DYNAMIC SCHEDULING OF STREAMING APPLICATIONS ON MULTICORES

A Thesis
Presented to
The Academic Faculty

by

Farhana Aleen

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
College of Computing

Georgia Institute of Technology
December 2010

DYNAMIC SCHEDULING OF STREAMING APPLICATIONS ON MULTICORES

Approved by:

Professor Santosh Pande, Advisor
College of Computing
Georgia Institute of Technology

Professor Sudhakar Yalamanchilli
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Hyesoon kim
College of Computing
Georgia Institute of Technology

Date Approved: August 2009

To my family,

for their never ending support and unconditional love.

ACKNOWLEDGEMENTS

First, I would like to thank my advisor Dr. Santosh Pande for providing me with valuable insights and guidance for this Thesis. Throughout my Masters he has given me the opportunity and freedom to work on several interesting problems and has helped me grow and explore the exciting research areas in compilers. I would like to thank my colleague and friend Ashwini Bhagwat and Tushar Kumar for their technical feedback and invaluable friendship. I would like to thank my committee members Dr. Sud- hakar Yalamanchilli, Dr. Hyesoon Kim and Dr. Nate Clark for their valuable time and feedback. Many thanks to the Sony Toshiba IBM Consortium for funding my research.

Finally, I would like to acknowledge Monirul Sharif for his unconditional support. I would not have come this far without his love and encouragement. Last but not the least, My family, spcially my dad, I can not thank them enough. Because of their love and inspiration today I am here.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The number and scope of data driven streaming applications is growing. Such streaming applications are promising targets for effectively utilizing multi-cores because of their inherent amenability to pipelined parallelism. While existing methods of orchestrating streaming programs on multi-cores have mostly been static, real-world applications show ample variations in execution time that may cause the achieved speedup and throughput to be sub-optimal. One of the principle challenges for moving towards dynamic pipeline balancing has been the lack of approaches that can predict upcoming dynamic variations in execution efficiently, well before they occur. In this thesis, we propose an automated dynamic execution behavior prediction approach based on compiler analysis that can be used to efficiently estimate the time to be spent in different pipeline stages for upcoming inputs. Our approach first uses dynamic taint analysis to automatically generate an input-based execution characterization of the streaming program, which identifies the key control points where variation in execution might occur with respect to the associated input elements. We then automatically generate a light-weight emulator from the program using this characterization that can predict the execution paths taken for new streaming inputs and provide execution time estimates and possible dynamic variations. The main challenge in devising such an approach is the essential trade-off between accuracy and overhead of dynamic analysis. We present experimental evidence that our technique can accurately and efficiently estimate dynamic execution behaviors for several benchmarks with a small error rate. We also showed that the error rate could be lowered with the trade-off of execution overhead by implementing a selective symbolic expression generation for each of the complex conditions of control-flow operations.

Our experiments show that dynamic pipeline balancing using our predicted execution behavior can achieve considerably higher speedup and throughput along with more effective utilization of multi-cores than static balancing approaches.

CHAPTER I

INTRODUCTION

Multicores have become the standard in today's desktop and laptop computers. With this shift, automated tools and compiler support to aid programmers to wring out the parallelism from programs and effectively utilize the available hardware has become important than ever before. Among the various application paradigms, streaming applications covering the vast domains of graphics, multimedia and digital signal processing are showing promise because of their inherent amenability to parallelism on multicores.

On one end of the spectrum, research on the stream programming paradigm has spurred many special-purpose stream languages such as StreamIt [22], Brook [2], CUDA[15], Cg [13], etc. On the other end, an undeniably large fraction of streaming applications are still written in C/C++, from which, due to their single instruction stream and monolithic memory, it is difficult to extract parallelism. Nonetheless, recent work on exploiting fine-grained [8, 16, 18] and coarse-grained [21] pipeline parallelism in C programs shows that there is still hope for porting legacy programs to multicores.

In order to perform *orchestration* or *pipeline balancing*, the scheduling of pipelined stages onto different cores compilers require an estimation of the execution time of the stages. To date, the primary focus of balancing and scheduling approaches have been static, utilizing a static execution profile (e.g. with StreamIt [22, 11]). However, recent streaming applications, especially in the domain of multimedia, are showing increasing dynamic variation in execution time. Emerging and newer standards such as MPEG-4 support variable data rates and types in the streaming data to pack

more information into the same bitstream. As streaming applications become more common, static balancing and scheduling may provide suboptimal speedups, and supporting dynamic approaches will be necessary to utilize the available hardware better. For dynamic pipeline balancing to be effective, we need to characterize the execution behavior in a manner that any change in execution timing can be predicted efficiently and accurately well in advance before they occur.

In this thesis, we present a novel approach for efficiently and accurately predicting variations in execution behavior of streaming applications. Our insight is that a program's input drives its execution. Based on this, our approach first automatically extracts an input-based execution behavior characterization of the program. Second, we automatically generate what can be considered a light-weight emulator for the program, which uses the execution characterization on new incoming input streams to simulate executions paths choices and generate execution time estimates. A dynamic approach can predict upcoming execution behavior variations using our system and effectively balance and schedule pipeline stages of the real program for higher speedup on multicores.

In order to extract the input-based execution behavior, we utilize dynamic multi-label taint propagation [14] on the streaming applications. With the analysis we generate an input-based *execution characterization graph*, that captures the key control points in the program where execution may vary and contains information to identify variation-causing input elements from within any input stream of the program. Our approach uses this graph, together with the profile information of static regions in the program, to automatically scan a given new input stream and compute an estimated execution time for pipeline stages if the program was executed with the input. Since the estimates can be generated up to the point of available inputs at a fraction of the actual execution cost, a dynamic pipeline balancing and scheduling approach can utilize our prediction system with buffered inputs beside the real

program to predict execution variations well in advance.

The input characterization graph is general enough to be usable to parse any unknown input stream and determine the branches taken and the number of iterations taken in loops for processing the inputs. Our execution time estimation method uses this information, together with the profile information of static regions in the program, to parse a given input stream and deduce an estimate of the execution timing pattern of any region in the program when it would be executed with that input.

For evaluating our approach, we have implemented a dynamic analysis tool based on PIN. Our system can take a streaming programs's compiled binary and a set of streaming inputs to automatically extract the input characterization graph. We have also built a tool that uses the automatically generated input characterization graph and partial profile information to simulate execution path choices on a new input stream (in near liner time), and generate execution time estimates. For the given input, the tool can generate estimated execution times for the pipeline stages annotated in the streaming program, for each pipelined loop iteration.

We have experimentally evaluated our approach on four real-world benchmarks. The detailed case-studies on the execution time predictions showed a maximum of 8% average error rate compared to the actual execution times for the benchmarks. In order to evaluate the expected speedup by a dynamic balancing and scheduling approach utilizing our system over a static one, we performed dynamic balancing and scheduling of the pipeline stages on the real benchmarks. For three out of the four benchmarks, dynamic balancing showed noticeable speedup gains over static balancing. For MPEG-4, we achieved an upper bound speedup of 3.5 compared to 2.6 achievable by the static approach. We also investigated the inaccracy of the prediction of execution time and we were able to improve the accuracy by 3% more with the introduction of huge overhead. We performed the precision analysis of our prediction apprach on the above four benchmarks and provided the experimental

results in this thesis.

We summarize the contributions of our paper below:

- We present a novel automated approach that enables efficient and accurate prediction of dynamic execution behaviors of streaming applications. Our approach is automated and the execution time estimates are generated by scanning the inputs only with a fraction of execution time required by the actual program.

- Based on our proposed approach, we have implemented a dynamic analysis tool using PIN for extracting the execution characterization from a compiled streaming application. We have also implemented an execution time prediction tool for estimating the execution times for pipeline partitions annotated in C programs.

- We provide experimental evidence that show our approach can predict extract input-based execution behavior from real-world programs and estimate their execution times for new input streams with sufficient accuracy and efficiency.

- Finally, we have done precision analysis on our proposed approach. With experimental results we have showed that the error rate of our execution prediction can go down to 4% with couple of orders magnitude overhead. We concluded that the accuracy of our approach is inversely proportional to the execution time.

Although we have focused on analyzing streaming applications written in C in this thesis, we believe our analysis approach should spur new research in analyzing programs in stream programming languages to identify their dynamic execution behavior from input streams, which may lead to better pipeline balancing and improved speedup on multicores.

CHAPTER II

MOTIVATION

The first step for a compiler to *orchestrate* streaming applications using pipelined parallelism [7] is to extract a stream graph that depicts the data dependency between different actors in a pipelined loop of the program. Thies et al. [21] has shown that with little modification in code and with the help of user annotations, independent code partitions similar to actors in StreamIt can be found in stream programs written in C, and they can be parallelized in similar fashion.

The dependencies in the stream graph are utilized to define stages in the software pipeline. Once the stages are defined, the estimated execution time of different stages are used to assign them to cores, so that workload is balanced. Better balance of workload leads to higher utilization of the cores and higher speedup.

Almost all pipeline balancing and scheduling approaches to date have been static [11, 22, 21]. In other words, the stage assignment, workload distribution and scheduling are planned before the program's execution commences. However, a lot of streaming applications show ample variations in execution time [12, 17], especially with recent multimedia standards where input streams may contain a lot of varying values, types and rates of data being presented to the application.

Figure 1(a) shows an example of a stream graph with three partitions. Figure 1(b) shows a balanced distribution based on expected execution times of partitions in the example stream graph, and Figure 1(c) shows the execution in steady state, depicting the iteration time required for performing an iterative module scheduling algorithm [19]. Figure 1(d) illustrates a situation where an imbalance in workload

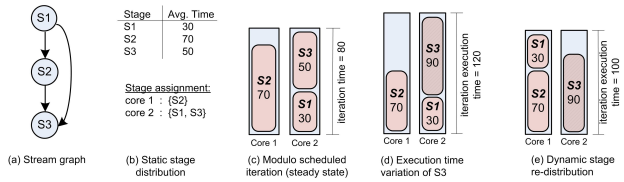

Figure 1: Static vs dynamic pipeline balancing

causes longer overall pipeline iteration time and some processor cores to remain under-utilized, effectively lowering the achievable speedup. Better speedup can be achieved by dynamically altering the workload distribution and scheduling based on foreseeable dynamic variations in the execution times of the stages. Figure 1(e) shows how re-distribution of workload can lower the iteration time, effectively increasing speedup.

One of the key challenges for dynamic pipeline balancing is the determination of dynamic variations in execution for a given input stream. For dynamic balancing to be effective, we present six key requirements for predicting dynamic execution behavior that we also use as our design goals:

1. **Real-time computation:** The approach of predicting variation should be applicable to new inputs to a program and computed on-the-fly by the system.

2. **Early prediction:** During execution of the program, upcoming dynamic behavior needs to be predicted well in advance, at least before the dynamic behaviors commence. Detecting varying execution time late or after they happen may nullify the advantage of dynamic pipeline balancing.

3. **Longer duration:** Execution timing patterns should be available for some time into the future because it can help taking balancing decisions that can provide the best speedup for a longer period of execution without requiring frequent redistribution.

4. **Accurate:** The estimated variations in execution should be as close to the actual variations as possible.

5. **Light-weight:** The approach for dynamic behavior estimation should itself be light-weight compared to the actual program's execution, so that the introduced overhead does not lower achievable speedup.

6. **Automated**: An automated system can relieve a programmer the complexity and hassle of manually analyzing a program or inputs provided to the program, especially for large streaming applications.

CHAPTER III

RELATED WORK

A large body of work has gone into orchestrating programs written in stream languages on various hardware platforms for pipelined parallelism [7, 4]. The StreamIt compiler [22] orchestrates actors on different cores by balancing workload using a greedy approach. Recently, [11] has shown improvements in speedup by using integer linear programming to distribute the workload. All coarse-grained scheduling in these approaches have been performed with a static execution profile whereas we show that with the help of dynamic behavior characterization, dynamic pipeline balancing and scheduling can provide benefits in speedup.

Earlier research has looked in to characterizing dynamic variations of streaming applications by using statistical models [20, 6, 17]. These approaches use a training-based step to generate a model that represents shifting between periodic timing variations. However, they model only the effects of dynamic variations rather than the causes, making it infeasible to predict variation in advance by considering program inputs, which are the root cause. In [12], program execution is analyzed with different input sets to identify which parts of the program have the highest statistical variance in execution time, which is mainly geared towards aiding programmers. We go a step further in actually analyzing the program in such a way that the execution time and its variations can actually be estimated for a new input given to the program.

Previously correlation of inputs with program branches have been performed in [9]. The main goal of that work was to identify the branches of the program from a single input set for which branch prediction accuracy are likely to vary significantly across multiple input sets. Our goals are different - we consider the entire program

control-flow to identify input elements that affect branches even if their position in the input stream may vary.

Reverse engineering input formats of programs have been looked at earlier by the security community. The closest approach to ours is Tupni [3]. Tupni primarily aims at identifying only the syntax of inputs and uses a bottom-up technique for forming higher level input structures. In contrast, we use a top-down approach that is more suitable for the general hierarchical input formats and directly relating inputs with the program's control flow.

CHAPTER IV

APPROACH

4.1 Concept

The key insight of our approach is based on the fact that a program's input drives its execution. Changes in the execution path affect the execution time required for processing the inputs. A program's execution path may vary at control-flow operations that have conditions containing values derived from the inputs. Conditions may be simple, containing only a single value taken directly from an element of the input stream. This mostly occurs in code that tackles the dynamic variations possible in the input stream format. For example, conditional branches may depend on a single value in the input that select one of several types that are possible in subsequent data in the input stream. Similarly, repetitive data may be handled by loops that depend on count-defining input elements or special end markers. In addition, conditions may be complex, using values obtained by performing computation on several input elements. The key to finding how the execution path varies is determining how these conditions are derived from the inputs. The challenge is to be able to achieve all of this in an efficient manner and for any new input that can be given to the program.

At the minimum, it is absolutely required to be able to identify variations in the input structure, so that we can handle any input given to the structure, so that we can handle any input given to the program. Since input formats are generally designed to designate variations in a relatively simple manner, by only handling simple conditions, we should be able to satisfy this requirement. However, since this may not cover all input-driven control-flow operations, accuracy of the predictions may not be high. By handling complex conditions, higher accuracy may be achieved, but by requiring

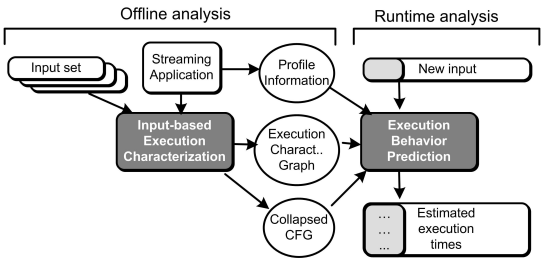

Figure 2: Overall approach

more time in computing the conditions, leading to higher overhead. This presents a tradeoff between accuracy and efficiency. Although it is not a fundamental limitation, we only handled simple conditions in our approach, which we found to be sufficient for real-world streaming applications.

Figure 2 depicts our proposed input-driven dynamic execution behavior prediction approach. It has two main phases -(1) input-based execution characterization, and (2) execution time prediction. In the first phase (Section 4), dynamic tainting is performed on the stream program with a set of representative inputs. The output is an input-based *execution characterization graph* (ECG) that directly captures which execution paths depend on inputs and how they can be identified in the streaming input sequence. The analysis of this phase is performed completely in an offline fashion. In the second phase (Section 5), the execution characterization is used together with execution profile information, to scan through new inputs and compute an estimation of execution time spent in specific regions of the program for each pipeline iteration. This phase is designed such that it can be executed in parallel to the original program and the live inputs presented to the program can be analyzed. A dynamic pipeline balancing and scheduling approach can control the prediction scope by controlling the amount of buffering done before presenting to the actual application.

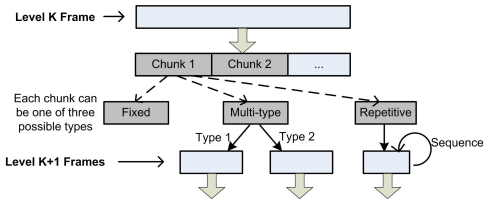

Figure 3: Hierarchical format of streaming inputs.

4.2 Input-based Execution Characterization

As described earlier, the main goal of the first phase in our system is to automatically generate an execution characterization using offline analysis. The main output of this phase is an execution characterization graph (ECG). The ECG is constructed to serve two primary purposes. First, it represents all control-flow operations of the program that use input-based conditions. Second, the graph allows each element of any input sequence to be identified uniquely. This enables parsing of any input sequence given to the program and locating input elements in the entire streaming input that are used in the conditions, thereby helping in determining which execution paths are taken at the control-flow points.

Since our execution characterization contains all input-based simple conditions in the program, it captures the input format of a streaming application being analyzed as a generalized hierarchical tree-like structure (Figure 3) that is based on the control-flow of the application. We call each node in this hierarchical format a *frame*. Each frame is made of several chunks, each of which designate lower level frames. At the topmost level, the level-0 frame is the entire input sequence. This frame can be broken up into several level-1 frames, and so on and so forth. Each chunk in a frame can be one of three different types- fixed, multi-type, or a repetitive sequence. A fixed chunk contains a single lower level frame, which is data of a static form. A multi-type chunk will have one of several different possible sub-frames. Finally, the third type of chunk may contain a repetitive sequence of the same type of sub-frame. The format of each sub-frame can be recursively defined to be a sequence of chunks. We say

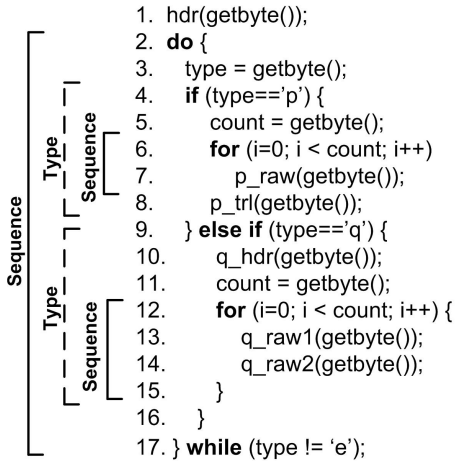

Figure 4: Running example of a streaming program.

that this format is directly generated from the control-flow of the application because simple input-based conditions used in branches are responsible for the multi-type and repetitive chunks.

We use the simple streaming application in Figure 4 as a representative running example. The program reads in a streaming input at the byte-level using the `getbyte` function that abstracts the input reading and buffering logic. The streaming input contains a header that is processed by the `hdr` function. The program processes a sequence of frames that can be of three possible types, which is identified by the first byte of each frame. The p-frames and q-frames, identified by the markers 'p' and 'q', respectively, contain data processed by the application. A special frame containing a single marker 'e' ends the input sequence. Both p-frames and q-frames contain chunks that are sequences of sub-frames whose length is determined by the inputs. The functions `p_raw, p_trl, q_hdr, q_raw1`, and `q_raw2` are functions with self-illustrating names that process various portions of the two types of frames.

4.2.1 Analysis Approach

In order to identify conditions in the program that are affected by inputs, we need to follow the propagation of input elements throughout the program. For this reason, we

Algorithm 1 Input Characterization Graph Generation

Initialize: $level = 0$, $index = 0$, $stack = \{\}$
Graph H has only one node st, and $u = st$
for each instruction $i \in T$ **do**
  Let $curbb \in V$, such that $i \in curbb$
  if $curbb$ enters a new loop body or conditional code **then**
    push $index$; $level = level + 1$; $index = 0$
  end if
  if $curbb$ exits a loop body or conditional code **then**
    pop $index$; $level = level - 1$
  end if
  if i parses new input **then**
    add new node v to H with FRI $level[index]$
    add new edge (u, v) to H with $cond(prvbb, curbb)$
    $index = index + 1$; $u = v$
  end if
  $prvbb = curbb$
end for

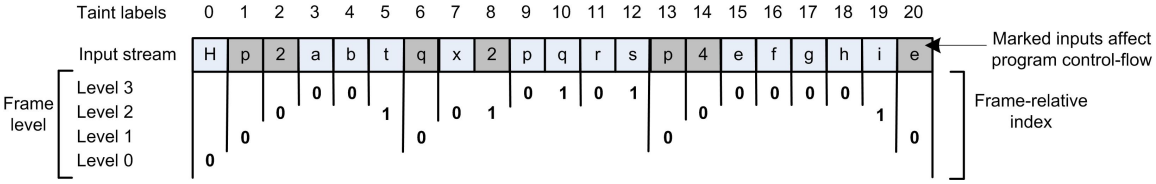

Figure 5: Example input processing.

base our analysis on the dynamic taint propagation [14] technique. We use multi-label tainting, which means that each memory address or CPU register may be tainted with one of several possible taint labels. We do this so that each input element can be given a unique taint label. In order to mark taint sources, calls to library functions that perform inputs (such as fread) can be intercepted to place taint labels on the returned buffer. We define taint propagation rules similar to the standard taint propagation approaches. If any source operand of an instruction is tainted, its target operand also becomes tainted. Taint is cleared when the target operand of an instruction is set to a specific value regardless of the values of the source operands. In case of source operands having different taint labels, we remove taint label from the target as well. This always ensures that any register or memory address can only have a single taint

label. As a side-effect, our approach can only identify *simple conditions*, or conditions that involve one element of the input only. We run the program with the given input and take an instruction-level trace, where an intermediate representation (IR) of each low-level instructions is available. Each IR instruction contains values of the operands as well as their taint labels. This self-contained IR trace enables subsequent off-line control-flow and dataflow analysis, which we describe below.

We introduce a few formal notations to help describe our algorithm. Suppose that the streaming input provided to the program is of length n. Let I be the set of all possible IR instructions. We are interested in instructions that process inputs. By processing, we mean that the instruction has performed an operation that is not just copying a source operand to the destination. For example, the function `getbyte` in the provided example is used to copy an input element from the input buffer to another variable. We consider such operations as moving data around, but not performing any useful processing on them. We use the formal function $label : I \rightarrow \{-1...n-1\}$ to indicate whether an instruction has *processed* any operand that is tainted. In case an instruction $i \in I$ does not process any tainted operand, $label(i) = -1$. The first instruction in the trace that processes an element in the input is said to *parse* the input. Therefore, an input element can be parsed by at most one instruction in the trace, but it can be subsequently processed several times. We specify a rule where if more than one source operands are tainted and they posses different taint labels the destination becomes untainted. Combining several inputs together is usually an indication of input processing that is irrelevant to our input characterization goals.

Suppose that $T = (i_1, i_2, ..., i_k)$ is the gathered k-length instruction trace, where $i_j \in I$ for $1 \leq j \leq k$. We use the addresses of each instruction in the trace and the control-flow semantics of the instructions to build dynamic control-flow graphs representing the executed paths of the program. Although in our analysis system, we take inter-procedural control-flow semantics into account and generate a CFG for

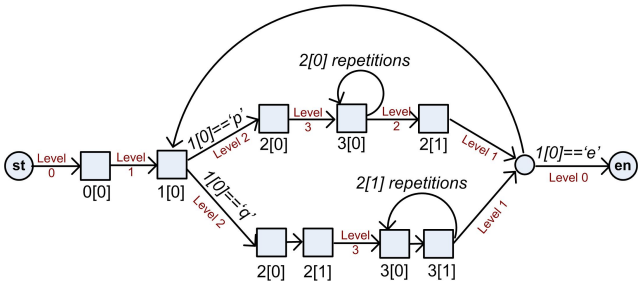

Figure 6: Input characterization graph for the input example.

each function in the program, for simplifying our algorithm description, suppose that the entire program is represented using one CFG $G = (V, E)$ where V represents the set of basic blocks and E the control-flow edges. Each basic block $v \in V$ is a set of instructions forming a subsequence in the dynamic instruction trace T because the same instruction of the program may be visited repeatedly at different points in execution. Building a CFG representation for the executed paths enable us to use loop detection [1] and control-flow dependence [5] analysis methods used in static analysis. Suppose that L represents the set of all loops identified using the loop detection analysis. Each member loop $l \in L$ is the set of basic blocks contained in its body, hence $l \subseteq V$. In our notation, a loop $l_1 \in L$ is nested inside another loop $l_2 \in L$ if $l_1 \subset l_2$. We use the function $dep : V \rightarrow V^*$ to represent control-dependence relation. A basic block $b_1 \in V$ is control dependent on another basic block $b_2 \in V$ if $b_1 \in dep(b_2)$. By using dep transitively, we can identify *conditional code*, or code that is contained in the `if` or `else` block under a conditional branch.

Algorithm 1 presents our approach of generating input characterization graph H. The execution characterization graph can additionally can generalize the input format and can be used to parse new inputs given to the program. Every node in the graph is a place holder for a unique member in a particular frame type. Each node has a frame-relative index (FRI) in the form $x[y]$ where x represents the nested level of a frame in the input sequence, and y represents a unique index for the data element in the frame being parsed. If there are multiple edges coming out of a node, each node will have a

condition. Conditions specify when the edge is to be taken when parsing the input. We use the function $cond(u, v)$ to denote condition used in an edge (u, v) in the CFG G if u contains a branch instruction that uses a tainted operand. Our approach is to use backward slicing on the branch instruction to construct the condition used at that point. We limit the construction of conditions to use elements of inputs that are contained in the frame being parsed and all parent frames containing it. In other words, an edge coming out of a node with index $x[y]$ can have conditions that may use nodes $x'[y']$ where $x' \leq x$.

The algorithm works as follows. Initially, the graph H only contains a single node - the start node st. The *level* in the input hierarchy as well as the *index* of the input element in the frame are set to 0. The stack is also initialized to be empty. The algorithm iterates through each instruction in the trace. For each instruction, the current basic block is identified in the CFG. If the basic block enters a loop or a conditional code region, the current *index* is pushed on to the stack, and it is set to 0 after increasing the *level*. If the current basic block exits from a loop body or reaches the post dominator of the last conditional code, then the *level* is reduced and the last index is popped from the stack. Finally, if the instruction parses a new input, then a node is added to H with FRI $level[index]$. New edges are added to the node with the determined condition if one exists in the corresponding control-flow edge. We limit the construction of the conditions to use elements of inputs that are contained in the frame being parsed and all parent frames containing it. In other words, an edge coming out of a node with index $x[y]$ can have conditions that may use nodes $x'[y']$ where $x' \leq x$.

As an example of how the algorithm works, we describe how a dynamic trace of the example program parsing the input provided in Figure 5 is analyzed to extract the ECG in Figure 6. When the input 'p' is processed in line 4, the input is assigned index 1[0] and an edge with the condition "1[0]=='p'" is added to the graph. Since

another loop is entered at line 6, *level* is set to 3 and $index = 0$. Each input element processed in this loop is assigned an index 3[0]. Notice that the loop iterates `count` number of times, which can be deduced from the loop condition and the induction variable increase of 1. The loop condition processes the input with taint label 2, which is assigned index 2[0]. A loop edge is added to the graph with 2[0] defining in the number of loop iterations. Upon exiting from this loop, the input parsed at line 8 of the program is given index 2[1]. During the next iteration of the outer loop, the input element 'q' causes another execution path to be taken by the program. A new edge with condition "1[0]=='q'" is added to graph that represents this execution path. Similar to the previous frame, the inputs that are parsed for this frame are added subsequently to the graph. The frame processing ends when the input at offset 20 is reached and the loop exit condition at line 17 is satisfied. In this case, an edge is added to the characterization graph with this condition. The resultant input characterization graph contains nodes for each input element in the top level and inner level frames parsed by the example program, and the edges correspond to control-flow edges in the program.

4.2.2 Correctness and Completeness

There are two issues involved in characterizing the input sequences to the streaming application. The first problem is related to the accuracy of our analysis in extracting the input parsing logic. Elements in the inputs may sometimes be processed out of order even though they are read sequentially by functions such as `getbyte`. In these cases, the index assigned by our analysis may be out of order as well. In order to handle such situations, we rely on an index correcting phase that utilizes the input stream offsets (taint labels) to correct the order of the determined frame-relative offsets. This phase can be performed every time the analysis moves from a lower level index to a higher one.

Algorithm 2 Execution Time Estimation

Input: Collapsed CFG $G' = (V', E')$, ECG H, Time $T(r)$ for regions $r \in V'$, input $X = x_1 x_2 ... x_n$
Initialize: $ET(p_i) = 0$ for $p_1, p_2, ..., p_k \in V'$, .
for each input element x_i **do**
  Traverse edge in H corresponding to x_i
  if condition found **then**
    identify path in CFG to this condtion
    for each region r in path **do**
      identify $p \in P$ where r is in
      $EP(p) \leftarrow ET(p) + T(r)$
    end for
  end if
end for

The second issue is regarding the completeness of our analysis results. Since we are using dynamic analysis, the inputs might not cause execution of feasible paths in the program that parse inputs. As a result, the graph may contain only a subset of the possible input formats supported by the program. A useful property of our constructed graph is that parsing will explicitly fail while processing such inputs in our system. By adding these failing inputs, a more complete input characteristics can be generated.

4.3 Dynamic Behavior Prediction

In this section, we describe the second phase of our system, which performs the actual prediction of the execution time of specific regions of the program for a new input stream. Besides determining the control-flow choices made by the program using the execution characterization graph, the predictor requires profile information of regions of the program that correspond to edges in the ECG. Section 4.3.1 describes how a collapsed CFG is generated to acquire the required profile information. Then, in Section 4.3.2, the actual execution time prediction method is described. Finally, Section 4.3.3 discusses the algorithmic complexity of our proposed algorithm.

4.3.1 Collapsed CFG and Profile Generator

At the high-level, the execution time predictor walks the ECG while scanning the input stream. The edges selected during this walk, designates what execution paths are taken in the actual CFG. However, in order to estimate the dynamic execution time, we need profile information of regions of code that correspond to edges in the ECG H. We acquire this by generating a collapsed control flow graph $G' = (V', E')$ from the control-flow graph $G = (V, E)$ of the program. First, we mark the ege in the CFG G that has a corresponding edge in the ECG H with an associated condition. We start with the collapsed CFG G' that is identical to the CFG G. We then repeatedly identify a strongly connected component of G', and collapse it into a single new node. A strongly connected component G' has only one incoming edge and one outgoing edge and no marked edges between any of its nodes. Each node in the collapsed control-flow graph is a *region*. If u is the node with the incoming edge and v is the node with the outgoing edge, then we represent the new node r as the tuple (u, v) because they can uniquely represent the region. Moreover, the execution of the region begins when execution enters u and ends when execution leaves v. We can construct regions by iteratively going through the adjacent basic blocks of a region and adding it if they are connected by an unmarked edge. In cases where a region consists of only one basic block, say u, we denote it as a region (u, u). In the end, the collapsed control-flow graphs are sufficiently condensed having edges that have corresponding edges to the input characterization graph.

We now determine the average execution time for each identified region. This is done by instrumenting the program at the entry and exit of all regions to log clock readings. Then the instrumented program is executed with several inputs to gather profile information. We use $T(r)$ to denote the average execution time of a region $r \in V'$.

4.3.2 Execution Time Prediction Approach

Without loss of generality, let $p_1, p_2, ...p_k$ be k partitions in the program where each partition $p_i \subseteq V'^*$ can be defined as a set of regions in the collapsed control-flow graph. Now, given a collapsed control-flow graph G', input-characterization graph H, and the profile information $T(r)$ for each region $r \in V'$, we present our algorithm (Algorithm 2) to determine the estimated execution time $ET(p_i)$ spent in each partition p_i for processing an input stream $X = x_1x_2...x_n$. We start by initializing the execution time of all partitions as 0. Initially, we set the current node in the graph H to be ST and in the collapsed CFG G' to be the empty node. We proceed by taking an input x_i from the stream, traversing an edge in the graph H and assigning the input element with an FRI specified in the next node.

If the outgoing edges from the current node have conditions, we take the edge for which the condition is satisfied. If during a traversal, an edge with a condition is traversed, we identify the corresponding marked edge in G', which must exist according to our constructions. Since x_i is being parsed, there of course has to be a corresponding node in H that takes an input. We assign the FRI to the input element during this traversal. We walk the graph G' from the last point till the marked edge. There should only be one such path in the graph G' because there can not be any node with multiple outgoing unmarked edges. For each region r in the path, we identify the partition $p \in P$ such that $r \in p$, and then add the execution time of the region to the partition. In other words, we set $ET(p) \leftarrow ET(p) + T(r)$. In this manner, we proceed to parse the next sequence of inputs until another edge is reached in H with a condition, then walk G' again until we reach another marked edge and update partition times accordingly while traversing the CFG G'.

4.3.3 Run-time Complexity Analysis

The presented algorithm can be shown to have run-time linear to the streaming input length when only conditional branches to the regions containing input parsing code are considered. In this case, the predictor can scan a new input stream and generate execution time predictions for the program in $O(nt)$ time, where n is the input length and t is the maximum number of types for any multi-type chunk in the input sequence. This is because each input element will always require at most one edge traversal in the graph. Since a node may have a maximum of t outgoing edges, the condition in each edge need to be evaluated. Since we are only considering simple input based conditions (where only one input element is in the condition), evaluation of a condition will take constant time.

While constructing the ECG, if we only consider input parsing branches and loops, we sacrifice the accuracy that can be achieved by our system. Since any conditional branches or loops that may have conditions that depend on the input but do not parse any new data will not be contained, the regions corresponding to them in the CFG will be collapsed. This means that even though we could be able to reason about the execution paths, their average execution times will be taken into account by our algorithm. This reduces the accuracy of our analysis in such cases. For example, a program may parse a sequence of inputs and store it in an array, so that latter it is processed in another loop. If the latter array processes the data stored in the array and its number of iterations depend on the input sequence length, the opportunity of estimating an accurate timing by considering the loop is missed. We solve this problem by modifying the input characterization graph construction to include edges corresponding to branches and loops with conditions similar to what we already handle, but do not contain any parsing instructions. The resultant algorithm is no longer in $O(nt)$ complexity, but with the extra cost more precision in execution time estimation can be gained.

4.4 Precission Analysis

In Section 4.3.3, we briefly mentioned how we constructed the ECG so that the execution time prediction for new inputs take time linear to the length of the input. This was achieved by limiting the computation for each input element to a constant time. This performance is achieved, however, at the price of precision. In this section, we describe methods to improve the precision of the system, and investigate its effects in terms of prediction overhead, memory usage and accuracy.

4.4.1 Reasons for Imprecision

The algorithm described in Section 4.3.2, works with an ECG where for each input element, there can only be one transition. The transition is one among the maximum number of possible transition selections, which is equal to the maximum number types in a multi-type chunk for the application. The implementation can be done in such a way that a hash table is used to select the correct transition, giving it a constant time.

There were two limitations that were imposed on the construction of the ECG, which keeps the algorithms complexity to constant time for each input element. First, we are only considering simple conditions when looking at branches and conditional statements. By simple conditions, we only include conditions that have one operator and one input element. Therefore, it may be a combination of a constant and an input element, but cannot have two input elements. In addition, there cannot be logical operators connecting more than one conditional expressions. Second, we are not taking any conditions in the program that are processing inputs rather than parsing. This means that if an input is being used in a condition that has been previously processed by an instruction, we are disregarding it. For example, if there is a loop in the program that is using some data that has been buffered in memory and later parsed, this will be missed.

The program sections that contain conditions that are not included eventually become collapsed in the collapsed CFG. Therefore, the execution time deduction algorithm takes the average execution time for these section. In other words, even if the execution time varies with input values, the average time that is computed by the profiling approach is only taken. The imprecision rises because the average time is taken instead of predicting the execution time induced by the inputs.

4.4.2 Methods for Improving Precision

We focus on the two imposed limitations we mentioned and discuss how we can improve them.

4.4.2.1 Handling Complex Conditions

In order to handle complex conditions, we need to construct the conditional expressions as they are computed using low level instructions in the program. For example, a condition such as $(a > 10)\&\&(b! = 0)$ is first computed by comparing the variable a with 10, storing the result in a temporary register or memory location t_1. Then the result t_2 of comparing b and 0 is applied with the previous result using a logical *and* operation to get the final result t.

One way to handle complex conditions may be to use *Symbolic Execution* [10]. Using this method, it is theoretically possible to generate symbolic formulas for register or memory contents based on the program inputs by simulating each instructions execution. The conditional expressions used in the branch or loops, which are also computed using low-level instructions, can also be constructed using the symbolic execution approach. However, there are two significant drawbacks. First, for real world streaming applications, the symbolic formulas may become extremely large, requiring a lot of memory. This will happen if several inputs are combined using different arithmetic computations. Moreover, symbolic execution limits the number of times loops are iterated to keep formulas from growing unmanageably. This also makes

results inaccurate. Therefore, employing pure symbolic execution is inappropriate for our solution.

In our efficient approach, if both a and b are derived from two separate inputs, t_1 and t_2 will receive two separate taint labels. However, when t is computed, since the operands have two different taint labels, the taint is removed. This limitation imposed on the taint propagation filters out any complex conditional expressions from being handled. In this case, for our purpose, we have to allow the combination of multiple taint sources and labels and still track the propagation of these values. However, we have to achieve this by keeping a restraint in memory usage for efficiency.

We improve our base approach by combining a limited form of symbolic execution with taint propagation. The limited form of symbolic execution actually does not symbolically execute the program, but rather generates symbolic formulas for specific types of values being computed by the program. We describe the additions to our base approach below:

1. **Modifications to Taint Propagation:** We introduce a new set of taint labels. The new set represent that the register or memory holds a *symbolic value*. The original taint labels represent that the register or memory holds an *input value*. In a system having 32-bit taint labels, a single bit can be utilized to indicate which type of taint label it is. In this approach, the system should be able to handle approximately 2^{31} or 2 billion inputs and symbolic values. The same taint is propagated to the destination as long as the instructions are simple `'mov'` operations that copy source values to target locations. Whenever an instruction is executed that processes the operands, and any one of the operands is either an input value or symbolic value, a new taint label is generated to indicate a new symbolic value and the destination is tainted with this label.

2. **Selective Symbolic Expression Generation:** Since generating symbolic expressions for destinations of all possible instructions is prohibitive, we selectively generated symbolic expressions. We only considered instructions that generate expressions that are usually used in conditions. Therefore, instructions that correspond to *relation operators* such as '>', '<', '==', etc., or *logical operators* such as '&&', '||', etc. were considered. Symbolic expressions were built in the same method as compilers build *abstract syntax trees*. Any symbolic expression would be a root of a tree with at most two nodes connected that may be a constant, an input (when the operand has an input value taint label), or another symbolic expression root (when the operand has symbolic value taint label). In general the expressions can be accessed in constant time by using a hash table where the symbolic value labels are the keys. If a symbolic expression is being generated by an instruction that is avoided by our algorithm, its existence is denoted by a special taint label.

The algorithm of propagating taint and construction symbolic expressions is given as Algorithm 3, which is applied when each instruction is executed. Assume that the taint labels for register or memory element p is given by $T(p)$. All taint labels are represented by a set L, where $L = L_I \cup L_E \cup \{l_\phi, l_\Gamma\}$. Here, $L_I \subset L$ is the set that labels inputs. Each input element $x_i \in X$ has a taint label $l_i \in L_I$. If a register or memory element p has no taint label or symbolic expression associated with it then $T(p) = l_\phi$. The set $L_E \subset L$ corresponds to the labels representing a particular symbolic expression. The function $E : L_E \to \Sigma$ is used to map symbolic expression taint labels to symbolic expression set Σ. Finally, the taint label l_Γ is used to indicate a symbolic expression whose construction has been avoided. Symbolic expressions can represent a constant, an input element, or two symbolic expressions connected by an operator. By mapping taint labels to symbolic expressions and register or memory elements to taint labels, the memory required for keeping symbolic expressions can

be conserved. To further conserve memory, in the practical implementation of the algorithm, a reference count can be kept for symbolic expressions. If the number of references to an expression drops to 0, it can be removed from the table, thus saving memory. The algorithm goes through each source operand of the instruction and iteratively builds a single taint label for the destinations, which is then placed onto each of the destination operands where to the value generated by the instruction is explicitly stored.

In the described method, although conditional expressions will be symbolically represented, not all possible conditions will be recognized. For example, if one of the operands in the conditional expressions is arithmetically computed from several inputs, we do not have it symbolic representation and thus sufficient information to compute the result of the condition. To omit these cases, we designate a special taint label that indicates that the value was computed from inputs, but it is not symbolically represented. If an instruction corresponding to a logical or relational operator has such an operand, the result is also is kept the same taint label and for this case a symbolic expression is not generated. Although we may miss several complex conditions with this approach, it does make the approach less memory intensive and at the same time allows us to include a large portion of complex conditions used in programs.

Another limitation of the approach is that while expressing inputs in the conditions, the frame relative index (FRI) has to be used instead of the absolute index of an input element. This is a required property while building the ECG allows the parsing of inputs in a general way for the given application. Therefore, if there are input elements that contain inputs not representable by their FRI because it clashes with another input element, the expression has to be omitted.

Algorithm 3 Taint Propagation Algorithm for Improving Precision

Input: Source and destination operands as $S = \{s_1, s_2, ..., s_k)$ and $D = (d_1, d_2, ..., d_l)$, Taint and symbol maps T and E.
Initialize: Destination taint $t = l_\phi$.
if instruction is candidate for taint propagation **then**
  if $T(s_i) == l_\Gamma$ for any i **then**
    $t = l_\Gamma$
  else
    if $T(s_i)! = T(s_j)$ for some i, j **then**
      $t =$new label $t \in L_E$
      $e =$new expression $e \in \Sigma$
      $E(t) = e$
      for each source operand s_i **do**
        $e = e \cup E(T(s_i))$
      end for
    else
      $t = T(s_1)$
    end if
  end if
else
  if $T(s_i)! = l_\phi$ for any i **then**
    $t = l_\Gamma$
  end if
end if
for each destination operand d_i **do**
  $T(d_i) = t$
end for

4.4.2.2 Building More Comprehensive ECG's

The next method for increasing precision is to build a more comprehensive ECG and include control-flow information that we did not include in the efficient version of our approach. Unlike the original efficient approach, we will not disregard branches or loops that do not contain any input parsing instructions. Therefore, many branches and loops that do not parse any new inputs and process previously parsed inputs will be considered. Although this may seem to have an ECG that will be as large as the program's CFG, but this will not be the case. We are still only considering branches and loops where at least one element is tainted with a label l where $l \in L_I \cup L_E$. In

other words, the condition itself should involve any element that contains an input value or contains an expression whose symbolic representation exists in our system.

The advantage of this approach is that we are considering any branch in the program that is affected by input and which can at least be represented by the symbolic expressions generated by our system. Although the symbolic expressions are limited forms of conditions, they should include the vast majority of complex conditions found in programs that directly use an input. When the run-time execution is predicted, additional regions that were collapsed in the efficient version will be taken into account, thus allowing more precise timing based on the given input.

4.4.3 Run-time Complexity

The new approach generates complex conditions and also includes loops and branches where input may not be parsed but processed as well. Therefore, it is obvious that it is no longer a linear algorithm. While a new input is provided to the predictor, for every input element, there may be several traversals in the ECG. In other words, the algorithm for run-time prediction will focus on traversing the ECG with the help of the new input rather than taking single transitions in the ECG for every input element. The complexity of the run-time prediction algorithm will, therefore, be bound by the ECG's size. Directly deriving a run-time complexity based on input length is hard because an ECG can in the worst case be equivalent in size to the CFG of the program, which can be arbitrarily large compared to the programs input.

4.5 Experimental Evaluation

In this section, we present our implemented system and the experimental results evaluating the effectiveness of our dynamic behavior prediction approach. We present detailed case studies on four benchmarks comparing the estimated dynamic execution times compared to actual execution time variations. We also present evaluation of accuracy and overhead of the execution time predictor of the actual program. Finally,

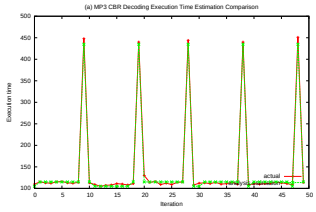

Figure 7: MP3 decodeblock behavior for CBR

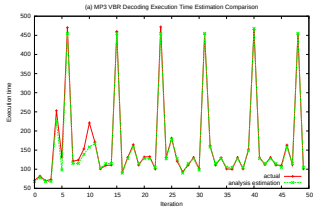

Figure 8: MP3 decodeblock behavior for VBR

we present both simulated and real-world speedups when using dynamic pipeline balancing and scheduling equipped with our dynamic behavior estimation system vs static approaches.

4.5.1 Implementation

We developed our dynamic analysis tool using the PIN instrumentation framework. For our analysis, we targeted compiled binary programs for the IA-32 architecture on Linux. For the intermediate representation of low-level x86 instructions we used *libdisasm* library instead of the IR access provided by PIN.

We constructed our taint analysis system over the instruction-level tracing mechanism provided by PIN. We enabled byte-level taint labeling and propagation. We maintained 32-bit taint labels for CPU registers as well as each memory address used by the analyzed application using a hash-table. For marking taint sources, we intercepted calls to C library functions such as `fread`. The offset in the file pointer is maintained in our analyzer to apply labels to the returned data-buffer in memory. Taint was propagated for each IR instruction by examining the taint labels in the source operands and propagating them accordingly to the destination operands. We handled all memory addressing modes, so that results obtained from table lookups using tainted indexes are also tainted. This helps decoded values from inputs to also be appropriately tainted. However, we did not consider taint propagation by implicit flows. In other words, data assigned in execution paths taken by tainted conditions are not tainted. The IR trace together with the operand values and taint labels were output to a log file for later analysis.

Since most streaming applications perform bit-level reading of inputs, using our byte-level tainting directly would cause a lot of imprecision in our analysis. Rather than taking the mammoth engineering task of creating a bit-level taint engine, we

took the path of providing information regarding the bit-level streaming input function (such as GetBit in MP3 decoder) in an application to our analysis by manually identifying them. This information is used to disregard the bit-level masking operations in the these functions as input processing. They are treated as regular data-moving operations for taint propagation.

In order to estimate the execution time for new inputs using our input characterization information, we require the profile information regarding the execution time spent in each iteration of specific loops and branches of specific condition in the program. After identifying the required code sections in the offline analysis, we used PIN to instrument the program on a fresh run. In this case, we disable instruction level tracing, but inserted `RDTSC` instructions at the beginning and end of the required program points to gather time spent in execution those code sections. We then averaged each pass though the code regions to get the baseline execution times to perform the deduction.

4.5.2 Execution Behavior Prediction Case Studies

For experimental evaluation of our dynamic execution behavior prediction approach, we applied our system on four benchmarks - MP3 decoder, MPEG-2 decoder, bzip2 and an MPEG-4 decoder. We first used our offline execution behavior characterization to generate the ECG for all benchmarks. For each benchmark, we provided input sets containing 10 different input files. Table 6 shows properties of each of extracted ECG's. It can be seen that simpler streaming formats tend to have smaller input characterization graphs. The graphs also show that the number of marked edges for MP3 and bzip2 are significantly smaller than MPEG-2 and MPEG-4, highlighting the fact that those input formats have little dynamism in the streaming formats.

Once the ECG's were extracted, we used our system to generate the collapsed CFG's and the profile information for the collapsed code regions of each benchmark.

Table 1: Extracted input characterization graphs

Benchmark	Nodes in ECG graph	Max. levels in ECG graph	Marked C-flow edges
MP3	12	3	6
MPEG-2	79	8	21
bzip2	15	2	8
MPEG-4	211	13	58

Table 2: Accuracy measurement for predicted execution times

Benchmark	Average error rate
MP3 (CBR)	5.2%
MP3 (VBR)	5.9%
MPEG-2	5.0%
bzip2	4.8%
MPEG-4	4.1%

We then used our execution time predictor for each program to generated predicted estimates of the execution time for each program to generate predicted estimates of the execution time for each pipelined iteration on a new input. For MP3, we provided two different inputs - one with VBR and the other with CBR. For MPEG2, we generated estimates for two different phases - the decoding block and the motion compensation stage. In order to obtain the actual execution time measurements, we executed each real program with the same inputs we provided to the predictors. Table 2 shows the measured error rate of our estimates compared to the actual execution timing we measured. The results show that our predictions were sufficiently accurate. In the following subsections, we discuss our experiences with generating predictions of the execution timing and the observed dynamic behavior of the program in details.

4.5.2.1 MP3 Decoder

We took the MP3 decoder benchmark program and pipelined the loop that processes each MP3 frame. We specified 6 partitions in the loop and validated the correct data-flow between them to ensure there are no cyclic or backward dependencies in

the stream graph.

We used our execution time prediction system to parse the two MP3 files to estimate the execution time required for the different partitions for each iteration of the pipelined loop. The first MP3 file was created using the default a constant bit-rate encoding (CBR), while the second file was encoded using a variable bit rate (VBR) encoding scheme. Among the 6 different partitions defined, the first phase, which is the input decoding phase showed the most noticable variation patterns. The results of the estimated times and the actual execution times for 50 iterations of the pipelined loop of the MP3 decoder application are shown in Figure 7 and Figure 8. The comparison shows that our estimation was sufficiently accurate in detecting the variations of execution timing. Investigation revealed that our system was able to identify that the bitrate field in the header identifies the length of the raw data in each MP3 frame, and a branch that is executed after a fixed number of data elements are read in (the reason for the spikes). For both the CBR and VBR encoded files, they were enough to reach a high accuracy in predicting the dynamic patterns.

4.5.2.2 MPEG-2 Decoder

For the MPEG-2 decoder, we partitioned the slice processing loop into 8 stages. We then estimated the execution timing for the stages for processing a new .m2v file. After gathering the results we found the most variation in execution time in the decoding and motion compensation phases. We show real and predicted execution behavior for 40 iterations at a randomly chosen starting in Figure 9 and Figure 10, respectively. For the decoding phase, a lot of small variations and a few large spikes were visible, all of which were predicted by our approach with sufficient accuracy. After investigating, we identified that the spikes were caused by I-frame decoding, which contained significantly more macro-blocks than P-frames. Also, the smaller

variations were due to the amount of data contained in the P-frames, which proportionally affected execution time due to loops. For motion compensation phase, the primary reason for variation detected by our approach was due to a branch not being executed for intra macroblocks, that are specific only to I-frames. We were able to achieve an average error rate of only 5% for both these scenarios.

4.5.2.3 bzip2

For the bzip2 benchmark, we initially used two partitions in the main uncompress loop. The first partition performs decode of move-to-front values and second one performs undo reversible transformation and CRC checks. Since two partitions were low in number, we further partitioned the first phase into four additional stages after inlining the function. Almost all partitions showed variations in execution time. We present the execution behavior of the undo reversible transformation partition in Figure 11. The variation were mainly caused due to the length of RLE data supplied in the input stream that controlled the number of loop iterations, which our system was able to successfully detect.

4.5.2.4 MPEG-4 Decoder

For testing MPEG-4 decoding, we utilized the `x264` library and its sample decoding application. We first identified the loop for decoding macro-blocks and inlined several functions to place 9 partition boundaries. We then ran our analysis to and also gathered the actual execution times for comparison. We identified the highest variation in the stage that performs the decoding of arbitrary shaped video objects (AS-VO). The results for the actual and estimated execution time for this partition for each iteration of the pipelined loop are shown in Figure 12. The results show that our estimations showed little error rates for this case as well.

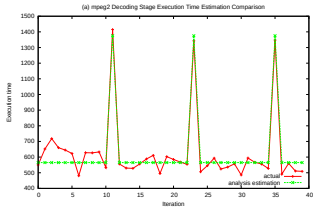

Figure 9: MPEG-2 decoding partition execution behavior

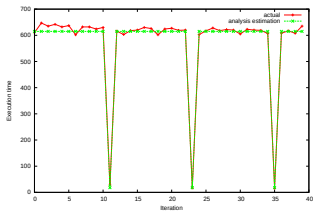

Figure 10: MPEG-2 motion compensation execution behavior

Table 3: Prediction overhead measurements

Benchmark	Exec. time of real app	Exec. time of predictor	Predictor overhead
MP3 (CBR)	7.3s	0.4s	5.5%
MP3 (VBR)	12.8s	0.8s	6.2%
MPEG-2	121.4s	5.9s	3.2%
bzip2	24.2s	1.1s	4.5%
MPEG-4	269.9s	9.9s	3.6%

4.5.3 Prediction Overhead Analysis

One of the main goals of designing our execution time predictor was to keep the overhead of scanning an input sequence and generating the execution time estimates very low. We evaluated the overhead of our execution time predictor by comparing the total execution time of the actual streaming applications and our execution time predictor on the same inputs. Table 3 shows the results. On average we found that predictor required only a fraction of the execution time of the actual program to generate the execution time estimates. In the worst case, the overhead was only 6.2% over the actual program.

4.5.4 Simulated Speedup Analysis

In order to evaluate the usefulness of predicting dynamic behaviors in streaming programs with our input-driven execution behavior estimation approach, we simulated dynamic pipeline balancing of the streaming program's pipelined partitions on a multicore system. In our simulation, we assumed a multicore setup such as mainstream quad-core systems from Intel or AMD. We consider a pipeline balancing approach in which different partitions are executed using separate threads that share the same address space, where the cost for communication or transferring data among the partitions can be neglected.

For simulating the execution of the partitions on different cores, we first gathered the actual execution time for each partition on each iteration of the pipelined loop.

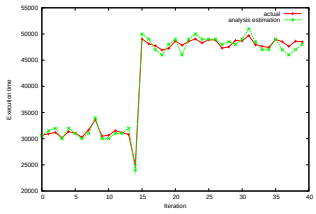

Figure 11: bzip2 undo-reversible transform execution behavior

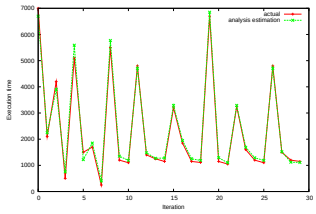

Figure 12: MPEG-4 arbitrary shaped video objects behavior

For static pipeline balancing and scheduling, we took the average execution time for each partition and used the greedy method to balance the loads on the four cores and assign the partitions to the cores. We then simulated the execution of each iteration of the pipelined loop, and computed the iteration time as the highest execution time required for any core. In this manner, we determined the total parallel execution time and used the ratio with the serialized execution time to compute the speedup.

For dynamic balancing, we added the input parsing time for each iteration as an additional partition to be executed along the partitions belonging to the original program. The difference between static vs. dynamic is that we periodically allowed the partition assignments to the cores to be redistributed based on the estimated execution times output by our system. The threads or processes on multicore systems in our assumed setup can be pinned to particular cores with the help of the OS. Since such core assignment to threads do not require any noticeable overhead, we did not add any dynamic redistribution cost in our simulation. Therefore, the speedup results show an upper bound of achievable speedup for dynamic pipeline balancing.

The results are shown in Table 4. For MP3, dynamic balancing was not able to achieve any gains in speedup. The reason is that the partitions that showed variations were had an average time that were insignificant compared to the other partitions. Any benefits were nullifying by the cost of predicting dynamic variations. In all other benchmarks, dynamic balancing showed significant benefits over static. Especially, for MPEG-4, which contained a lot of dynamic variations in the inputs and execution time, the speedup gains were almost 40% over the static counterpart.

4.5.5 Real-world Speedup Analysis

While simulated speedup results in the previous section provide an estimated upper bound of the speedup that may be obtained through dynamic pipeline balancing and

Table 4: Simulated speedup - static vs dynamic balancing

Benchmark	Static balancing	Dynamic balancing	% improvement
MP3	2.31	2.33	0.1%
MPEG-2	2.15	2.69	25.1%
bzip2	2.46	2.99	21.4%
MPEG-4	2.56	3.54	38.3%

Table 5: Speedup - static vs dynamic balancing

Benchmark	Static balancing	Dynamic balancing	% improvement
MPEG-2	2.15	2.69	25.1%
MPEG-4	2.64	3.46	31.1%

scheduling leveraging our execution behavior prediction approach, we performed real-world speedup analysis for a more realistic evaluation. Since the goal of our paper was to demonstrate the feasibility and accuracy of our dynamic behavior prediction approach, building a complete tool that automatically generates parallelizable code supporting dynamic balancing of pipelined stages was out of the scope of our paper. Rather than taking that route, we took a more tangible approach of manually modifying the some benchmarks to insert thread generation and synchronization code for pipeline stages and dynamically assign threads to cores. We wrote a thread to core assignment routine that uses the `pthread_setaffinity_np` system call. This routine is called before each pipelined loop begins. We manually modified the MPEG-2 and MPEG-4 benchmarks and the results are shown in Table 5. The real-world speedup results were close to the actual ones. The achieved improvement in speedup was 19.6% and 31.1% for MPEG-2 and MPEG-4 - a bit lower than the 25.1% and 38.3% speedup achieved in simulation results. In any case, the results show the viability of our prediction approach's use in a dynamic balancing system of streaming applications.

Table 6: Extracted more precise ECGs

Benchmark	Nodes in ECG graph	Max. levels in ECG graph	Marked C-flow edges
MP3	81	4	38
MPEG-2	130	8	41
bzip2	85	4	24

Table 7: Accuracy measurement for more precise approach

Benchmark	Average error rate (previous)	Average error rate (newer)	
MP3 (CBR)	5.2%	3.9%	23.6%
MP3 (VBR)	5.9%	4.2%	29.3%
MPEG-2	5.0%	4.3%	14.0%
bzip2	4.8%	4.1%	14.5%

4.5.6 Precision Improvement Analysis

In order to test the approach of improving precision, we implemented the improved version of taint propagation and more comprehensive ECG creation algorithms. For understanding how much complex the new ECG was with the more precise approach, we used the same set of inputs on the benchmarks. The information regarding the resultant ECGs are given in Table **??**. For our testing, however, the MPEG4 benchmark could not be analyzed with the more precise mechanism because the graphs grew bigger than the size of memory available, thus showing a limitation that the more precise approach is not applicable to all benchmark cases. In the results, it can be seen that the sizes of the graph increases and a lot of control-flow edges appear. This directly shows that more branches and loops are taken into account in the graph. These branches and loops may contain conditions that are also complex.

In order to find out how much precision had improved by using the latter approach, we took the time taken in real time for each iteration of the tested benchmarks and the time predicted for each iteration. The results are shown in Table 7. The lowest error rate that could be achieved with the latter method was 3.9% whereas the former method had given a lowest of 4.8%. However, precision improvements per benchmarks

Table 8: Prediction overhead of precise approach

Benchmark	Exec. time of real app	Exec. time of predictor	Predictor overhead
MP3 (CBR)	7.3s	6.8s	93.1%
MP3 (VBR)	12.8s	13.4s	104.6%
MPEG-2	121.4s	95.5s	78.6%
bzip2	24.2s	53.1s	219.4%

was as high as 29.3%. Although it does seem a lot of improvement, but the actual average error rate of the predictions became 4.2% instead of 5.9%.

Finally, we show the prediction overhead measurements for the newer more precise approach that we introduced. The results showed a huge amount of overhead ranging from 78.6% to a almost 220%. This shows that although we could gain a tiny amount of precision through the new method, but the new ECG and algorithm for prediction is not efficient enough to be used for dynamically scheduling workload on multicores.

Some other research was once performed.

CHAPTER V

CONCLUSION

In this thesis, we presented a novel approach for estimating dynamic execution behavior in streaming applications by only observing inputs. Our approach for extracting input characterization has been shown to be applicable to real world multi-media stream applications with complex input formats. Our results show that significant execution variations can be found in many programs specially, for those that support newer and advanced compression standards. By being able to predict such variations with our approach, dynamic pipeline balancing and scheduling shows promise as a viable method to gain noticeable speedups over static scheduling. Future research on dynamic balancing techniques may make the approach a practically useful better alternative for executing pipelined programs on multicores.

REFERENCES

[1] Aho, A., Lam, M., Sethi, R., and Ullman, J., *Compilers—Principles, Techniques, & Tools*. Addison Wesley, 2006.

[2] Buck et al., "Brook for gpus: Stream computing on graphics hardware," in *ACM Transactions on Graphics, 23(3):777-786*, 2004.

[3] Cui, W., Peinado, M., Chen, K., Wang, H., and Irun-Briz, L., "Tupni: Automatic reverse engineering of input formats," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2008.

[4] Douillet, A. and Gao, G. R., "Software-pipelining on multi-core architectures," in *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, 2007.

[5] Ferrante, J., Ottenstein, K., and Warren, J. D., "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, July 1987.

[6] Ghamarian, A. H., Geilen, M. C. W., Basten, T., and Stuijk, S., "Parametric throughput analysis of synchronous data flow graphs," in *Proceedings of the conference on Design, automation and test in Europe*, 2008.

[7] Gordon, M. I., Thies, W., and Amarasinghe, S., "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[8] Gummaraju, J. and Rosenblum, M., "Stream programming on general-purpose processors," in *Proceedings of the International Symposium on Microarchitecture*, 2005.

[9] Kim, H., Suleman, M. A., Mutlu, O., and Patt, Y. N., "2d-profling: Detecting input-dependent branches with a single input data set," in *Proceedings of the International Symposium of Code Generation and Optimization*, 2006.

[10] King, J., "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, July 1976.

[11] kudlur, M. and Mahlke, S., "Orchestrating the execution of stream programs on multicore platforms," in *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[12] Kumar, T., Cledat, R., Sreeram, J., and Pande, S., "Statistically analyzing execution variance for soft real-time applications," in *In Proc. of the 21th Annual Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2008.

[13] Mark., W. R., Glanville, R. S., Akeley, K., and Kilgard, M. J., "Cg: A system for programming graphics hardware in a c-like language," in *Proceedings of the Internationl Conference on Computer Graphics and Interactive Techniques*, 2003.

[14] Newsome, J. and Song, D., "Dynamic taint analysis for automatic detection, analysis and signature generation of exploits on commodity software," in *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, 2005.

[15] Nickolls, J. and Buck, I., "Nvidia cuda software and gpu parallel computing architecture," in *In Microprocessor Forum*, 2007.

[16] Ottoni, G., Rangan, R., Stoler, A., and August, D. I., "Automatic thread extraction with decoupled software pipelining," in *Proceedings of the International Symposium on Microarchitecture*, 2005.

[17] Poplavko, P., Basten, T., and van Meerbergen, J., "Execution-time prediction for dynamic streaming applications with task-level parallelism," in *Proceedings of the Euromicro Conference on Digital System Design Architectures, Methods and Tools*, 2007.

[18] Rangan, R., Vachharajani, N., Vachharajani, M., and August, D., "Decoupled software pipelining with the synchronization array," in *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, 2004.

[19] Rau, B. R., "Iterative modulo scheduling: An algorithm for software pipelining loops," in *In Proceedings of the 27th Annual International Symposium on Microarchitecture*, 1994.

[20] Theelen, B. D., Geilen, M. C. W., Basten, T., Voeten, J. M., Gheorghita, S. V., and Stuijk, S., "A scenario-aware data flow model for combined long-run average and worst-case performance analysis," in *Proceedings of the IEEE/ACM International Conference on Formal Methods and Models for Co-Design*, 2006.

[21] Thies, W., Chandrasekhar, V., and Amarasinghe, S., "A practical approach to exploiting coarse-grained pipeline parallelism in c programs," in *Proceedings of the International Symposium on Microarchitecture*, 2007.

[22] Thies, W., Karczmarek, M., and Amarasinghe, S., "Streamit: A language for streaming applications," in *Proceedings of the Internationl Symposium on Compiler Construction*, 2002.