2010

EXTRACTION AND PREDICTION OF SYSTEM PROPERTIES USING VARIABLE-N-GRAM MODELING AND COMPRESSIVE HASHING

Muthulakshmi Muthukumarasamy
University of Kentucky, mmuthulakshmi@gmail.com

ABSTRACT OF DISSERTATION

Muthulakshmi Muthukumarasamy

The Graduate School

University of Kentucky

2010

EXTRACTION AND PREDICTION OF SYSTEM PROPERTIES USING VARIABLE-N-GRAM MODELING AND COMPRESSIVE HASHING

ABSTRACT OF DISSERTATION

A dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy in the College of Engineering
at the University of Kentucky

By

Muthulakshmi Muthukumarasamy

Lexington, Kentucky

Director: Dr.Henry G. Dietz, Department of Electrical and Computer Engineering

Lexington, Kentucky

2010

ABSTRACT OF DISSERTATION

EXTRACTION AND PREDICTION OF SYSTEM PROPERTIES USING VARIABLE-N-GRAM MODELING AND COMPRESSIVE HASHING

In modern computer systems, memory accesses and power management are the two major performance limiting factors. Accesses to main memory are very slow when compared to operations within a processor chip. Hardware write buffers, caches, out-of-order execution, and prefetch logic, are commonly used to reduce the time spent waiting for main memory accesses. Compiler loop interchange and data layout transformations also can help. Unfortunately, large data structures often have access patterns for which none of the standard approaches are useful. Using smaller data structures can significantly improve performance by allowing the data to reside in higher levels of the memory hierarchy. This dissertation proposes using lossy data compression technology called 'Compressive Hashing' to create "surrogates", that can augment original large data structures to yield faster typical data access.

One way to optimize system performance for power consumption is to provide a predictive control of system-level energy use. This dissertation creates a novel instruction-level cost model called the variable-n-gram model, which is closely related to N-Gram analysis commonly used in computational linguistics. This model does not require direct knowledge of complex architectural details, and is capable of determining performance relationships between instructions from an execution trace. Experimental measurements are used to derive a context-sensitive model for performance of each type of instruction in the context of an N-instruction sequence. Dynamic runtime power prediction mechanisms often suffer from high overhead costs. To reduce the overhead, this dissertation encodes the static instruction-level predictions into a data structure and uses compressive hashing to provide on-demand runtime access to those predictions. Genetic programming is used to evolve compressive hash functions and performance analysis of applications shows that, runtime access overhead can be reduced by a factor of $\sim 3x - 9x$.

KEYWORDS: Instruction-level cost models, Variable-N-Gram analysis, Power prediction, Compressive hashing, Genetic Programming

Muthulakshmi Muthukumarasamy

2010

EXTRACTION AND PREDICTION OF SYSTEM PROPERTIES USING VARIABLE-N-GRAM MODELING AND COMPRESSIVE HASHING

By

Muthulakshmi Muthukumarasamy

Dr. Henry G. Dietz
(Director of Dissertation)

Dr. Stephen Gedney
(Director of Graduate Studies)

2010

RULES FOR THE USE OF DISSERTATIONS

Unpublished dissertations submitted for the Doctor's degrees and deposited in the University of Kentucky Library are as a rule open for inspection, but are to be used only with due regard to the rights of the authors. Bibliographical references may be noted, but quotations or summaries of parts may be published only with the permission of the author, and with the usual scholarly acknowledgments.

Extensive copying or publication of the dissertation in whole or in part requires also the consent of the Dean of the Graduate School of the University of Kentucky.

A library which borrows this dissertation for use by its patrons is expected to secure the signature of each user.

Name Date

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

DISSERTATION

Muthulakshmi Muthukumarasamy

The Graduate School

University of Kentucky

2010

EXTRACTION AND PREDICTION OF SYSTEM PROPERTIES USING VARIABLE-N-GRAM MODELING AND COMPRESSIVE HASHING

DISSERTATION

A dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy in the College of Engineering
at the University of Kentucky

By

Muthulakshmi Muthukumarasamy

Lexington, Kentucky

Director: Dr.Henry G. Dietz, Department of Electrical and Computer Engineering

Lexington, Kentucky

2010

ACKNOWLEDGMENT

First, I would like to sincerely thank my doctoral thesis chair Dr.Hank Dietz for his academic and moral support throughout this dissertation research. His constant drive for not so obvious solutions always has motivated me to strive for more and I am grateful for the extensive knowledge that I gained, doing research under his guidance. I really thank Dr.Dietz for spending so many hours over the past three months to go over my dissertation and his effort in making this dissertation defense possible in the planned time frame.

Next, I would like to thank my masters thesis chair Dr.Robert Heath for serving in my doctoral committee as well and providing the support I needed. I would like to extend my thanks to Dr.Robert Adams for agreeing to be in my committee and Dr. Jerzy Jaromczyk for serving in my committee and for providing some constructive comments during the defense. I also would like to thank Dr. Grzegorz Wasilkowski for his time in serving as an outside examiner.

Finally, I would like to thank my family - my father, mother, sister and mother-in-law for their constant support and encouragement, which helped me complete my dissertation on time. This dissertation defense would not have been possible without my husband Balaji's constant effort to keep up my spirits, and provide the motivation and encouragement needed to finish this dissertation on schedule.

Table Of Contents

List Of Figures

List Of Tables

CHAPTER 1: INTRODUCTION

Great technological improvements and new architectural innovations have resulted in tremendous improvement in processor performance. In early 1970s, semiconductor DRAM memories started to appear and they came close to processor performance in that period. Since then, due to different semiconductor fabrication technologies being used for designing both the memory and processor chips, performance of both the entities did not scale together. As a result difference between the processor and memory performance has been increasing steadily over the past two decades. While the memory capacity has been increasing along with the processor performance, memory access speed increase at a rate of almost 10% annually, where as the processors have been improving at a rate of 60% annually [48]. This 'Processor Memory Performance Gap', which is the primary obstacle in improved computer system performance is rising fast, making memory reference behavior a dominant factor in determining the performance of many important applications.

Power management is another major performance critical factor in modern systems ranging from handheld devices to supercomputers. Existing methods of energy management often provide reactive control. When it comes to energy use, predictive control has the potential to be more effective than reactive measures, because predictions obtained at runtime can only provide estimates and not bounds on future system performance. By providing the runtime system with static predictions, the system has the ability to anticipate worst possible circumstances and can take preventive measures to avoid future damage.

1.1 Memory Performance

To bridge the processor-memory performance gap, system architects have employed cache hierarchies [56], to store the most needed or most recently referenced information close to the processor thereby improving the performance of common case. Caches are now designed with multiple levels of hierarchies to hide the memory latency. Even though it makes the common case faster, main memory is now farther away from the processor with delayed access times for worst case memory references. Modern multi-core processors have multiple hierarchies in individual cores, thus increasing the latency further. In addition, limited space availability and simple replacement mechanisms of the caches often result in higher miss rates and degraded performance. When one application program pattern may benefit from a particular cache design, others may not, and there is no generic way of predicting the data to be in caches all

the time for all application programs. In addition to that, the paging mechanisms used by the modern operating systems employ Translation Lookaside Buffers (TLBs) to do the address translation for allocating main memory space. All the main memory addresses are translated from logical to physical addresses for the caches to index. This address translation is done with the help of one or two levels of TLBs. TLBs are usually very small size tables with entries typically ranging from 32 to 128. Because of this reason, if the real-memory address of the desired page is not in TLB a further delay is incurred and these TLB misses can be costly which also accounts for the performance degradation.

1.1.1 Memory Access Patterns

Applications can not benefit from caches if they lack locality and in that case, most of the time is spent waiting for data accesses from memory. Loop transformations [44] and data layout optimizations [32] have been done in the past to improve the performance of applications. These transformations and reordering helps only to improve the performance of linearly strided or sequential access patterns. Memory hierarchical organization affects how the main memory gets accessed and, irregular data access patterns are still a problem when it comes to performance. Moreover modern processors employ deeper pipelining, out-of-order execution, out-of-order reordering and automatic prefetch features etc, which further complicates predicting the memory access patterns.

Even though software mechanisms to enhance locality through compiler loop and data layout transformations were employed in the past as a step for improving the performance of accessing huge data structures, random, and irregular access patterns still pose a problem in modern architectures. Dietz et al, [18] show that such random memory access patterns can become the major performance bottleneck for existing and emerging applications. Memory accesses are now a complex function of architectural features and reference patterns, which when accessed randomly might lead to more than hundreds of clock cycles even for a single random access pattern. With complex architectural features, access patterns are hard to predict and the compilers face a huge task of utilizing smart hardware designs to keep up the performance of application programs.

In order to improve the memory access times of data sets with irregular access patterns, this dissertation concentrates on developing a technique that creates a surrogate representation of the original data structure through compression techniques. The compressed surrogates are created using a technique called ‘Compressive Hashing’ (CH). The created surrogates may not be the exact replacement of the original, but can occupy higher levels of memory hierarchy thereby reducing the memory footprints and enhancing the memory access performance.

1.1.2 Applications for Compressive Hashing

For Compressive Hashing (CH) mechanism to work well, the applications should have a large enough data structure that cannot be fit in higher-level cache memories and should have irregular memory access patterns. The applications should possess these properties for the random memory access performance improvements to be visible. Most of the applications either did not fit in this category or turned out to be proprietary. So, the focus of the dissertation was shifted to finding and/or developing applications that can be used with CH. Static power prediction [17] and runtime access to such static data, was identified as having the potential possibility of being an application. The idea was to extract program properties such as energy and time into a static data structure and then apply CH mechanism to reduce the overhead of accessing energy/time values at runtime. Extracting program properties such as energy and time needed detailed instruction-level models. Thus, as a next step in developing an application for CH, this dissertation concentrated on developing instruction-level cost models to derive system-level properties such as energy and time.

1.2 Energy and Time

Power consumption and energy use are one of the major limitations for systems ranging from embedded microcontrollers to high-performance supercomputers. On such systems, if the thermal budget is exceeded, the consequences of overheating can damage the entire system. Even if thermal sensors are deployed on board, by the time, the problem is detected by the sensors, the damage can be inevitable no matter what action the system takes. One can effectively manage this situation by predicting its thermal properties. Thermal properties of the system respond very slowly, so the chances of mispredicting can be very high by predicting them at runtime. Also, any possible worst case runtime behavior can easily damage the whole system without leaving any time for action. Thus, runtime predictions cannot provide worst-case or best-case bounds on future performance. In such situations, static prediction of system properties can be more useful than dynamic predictions as it has the potential ability to foresee all things a program might possibly do. Using such information, it is possible to design a conservative static prediction model that can traverse all possible paths a program might take and provide the runtime system with peripheral bounds on system behavior. The runtime system in turn can use this information to efficiently manage the whole system. By statically predicting the system's energy use and by determining ways to handle the worst case scenario well beforehand, system damage can be prevented. Compilers can utilize such static predictions of energy use as additional parameters for optimization and parallelization transformations. At

runtime, the availability of such static predictions on future behavior of each process will allow the operating system to implement scheduling policies and throttling of clock rate and voltage to maximize performance without exceeding the available power and cooling.

1.2.1 Static Prediction

Accurate prediction of system's run-time properties, such as energy use and execution time, requires a detailed model. Static models based on computationally intensive architectural implementation simulations could be used, but frequent revision of processor implementation characteristics and the proprietary nature of some relevant details make sufficiently accurate models impractical to create and maintain. At the same time, models that does not account for processor pipeline information also remains questionable because lack of such details can considerably affect prediction accuracy. Thus, the ideal model would not be tied to knowledge of architectural implementation details, but would be able to automatically use empirical measurements to accurately account for the impact of such details.

This dissertation describes an instruction-level cost model that is not directly derived from architectural details of the underlying processor, but accounts for the state of a complex processor implementation by modeling the execution context. In computational linguistics, the context often is modeled using a Markov Model variant called N-Gram analysis. The instructions of a computer program are similar to the characters of human written text, so this dissertation proposes to use the same linguistic N-Gram analysis to a computer program to predict high-level properties of program execution. However, traditional N-Gram analysis quickly becomes inefficient as N, the number of symbols considered as a sequence, becomes greater than 3; but in order for the instruction-level N-Gram analysis to model the processor pipeline state information, a significantly larger context may be needed. It is not uncommon for a modern processor pipeline to employ deeper superscalar pipelines [29, 15]. Thus, it was necessary to develop a method by which a large N ($N >= 100$) can be efficiently supported. Efficiency dictates that large values of N should be used only where necessary, thus the method should allow a variable N, using smaller N for portions of the model where prediction accuracy is not adversely affected.

Compile-time lookahead analysis is commonly used to improve the quality of generated code, for example looking ahead a few to few dozen instructions in order to find a better static schedule of instructions. Dietz et al, [17] developed one such static deep lookahead marking algorithm across arbitrary control flow, which when attributed with the instruction-level properties can help the runtime system in predicting system properties. This dissertation discusses the lookahead algorithm in detail and explains how the instruction-level N-Gram predictions are

applied in this algorithm to provide the runtime system with accurate energy bound predictions looking ahead for time periods spanning execution of thousands of instructions. Thus, the runtime system can know with good precision the best and worst-case energy consumption implied by each operating system scheduling decision.

1.2.2 Runtime Support

Most of the runtime prediction mechanisms increase the operating system's overhead and interfere with the prediction of system properties. One way to support runtime predictions and to utilize advantages offered by the static predictions is to make the static predictions available at runtime. This dissertation concentrates on using CH mechanism as a runtime technique to reduce the overhead associated with providing static predictions to the runtime system. The instruction-level predictions obtained through static analysis are first mapped onto a table based on the program counter values. This lookup table can then be accessed by the runtime system on demand. Application programs that transform into large lookup tables always face a performance penalty with respect to its size, when it comes to irregular accesses. On the other hand, performance can be significantly improved, if the table structure can fit in higher levels of memory hierarchy. We utilize the redundancies in the table to create a compressed surrogate table that represents the original table. In most of the cases, the created compressed table may not fully represent the original table i.e., the table can be lossy. In that case, the compressed table just augments the original table and hence storage needs for the application might increase from its original requirement. This CH approach [46] is used to considerably reduce the overhead incurred by the runtime predictive controller. The idea is to create hash functions based on program counter values such that the program counter entries with similar predictions hash to the same location. If more than one runtime property has to be predicted, each can be separated out into separate hash tables with the additional overhead of several hash function calculations. Overhead incurred through hash function calculation is always preferred over several page faults, and this table mapping mechanism could considerably help the runtime predictive controller.

1.3 Dissertation Outline

The research goal of this dissertation started as finding ways to improve memory access performance of application programs and concentration was laid on utilizing a technique called Compressive Hashing [46]. As, finding application programs to validate this technique proved to be real hard, the focus was shifted to finding applications that can be used with Compressive

Hashing. Continuing in that path, energy prediction was chosen as an application and effort was put into deriving energy/time values for application programs. Compressive Hashing was then used as a run-time prediction access mechanism which can considerably reduce overhead for accessing static prediction values at runtime. This section gives details on the complete structural organization of this dissertation.

Chapter 2 introduces the concept of Compressive Hashing (CH) with details on lossy and lossless compressive hashing mechanisms. Genetic programing (GP) is then introduced and how GP is used to evolve compressive hash functions is discussed. Performance analysis of a weather-prediction application table is presented next followed by a reverse engineering validation of CH mechanism.

Developing cost-models to derive system-level properties is the first step in providing static energy predictions. Chapter 3 discusses various such instruction-level cost models and explains the base instruction-level model developed in this dissertation. The concept of N-Gram analysis and how it spawns into a fixed and a variable length N-Gram models is described next. The use of Genetic Algorithm (GA) and Genetic Programming (GP) techniques for solving the fixed and variable N-Gram models is presented along with the corresponding energy and timing analysis. Validation of the N-Gram models is also discussed.

The calculated instruction-level system properties are used for static and run-time predictions as explained in chapters 4 and 5. The deep lookahead algorithm which uses the instruction-level properties for performing static analysis and prediction is discussed in Chapter 4. State machine construction for a program under test, and a tool for annotating the state machine with calculated instruction-level system properties is explained, followed by the description of the long range lookahead algorithm. A small example which shows how the state machine prediction can be done is presented next along with the time to compute lookahead information for several benchmark programs.

Chapter 5 discusses the application of using instruction-level properties in runtime prediction. It starts by discussing possible ways to minimize runtime prediction post cost using Compressive Hashing (CH). Utilizing CH as a mechanism to support runtime prediction is explained next with details on construction of the static data structure and lossy CH techniques. Two test cases *DCRAW* and *GDB* were chosen and the resultant performance improvements obtained were discussed.

In an effort to validate the variable N-Gram analysis techniques developed in chapter 3 in the field from which the idea was developed, the technique was applied to linguistic N-Gram analysis as explained in chapter 6. GP techniques were used to evolve both fixed and variable N-Grams for a sample data set corpus and the results were given.

Chapter 7 concludes the dissertation by summarizing the thesis and the results. Potential future research avenues in terms of instruction-level modeling, static and runtime prediction techniques is also discussed.

CHAPTER 2: COMPRESSIVE HASHING

Compression schemes to improve memory system performance through hardware architectural features existed in the past. Instructions of VLIW (Very Long Instruction Word) architecture often contained redundant or empty fields and the Multiflow Trace architecture took advantage of this fact by having processor hardware fetch compressed blocks of VLIW instructions and decompress them on the fly. The complex instructions of the Intel 432 [28] were Huffman encoded as bit sequences and extracted directly from the code stream by the processor hardware. Although modern processor architecture implementations could benefit from such a hardware-driven approach, the benefit is not as great as one might expect because code stream address reference entropy is relatively low – spatial locality is very good.

Redundancy in data, instructions, and addresses have been exploited to enable compressed representation of the program data and to attain improved performance [41]. Reduced page faults and improved disk access times have been achieved using a compressed memory system [50]. C-RAM (Compressed Random Access Memory) [47] was designed to compress cache lines and dynamically decompress them when cache misses occurred. Even though these hardware additions improve the performance, they impose serious design considerations and incur additional hardware cost.

Transformations that modify the representation of dynamic data structures have achieved modest compression [64], but the majority of compiler techniques have been developed to translate code written as "dense" matrix operations to use "sparse" data structures [3]. The sparse representations assume that the majority of data elements have the same value (most often, zero) and most often are efficient only for fairly regular access patterns. Despite this, these compiler code and data transformations, and the associated analysis, are very closely related to our more general notion of using compression as a technique to reduce memory access cost.

In modern architectures, traditional compiler optimizations for improving memory access performance in accessing huge data structures still cannot be successfully applied for the case of random and irregular access patterns. High-level implementation of compression schemes to improve the performance of such irregular memory accesses is relatively new and has not been considered in the past as an optimization mechanism. For data structures with fixed access patterns, traditional compression schemes can be applied, but most of the available compression techniques are not suitable for applications with variable and irregular access patterns. Hash functions can be used to significantly reduce the space necessary to hold the data structure and at the same time to implement the original variable access lookup [18]. Such hash functions

can be effective in compressing large lookup tables that are accessed irregularly. This chapter explains the concept of using hashing as a compression mechanism to improve memory access times of data structures, discusses the technique used to derive application specific compressive hash functions and analyzes the performance improvements obtained through compressive hashing.

2.1 Lossy and Lossless Compressive Hashing

Unpredictable complex memory references are clearly the major performance limiting factor for many applications that have frequent irregular memory accessed. This dissertation aims at reducing the memory access speeds of such random access patterns in huge data structures by creating a surrogate representation of the original structure through compressive hashing. The compressed surrogate need not be an exact replacement of the original, but can be a 'lossy' representation; meaning whenever the surrogates yield a wrong value, there should be ways to indicate it and recover. The basic idea here is to determine one or more hash functions specific to the applications to create the surrogate structure, thereby making the original data structure occupy less memory space and at the same time implement original lookup function.

In order to apply hashing as a form of compression, application specific hash functions need to be determined. A hash function is an abstract data type that is used to represent a domain of n elements in a domain of m elements, where $m <= n$. One disadvantage in mapping a domain of larger set values to a domain of smaller set values (i.e. n: m mapping), is that it might lead to collisions. Collision means that more than one value from the larger set maps to the same location in the smaller set. There are traditional methods to reduce collision: collision resolution by chaining, collision resolution by open addressing through linear or quadratic probing and by double hashing [59, 34].

Perfect hash functions are hash functions which can achieve a 1:1 and onto mapping. Search for cost-effective nearly perfect hash functions corresponding to particular applications can be tediously complex. Research work done in the late 1970s and 1980s [57, 10, 52] to find minimal perfect hash functions concentrated on finding hash functions for reserved key words in high-level languages or for English words in a small lookup table. Searching for a value in the look up table can be considered similar to searching split-trees [53]. Trees are the best suited data structures when the height of the tree is small. Perfect hash functions, however, do not compress the original data structure. In order to provide compression, the hash function should be n:1. This can be achieved only if multiple original domain elements that map into the same range value are translated into the same hash domain element.

In this way, a hashing mechanism can be used to transform a large data set with enough redundancies into a losslessly compressed representation. When the original data structure is represented in its compressed form, memory access footprints is reduced because even random accesses are random within a smaller domain – ideally, one that fits in cache. Even lossy compression can reduce memory footprints because at least some accesses can be satisfied within the smaller hash table, thus making wide-ranging random accesses to the original table less frequent. Sparseness in the traditional sense is not required, but having many redundancies in the original table makes the search for an appropriate hash function easier.

The basic idea of compressive hashing is to determine one or more hash functions, specific to the application, to create a fully compressed structure. This makes the original table occupy less memory space and at the same time implement the original lookup function. In cases, when the original table is not fully compressible, the nearly-good compressed structure acquired so far augments the original table and the storage needs in such cases exceeds the original requirements. Even though the compressed table is not an exact replacement of the original, it helps in reducing the memory foot prints. Here, the compression becomes 'lossy' and can still be tolerated because, our aim is not in reducing the storage requirements but to reduce the memory footprint that enables faster random access.

Let us assume that the original data structure is represented by an array $D[]$ and the lookup function implemented by indexing $D[]$ is given as, $L[k] = D[k]$. In implementing the lookup function, it is possible that $L[k]$ yields the same value for two or more array indexes (i.e.),

$$L[k_i] = L[k_j] \, for k_i \neq k_j$$

meaning that there might be redundant values in the original lookup. In this case, one copy of the original value can be retained and others can be eliminated. So finding a hash function $H[k]$ for all values of k, will transform the original table into a table with fewer entries and if,

$$H[k_i] = H[k_j] \, for k_i \neq k_j \, then,$$

$$L[k_i] = L[k_j] \, for k_i \neq k_j$$

But $L[k_i] = L[k_j] \, for k_i \neq k_j$ does not mean $H[k_i] = H[k_j] \, for k_i \neq k_j$. Duplicate redundant entries will also occur in the compressed table provided the total space to represent the data structure is still reduced. By compressing the original huge table into a smaller table that might fit into higher levels of memory, the compressive hash functions can be used as a means

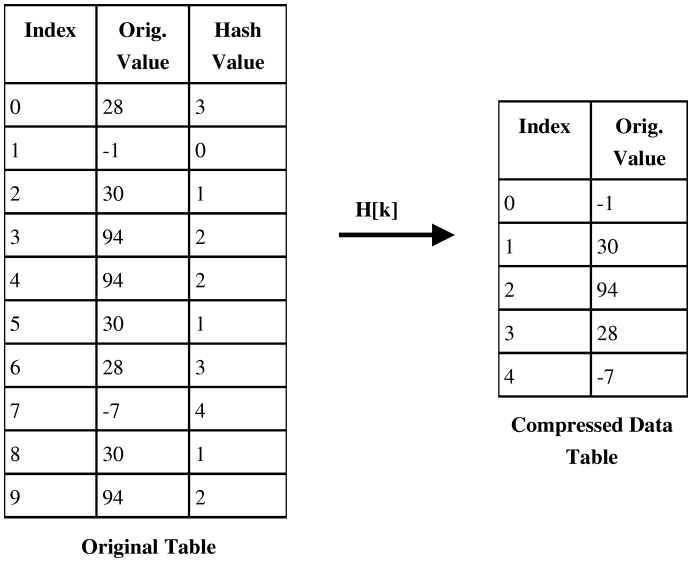

Figure 2.1: Lossless Compressive Hashing

to reduce frequent main memory references. Figure 2.1 gives a high-level idea of lossless compressive hashing mechanism.

The above compressive hashing scheme can also be 'lossy' in the sense that, the hash functions used to compress the huge table might not yield full compression (i.e.) the scheme can yield correct values only for some of the inputs. This lossy scheme can be tolerated by having a checker function that will point to another lookup table which has single bit-wide entries to indicate the correctness of the given hash function for the specified original table. As the checker function represents a bit-wise table, it should be relatively easy to implement. Depending on the correctness value from the checker table, these two lookup tables alone can be used to represent the huge table and the wrong values can be redirected to be extracted from the next lower level of memory yielding two/three lookups per access. The same compressive hashing mechanism can be applied again to the checker table to represent it as a smaller data table and this whole process in turn can be recursively applied until an affordable and an effective solution is found. Figure 2.2 explains the lossy compressive hashing mechanism.

2.2 Genetic Programming (GP) to evolve Compressive Hash functions

Finding application specific hash functions is the first step in generating compressed representation of lookup tables through compressive hashing. Hash functions that perform total table compression are not necessary in this case. Even if some hash table entries are wrong, it can be fetched from a different (may be original) table provided there is a way to tell that it has to be fetched from different table. Accessing main memory for $10\% - 20\%$ of the time is always

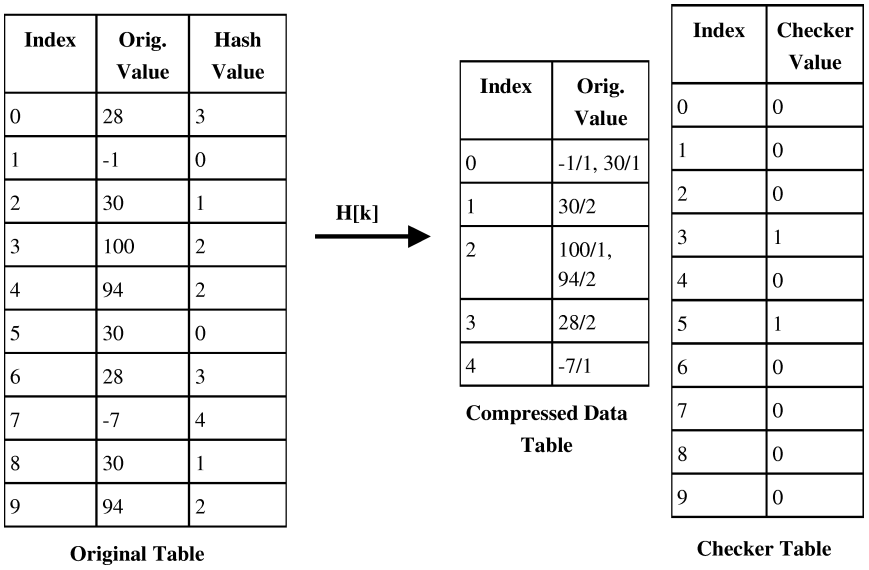

Figure 2.2: Lossy Compressive Hashing

better than accessing nearly 100% of the time.

Search for a nearly perfect hash functions in which the original table consists of millions and millions of $32-bit$ or $64-bit$ data is very complex and might require complex algorithms. Many approaches such as Genetic Search, Adaptive Methods etc. can be applied to search for hash functions. In recent years, evolutionary computing techniques like genetic programming have proved to be a solution for various engineering problems [36]. It has proved to be successful in automatically searching a huge search space for solutions.

Genetic Programming (GP) works by creating a programming model that automatically generates solutions to a specific user defined problem through the concept of natural evolution process. It utilizes the principle of natural evolution and fitness based selection and recombination. This dissertation uses GP techniques to search for application specific hash functions to aid compressive hashing in creating surrogates. A high-level idea of how an evolutionary process through GP is done can be understood thorough the following steps:

1. Create random initial population members that are valid candidates for the problem solution.

2. Evaluate fitness for each member in the population. Fitness calculation is done to meet the criteria for solution to the specific problem.

3. Replace the relatively week member(s) by crossover and/or mutation operations

(a) Crossover - Create new members by combining features from two different population members.

(b) Mutation - Create new members by mutating features of an individual member.

4. Repeat from step 2 for sufficient number of generations.

5. Stop the whole process after desired number of generations has evolved or best enough solution has been found.

To apply GP to the given problem of determining hash functions, the individual population members are represented by an array with operands following the predefined operators. The individual population members are generated by a random recursive method that chooses either a terminal or non-terminal for each array value. If a terminal operator is chosen the recursion terminates. But if a non-terminal is chosen, the recursion is applied again to determine the operands for that operator and the recursion is continued. The array size is usually prefixed and size of the arrays generated is maintained within this limit while generating the members. Thus each member of the population will evaluate to a potential hash function.

One of the major goals in the design of hash functions is to make it simple and cost effective with respect to the underlying target machine's instructions. Operators can be chosen depending on the target processor's architecture while aiming for performance. For example, Athlon has a very fast integer multiply unit [14] which makes the multiply operator a viable candidate for hash function generation.

A reasonable size for the hash table is determined and the fitness for every hash functions is determined by how they affect filling the small table, (i.e.) an individual member and its potential hash function is better if more entries from the original table gets filled in the small compressed hash table. For lossless compressive hashing, the hash function is calculated with the original lookup table indices and the values, that correspond to those indices get filled in the calculated hash table location. If the values differ while hashing other entries, the location itself is marked as invalid and the original count of filled entries is decreased by the previous count in this particular location.

For lossy compressive hashing, a separate count of each value being hashed to a particular location is maintained and finally, that location gets marked as valid with the value that has the maximum count. This way of lossy compressive hashing needs a checker table for all the original entries with bit-wide entries (0 - get from hash table, 1 - get from the original table). If needed, the checker table can also be extended to 2 or more bit-wide entries depending on the number of small hash tables that result from GP. For each member, the whole hash table

is filled and evaluated based on the number of filled entries. After fitness computation for all members in each generation, some percentage of weak members having less filled entries undergo mutation and crossover operations to produce new members for the next generation. The whole evolutionary process continues for generations until a good possible solution is determined. Currently the above mentioned GP technique is being used to do the search and it has proved to be useful for the cases considered here. As the search is random, it requires runs of several hours to do a good search in most cases.

2.3 Validation through Reverse Engineering and Performance Analysis

To validate the compressive hashing mechanism in terms of performance improvements and also to verify whether a GP approach to evolve hash functions is indeed an acceptable method, the concept of reverse engineering was chosen to test whether GP can yield reasonable hash functions for pre constructed tables. This section presents in detail, the experimental setup and the test cases being considered for implementing compressive hashing through GP. The performance results are also evaluated based on the execution times, memory footprints and the target architecture's parameters. Through the concept of reverse engineering and predefined hash functions, four lookup table test cases of approximately 3MBytes each were constructed and filled with 32-bit integer values with their appropriate hash functions to yield a compressed hash table of size 4KBytes each. Hash functions are chosen randomly utilizing logical, shift, arithmetic, and population count operations. Building tables in this manner was done as a validation mechanism to test the concept of creating a compressed table that represents the original and also to see whether it results in any performance gains. Additionally this mechanism also verifies how good the GP system was, in finding the hash functions, given that we already knew that a hash function exists for the given table.

The first reverse engineered test case constructed a table with a sparseness level of 54% and hence is identified throughout the paper as H1SP54 (i.e. H1SP54 - Hash Function 1 with 54% Sparseness). The other test cases are chosen such that three of them use the same hash function to create the original tables, but the sparseness of the table varies considerably with 49%, 37% and 22% respectively. These reverse engineered test cases are identified with the names H2SP49, H2SP37 and H2SP22 according to their varied sparseness levels.

The GP system was run on a 128-node Linux cluster called KASY0 [43] that employs 2GHz AMD Athlon(tm) XP 2600+ processor nodes. In most cases, the GP system found the hash function overnight; in the worst case, it took 4 days. For one of the reverse-engineered test cases, GP came up with a completely different and slightly better hash function than the one from which the case was constructed. For all these reverse-engineered test cases, the con-

structed hash function is lossless i.e., each lookup in the original table corresponds to only one look up in the compressed hash table along with the correct hash value calculation.

A benchmark was created to access these tables in linear strides and in random manner to measure the performance for all the test cases and was run on a 700 MHz AMD Athlon processor with 64KByte L1 cache and 512 Kbytes on-chip L2 cache. Clearly, none of the above tables fit in the caches completely. PAPI [20] was used to measure the hardware performance counter values for L1, L2, TLB data misses and clock cycle count for each access. The comparison between accessing the original table and the compressed table is presented in the following paragraphs. The hash function used for H1SP54 test case is given as,

```
hash value = ((z ^ ((x ^ y) << 7)) >> popcnt(z))
```

where `x, y, z` are the three dimensional indices to access the original table and `popcnt()` determines the number of ones in a given value [14]. The following graphs visually summarize the tremendous performance difference between the original and the compressed versions. Figure 2.3 compares performance results obtained for H1SP54.

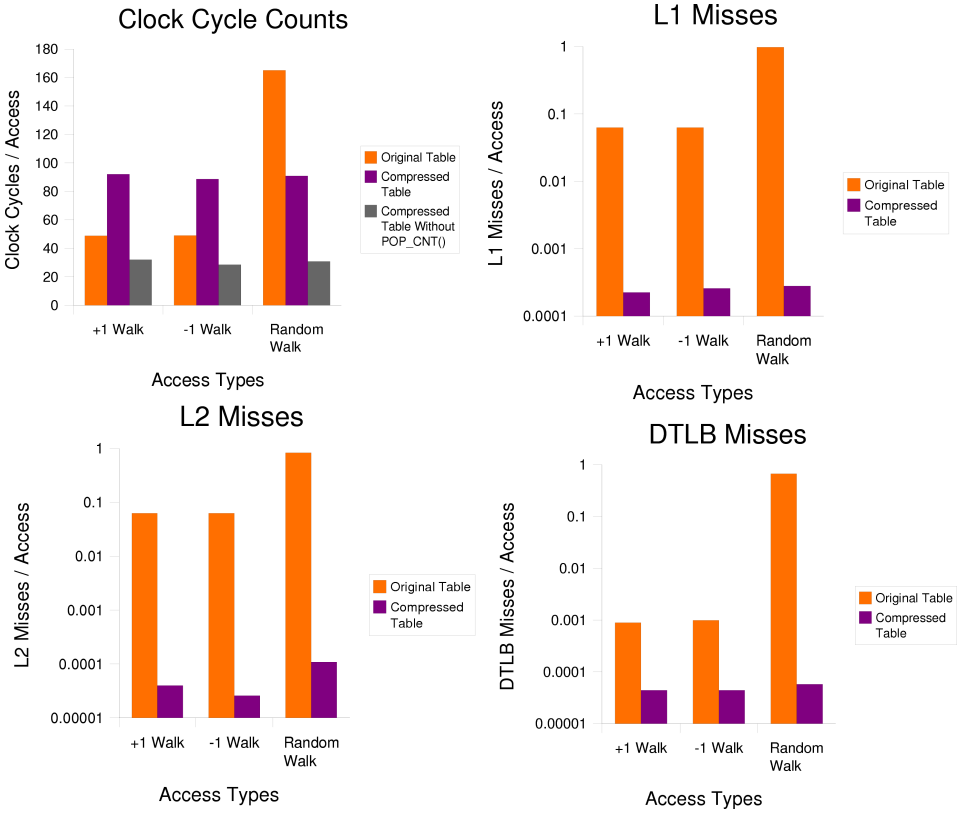

Figure 2.3: Performance comparison for Test Case1 (H1SP54)

The benchmark accesses the original and the compressed tables using loops and the hardware counter values were read before and after the loop. To get accurate results, the measurements were taken more than once (hundreds of times) and the minimum value was taken to be the true measure, since interference can only increase the measured values. Because the array bounds are not a power of two for the original table, a modulus operation was used to calculate the random test indices. Each such modulus operation by itself takes on the order of 20 clock cycles in an Athlon, so the modulus operations in the test case generation code were measured separately and deducted from the total clock cycle count.

Figure 2.3 shows the clock cycle count for a single stride linear access and random access for the original and the compressed table. The number of clock cycles for the compressed table is the sum of clock cycle counts for calculating the hash function and clock cycle count for accessing the compressed table. The total clock cycle count for linearly accessing the compressed table is more when compared to the original table's linear access count. This is due to the population count operation in calculating the hash value index for the compressed table. It takes on the order of about 60 clock cycles in an Athlon to perform a single population count operation. For random accesses, the compressed table still performs better than the original table, even with the population count operation in the hash function. In later test cases, this insight is taken into account and population count operation was eliminated while constructing the hash functions.

Figure 2.3 also shows the number of clock cycles for accessing the compressed table without the population count operation. The number of DTLB, L1 and L2 misses are zero or close to zero when accessing the compressed table either linearly or randomly. To visualize this fact better, the number of TLB, L1 and L2 misses are plotted using a logarithmic scale. Linear access of the compressed table does not mean single stride access, (i.e.) stepping through the original table indices linearly (+1/-1) may result in not-so-linear indices for accessing the compressed table due to the hash function calculation. This is the reason for the clock cycles count being almost the same for either of the accesses. In contrast, a linear walk of the original table has far fewer misses than random accesses, yielding dramatically different execution times.

Performance comparison of original versus compressed table for the second test case H2SP49 is shown in the above Figure 2.4. The above test case 2 (H2SP49) and test cases 3 (H2SP37) and 4 (H2SP22) have been created using the following hash function with decreasing sparseness in their values respectively.

```
hash value = ((((x<<13) ^ (y>>27)) | z) + ((y-z) ^ (x&11)) * z)
```

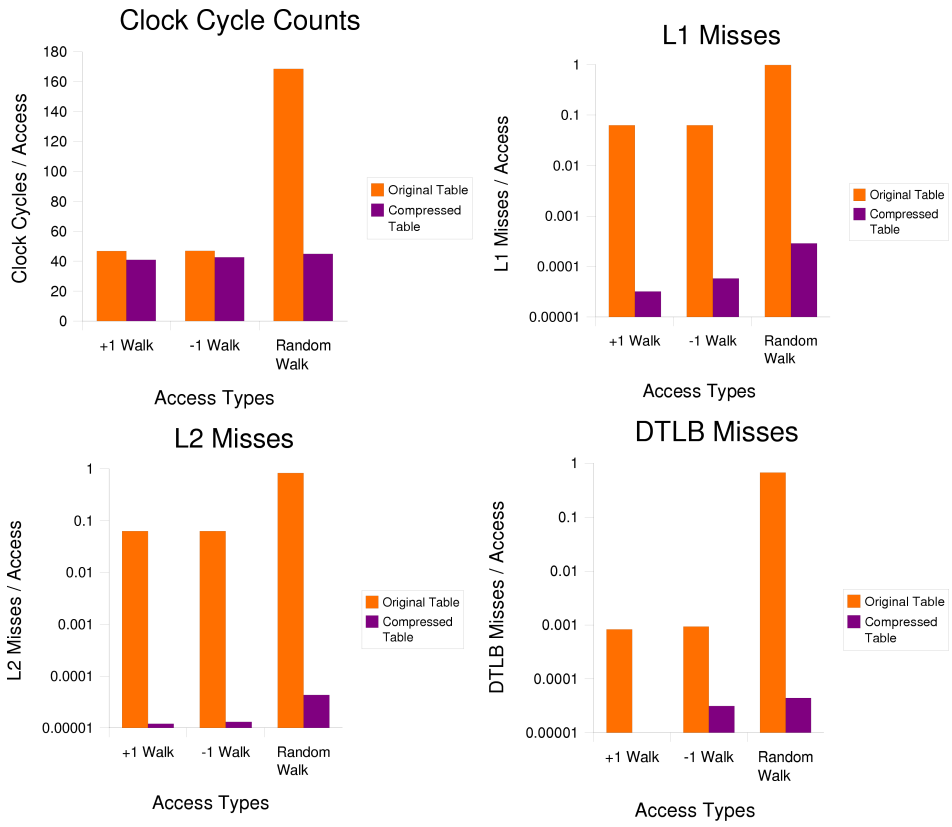

Figure 2.4: Performance comparison for Test Case2 (H2SP49)

Interestingly, the same GP system as used before also found the following better function for H2SP22.

```
hash value = ((((0xb & x) ^ (y - z)) * z) + z)
```

where `x`, `y`, `z` are the three dimensional indices to the original table. The number of clock cycles for accessing the compressed table linearly including the hash function calculation is considerably less than accessing the original table linearly and dramatically less when the tables are accessed in a completely random manner. Similar to H1SP54, the compressed table misses less often than the original table. Similar results were obtained for test case 3 (H2SP37) and test case 4 (H2SP22) and the following graphs in Figure 2.5 and Figure 2.6 present their performance comparison of the original and the compressed table. Figure 2.6 also shows the clock cycle count for the GP evolved hash function that is different from the original, and it can be seen that it proves to be slightly better than the given hash function.

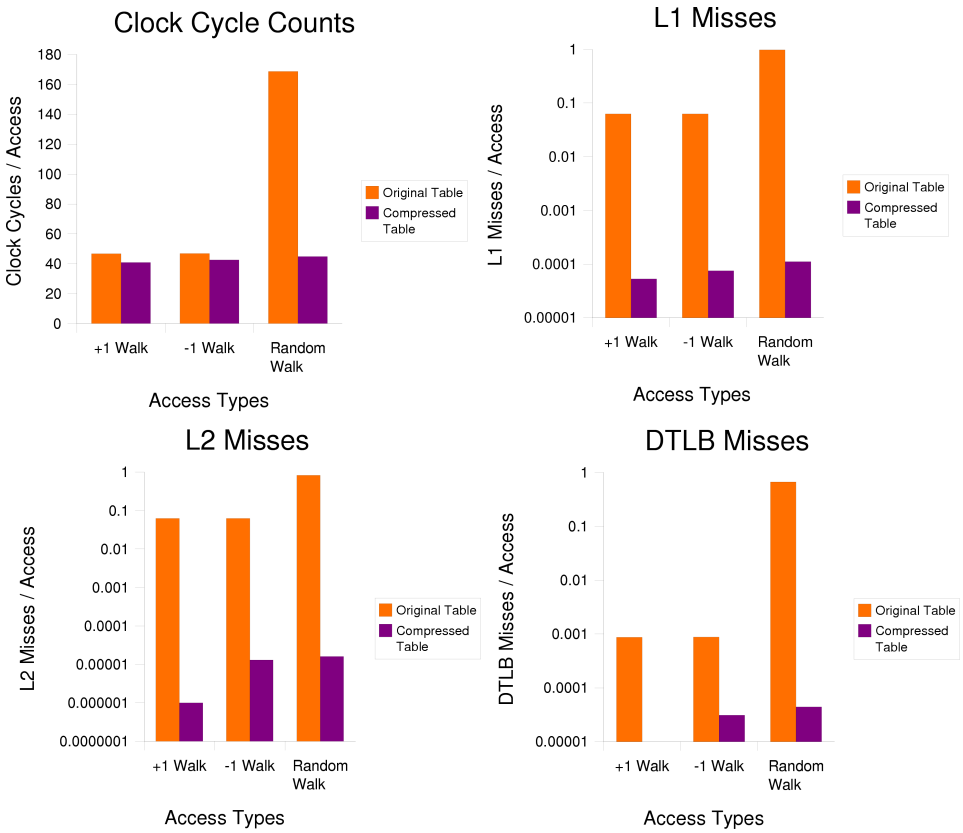

Figure 2.5: Performance comparison for Test Case3 (H2SP37)

2.4 Test Case Performance Evaluation

Compressive hashing was then applied to a weather prediction (WP) table data set which was size of approximately 3MBytes and was filled with 32-bit floating point values. The WP table turned out to be somewhat sparse – although not sparse enough with any regular patterns for the usual sparse data structure methods to be directly useful. The GP system was run for several days with the goal of creating a lossless compressive hash function for the WP table. With a chosen hash table size of 4Kbytes, GP could successfully fill ~6/7th of the original table entries. Hashing the remaining ~1/7th of the entries turned out to be very difficult, even with a increase in hash table size. So, the GP system was modified to evolve for a lossy hash table in which one original table lookup corresponds to more than one table lookup (i.e. one compressed table lookup and one original table lookup).

GP created the following hash function,

```
hash value = (z - ((0x7430b989 * ((2*x)+1)) >> 0x12))
```

where `x, y, z` are the indices to the original table. It can be noted that y index never got used

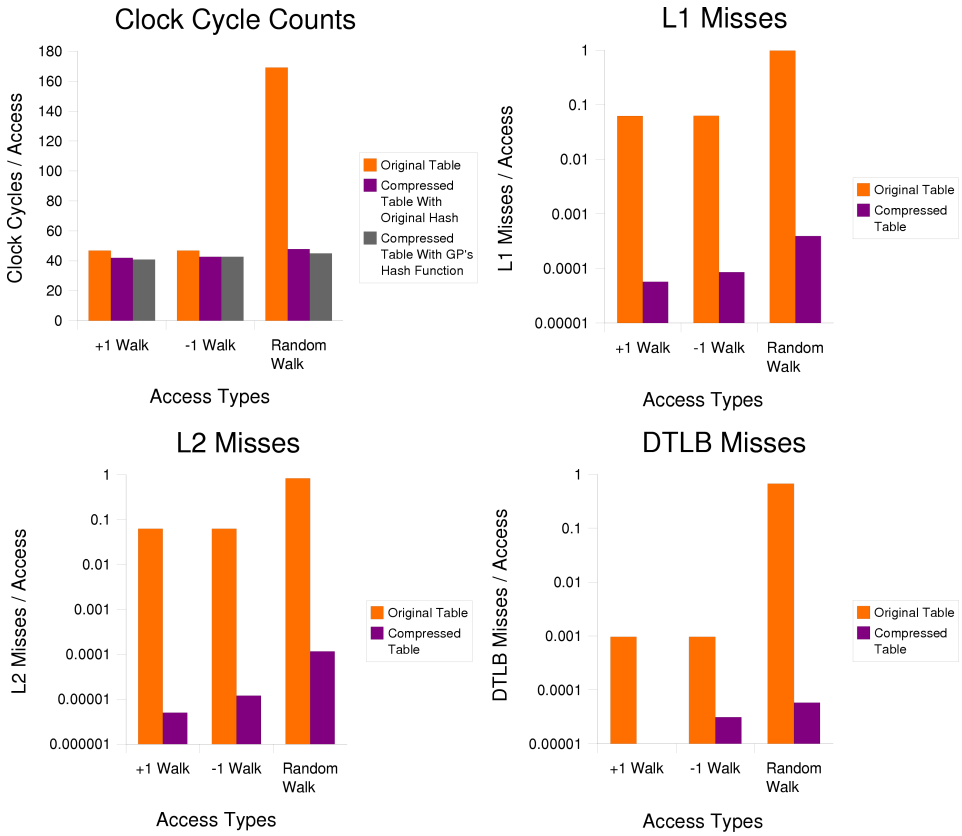

Figure 2.6: Performance comparison for Test Case4 (H2SP22)

in the hash function derivation. The above lossy hash function filled the compressed data table with ~6/7th of the original table entries and the remaining ~1/7th was directed to be retrieved from the original table. To avoid having a separate checker for all the original table entries, the invalid entries are marked directly in the hash table to be accessed from the original table. The following Figure 2.7 shows the performance results obtained for the WP table.

As can be seen from the above figure that, even with the increase in the resultant total table size, total clock cycle count (clock cycle count for the hash function calculation + clock cycle count for deciding between accessing the small compressed or the large original table + clock cycle count for accessing the table itself) is still less when compared to randomly accessing the large original table directly. The clock cycle count for linearly accessing the original table is slightly less than the compressive hashed table. This might be due to the fact that, linear access of the original table does not necessarily mean linear access for the compressive hashed table and there is also additional overhead associated with the branching involved in accessing the correct table.

Graphs that correspond to DTLB, L1 and L2 misses also portray similar facts as the previous reverse engineered cases when comparing the original and the compressive hashed versions.

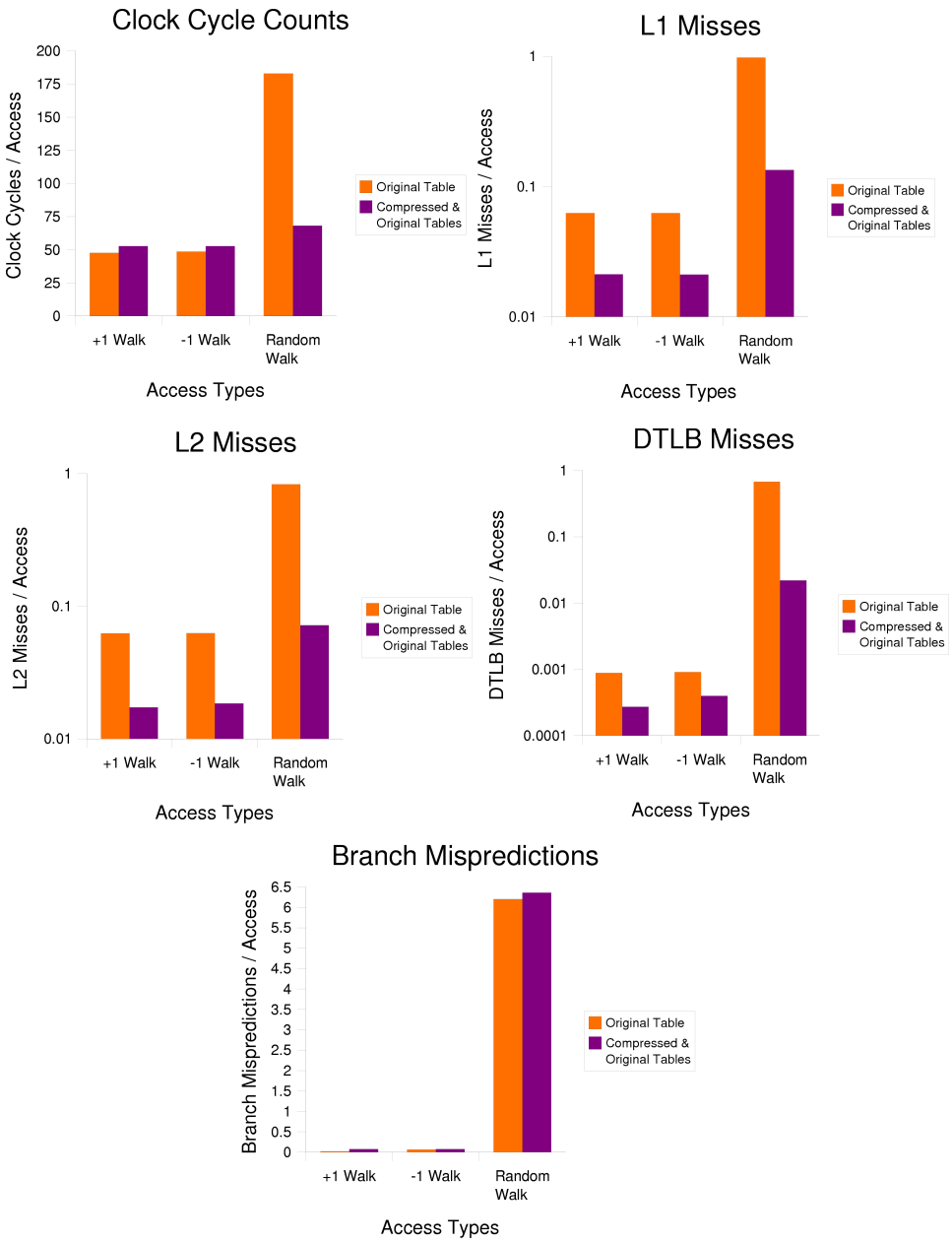

Figure 2.7: Performance comparison for WP Table

The misses here are relatively more when compared to the previous test cases which can be obviously understood due to the increase in the table sizes and the branch mispredictions that might occur in accessing the correct tables. The above figure also shows the branch mispredictions that occurred while accessing the tables linearly and randomly. As expected, the number of mispredictions are more for the compressive hashed table than the original.

All the above test cases show that, smaller data table representations created using compressive hashing techniques make most of the table entries to be fit in caches and hence considerably

reduce the cost for irregular memory accesses. The hash functions for creating the compressed data table also play a major role in cost reduction, and so the GP operators needed to create these hash functions should also be chosen wisely based on the underlying target architecture, so that it is efficient enough to decrease the whole memory access cost.

Runtime support [17] for providing system-level power predictions is bound to have a large data structure with irregular accesses to provide on-demand predictions. Such irregular accesses can greatly benefit from using CH. In order to come up with static predictions to provide to the runtime system, detailed instruction-level models and static analysis of such data are needed. Following chapters 3 and 4 discuss the design of N-gram cost models to derive energy and time values of application programs and describe how a static analysis and prediction can be done with the derived instruction-level values.

CHAPTER 3: INSTRUCTION-LEVEL COST MODELS

System properties for an application program can be measured at different granularities - either at functional unit (IU, FPU, etc.,) level, at operations (memory, alu etc.,) level or at the instruction level. This dissertation presents an approach that is based on measuring program properties at the instruction-level. Even though there are a variety of techniques that measure performance metrics at the instruction-level, the cost models described in this dissertation differ in the following ways:

- It does not directly depend on low-level hardware architectural details
- It measures instruction properties in their *idiomatic context*

Most of the modern processors are heavily pipelined [29, 15], with more than one instruction executing in a particular pipeline stage at any given time. They also employ complex prefetch and out-of-order instruction scheduling logic to hide per-instruction execution latency. All this to say, modern processors are complex circuits to model. Even if one goes to the extent of modeling the complex architectures, insufficient detailed information about the design and frequent processor revisions makes it extremely difficult to create and maintain a generic model. In an effort to overcome the above mentioned problems and to make the model independent of low-level details, a generic performance model was developed that can be applied to processors ranging from real-time embedded processors to a high-performance supercomputer.

Another important aspect of the model developed in this dissertation was its ability to predict performance metrics statically. In order to make bounded performance predictions rather than estimated predictions, static prediction was selected over dynamic mechanisms. To determine application performance metrics statically, empirical measurements over compiler generated code segments were used to collect individual instruction's performance values.

Most of the instruction-level performance models described in the literature measures instruction properties in isolation. In other words, most methods repeat the same instruction several number of times inside a loop and take the median value of the measured values to be the resultant metric of the instruction under test. Rather than measuring instruction properties in isolation or within artificial test frameworks, this dissertation work measures instruction properties in the *idiomatic sequences that the compiler normally generates them* and also records the approximate context information in which each measurement was made. This way, the effect of various instructions on the performance of the instruction under test can be determined.

This chapter discusses the novel approach in developing instruction-level cost models and analyzes the performance metrics. Section 3.1 presents the different instruction-level cost models available in the literature for energy and time predictions and section 3.2 discusses the base instruction-level cost model. The cost models developed in this dissertation research closely follows N-Gram analysis - a technique that is commonly used in the field of linguistics for text classification, determining ownership of documents etc. Section 3.3 explains how the benchmarking procedure for the empirical analysis is done.

A brief explanation of N-Grams and their analysis is presented in section 3.4 followed by section 3.5 which explains the Fixed-N-Gram instruction-level cost model. Genetic Algorithm (GA) to solve the Fixed-N-Gram model is discussed in section 3.5.2. Performance metrics such as execution time and energy consumption are computed using the model and their instruction-level analysis is presented in section 3.5.3. A new approach to support Variable-N instruction-level model was developed in an effort to improve the prediction accuracy of Fixed-N-Gram model and is explained in section 3.6. Section 3.6.1 presents a unique hybrid evolutionary algorithm for solving Variable-N-Gram model. Instruction-level performance metric analysis using variable-n-grams is explained in section 3.6.3 followed by the model and algorithm validation using an abstract machine representation which is explained in section 3.6.2.

3.1 Related Work

Most of the instruction-level performance modeling done in the past either involves complex computations to model processors commonly used in High-Performance Computing (HPC) or requires detailed architectural models that are difficult to obtain and may change with each "minor" revision of a processor.

Hardware Description Language (HDL) simulation of a processor implementation can provide highly accurate performance metric (power) information [8], but is not feasible for a significant number of instructions when simulating superscalar, deeply pipelined modern processors. Higher-level simulations require less computation, and can still provide accurate power estimation [4, 5, 38, 51]. Unfortunately, all of these methods need low-level details of the processor's architecture to derive the power model.

Even if appropriate architectural models can be created, it is not clear that all significant parameters can be reasonably known by a static analysis tool. Execution time and energy consumption of a code sequence on a modern processor are functions of the very complex state of the computer system at the precise moment each instruction is executed. Thus, it is possible that execution time and energy use can be significantly altered by:

- Details of the processing pipeline and its current contents
- TLB, cache, and memory system characteristics and current state
- The actual values being operated upon (e.g., it used to be common that multiply timing was a function of 1-bit placement in the operands; now value dependence is most often associated with triggering speculative operations such as memory prefetches)
- The current state of the rest of the computer system hardware (e.g., even if video card, disk drive, and other I/O subsystems are modeled, these components might interfere with the processor's handling of the current process in complex ways)

These are not properties normally available to static analysis, neither is it feasible to make them available.

Instruction-level power analysis techniques without using micro-architectural information to estimate power consumption for embedded DSP [60] and less complex general purpose processors [61] measure current for a single instruction by repeating several instances of the same instruction in a loop and averaging over all the iterations to determine the instruction's base cost. The instruction-level power model is then derived as a summation of instructions' base cost and the cost due to inter-instruction effects such as circuit state switching. The same basic model also was extended to include energy consumption of external buses [58]. Klass et al [33] claim that power prediction models are not affected by all possible inter-instruction effects, but only by change from one type of instruction to another – essentially equivalent to a crude 2-Gram model. Thus, they use a single number to represent the instruction overhead for all possible combinations of inter-instruction effects in a DSP processor. Krintz, Wen, and Wolski predicted power consumption of a program by grouping instructions into integer/floating point and register/memory operations [37].

The above methods do not require detailed architectural information and have been shown to perform well for DSPs or general purpose processors like the Intel 80486. They do not, however, account for the complex inter-instruction effects of more modern processors. In particular, the methodology used to measure an instruction's base cost does not consider the effect of surrounding instruction sequences on a particular instruction. Put another way, the benchmark code is idiomatically different from that normally occurring in compiled application code. Those idiomatic differences introduce errors because other aspects of the state of the machine have potentially large and uncontrolled impact on the measurements made. Gebotys and Gebotys developed a model that uses the five parameters they found to most influence current consumption for a set of DSP benchmarks [23, 9]. They were able to predict current consumption within 2%, but the DSP they used is much simpler than a modern out-of-order processor.

The primary weakness in previous static instruction-level models is that interactions between instructions are ignored, oversimplified, or modeled using a level of implementation detail which is often impractical for modern processors. These interactions are potentially significant, as demonstrated by the dynamic method proposed by Bellosa et al [1, 2]. They use a calibration technique to associate power costs with specific performance counter events (e.g., cache misses) – in essence, they ignore instructions and model *only* interactions between them. This runtime analysis produced estimates with an average error of approximately 20%. It is not trivial for a static tool to accurately predict performance counter events. Doing so would introduce additional error, but clearly these inter-instruction events should not be ignored.

3.2 Base Instruction-Level Cost Model

The models developed in this dissertation are based on instruction analysis, but takes care to measure instruction properties in their idiomatic context. In other words, rather than measuring instruction properties in isolation or within artificial test frameworks, the instruction properties are measured in the idiomatic sequences that the compiler normally generates them – and the approximate context in which each measurement was made was also recorded. Each code segment of an application program is treated as having properties that are the result of linear combination of the segment's component instruction-in-context properties, which individually can encode highly non-linear relationships.

Suppose that a benchmark code segment contains a sequence of instructions:

$$I_a, I_b, I_c, I_b, I_d, I_a, I_b, I_d, I_c, I_a, I_b, I_d, I_c, I_a, I_b, I_d$$

Each of the instructions appearing in the code segment can be distinguished based on opcodes, addressing modes, etc., so that I_a simply means an instruction I_a exhibiting a particular set of characteristics. It also is possible to distinguish instruction types based on properties such as "causes an L1 data cache miss," or "depends on previous 2 instructions" etc.,. Thus the model described here is flexible enough to combine any of the different characteristics. For example, if an instruction *mov* appears in a code segment, it can be encoded differently as $mov\,(opcode)$, $mov-load\,(addressing\,mode)$, $mov-load\,(likely\,to\,cause\,L1\,miss)$, $mov-load\,(value\,depends\,on\,previous\,K\,instructions)$ etc.,.

Let T_{code}, E_{code} represent the execution time and energy consumption of the above mentioned code segment and let $E(I_n)$, and $T(I_n)$ represent formulae for the energy cost and execution time for instruction I_n respectively. The base instruction-level cost model can then be constructed using equations of the following form:

$$T_{code} = T(I_a) + T(I_b) + T(I_c) + T(I_b) + T(I_d) + T(I_a) + T(I_b) + T(I_d) + T(I_c) + T(I_a) + T(I_b) + T(I_d) + T(I_c) + T(I_a) + T(I_b) + T(I_d) \quad (3.1)$$

$$E_{code} = E(I_a) + E(I_b) + E(I_c) + E(I_b) + E(I_d) + E(I_a) + E(I_b) + E(I_d) + E(I_c) + E(I_a) + E(I_b) + E(I_d) + E(I_c) + E(I_a) + E(I_b) + E(I_d) \quad (3.2)$$

The above equations represent the base model which encodes completely linear relationships. However, to incorporate potential non-linear interactions between instructions, the N-Gram models described in the following sections allow $E(I_n)$, and $T(I_n)$ to be functions not only of instruction I_n, but also of the $N-1$ instruction sequence context preceding that instruction.

3.3 Benchmark Procedure

The requirement that the code being measured be idiomatically correct for the instruction-level analysis does impose a constraint on how benchmark measurements should be made. Mechanically generating assembly-level code that tests each instruction in isolation is not idiomatically correct, nor is it idiomatically correct to measure assembly code generated to have examples covering all possible instructions. Measurements can be performed either using "real" application programs or synthetic code that has been mapped into the appropriate idiomatic sequences by using the same compiler optimization and code generation methods used for applications. Almost all real application code contains control flow constructs which makes it harder to develop an accurate model because of the uncertainties involved in branch prediction. So to develop straight-forward cost models, several random synthetic benchmarks that contains basic block code segments were created. However, any of the models described in this dissertation can be extended to include appropriate contribution from corresponding branching operations by weighting the alternative instruction sequences using the measured branch probabilities. Thus, conditional branching instructions are not conceptually different and are not likely to cause greater error in predictions, but merely more awkward to benchmark.

The actual decomposition of a benchmark code into instruction types (and execution count for each type) for the models can be accomplished either by processing the assembly code or by disassembly of the executable object file. A set of AWK scripts are used to process AMD Athlon assembly language source code to generate the instruction types for the models along with their respective execution counts.

In all the experiments, the execution time of the benchmarks is measured using the processor tick counter, and power is measured with a WATTS UP? POWER METER. This power meter

measures power delivered by the 120 VAC electrical outlet and thus includes the entire system. The WATTS UP? can only report one average power measurement each second. To get a consistent reading, the benchmark is run until the system reaches a steady power state – i.e., the power reading from the WATTS UP? stays within a threshold of ε for the last m readings . A threshold of $\varepsilon = 5\%$ and number of readings $m = 5$ were used for all of the measurement data. This allows the code to have run for a long enough time to get a greater accuracy on the average execution time and the corresponding energy consumption of the code under test.

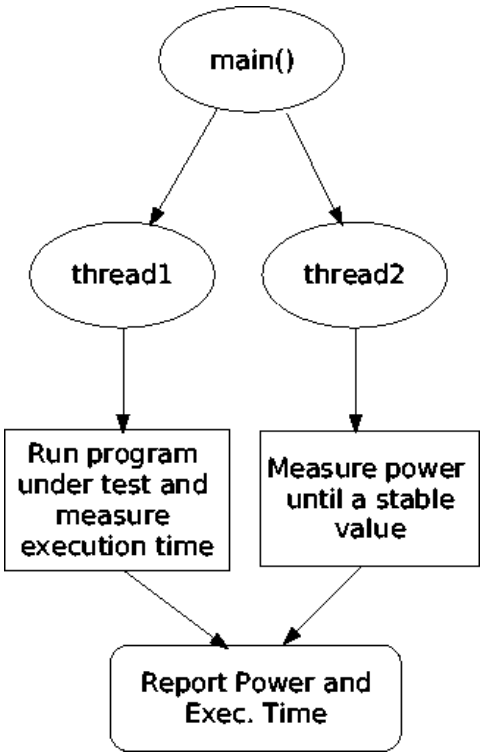

Figure 3.1: Experimental Setup

As shown in figure 3.1, the measurement program has two threads running concurrently: one runs the code to be measured and the other measures power readings from the power meter. The measurement process wakes once a second and runs for a short enough time so that execution of the measurement process has negligible impact on the values read. The measurement program waits for the system to reach a steady state before starting a benchmark. Once the readings become stable, the benchmark is run until power reaches a new steady state. Additional energy consumed for running the benchmark, E, is computed from the power and time measurements straightforwardly as $E = (P_{benchmark} - P_{idle}) \cdot T$, where $P_{benchmark}$ is the power measured while the benchmark executes, P_{idle} is the power measured is when the system is idle before the benchmark, and T is the benchmark execution time. The measurement program repeats this

process to get time and energy measurements for all of the synthetic benchmark programs.

3.4 N-Gram Analysis

N-Gram analysis is commonly used in various fields ranging from identifying gene homologies to correcting spelling errors in Google searches. It is also used in information retrieval and securely identifying file contents and types for malware and intrusion detection. In computational linguistics, it has been used to create statistical models for text classification and prediction [6], text processing [62], text categorization [7], etc. In linguistic analysis, an N-Gram is constructed by extracting a sequence of N contiguous lexemes from a text. For example, the text "abcb" contains the 2-Grams "ab," "bc," and "cb." N-Gram models are otherwise known as statistical N-th order Markovian models, (i.e.,) in linguistic terms, N-Gram analysis calculates the probability of a word occurring in a sentence, given the conditional probability of previous $N-1$ word sequences $[P(X_i|X_{i-1}, X_{i-2}, ..., X_{i-N})]$.

In human languages, the statistical properties of N-Grams, such as the relative frequency of occurrence of different N-Grams, are remarkably stable. Because "ell" is a much more common 3-Gram in Spanish than it is in English, a text containing many "ell" 3-Grams is much more likely to be written in Spanish than in English. Of course, this classification is not perfect – "Hello" is not Spanish – but the statistical properties for human languages are so stable that they have been used to identify authorship of documents in disputed cases. However specialized the language, computer programs are human-written language texts, and so are compiler code generation templates, so it makes sense that similar statistical consistency should be present.

Linguistic N-Grams are statistically extracted from training sets based on huge corpus of text. Similarly, N-Grams are extracted by generating thousands of compiler generated code. The stochastic Markovian model for the instruction-level cost analysis gets translated into $[C(I_i|I_{i-1}, I_{i-2}, ..., I_{i-N})]$, where C is the performance cost (energy, power or time), I_i is the instruction I in i^{th} position in the code segment. Even though long-range correlation of linguistic N-Grams drop exponentially with distance, the N-Gram model developed in this dissertation for instruction-level analysis is capable of correlating distances on the order of 100 grams or more to capture deeper processor pipeline state information.

3.5 Instruction-Level Fixed-N-Gram Model

The instruction-level analysis for an N-Gram model is based on instruction types which are N-Grams. However, linguistic N-Grams are uninterpreted; a definition of the precise meaning

of an N-Gram was needed in order to associate cost measurements with N-Grams. Suppose that an instruction sequence I_a, I_b, I_c appears in a code segment. Not surprisingly, the 1-Gram instruction types would be I_a , I_b , and I_c. However, 2-Gram instruction contexts are constructed somewhat differently. Given I_a, I_b, I_c, we would create the instruction 2-Grams: $(*I_a)$, (I_aI_b) , and (I_bI_c) , where (I_aI_b) would be read as "I_b in the context of following I_a." The purpose of this context is thus establishing a portion of the machine state *before* execution of the instruction in question. Given this definition, the 2-Gram $(*I_a)$ is particularly interesting – because we do not know the context before our sample began, we need an additional symbol, $*$, to represent an unknown previous instruction which could be any type of instruction.

The base equations 3.1 and 3.2 for the sample code segment in section 3.2 can be otherwise treated as a *Fixed-1-Gram* model equations. For the example code segment, the *Fixed-3-Gram* model equations are,

$$\begin{aligned} T_{code} = {} & T(I_a|**) + T(I_b|*I_a) + T(I_c|I_aI_b) + T(I_b|I_bI_c) + T(I_d|I_cI_b) + T(I_a|I_bI_d) \\ & + T(I_b|I_dI_a) + T(I_d|I_aI_b) + T(I_c|I_bI_d) + T(I_a|I_dI_c) + T(I_b|I_cI_a) + T(I_d|I_aI_b) + \\ & T(I_c|I_bI_d) + T(I_a|I_dI_c) + T(I_b|I_cI_a) + T(I_d|I_aI_b) \qquad (3.3) \\ E_{code} = {} & E(I_a|**) + E(I_b|*I_a) + E(I_c|I_aI_b) + E(I_b|I_bI_c) + E(I_d|I_cI_b) + E(I_a|I_bI_d) \\ & + E(I_b|I_dI_a) + E(I_d|I_aI_b) + E(I_c|I_bI_d) + E(I_a|I_dI_c) + E(I_b|I_cI_a) + E(I_d|I_aI_b) + \\ & E(I_c|I_bI_d) + E(I_a|I_dI_c) + E(I_b|I_cI_a) + E(I_d|I_aI_b) \qquad (3.4) \end{aligned}$$

where $E(I_d|I_aI_b)$ represents the energy cost for instruction I_d in the context of I_a and I_b, with I_a and I_b preceding instruction I_d in that order. To determine per-instruction energy costs and execution time of the target processor, several empirical benchmarks were constructed to solve for instruction-level costs.

The data in Table 3.1 is for the AMD Athlon instruction set as generated by GCC. It is very significant that GCC was used to generate the code; there are many different Athlon instruction types as determined by opcode, but only 34 1-Grams are in GCC's idiom. Using *Fixed-N-Gram* instruction-in-context types dramatically increases the number of distinct types of instruction-level objects, as shown in Table 3.1. With 34 distinct instruction 1-Gram classes, there are actually $34 \times 35 = 1,190$ distinct instruction 2-Gram types. The number of distinct N-Grams for K distinct 1-Grams is $K \times (K+1)^{N-1}$, where the $+1$ is due to inclusion of the $*$ symbol for an unspecified context instruction when $N > 1$. Figure 3.2 show the rate at which the number of N-Grams grow as N increases.

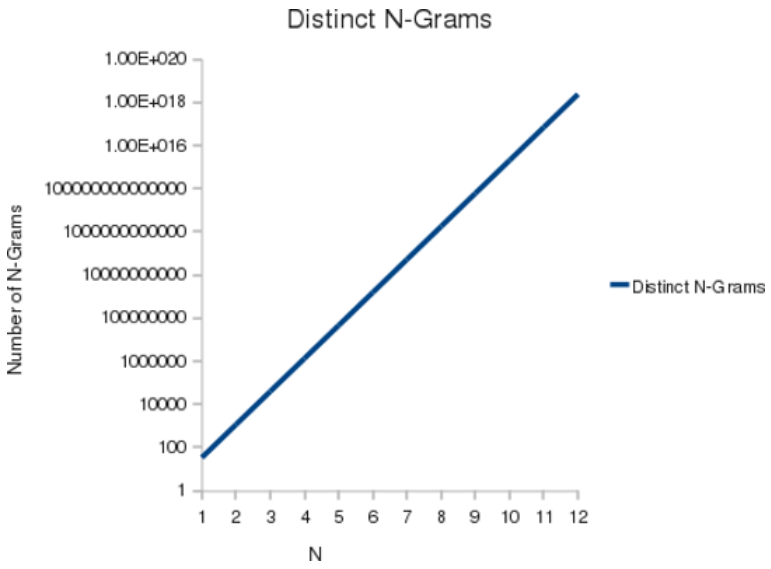

Figure 3.2: Number of N-Grams

Fortunately, the set of N-Grams that actually occur in human-written and mechanically compiled code grows somewhat more slowly; although 100% of the possibly distinct instruction 1-Grams are present in the benchmark suite described by Table 3.1, only 49% of the possible 2-Grams are present and just 16% of all possible 3-Grams actually occurred. Note that N-Grams which did not occur in our measurements can still have predicted values – for example, data for the 2-Gram $(I_a I_b)$ can be approximated as $(*I_b)$ if $(I_a I_b)$ did not occur in our measurements. Thus, although the table data does not reflect this optimization, it also is possible to set an arbitrary maximum on the number of N-Grams used for the analysis by merging excess entries into the corresponding $*$ entries. Even more data reduction is possible using N-Grams described by patterns, for example, merging $(I_a I_b)$ and $(I_c I_b)$ into a single $((I_a|I_c)I_b)$ type if $(I_a I_b)$ and $(I_c I_b)$ are found to have a similar property – without implying that I_a and I_c have other properties in common.

Table 3.1: Empirical Scaling of N-Gram Complexity

N	Instruction Types	Distinct N-Gram Types	N-Gram Types Present
1	34	34	34 (100%)
2	“unknown”+34	1,190	587 (49%)
3	“unknown”+34	41,650	6,604 (16%)

Statistical collection of N-Gram instructions were obtained using 10,000 idiomatically correct synthetic benchmarks. This number was chosen as an upper bound to cover the number of distinct instruction types generated by *gcc* and also to cover the practical limit of $N = 3$ and

Algorithm 3.1 Cost Estimation of Individual Instruction

Let T be the set of all distinguishable instruction types t.
Let C_t be the cost of an instruction of type t.

1. Create $n | n \gg |T|$ random simple source code program fragments, $S_0..S_{n-1}$, such that each distinguishable instruction type is likely to appear in many of the programs
2. Using the same compiler and optimization options normally used for applications, compile each program S_i into an assembly language segment A_i that has L_i lines of code; this is important because it maintains the compiler's idiomatic instruction patterns
3. Let $a_{it\,j}$ represent the assembly instruction of type t appearing at line j in segment A_i.
4. Wrap each segment A_i in a timing framework to create program F_i
5. Execute each F_i to obtain an equation of the form $k_i = \sum_{t \in T, j \in L_i} a_{it\,j} \times C_t$, where k_i is the empirically determined average total cost (energy/time) to execute segment A_i
6. Solve the set of overspecified equations resulting from Step 5

$6,604$ *Fixed-3-Grams*. If the synthetic benchmarks leave too many real application N-Grams covered using $*$ entries, then either portions of the applications themselves or new synthetic benchmarks can be added to the measurement suite.

3.5.1 Experimental Procedure

To obtain valid timing and energy statistics for the instruction-level *Fixed-N-Gram* model that can be used to analyze any application program, our hope is that it is sufficient to obtain accurate execution time and energy measurements for a suite of benchmark programs that was generated as in section 3.3. The basic algorithm is described in Algorithm 3.1, where cost refers to either execution time or energy values.

The set of equations resulting from Step 5 is grossly overspecified. Thanks to noise, cache effects, and other details that are not necessarily controlled nor measured, it is unlikely that a single, consistent, solution can be found by standard simultaneous equation solving methods. To find a solution that minimizes error in some sense, we have considered the method of least squares, Chahine's algorithm [21], and a Evolutionary Algorithm search. Solving the equations with Least Squares using Singular Value Decomposition (SVD) does not guarantee all solutions be non-negative; some instructions were given negative time estimates. The method proposed by Chahine [21], directly enforces all non-negative solutions, but it could not handle large

systems (i.e., $10,000 \times 6,604$ needed for *Fixed-3-Gram* analysis). Evolutionary search on the other hand can enforce all non-negative solutions and, empirically, works best.

3.5.2 Solving Fixed-N-Gram Model Using Genetic Algorithm

Evolutionary search techniques such as Genetic Algorithm (GA) and Genetic Programming (GP) techniques have been used in the scientific and engineering community whenever a design problem has a huge solution search space. An evolutionary algorithm starts off with thousands of random individuals (values or computer programs), that evolve over generations using biologically inspired operators such as crossover and mutation. A problem-specific fitness function determines the best individual every generation. The number of generations or steady-state death/birth events to evolve depend on the complexity of the problem and the availability of computing resources. Individual population members in GP are usually represented as variable-size syntax trees corresponding to the programs and most of the GA techniques represent individuals as fixed-length bit-strings or arrays of values (integer, floating point etc.).

A Genetic Algorithm (GA) was used to solve the *Fixed-N-Gram* model for calculating instruction-level time and energy metrics. Separate GA searches were used to solve for energy and time independently. The search for cost metrics starts with a random population of non-negative performance estimates for all the instruction types and N-Gram patterns. The fitness of each individual can be computed by either summing the squares of the differences between estimated and actual cost for all of the benchmark code segments or by calculating the average percentage error between the actual cost and the model estimated cost of over all the benchmark programs. Three members (2 best and 1 worst), were selected from the pool through tournament selection from a random subset of individuals in the population.

The selected worst member is replaced by a new member for the next iteration, through value crossover or mutation. Value crossover is performed either by combining the mantissas of the estimates of the two best fit members using bitwise operations or by using arithmetic operations on the two estimates. Value mutation is similar to value crossover except the second best member is replaced by a randomly generated member. The constraint of non-negativity is preserved while performing these operations and care is taken not to replace the best individual found so far. The evolutionary process repeats until a fixed number of iterations has passed. Number of iterations and the % of individuals to compete in the tournament are user-supplied search parameters.

3.5.3 Energy and Timing Analysis Using Fixed-N-Gram Model

Experimental analysis of energy and time for the N-Gram models was performed using GCC and an AMD Athlon processor in a typical desktop configuration. Note, however, that we do not suggest that all Athlons have the same energy and time model, nor even that all versions of GCC do given the same target. Part of why automation of model creation is so important is that a new model must be created for each (possibly apparently subtle) change in configuration. In parallel computing, it is normal practice that there will be many identical nodes in a system; thus, we suggest that the analysis could be performed once and applied to all nodes. The analysis would only need to be repeated if nodes with a different configuration were added or the configuration of existing nodes was changed.

Figure 3.3 shows the accuracy of predicting execution time using a *Fixed-1-Gram* model and figure 3.4 shows the accuracy using instruction classification incorporating addressing modes, both for 10,000 synthetically-created idiomatically correct benchmark test cases. One would expect that incorporating addressing modes would make a huge difference because it would distinguish instructions operating on registers from those accessing memory – but the difference was not huge. The average difference between measured and predicted execution time was 9.75% without considering addressing modes and improved to 8.13% as the number of instruction classes was expanded. Table 3.2 shows the execution time values for some sample instructions using the above two *Fixed-1-Gram* models. As can be seen from the table, the *Load* (`movl_ld`, `subl_ld`) and *Store* (`and_st`) instructions accounted for a very small increase in their execution times when modeled using the addressing mode.

Table 3.2: Execution Time for *Fixed-1-Gram* Models

Instructions	Execution Time (Secs)	Instructions with addressing mode classification	Execution Time (Secs)
andl	1.03497×10^{-9}	andl_st	4.67243×10^{-9}
addl	4.68748×10^{-10}	addl_imm	2.60671×10^{-10}
movl	4.7517×10^{-10}	movl_ld	6.10613×10^{-10}
imull	3.66758×10^{-9}	imull	3.85621×10^{-9}
subl	6.36263×10^{-10}	subl_ld	1.23661×10^{-9}
negl	8.11159×10^{-11}	negl	5.00433×10^{-11}

Similarly, Figures 3.5, and 3.6 show how accurately both the *Fixed-1-Gram* models predict energy consumption. The energy predictions had an average error of 7.68% and 6.12% respectively for the two models. It can be seen from the graphs that some of the test cases that were

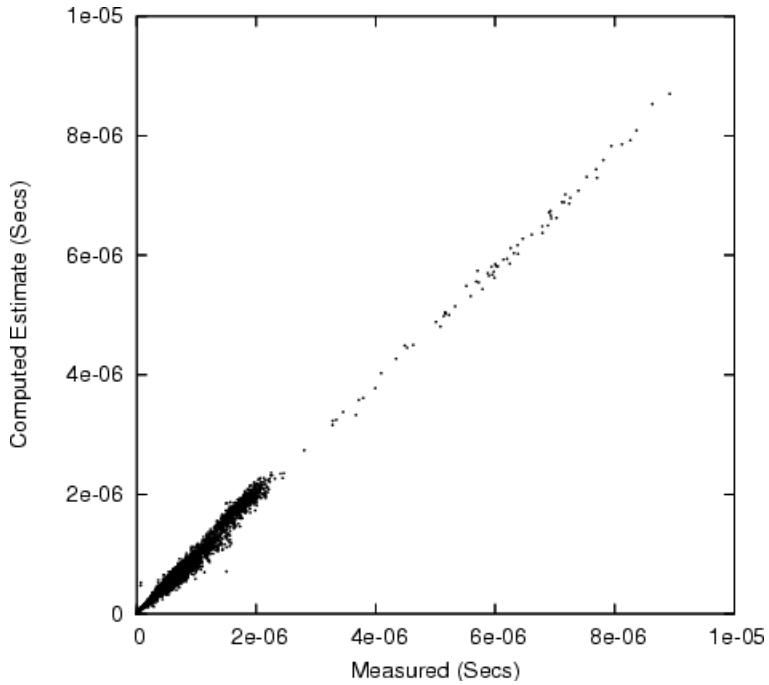

Figure 3.3: Accuracy of Time Estimation using *Fixed-1-Gram* Model

underpredicted without the addressing mode classification get closer to the actual value when using addressing mode classification, which in turn can be the reason for the decrease in the average error. From Table 3.3, one can see that the memory access instructions were calculated to consume more energy. However, the improvement in accuracy by distinguishing many more instruction classes is modest.

Table 3.3: Energy Consumption for *Fixed-1-Gram* Models

Base Model Instructions	Energy Consumption (Joules)	Instructions with addressing mode classification	Energy Consumption (Joules)
andl	5.02657×10^{-9}	andl_st	2.78159×10^{-8}
addl	2.21599×10^{-9}	addl_imm	1.29141×10^{-9}
movl	1.44201×10^{-8}	movl_ld	1.91279×10^{-8}
imull	2.41399×10^{-8}	imull	2.40212×10^{-8}
xorl	3.00679×10^{-15}	xorl_ld	1.85382×10^{-8}
subl	7.79287×10^{-9}	negl	4.00407×10^{-8}

The GA search converges on a good solution more quickly when it starts out with a population that is close to optimal. *Fixed-N-Gram* models were constructed for the synthetic benchmarks for $N = 2$ and $N = 3$ and to speed convergence on the 2-Gram and the 3-Gram models, the initial population is seeded with the results from 1-Gram and 2-Gram GA searches respectively. It

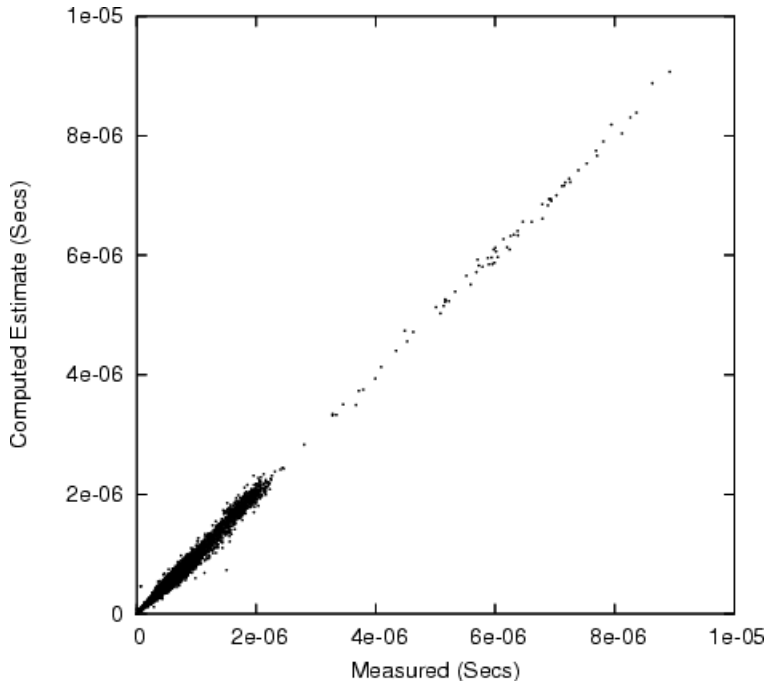

Figure 3.4: Accuracy of Time Estimation using *Fixed-1-Gram Model* with Addressing Mode Classification

was found that the average error rate for any of the properties (time and energy) did not improve significantly with inclusion of this modest amount of context information. More precisely, the GA was unable to make a significant improvement beyond this initial estimate of the solution – the numbers are *virtually identical* to the 1-Gram figures.

3.6 Instruction-Level Variable-N-Gram Cost Model

The amount of context information for all possible instruction types appearing in the benchmark code segments and the number of benchmarks required in turn to determine the average error can easily exceed the practical computational limits, when N gets larger for Fixed-N-Gram models. Even though Table 3.1 in section 3.5 shows only a fraction of the distinct N-Gram types are present in the benchmarks, the complexity of analyzing those types still is incurred. For example, if we consider $N = 10$ and assume just 1% of the 10-Grams to be present in the benchmark code segments, the model still has to analyze 1% of the 2,679,731,714,843,750 instruction-in-context types and test cases. Due to the exponential complexity involved in analyzing the *Fixed-N-Grams,* the model was constructed only for typical values of $N \leq 3$, which is often the case for linguistic N-Gram analysis. As modern processors employ very deep pipelines and dynamic out-of-order instruction execution, it seems likely that a context larger than $N = 3$ will be needed in order to capture enough information about the

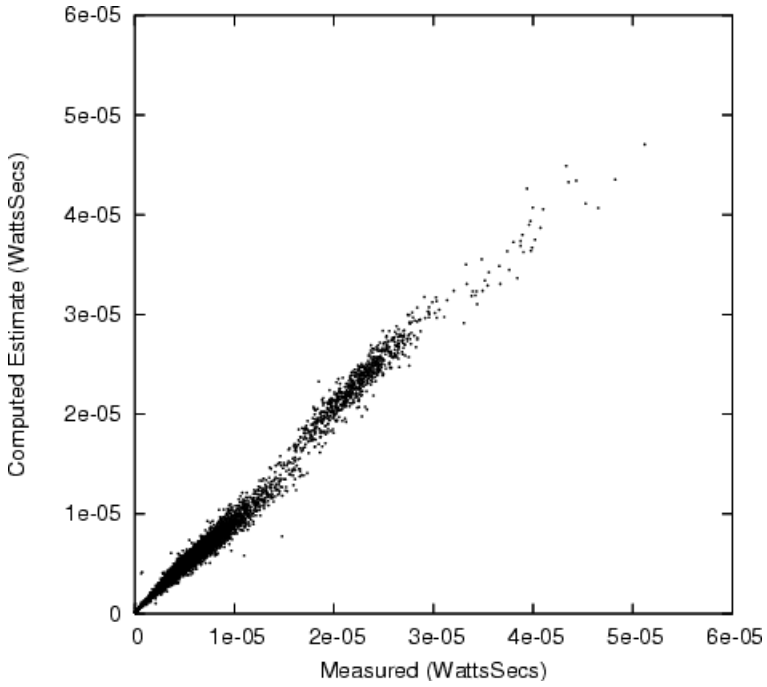

Figure 3.5: Accuracy of Energy Estimation using *Fixed-1-Gram* Model

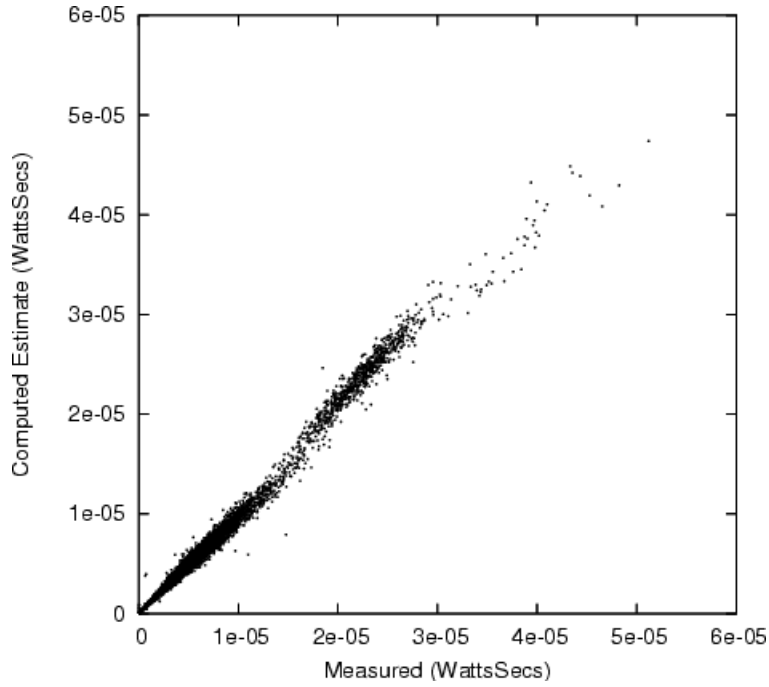

Figure 3.6: Accuracy of Energy Estimation using *Fixed-1-Gram Model* with Addressing Mode Classification

relevant internal state. With the complexity of *Fixed-N-Gram* model increasing exponentially with increasing context, one way to include more context information at a reduced complexity, is to develop a model where cost of some instructions can be determined by including a larger context window (for example, 100-Grams) while some others can still have less or no context at all. We call this a *Variable-N-Gram* model because of its ability to determine performance cost of individual instructions within a variable-size context of instruction execution. The creation of a model that extends context only where needed, is very complex, but we have automated the process (see section 3.6.1) and the resulting model is computationally efficient. One possible *Variable-N-Gram* model equations for the above sample code segment for $N \leq 5$ is given as,

$$\begin{aligned} T_{code} = {} & T(I_a) + T(I_b) + T(I_c|I_b) + T(I_b) + T(I_d) + T(I_a|I_bI_d) + \\ & T(I_b) + T(I_d) + T(I_c) + T(I_a|I_aI_bI_dI_c) + T(I_b|I_a) + T(I_d|I_cI_aI_b) + \\ & T(I_c) + T(I_a) + T(I_b) + T(I_d|I_cI_aI_b) \end{aligned} \tag{3.5}$$

$$\begin{aligned} E_{code} = {} & E(I_a) + E(I_b) + E(I_c|I_b) + E(I_b) + E(I_d) + E(I_a|I_bI_d) + \\ & E(I_b) + E(I_d) + E(I_c) + E(I_a|I_aI_bI_dI_c) + E(I_b|I_a) + E(I_d|I_cI_aI_b) + \\ & E(I_c) + E(I_a) + E(I_b) + E(I_d|I_cI_aI_b) \end{aligned} \tag{3.6}$$

As can be seen from the above equations, some instructions were recorded with no context while others have context information of up to four instructions. For example, the sixth term in the energy equation $E(I_a|I_bI_d)$ represents a 3-Gram instruction context for I_a with $I_b$and I_d preceding it in order. Similarly, $E(I_a|I_aI_bI_dI_c)$ represents a 5-Gram energy term for I_a with instructions I_a, I_b, I_d, I_c preceding it in that order. It has to be noted that, depending on the value of N and the random selection of instruction sequences to examine the performance relationships, the model equations differ and so does the number of variables.

The main idea of both the *N-Gram* cost models is to allow a linear system of equations to be used to determine the cost. In 1-Gram models, where instruction I_a takes the same value throughout the code segment, the cost is entirely linear; in the *Fixed-3-Gram* and the *Variable-5-Gram* models described above, I_a takes different values depending on the context information before it, possibly hiding non-linear relationships within the costs of individual Grams. The algorithm for estimating performance metrics of individual instructions using the *Variable-N-Gram* model is similar to the algorithm 3.1 described in section 3.5.1. However, a hybrid evolutionary algorithm as described in the following section is used in place of a simple GA search to solve the resulting system of equations corresponding to the *Variable-N-Gram* model.

3.6.1 Solving Variable-N-Gram Model using a Hybrid Evolutionary Algorithm

Even though solving a *Variable-N-Gram* model is not as computationally infeasible as solving a *Fixed-N-Gram* model for larger N, the search space in determining optimal values of N for different instruction types is still huge. In order to solve the *Variable-N-Gram* model at a relatively reduced complexity, we again resort to evolutionary computation techniques. This evolutionary search technique to solve the *Variable-N-Gram* model is not straight forward - in that it combines properties of both the GA and GP.

Technically, the approach is a GA, but looks more like GP. Much like GP, the genome for each individual is represented as a variable-size, variable-shape, tree. However, nodes do not represent operations as in GP; instead, each node represents a particular type of instruction in a tree of possible predecessors to the instruction whose cost the tree represents. Each node is attributed with cost – either energy or time values – and these costs also evolve as the tree changes size and shape. A path in a tree represents a pattern of instructions that might appear in the code segment working backward for a maximum length of N instructions such that an N-node walk represents an N-Gram. Of course, different length paths yield different values of N.

The tree in figure 3.7 represents one of many possible genome structures for a population member. Any individual instructions and patterns that are not covered by the genome structure fall into the $*$ category – also known as the "unknown" category. To differentiate the performance costs of uncovered instructions, each could be treated as a separate instruction category instead of a single $*$ category as shown in figure 3.8, but the system is capable of evolving such distinctions automatically. Because the context grows in the backward direction, a N-Gram pattern is formed reading from a leaf node to the root. Resultant patterns out of this genome for instructions ending in I_a and I_d are: $I_aI_bI_dI_cI_a$, $I_bI_dI_cI_a$, $I_dI_cI_a$, I_cI_a, $I_bI_dI_a$, I_dI_a, I_a, and $I_cI_aI_bI_d$, $I_aI_bI_d$, I_bI_d, I_d respectively.

To derive performance model equations for energy E_{code} using the above genome, let us reconsider the sample benchmark code segment from section 3.2, (i.e.)

$$I_a, I_b, I_c, I_b, I_d, I_a, I_b, I_d, I_c, I_a, I_b, I_d, I_c, I_a, I_b, I_d$$

Uncovered instructions I_b and I_c are treated separately and the energy model equation is:

$$\begin{aligned} E_{code} = {} & E(I_a)+E(I_b)+E(I_c)+E(I_b)+E(I_d|I_b)+E(I_a|I_bI_d)+E(I_b)+E(I_d|I_aI_b)+E(I_c)+ \\ & E(I_a|I_aI_bI_dI_c)+E(I_b)+E(I_d|I_cI_aI_b)+E(I_c)+E(I_a|I_aI_bI_dI_c)+E(I_b)+E(I_d|I_cI_aI_b) \end{aligned} \tag{3.7}$$

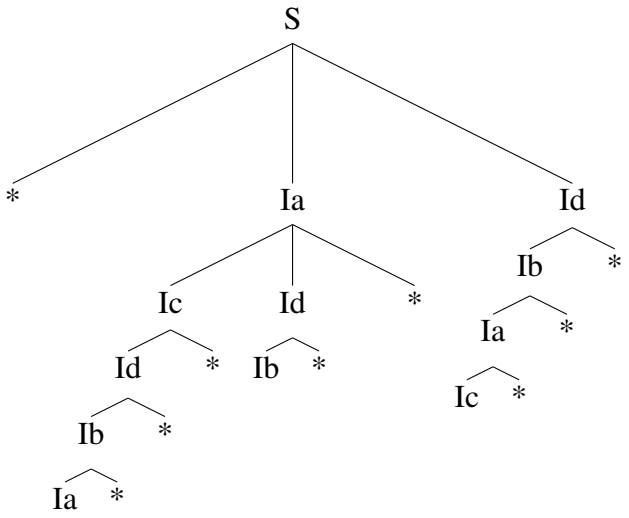

Figure 3.7: Genome of a Population Member with Uncovered Instructions

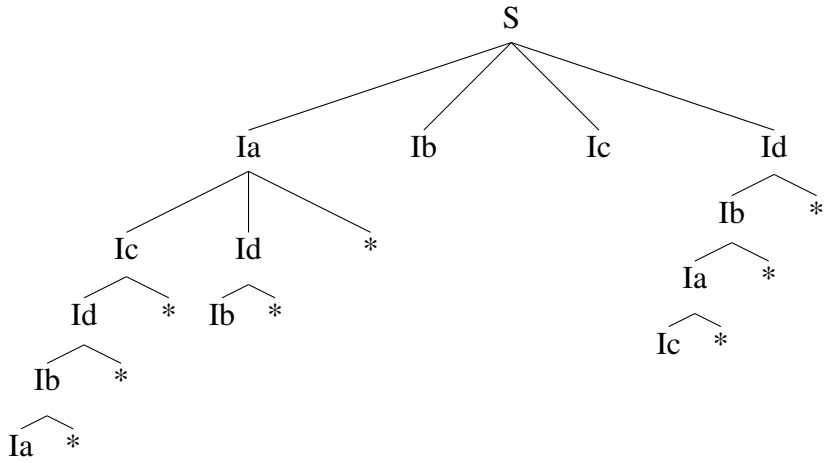

Figure 3.8: Genome of a Population Member with Covered Instructions

Similar performance model equations can be constructed for T_{code} as well. It is useful to note, however, that the N-Grams used to predict one cost measure need not be the same as those used to predict another. It is perfectly acceptable to have one tree for energy estimation and a completely different one for time estimation. Genome structures undergo fitness estimation where fitness of each individual is determined by the average sum of absolute errors for the code segments. In our steady-state evolutionary process using tournament selection, the worst fit member of the tournament gets replaced by crossing two members or mutating one member out of the best fit ones that participate in the tournament. In GP, tree branches are typically chosen randomly to perform crossover or mutation. Here, the patterns are chosen for crossover or mutation with a bias towards reducing the average error. The process is as follows:

1. A small number of benchmark code segments are selected to use for evaluating individual patterns.

2. Select the single pattern, out of the worst fit member's patterns, that contributed the highest percentage towards the overall average error in the equations for the benchmarks selected. If no such pattern exists, a random pattern out of the worst fit member's patterns is selected.

3. During pattern crossover, this selected pattern is replaced by one of the patterns of the best fit member, and the performance cost of this pattern also gets changed to a different value by performing crossover on the performance values of two best members that were selected through tournament selection. Value crossover is performed either by combining the mantissas of the estimates of the two best fit members using bitwise operations or by using arithmetic operations on the two estimates. The worst-fit member undergoes pattern mutation by replacing the selected pattern with a randomly selected pattern. The performance metrics are mutated through random bit manipulation on the mantissas or through some arithmetic operations. Figure 3.11 shows the resultant child of one sample pattern crossover operation, and the selected patterns out of Parent1 and Parent2 as shown in figures 3.9 and 3.10 are shown in parentheses. The resultant Child is the same as Parent1 except pattern I_bI_c gets replaced by $I_bI_dI_aI_c$ of Parent2. Mutation in figure 3.12 is similar except instead of copying a pattern from the best-fit member, a random pattern gets included in the worst-fit member's set of patterns.

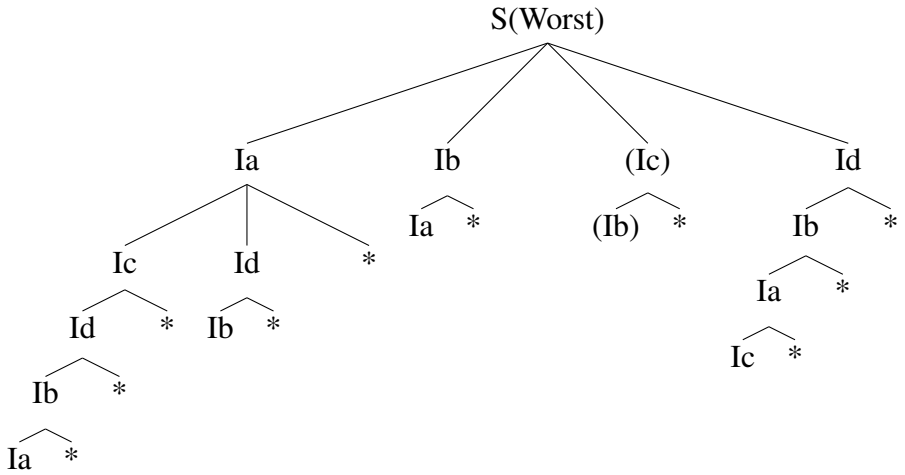

Figure 3.9: Parent 1 - Pattern Crossover

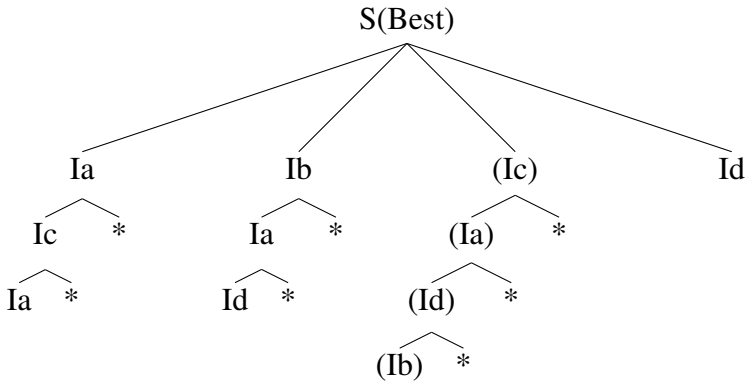

Figure 3.10: Parent 2 - Pattern Crossover

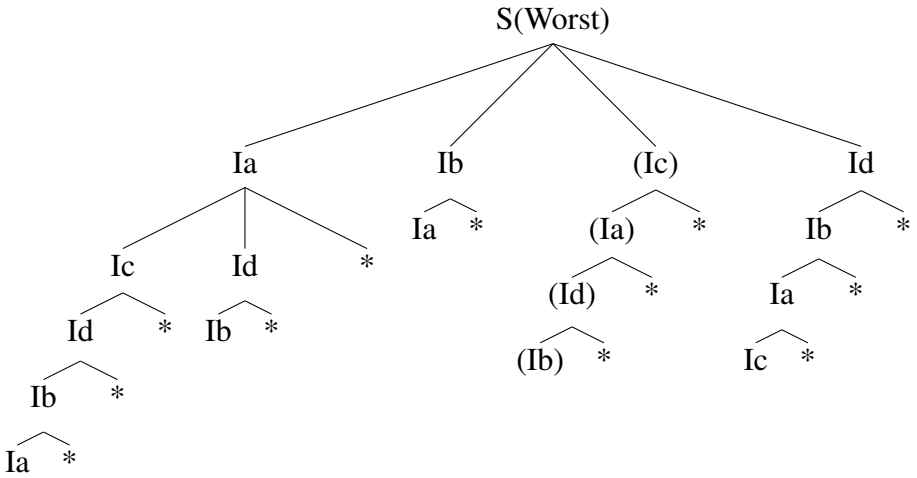

Figure 3.11: Resultant Child - Pattern Crossover

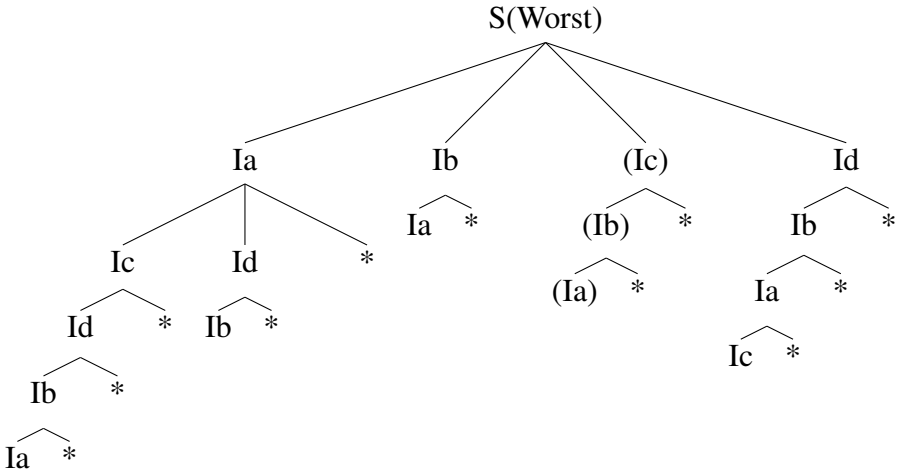

Figure 3.12: Pattern Mutation

3.6.2 Validation Through Reverse Engineering

In theory, the *Variable-N-Gram* approach should be able not only to evolve good costs, but also to evolve the appropriate values of N for each sequence. In other words, it should not only create good estimates, but also a reasonable structural model of the interactions between instructions. To validate the algorithm, it is necessary to know not only costs, but also what the structure should be. Unfortunately, such data is not readily available, so a test case was synthesized using specific costs on particular N-Grams – essentially the generative use of one of these models. One hundred test cases were constructed using four simple instructions $\{ld, st, op, br\}$ to be consistent with the patterns $\{ld.ld, st.ld, ld.op.ld, ld.br.ld, br.br\}$ and the weights assigned to them. The set of overspecified *Variable-N-Gram* model equations that resulted out of the constructed sample test cases were solved using the hybrid evolutionary algorithm for energy and time separately.

The evolutionary algorithm resulted in 0.86% average error for energy. Table 3.4 shows the generative and automatically-determined energy costs for the instructions and patterns. As can be seen from the table, the evolutionary algorithm did not create a special case for the *br.br* $(1.5 \times e^{-08})$ pattern, instead it made a pattern *br.br.br* that had the energy cost of $1.39 \times e^{-08}$. Except for this pattern, the algorithm was effective in evolving both the given patterns and costs. Figure 3.13 shows the pattern tree (1-Gram patterns *st* and *op* not shown) derived by the evolutionary algorithm for determining instruction-level energy costs. As expected, it also has additional patterns that were harmless but not needed to distinguish the energy costs of the code segments. The algorithm was run separately for determining the instruction-level time costs, yielding an average error of 1.35% – not as good as for energy, but still quite acceptable. Table 3.5 gives the time costs and figure 3.14 gives the pattern tree for time metric. These results are encouraging considering the small number of generated benchmarks from which the model was evolved. The greater the number of equations, the higher the quality of the cost and structural match.

Table 3.4: Energy Costs from Evolutionary Algorithm (EA)

Instructions/Patterns	Energy (Given)	Energy (EA)
ld	$1.5 \times e^{-08}$	$1.45 \times e^{-08}$
st	$1.5 \times e^{-08}$	$1.56 \times e^{-08}$
op	$3.0 \times e^{-08}$	$3.07 \times e^{-08}$
br	$1.5 \times e^{-08}$	$1.45 \times e^{-08}$
ld.ld	$4.5 \times e^{-08}$	$4.4 \times e^{-08}$
st.ld	$3.0 \times e^{-08}$	$2.95 \times e^{-08}$
ld.op.ld	$4.5 \times e^{-08}$	$3.78 \times e^{-08}$
ld.br.ld	$4.5 \times e^{-08}$	$4.31 \times e^{-08}$
br.br	$1.5 \times e^{-08}$	–

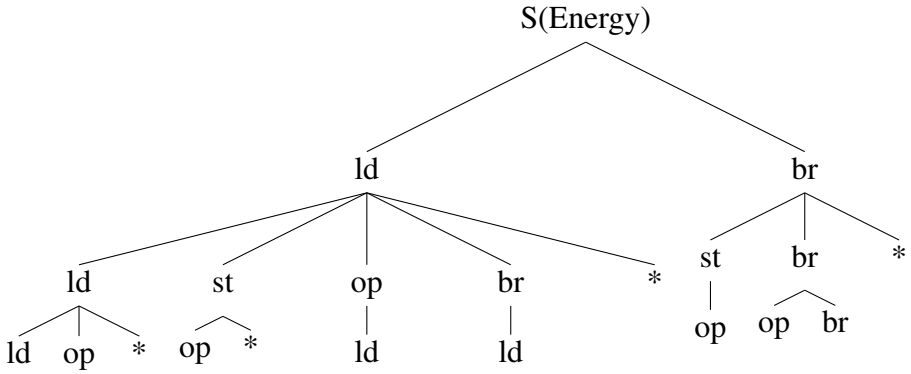

Figure 3.13: Pattern tree for Energy

Table 3.5: Time Costs from Evolutionary Algorithm (EA)

Instructions/Patterns	Time (Given)	Time (EA)
ld	$3.0 \times e^{-10}$	$3.52 \times e^{-10}$
st	$1.5 \times e^{-10}$	$1.74 \times e^{-10}$
op	$1.5 \times e^{-10}$	$1.42 \times e^{-10}$
br	$4.5 \times e^{-10}$	$4.37 \times e^{-10}$
ld.ld	$6.0 \times e^{-10}$	$6.24 \times e^{-10}$
st.ld	$4.5 \times e^{-10}$	–
ld.op.ld	$4.5 \times e^{-10}$	–
ld.br.ld	$3.0 \times e^{-10}$	–
br.br	$7.5 \times e^{-10}$	$7.63 \times e^{-10}$

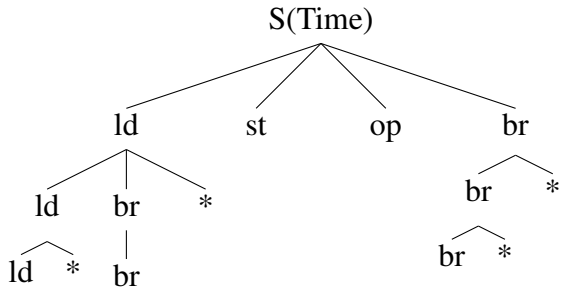

Figure 3.14: Pattern Tree for Time

3.6.3 Energy and Timing Analysis Using Variable-N-Gram Model

Having confirmed that the evolutionary algorithm works well in rediscovering an underlying structure and costs for a purely synthetic example, it is appropriate to determine if the approach can work as well where the underlying cost relationships are much more complex and not directly knowable. Experimental analysis for determining instruction-level costs using the *Variable-N-Gram* model, was also performed using the same version of GCC over AMD Athlon processor. The hybrid evolutionary algorithm described in section 3.6.1 was used to solve for the *Variable-N-Gram* model with maximum context set at $N = 100$. Figures 3.15 and 3.16 show the accuracy of time and energy estimates with the *Variable-N-Gram* model (without instruction classification based on addressing mode). Even though the graph looks identical to that of the *Fixed-1-Gram* models, the error rates of time and energy decreased from 9.75% to 9.71% and 7.68% to 7.58% respectively.

Given such a large context, the accuracy improvement achieved was disappointingly small. However, this result is quite encouraging in that it suggests that good predictions can be made even without considering context. The generated test in Section 3.6.1 showed that a model accurate to within 1% or 2% is easily recovered *when such model truly underlies the data*. Even if further improvements cannot be made, note that *all* the static predictions reported here are twice as accurate as the after-the-fact dynamic estimates made by Bellosa et al [1, 2], which made its measurements on a comparably complex modern system and achieved good results using the measures as a basis for control. Thus, the static prediction mechanisms reported here are more than adequate for use in predictively optimizing system performance, even if little or no context is considered.

It is likely that the lack of significant improvement with addition of large amounts of context may reflect the fact that the error rates obtained were already low enough so that any improvement was essentially below the noise floor of the experiment. The noise floor was essentially set by the measurement stability criterion, which assumed a 5% variation between measure-

ments was acceptable. However, it is likely that complex nodes have enough noise to make it impossible for *any* static mechanism to reliably achieve very highly accurate predictions. Given that the noise floor in executing the idiomatically correct synthetic benchmarks is far lower than the noise floor would be when executing in a more complex runtime environment, it may be that the fundamental limit on accuracy of predictions is destined to be the noise floor – not the cleverness and detail level of the static model.

The extracted instruction-level predictions have to undergo a static analysis with respect to the underlying program's state machine representation to form a prediction data structure. Following chapter 4 explains how the static analysis and long-range static prediction lookahead is done.

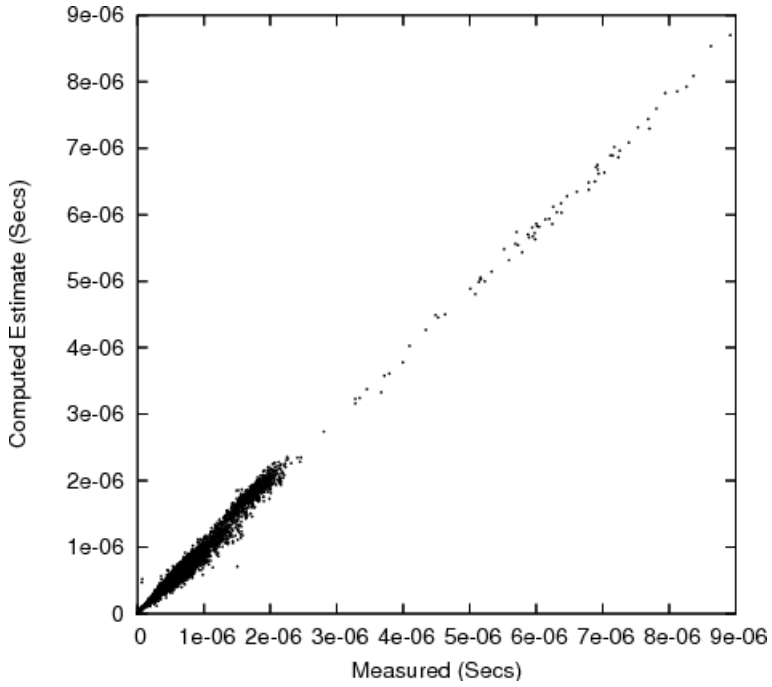

Figure 3.15: Accuracy of Time Estimation using Variable-N-Gram Model

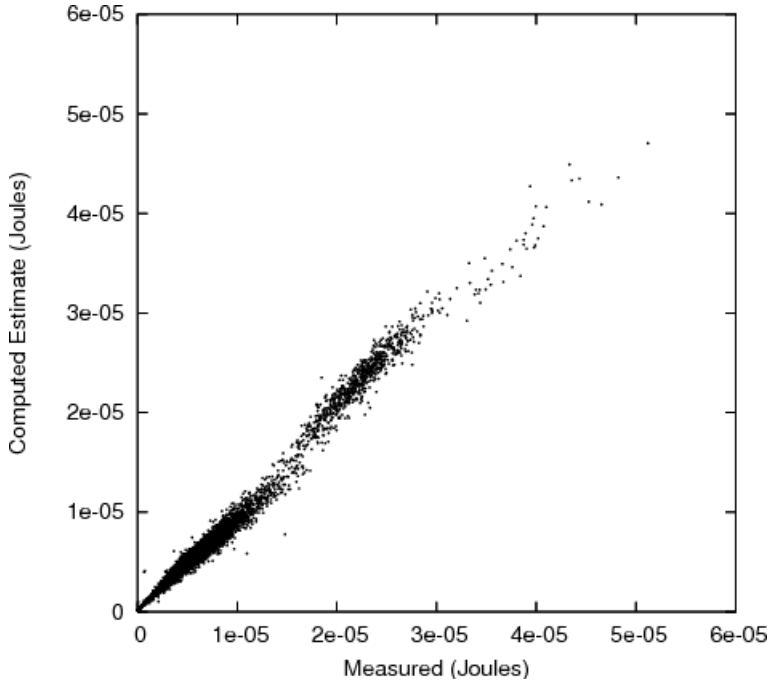

Figure 3.16: Accuracy of Energy Estimation using Variable-N-Gram Model

CHAPTER 4: STATIC ANALYSIS AND PREDICTION

Energy consumption being one of the major bottlenecks for hardware systems and runtime software, compiler researchers have been trying to develop compiler technology that treats power consumption as one of the optimization parameters. By predicting future energy behavior of the system at compile time, the system software can tune its actions accordingly to execute the anticipated behavior and to consume energy efficiently. For example, if the system software knows that the floating point (FP) unit is not going to be utilized for another millions of instructions, it can instruct the underlying hardware system to turn off the FP unit for some amount of time and turn it on just before when the system needs it.

Many runtime mechanisms attempt to predict future properties, but only static analysis has the theoretical ability to know everything that a program might do. The accuracy of prediction at runtime usually is poor because a very simple window-based lookahead mechanism is often used which might look a few instructions into the future (e.g., for out-of-order instruction pipeline scheduling), but primarily consists of historical data. At runtime, a complex lookahead mechanism might not be feasible because complex models consume more execution time and power, increase hardware and software complexity, and might not make significantly more accurate predictions. In contrast, static mechanisms theoretically can look forward as easily as backward and complex, slow, analysis is far more acceptable as an option at compile time rather than at runtime.

Compiler researchers have been focusing on developing technology [13, 35], that utilizes energy consumption as an optimization parameter and options to compile code to use less power. This dissertation does not concentrate on compiler technology for consuming less power, but explains how instruction-level estimates can be utilized to statically predict energy consumption. Modest levels of compile-time lookahead analysis are commonly used to improve the quality of generated code, for example, looking ahead a few to a few dozen instructions in order to find a better static schedule of instructions. Operating system scheduling events happen in intervals on the order of milliseconds – which corresponds to several thousands of instructions for processors with faster clock rates. Thermal properties of a system respond even more slowly, i.e. temperature at a sensor can continue to rise long after a heat source has had its power cut, so reactive control of power based on temperature monitoring must be very conservative. Power control based on long-range predictions using static analysis can be much more aggressive, getting the speed benefits of using more power without exceeding specified thermal limits.

This chapter discusses how the instruction-level predictions obtained in chapter 3 can be applied to static prediction analysis. Section 4.1 explains how a basic block state machine representation is constructed for a program under test and describes the procedure for annotating the states with instruction-level properties such as energy and time. The basic prediction lookahead analysis and the lookahead algorithm developed by Dietz et al [17] is expanded and explained in sections 4.2 and 4.2.2. An example that walks through the algorithm in detail is explained next followed by section 4.2.3 that provides static instruction-level predictions for sample programs.

4.1 State Machine Representation and Annotation

The lookahead analysis described in this chapter not only spans sequences of instructions, but also crosses all types of control flow constructs, including conditionals, loops, function calls, and even recursion. Dietz [16] developed a transformation technique to support speculative predication across arbitrary control flow. The same basic analysis is used here to transform the programs into their respective state machine representation. For example, Figure 4.1 shows the pseudo code of a sample C program containing a recursive function call. A function call is really just a `goto` accompanied by some data manipulation; function `return` is essentially a computed `goto` based on the data saved when the function was called. The resulting state transition graph is shown in figure 4.2 which can then be annotated with instruction level predictions. These predictive annotations can in turn be used by the runtime system on demand or can be embedded in the code to make static predictions available at runtime. This tool can be dynamically loaded by the loader to modify the object file and add the annotations to make the static predictions available at runtime.

```
main(....) {
        A g(....); B g(....); C
}

g(....) {
        D if (E) { F g(....); G} H
        return;
}
```

Figure 4.1: Sample Program

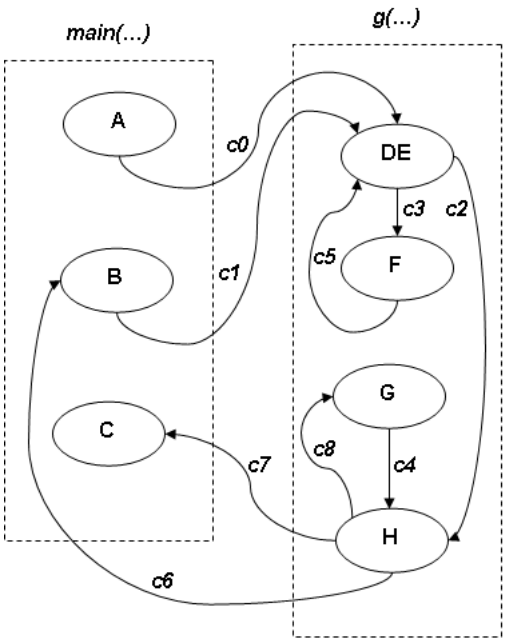

Figure 4.2: State Machine Graph

It is possible to directly take the assembly language output of a compiler as input to the static prediction algorithm, however, assembly language code output by a compiler like GCC is not necessarily a complete representation of the program – assembly code is generated per compilation unit. Thus, a program whose source code is spread across multiple files generally results in multiple assembly language files that must be analyzed together as a single entity in order to make complete predictions. The most reasonable input to a static prediction tool is a complete executable object file. Statically-linked object files generally contain all the routines that will ever be called by the program, thus, even paths through library code can be examined for prediction.

Using executable file as input introduces some additional complexity in that the code must be at least partially disassembled in order to identify the types of instructions used. However, tools like *objdump* can be used to produce the disassembled file, which in turn also provides some additional information, such as (logical) addresses for each instruction and data object. It translates the machine language contents of an executable file into a directly usable assembly code listing including logical code addresses and details about the instruction encoding. For example, many assembly languages allow a single assembly language instruction to represent a choice between several similar instructions with slightly different attributes, such as different length immediate constant operand encoding; the disassembly clearly distinguishes how these choices were made.

Whether the input is obtained from assembly language output of a compiler or from disassembly of an object file, there is a significant amount of preprocessing needed. This is done using a surprisingly complex set of `sed` and `awk` scripts for preprocessing:

- Extraction and normalization of the format of the relevant portion of the assembly listing
- Classification of instructions, and their respective properties for analysis
- Recognition of control flow instructions and their targets
- Construction of the basic-block state machine
- Annotating each state with the corresponding properties

A basic block contains no control flow: every instruction will be executed if any instruction is executed. Block edges can be found easily by analyzing and processing the assembly code. However, static analysis might not be able to get complete information about indirect jumps and calls in which the target address is the result of a computation. It is possible for a compiler to augment each such indirect control transfer with a list of possible targets:

- An indirect jump used to encode a `switch` or computed-`goto` statement has a set of possible targets that had to be known by the compiler in order for it to have generated that encoding
- An indirect call used to encode a dynamically-linked invocation of a library routine trivially is known to invoke that particular library routine; static linking of library functions might prevent generation of these indirect calls
- An indirect call generated by an explicitly programmed use of a function pointer can only reference a function whose address is explicitly taken

Such markings are not commonly found on indirect jumps and calls in assembly code output by compilers nor in assembly code created by disassembly of an executable object file. Fortunately, indirect control flow constructions are not common in most programs, so treating these constructs as preventing further lookahead analysis often is acceptable. In this analysis, indirect calls are treated as "expensive nops" and indirect jumps as "expensive returns." Similar special-case treatment also could be used for constructs such as calls to `longjmp()`, `exit()`, and various system calls.

4.2 Prediction Lookahead Analysis

This section discusses the long-range prediction of properties using a deep lookahead algorithm. Given the state machine described above in section 4.1, the propagation of instruction-level properties along paths is straightforward, but not necessarily very efficient. For example, Dijkstra's well-known shortest-path algorithm[19] can be used to find the next runtime event with good efficiency. Such an efficiency is achieved by placing constraints such as nodes in cycles (loops and recursions) need not be examined multiple times. However, finding the longest path up to a specified length may require many evaluations of nodes in cycles. It is very easy to code this type of algorithm such that a high-degree polynomial or even exponential time is required. Before explaining the algorithm in detail, the following section shows how energy consumption can be predicted for the small example in figure 4.1.

4.2.1 A small example

In general, let us consider each state in the flow graph having two basic properties of interest: X and Y. Let the path length be measured in units of X, and property Y be the attribute we wish to summarize over the complete path length. For example, X can be the number of instructions and Y can be the number of floating-point operations or the energy consumed, in which case our goal might be to determine the maximum number of floating-point operations that might be performed or maximum energy that might be consumed in executing the next several thousand instructions. Long range prediction of system properties at compile time can be done through the following steps :

1. Determine instruction-level properties statically for the underlying system as described in chapter 3
2. Compute basic block information and determine the state flow graph for the program under test
3. Compute system properties for each basic block state and annotate the state machine graph with calculated properties
4. Use long-range lookahead algorithm as described in section 4.2.2 below to predict system level properties of interest

Predicting energy consumption over several instructions can be explained for the small example in figure 4.1 as follows. Consider labeling each state in the flow graph with expected time

and power consumption as shown in figure 4.3. For state A, the power prediction over the next10 units of time can be calculated as follows. Starting from state A, program execution can flow through any of the following paths as shown in figures 4.4 through 4.8. For the path, $A->DE->F->DE->F$..... the energy prediction can be calculated as $((1*2+1*5+2*4+1*5+2*4+...)/(2+5+4+5+4+...))$. Since, we are interested in the first 10 units of time, the final prediction for this path yields 15 units of energy in 11 units of time $(15/11)$ at state F. The following table 4.1 shows different paths, terminal states and their energy predictions for the starting state of A for the next 10 time units. Prediction from path $A->DE->F$ can be termed as the final prediction because of the maximum energy consumed in minimum time. Figure 4.9 shows the final energy predictions for all states.

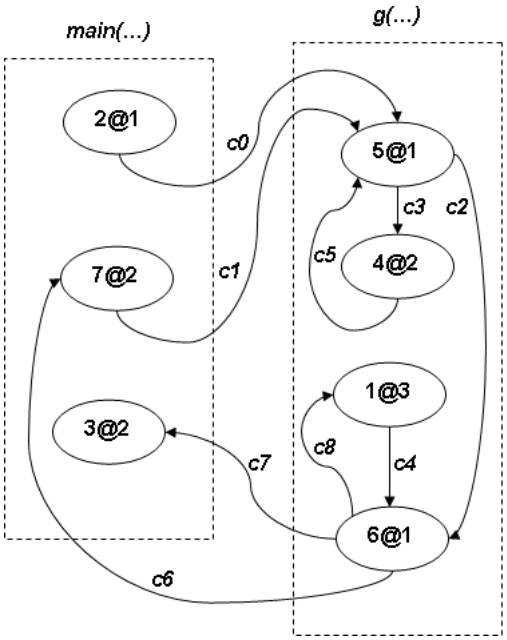

Figure 4.3: Timing @ Power Labeling

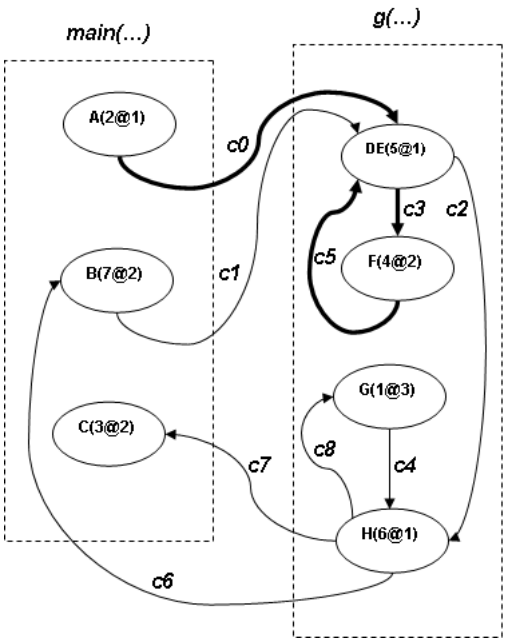

Figure 4.4: Execution Path A-DE-F-DE

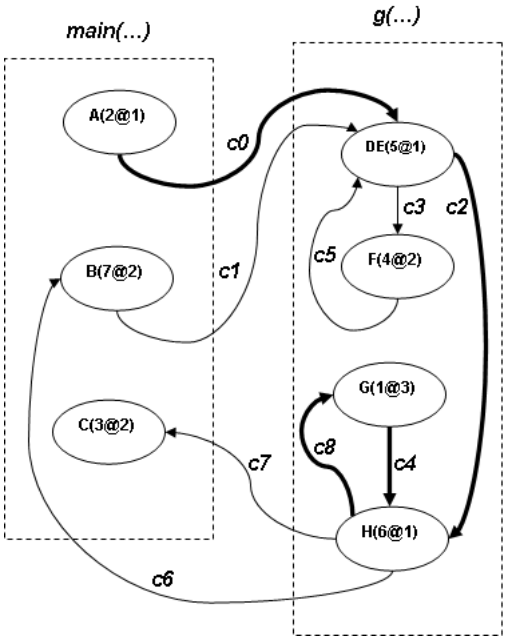

Figure 4.5: Execution Path A-DE-H-G-H

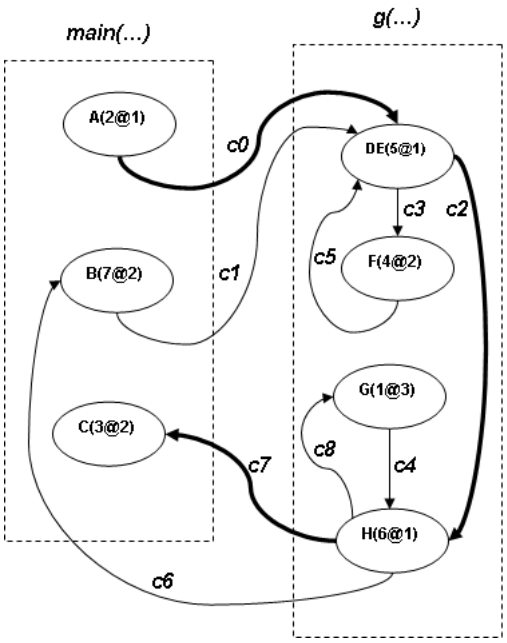

Figure 4.6: Execution Path A-DE-H-C

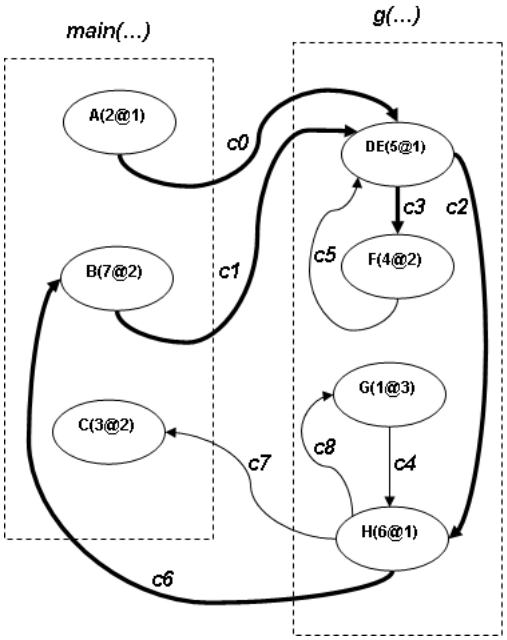

Figure 4.7: Execution Path A-DE-H-B-DE-F

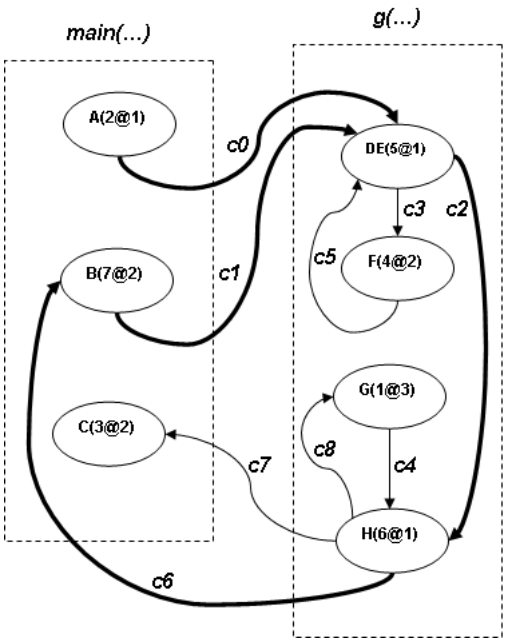

Figure 4.8: Execution Path A-DE-H-BE-DE

Table 4.1: State and Energy Prediction for Starting State A and Time=10 units

Paths	Energy/Time Prediction	Terminal State
A->DE->F	15/11	F
A->DE->H	13/13	H

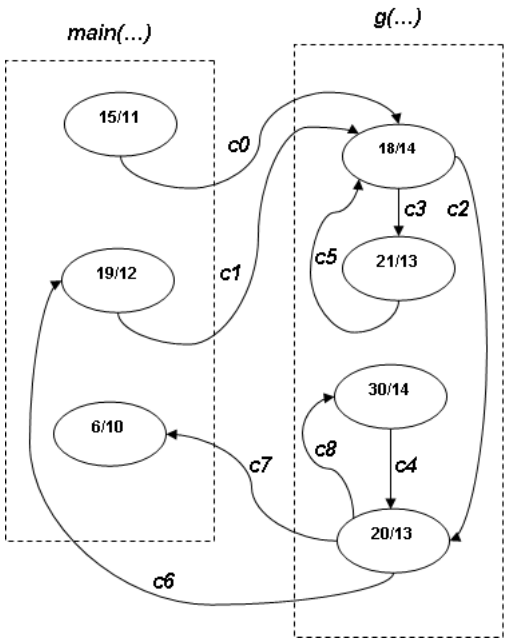

Figure 4.9: Energy Prediction for all States for Time=10 units

4.2.2 Lookahead Algorithm

The obvious depth-first search from each state in the state machine becomes computationally infeasible for very modest lookahead depths. However, a breadth-first search with careful pruning, as described in Algorithm 4.1, can be practical for thousands of instruction lookahead with typical state sizes. The efficiency comes primarily from the concept of truncating search of a path as soon as the path is known to be equivalent or inferior to another path that had the current state in common. If on one path state s is reached with properties a, b, then any future path that reaches state s with properties c, d such that $c \geq a \wedge d \leq b$ immediately can be pruned from the search.

Note that the algorithm collects all path endpoint tuples, so it is possible to use relatively sophisticated analysis of the endpoint set to determine the prediction, rather than simply taking the maximum value in any tuple. For example, if an end state contains 5 instructions, simply taking the maximum would be counting the contribution from all 5 instructions, whereas a smarter summary could adjust the value to count only the portion of the block that was within the lookahead range. An additional benefit of Algorithm 4.1 is that it easily can be modified to incorporate branching probabilities as weightings to applied when summarizing R.

Algorithm 4.1 Precise X-Bounded Y-Maximum-Sum Property Lookahead

Let S be the set of all states s.
Let x_s, y_sbe the X, Y properties of state s.
Let m be the maximum sum of x_s values required for any path through S.
Let $C(s)$ be the cover set of tuples for state s.
Let W be the working set of tuples.
Let R be the result set of path endpoint tuples.

1. For each state $s \in S$, repeat steps 2-18
2. Empty the cover sets: $\forall e | e \in S, C(e) = \{\}$
3. Place the starting state in the appropriate cover set:$C(s) = \{(x_s, y_s)\}$
4. Add the starting state to the working set: $W = \{(s, x_s, y_s)\}$
5. While $W \neq \{\}$, repeat steps 6-13
6. Pick a path position in the working set, $w | w \in W$, denoted $w = (s_w, x_w, y_w)$
7. Remove this path position from the working set: $W = W - \{w\}$
8. If $\exists (x_t, y_t) \in C(s_w) | x_t \leq x_w \wedge y_t \geq y_w$ then w is covered; continue the loop at step 5
9. Add this to the cover set: $C(s_w) = C(s_w) \cup \{w\}$
10. Reduce $C(s_w)$ by removing any entries which are covered by the new entry, i.e., for each $(x_t, y_t) \in C(s_w) | x_t \geq x_w \wedge y_t \leq y_w$, $C(s_w) = C(s_w) - \{(x_t, y_t)\}$
11. If $x_w \geq m$ then w is the end of a path; continue the loop at step 5
12. For each state $n | s \rightarrow n$ (i.e., which is a successor of s), repeat step 13
13. Add successor to the worklist: if $\nexists (x_t, y_t) \in C(s_w) | x_t \geq x_w + x_n \wedge y_t \leq y_w + y_n$, then $W = W \cup \{(n, x_w + x_n, y_w + y_n)\}$
14. Empty the result set: $R = \{\}$
15. For each state $n | n \in S$, repeat steps 16-17
16. All covers at least m long are terminal: for each $(x_t, y_t) \in C(n) | x_t \geq m$, $R = R \cup \{(n, x_t, y_t)\}$
17. All covers for a node without a successor are terminal: if $\nexists v | n \rightarrow v$, $R = R \cup C(n)$
18. Apply a user-defined function to summarize the prediction over R for state s ; for example, the summary might be the maximum $y_t | (n, x_t, y_t) \in R$

Most of the time, looking hundreds of instructions ahead is sufficient to drive some runtime hardware decisions. For example, dynamically turning power on and off for various processor subsystems can save power, but power changes do not happen instantaneously. Knowing that no floating point operations will occur in the next few hundred instructions would allow processor hardware to dynamically turn-off the floating-point unit to save power. Similarly, knowing when floating-point operations will resume would allow processor hardware to power up the floating-point unit enough *before* that point so that the unit is ready by the time the next floating-point instruction is dispatched to the unit.

Unfortunately, operating system control time constants are much longer than those of hardware, and a few hundred instructions lookahead is not sufficient to make runtime decisions. The key to improving lookahead efficiency is approximation. Rather than obtaining a precise lookahead answer for the minimum or maximum (as in Algorithm 4.1), it is usually sufficient to obtain an approximate answer. This is particularly effective when the expected error introduced by the approximate lookahead algorithm is small relative to the uncertainty with which the basic properties for individual blocks are known.

One way in which complexity of the lookahead analysis can be reduced dramatically is to simplify the state machine before performing the lookahead analysis. Simple rules for grouping of states are given in Algorithm 4.2. The precision of the lookahead analysis is closely related to the granularity of the states, so summarizing states might reduce precision by the factor in which the state size increases. Further, because expected execution counts are used to weight properties, accuracy of lookahead information can be adversely affected by poor estimates of branch probabilities or loop iteration counts. Purely static analysis methods can be used, but better estimates might be obtained using profiling tools like `oprofile` to observe branch probabilities during execution. Profile data could be used to refine predictions by regenerating the predictions after some number of runs of the program.

Algorithm 4.2 State Summary Rules

1. Any sequence of states $s_i \rightarrow s_j$ such that $\nexists s_k | s_k \neq s_j \wedge s_i \rightarrow s_k$ and $\nexists s_k | s_k \neq s_i \wedge s_k \rightarrow s_j$ can be merged into a single new state with properties $x_{s_i} + x_{s_j}, y_{s_i} + y_{s_j}$;
   this is the standard rule for maximizing size of basic blocks

2. Any sequence of states $s_i \rightarrow s_j \rightarrow ... \rightarrow s_k$ in which $s_i \rightarrow s_j$ implies that s_k will be reached is a nested region that can be collapsed into a representative state $s'_j | s_i \rightarrow s'_j$ such that $\forall s | s_k \rightarrow s$, $s'_j \rightarrow s$
   and s'_j has properties $\sum_{s \in s_j \rightarrow ... \rightarrow s_k} x_s \times E(s), \sum_{s \in s_j \rightarrow ... \rightarrow s_k} y_s \times E(s)$,
   where $E(s)$ is the expected execution count of state s given that $E(s_i) = 1$

3. In the special case where $s_i = s_k$, rule 2 replaces a cycle (loop) with a summary state; $E(s)$ can be greater than 1 due to multiplication by the loop's expected iteration count

4. Various standard compiler techniques can be used to restructure the graph to make the above rules apply in more productive ways

5. Any combination of the above transformations can be recursively applied

---

An alternative method for reducing lookahead analysis time is to propagate the path information backward. Because analysis of Algorithm 4.1 follows forward examination of the paths, there is no data shared across lookahead analysis starting in different states. Propagating properties in the opposite direction essentially processes paths from all starting states simultaneously. The result is Algorithm 4.3, which is capable of performing several thousand instruction lookahead computations in an acceptably short time – without requiring the aggressive state machine simplification suggested by Algorithm 4.2.

Algorithm 4.3 Approximate X-Bounded Y-Maximum-Sum Property Lookahead

Let S be the set of all states s.
Let x_s, y_sbe the X, Y properties of state s.
Let m be the maximum sum of X property values required for any path through S.
Let $C(s)$ be the cover set of tuples for state s,
organized as a minimum X property priority queue for each state.
Let L be the user-tunable maximum length of any priority queue;
smaller values of L speed up the algorithm with reduced accuracy.
Let r_s be the resulting prediction of maximum Y sum for any path starting at s .

1. For each state $s \in S$, initialize $C(s) = \{(x_s, y_s)\}$
2. While some state has a non-empty priority queue, $\exists e | e \in S, C(e) \neq \{\}$, repeat steps 2-12
3. Pick a working state $w | w \in S, C(w) \neq \{\}$;
   empirically, round robin selection works well,
   better if stochastically biased toward taking a w with a larger $|C(w)|$;
   this takes constant time
4. Remove the minimum x' entry, (x', y') from $C(w) : C(w) = C(w) - \{(x', y')\}$;
   this takes log time using the usual heap-based priority queue
5. Update the prediction for state w : if $r_w < y'$, then update $r_w = y'$
6. If $x' \geq m$, this path has reached the full lookahead length;
   continue the loop at step 2
7. For each state $n | n \rightarrow w$ (i.e., which is a predecessor of w), repeat steps 8-12
8. Compute (x'', y'') such that $x'' = x' + x_n$ and $y'' = y' + y_n$
9. An existing entry of $C(n)$ might cover (x'', y'') ;
   if $\exists (x''', y''') \in C(n) | x''' \leq x'' \wedge y''' \geq y''$ then (x'', y'') is covered; continue the loop at step 7;
   this cover check takes linear time in the (uncommon) worst case
10. An existing entry of $C(n)$ might be covered by (x'', y'') ;
    if $\exists (x''', y''') \in C(n) | x''' \geq x'' \wedge y''' \leq y''$ then (x''', y''') can be deleted;
    we have found it beneficial to only check cases where $x''' = x''$,
    in which case we replace (x''', y''') by (x'', y'') and continue the loop at step 7;
    in practice, this check can be incorporated in step 9
11. If $|C(n)| = L$, there is no space in the priority queue for a new entry;
    continue the loop at step 7
12. Make a new entry, $C(n) = C(n) \cup \{(x'', y'')\}$, and continue the loop at step 7

4.2.3 Instruction Lookahead for Sample Programs

This section first presents a detailed walk-through of the backward propagation algorithm for the small example in figure 4.1 for predicting maximum energy in $T = 10$ time units. It also shows sample instruction lookahead times for *SC 2000 HPC Games benchmark distribution.*

4.2.3.1 Prediction for Sample Recursive Program

In order to apply the backward propagation algorithm, parent nodes from the child graph are computed first and a priority queue is constructed for all states with their initial values of time and energy. Figure 4.10 shows the sample recursive program with node numbers and table 4.2 shows the parent states for all individual states.

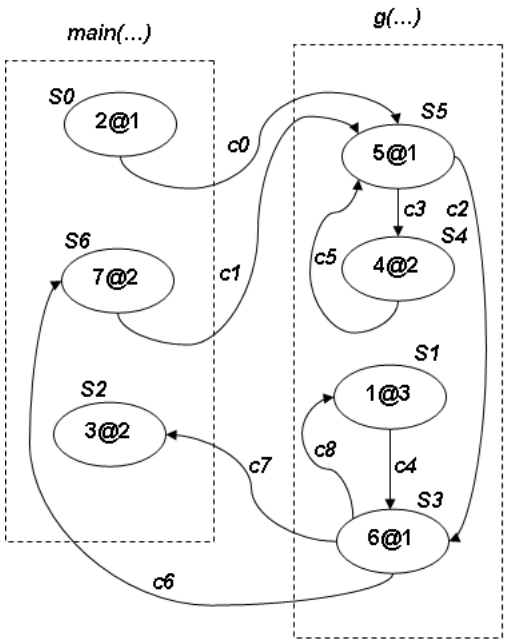

Figure 4.10: Sample Recursive Program with State Numbers

Table 4.2: Parent-Child States

State	Parent->Child
S0	-
S1	S3->S1
S2	S3->S2
S3	S1->S3; S5->S3
S4	S5->S4
S5	S0->S5; S4->S5; S6->S5
S6	S3->S6

Priority queue is constructed for all states with the initial time and energy values and the length of the priority queue is set to a limited default value (14). Then, individual states are randomly selected and analyzed as long as there is no empty priority queue. Let us say that, state $S1$ is selected. Based on the initial values in the priority queue, it predicts 3 units of energy $(power * time = 3 * 1)$ in 1 unit of time $(1,3)$. Since, these values are completely analyzed and did not meet the time requirements, this tuple $(1,3)$ gets removed from the queue. Parent state of $S1$ is $S3$ and has to be instrumented with the cumulative energy and time values. Cumulative energy and time values for $S3 \rightarrow S1$ is $(7,9)$, which translates to $9\,(6(S3)+3(S1))$ units of energy in $7\,(6(S3)+1(S1))$ units of time. As the parent state itself has not been analyzed yet, this new cumulative values gets inserted into the priority queue leaving the queue state as: $S3 : (6,6), (7,9)$.

Similarly, lets say the next state that gets to be analyzed is $S2$. Based on its initial value in the queue, state $S2$ predicts 6 units of energy in 3 units of time $(3,6)$. Since, the time is not met $(3 < 10\, units\, of\, time)$, this tuple gets removed from the queue. The cumulative tuples of energy and time is then inserted into its parent state $S3 : (6,6), (7,9), (9,12)$. State $S3$ is fetched next and its initial prediction is $(6,6)$. Then, the cumulative tuple values gets inserted into all parent states $(S1\, and\, S5)$ of $S3$. Since, $S1's$ priority queue is empty from its own analysis, the new cumulative tuple value $(7,9)$ of $(S1+S3)$ gets inserted into its first position in the queue. And, $S5's$ queue becomes $(5,5), (11,11)$ with $(11,11)$ being the newly inserted tuple. The following table 4.3 shows the first pass analysis of all states along with its priority queues. The tuples in the following tables are shown in (Time, Energy) format.

Table 4.3: First Pass Prediction and Priority Queue Construction

State	Initial Energy Prediction	Current State Removed Tuple	Parent Priority Queue Insertion
S1	(1,3)	(1,3)	S3: (6,6), (7,9)
S2	(3,6)	(3,6)	S3: (6,6), (7,9), (9,12)
S3	(6,6)	(6,6)	S1: (7,9) S5: (5,5), (11,11)
S5	(5,5)	(5,5)	S0: (2,2), (7,7) S4: (4,8), (9,13) S6: (7,14), (12,19)
S6	(7,14)	(7, 14)	S3: (7,9), (9,12), (13,20)
S0	(2,2)	(2,2)	
S4	(4, 8)	(4,8)	S5: (9,13), (11,11)

The above table was constructed with the assumption that all states are chosen randomly in the sequential order mentioned in the table. The following table 4.4 shows the final priority queues for all states after the first pass. State $S0$ is the initial state of the program and hence it has no parent states.

Table 4.4: State of the Priority Queues after First Pass

State	Priority Queue
S0	(7,7)
S1	(7,9)
S2	
S3	(7,9), (9,12), (13,20)
S4	(9,13)
S5	(9,13), (11,11)
S6	(12,19)

Since the priority queues are not empty, the lookahead computation continues for all states. The following table 4.5 shows the states, their corresponding prediction and the state of the priority queues after the first pass. Lookahead computation for states $S3$ through $S5$ in table 4.5 happens as regular priority queue insertions as explained above. For state $S6$, parent state priority queue insertions are not needed because, the state has reached its terminal computation time limit $(12 > 10\, units\, of\, time)$ and predicts 19 units of energy. Also, $(12,19)$ is not covered by $(16,27)$ because the energy is not maximum $(19 < 27)$. For state $S1$, the new cumulative tuple value $(13,15)$ is not inserted in its parent state $S3's$ priority queue because, it gets covered

by an already existing entry - $(13, 20)$ and thus, this path can be completely pruned from the search for $S1$. The state of the priority queues after the second pass analysis is shown in table 4.6.

Table 4.5: Second Pass Prediction and Priority Queue Construction

State	Current Priority Queue	Prediction	Current State Removed Tuple	Parent Priority Queue Insertion
S3	(7,9), (9,12), (13,20)	(7,9)	(7,9)	S1: (7,9), (8,12) S5: (9,13), (11,11), (12,14)
S4	(9,13)	(9,13)	(9,13)	S5: (9,13), (11,11), (12,14), (14,18)
S5	(9,13), (11,11), (12,14), (14,18)	(9,13)	(9,13)	S0: (7,7), (11,15) S4: (13,21) S6: (12,19), (16,27)
S6	(12,19), (16,27)	(12,19)	(12,19)	
S0	(7,7), (11,15)	(7,7)	(7,7)	
S1	(7,9), (8,12)	(7,9)	(7,9)	S3: (9,12), (13,20), ***(13,15)***

Table 4.6: State of Priority Queues after Second Pass

State	Priority Queue
S0	(11,15)
S1	(8,12)
S2	
S3	(9,12), (13,20)
S4	(13,21)
S5	(11,11), (12,14), (14,18)
S6	(16,27)

After the completion of second pass for all the states, let's say state $S5$ gets chosen randomly in the third pass. Head of $S5's$ priority queue $(11, 11)$ does not contribute to its lookahead prediction because, its current energy prediction (13) is greater than 11. Also, since its time prediction already reached the maximum allowed limit, there is no need for the parent state priority queue insertions. So, state $S5$ predicts $(12, 14)$ and the entries $(11, 11), (12, 14)$ are removed from the queue. Tables 4.7 and 4.8 show third pass analysis and the state of priority queues after the completion of third pass respectively. From the table 4.7, it can be seen that

entries $(14, 17)$ and $(14, 18)$ corresponding to states $S5$ and $S3$ when analyzing states $S3$ and $S1$ are already covered by existing entries and the respective paths are pruned from the search.

Table 4.7: Third Pass Prediction and Priority Queue Construction

State	Current Priority Queue	Prediction	Current State Removed Tuple	Parent Priority Queue Insertion
S5	(11,11), (12,14), (14,18)	(12,14)	(11,11), (12,14)	
S6	(16,27)	(16,27)	(16,27)	
S3	(9,12), (13,20)	(9,12)	(9,12)	S1: (8,12), (10,15) S5: (14,18), ***(14,17)***
S4	(13,21)	(13,21)	(13,21)	
S0	(11,15)	(11,15)	(11,15)	
S1	(8,12), (10,15)	(8,12)	(8,12)	S3: (13,20), ***(14,18)***

Table 4.8: State of Priority Queues after Third Pass

State	Priority Queue
S0	
S1	(10,15)
S2	
S3	(13,20)
S4	
S5	(14,18)
S6	

After the fourth pass for remaining states with non-empty priority queues, the final energy predictions for all the states is shown as in table 4.9. As $S2$ is a terminal state with no outgoing arcs, its prediction stay at $(3, 6)$.

Table 4.9: Final Energy Predictions

State	Energy prediction	(Time, Energy) tuples
S0	15	(11,15)
S1	15	(10,15)
S2	6	(3,6)
S3	20	(13,20)
S4	21	(13,21)
S5	18	(14,18)
S6	27	(16,27)

4.2.3.2 Prediction for Sample Benchmarks

The back-propagation analysis was applied to the benchmark programs listed in Table 4.10 to compute the maximum energy consumed within a lookahead window specified by certain time units. Instruction-level energy and time predictions for the sample benchmarks are derived from the variable-n-gram model explained in chapter 3. All but two of the benchmarks come from the *SC 2000 HPC Games benchmark distribution* assembled by James Kohl at Oak Ridge National Laboratory. The two exceptions are the `sweep3d.single` benchmark, which is a particle transport application designed to be representative of structured grid computations [27], and the `recurex` benchmark, which is the program represented by Figure 4.1. The state graphs of the benchmarks vary widely in complexity, as summarized in the table. The time to compute lookahead was measured using the "user time" reported by the Linux `time` command on a 700 MHz AMD Athlon.

The same back-propagation analysis can be applied to predict system events such as the usage of floating point units, usage of network card, system calls etc. Thus, the present day compilers can employ prediction mappings and use compile time analysis to predict directly observable static properties, such as instruction counts and occurrence of program features.

Table 4.10: Properties of Lookahead Benchmarks

Benchmark	Nodes	Max. Successors	Lines of Code	Lookahead time as a function of time units					
				1	10	100	1,000	10,000	100,000
1000s	71	8	752	0.00	0.00	0.00	0.04	0.41	4.66
c4	555	17	1,330	0.01	0.01	0.03	1.72	40.96	400.42
dhry21	173	62	1,464	0.00	0.00	0.01	0.18	4.42	53.59
fib	78	12	509	0.00	0.00	0.00	0.06	0.91	6.81
flops	88	22	1,154	0.00	0.00	0.00	0.01	0.29	2.38
hanoi	29	4	638	0.00	0.00	0.00	0.00	0.14	1.03
heapsort	67	8	755	0.00	0.00	0.00	0.01	0.25	2.3
livermore	1,654	264	8,051	0.2	0.2	0.27	10.27	200.44	1199.18
lud	90	17	805	0.00	0.00	0.00	0.01	0.17	2.25
mdbnch	425	105	2,152	0.01	0.01	0.01	3.47	91.36	1028.1
nbench	1,507	55	8,752	0.11	0.12	0.13	8.35	189.08	719.57
nsieve	82	14	884	0.00	0.00	0.00	0.03	0.44	4.42
queens	91	7	850	0.00	0.00	0.01	0.23	5.99	69.86
recurex	7	3	—	0.00	0.00	0.00	0.00	0.02	1.11
shuffle	56	7	712	0.00	0.00	0.00	0.00	0.02	0.23
sim	514	20	1,574	0.02	0.02	0.02	1.65	20.25	228.81
streadm_d	106	20	242	0.00	0.00	0.01	0.06	0.6	6.06
sweep3d.single	671	56	2,085	0.00	0.02	0.03	0.82	34.41	281.63
vector_me	223	24	13,730	0.01	0.01	0.01	0.03	0.47	5.18
whetstones	132	10	284	0.00	0.00	0.00	0.16	0.97	3.26

CHAPTER 5: RUN-TIME PREDICTION SUPPORT USING COMPRESSIVE HASHING

Runtime power prediction mechanisms often rely on dynamic monitoring or program trace and execution history information to provide reasonable predictions. Predictions using dynamic monitoring has the potential to incur additional overhead and thus has more possibility for mispredictions. Also, runtime prediction mechanisms can only provide estimates and not bounds on the system behavior; compile time analysis on the other hand, can provide a safer bound on the values of interest. Thus, prediction of system's thermal properties such as energy consumption can benefit more from static rather than runtime prediction mechanisms. For example, when a system exceeds its allowed thermal limits, it is almost impossible to recover the system from damage when only a runtime prediction mechanism is used. On the other hand, if a system employs static power prediction mechanisms along with runtime control, the system has the ability to know about the thermal behavior well in advance, allowing it to prevent the damage by scheduling the work between between multiple processors. Thus, static predictions can warn the system to take preventive measures without actually overheating the system or causing considerable damage.

After having extracted the static predictions into a data structure through chapters 3 and 4, the runtime prediction mechanism can now access that data structure when the operating system queries for a prediction. Since, random queries from the operating system can be dynamic and irregular, accessing the static data structure can incur additional overhead if the data is not readily available (i.e - main memory has to be accessed for the prediction). This chapter concentrates on using Compressive Hashing as explained in chapter 2 to access the power prediction data structure.

Section 5.1 presents some background research work and Section 5.2 discusses different ways of posting static predictions to runtime control. Section 5.3 explains how Compressive Hashing technique applies to the problem of providing run-time support to access static predictions and section 5.3.3 presents simulation results.

5.1 Related Work

Performance monitoring counters have been used by various researchers to predict energy consumption dynamically at runtime. Most of them rely on history information and assume that the programs exhibit repetitive behavior to base their future predictions. A runtime prediction model [31] uses performance monitoring counters to dynamically monitor information such

as memory accesses and CPI by sampling the workload at every 100 Million instructions, and uses this prediction to dynamically manage power for the next execution interval.

A runtime phase predictor [30] uses continuous monitoring of performance counters on a Pentium-M processor for on-the -fly dynamic power management using DVFS [Dynamic Voltage and Frequency Scaling]. It classifies the application programs into different phases (cpu-bound, memory-bound etc.,) and uses history information to construct a GPHT (Global Phase History Table) to predict power consumption.

For embedded systems, a runtime feedback based energy estimation model was developed [24] that uses feedback from a battery monitor along with performance counter data to calculate its predictions. Contrras et al., [12] base their cpu and memory power prediction by assigning power weights for performance events obtained from monitoring the hardware performance counters. Because of continuous monitoring, all the runtime prediction techniques explained above has the potential possibility of increasing the system overhead. Alternatively, we propose using "Compressive Hashing (CH)" to help predict system properties at runtime with minimal overhead.

The idea here is to construct a static data structure from statically determined instruction-level predictions, create a compressed surrogate representation of the data structure such that the majority of it occupies higher levels of memory, and make the runtime system access this compressed data structure on-demand. As the runtime system is accessing the prediction values on demand, the memory footprints for accessing the prediction data structure can be highly irregular which can degrade the performance of the runtime system. By using CH, we have a lightweight mechanism that can make reasonably accurate predictions at runtime with less overhead.

5.2 Minimizing Runtime Post Cost

High performance computing (HPC) systems often need the bounds on performance metrics to effectively control the system and sustain the performance. Dynamic predictions at runtime can only provide estimates of performance metrics, along with the possibility of considerably increasing the overhead associated with the runtime mechanism. Only, static predictions has the ability to provide bounds on the performance metrics needed by a HPC system.

This section discusses possible ways to post static predictions to the runtime system at a reduced cost. Static predictions obtained through lookahead analysis described in section 4.2 can be posted at each entry point for corresponding states in the state flow graph, but that might cause considerable execution overhead. The predictions can be stored in memory locations

that can be seen by the runtime control and can be accessed by instructions in the program, but this memory operation itself could be affecting energy/time predictions of the surrounding instructions. Alternatively, a minimum number of posting points can be preselected where the predictions can be posted. All these attempts can be wasteful, if the runtime application does not require predictions at that time of execution.

This dissertation concentrates on using a support mechanism for providing on demand prediction rather than active posting to the runtime system, (i.e.) instead of posting prediction values in the code actively at predetermined points in the program, all predictions are encoded into a static data structure which can then be accessed on demand by the runtime system. The static data structure can be created by mapping the application's program counter (PC) value to its predictions. The runtime control can then use the PC value to index the data structure and access its corresponding predictions. Since more than one PC value can have same predictions, the size of the data structure can be considerably reduced by making use of this redundancies. Thus, the idea here is to find a hash function based on the PC value such that, the hashed location will have the same prediction values.

5.3 Compressive Hashing as a Runtime Support Mechanism

To utilize the advantages offered by the static prediction mechanisms at runtime, this section discusses how compressive hashing described in section 2.1 can be used as a support mechanism to access the static predictions at runtime. On-demand access of static predictions can be more useful for a runtime system than posting predictions at pre-determined points in the program. Accessing predictions on-demand can lead to unpredictable memory access patterns. Having validated the concept of compressive hashing as a mechanism to improve the performance of applications that have irregular memory access patterns in chapter 2, this section explains how the compressive hashing scheme can be used to access static performance predictions at runtime at a reduced cost.

The predictions obtained through compiler state machine prediction lookahead analysis as described in section 4.2, can either be posted at every entry to each basic block or can be made accessible by the runtime system on demand. Clearly, the former approach of posting the predictions for each basic block in the program can constitute significant overhead because, if the posting operation was implemented as an instruction visible to the runtime system, the instruction itself will cause the energy predictions to be inaccurate and accessing the prediction variables for every block will increase the overhead. Most often, energy predictions or other performance metric predictions are not needed for all basic blocks. So to reduce the overhead, it is reasonable to make the runtime system access the predictions only on demand - (i.e.)

when the runtime system needs the metrics. On demand runtime predictions is often the case when a system needs to query the prediction values based on system's load factor, user event, scheduling policies, voltage regulation etc,.

As the predictions are statically available, rather than posting the values at each basic block state, the predictions can be encoded into a static data structure. One possible way to differentiate instructions in a program is through their program counter (PC) values. The static data structure comprising of energy/time values can then be indexed through the PC values for the runtime predictive system to access. A runtime system randomly querying for a prediction can lead to irregular accesses to the data structure. Also, since more than one PC value can have the same predictions, the redundancies in the data structure can get the benefits of lookup table compression through compressive hashing. So the idea here is to find a hash function based on PC values, that when hashed into a location of the compressed table will yield the same prediction values. The hash function can then be compiled into the process as a signal handler and whenever a new prediction is needed, the runtime controller signals and the handler responds by hashing the saved PC value utilizing the hash function and posts the respective predictions.

The GP system as explained in section 2.2 was the base programming technique used to evolve hash functions for accessing static predictions. As can be seen from section 2.3, the sparseness of the original table is an important factor in determining how fast the GP can evolve hash functions and how well the hash function could compress the table resulting in improved access performance. Even though the static predictions table is not as sparse as one would expect for the CH mechanism to work well, the main idea behind using CH to access the static predictions is to utilize the redundant energy/time entries in the table to reduce the overhead as much as possible than direct access.

5.3.1 Construction of Static Data Structure

The instruction level energy and time values obtained through the N-Gram models as described in section 3.6 has to be encoded into a static data structure in order for the CH mechanism to be applied. The steps involved in constructing a data structure out of the instruction-level values for a program under test are given as follows:

1. Obtain the program's relevant assembled or disassembled instructions along with the PC values

2. Calculate the control flow and target address information

3. Classify the instructions in the program into known types and associate them with corresponding energy/time values

4. Form basic block states and calculate their corresponding energy/time values

5. With the above information, form an associated data structure with PC value as key and energy/time predictions as the value

Assembly language instructions of a source program can be obtained either through GCC's *'-s'* option or through disassembling an executable. When a program is split between different source files and dynamically linked with several different object files, it becomes a cumbersome process to gather assembly language instructions from different sources manually. So, we chose the approach of disassembling the executable file using *objdump,* which also provides the corresponding PC values of all instructions. In calculating the control flow, longjmp and indirect jumps are treated as expensive nops since it is difficult for a static analysis tool to know this information. The instructions are then classified into different types as determined in section 3.5 and are associated with their predicted energy/time values. The basic blocks are formed as described in section 4.1 and the energy/time values for each basic block state are then calculated. Since the energy/time prediction values are calculated at the granularity of basic block states to form the static data structure, all the PC values in a particular basic block state will have the same calculated energy/time value. Several *awk* and *sed* scripts were used in the whole process to construct the associated data structure.

5.3.2 Lossy Compressive Hashing (LCH)

Depending on the sparseness of the underlying table, often an extensive lossy compressive hashing (LCH) mechanism has to be used when compressing a huge data structure. Lossless CH [46] typically yields one small lookup table for all of the original table entries, whereas Lossy compressive hashing (LCH) yields more than one table lookup and the correctness of the lossy nature of the compressed table is preserved using a checker table identifier as described in sections 2.1 and 2.2. The second level lookup can simply be from the original table or can point to several other small n-level lookup tables. The checker table determines which of the small tables provide the correct lookup. The number of small lookup tables obtained using the LCH mechanism fall into one of the following two categories:

- Large number of original table entries can hash into the first small table, and the rest can hash completely into one or more small table(s)

- All of original table entries can be scattered and hashed into more than one table

The above categories in turn depend on the following factors:

- Sparseness of the original table
- Size allowance for the small table(s)
- Selection of hash function parameters for GP
- Availability of time and resource to run the GP system

The GP technique for obtaining smaller tables of energy/time values is similar to the GP system explained in section 2.2. Even though the original energy/time prediction table has redundant entries, it is not sparse enough to be directly compressed using lossless CH; lossy CH mechanism is often the case. The following algorithm 5.1 describes the sequence of steps involved in constructing small lookup tables using CH for runtime energy/time prediction.

Algorithm 5.1 Construction of Small Lookup Tables for Runtime Energy/Time Prediction

---

1. Construct the static data structure as explained in section 5.3.1 for the program under test.
2. Use a GP technique similar to the one described in section 2.2 to evolve lossy/lossless compressive hash functions. Modify the fitness function according to lossless or lossy CH.
3. Using the evolved hash function, form a new $n+1^{\text{th}}$ level data structure for the original table entries that did not perfectly hash into the n^{th} level compressed table, where $n > 1$ and represents the table number. This data structure forms the primary data structure for next iteration. Repeat steps 2 and 3 with the new data structure until all or most of the entries are hashed.
4. After compressing the original table completely using lossy/lossless CH mechanism, construct the small tables using the different evolved hash functions.
5. When all the original table entries are hashed completely into small tables, form an associated data structure for the checker table identifier (if lossy CH was used initially) with PC as the index and table number as the value. Similar to the original table, lossy/lossless CH can again be used to compress this checker table and to form checker hash tables.

5.3.3 Performance Evaluation

This section explains how CH mechanism was evaluated as a runtime prediction access mechanism. Two benchmark test cases (*DCRAW* and *Gdb*) were chosen and static data structures were constructed as explained in section 5.3.1, and GP as explained in section 2.2 was used to evolve lossy compressive hash functions. In both the cases, the checker table was also compressed using CH mechanism. To reduce additional levels of table identifiers, the checker tables were compressed in such a way that the invalid locations are marked with a special marker and the entries hashing to this location are redirected to the next level(s) following different hash function(s). Small lookup tables representing the original and the checker table were constructed as explained in algorithm 5.1.

5.3.3.1 Case 1: Dcraw

Dcraw is a open source software used to convert raw image formats obtained from most commercial digital cameras into standard image formats (ppm, jpg etc.). Static data structures for energy and time predictions are obtained by following the procedure in section 5.3.1. For *Dcraw's* original data structure ($1.5MBytes$), the GP technique yielded a final compressed data structure of $0.13MBytes$ for energy prediction and $0.09MBytes$ for time prediction tables. For energy and time prediction, the GP evolved final compressed data structure consists of four hash tables and four checker tables. The checker table entries are 2-bits wide to determine which of the four tables has the final value and the four hash table entries are 32-bits wide containing the energy values corresponding to the PC values. GP was run over night to yield respective hash functions for the hash tables and the checker tables. The following tables 5.1 and 5.2 show the resultant hash functions and the sizes of the hash and the checker tables for energy and time predictions respectively, where x refers to the PC value, and $INCR(x) = x+1$, $DECR(x) = x-1$, and $GRAY(x) = xxor(x >> 1)$.

Table 5.1: GP Evolved Hash Functions for DCRAW's Energy Prediction

	Hash Functions	Size (Bytes)
Checker Table 1	(((((((x+1) >> 0x10) + x)+1) << 0x6) >> 0x7) - (~(~(((x >> 0xe)))) - (x >> 0x7)))	25K
Checker Table 2	(((((x >> 0x10)) + x) << 0xf) >> 0x8)	2.5K
Checker Table 3	(((-((((x << 0x8)-1) + ((x - INCR(-((((((x+1) >> 0x8))) - INCR(-(((x-1) >> 0x8)))))))+1))) - INCR(-(((((x << 0xd))) - INCR(-(((x))))))))+1)+1)	2.5K
Checker Table 4	DECR((((((x + x) + (((x >> 0x4)+2))) + ((((((x+1) >> 0x4)+1) + x)+2))) + x) + ((((((x) >> 0x4) + x) + x) << 0xc) >> 0x4))	2.5K
Hash Table 1	((~((x >> 0x4)) + (~((((((~((((((((((x >> 0x4)-1) + x) >> 0x4) + (x >> 0x4)) >> 0x4)))))) + x) + x) >> 0x4)))) + x)) >> 0x4)	40K
Hash Table 2	(~(((((x+1) - ~(~(((x - ~((((x-1) - ~(x)) - ~(DECR(DECR(~((DECR(DECR(~(((((~(x)-2)) - ((((x+1) - ~((((x - ~((~(x) - ~(DECR(DECR(~((((~(x)-2)) >> 0x4)))))))) >> 0x4) - ~((x - ~(x)))))) >> 0x4) - ~(x))) >> 0x4)))) >> 0x4)))))))) >> 0x4)))) >> 0x4)-1))+1)	40K
Hash Table 3	((((x >> 0x3) + (GRAY(((~(x) | x) & x)) >> 0x9)) - (x >> 0xe))-1)	20K
Hash Table 4	-(((-(((x ^ (x>>1)) >> 0x6)) << 0x10) ^ ((((x-1) >> 0x6) ^ x) >> 0x6)))	0.5K

Table 5.2: GP Evolved Hash Functions for DCRAW's Time Prediction

	Hash Functions	Size (Bytes)
Checker Table 1	(~((DECR(DECR((x + ~((DECR(DECR(DECR(((DECR((x + ~((DECR((x + ((DECR((x + ~((((((((((x + ~((((x + (x + x))-1) >> 0x4)))-1) + x) + (x + ~(((x-1) >> 0x4))))))-1) >> 0x4)-2))) >> 0x4) + x))) >> 0x4)))) + x) + (x + ~(((x-1) >> 0x4)))))))) >> 0x4))))) >> 0x4))	2.5K
Checker Table 2	(-((~((((~(((-(INCR((((~(-(((((((~(-(INCR((((~(-(x+1)) + (((~(-((((((~(-(x+1)) + x + 1)))))) >> 0x3) >> 0x4) + x))))))) >> 0x3) >> 0x4) + x) >> 0x3) ^ (~((0xc ^ (0xc >> 1)))+1)))) + x)))))) >> 0x3) >> 0x4) + x)) >> 0x3))	2.5K
Checker Table 3	((~(((x ^ INCR(~(GRAY(GRAY(GRAY(((((((((x | x-4) >> 0x3) | x) >> 0x3) ^ (((((x | x-4) >> 0x3) | x) >> 0x3) >> 1)+2))))))))) >> 0x3))+1)	0.25K
Checker Table 4	(((((0x3 ^ (((-(((((-((x ^ x>>1))+1) >> 0x3)+1) ^ ((((-((x ^ x>>1))+1) >> 0x3)+1)>>1)))+3))))) >> 0x3)+2)	0.03K
Hash Table 1	((((((~((((-(((~((((-((((((x >> 0x3) - x) >> 0x4))+1))) - x-1) + x)))))+1) - x) + (x >> 0x3)-1)) - ((x >> 0x3) - x)) - -(x)) >> 0x4) - x) >> 0x4)	40K
Hash Table 2	((DECR(~(DECR(~(((~(0x6) +((-(x-1)))) +(DECR((DECR(~(DECR(~(DECR(~((((-(x-1)) +(DECR((INCR(-((DECR(~(DECR(~((((-(x-1)) - (x+1)) - DECR(~((((-(x-1)) +(DECR((INCR(-((DECR(~(DECR(~((((-((((x-1) >> 0x4))-1)-1) - INCR(~((-(x-1))))) +((-(x-1)))))))) >> 0x4))) - x)))) +((-(x-1)))))))))) >> 0x4))) - x)))) +((-(x-1)))))))))) >> 0x4)))))))) >> 0x4)	40K
Hash Table 3	((GRAY((DECR(GRAY((GRAY(x) - (~(x) >> 0x4)))) >> 0x10)) - ~(((~(x) >> 0xf) - DECR(GRAY((GRAY((~(x) >> 0xf)) - (~(x) >> 0x4)))))))	4K
Hash Table 4	(((GRAY((((x ^ x>>1) - x) ^ x)) - x) * DECR(((x + x) << 0x7)))	2K

5.3.3.2 Case 2: Gdb

Gdb's energy prediction table ($13.5MBytes$) was not sparse enough for GP to hash all entries in the hash tables. To avoid the increase in number of checker table levels. the remaining unhashed checker table entries along with their PC values are left in their original format, to provide a total compressed size of $1.5MBytes$. The time prediction table of *Gdb* was compressed fully using lossy CH yielding six checker and hash tables for a total size of $1MBytes$. Tables 5.4 through 5.7 show the checker and compressed hash functions for energy and time predictions respectively.

Table 5.4: GP Evolved Checker Hash Functions for GDB's Energy Prediction

	Hash Functions	Size (Bytes)
Checker Table 1	(~((DECR((x - ((x+1) >> 0x4))) >> 0x4)) - ((((x+1) >> 0x4) >> 0x9) >> 0x4))	37K
Checker Table 2	((((x-1) >> 0x3) + INCR(((x+2)))) >> 0x3)	37K
Checker Table 3	(-((((x) ^ (~(((GRAY(GRAY(x)) ^ x)) ^ (-(-(((~(x) ^ x))) << 0xf))) ^ x)) << 0xf)	37K
Checker Table 4	-((((((((((-((x >> 0x3)) - x) >> 0x3) << 0xe) >> 0x3) >> 0x3) >> 0x4) >> 0x3) >> 0xd) & (((-((x >> 0x3)) - x) >> 0x3) << 0xe)) >> 0x3) >> 0x4))	37K
Checker Table 5	(INCR(((-(DECR((DECR((DECR((INCR((x + x)) >> 0x3)) + x)) >> 0x3))) + x) + x)) >> 0x3)	37K
Checker Table 6	(INCR(INCR((~(x) + INCR(INCR(-(((INCR(-(((x+1) + (x+1)))) >> 0x3) + x))))))) >> 0x3)	37K
Checker Table 7	((DECR(((-((GRAY(DECR(DECR(DECR(((GRAY((GRAY((x-1)) >> 0x9)) >> 0x9) << 0x7))))) >> 0x9)) & (DECR(DECR(((GRAY((DECR((x-1)) << 0x7)) >> 0x9) << 0x7))) << 0x7)) >> 0x9)) << 0x7) >> 0x9)	37K
Checker Table 8	~(INCR(INCR((INCR(INCR(-((-(-(INCR(~(~((~(DECR((((x+1) + (~(DECR((x + x))) >> 0x3)) + x))) >> 0x3)))))) << 0xf)))) ^ (DECR(-(x)) ^ x)))))	37K
Checker Table 9	INCR((INCR((~((x-1)) ^ (-(x) << 0xa))) >> 0xc))	37K
Checker Table 10	-(INCR(GRAY(GRAY(INCR(GRAY(DECR(DECR(GRAY(-(GRAY(INCR(GRAY(INCR(GRAY(GRAY((GRAY((x - (GRAY(x) ^ x))) ^ GRAY(GRAY(GRAY(GRAY((x - (GRAY(x) ^ x)))))))))))))))))))))))))	37K
Checker Table 11	DECR(-(-(GRAY((GRAY(~(GRAY(~(GRAY((~(GRAY(~((GRAY(GRAY(x)) ^ x)))) ^ (GRAY((x ^ x)) ^ x))))))) ^ GRAY(~(GRAY(x)))))))	37K
Checker Table 12	~((INCR(((-(x) << 0xc) + DECR(-(INCR(((INCR(DECR(((DECR(-(INCR(((INCR(DECR(((-(x) << 0xc) + (x-1)))) << 0xe) + x)))) + (x-1)) + (x-1)))) << 0xe) + x)))))) << 0xb))	37K
Checker Table 13	~((((((GRAY(((-((~(~((x & x))) - (-(0xb) << 0x6))) & x) ^ ((x+1) - x))) - (~(~(DECR(INCR(((0xb - x) >> 0xe))))) & x)) >> 0xe) - x) << 0x3))	37K

Table 5.5: GP Evolved Hash Functions for GDB's Energy Prediction

	Hash Functions	Size (Bytes)
Hash Table 1	((((x >> 0x6)) + (((x >> 0x6)) + (~((x >> 0x3)) + x))) >> 0x4)	391K
Hash Table 2	((INCR(-(((x >> 0x3) + (x)))) + ((x >> 0x6) + (((x >> 0x3) + x) >> 0x6))) >> 0x6)	118K
Hash Table 3	(((~(x) - ((-(x) - (x + x)) >> 0x5)) - (-(x) - x)) >> 0x5)	118K
Hash Table 4	(((x >> 0x4) << 0xc) - (x >> 0x6))	78K
Hash Table 5	GRAY((-((((x >> 0xe)-1) + x)) >> 0x4))	78K
Hash Table 6	(DECR((~(((((x >> 0x4) << 0x11)) >> 0x6)) - x)) >> 0xc)	4K
Hash Table 7	INCR((INCR(GRAY(x)) >> 0x6))	4K
Hash Table 8	(INCR(-(~((GRAY(x) - x)))) | x)	0.5K

Table 5.6: GP Evolved Checker Hash Functions for GDB's Time Prediction

	Hash Functions	Size (Bytes)
Checker Table 1	(INCR(-((-((-((DECR(x) - (~(((-(x) >> 0x4) - DECR(x))) >> 0x4))) - ((-((DECR(x) - (~((-((-((DECR(x) - (~(((-(x) >> 0x4) - DECR(x))) >> 0x4))) >> 0x4)) - DECR(x))) >> 0x4))) >> 0x4) >> 0x4))) >> 0x4)))	37.5K
Checker Table 2	(~((-((-(((x >> 0x4) >> 0x4)) + (x >> 0x4))) + -(((-(((x >> 0x4) >> 0x4)) >> 0x4) >> 0x4))))	37.5K
Checker Table 3	((GRAY((((-(INCR(x)) - x) ^ x) >> 0x3)) ^ (x >> 0x3))	11.25K
Checker Table 4	((((((x << 0x3) >> 0x9) << 0x3) >> 0x9) ^ ((x >> 0x6) ^ GRAY(((((~((x >> 0x5)) - x) >> 0x5) << 0x3) >> 0x9))))	0.375K
Checker Table 5	(INCR((-(~(INCR(-(GRAY(((INCR(x) ^ x) >> 0x9)))))) - (GRAY((INCR(INCR((((INCR(x) ^ ((INCR(x) ^ (((INCR(x) ^ ((INCR(x) ^ x) >> 0x9)) >> 0x4) - ((INCR(x) ^ (x >> 0x4)) >> 0x9))) >> 0x9)) >> 0x4) - x))) - x)) >> 0x6)))	0.375K
Checker Table 6	((DECR(INCR(DECR((x + ((DECR((DECR(x) >> 0x3)) ^ (((x >> 0x9) + INCR(INCR((x >> 0xf)))) + x)) >> 0xd))))) ^ (x ^ x))	0.375K

Table 5.7: GP Evolved Hash Functions for GDB's Time Prediction

	Hash Functions	Size (Bytes)
Hash Table 1	(-((-(((-((-(((-((DECR(((-(((-(((-((x >> 0x4)) - -((-((x >> 0x4)) - x)))) >> 0x4) - x)) >> 0x4) - x)) - x)) >> 0x4)) - x)) >> 0x4) - x)) >> 0x4))	400K
Hash Table 2	((-(((DECR(x) + (-((((~(~(x)) - x) + x) - DECR(-(((-(((DECR((-(x) >> 0x4)) + -(-(x))) >> 0x4)) + -(-((DECR(x) + DECR((((GRAY(((DECR(INCR(x)) + -(-(x))) >> 0x4)) >> 0x8) + x) - DECR((~(~(x)) - x)))))))) >> 0x4))))) >> 0x4)) + DECR((DECR(-(INCR(x))) + DECR(-(-(x))))))) >> 0x4)	400K
Hash Table 3	(((x ^ ((x | (~(((x ^ ((x | GRAY((((((x | GRAY(x)) >> 0x4) | ((DECR((((x & x) << 0xd) >> 0x7)) & x) | (~(((x ^ ((x | GRAY((x | ((x & x) << 0xd)))) >> 0x4)) >> 0x4)) ^ ((x ^ (((((~((x << 0xc)) - x) + x) + x) - x) >> 0x10)) >> 0x4)))) >> 0x4) ^ x))) >> 0x4)) >> 0x4)) ^ x)) >> 0x4)) >> 0x4)	120K
Hash Table 4	(INCR (INCR (INCR (INCR (INCR (GRAY (INCR (INCR (INCR (INCR (INCR (GRAY (INCR (INCR (INCR (INCR (GRAY (INCR (((((-x)+2)) >> 0x3))))))))))))))))))	16K
Hash Table 5	(((((((~(-(INCR(~((GRAY((x >> 0x9)) ^ x))))) ^ GRAY((x >> 0x9))) >> 0x6) ^ GRAY((x >> 0x9))) >> 0xd) ^ -(DECR(~(~(((x >> 0x9) ^ (-(~(x)) >> 0x3)))))))	4K
Hash Table 6	(~(~((INCR(GRAY(x)) ^ ~((((x + x) >> 0x9) + x)))))	2K

5.3.3.3 Performance Comparison

Performance of CH in the above cases were evaluated by comparing the performance of accessing the original table vs. accessing the compressed tables. The performance results were obtained by randomly querying for a PC value to get the energy/time prediction. Binary search was used to access the original table. The compressed table results were obtained by first accessing the checker tables to identify the correct hash table, and then the value was retrieved from that hash table using the corresponding GP determined hash function as index. A series of if-else conditions were used to traverse the checker table levels. Clock cycle count for accessing the compressed tables includes cycle count for control flow, hash function calculation and the actual table access. The following tables 5.8 and 5.9 and figures 5.1 and 5.2 show the performance comparison results.

Table 5.8: Size and Performance Comparison for Energy Prediction

Test Cases	Size (Original in MB)	Size (CH in MB)	Clock Cycles (Original)	Clock Cycles (CH)
Dcraw	1.5	0.22	2123	876
Gdb	13.5	2.12	5924	2052

Table 5.9: Size and Performance Comparison for Time Prediction

Test Cases	Size (Original in MB)	Size (CH in MB)	Clock Cycles (Original)	Clock Cycles (CH)
Dcraw	1.5	0.15	1962	411
Gdb	13.5	1.15	6008	649

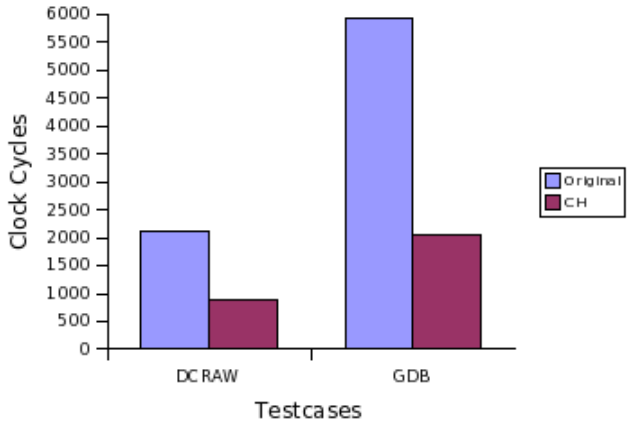

Figure 5.1: Performance Comparison for accessing Energy Prediction Table

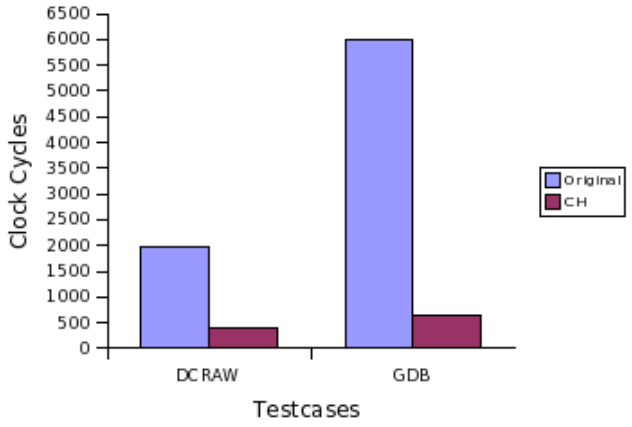

Figure 5.2: Performance Comparison for accessing Time Prediction Table

It can be seen from the above figures that the static prediction data structures can indeed be accessed at runtime with minimal overhead when a CH mechanism is applied. The two test cases considered above has shown $3x$ *to* $9x$ improvements in accessing the static predictions when compared to accessing them from the original table. The increased clock cycle count in accessing energy predictions when compared with accessing time predictions may be due to the following reasons:

- Size of the individual hash and checker tables for energy prediction table was not a power of 2 and this required some modulus operations to calculate the hash table index
- Number of GP determined hash and checker tables for energy prediction were more when compared to time prediction tables, and thus causing more levels of indirection

Even with the control flow, accessing the compressed table(s) yielded considerable performance improvements over accessing the original table. This shows that the compiler static prediction mappings along with CH mechanism can be successfully used by the runtime system to derive system level predictions.

CHAPTER 6: N-GRAM ANALYSIS

The Variable-N-Gram model developed in chapter 3 for calculating instruction-level predictions showed that, a variable and deeper context may not necessarily be needed for energy/time prediction. In an effort to determine how well the same variable-N-gram analysis tool developed in section 3.6.1 worked in the field other than energy/time prediction, this chapter applies the developed concept to the area of linguistic N-gram analysis and discusses how a genetic programming approach can be used for the same. Section 6.1 discusses research work done on Variable-N-Gram analysis in various fields including linguistics, bio-informatics, speech processing and software engineering. A genetic programing system similar to the one described in section 3.6.1 was developed to determine fixed and variable N-Gram predictions. GP evolved ngrams and the predictions/mispredictions obtained for a linguistic data set is shown in section 6.3.

6.1 Related Work

N-Gram analysis is most commonly used in the field of linguistics than any other research areas. Cohen et al [11] present a greedy algorithm for finding best variable-length predictive rules for predictions in Variable-N-Gram models that will result in less prediction errors. Their approach performed equally better as fixed-N-Gram predictions but only with less space requirements. A model in which an N-gram is not counted when it is a part of a larger reduced N-gram is called a reduced N-gram model. In addition to having reduced storage space, language modeling using reduced ngrams [25] also has the advantage of being semantically complete than traditional models. When applied to a large English and Chinese corpora, the reduced N-Gram model showed improvement in perplexities when compared to traditional N-Gram model sizes.

Apart from the field of linguistics, research has also been done in the field of bio informatics, speech processing, software engineering etc. Yeon et al applied N-Gram analysis to predict the presence of specific enzymes in glycoproteins [63]. They used a Variable-N-Gram analysis approach to predict glycotransferases that synthesizes a glycoconjugate. A Variable-N-Gram approach was selected for this model, because the amount of glycotransferases depended only on a variable small portion of the glyco added.

A Variable-N-Gram generation scheme [42] generated N-Grams based on classifying and splitting words as parts of speech (POS) word groups. This model resulted in a lower perplexity

when compared to conventional bigrams or trigrams and when applied to speech recognition resulted in a better recognition rate than conventional bigrams. A language modeling approach using ngrams [22] utilized rich lexical information attached to the words such as parts-of-speech tags and syntactic/semantic feature tags to predict words in the given corpus. It showed reduction in perplexity when compared to traditional trigram models on preliminary tests performed on parts of WSJ corpus.

A variable-n-gram algorithm was applied to conversational speech to selectively skip words and include class-in-word contexts [55]. The algorithm with N-Gram size of 5 applied to a switchboard corpus performed better than a classic trigram model. Usually phrases (multi word sequence - treated as a single unit entity) are selected by hand, or learned in structured domains such as air-traffic systems. Siu et al., present an algorithm that learns context dependent phrases and implemented the phrase grammar in variable-n-gram framework [54]. Their experiments show that when variable-n-gram is used along with phrase grammar, the performance is better on recognizing conversational speech for various tests (language modeling, number of silences, number of words etc.).

A dynamic opcode N-Gram based technique [40] was used to determine software birthmarks. An N-gram set of executable instruction opcode sequence extracted from the key region of the program determined by the user forms a software birthmark. Experimental analysis show that the software birthmarks are robust to code compression and can be applied to various different fields such as finding viruses, trojans, copyright information, identifying piracy and so on.

Privileged processes are running programs that require system resources and are inaccessible to ordinary user (i.e.: sending/receiving mail). An N-Gram based anomaly intrusion detection system [26] at the level of privileged processes was proposed in which a sequence of system calls are chosen as N-Grams, and an abnormal behavior is detected by a deviation from the normal set of sequences. They used specific measures to detect abnormal behavior and these measures were successfully used to detect intrusions in UNIX programs like sendmail, lpr and ftpd. An N-Gram based indexing approach is used in an information retrieval (IR) system [45]. The main advantage of using N-Grams in an IR system is that, many language dependent requirements of word based IR systems can be avoided and thus this approach can be applied to IR in different languages. Their experiments used a N-Gram size of 5 which gave reasonable retrieval performance without requiring too much memory.

6.2 N-Gram Analysis using Genetic Programming

N-Gram analysis and prediction using evolutionary computing techniques have not been done in the past. Genetic algorithms and genetic programming have mainly been used in the field of science and engineering. This section introduces the novel concept of using a genetic programing (GP) approach for the purpose of linguistic N-Gram analysis.

Genome of an individual in the population is represented as a tree, with each node representing an abstract gram. A gram can be an individual character or a word. Trees can also have a special gram $*$ as their nodes, where $*$ represents an 'unknown' sequence of gram(s). However, such special gram applies only to variable-N-gram analysis. Figures 6.1 and 6.2 show one of many possible genome structures of a Fixed-N and a Variable-N members respectively, where node *Ca* represents a character $'a'$.

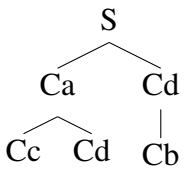

Figure 6.1: Genome of a Fixed-N-Gram Member

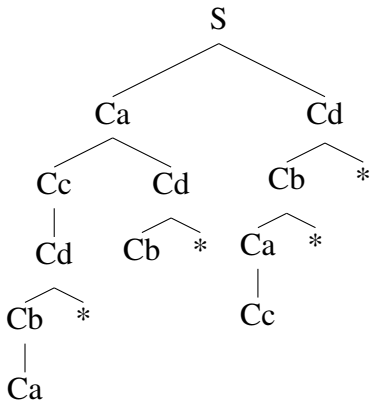

Figure 6.2: Genome of a Variable-N-Gram Member

A path in the tree represents a possible N-gram pattern that might appear in a linguistic document. The N-gram patterns are formed by traversing backwards from the leaf node towards top of the tree. The resultant 2Gram patterns out of the genome structure in figure 6.1 are $\{CcCa\}$,$\{CdCa\}$,and $\{CbCd\}$. Resultant patterns for the variable-N-gram genome in figure 6.2 are: $\{CaCbCdCcCa\}$, $\{C*CdCcCa\}$, $\{CbCdCa\}$,$\{C*CdCa\}$, $\{CcCaCbCd\}$, $\{C*CbCd\}$, and $\{C*Cd\}$.

A random population of members are created with random genome structures as a first step in the process. Member trees are created depending on the choice of N-Gram analysis chosen by the user. Fitness calculation for each member in the population is evaluated by counting the number of N-Gram pattern occurrences in the given linguistic corpus and the resultant mispredictions. For example, if a member has patterns $\{CcCa\}$ and $\{CdCa\}$, any other 2Gram patterns ending in Ca are counted towards mispredictions.

Following the traditional GP approach, selected evaluated members undergo either crossover or mutation and their fitness gets reevaluated. Members are selected for reevaluation by participating in a tournament selection. Two types of crossover are performed: either a member node gets replaced or a subtree gets replaced from another member. Mutation is done by replacing a certain randomly chosen node in the member's genome structure. This process continues for a chosen set of iterations and the final best member out of the population is determined. An STL-based tree implementation [49] was used for the basic tree structure implementation, node additions and replacements.

6.3 N-Gram Predictions and Mispredictions

Reuters-21578 Distribution 1.0 [39] data collection which consists of 22 data files was used for our GP based linguistic N-Gram analysis. The GP technique as explained in section 6.2 was used to train one data set (reut-2) from the collection and the resulting NGrams were applied to the remaining 21 data sets. To help the GP's convergence factor, all of the space and punctuation characters were removed from the data set. This helped in reducing the GP's overall runtime and reasonable results were produced in less than an hour for running million generations.

Most of the research work shown in the literature performed linguistic analysis with $N = 5$ as the upper bound, even for variable-N-gram analysis. The GP approach explained here has no such limits and can be used for any $'N'$. However, the size of the tree data structure and other parameters in the implementation has to be modified accordingly to accommodate for growing $'N'$ and for evolving good predictions. The variable-N-gram tree evolved from GP has the capability of growing for $N >= 100$, provided such patterns can be found in the given data set and that can meet the desired fitness metrics.

The fixed-N-Gram analysis for the Reuters-21578 data set was applied for N ranging from $2-5$ and the following tables 6.1 and 6.2 show some sample fixed and variable N-grams evolved by GP and their respective counts.

Table 6.1: GP evolved Fixed-Grams and respective counts

Size (N)	Grams	Count
2	ma	2363
2	on	6255
2	er	9982
2	es	12880
2	ti	6595
2	su	2392
2	vp	6
3	rob	128
3	tha	954
3	ahn	2
3	the	6318
4	comp	2663
4	bara	5
4	leaw	1
4	cial	389
4	scom	2108
5	ompan	2389

Table 6.2: GP evolved Variable-N-Grams and respective counts

Grams	Count
atiy	1
may	96
rso	215
to	6778
he	7546
pl	5006
ti	6595
ah	306
es	12880
an	11865

Following figures 6.3 and 6.4 show the fixed (2G - 5G) and variable N-Gram predictions and mispredictions respectively.

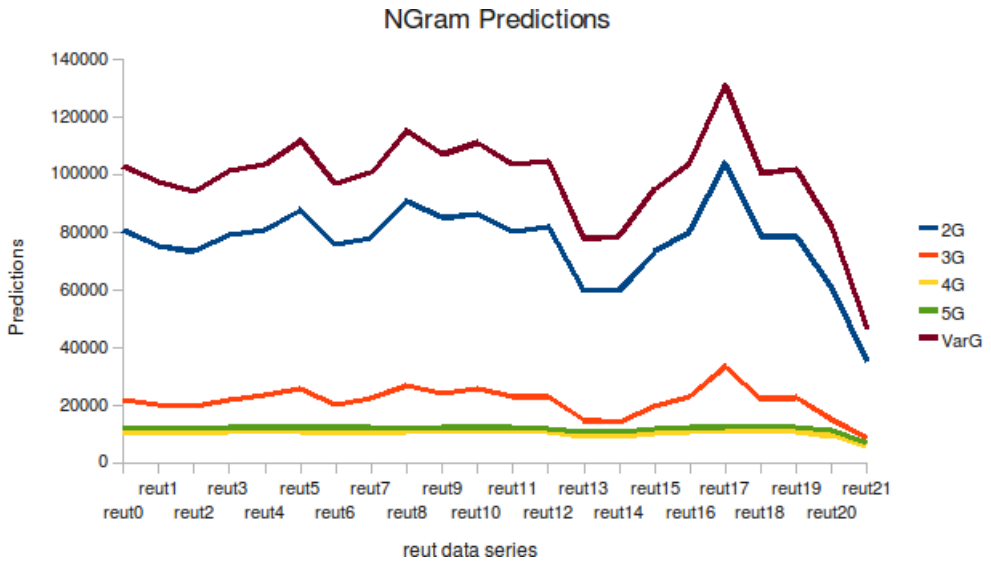

Figure 6.3: N-Gram Predictions for Reut Data Series

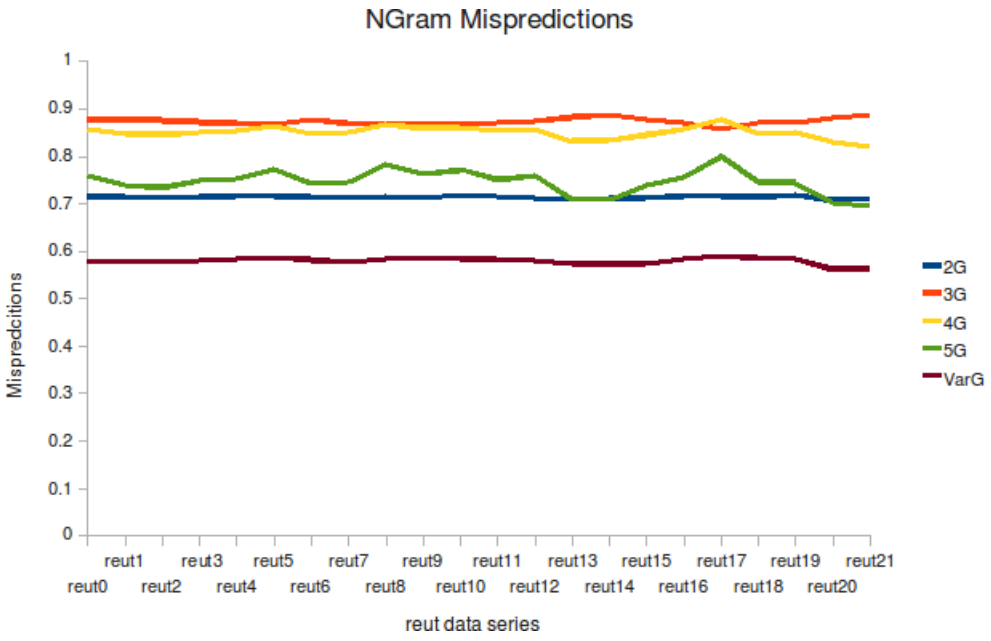

Figure 6.4: N-Gram Mispredictions for Reut Data Series

The figures clearly show that variable-N-gram performs better ($\sim 50\% mispredictions$) when compared to fixed-N-grams ($\sim 70\% - 90\% mispredictions$). It is interesting to note that, out of the fixed grams, the 2G model has less mispredictions than other fixed Ngrams. There are various factors that can be tuned in the GP system to improve the prediction/misprediction results. Currently, the GP system was run only for an hour to produce these results. This time limit can be relaxed and the GP can be run for several more hours and generations to

see whether better results are evolved. Size of the tree data structure also plays an important role in flexibility of the evolution. That parameter can also be modified for possibly better results. Also, for current results, one data set (reut-2) was chosen randomly to be the training set. There is no particular reason for this to be the case and the training set can also be varied for possible prediction improvement. The goal of this chapter was to determine the feasibility and performance of the earlier developed evolutionary approach in linguistic n-gram analysis and the results in figures 6.3 and 6.4 are a proof that shows that the developed evolutionary algorithm for variable-N-gram analysis can be applied for fields other than power prediction.

CHAPTER 7: CONCLUSION AND FUTURE RESEARCH DIRECTIONS

Ranging from GPUs to high performing parallel processors, there is a always a continual need for high performance and low power consumption. Modern processors employ deeper pipelines with superscalar architectures, prefetching, out-of-order execution etc. to do more work at any given single unit of time to achieve high performance. With millions of transistors on chip to perform the functionality needed to achieve high performance, there always comes the fact of increased power consumption. The same architectural advances including caches also distance main memory further, causing memory performance another major bottleneck.

7.1 Memory Performance

As the gap between processor function unit speed and memory speed continues to increase, it becomes appropriate to consider more aggressive methods for making data available when the processor needs them. Using smaller data structures can significantly improve performance by allowing the data to reside in higher levels of the memory hierarchy. This dissertation developed a concept called Compressive Hashing which uses lossy data compression technology to augment larger data structures and make irregular data accesses faster. Both the complexity of creating compressive hash functions and the effectiveness of using them were empirically evaluated with the goal of being able to predict how effective such an approach will be for various data structure and target architecture parameters. The performance analysis of applications considered in this dissertation show that, irregular memory accesses can really benefit from using CH.

7.2 Power Prediction

Prediction of system properties ranging from functional units usage to energy consumption, will help the system achieve its thermal and performance goals. Even for battery operated hand-held devices, energy consumption is a major issue. Static prediction models can accurately predict energy consumption for future time intervals, so that the system can take action before energy consumption changes. Making predictions based on low-level architectural features is impractical because, in addition to analysis often being slow, the architectural details needed often are not available to anyone except the device manufacturer. Energy hungry parallel systems and GPUs amplify the importance of being able to make accurate compile-time predictions based on empirical measurements.

This dissertation has introduced the novel concept of applying N-Gram analysis to the static determination of instruction-level properties. We have introduced instruction-level N-Gram models for predicting system properties like energy consumption and execution time. Unlike models described in the literature, which require either low-level details of the processor or additional measurements for inter-instruction effects, our model utilizes the idiomatic context present in the application program to derive instruction-level properties. Instead of using statistical solvers to minimize the error in the model, our analysis uses evolutionary algorithm search that is capable of exploring deeper execution contexts.

In addition to presenting efficient methods for *Fixed-N-Gram* analysis, an entirely new approach to handling *Variable-N-Gram* analysis was developed to deal with the huge contexts that were expected to be relevant in modeling modern pipelined, out-of-order, processors. Evolutionary computing techniques were developed to automate construction of all the cost models from empirical data, without needing information about architectural implementation details. The accuracy of these predictions is comparable to or better than most other models in the literature, despite the fact that our analysis statically tracked costs only for individual instructions with and without context.

As reverse-engineering of constraints from benchmarks generated to a known context-sensitive cost model shows, in cases where the cost data do precisely fit a model, the *Variable-N-Gram* analysis will uncover it. Where little improvement is possible, the *Variable-N-Gram* analysis also will reveal this fact – making it easy to use an $N = 1$ or other lower-order model. Tests predicting energy and time costs of instructions using a complex modern processor yielded modest improvement as more instruction classes were distinguished, and very little improvement as context was increased; however, *all* the cost measurements were well within 10%, which compares favorably with the accuracy reported by others modeling comparably dynamic computing systems. This may be close to the noise limit for static prediction of runtime costs in modern computers, and certainly is sufficient for many uses of static prediction of dynamic properties. The feasibility of the developed *Variable-N-Gram* analysis technique was also applied to linguistic N-Gram analysis and was shown to produce prediction results better than fixed N-Gram analysis.

This dissertation also showed that long-range instruction-level static prediction of runtime properties is feasible for significant programs. The combination of new algorithms with conversion of the program into a state machine yields static analysis times that are acceptably short and empirically are shown to grow roughly linearly with the lookahead depth of predictions. Looking hundreds of thousands of instructions ahead, from all points in a program, took from seconds to minutes. Thus, this algorithm most naturally can be used to predict directly observ-

able static properties, such as instruction counts and occurrence of programmed features (e.g., floating-point instructions or operating system calls).

Existing runtime prediction mechanisms are prone to introducing extra overhead and possible mispredictions when monitoring the system properties dynamically. This dissertation uses an alternative approach of using static predictions at runtime rather than predicting values directly at runtime. This way, the dynamic runtime calculation overhead is avoided and the resulting data access overhead is also reduced by the use of a novel concept called 'Compressive Hashing (CH)'. Since, the runtime system can query for energy predictions at irregular time intervals, random access overhead can considerably be reduced by using CH. Thus by using CH, the runtime system can request for system predictions anytime on-demand with less access overhead.

Genetic programming as a technique for evolving compressive hash functions was explained and a lossy CH mechanism was developed to improve the ability to hash more entries into the same table. Two application programs (*DCRAW* and *GDB*) were chosen and CH was used to improve the runtime access of energy/time predictions by $3x$ to $9x$. In a parallel supercomputer where the power prediction error gets amplified by the number of nodes, it seems obvious that the static prediction mappings be created for runtime use. Thus, current and future processors can benefit greatly from the static prediction and runtime access of system properties.

7.3 Future Work

Following lists the areas and research directions that can be explored to further improve this thesis findings.

- Instruction-level cost models described in this thesis do not include dynamic events such as cache mispredictions or branching effects because of the fact that memory footprints and branching information are not completely available at compile time. Even if gathering accurate information is not entirely possible, ways to get probabilistic information can be studied and used.

- Other ways to improve the instruction-level analysis such as including additional instruction classes can also be explored.

- Not all prediction can be done at compile time. The compiler cannot know which dynamically linked libraries an executable will use. Ways to include such runtime information can be explored further. For example, predictions computed for all paths can then be combined with predictions for the dynamically linked libraries when the program is

loaded. To reduce loading time, the merged predictions can be cached and updated only when the executable or dynamically linked library changes.

- The CH technique explained in this thesis uses GP technique for evolving compressive hash functions. Currently, the GP software takes several hours to run and produce the results. Different efficient techniques that improves the throughput of GP can be studied.

BIBLIOGRAPHY

[1] Frank Bellosa. The benefits of event-driven energy accounting in power-sensitive systems. In *EW 9: Proceedings of the 9th workshop on ACM SIGOPS European workshop*, pages 37–42, New York, NY, USA, 2000. ACM Press.

[2] Frank Bellosa, Andreas Weissel, Martin Waitz, and Simon Kellner. Event-driven energy accounting for dynamic thermal management. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power (COLP'03)*, New Orleans, LA, September 27 2003.

[3] A. J. C. Bik and H. A. G. Wijshoff. Advanced compiler optimizations for sparse computations. In *Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 430–439, New York, NY, USA, 1993. ACM Press.

[4] Carlo Brandolese, William Fornaciari, Fabio Salice, and Donatella Sciuto. An instruction-level functionally-based energy estimation model for 32-bits microprocessors. In *Design Automation Conference*, pages 346–351, 2000.

[5] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA '00: Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 83–94, New York, NY, USA, 2000. ACM Press.

[6] Peter F. Brown, Peter V. Desouza, Robert L. Mercer, Vincent J. Della Pietra, and Jenifer C. Lai. Class-based n-gram models of natural language. *Computational Linguistics*, 18(4):467–479, 1992.

[7] William B. Cavnar and John M. Trenkle. N-gram-based text categorization. In *Proceedings of SDAIR-94, 3rd Annual Symposium on Document Analysis and Information Retrieval*, pages 161–175, Las Vegas, US, 1994.

[8] C. Chakrabarti and D. Gaitonde. Instruction level power model of microcontrollers. In *ISCAS'99: Proceedings of the IEEE International Symposium on Circuits and Systems*, volume 1, pages 76–79, Orlando, FL, USA, July 1999.

[9] C.H.Gebotys and R.J.Gebotys. Statistically based prediction of power dissipation for complex embedded DSP processors. *Microprocessors and Microsystems*, 23(3):135–144, October 1999.

[10] Richard J. Cichelli. Minimal perfect hash functions made simple. *Commun. ACM*, 23(1):17–19, 1980.

[11] Paul R. Cohen and Charles A. Sutton. Very predictive ngrams for space-limited probabilistic models. In *Advances in Intelligent Data Analysis V*, pages 134–142, 2008.

[12] Gilberto Contreras and Margaret Martonosi. Power prediction for intel xscale®processors using performance monitoring unit events. In *ISLPED '05: Proceedings of the 2005 international symposium on Low power electronics and design*, pages 221–226, New York, NY, USA, 2005. ACM.

[13] Keith D. Cooper and Todd Waterman. Understanding energy consumption on the c62x. In *Workshop on Compilers and Operating Systems for Low Power (COLP 02, co-located with PACT 02)*, Charlottesville, Virginia, USA, September 2002.

[14] Advanced Micro Devices. Amd athlon processor x86 code optimization guide. In *AMD Technical Document 22007*, 2002.

[15] Advanced Micro Devices. Software optimization guide for amd family 10h processors. In *AMD Technical Document 40546*, 2008.

[16] H. G. Dietz. Speculative predication across arbitrary interprocedural control flow. In *Languages and Compilers for Parallel Computing: 12th International Workshop, LCPC'99*, volume 1863, pages 432–446, London, UK, June 2000. Springer-Verlag.

[17] Henry G. Dietz and William R. Dieter. Compiler and runtime support for predictive control of power and cooling. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2006), Workshop on High-Performance Power-Aware Computing (HPPAC)*. IEEE, IEEE Press, April 2006.

[18] H.G. Dietz and T.I. Mattox. Compiler optimizations using data compression to decrease address reference entropy. In *Proceedings of Languages and Compilers of for Parallel Computing*, July 2002.

[19] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerishce Mathematik*, 1:269–271, 1959.

[20] Jack Dongarra, Kevin London, Shirley Moore, Phil Mucci, and Dan Terpstra. Using papi for hardware performance monitoring on linux systems. In *Linux Clusters: The HPC Revolution*, June 2001.

[21] Fabio Ferri, Gabriella Righini, and Enrico Paganini. Inversion of low-angle elastic light-scattering data with a new method devised by modification of the Chahine algorithm. *Applied Optics*, 36(30):7539–7550, October 1997.

[22] L. Galescu. Augmenting words with linguistic information for n-gram language models, 1999.

[23] C. Gebotys and R. Gebotys. An empirical comparison of algorithmic, instruction and architectural power prediction models for high performance embedded DSP processors. In *ISLPED'98: Proceedings of the International Symposium on Low Power Electronics and Design*, pages 121–123, 1998.

[24] Selim Gurun and Chandra Krintz. A run-time, feedback-based energy estimation model for embedded devices. In *CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, pages 28–33, New York, NY, USA, 2006. ACM.

[25] Le Q Ha, P Hanna, D W Stewart, and F J Smith. Reduced n-gram models for english and chinese corpora. In *Proceedings of the COLING/ACL on Main conference poster sessions*, pages 309–315, Morristown, NJ, USA, 2006. Association for Computational Linguistics.

[26] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *J. Comput. Secur.*, 6(3):151–180, 1998.

[27] Adolfy Hoisie, Olaf Lubeck, and Harvey Wasserman. Performanc and scalability analysis of teraflop-scale parallel architectures using multdimensional wavefront applications. *Internation Journal of High Performance Computing Applications*, 14(4):330–346, 2000.

[28] Intel. Appendix a.2. In *iAPX 432 General Data Processor Architecture Reference Manual*, pages A–13 – A–22, January 1981.

[29] Intel. Intel 64 and ia-32 architectures software developer's manual. In *Intel Technical Document 253665*, 2008.

[30] Canturk Isci, Gilberto Contreras, and Margaret Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 359–370, Washington, DC, USA, 2006. IEEE Computer Society.

[31] Sang jeong Lee, Hae kag Lee, and Pen chung Yew. Runtime performance projection model for dynamic power management, 2007.

[32] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. Improving locality using loop and data transformations in an integrated framework. In *In Proceedings of the 31th IEEE/ACM International Symposium on Microarchitecture*, pages 285–297, 1998.

[33] Ben Klass, Don Thomas, Herman Schmit, and David Nagle. Modeling inter-instruction energy effects in a digital signal processor. In *Power Driven Microarchitecture Workshop in Conjunction with International Symposium on Computer Architecture*, Barcelona, Spain, June 1998.

[34] Donald E. Knuth. In *The Art of Computer Programming*, volume 3, 1975.

[35] Masaaki Kondo, Shinichi Tanaka, Motonobu Fujita, and Hiroshi Nakamura. Reducing memory system energy in data intensive computations by software-controlled on-chip memory. In *In Proceedings of the Workshop on Compilers and Operating Systems for Low Power (COLP 02), co-located with PACT 02*, 2002.

[36] John R. Koza. *Genetic programming: on the programming of computers by means of natural selection*. MIT Press, Cambridge, MA, USA, 1992.

[37] Chandra Krintz, Ye Wen, and Rich Wolski. Predicting program power consumption. Technical report, University of California, Santa Barbara, July 2002.

[38] Sheayun Lee, Andreas Ermedahl, Sang Lyul Min, and Naehyuck Chang. An accurate instruction-level energy consumption model for embedded RISC processors. In *LCTES'01: Proceedings of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 1–10, June 2001.

[39] David D. Lewis. In *http://www.daviddlewis.com/resources/testcollections/reuters21578/.*

[40] Bin Lu, Fenlin Liu, Xin Ge, Bin Liu, and Xiangyang Luo. A software birthmark based on dynamic opcode n-gram. *International Conference on Semantic Computing*, 0:37–44, 2007.

[41] Nihar R. Mahapatra, Jiangjiang Liu, and Krishnan Sundaresan. The performance advantage of applying compression to the memory system. In *Proceedings of the workshop on memory system performance*, 2003.

[42] H. Masataki and Y. Sgisaka. Variable-order n-gram generation by word-class splitting and consecutive word grouping. *Acoustics, Speech, and Signal Processing, IEEE International Conference on*, 1:188–191, 1996.

[43] Timothy I. Mattox, Henry G. Dietz, and William R. Dieter. Sparse flat neighborhood networks (sfnns): Scalable guaranteed pairwise bandwidth and unit latency. In *Proceedings of the Fifth Workshop on Massively Parallel Processing (WMPP'05) held in conjunction with the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2005)*, 2005.

[44] Kathryn McKinley and Steve Carr. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18:424–453, 1996.

[45] Ethan Miller, Dan Shen, Junli Liu, and Charles Nicholas. Performance and scalability of a large-scale n-gram based information retrieval system. *Journal of Digital Information*, 1, 2000.

[46] Muthulakshmi Muthukumarasamy and Henry Dietz. Empirical evaluation of compressive hashing. In *Workshop on Compilers for Parallel Computers*, January 2006.

[47] P.A.Franaszek and J.T.Robinson. On internal organization in compressed random access memories. *IBM Journal of Research and Development*, November 2001.

[48] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, I Thomas, and Katherine Yelick. A case for intelligent ram: Iram. *IEEE Micro*, 17, 1997.

[49] Kasper Peeters. In *http://www.aei.mpg.de/ peekas/tree/.*

[50] Sumit Roy, Raj Kumar, and Milos Prvulovic. Improving system performance with compressed memory. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium*, pages 23–27, April 2001.

[51] M. Sami, D. Sciuto, C. Silvano, and V. Zaccaria. Instruction-level power estimation for embedded VLIW cores. In *CODES '00: Proceedings of the eighth international workshop on Hardware/software codesign*, pages 34–38, New York, NY, USA, 2000. ACM Press.

[52] Robert W. Sebasta and Mark A. Taylor. Minimal perfect hash functions for reserved word lists. *SIGPLAN Not.*, 20(12):47–53, 1985.

[53] B. A. Sheil. Median split trees: a fast lookup technique for frequently occuring keys. *Commun. ACM*, 21(11):947–958, 1978.

[54] Manhung Siu and M. Ostendorf. Integrating a context-dependent phrase grammar in the variable n-gram framework. *Acoustics, Speech, and Signal Processing, IEEE International Conference on*, 3:1643–1646, 2000.

[55] Manhung Siu and M. Ostendorf. Variable n-grams and extensions for conversational speech language modeling. *Speech and Audio Processing, IEEE Transactions on*, 8:63–75, 2000.

[56] Alan Jay Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, 1982.

[57] Renzo Sprugnoli. Perfect hashing functions: a single probe retrieving method for static sets. *Commun. ACM*, 20(11):841–850, 1977.

[58] S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel. An accurate and fine grain instruction-level energy model supporting software optimizations. In *PATMOS'01: Proceedings of the International Workshop on Power And Timing Modeling, Optimization and Simulation*, September 2001.

[59] Ronald L. Rivest Clifford Stein Thomas H. Cormen, Charles E. Leiserson. In *Introduction to Algorithms*, 2001.

[60] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: A first step towards software power minimization. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(4):437–445, 1994.

[61] V. Tiwari, S. Malik, A. Wolfe, and M. Lee. Instruction level power analysis and optimization of software. *Journal of VLSI Signal Processing*, pages 1–18, 1996.

[62] Julian R. Ullmann. A binary n-gram technique for automatic correction of substitution, deletion, insertion and reversal errors in words. *The Computer Journal*, 20(2):141–147, 1977.

[63] Shougo Shimize Yeon-Dae Kwon and Hisashi Narimatsu. Prediction of glycosyltransferases synthesizing glycoconjugates using variable-length n-gram model. *Japanese Society for Bioinformatics*, 2004.

[64] Y. Zhang and R. Gupta. Data compression transformations for dynamically allocated data structures. *International Conference on Compiler Construction*, pages 14–28, April 2002.

VITA

Muthulakshmi Muthukumarasamy was born in Tirunagar, Madurai, a town in southern India on May 9, 1978. She received her undergraduate degree in Electronics and Communication Engineering from Thiagarajar College of Engineering, Madurai in May 1999. She joined University of Kentucky in Spring 2000 for her graduate education in the department of Electrical and Computer Engineering. She held various teaching and research assistant positions throughout her graduate student career, and joined the KAOS research group in the Fall of 2004 to pursue her doctoral research. She is currently working as a Senior Member of Technical Staff at Cadence Design Systems, Chelmsford, MA.