國立臺灣大學電機資訊學院資訊工程學系
碩士論文
Department of Computer Science and Information Engineering
College of Electrical Engineering and Computer Science
National Taiwan University
Master Thesis

尋找多核心系統架構程式最佳化機會之新方法

A New Prospect in Finding Optimization Opportunities for Multicore Architectures

林敬棋
Ching-Chi Lin

指導教授：劉邦鋒 博士
Advisor: Pangfeng Liu, Ph.D.
共同指導教授：游本中 博士
Co-Advisor: Pen-Chung Yew, Ph.D.

中華民國99年7月
July 2010

國立臺灣大學碩士學位論文
口試委員會審定書

尋找多核心系統架構程式最佳化機會之新方法

A New Prospect in Finding Optimization Opportunities for Multicore Architectures

本論文係林敬棋君（學號 R97922128）在國立臺灣大學資訊工程學系完成之碩士學位論文，於民國 99 年 7 月 1 日承下列考試委員審查通過及口試及格，特此證明

口試委員：

(指導教授)

系主任

誌謝

首先要感謝指導教授劉邦鋒教授兩年來的教導，無論是在專業知識、研究態度或者是報告技巧方面，都令我獲益良多，從最初甫進實驗室對研究領域一知半解，到今日能順利完成這篇論文，得到碩士學位，都要感謝老師的教導有方。雖然平常看起來很嚴肅，老師實際上人很好，總是用生動詼諧的例子來指導我們。

感謝實驗室所有的學長同學以及學弟們，這兩年的研究路上，因爲有你們，所以我們一路走來，並不寂寞。一同奮鬥的同學們，津豪、宇修、志瑋、以千、志庭，感謝你們這兩年來，無論是在修課、報告，茶餘飯後的閒聊，到後來論文寫作、口試準備時的各種幫忙。這兩年中靠著互相的嘴砲以及垃圾話，給實驗室帶來了許多的歡樂。

另外要感謝大學的老朋友們，不論是在研究所一起努力，或者是去當兵跟就業的各位，與你們保持聯繫，吃飯聊天或者出去玩都是一件很快樂的事情。還要感謝這兩年在台大的日子裡，台大資訊系及中研院資訊所各位老師對我的教導，以及其他曾對我有過幫助的人們，謝謝你們。

最後，感謝我的家人一路走來對我的支持，我能有今天的成就，全是因爲在成長的路上，有你們對我的包容和關愛，以及對我的信任。我愛你們。

結束是爲了迎接下一個開始，祈許未來的路上我能確立自己的目標，找到值得陪伴的人，全心全意的過著每一天。

摘要

程式最佳化主要可以分爲兩種方式－靜態最佳化以及動態最佳化，這兩種最佳化方式在處理單核心系統架構中的程式都有不錯的表現。但是在多核心系統新架構下，兩種最佳化方式在找尋程式中的最佳化機會時，都沒有將多執行緒程式執行 緒間的互動列入考慮。我們的目的是要發展出一個能夠辨認出執行緒間互動的技術，並且利用這些資訊來幫助程式最佳化。經由觀察發現，執行緒間的互動像是 競爭公用快取記憶體，可能導致“不穩定”的程式行爲。也就是說，執行程式中 相同的程式碼片段，理論上會有相同或相似的效能，但實際上卻有很大的差異， 就會被稱作“不穩定”。我們將這些不穩定的程式片段視爲“最佳化的機會”， 希望能夠藉由最佳化這些片段，使它們恢復穩定，進而提升執行效能。

我們提出了一個簡單而且有效率的方法，藉由取樣以及分析基本區塊的效能變化，來分辨出哪些基本區塊是“穩定”，而哪些是“不穩定”。分析得到的結果能夠讓動態最佳化器用來分辨在程式執行的過程中，哪些基本區塊是不穩定的，需要特別注意或特殊處理。我們可以藉由將取樣到的指令指標對應回它所屬的基本區塊，進而找到出現次數最多的基本區塊，用來當作程式執行區間的代表。藉此可以比較不同執行緒中有相同代表的程式區段的效能，計算出每個區段的效能變化。這個方法也能夠應用在單一執行緒的程式。

關鍵字　　多核心架構，動態最佳化，程式行爲

Abstract

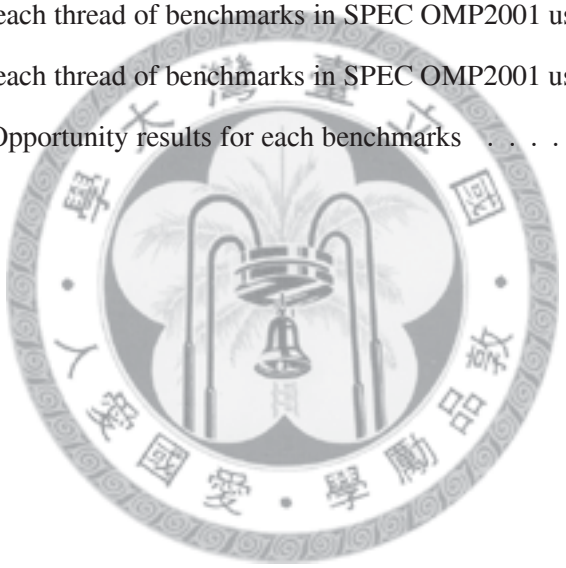
There have two groups of works in optimizing program execution in the literature – *static* and *dynamic* program optimization. To our best knowledge, neither of these optimizations, while looking for optimization opportunities, considers interactions among threads in multi-core architecture. Therefore we would like to develop techniques that can identify the presence of thread interactions and use it to guide possible optimization. We observe that interaction among threads, like competition for shared cache, can lead to "unstable" execution performance. That is, the same part of program will have very different performance characteristics, therefore we identify those parts of program as *dynamic optimization opportunity*, so that they can be optimized for better performance.

We propose a simple and efficient sampling method that analyzes performance variance among basic blocks, so as to differentiate "unstable" and "stable" basic blocks. The results from the analysis can be used as a reference to determine which parts of the program on which dynamic optimizer should make extra efforts during execution. By mapping EIP of each sample back to its basic block, we are able to choose representative basic block for each interval during execution, and compare the performance of each thread, so as to calculate the performance variance of each basic block. This sampling technique can also be applied to single-threaded programs.

Keywords Multi-core architecture, Dynamic optimization, Program behavior.

Contents

List of Figures



List of Tables

Chapter 1

Introduction

There have two groups of works in optimizing program execution in the literature – *static* and *dynamic* program optimization. Static optimization is performed by compiler while generating the executable program. Static optimizations, such as loop-unrolling, function in-lining, rename-register, are performed, depending on which level of optimization is specified by the user. The purpose of static optimizations is to reduce the size of the resulting machine code, or to create code that runs much faster, potentially increasing its size.

Dynamic optimization, on the other hand, works during execution runtime. The idea of dynamic optimization is that if a sequence of basic blocks, or *traces*, are "hot", meaning that the traces are executed very frequently, then optimizing these traces should improve performance. Thus dynamic optimization focuses on the *real execution behavior* of the program, rather than information that could be obtained from the source code during static compilation. There have been many dynamic optimizers in the literature, such as Dynamo [1], DynamoRIO [2], ADORE [3], and Mojo [4]. Other works like [5, 6] aim to optimize power consumption by configuring micro-architecture features according to the program behaviors at runtime .

Recently, Multi-core architecture becomes the mainstream in the computer industry. Thus, the multi-core computing raises new issues and opportunities for dynamic program optimizers. For example, most of the multi-core architectures have shared cache among cores. Competition for shared cache among threads may affect thread behavior and degrade overall performance. This is an issue in multi-core architectures but not in single core architectures.

The cache competition on multi-core architecture can cause *unstable* program behaviors, and static optimizers can not handle the cache competition in multi-core architectures since it can not assume that the generated code will only run in multi-core architectures. Hence the cache competition can only be handled by a dynamic optimizer during runtime.

During program execution, the dynamic optimizer can detect *unstable* program behaviors, and apply runtime optimization on those code segments. As a result we can view those code segments

that have an *unstable* behaviors as *dynamic optimization opportunities* in multi-core architectures.

The goal of this paper is to to identify which parts of the program execution are unstable, and which parts are stable in multi-core architectures. We want to make this distinction so that we can apply dynamic optimization techniques on those stable parts, and use *different* techniques on the unstable parts during runtime. It is important to have this distinction because we can only apply correct optimization after we know the behaviors of the program.

Our first try is to identify stable and unstable program execution from the time domain, namely to determine which time slots have stable execution, and which do not. We analyzed the runtime characteristics of parallel benchmarks from SPEC OMP2001, and found that even if these benchmarks run in multi-thread on multi-core with shared memory, most of the time the behaviors of individual threads are still very similar. The interferences we expected due to competition for shared memory/cache from different threads is not significant, and we were not able to determine which part of program execution is stable or unstable in time domain.

In order to reveal more information about the interferences among different threads, we propose a new observation method that observes the behaviors of threads not from time domain, but from *basic block domain*. For every basic block we determine whether it is "stable" or not by its behavior over the whole execution time. For those basic blocks with very unstable behaviors, i.e., with a large variance in the metrics we are interested, these basic blocks might provide good optimization opportunities.

It is very time-consuming to observe the performance of individual basic blocks, and the overheads from the observation can affect the original program behavior. In order to minimize the observation overheads, we use sampling techniques to measure the performance of individual interval at runtime, and choose a *representative* for that interval. By comparing the performance of each interval with the same representative, we can decide which intervals are "stable", and which are "unstable".

The major contribution of this work is that we propose a new prospect in finding optimization opportunities in multi-thread programs on multi-core architecture. Our simple and fast methods can be applied to identify those opportunities in programs. Both static and dynamic optimizers can use these opportunities to generate more efficient executables or perform more aggressive runtime optimizations.

The rest of the paper is as follows. Section 2 introduces related works. Section 3 discusses how to choose a representative extended instruction pointer (EIP) for a set of samples. Section 4 describes our sampling methodology. Section 5 describes our experimental results and analysis. Conclusion and future works are presented in Section 6.

Chapter 2

Related Works

A substantial amount of researches have been conducted on dynamic optimization. Dynamic optimizers such as Dynamo [1], DynamoRIO [2] and ADORE [3] use *hot trace* to improve execution efficiency. A hot trace is a series of frequently executed basic blocks. Hot trace is put into a code cache so that it can be accessed much faster next time it is required. Kister et al. [7] describes a continuous optimization framework that looks for stable phases in un-optimized code, and phase changes in previously optimized code before optimizing code.

Those dynamic optimizers identify traces according to the execution frequency of traces, but not to the *performance information* of traces, such as CPIs information. Our techniques can provide such information about those unstable program segments so that those dynamic optimizers can utilize those information to improve performance.

Phase detection is an important component of dynamic optimizers. A *phase* is defined as a set of intervals that executing the same parts of program, hence similar runtime behavior [8]. Basic Block Vector (BBV) are widely used as code signature of an execution interval. The BBV of an interval records the number of times each basic block is executed during execution of this interval.

There are many studies on phase detection and prediction [9, 10, 11, 12, 13] in recent years. Annavaram et al. [14] and Lau et al. [15] studied the relation between code and performance, and concluded that there exist a strong correlation between code and performance, and a weaker correlation between sampled code signature and performance. In [15], Lau et al. used receiver operating characteristic curves and low intra-phase coefficient of variation of CPI to prove their conclusion.

Jiang et al. [16], enlightened by the existence of strong correlations among program behaviors, propose a regression based framework to automatically identify a small set of behaviors that can lead to accurate prediction of other behaviors in a program. Sherwood et al. [17, 18, 19] use BBV as code signature to find periodic behavior of program phases. By comparing BBV between intervals, it is possible to detect stable phases and phase changes. Sherwood et al. [20] and Merten

et al. [21] proposed a hardware mechanism for detecting phase changes to support runtime optimization. Davies et al. implement iPART [22], an full-automated phase analysis and recognition tool.

Phase detection is also important in choosing simulation points. Hamerly et al. [8] use BBV for clustering, and choose simulation points from each cluster. Perelman et al. [23] also use clustering to help choose simulation points on multi-thread benchmarks.

Researches of phase detection usually use k-means [24, 25] clustering as classification method. The value of k is chosen such that it has the highest Bayesian Information Criterion (BIC) from 1 to K, where K is the maximum number of phases allowed. Our work avoid using k-means because it reduces the dimension of basic block vectors and potentially increases the risk of information lose. In addition with different random seed, k-means algorithm produces very different clustering results, which could not be comparable directly between two runs of k-means.

We summarize the difference of our work and previous works as follows. First, most of the previous works on phase detection and dynamic optimization target at sequential programs and evaluated their techniques with sequential benchmark suites such as CPU2000 or CPU2006. We aim to study interaction and interference between multiple threads in parallel programs, such as OMP2001 benchmark programs. Furthermore, in previous works, the goal of phase detection is either to find simulation points or to predict phases at runtime. Our goal is to find optimization opportunities (i.e., code segments that have unstable behaviors) in multi-thread programs, which requires very different phase analysis techniques.

Chapter 3

Choosing Representative

Many works in the literature use basic block, i.e., a block of instructions with a single entry and a single exit, as a basic unit in program behavior analysis. However the overheads of observing individual basic block is extremely large, which can destroy the program behavior we want to observe. Instead of taking every basic block into consideration, we take samples for every fixed amount of time. Previous studies show that take samples every 10^5 to 10^6 instructions, and group 10^7 to 10^8 instructions (about 100 samples) together as an interval is reasonable, so we will use this sampling rate throughout this paper.

Most of the previous works use Basic Block Vector (BBV) to represent an interval. A BBV contains the number of times a basic block being executed during an interval. Usually the vector is normalized according to the total number of samples in that interval.

The K-means clustering is widely used as a classification method in phase detection. The goal is to classify basic block vectors into clusters so that each cluster contains intervals with similar execution behavior.

Use K-means clustering method on BBVs can achieve good classification results, but it also raises the following issues. First, the number of cluster k is difficult to determine beforehand. Second, K-means clustering can generate different results due to initial random seeds. Last but not the least, before applying K-means clustering, we need to randomly project BBV into lower dimension to avoid the "dimension curse", i.e., the high overheads in dealing with the very large number of dimensions in original BBV. Therefore, the clustering results are not directly comparable between each other or between different runs, and we need another metric to represent an interval instead of using BBV.

3.1 Average EIP

At first we try to use the *average EIP* as a representative for an interval. Every interval can be identified and compared to each other by the average EIP of samples. This is a fast and intuitive way compared to BBV. But after some analysis, we find that even if this method is simple and fast, average EIP can not be used as our representative of an interval.

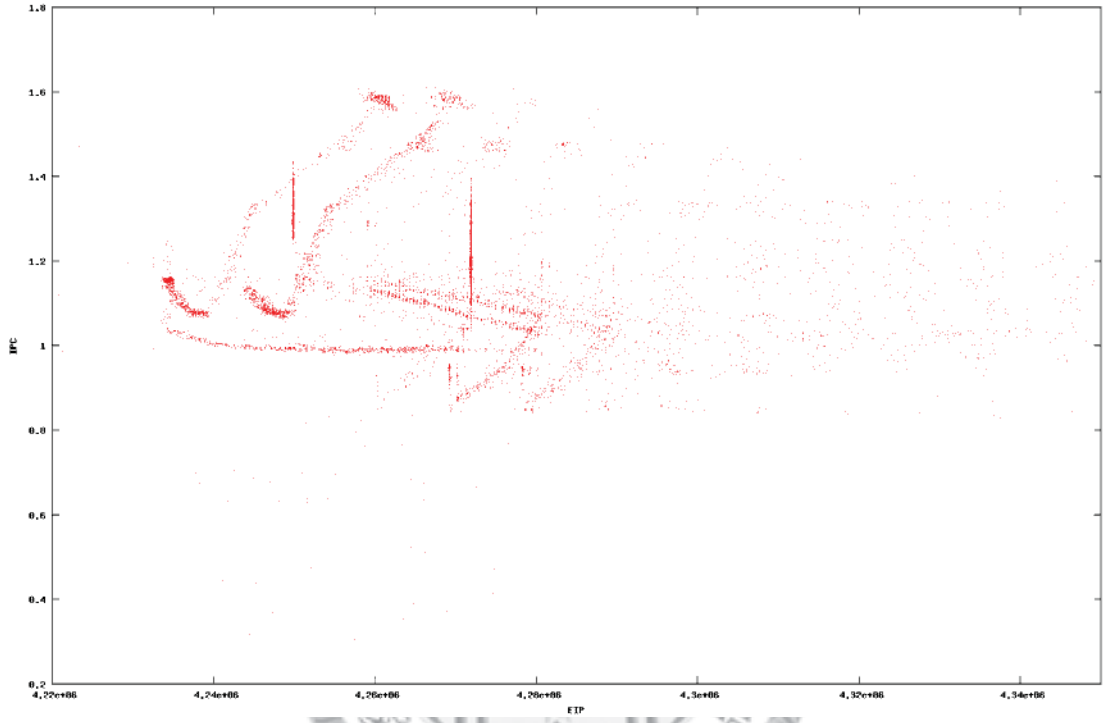


Figure 3.1: EIP-IPC relation for 314.mgrid_m of SPEC OMP2001

Figure 3.1 shows the performance results from mgrid using average EIP. Mgrid is a benchmark from SPEC OMP2001. Each data point represents an interval. The x coordinate is the average EIP of the interval and the y coordinate is the IPC(Instruction Per Cycle) of that interval. As we can see, there exists similar patterns in figure 3.1. For example, there are two sets of points that form a similar pattern at the left part of the figure. These repeated patterns raise an issue that if using average EIP as representative a good choice.

To find the reason that cause these repeated patterns, k-means clustering is applied. For each interval we collect its sampled BBV and apply k-means. Each cluster is assigned with different colors. Figure 3.2 shows the colored result. The color of each point in figure 3.2 indicates which cluster an interval is belonged to using k-means.Intervals from the same cluster should have the same color.

From Figure 3.1 and Figure 3.2, we can conclude that it is not appropriate to use the average EIP to represent an interval. A desirable result should be data points with the same color gathering

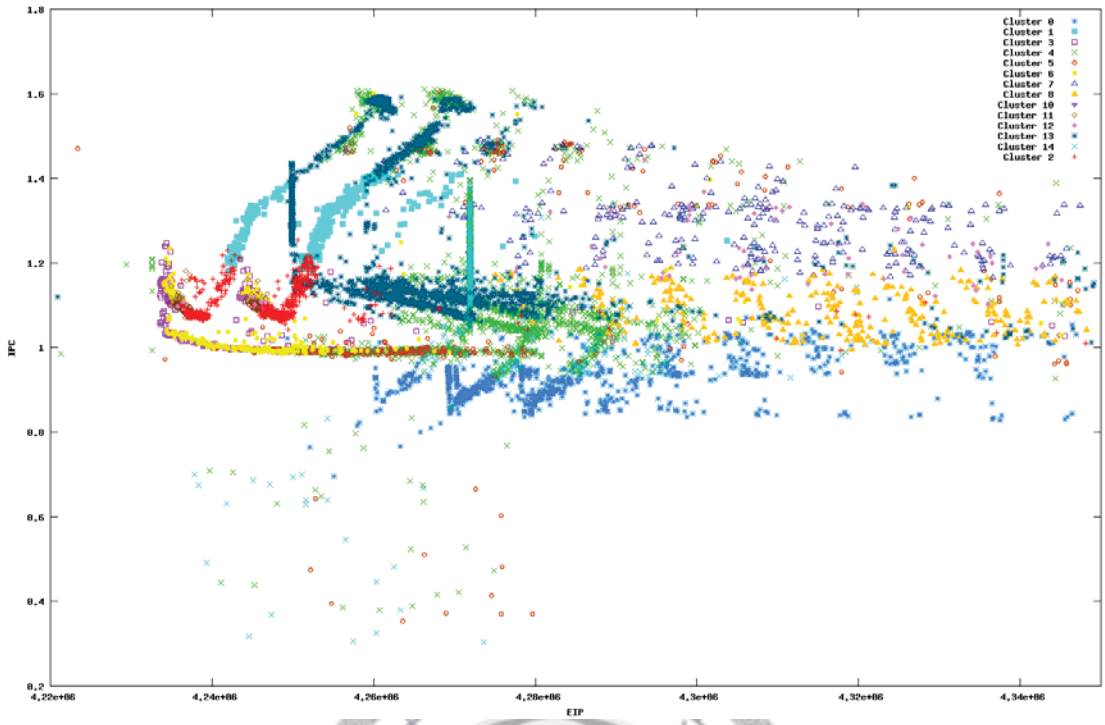


Figure 3.2: EIP-IPC relation after clustering for 314.mgrid_m of SPEC OMP2001

together, having similar EIP as representative. But as the figures show, those points with the same color separated through x-axis, and cause repeated patterns. The reason is that during program execution, instuction pointer may jump to instructions with large EIP, which can significantly increase the average EIP. For example, there are two sets of red points on the left part of Figure 3.2. These two set of intervals are from the same cluster result by applying k-means, but those intervals in the right set have samples with large EIP while the intervals in the left set do not. This result shows that even there are only very few samples with large EIP differ, the average EIP of an interval can have a significant change.

There is another reason for not using average EIP as representative. The basic block an average EIP belongs to may not actually be executed during that interval. We can conclude that it is not reasonable using the average EIP as representative for an interval.

3.2 Mode

To overcome the repeated pattern problem in Figure 3.1, we propose using the `mode` (i.e., the one that occurs most frequently) of EIP sampled in an interval to represent that interval. We use PIN [26] to build a EIP to basic block mapping table. For every samples in an interval, its EIP can be mapped to the corresponding basic block by mapping table. We choose the basic block that occurs most and use the EIP of the first instruction of that basic block to represent this interval.

Using the `mode` is much more suitable than using the average EIP. As discussed in the previous section, the average EIP will shift significantly even when there is only one very large EIP in the samples. By using `mode` in choosing representative, those instructions with large EIP will be treated as "noise" and filtered out, so that using `mode` will not be affected by them. The representative chosen using `mode` is surely to be executed during an interval.

Chapter 4

Methodology

Our goal is to find optimization opportunities of a program. If a set of basic block has an unstable behavior during runtime, then we consider it as an optimization opportunity. To compare the behavior of each basic block, we need to collect the information of behaviors during the program execution.

4.1 Sampling and Grouping

We collect the performance and execution information by sampling at program runtime. Perfmon [27], a performance monitoring tool, is used to help us to do sampling. A sample is taken every fixed number of instructions. For every sample, Perfmon records one EIP, clock cycles used during this sample, and some other relevant information. And for every fixed size of samples, we group them into a data point, or an interval. The total sum of clock cycle used in an interval is calculated, and divided by the total instruction retired to get the cycle per instruction (CPI) to be the performance of this interval.

4.2 Representative

The most appearance basic block of an interval is used as representative. Choosing the representative EIP from an interval is an interesting problem. Section 3 discussed the reasons of using the `mode` (one that occurs most frequently) in choosing the representative instead of using sample BBV or average EIP. The result indicates that using the average EIP as the representative can cause repetitive patterns on the EIP-IPC graph like Figure 3.1, even if the data points of those patterns are executing the same part of program. In another word, data points from executing the same part of program could be separated and not be able to compare with each other. Another reason is that the most appearance EIP is surely to be executed during that interval, which the average one may not.

One thing worth to mention is that an instruction is too small to be an optimization unit. A much more suitable unit should be one or a set of basic block. As a result, for every EIP Perfmon sampled, which basic block that EIP belongs to is important. We build a lookup table by PIN [26], which contains the EIP range of each basic block. With this table each sampled EIP can be mapped to basic block it belongs. The most appearance basic block of an interval is selected, and the first EIP is used to be the representative of this interval.

4.3 Optimization Opportunity

After collecting the representative EIP and performance data for each interval, the standard deviation(Std) and coefficient of variance(CoV) of CPI for intervals with the same representative can be calculated. EIPs with large Std or CoV of CPI will be marked as "unstable". If an EIP is said to be "unstable", it means that the basic block starting with this EIP and its neighboring basic blocks have unstable behavior during execution. It is with high possibility to get performance gain applying dynamic optimization to those basic blocks during runtime.

To fit out goal of identifying large performance variance, CoV of CPI is chosen as the evaluation metric to decide if an EIP is unstable or not. But using only CoV of CPI is not good enough, due to the fact that some EIPs have only a few intervals but with large CPI variance. Trying to optimize those during runtime can produce tiny performance gain but raise more overheads. The coverage of interval for a representative EIP should also be taken into consideration. The EIPs chosen as "unstable" should be with large CoV of CPI and medium coverage during execution. The evaluation function can be defined as:

$$OptimizationOpportunity = CoVofCPI \times Coverage \tag{4.1}$$

If the optimization opportunity is larger, it is more likely to get performance gain during runtime optimization. A threshold T_{oo} is applied, and only EIPs with opportunity over T_{oo} will be marked and noticed by optimizer.

Chapter 5

Experimental Results

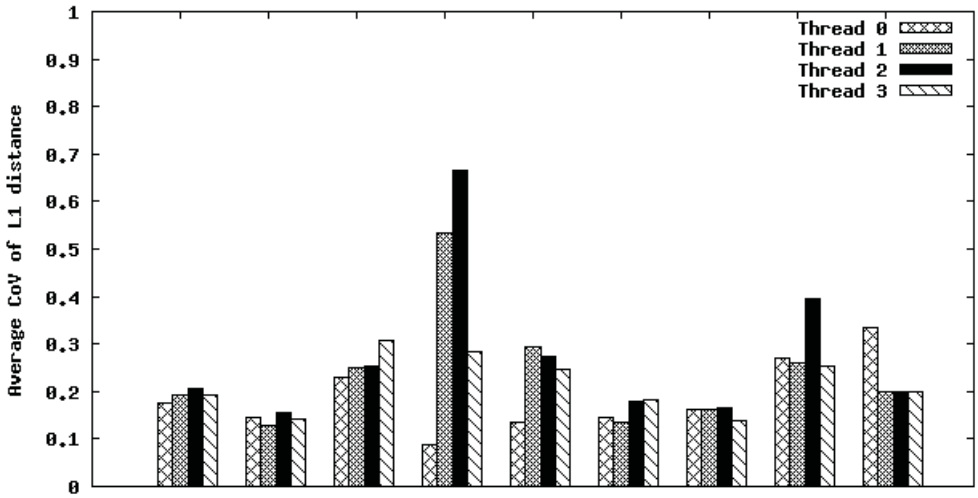
5.1 Experimental Settings

The experimental environment is as follows: We use Intel Nehalem Core i7, CPU model Intel(R) Core2(R) CPU 975 @ 3.33GHz as our platform. A CPU has four cores, each with a 256KB L2 cache and shared a 8MB L3 cache. The memory size is 12 GB. Spec OMP2001 is used as our benchmark suite, with medium size dataset. Every benchmark runs with 4 threads, one per core, and is monitored during execution with Perfmon [27]. Sampling rate is set to be 10^5 instructions per sample.

We further analyze the monitored results of Perfmon to get the representative and performance data, which are described as follows. The monitored results are recorded in four text files, one for each thread, with the EIP and performance information for each sample. Then we use basic block look up table to map EIPs to corresponding basic blocks. The basic block look up table is generated before the experiment using PIN [26]. After mapping EIPs to basic blocks, samples are grouped into intervals. The interval size is set to 100 samples. The most frequently occurring basic block is chosen as the representative of this interval. The performance behavior, in our work, CPI, is calculated by the performance information of those samples in an interval.

5.2 Classification Results

Before comparing the results of each thread, we first examine the effectiveness. After knowing the representative of each interval, we group those intervals with the same representative basic block into a cluster, and form an average BBV of the cluster. Having the average BBV, we calculate the Manhattan distance between BBVs and the average BBV in a cluster. The idea of using Manhattan distance is that if two BBVs have a large Manhattan distance, the basic blocks they execute should be very different. We calculate the correlation of covariance (CoV) of the Manhattan distance for each cluster in each thread.



	ammp	applu	apsi	art	equake	fma3d	mgrid	swim	wupwise
Thread 0	0.174	0.146	0.230	0.088	0.134	0.144	0.162	0.271	0.334
Thread 1	0.192	0.128	0.251	0.533	0.294	0.135	0.163	0.261	0.198
Thread 2	0.207	0.154	0.254	0.666	0.274	0.179	0.166	0.394	0.201
Thread 3	0.193	0.142	0.306	0.285	0.245	0.182	0.137	0.255	0.201

Figure 5.1: Average CoV of the L1 distance between BBVs using the same representative

Figure 5.1 shows the average CoV of Manhattan distance for clusters in each thread using O2 optimization level. From the table we observe that most of the benchmarks have small average CoV (under 0.3), which indicates that the BBVs in the same cluster are similar. But for threads of some benchmarks, such as *art*, the value is over 0.5, which is large. The reason is that for these benchmarks there are many other ”frequent” basic blocks other than the most frequent basic block. If two different execution sequences have the same `mode` basic block, they will be placed into the same cluster and cause the CoV of this cluster to increase. But still, for most cases, the CoV of clusters are small.

5.3 Performance Comparison Results

After choosing the representative basic block for each interval and making those with the same representative into a cluster, we compare the performance results for each cluster. First we compare the results between different optimization level, then pinpoint those “unstable” parts of programs.

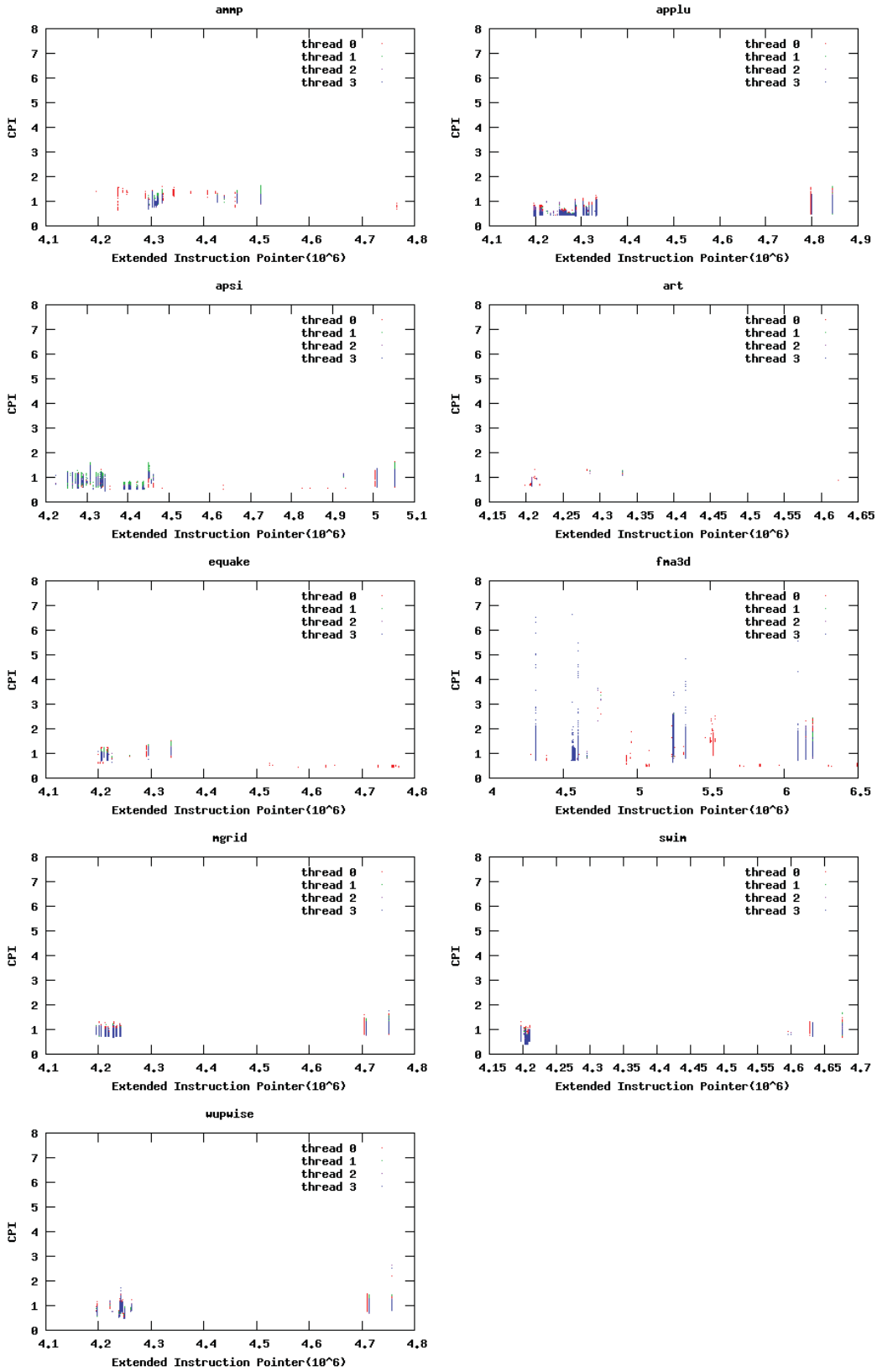


Figure 5.2: CPI results of each thread of benchmarks in SPEC OMP2001 using O0

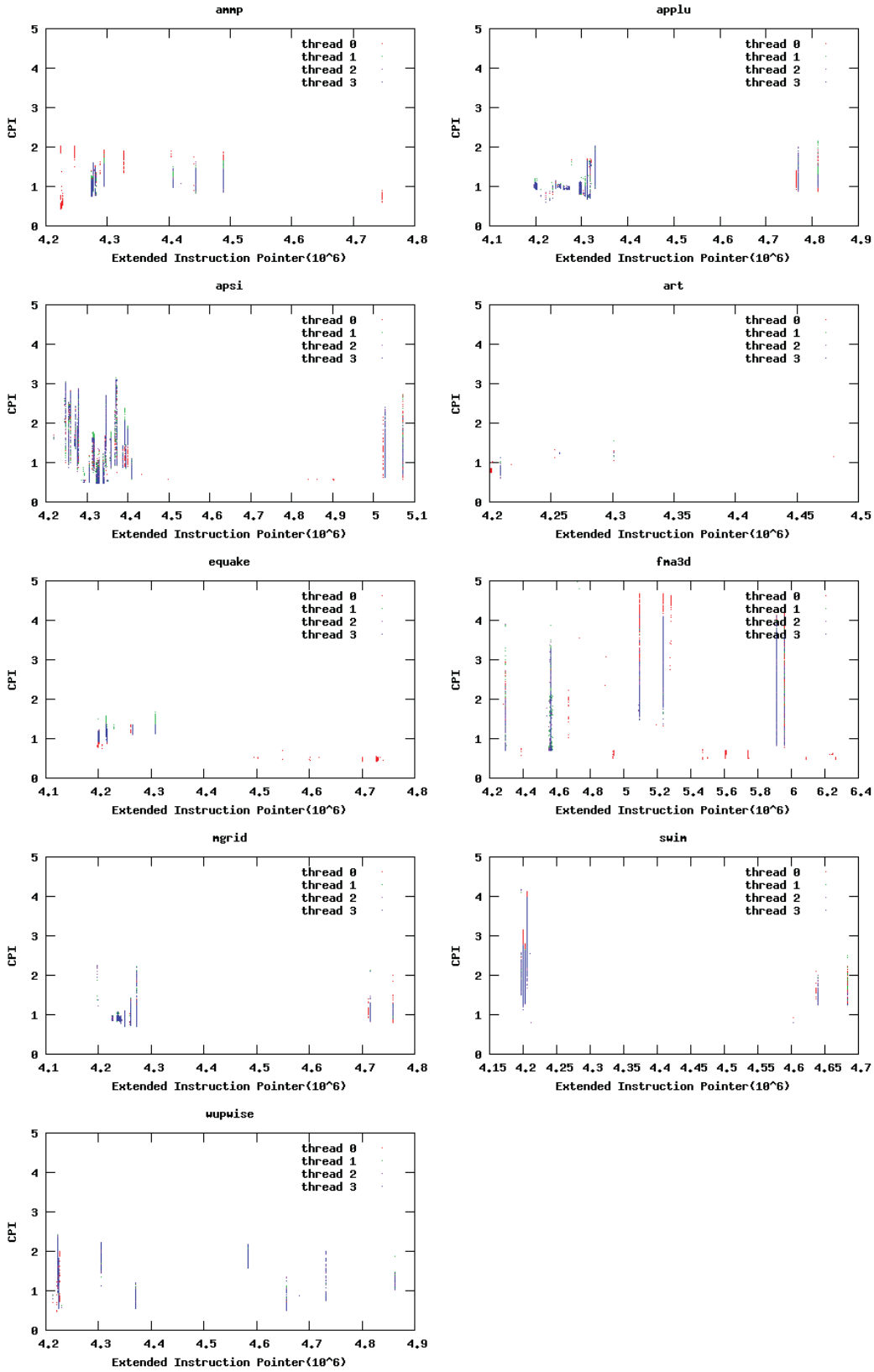


Figure 5.3: CPI results of each thread of benchmarks in SPEC OMP2001 using O2

5.3.1 Optimization Level: O0 vs O2

Figure 5.2 and Figure 5.3 show the CPI results for each thread in different benchmarks. Every data point indicates an interval. The x-axis of each sub-figure is the beginning EIP of the representative basic block, and the y-axis is CPI value. Different colors indicate which thread an interval is taken from. Data points with the same x-coordinate means that they have the same representative basic block. If data points with the same x-coordinate spread widely, i.e., a long vertical line, then we say this basic block is "unstable". On the other hand, if the data points are close to each other, then the basic block is "stable" on performance in terms of CPI.

From Figure 5.2 we can see that for most of the benchmarks with O0 optimization level, the CPI values locate within the range $[0.5, 1.5]$. However, with O2 optimization level, both the average CPI and the variance become larger as Figure 5.3 shows. The main reason is that in order to produce executables that run faster, static optimization techniques such as loop-unrolling, function in-lining, rename-register, etc. are performed. The instructions used in the executable are more complicate compared to those with no optimization, which results in larger CPI. We choose O2 optimization level as our target of finding optimization opportunities because the CPI variances are much more significant at this optimization level.

5.3.2 Unstable parts

Intervals with the same representative(same cluster) that have large correlation of covariance(CoV) of CPI are called "unstable". But it is not good enough using only CoV of CPI as mentioned in section 4.3. Coverage of the representative is taken into consideration to help deciding optimization opportunity. Figure 5.4 shows the optimization opportunity, defined in Section 4.3, of each representative in each benchmark. The threshold T_{oo} is set to be 0.01. If optimization opportunity of a cluster exceeds the threshold T_{oo}, the cluster is marked as "unstable". Note that these threshold values are determined by observing, further work is needed to obtain appropriate thresholds.

By comparing Figure 5.3 and Figure 5.4, it is clear that some basic blocks seem to have large CPI variance, but small optimization opportunity due to small overall coverage. From Figure 5.4 we can observe that for most basic blocks, the optimization opportunity values are likely to be small. But still there are some with large values, which will be marked as "unstable".

After marking those "unstable" basic blocks, we can generate files containing information including EIPs of the "unstable" basic blocks. Dynamic optimizer will be able to use these information to improve performance during program execution. It is not clear how much performance gain can be achieved due to lack of such dynamic optimizers that can take advantage of these information. Nevertheless, we provides a mechanism that may help future dynamic optimizer design. First, our method can be used to collect information on which part of the program is "unstable".

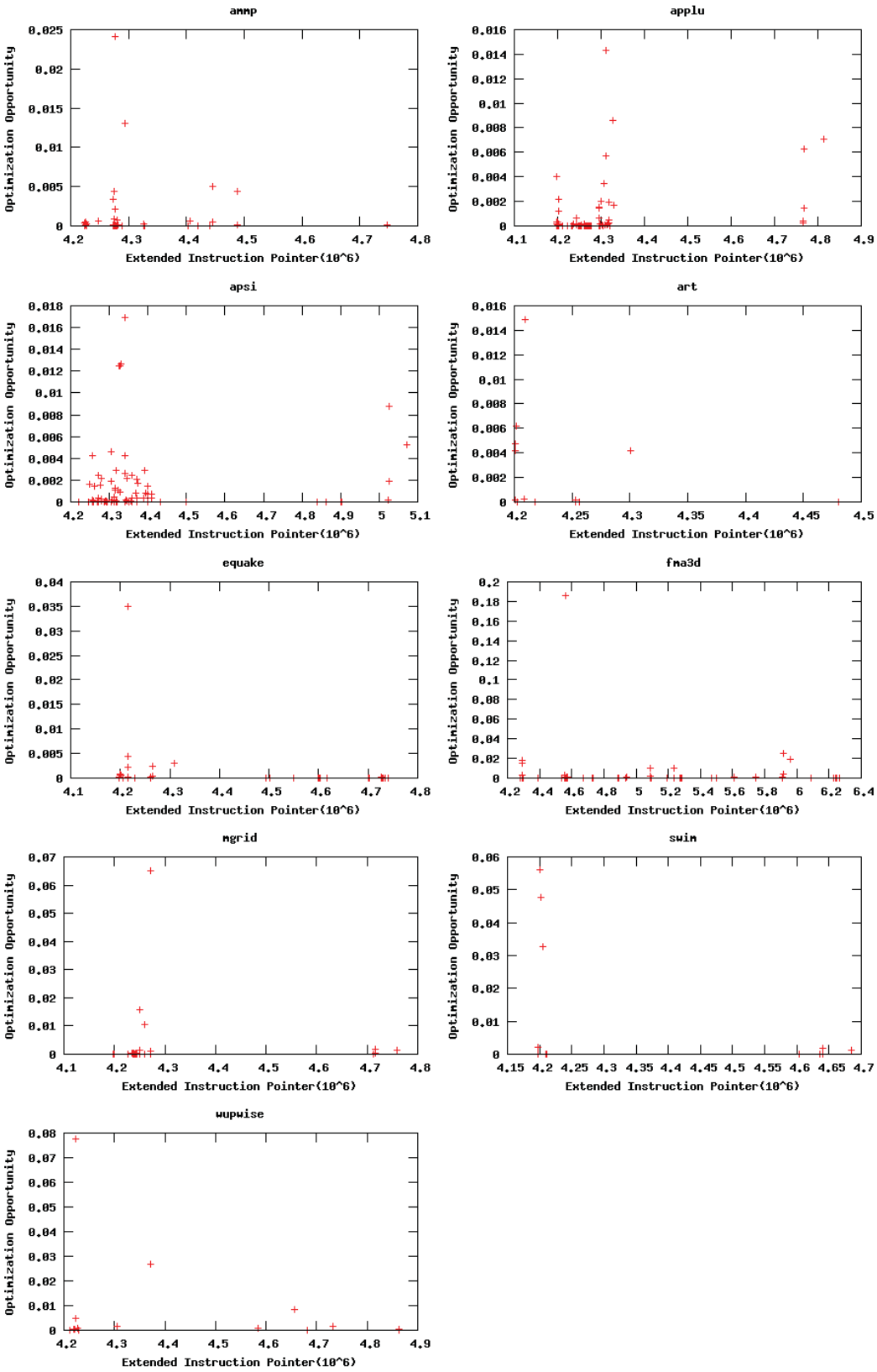


Figure 5.4: Optimization Opportunity results for each benchmarks

Then a dynamic optimizer can set "check points" at the first EIP for those unstable basic, and take extra care in generating high performance code instead of simply forming a trace and placing it into code cache.

5.4 Case Study: Swim

In this section we use benchmark *swim* to test if the "unstable" information generated by our work really helps performance. Due to the lack of help from dynamic optimizer, we can only do some optimization at source code level. First we choose an "unstable" basic block from our results of benchmark swim. Having this basic block we can look it up in the assembly code. The assembly code is generated from source code compiled by Intel(R) Compiler. Thus we can find the corresponding source code of the basic block, and do some optimization by human knowledge.

Table 5.1: Execution time results

	original	modified
	4m36.743s	4m35.948s
	4m36.546s	4m35.745s
	4m36.547s	4m35.947s
	4m36.544s	4m35.950s
	4m36.546s	4m35.947s
	4m36.546s	4m35.950s
	4m36.547s	4m35.946s
	4m36.546s	4m35.746s
	4m36.547s	4m35.945s
	4m36.747s	4m35.745s
avg	4m36.586	4m35.887s

The optimization shows insignificant performance improvement. After the optimization, the source code is re-compiled using the same setting as the original program. The execution time results are shown in table 5.1. The result shows that after modification, the program runs slightly faster(about 0.7 secs) compared to the original one. The speedup is pretty steady. The result shows that by optimizing those "unstable" parts identified by our method, we can get some performance gains for sure. Still we should remember that this optimization is done by hand, and only modify the corresponding part of source code. It is expected to achieve more performance gains using dynamic optimizer during runtime.

5.5 Summary of Experimental Results

Our experiment results indiciate that the effectiveness of our method is quite good because most benchmarks have small average CoV (under 0.3), which indicates that the BBVs in the same cluster are similar. From Figure 5.4, we observe that for most basic blocks, the optimization opportunity values are likely to be small. But still there are some with value over threshold T_{oo}, and will be marked as "unstable". For the benchmark *swim*, we applied simple optimization to the "unstable" regsion of the source code manually. The experiment result shows insignificant but steady performance gain of the optimization.

Chapter 6

Conclusion

We propose a simple and fast method to find optimization opportunities for dynamic optimization. Our method works on both single and multi-threaded programs. By mapping the EIP of each sample back to basic block and choose a representative, we are able to compare the performance (in our case, CPI) of each thread, and calculate the performance variance of basic blocks. If the variance is small to a basic block, then we say that this basic block is "stable". On the other hand, if the variance is large, then we mark this basic block as "possible optimization opportunity" or "unstable". There are many reasons that can cause large performance variances. For example, a basic block with lots of memory instructions, or cache competitions between threads. Dynamic optimization may be able to utilize the information on those "unstable" regions and improve performance during runtime.

We also compare the two methods to present a set of samples using an EIP. In most of the existing phase detection works, EIP vector or basic block vector is used to present a set of samples. In this work, we try to find other ways than using cluster ID to represent a vector. The first method is using average EIP, but it turns out that the results are easily affected by EIPs sampled. Using average EIP as representative can cause repeated patterns. The other method is using the most frequently occurring EIP sampled as representative. We conclude that using the most frequent EIP sampled as representative is a much better choice than using the average EIP.

There are still many issues that need to be analyzed and discussed. The most important one is that we need an optimizer to help us evaluate the overall performance gain by applying optimization to a program using our analysis results. Another issue is to find a way to compare the performance of the same code segment using different optimization levels, which might tells us about which optimization level is more suitable to a procedure or a sequence of basic blocks in program. Compiler might be able to use such information to perform more accurate optimization. Analyzing other benchmarks on different architectures is also an important part of the future work.

Bibliography

[1] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *SIGPLAN Not.*, 35(5):1–12, 2000.

[2] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society.

[3] J. Lu, H. Chen, P.C. Yew, and W.C. Hsu. Design and implementation of a lightweight dynamic optimization system. *Journal of Instruction-Level Parallelism*, 6, April 2004.

[4] W.K Chen, S. Lerner, R. Chaiken, and D.M. Gillies. Mojo: A dynamic optimization system. In *Proceedings of the 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization*, 2000.

[5] C. Pereira, J. Lau, B. Calder, and R. Gupta. Dynamic phase analysis for cycle-close trace generation. In *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 321–326, New York, NY, USA, 2005. ACM.

[6] A.S. Dhodapkar and J.E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 233–244, Washington, DC, USA, 2002. IEEE Computer Society.

[7] T. Kistler and M. Franz. Continuous program optimization: A case study. *ACM Trans. Program. Lang. Syst.*, 25(4):500–548, 2003.

[8] G. Hamerly, E. Perelman, and B. Calderd. How to use simpoint to pick simulation points. *SIGMETRICS Perform. Eval. Rev.*, 31(4):25–30, 2004.

[9] A. Das, J. Lu, and W.C. Hsu. Region monitoring for local phase detection in dynamic optimization systems. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 124–134, Washington, DC, USA, 2006. IEEE Computer Society.

[10] P. Nagpurkar, C. Krintz, M. Hind, P.F. Sweeney, and V.T. Rajan. Online phase detection algorithms. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 111–123, Washington, DC, USA, 2006. IEEE Computer Society.

[11] G. Fursin, A. Cohen, M. O'Boyle, and O. Temam. A practical method for quickly evaluating program optimizations. In *Proceedings of the 1st International Conference on High Performance Embedded Architectures and Compilers*, pages 29–46, November 2005.

[12] J. Lau, M. Arnold, M. Hind, and B. Calder. Online performance auditing: using hot optimizations without getting burned. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 239–251, New York, NY, USA, 2006. ACM.

[13] C. Dubach, J. Cavazos, B. Franke, G. Fursin, M.F.P. O'Boyle, and O. Temam. Fast compiler optimisation evaluation using code-feature based performance prediction. In *CF '07: Proceedings of the 4th international conference on Computing frontiers*, pages 131–142, New York, NY, USA, 2007. ACM.

[14] M. Annavaram, R. Rakvic, M. Polito, J.Y. Bouguet, R.A. Hankins, and B. Davies. The fuzzy correlation between code and performance predictability. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 93–104. IEEE Computer Society, 2004.

[15] J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder. The strong correlation between code signatures and performance. In *ISPASS '05: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005*, pages 236–247, Washington, DC, USA, 2005. IEEE Computer Society.

[16] Y. Jiang, E.Z. Zhang, K. Tian, F. Mao, M. Gethers, X. Shen, and T. Gao. Exploiting statistical correlations for proactive prediction of program behaviors. In *CGO '10: Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 248–256, New York, NY, USA, 2010. ACM.

[17] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[18] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 45–57, New York, NY, USA, 2002. ACM.

[19] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and exploiting program phases. *IEEE Micro*, 23(6):84–93, 2003.

[20] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 336–349, New York, NY, USA, 2003. ACM.

[21] M.C. Merten, A.R. Trick, C.N. George, J.C. Gyllenhaal, and W.W. Hwu. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In *ISCA '99: Proceedings of the 26th annual international symposium on Computer architecture*, pages 136–147, Washington, DC, USA, 1999. IEEE Computer Society.

[22] B. Davies, J. Bouguet, M. Polito, and M. Annavaram. *iPART: An Automated Phase Analysis and Recognition Tool*. Technical Report IR-TR-2004-1-iPART, Intel Corporation, February 2004. ftp://download.intel.com/research/library/IR-TR-2004-1-iPART.pdf.

[23] E. Perelman, M. Polito, J.Y. Bouguet, J. Sampson, B. Calder, and C. Dulong. Detecting phases in parallel applications on shared memory architectures. In *IPDPS '06: Proceedings of the 20th International Parallel and Distributed Processing Symposium*, April 2006.

[24] G. Hamerly and C. Elkan. Alternatives to the k-means algorithm that find better clusterings. In *CIKM '02: Proceedings of the eleventh international conference on Information and knowledge management*, pages 600–607, New York, NY, USA, 2002. ACM.

[25] D. Pelleg and A.W. Moore. X-means: Extending k-means with efficient estimation of the number of clusters. In *ICML '00: Proceedings of the Seventeenth International Conference on Machine Learning*, pages 727–734, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[26] C.K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[27] Perfmon. http://perfmon2.sourceforge.net/.