12][15][28] have focused on translating high-level code into custom circuits. For GPUs, languages such as CUDA [20], Brook [2], and OpenCL [16] provide constructs for explicit parallelism that can be compiled onto numerous cores. One main limitation of all such tools is that efficiency is fundamentally limited by the single algorithm described in application code. The implementation planning heuristic in this paper is complementary, enabling rapid evaluation of multiple candidate implementations.

Previous work on adaptive software libraries is conceptually similar to implementation planning. FFTW (Fastest Fourier Transform in the West) [7] is an adaptive implementation of FFT that tunes its implementation by measuring the execution time of alternative ways of performing an FFT and choosing the fastest. ATLAS [29] is a software package of linear algebra kernels that are capable of automatically tuning themselves through a combination of tweaking code parameters, custom code generation, and choosing between alternative implementations. SPIRAL [23] is a similar framework but explores algorithmic and implementation choices to optimize DSP transforms. Such approaches perform a limited form of implementation planning specific to microprocessor architectures and particular domains. The presented heuristic is a solution for the more general problem of choosing between any implementation, for any domain, running on any device, invoked with any input parameters.

III. The Planner Heuristic

Due to the large exploration space of implementation planning, an exhaustive solution is clearly not feasible. To solve implementation planning in a reasonable amount of time, performance prediction must be used to estimate a large percentage of the exploration space. Although a large amount of previous work has focused on performance prediction, those approaches are specific to particular architectures [10][13][22][26] and/or applications [7][29], often requiring significant developer effort to adapt the performance predictor for new devices and applications. Implementation planning largely automates this process by executing and measuring actual execution time on different devices using different inputs, allowing the effective prediction of performance for *any* possible implementation written in any language, for *any* resource combination provided by the target system, and for *all* invocation parameters. For brevity, the following sections limit discussion to a single implementation and resource combination. Extending the heuristic for all implementations and resource combinations is trivial, only requiring a loop around the presented heuristic.

A. Overview

For simplicity, we initially describe the heuristic for implementations whose execution time depends on a single-dimensional parameter space, such as a function whose execution time depends on the input size. *Although we use input size for the descriptions in this section, the heuristic is capable of handling multi-dimensional parameter spaces*, as discussed in Section IV.

One naïve possibility for an implementation planning heuristic would involve selecting random input parameters, measuring the actual execution time (which we refer to as *sampling*), and then interpolating the results for all inputs. Although straightforward, such an approach is not adaptive and may collect too few samples in regions of high variance and too many samples in regions where interpolation would suffice. This approach additionally raises questions such as how many samples to collect. Another approach would be to select input parameters at fixed intervals (e.g., every 100^{th} input size) with interpolation used in between. While this approach answers the question of how many samples to collect, it raises other questions such as what spacing to use between samples and still does not adapt to the variance in the profile. Because a performance prediction heuristic must invoke the implementation and measure its execution time for each sample collected, it is critical for a heuristic to minimize the total number of samples required while not sacrificing accuracy.

To address these problems, we present an adaptive heuristic, referred to as the *planner*. Given an implementation and a set of prediction parameters that specify the input sizes, accuracy, and confidence levels, the planner generates a two-dimensional piecewise linear function (e.g., Fig. 1), referred to as a *performance profile*, with input size on the x-axis and estimated execution time on the y-axis. The performance profile is stored as the ordered set of points describing each piecewise linear segment, allowing for compilation tools to estimate execution time for any input size by interpolating the nearest points. The adaptive capabilities of the planner are demonstrated in Fig. 1, where the profile uses more linear segments for input ranges with rapid changes in execution time.

The planner comprises of two iterative steps – *segment generation* and *profile generation* – that focus on predicting

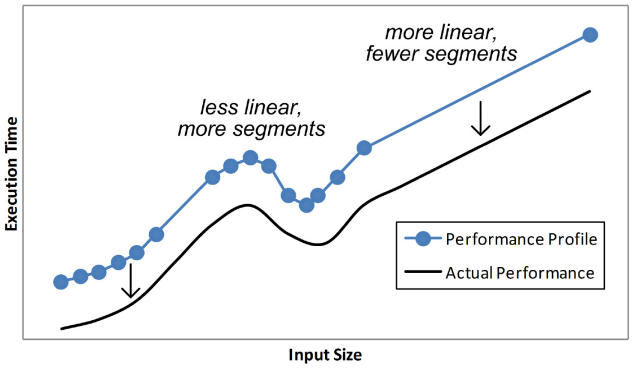

Figure 1. A comparison of actual implementation performance and the performance profile output of the planner (*shifted up for visualization*), which determines maximum-sized regions that can be linearly approximated.

performance for a small region of input sizes, referred to as the *interval of interest (IOI)*, which gradually shifts across the entire input parameter space starting from the smallest sizes in order to create the resulting performance profile.

The main purpose of segment generation is to identify groups of samples that can be approximated by a segment of a linear regression. When high accuracy is requested, such segments may represent only several, closely located samples. When accuracy requirements are lowered, segments may represent distant samples. Segment generation works by sampling an input size within the IOI, measuring the execution time of the implementation at that input size, and then statistically analyzing intervals of samples to find linear trends that meet specified accuracy requirements.

Profile generation selects the best subset of these segments to create the performance profile. Profile generation starts at a segment with a point at the lowest input size and proceeds to connect segments that progress the performance profile towards the largest input size, all the while attempting to minimize the number of segments and the total number of samples with large error. As a result, the performance profile "grows" from the smallest to the largest input size as more samples are added and sufficiently linear segments are found. After expanding the performance profile, profile generation adjusts the IOI and the planner repeats segment generation. The planner completes once profile generation finds a set of segments that span the entire specified input range.

B. Segment Generation

Segment generation is responsible for sampling the relationship between input size and execution time, and then inferring the most-likely linear trends, represented as segments, from those samples. For each iteration of the planner, segment generation starts by randomly selecting an input size within the current IOI, and then measuring the execution time of the implementation at that input size to form a new sample data point. Details regarding the determination of the IOI are discussed in the profile generation section. Although segment generation could potentially use any form of time measurement, we currently use time-stamp counters on the processor to ensure high-precision. Segment generation then generates new segments, and corrects previously generated segments, by considering the new sample collectively with all previous iterations.

Segment generation infers the linear trends from the samples by performing a linear regression analysis. Linear regression analysis is a standard statistical tool that calculates the line that minimizes the mean-squared distance between the line and the samples. Segment generation then shortens the line to a segment by bounding its input size to have the same range as the samples used for the analysis. In addition to the segment, the regression analysis also allows determination of the confidence in the linear trend by calculating the corresponding confidence interval. A confidence interval is another standard statistical tool that specifies how accurately the segment can specify the execution time at any particular input size. For example, the confidence interval might specify that at an input size of 100, the generated segment is 95% confident that the average execution time is within 95-105ms. The results of the linear regression analyses, and the samples themselves, are all stored within a data structure referred to as the *sample matrix*.

Segment generation performs a linear regression analysis on all possible intervals of sequential samples, and stores the corresponding results as cells within the sample matrix data structure. The sample matrix is an upper-right triangular matrix with the number of rows and columns equal to the number of collected samples that have unique input sizes (segment generation combines samples with identical input sizes). Each cell within the sample matrix stores the output of a single linear regression analysis (in addition to other information) performed on a subset of the samples collected. The coordinates of each cell specifies the interval of samples used for that analysis, with the row index equaling the index of the starting sample and the column index equaling the index of the ending sample. As sample generation indexes all of the samples in ascending input size order, an interval of samples corresponds to all of the samples collected within a range of input sizes. For example, the linear regression analysis of the samples indexed between 10 and 20 would be found in the matrix in the cell located at row 10, column 20. If the input size of the sample at index 10 was 35 and the input size of the sample at index 20 was 110, the linear regression analysis would then similarly correspond to any linear trend inferred for the input size range of 35 through 110. As a linear regression analysis requires only the accumulation of data-point statistics for its calculations, sample generation significantly reduces the processing time required to populate the sample matrix by using a dynamic programming algorithm that calculates the linear regression of larger segments from the intermediate results saved from smaller sub-segments. As a result, sample generation can perform the linear regression analysis on one interval of samples by partitioning the interval into two sub-intervals and simply accumulating the intermediate results stored in the corresponding cells of the two sub-intervals. Lastly, the cells located along the matrix's diagonal represent intervals of only a single sample (or multiple samples that all have the same input size). Segment generation populates the diagonal cells with the statistics of the collected samples directly, which in turn form the base cases for generating segments of larger intervals using the dynamic programming algorithm.

When populating each cell of the sample matrix, segment generation performs several checks to gauge the appropriateness of the segment approximation. The first check is to verify that the current interval contains at least three samples so that the linear regression result is non-trivial. The second check is to see if the segment approximation is sufficiently accurate by excluding segments with too wide of a confidence interval. The variance and spread of the samples determines the width of the confidence interval. Likewise, the wider the confidence interval, the less accurately the segment predicts the execution time. A prediction parameter, referred to as the *segment error threshold*, specifies the maximum percent width (i.e., the width of the confidence interval relative to its execution time) allowed by a segment's confidence interval. If any of these checks fail, segment generation flags the corresponding cell as not containing a valid segment. If the segment does pass all the checks, segment generation calculates one last metric, referred to as the *sample out-of-range count*, to use for the profile generation step. Sample generation calculates the sample out-of-range count by counting the number of samples within the interval that have a "large" percent error when compared to the value estimated by the segment. A prediction parameter, referred to as the *sample error threshold*, determines the threshold of what constitutes a "large" percent error. The sample out-of-range count provides a fair comparison between alternate segments in the sample matrix as it lessens the impact of any anomalous samples. Profile generation will then attempt to minimize this count when finding the best set of connected segments to use to form the resulting performance profile.

As illustrated in Fig. 2 (and described in pseudo-code in Fig. 3), each iteration of sample generation requires collecting a new sample, finding the sorted index of the new sample, inserting a new row and column into the sample matrix (assuming the sample has a unique input size), and then reprocessing the cells whose results may change as a result of the new sample. Any cells located at a column less than the newly inserted column or with a row greater than the newly inserted row correspond to intervals of samples located entirely before or after the newly inserted sample, and therefore do not need regeneration. The remaining cells are located in a rectangular region bordered by the newly inserted row and column inclusively. Segment generation populates the newly created diagonal cell with the statistics of the new sample and then processes the remaining cells in decreasing row and increasing column order. Processing the cells in this order allows for the reuse of intermediate calculations for the linear regression analysis, as discussed previously.

C. *Profile Generation*

Profile generation determines the longest set of segments from the sample matrix that can be connected together to form the performance profile that minimizes the total sample out-of-range count. Profile generation uses a dynamic programming algorithm that "grows" the performance profile from the smallest input size towards the largest. The algorithm operates by traversing the sample matrix in increasing row and increasing column order. As the algorithm reaches each cell, it determines and saves in the cell the best candidate segment to precede the current segment, such that the resulting connected

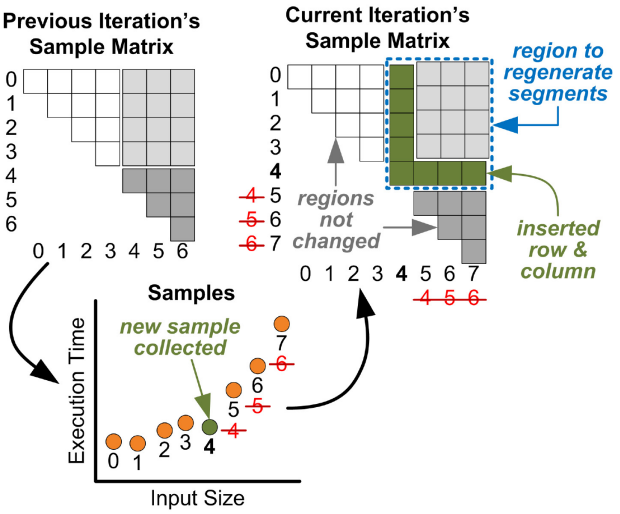

Figure 2. A single iteration of segment generation.

```
-- Information from previous iteration
Let count = previous iteration's # of samples
Let ioi = interval of interest
-- Create new sample data-point
sample.x = RandomInputSizeInRange(ioi)
sample.y = MeasureExecutionTime(sample.x)
-- Insert into sample matrix
Let i = DetermineSortedIndex(sample.x)
For row = count-1 to i step -1
    For col = row to count-1
        Move (row, col) to (row+1, col+1)
    End for
End for
count = count + 1
-- Regenerate changed cells
PopulateDiagonalCell (i, i) with sample
For row = i to 0 step -1
    For col = i to count-1
        GenerateSegment(row, col)
    End for
End for
```

Figure 3. Pseudo-code of the segment generation algorithm.

set of segments, referred to as a *chain*, has the lowest sample out-of-range count (or the least number of total segments in the event of a tie). A candidate segment is any segment which starts prior and ends during the interval of the current segment (i.e., the segments overlap and the endpoint is progressing forward). After the algorithm completes, the best chain of segments is the one that ends at the largest input size, with the lowest sample out-of-range count, and with the fewest number of segments (evaluated in that order). The endpoint of the best chain of segments corresponds to how much of the performance profile the planner has sufficiently determined so far. The planner is complete when the best chain of segments extends all the way to the last valid input size. Upon completion, the planner returns the performance profile as the intersection points between the segments as well as the endpoints for the first and last valid input size values.

Profile generation is also responsible for locating the IOI to encourage the picking of samples that will promote the growth of the best chain of segments in the subsequent iteration. In the

first few iterations when the sample matrix contains no segments, the planner locates the IOI at the lower bound of the input size to pick samples that will help establish the first segments. In later iterations, profile generation centers the IOI at the input size of the last endpoint of the best chain of segments. Centering the IOI at the last endpoint extends the interval beyond the end of the chain, promoting the generation of new segments that will further lengthen the chain. Additionally, the interval extends prior to the last endpoint to allow for the reinforcement or correction of segments recently generated. The width of the IOI is adaptive and set proportionally to the product of execution time and the inverse of the slope of the last endpoint. A prediction parameter, referred to as the *interval growth factor*, specifies the proportionality factor.

As illustrated in Fig. 4 and Fig. 5, each iteration of profile generation requires the reconsideration of some of the cells in the matrix as segments in the best possible chain. Any cell with a column less than the index of the newly inserted sample (for the current iteration), would not have any of its predecessors affected by the sample insertion, and therefore would still be valid. However, any cells with a column greater-than or equal to the newly inserted sample could either itself have changed and/or might have a different selection of predecessors, possibly invalidating any previous predecessor decisions. Profile generation loops through the cells that require reconsideration in increasing row and increasing column order, so that all of the possible predecessors are already valid. To reduce the time complexity, the algorithm keeps track of the best predecessor segment as it traverses the cells (omitted from the figures for simplicity), as opposed to searching for the best predecessor candidate for each cell individually. Lastly, the algorithm searches the sample matrix and returns the best chain (omitted for brevity). Implementation assessment is complete if the chain extends to the last valid work metric value, or alternatively uses the chain to determine the next iteration's IOI.

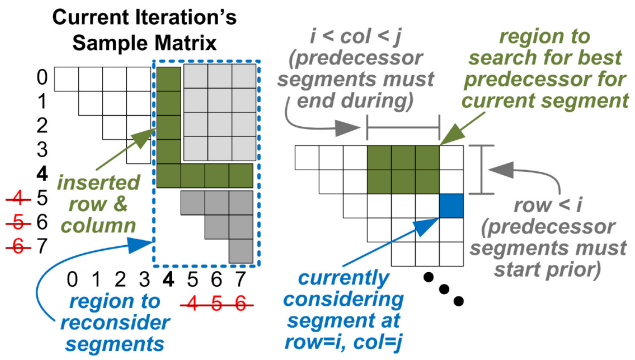

Figure 4. A single iteration of profile generation.

```
-- Information from sample generation
Let count = current iteration's # of samples
Let i = index of newly inserted sample
-- Reconsider segments in sample matrix
For row = 0 to count-1
    For col = Max(i, row) to count-1
        -- Find best predecessor for current cell
        pred = NULL
        For row2 = 0 to row-1
            For col2 = row+1 to col-1
                pred2 = cell(row2, col2)
                If (pred2 better than pred)
                    pred = pred2
                End if
            End for
        End for
        cell(row, col).pred = pred
    End for
End for
-- Find best chain with largest column (omitted)
```

Figure 5. Pseudo-code of the profile generation algorithm.

IV. The Adapter Interface Abstraction

Abstraction is required to keep the planner independent of the implementation. For example, each implementation expects specific invocation syntax and semantics to execute. Software implementations require valid initializations of input parameters, some of which may require the further allocation and population of data arrays (e.g., invoking a sorting subroutine requires the population of an input array with data to sort). Similarly, heterogeneous implementations may require device-specific code to initialize resources and initiate execution. Since the goal of the planner is to predict performance for implementations running on any resource in a multi-core heterogeneous system, keeping these details out of the planner is critical for wide applicability.

To achieve these goals, the planner uses an implementation-specialized abstraction layer between the planner and the implementation, which we refer to as the *adapter*. The main responsibility of the adapter is to adapt (i.e., map) the abstract interface of the planner to the specific interface required by the implementation. The planner defines an abstract quantity called the *work metric* that it passes into the adapter during sample execution. The adapter internally maps the work metric to input parameters for the implementation, executes the implementation with those parameters, and then returns the implementation's execution time to the planner. The adapter additionally performs any necessary initialization and tear-down required by the implementation, which is excluded from the returned execution time. The previous discussion of the planner assumed that the adapter used the input size as the work metric.

Note that although it may be possible in some situations to automatically create an appropriate adapter for a given implementation, we currently assume that the adapter is designer specified. One envisioned usage case is that designers of a specialized function library with multiple implementations of each function would also provide an appropriate adapter for each implementation. Automatic generation of adapters is left as future work.

Automatically adapting the work metric to an implementation's input parameters is essential for the effective operation of the planner. As far as the planner is considered, the sampled collection of work metric and execution time pairs is the only information the planner can collect from an implementation. As the number of samples increases, the underlying trend of how the work metric affects execution time is central to the statistical analyses steps. As a result, making

```
InsertionSortAdapter(work_metric : integer) {
    // Create array of 'work_metric' size
    // and populate with random values
    values[1..work_metric] = rand();

    // Start execution timer
    Timer.Start();

    // Use insertion-sort algorithm to sort values
    InsertionSort(values);

    // Stop execution timer
    Timer.Stop();

    // Return elapsed execution time to planner
    return Timer.Elapsed();
}
```

Figure 6. Pseudo-code of an insertion-sort adapter.

these trends identifiable should be a design goal of the adapter's designer.

In many cases the adapter can directly map work metrics to input parameters. For example, Fig. 6 demonstrates an adapter for an insertion-sort implementation that maps the work metric to the size of the input array to sort. Internally, the adapter allocates and populates an array with random values before actually invoking the insertion-sort subroutine. The adapter measures the execution time of the insertion-sort subroutine, which is then returned to the planner. For this sorting example, the points on a work metric versus execution time graph will show an underlying trend of quadratic execution time growth with increasing work metric (as insertion-sort is an $O(n^2)$ algorithm where n is the number of items to sort).

Similar techniques work for implementations whose execution time is dependent on input values as opposed to size. For example, an adapter for a Fibonacci function implementation could map the work metric to the particular input value. In many cases, a particular implementation's execution time may depend on both input size and input values. For example, a sorting implementation may perform differently depending on whether the data is mostly sorted or randomly distributed. To deal with these situations, multiple adapters could potentially be used to create multiple performance profiles, which a user could select based on characteristics of their targeted application.

Creating an adapter for multiple-parameter implementations requires a bit of ingenuity on behalf of the adapter designer, but can typically be done by taking advantage of knowledge of the underlying algorithm. As an example, consider discrete circular convolution that convolves two input arrays. Unlike the sorting example, whose input size mapped easily to the work metric, convolution has two input parameters whose sizes both significantly affect the execution time of the implementation. The adapter in this case can be written by taking advantage of the asymptotic performance analysis of the circular convolution algorithm. For example, if the designer knows that the asymptotic performance of the implementation is $\Theta(|x|*|h|)$, that is the execution time is proportional to the product of the sizes of the two operands, then a simplification can be made by noting that the proportionality factor in the

```
CircConvolutionAdapter(work_metric : integer) {
    // Create array of 'work_metric' size
    // and populate with random values
    x_operand[1..work_metric] = rand();

    // Create array of fixed size and populate
    constant FIXED_H_LENGTH = 16;
    h_operand[1..FIXED_H_LENGTH] = rand();

    // Start execution timer
    Timer.Start();

    // Perform discrete circular convolution
    CircularConvolution(x_operand, h_operand);

    // Stop execution timer
    Timer.Stop();

    // Return elapsed execution time to planner
    return Timer.Elapsed();
}
```

Figure 7. Pseudo-code of a circular convolution adapter.

asymptotic analysis should be approximately constant for all lengths of x and h. In other words, the execution time of convolving a 20-element array with a 30-element array is likely to be similar to convolving a 10-element array with a 60-element array, due to the product of both being 600. As a result, the adapter can be written as shown in Fig. 7, which maps the work metric to the size of one of the input operands and fixes the size of the second to a constant value (16 in this example). The resulting performance profile can then be used to predict the performances of any invocation by simply finding the product of the sizes of the two operands and dividing that product by 16. Similar methods can be applied to create adapters for many other multiple-parameter implementations.

Note that as opposed to being "duct tape" that makes the planner support multiple dimensions, the adapter provides several important advantages. First, by mapping to a single dimension, the adapter enables the planner to complete quickly, as shown by the results. In addition, the adapter also reduces the size of the resulting performance profile, which potentially enables implementation selection at runtime. For example, a runtime optimization framework could implement a function call by first determining available resources and the current input parameter values, and then using the performance profiles to quickly identify the most efficient implementation for the current situation. Lastly, the simplicity of a single dimensional performance profile allows for efficient post-processing, such as overlaying multiple performance profiles and storing only the lowest envelope (i.e., fastest) implementation. For future work, we plan to extend the adapter to natively support multiple dimensions by extending the planner to perform multi-dimensional regressions using multi-dimensional intervals of interest. However, even with support for multiple dimensions, there are likely many situations where mapping to a single dimension will greatly reduce planning time, while still meeting accuracy requirements.

In addition to mapping the work metric to input parameters, the adapter also provides an inverse mapping to map from

input parameters to a corresponding work metric. As the planner deals only with the work metric abstraction, the performance profile output is defined in terms of the work metric, essentially predicting the execution time for a given work metric. As a result, using the performance profile to predict the execution time of a particular invocation requires mapping that invocation's parameters to its corresponding work metric. In most cases, this mapping from the parameters to the work metric is simpler than the reverse. In the insertion-sort example, the inverse mapping would simply use the number of elements of that sort as the work metric (as the adapter used the work metric as the number of elements to sort). In the circular convolution example, the inverse mapping would be to calculate the work metric by multiplying the lengths of the two input operands and dividing by 16.

V. LIMITATIONS

Most of the limitations of the planner arise from some implementations not having suitable adapters. First, an implementation should have predictable and well-behaved performance characteristics to allow for the creation of an adapter. For implementations that are not deterministic (e.g., random execution times) or do not have a good mapping for the work metric (e.g., multi-dimensional parameter spaces that can't be accurately approximated with a single dimension), the planner will likely have reduced prediction accuracy. Second, caching, data alignment, and other architecture-specific effects may add nondeterministic effects that will increase the error of the performance profile. However, as shown by the results, in many cases these effects are minimal and have little effect on the predicted execution times. Lastly, adapters for multi-dimensional implementations typically neglect corner cases. For example, the asymptotic analysis of the circular convolution algorithm, described in Section IV, assumed a constant proportionality factor for all input parameter combinations. However, this assumption is likely incorrect for corner cases such as when one of the input vectors is a single element (e.g., convolving 1,000,000 elements with 1 element would likely have a significantly different execution time than convolving two 1,000 element vectors, despite having the same product). This limitation could potentially be improved with more complicated adapters or by extending the work metric to directly handle multiple dimensions, which we plan as future work.

VI. EXPERIMENTS

A. Experimental Setup

Table I describes the implementations that we created to evaluate the planner. We selected these implementations in order to represent common functions from different programming and application domains. *Implementation* describes the function/algorithm used by the implementation. *Work Metric Range* defines the range of work metrics for which the planner predicted performance. Note that in order to allow the planner to complete in a reasonable amount of time, we chose all work metric ranges such that the worst-case sample execution (usually the largest work metric) took less-than a few seconds to complete. Although these limits exclude portions of the input parameter space, the evaluated ranges are representative of common usage. *Adapter Details* explains the methodology the adapter uses to map work metrics to input parameters. The table groups the implementations into three different types. The *Single-threaded Implementations* group lists the implementations using sequential algorithms. The *Multi-threaded Implementations* group lists the implementations that can partition work across one or more threads. Lastly, the *FPGA Implementations* group lists the implementations that perform their processing on an FPGA with basic support from a microprocessor to transfer data. The single-threaded and FPGA implementations can execute on only one specific set of resources and therefore require only a single performance profile for a system. The multi-threaded implementations, however, have different execution times based on the number of available CPUs and therefore require a different performance profile for different CPU counts.

TABLE I. IMPLEMENTATION DETAILS

Implementation	Work Metric Range	Adapter Details
Single-threaded Implementations		
Heap Sort	[1, 4000000]	Work metric is size of sort. Random data populates input array.
Insertion Sort	[1, 65000]	Work metric is size of sort. Random data populates input array.
Longest Common Subsequence	[1, 1000000]	Work metric is length of one string, other string is fixed to length 256.
Quick Sort	[1, 10000000]	Work metric is size of sort. Random data populates input array.
Multi-threaded Implementations		
2D Convolution	[1, 10000]	Work metric is number of rows of image, number of columns is fixed to 128. Convolving window is fixed to 8x8
Circular Convolution	[1, 2500000]	Work metric is length of one of the operands, other operand is fixed to length 256.
Floyd-Warshall	[1, 600]	Work metric is number of vertices. All edges have random weights.
Inner Product	[1, 10000000]	Work metric is length of both input operands.
Matrix Multiply	[1, 2500000]	Work metric is one dimension's length, other two dimensions are fixed to length 16.
Mean Filter	[1, 25000]	Work metric is number of rows of image, number of columns is fixed to 256.
Optical Flow	[1, 10000]	Work metric is number of rows of image, number of columns is fixed to 128. Template image is fixed to 8x8.
Prewitt	[1, 25000]	Work metric is number of rows of image, number of columns is fixed to 256.
FPGA Implementations		
Circular Convolution	[1, 1000000]	Work metric is length of one of the operands, other operand is fixed to length 2048.
Inner Product	[1, 1048576]	Work metric is length of both input operands.
Matrix Multiply	[1, 4096]	Work metric is one dimension's length, other two dimensions are fixed to length 256.

We evaluated the planner using twelve examples. *Insertion Sort*, *Heap Sort,* and *Quick Sort* are in-place sorting algorithms

TABLE II. SUMMARY OF PLANNER RESULTS

Implementation	Planner Time	Samples	Profile Points	Mean Error	RMSE
2D Convolution	36.2 sec	295	11.6	5.9%	8.5%
Circular Convolution	42.4 sec	534	4.4	11.0%	11.5%
Floyd-Warshall	31.9 sec	382	16.4	5.7%	8.2%
Heap Sort	76.8 sec	633	14.7	1.8%	2.2%
Inner Product	20.6 sec	392	7.6	1.6%	2.2%
Insertion Sort	81.4 sec	582	23.0	2.5%	2.8%
LCS	97.4 sec	578	13.7	2.1%	3.7%
Matrix Multiply	64.2 sec	502	9.5	38.3%	41.1%
Mean Filter	24.8 sec	350	6.6	2.0%	3.9%
Optical Flow	42.8 sec	304	8.9	0.6%	1.0%
Prewitt	33.9 sec	353	10.4	1.4%	2.3%
Quick Sort	60.8 sec	654	12.7	1.3%	1.6%
Average	**51.1 sec**	**463**	**11.6**	**6.2%**	**7.4%**

with different asymptotic complexities. *Longest Common Subsequence (LCS)* finds the longest, not necessarily contiguous, series of common characters between two strings using a dynamic programming-based Θ(|a|*|b|) algorithm, where |a| and |b| are the lengths of the two input strings. *2D Convolution, Mean Filter, Optical Flow,* and *Prewitt* apply a sliding window to an image using an Θ(x*y) algorithm, where x and y are the dimensions of the image. *Circular Convolution* convolves two vectors using an Θ(|x|*|h|) algorithm, where |x| and |h| are the sizes of the two input operands. *Floyd-Warshall* is a dynamic programming-based graph algorithm that finds the shortest path between all pairs of vertices in a directed weighted graph. *Inner Product* calculates the inner-product on two identically sized vectors using an Θ(n) algorithm, when n is the length of each vector. *Matrix Multiply* multiplies two, not necessarily square, matrices using an Θ(m*n*p) algorithm, where one operand is of dimension m x n and the second operand is of dimensions n x p.

We evaluate the planner on three platforms. Platform #1 has a hyper-threading 3.2GHz Intel Xeon processor with an attached Nallatech H101-PCIXM FPGA accelerator board, which has a Xilinx Virtex IV LX100 FPGA. Hyper-threading makes platform #1 appear as though it has two cores, but the cores must partially contend for the same processing resources, which increases the difficulty of implementation planning. Platform #2 has eight 2.4GHz dual-core AMD Opteron processors (16 cores total). Platform #3 has two 2.6 GHz quad-core Intel Xeon processors (8 cores total). We selected these platforms due to their numerous processing resources and differing system architectures. We wrote the planner and software implementations in C++ and compiled them on each platform individually using g++ with highest-level optimizations. We wrote the FPGA implementations in VHDL and compiled them using Xilinx ISE 10.1i.

B. *Analysis of Prediction Error*

In this section, we evaluate the planner's prediction error by comparing the predicted and actual execution times of 100 executions using random input parameters. For all of the examples, the planner uses prediction parameters specifying that the segment error threshold, sample error threshold, and the interval growth factor are all set to 10%. Additionally, all confidence calculations use a confidence level of 95%.

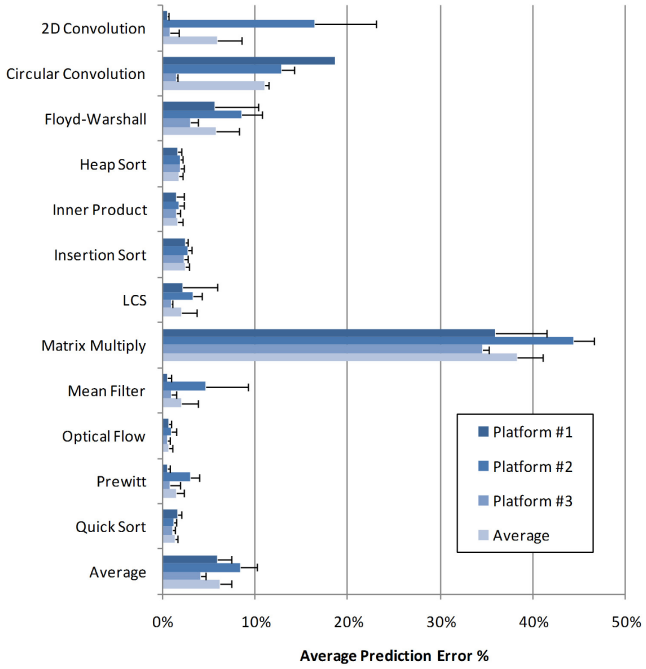

Figure 8. Average prediction error % (bars) and root-mean-squared error % (lines) of the performance profiles generated for each implementation.

Table II summarizes the results of the profiling process for each implementation. The results are averaged across all platforms and differing resource amounts, for each implementation. *Implementation* is the name of the implementation. *Planner Time* is the average time required for the planner to create the performance profile, which includes the time executing each sample. *Samples* is the average number of samples the planner collected to create the performance profile. *Profile Points* is the average number of points in the resulting performance profile. *Mean Error* is the average percentage of prediction error of the performance profile. *RMSE* is the average percentage of root-mean-squared error of the performance profile.

Fig. 8 shows the average percent prediction error, represented by the bars and the root-mean-squared error, represented by the lines, for each implementation. The results for each implementation were averaged across all resource amounts available on a platform (for multi-threaded and heterogeneous implementations). For example, on platform #2, which has 16 cores, the results represent the average error for all possible resource allocations (i.e., 1 to 16 cores).

On average, the planner collected 463 samples, which required only 51.1 seconds to complete, and generated a performance profile averaging only 11.7 points for each implementation. Although the planning time will increase for longer-running implementations, the small number of samples should enable planning for many commonly used functions. Despite the low number of profile points, the performance profile achieved an average prediction error of less-than 6% and a root-mean-squared error of less-than 9% for all but two of the implementations. The circular convolution implementation achieved a prediction error of 11.0% and

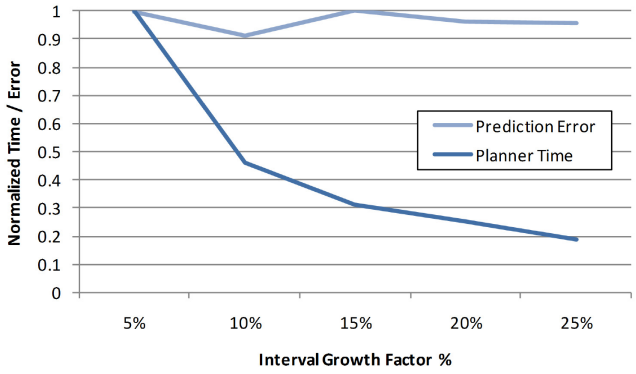

Figure 9. Effect of changing the interval growth factor on planner time and prediction error.

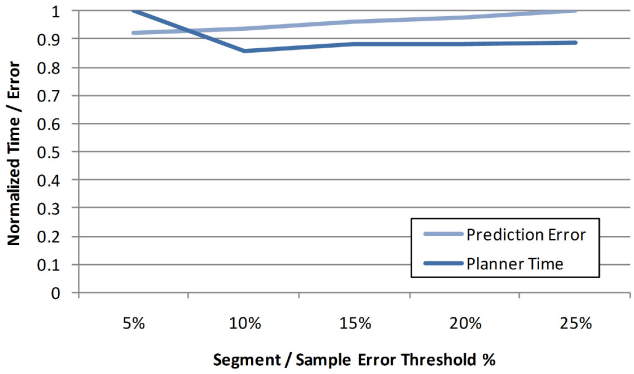

Figure 10. Effect of changing the segment error threshold and sample error threshold (both set to the same value) on planner time and prediction error.

matrix multiply achieved an error of 38.3%. The larger prediction error of circular convolution and matrix multiply is largely due to inaccurate assumptions made by their adapters. Both adapters assumed the number of multiply-accumulate operations was a good predictor of processing time for the implementation (i.e., the implementation was computation-bound). Likewise, the adapter assumed that input parameters requiring a similar number of multiply-accumulates would require approximately the same amount of execution time. On the evaluated systems, the implementations were partially data-bound, with their processing time related more to the amount of data read and written by the implementation. Platform #3 showed the least prediction error for both circular convolution and matrix multiply due to its significantly faster memory reducing the data-transfer bottleneck and improving the computation-bound assumption. Results for these examples could potentially be improved by integrating micro-benchmarking results into the adapter to better estimate the effects of data transfer times. We leave such extensions as future work.

Note that although a 38% prediction error for matrix multiply may seem limiting, there are instances of implementation planning where such an error may be acceptable. For example, if a compiler was attempting to identify the best implementation for a system with a microprocessor and FPGA, the FPGA implementations may often be orders of magnitude faster, which makes the 38% prediction error negligible.

There are of course situations where a 38% error is not acceptable. As future work, we plan to extend to the adapter to natively support multiple dimensions by extending the planner to perform multi-dimensional regressions using multi-dimensional intervals of interest. Note that even after directly supporting multiple dimensions, mapping onto a single dimension has unique advantages, as discussed in Section IV. We also plan to add a tuning process that refines the performance profile over time as actual executions show that particular results are inaccurate.

C. *Analysis of Prediction Parameters*

In this section, we analyze the effects of the prediction parameters on both the planner execution time and average prediction error of the planner. For all reported results, the corresponding values were first averaged across all platforms and different resource amounts and then normalized. The confidence level was kept constant at 95%.

Fig. 9 illustrates how planner time and prediction error are affected by the interval growth factor prediction parameter. The segment error threshold and sample error threshold were kept constant at 10%. As shown in the figure, increasing the interval growth factor can significantly improve the planner execution time without worsening the prediction error. The factor is inversely related to the planner's execution time, as demonstrated by the halving of the execution time as the factor is doubled. This makes sense as the width of the IOI, and correspondingly the average space between samples, increases linearly with the interval growth factor. Most interestingly, the prediction error was not significantly affected by changing the factor, which is largely attributed to the segment error threshold still being sufficient for segment generation to determine the appropriateness of segments, despite the increased spacing between samples.

Fig. 10 illustrates how the planner time and prediction error are affected by the segment error threshold and sample error threshold. Both error thresholds were set to the same value. The interval growth factor was kept constant at 10%. The results show that the error thresholds have only a minor impact on the planner execution time and prediction error. Increasing the error thresholds from 5% to 25% linearly increased the (normalized) average prediction error from 0.92 to 1. An increase in the prediction error is expected as the thresholds for rejecting a segment in the segment generation step are increased. The relatively small change in prediction error is likely due to the out-of-range count still being sufficient for determining the best chain of segments to form the performance profile. The planner execution time initially decreases as the error threshold sweeps from 5% to 10%, and then increases from 10% to 25%. The initial decrease is due to the lessening number of samples required to meet the segment error threshold. The subsequent increase is due to the greater number of candidate segments in the profile generation step, as more segments meet the segment error threshold.

VII. Conclusions

To deal with the widely varying algorithms required by different devices on multi-core heterogeneous systems, future compilers will need to identify efficient implementations from among numerous possibilities for any possible resource and combination of invocation parameters. To help solve this problem, which we defined as implementation planning, we present a performance prediction heuristic that automatically samples input parameters of an implementation, statistically analyzes the resulting execution times, and creates a performance profile that may then be used by a compiler to predict the execution time of the implementation for any combination of input parameters. On average, the heuristic achieved a prediction error of 6.2% and a root-mean-squared error of 7.4% for three widely varying systems, while only sampling 463 points per implementation, resulting in an average execution time of only 51 seconds.

References

[1] T. Austin, E. Larson, and D. Ernst, "Simplescalar: An infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, pp. 59–67, 2002.

[2] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for gpus: stream computing on graphics hardware," in *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, (New York, NY, USA), pp. 777–786, ACM, 2004.

[3] B. Buyukkurt, Z. Guo, and W. Najjar, "Impact of loop unrolling on throughput, area and clock frequency in ROCCC: C to VHDL compiler for FPGAs," in *ARC '06: Proceedings of the International Workshop On Applied Reconfigurable Computing (ARC 2006)*, 2006.

[4] K. D. Cooper, D. Subramanian, and L. Torczon, "Adaptive optimizing compilers for the 21st century," *J. Supercomput.*, vol. 23, no. 1, pp. 7–22, 2002.

[5] A. DeHon, "The density advantage of configurable computing," *Computer*, vol. 33, no. 4, pp. 41–49, 2000.

[6] A. Eichenberger, K. O'Brien, P. Wu, T. Chen, P. Oden, D. Prener, J. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind, "Optimizing compiler for the cell processor," in *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*, pp. 161–172, Sept. 2005.

[7] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[8] M. Girkar and C. D. Polychronopoulos, "Extracting task-level parallelism," *ACM Trans. Program. Lang. Syst.*, vol. 17, no. 4, pp. 600–634, 1995.

[9] B. Grattan, G. Stitt, and F. Vahid, "Codesign-extended applications," in *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*, (New York, NY, USA), pp. 1–6, ACM, May 2002.

[10] E. Grobelny, C. Reardon, A. Jacobs, and A. George, "Simulation Framework for Performance Prediction in the Engineering of RC Systems and Applications," in *ERSA '07: Proc. of 2007 International Conference on Engineering of Reconfigurable Systems and Algorithms*, June 25-28, 2007.

[11] Z. Guo, W. Najjar, F. Vahid, and K. Vissers, "A quantitative analysis of the speedup factors of FPGAs over processors," in *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, (New York, NY, USA), pp. 162–170, ACM, 2004.

[12] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "Spark: a high-level synthesis framework for applying parallelizing compiler transformations," in *VLSI Design, 2003. Proceedings. 16th International Conference on*, pp. 461–466, Jan. 2003.

[13] B. Holland, K. Nagarajan, and A. D. George, "Rat: Rc amenability test for rapid performance prediction," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 1, no. 4, pp. 1–31, 2009.

[14] C. S. Ierotheou, S. P. Johnson, P. F. Leggett, M. Cross, E. W. Evans, H. Jin, M. Frumkin, and J. Yan, "The semi-automatic parallelisation of scientific application codes using a computer aided parallelisation toolkit," *Sci. Program.*, vol. 9, no. 2,3, pp. 163–173, 2001.

[15] Impulse Accleratated Technologies. 2010. http://www.impulseaccelerated.com/.

[16] Khronos Group. OpenCL 1.0. 2010. http://www.khronos.org/opencl/.

[17] P. M. W. Knijnenburg, T. Kisuki, and M. F. P. O'Boyle, *Iterative compilation*, pp. 171–187. New York, NY, USA: Springer-Verlag New York, Inc., 2002.

[18] G. Madl, N. Dutt, and S. Abdelwahed, "Performance estimation of distributed real-time embedded systems by discrete event simulations," in *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*, (New York, NY, USA), pp. 183–192, ACM, 2007.

[19] G. R. Nudd, D. J. Kerbyson, E. Papaefstathiou, S. C. Perry, J. S. Harper, and D. V. Wilcox, "Pace–a toolset for the performance prediction of parallel and distributed systems," *Int. J. High Perform. Comput. Appl.*, vol. 14, no. 3, pp. 228–251, 2000.

[20] Nvidia. CUDA Programming Guide. 2008. http://www.nvidia.com/object/cuda_develop.html.

[21] F. Petrini, G. Fossum, J. Fernandez, A. Varbanescu, N. Kistler, and M. Perrone, "Multi-core surprises: Lessons learned from optimizing sweep3d on the cell broadband engine," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pp. 1–10, March 2007.

[22] W. Pfeiffer and N. J. Wright, "Modeling and predicting application performance on parallel computers using hpc challenge benchmarks," in *22nd IEEE International Parallel and Distributed Processing Symposium, Hyatt Regency Hotel, Miami, FL, 2008*, 2008.

[23] M. Puschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. Johnson, and N. Rizzolo, "Spiral: Code generation for dsp transforms," *Proceedings of the IEEE*, vol. 93, pp. 232–275, Feb. 2005.

[24] S. Sirowy, G. Stitt, and F. Vahid, "C is for circuits: capturing FPGA circuits as sequential code for portability," in *FPGA '08: Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, (New York, NY, USA), pp. 117–126, ACM, February 2008.

[25] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha, "A framework for performance modeling and prediction," in *Supercomputing, ACM/IEEE 2002 Conference*, pp. 21–21, Nov. 2002.

[26] Y. tsun Steven Li, S. Malik, and A. Wolfe, "Performance estimation of embedded software with instruction cache modeling," in *ACM Transactions on Design Automation of Electronic Systems*, pp. 380–387, 1995.

[27] P. Trancoso and M. Charalambous, "Exploring graphics processor performance for general purpose applications," in *Digital System Design, 2005. Proceedings. 8th Euromicro Conference on*, pp. 306–313, Aug.-3 Sept. 2005.

[28] F. Vahid, G. Stitt, and R. Lysecky, "Warp processing: Dynamic translation of binaries to FPGA circuits," *Computer*, vol. 41, pp. 40–46, July 2008.

[29] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimization of software and the ATLAS project," *Parallel Computing*, vol. 27, no. 1–2, pp. 3–35, 2001.