Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 EECS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 04-038

Performance of Runtime Optimization on BLAST

Abhinav Das, Jiwei Lu, Howard Chen, Jinpyo Kim, Pen-chung Yew,
Wei-chung Hsu, and Dong-yuan Chen

October 15, 2004

Performance of Runtime Optimization on BLAST

Abhinav Das, Jiwei Lu, Howard Chen, Jinpyo Kim, Pen-Chung Yew, Wei-Chung Hsu
{adas, jiwei, chenh, jinpyo, yew, hsu}@cs.umn.edu

Dong-Yuan Chen (Microprocessor Research Lab, Intel Corporation)
dong-yuan.chen@intel.com

Abstract

Optimization of a real world application BLAST is used to demonstrate the limitations of static and profile-guided optimizations and to highlight the potential of runtime optimization systems. We analyze the performance profile of this application to determine performance bottlenecks and evaluate the effect of aggressive compiler optimizations on BLAST. We find that applying common optimizations (e.g. O3) can degrade performance. Profile guided optimizations do not show much improvement across the board, as current implementations do not address critical performance bottlenecks in BLAST. In some cases, these optimizations lower performance significantly due to unexpected secondary effects of aggressive optimizations. We also apply runtime optimization to BLAST using the ADORE framework. ADORE speeds up some queries by as much as 58% using data cache prefetching. Branch mispredictions can also be significant for some input sets. Dynamic optimization techniques to improve branch prediction accuracy are described and examined for the application. We find that the primary limitation to the application of runtime optimization for branch misprediction is the tight coupling between data and dependent branch. With better hardware support for influencing branch prediction, a runtime optimizer may deploy optimizations to reduce branch misprediction stalls.

1. Introduction

Compiler optimizations aim to generate the most efficient code for a program. Optimization algorithms follow general rules of efficiency. For example, fewer instructions will result in improved execution efficiency. Thus, redundancy elimination optimizations such as loop invariant code motion, common sub-expression elimination and many others are commonly implemented in compilers. These optimizations are indeed valuable, and applying these optimizations sometimes leads to vastly improved performance. However most modern day applications are dominated by two critical performance bottlenecks (1) Cache misses and (2) Branch mispredictions.

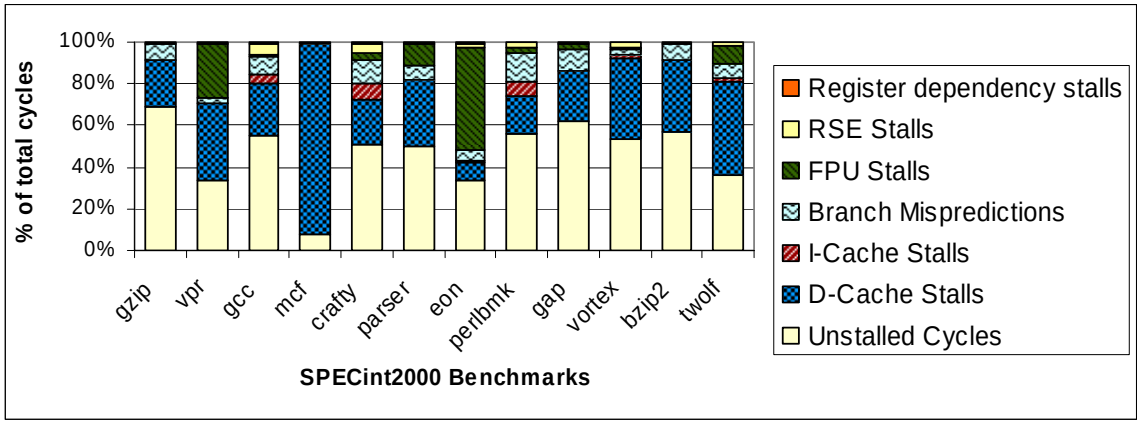

Figure 1: Breakdown of cycles for SPECint2000 benchmark programs on systems with Itanium-2 processors [16].

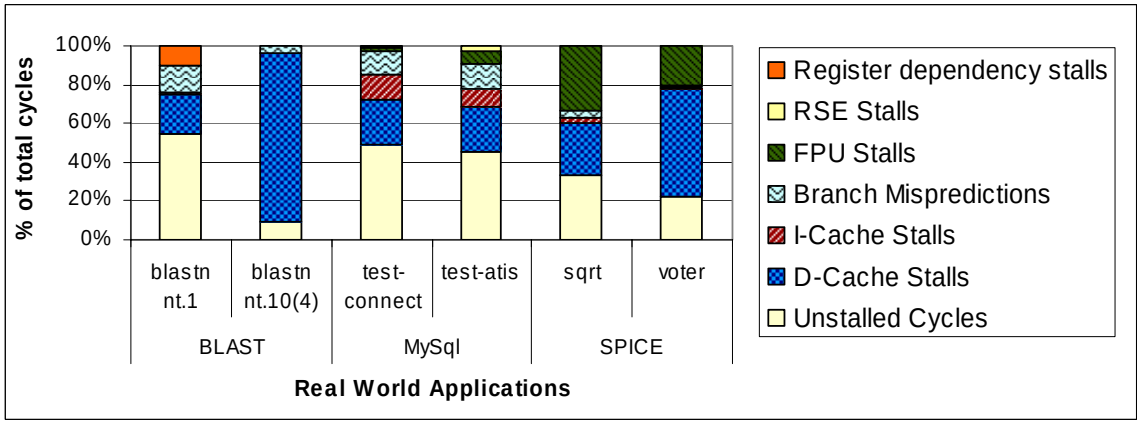

Figure 2: Breakdown of cycles of some real world applications on systems with Itanium-2 processors.

Figure 1 and Figure 2 show that SPEC benchmarks and some real world applications are dominated by such bottlenecks. Traditional optimizations cannot effectively address these bottlenecks. For example, in most compilers, loads are scheduled assuming they hit in the cache as scheduling for higher latency may lead to inferior performance. In particular, if such performance bottlenecks shift from one place to the other depending on the input data set, a statically generated binary is unlikely to address the performance needs effectively.

Profile-based optimizations that collect the runtime behavior of certain input sets have been applied successfully in code layout [4] [5], trace [6], super-block [7] and hyper-block formation [8]. Using execution profiles to identify biased branches has been quite effective. Hence, recent research has attempted to extend the idea of branch profile to value [9], cache miss [10] and data dependency [11] profiling. Many research results indicate that by using a few input sets one can accurately predict the program behavior of other input sets. This seems to suggest that a profile generated by a set of candidate inputs may be used to generate an optimized binary that is generally better than the original binary. However, the effectiveness of such optimizations in commercial compilers has so far been limited to improving program layout and trace/super-block formation. Recent research has also reported the difficulty in collecting a representative profile [12]. For real-world applications, since the input sets can vary significantly, it is not clear whether performance-critical cache misses and branch misprediction events can be accurately predicted by using a small set of training profiles.

In this study, we use a popular real world application, BLAST, to understand and evaluate the effectiveness of aggressive static optimizations, and compare it to a preliminary runtime optimization system. While it is difficult to improve the performance of the application using existing compiler optimizations, the application can benefit significantly from a runtime optimization system. This raises the question whether it is more effective to identify and optimize for certain performance critical events at runtime, or to rely entirely on profile-based optimizations in a compiler to achieve the same.

We begin with a brief discussion of the algorithm of BLAST in section 2, followed by the performance profile of BLAST with different queries in section 3. Section 4 discusses the effect and limitations of static and profile-based compiler optimizations. The performance of dynamic optimizations on BLAST is evaluated in section 5. Branch mispredictions are discussed in section 6 and section 7 discusses related work. Conclusion and future work are discussed in section 8.

2. Why BLAST?

BLAST, which stands for Basic Local Alignment Search Tool, is an open-source application developed at the National Center for Biotechnology Information (NCBI). It is the most popular Bio-Informatics application. It has more than a million lines of code and uses a huge database of known DNA and amino acid sequences. It is updated regularly as new sequences are discovered. It demands high computational resources and this demand keeps increasing. BLAST uses a similarity searching heuristic that determines sequences from a database that are most similar to a query sequence. These sequences could be base-pairs or proteins. Similarity measures generally start with a matrix of similarity scores for all possible pairs of residues. Similarity score of two aligned sequences is the sum of similarity scores of each pair of aligned residues. The basic strategy is to find a pair of segments of identical length from two sequences such that extending or shortening both segments will not improve the similarity score of the segment pair. BLAST uses a heuristic that compromises selectivity (number of matched segments) for speed. The detailed algorithm and many other performance enhancements are discussed in [1], [2] and [3].

BLAST runs on a broad set of inputs with different options based on the type of sequence comparison. We performed all experiments using a set of queries containing genes or amino acid sequences. Two databases, *nt* and *nr*, were used in our experiments. The *nt* database is a nucleotide sequence database and the *nr* database is non-redundant protein sequence database. A summary of various modes along with the databases used is detailed in Table 1. A set of gene sequence queries are contained in files *nt.1* and *nt.10*. *nt.10* has 10 such queries while *nt.1* has

only one query. Similarly amino acid sequences are contained in files aa.1 and aa.10. Each mode in BLAST uses a particular combination of query and database type which is also described in the table. To refer to the 5th query in file *nt.10*, for example, we will write it as *nt.10(5)*.

Modes	Database	Queries	Sub-Queries
blastn – compares gene sequences	nt (May 28, 2003)	nt.1	None
		nt.10	10
blastp – compares amino acid sequences	nr (June 29, 2003)	aa.1	None
		aa.10	10
blastx – compares translated genes with amino acids	nr (June 29, 2003)	nt.1	None
		nt.10	10
tblastn – compares translated amino acids with gene sequences	nt (May 28, 2003)	aa.1	None
		aa.10	10

Table 1: BLAST execution modes with databases used and queries tested.

3. Performance Profile of BLAST

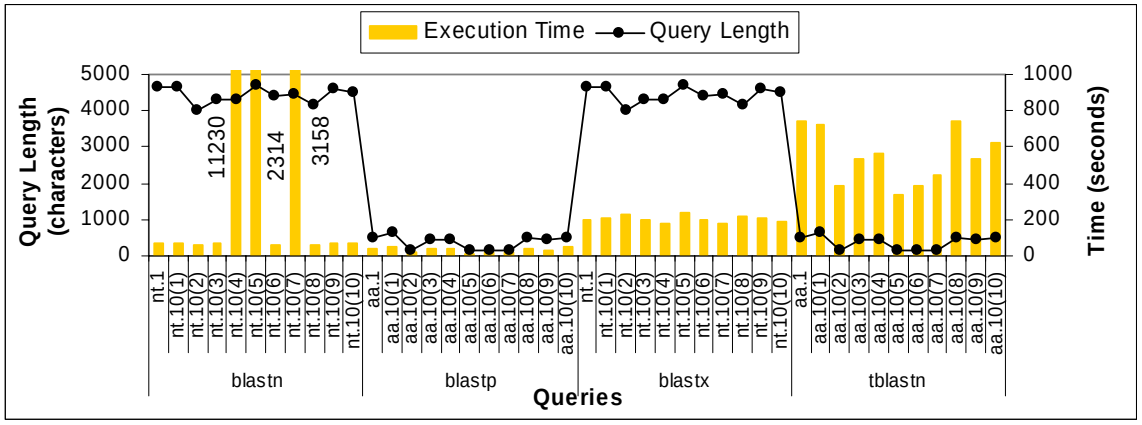

Figure 3: Relation between query length in characters and execution time for all queries in BLAST.

In general, the execution time of a BLAST query depends on two factors: (1) Query length and (2) Query Content. Figure 3 shows that for some cases query length does not necessarily affect execution time. This is accentuated by *nt.1* and *nt.10(4)* queries in *blastn* mode. Both queries have almost the same query length and one query runs for a minute and the other for about 3 hours. To understand the reason for this behavior we collected function profiles and cycle breakdown information for all queries and modes listed in Table 1. All data in this paper was

collected on HP zx6000 workstations (dual Itanium-2 processors) running Enterprise Redhat Linux 2.4.18.

Function Name	blastn nt.1	blastn nt.10(5)	blastp aa.10(1)	blastx nt.10(6)	Tblastn aa.10(9)
BlastNtWordFinder	77.42%	2.33%			
BlastNtWordExtend	11.73%				
ALIGN_packed_nucl	2.96%	4.62%			
RealBlastGetGappedAlignmentTraceback		65.40%			
BlastCheckHSPInclusion		21.44%			
BlastWordFinder_mh_contig			54.03%	52.20%	66.93%
SEMI_G_ALIGN_EX			23.29%	25.63%	7.82%
BlastWordExtend_prelim			17.66%	18.39%	4.53%
BlastTranslateUnambiguousSequence					18.75%

Table 2: Top function profile for selected queries. Only the most dominant functions along with the percentage of time spent in those functions are listed. Cells that are blank represent functions that are not dominant for that query.

Top function profiles and cycle breakdown data were obtained from Intel's ORC (Open Research Compiler v2.1) compiled binaries at O2 optimization level. Top function profile in Table 2 shows that for different queries and modes of execution a different profile is obtained. Not all queries are listed in this table, but only the ones that differ from one another. Other queries show function profiles very similar to some of these profiles.

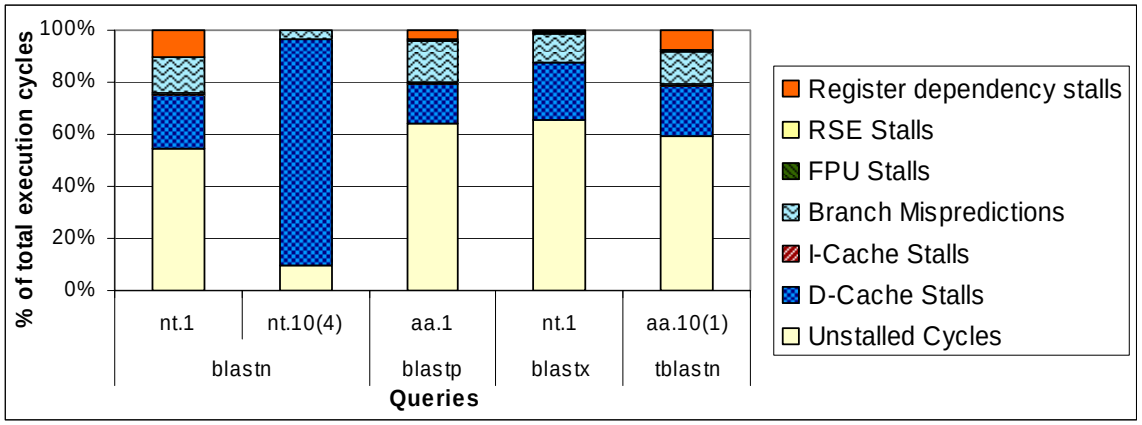

Figure 4: Cycle breakdown for selected queries from every mode. Queries not shown have a cycle breakdown similar to one of these queries in the same mode.

Figure 4 shows the cycle breakdown for some queries obtained using hardware performance monitors. For the *blastn* mode, the 4th nt.10 query stalls for more than 90% of the execution time, and almost all of it due to data cache misses. All of the long running queries in *blastn* have a

cycle breakdown similar to this. Other queries do not have such a high percentage of data cache stalls. They are spread among data cache stalls, branch mispredictions and register dependency stalls. Execution profiles of BLAST queries show the dynamic behavior of this application under different inputs. Since execution time is content dependent many other queries not in this evaluation could show a different execution profile. It is also clear that the major performance bottlenecks are data cache stalls and branch misprediction stalls.

4. Effect of Static Optimizations

In order to understand and evaluate the performance of aggressive static optimizations, we tried to apply optimizations at varying levels to see their effect on performance. We compiled BLAST with various optimizations levels, using three different compilers. The compilers used are the GNU C compiler (GCC v2.96), Electron C compiler (ECC v7.1) and the Open Research Compiler (ORC v2.1). Our goal was to identify a compiler and an optimization level that resulted in the fastest binary and eliminated most of the bottlenecks seen in the previous section.

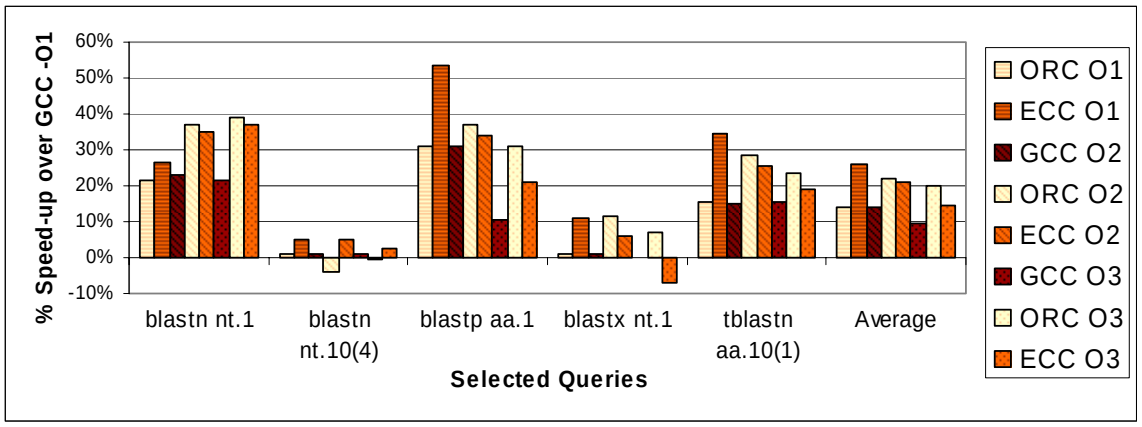

Figure 5: Effect of static optimizations on various modes of BLAST across selected queries. The bars represent speed-up over GCC O1 optimization. Same compilers with different optimizations have the same color. Different compilers with the same optimization level are placed together.

Figure 5 shows the speed-up of three optimization levels for the three different compilers with GCC O1 as the baseline for selected queries. For ORC, O2 optimization has the best performance across most BLAST modes and for most queries. ORC O3 performs only slightly worse than

ORC O2. GCC also shows similar results. However, ECC O3 has a significant slowdown as compared to ECC O2. For some queries, ECC O3 is around 12% slower than ECC O1. Further investigation revealed that ECC aggressively optimizes programs at higher levels of optimizations, so much so that performance is hampered in many cases. In one case, we found that ECC inserts prefetch instructions in a hot loop where prefetching is not needed. We discuss the effect of redundant prefetch instructions in section 4. Queries 4, 5, and 7 in nt.10 in the *blastn* mode (only *nt.10(4)* is shown, *nt.10(5)* and *nt.10(7)* are similar) do not show much performance improvement. These queries are dominated by data cache stalls. The three compilers used could not issue the needed prefetches, so they were not able to optimize these queries. We will discuss more about the performance characteristics of these queries in section 5.2.

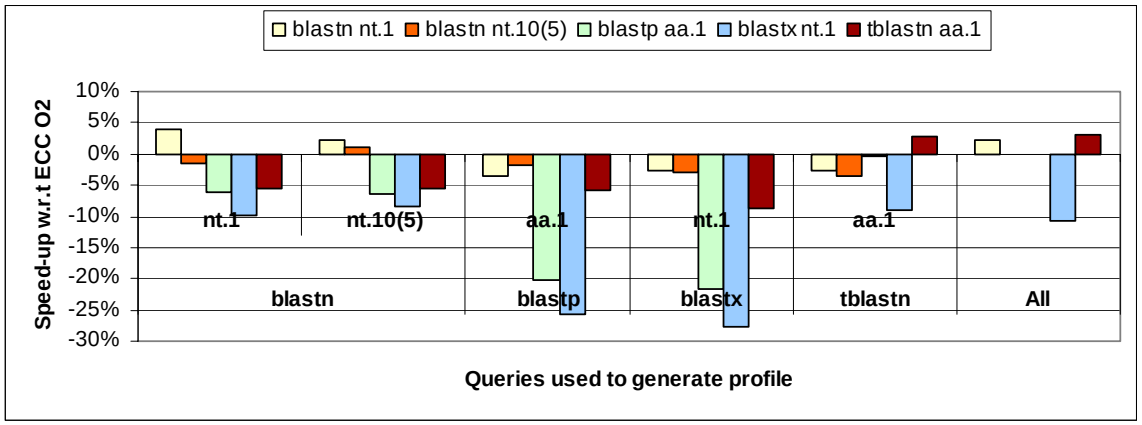

Figure 6: Performance of profile based optimization using ECC 8. The X-axis shows the queries that were used as profiles to generate profile trained binaries. The bars show the speed-up of various queries on profile trained binaries.

Since BLAST shows dynamic program behavior we tried profile-guided optimizations (PGO) using GCC, ORC and ECC to see if profiling can identify and eliminate major performance bottlenecks. To test the performance of profiling, we collected a profile for each query and applied them individually and also by combining all profiles. When a compilation failed, we reduced the optimization levels for failed modules to obtain a profile-trained binary. Despite our best efforts we were unable to generate an instrumented binary using GCC v3.4.0 (as this version supports profiling) ORC and ECC v7.1. However, we were successful in generating profile-

trained binaries using ECC v8, so we present the results of profile based optimizations using this version of ECC. Figure 6 shows minor speed-up in some cases (*blastn, nt.1*) while others suffer significant slowdown. Even when the binary is trained with the same input, the *aa.1* query on *blastp* mode slows down significantly. A 100 fold increase in the system time caused by data TLB misses, which increased from several hundred misses to more than 2 million misses, resulted in the slowdown. This increase was caused by a speculative load that generated a software exception on a TLB miss. This is the default behavior of Linux on Itanium-2 platform.

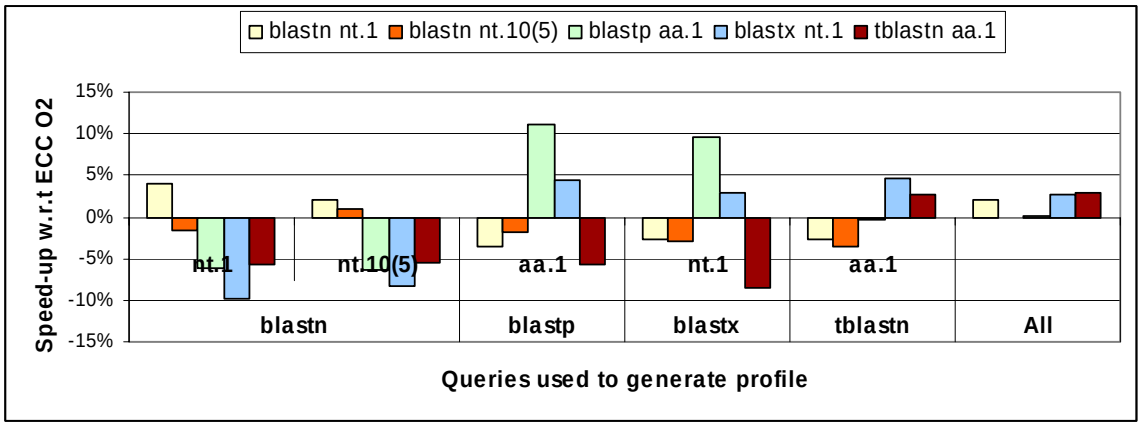

Figure 7: Speed-up obtained from PGO after reconfiguring the kernel to defer TLB miss exceptions on speculative loads to hardware.

When we reconfigured the kernel to defer TLB misses on speculative loads, system time decreased to normal values and gave a speed-up of 11%. Figure 7 shows the performance of PGO using the reconfigured kernel. Figure 6 also shows that binaries trained using a single profile have a negative impact on performance of other queries. Even when the profiles are combined we see that moderate performance gains. These results indicate that the existence of a representative input is essential for profile-guided optimizations. For real world applications, such an input may be difficult to generate. We also see that in an effort to increase performance, profile-guided optimizations use aggressive speculative operations that might have unexpected secondary effects (such as speculation failure), reducing performance.

4.1. Effect of Inserting Prefetches

In this section, we examine the negative effect of prefetches inserted imprudently by a static compiler. At high optimization levels, compilers become very aggressive in applying optimizations such as data prefetching. However, excessive use of prefetch instructions may add extra cycles in loops. It may also replace active data from the cache. There are also some subtle micro-architectural effects that one must take into consideration when issuing prefetches. For example, the Itanium-2 processor's L2 cache handles many demand loads and prefetch operations simultaneously in a non-blocking fashion by using a queue. Issuing excessive prefetch operations could fill up this queue and delay demand loads from being issued to the L2 cache. Earlier, we saw that ECC O3 is slower than ECC O1. For a query in the *blastx* mode, we found that a prefetch instruction is added in a loop with an average trip count of 10. Data for 64 iterations in the future was fetched, so the prefetched data was infrequently used. We found that micro-architectural bottlenecks also contribute to the slowdown. We used performance counters to quantify the slowdown due to micro-architectural bottlenecks. Figure 8 shows a stall-cycle breakdown of this query using ECC O1 and O3 optimizations and a hand optimized O3 binary.

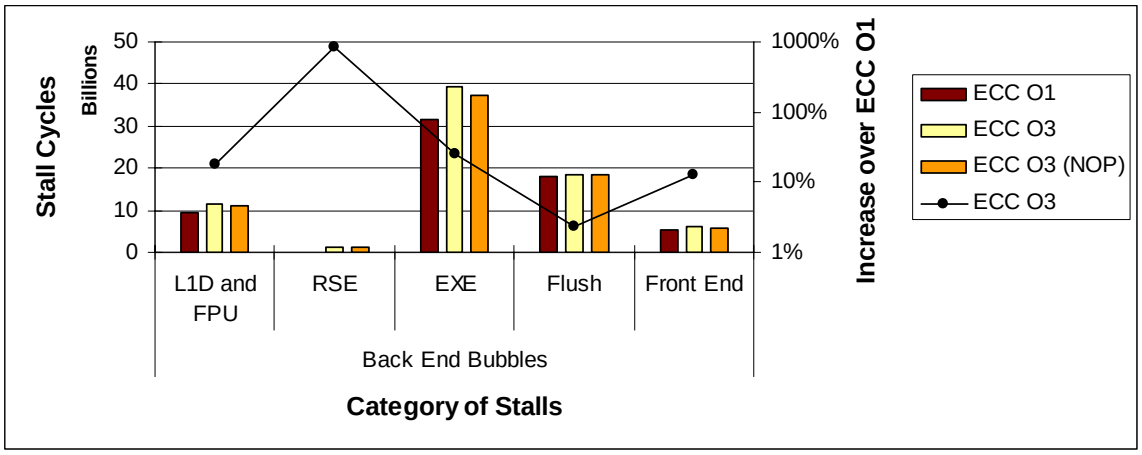

Figure 8: Breakdown of stalls for ECC O1 and O3 and for a binary with the useless prefetch instruction converted to a NOP called ECC O3 (NOP). The line indicates the percentage increase of counters in ECC O3 over O1 (this scale is logarithmic).

Two observations can be made:

1. The number of stalls due to the register stack engine (RSE) has gone up by more than 800%. Register stack engine stalls correspond to an increased register pressure. This is expected as aggressive optimizations tend to use more registers due to larger basic blocks and from speculative and prefetch operations. However, it is a small percentage of the total stall cycles.
2. A large increase is seen in the number of stall cycles in the execution unit (EXE). This was due to an increase in the latency of loads causing an increase in the execution unit stalls.

Load Operations	Measures	ECC O1	ECC O3	ECC O3 (NOP)
Prefetched Load	Total Latency	519,209	540,465	594,354
	Average Latency	16.74	17.02	18.08
	Frequency of Samples	16.76%	16.57%	20.52%
Another Load	Total Latency	559,195	784,085	587,684
	Average Latency	12.16	12.37	12.24
	Frequency of Samples	24.80%	33.03%	25.38%

Table 3: Information about two loads in a hot loop collected by sampling on data cache misses for a query in blastx mode. Note that total latency is the sum of individual latencies of samples. Actual "Total Latency" for this load would be much higher. The unit of latency is a clock cycle.

We sampled performance counters to determine latencies of individual loads. Table 3 lists some statistics collected for loads in the hot loop. ECC at O3 issues a prefetch for a load in the hot loop. Due to the prefetch, another load in the same trace now has a higher total latency and also has a higher frequency of occurrence in the samples collected, which implies a higher miss ratio for this load. The load that is prefetched has no improvement in the total latency. On removing the prefetch, total latency for this load does not change by much. However, the second load's total latency goes down significantly, close to that in ECC O1. This indicates that either cache pollution has occurred due to the useless prefetch or that some micro-architectural bottleneck occurs due to the extra memory operations. This argues for the fact that the compiler cannot always make intelligent decisions about the behavior of loads, and aggressive optimizations may lead to slowdown.

Static optimizations generate a single optimized binary to target performance bottlenecks across different inputs and runtime conditions. If the inputs show conflicting behavior across

inputs, or certain runtime constraints limit the effectiveness of optimizations, static and profile-guided optimizations may negatively impact performance. In the face of such restrictions in compiler optimizations, we tested performance of BLAST on a runtime optimization framework, called ADORE, which is discussed next.

5. Effect of Dynamic Optimizations

Dynamic optimization is a technique that deploys optimizations based on runtime characteristics of programs. Many applications show dynamic phase behavior. For example, a load may hit in the cache during one phase and miss in the other. It is difficult to optimize such load operations using static or profile-guided optimizations. Dynamic optimization is a viable solution as it can detect such phase changes and tweak optimizations on phase changes. Dynamic optimization can also compliment profile guided optimizations, by modifying the deployed optimizations to adapt to the current runtime environment. For example, if the dynamic optimizer detects an increased number of TLB misses or data speculation failures due to speculative loads, it can decide to convert these speculative loads to standard loads to reduce TLB misses or the cost of recovery code. Since a runtime optimizer deploys optimization strategies based on the current input, it can effectively address code regions not optimized by PGO. This section details our experience using a recently proposed dynamic optimization framework called ADORE [13][14][15]. We discuss the system briefly followed by an explanation of its effect on performance.

5.1. The ADORE Framework

ADORE (ADaptive Object-code REoptimization) is a completely transparent dynamic optimization system that optimizes programs in 4 phases: (a) *Profiling* (b) *Phase Detection* (c) *Optimization* and (d) *Deployment*. Profiling involves collecting runtime characteristics of a program to determine a *stable phase* (a portion of execution that has stable characteristics for the

purpose of prefetching) and to provide information about loads that frequently miss in the cache. In the case of prefetching, a stable phase occurs when the program executes almost the same code with a relatively stable CPI and data cache miss rate. The Itanium architecture [16] provides various counters for analyzing performance. These counters can be sampled periodically to get the latest performance characteristics of the program. Profiling is carried out by monitoring the Linux *perfmon* [18] interface in another thread attached to the original program. Samples of performance events are collected from this interface and stored in a buffer. Every 1 second this buffer is analyzed to decide if the main program incurs a phase change. A phase detector is invoked periodically that calculates the CPI, the data-cache misses per instruction (DPI) and the average value of the program counter. Comparing these values with values from the earlier computations, a phase change can be detected. Along with this measure, each sample provides information to build a single-entry multiple-exit unit of code called *trace* on which optimizations are performed. This information includes the current program counter and a history of the last four taken branches from which an execution path leading to the sampled bottleneck can be formed. Many traces can be formed in a stable phase with each trace being optimized by the trace optimizer. Currently, ADORE does data prefetching for three types of references (1) direct array (2) indirect array and (3) pointer chasing. For direct array references, a stride is calculated and prefetch instructions are generated several iterations ahead. The Itanium architecture provides an instruction called *lfetch* for issuing such prefetches. It is a non-binding and non-faulting load, so this optimization is architecture-state safe. Prefetching for pointer chasing is initiated using an approach similar to induction pointers. Once the collected traces are optimized, the memory image of the binary must be patched to execute these traces instead of the original code. This is achieved by modifying the bundle at the beginning of the trace, so it can jump to the optimized trace. This modification is also recorded by ADORE so that it can undo the optimization, if needed.

5.2. Performance of ADORE

Figure 9 shows the performance of ADORE on various queries. Binaries compiled at –O2 are used as baseline comparison because, they have the best overall performance. The graphs show some variance in speed-up on binaries compiled using different compilers due to the differences in code generation and optimization strategies of these compilers. Some queries benefit by a significant amount from ADORE while others suffer slowdown of about 0.7-3.5%. The queries that are optimized are the long running queries whose stalls cycles contribute a large percentage to the total cycles. One of the long running *blastn* queries (*nt.10(4)*) compiled by ECC runs as much as 58% faster. An immediate advantage of such a runtime optimization system is that only one implementation of various optimizations works for binaries compiled with different compilers. Due to space restriction we will discuss performance of ADORE on ORC compiled binaries only.

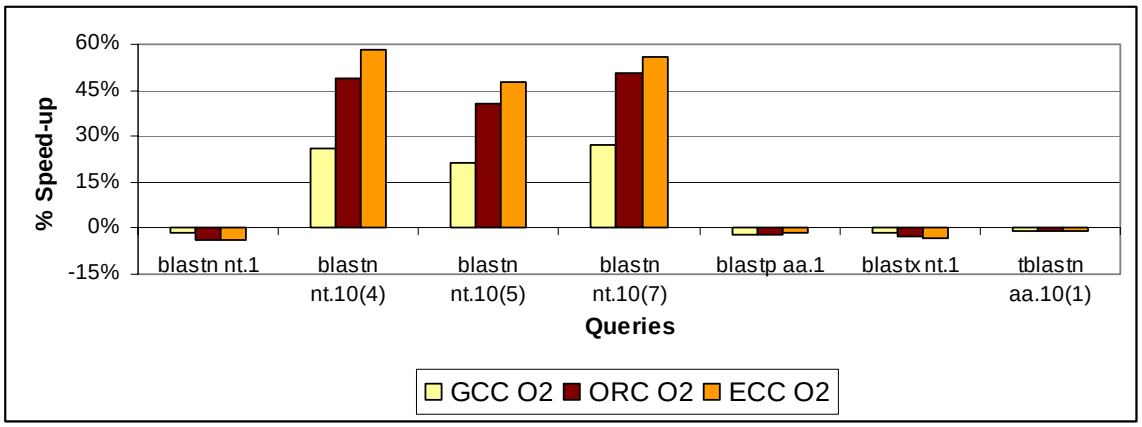

Figure 9: The speed-up from applying ADORE to GCC, ORC and ECC compiled binaries. All binaries are compiled at –O2 optimization. The speed-up is with respect to the O2 binary from the same compiler without runtime optimizations. Not all queries are shown here and those that are not shown suffer a minor slowdown similar to a query from the same mode that has a slowdown.

To see why ADORE speeds up some queries and not the others, we plotted the running CPI ($CPI_{original}$) for each of these queries. We also calculated $CPI_{unstalled}$ by subtracting the stall cycles from the execution cycles to give the actual number of cycles the program needs in the absence of all bottlenecks. Since $CPI_{unstalled}$ is calculated on each interval for which the actual CPI is

calculated, $CPI_{unstalled}$ is as long as the CPI. $CPI_{unstalled}$ shows the maximum performance potential of a particular interval. Thus $CPI_{unstalled}$ in a way represents the inherent ILP of the program for this architecture. We also plot the CPI after applying runtime optimizations (CPI_{ADORE}). Figure 10 shows a plot for a long running query in *blastn* mode that was sped up. The CPI starts off very close to the ideal or unstalled CPI. Then execution moves to a different function and the CPI goes above 3. However, the $CPI_{unstalled}$ remains at around 0.273. From the stall cycle breakdown, we can conclude that such a large departure from the idealized CPI is due to data cache misses.

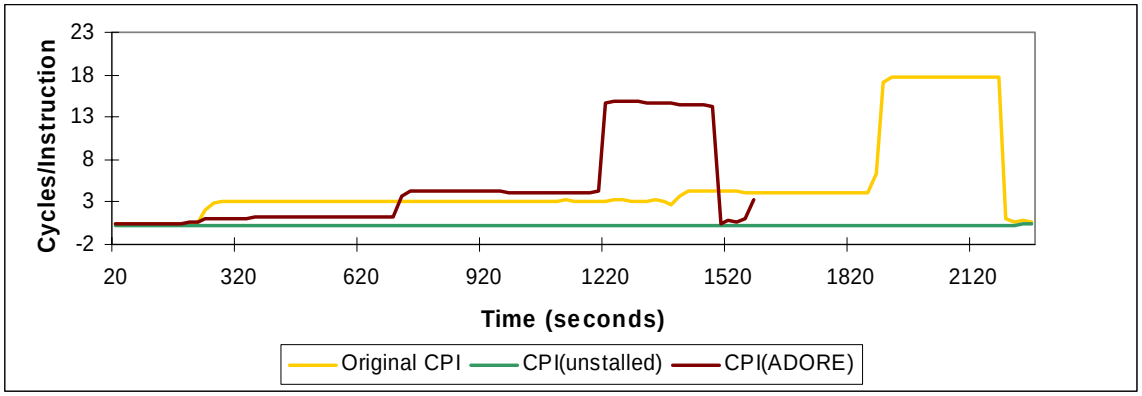

Figure 10: Plot of running CPI calculated by sampling performance counters periodically. CPI_{ADORE} is shorter than original CPI due to speed-up from ADORE.

```
for (index=0; index<hspcnt; index++) {
    hsp = hsp_array[index];
    for (index1=0; index1<index; index1++) {
        hsp1 = hsp_array[index1];
        if(... hsp1->query.offset ...)
```

(a) Indirect Loads

```
for (index=0; index<new_hspcnt; index++) {
    for (seqalign_var=*head;
        seqalign_var->next != NULL;) {
            seqalign_var = seqalign_var->next;
```

(b) Pointer Chasing

Table 4: Part (a) shows code that is optimized by prefetching indirect loads and part (b) shows pointer chasing code that is prefetched using an approach similar to induction pointers.

Table 4 shows code snippets in which ADORE was successful using data prefetching. Table 4(a) is the code that causes a CPI of around 3 and this is due to frequent misses in the indirect load *hsp1->query.offset*. ADORE initiated a prefetch for this load that decreases CPI to around 1.23. In the next phase change, CPI increased to 4 and the bottleneck again was data cache misses due to indirect loads. However, ADORE was not able to prefetch this load as the branch in the trace was unbiased. This happens on low trip counts, and prefetching, if initiated, would not have

been useful. In the presence of unbiased paths and early exiting loops, ADORE does not apply prefetching to avoid bringing useless data into the cache. In the next phase, the program executes a pointer-chasing loop shown in Table 4(b) that causes very high latency cache misses. ADORE assumes that a stride exists between pointers, and uses the difference between successive addresses to prefetch for future iterations. Instrumenting this loop revealed that strides exist for some of the accessed data. Inconsistent stride coupled with very high latency caused the CPI to remain considerably higher than $CPI_{unstalled}$ but lower than the original CPI.

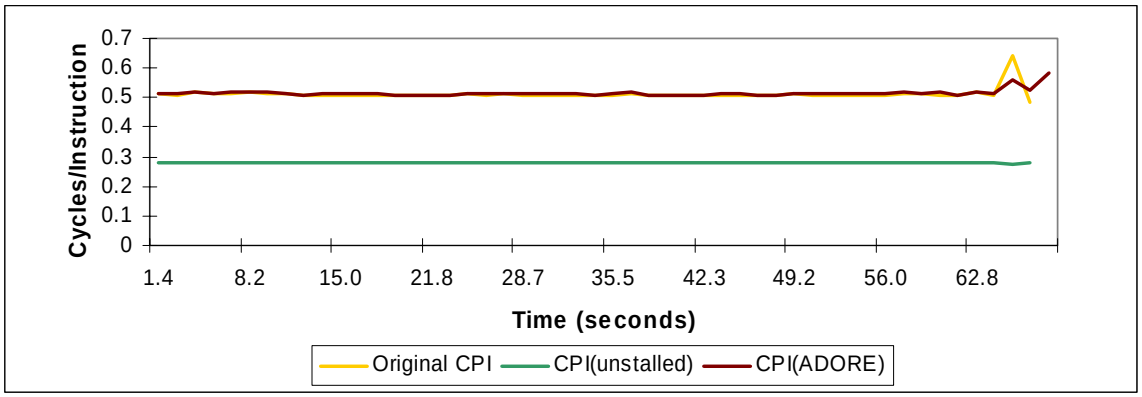

Figure 11: Plot of running CPI for a short running *blastn* query. The CPI plots for other modes and queries that do not speed up have CPI plots closely following this plot.

Figure 11 shows the plot of CPI for a short running query in *blastn* mode. The running CPI (around 0.51) hardly changes and is close to $CPI_{unstalled}$ (around 0.28). There are no significant data-cache bottlenecks in this program. ADORE recognizes this fact and does not try to optimize the code. An attempt at prefetching for such queries could have increased the runtime due to reasons we discussed in section 4. For queries that are not optimized, ADORE adds only a minimum amount of overhead, which is the overhead of sampling and phase detection.

Thus, ADORE dynamically detects dominating performance bottlenecks and deploys optimizations to the target code. As seen earlier, current implementation of PGO cannot take advantage of such opportunities due to lack of cache profile. The optimizations that ADORE deploys can also be deployed statically. However, there is a tradeoff of *cost of optimization* versus *benefit of optimization*. This forces a compiler to deploy such optimizations only if it is

sure that the optimization would benefit the application across all inputs. Furthermore, the compiler may not know the micro-architecture on which the optimized binary will execute. This complicates the decision as the bottleneck may be micro-architecture specific.

6. Optimizing Branch Mispredictions

Much research has been done on improving branch prediction using hardware schemes, but few software schemes have been developed for adapting to dynamic branch behavior [19]. Predication can be used to eliminate branch mispredictions, but it is difficult to use predication with while loops. The cycle breakdown of BLAST (Figure 4) shows that, for some queries, there are a significant number of stalls due to branch mispredictions. To find out the reason for these mispredictions, we simulated an idealized hardware for branch prediction using an instrumentation tool called PIN [17]. Itanium-2 stores branch prediction information associated with each instruction cache line. It uses a 4-bit local history to select a 2-bit saturating counter from a 16K-entry pattern history table. These are the structures used for direct branches that represent almost all types of dynamic branch instructions found in BLAST. The simulated hardware stores m-bit saturation counter for all n-bit local patterns for a particular branch. We used the query *nt.1* for the *blastn* mode as this query has a significant amount of branch mispredictions. Using performance counters and simulation, we found that a significant portion of branch mispredictions occur due to one branch. In most cases when the branch is mispredicted, it is *not-taken* while the counters are saturated at *taken*. We found that increasing branch history does not give any improvement. Even with a 10-bit local history, branch prediction did not improve. From the results of simulations, we concluded that the pattern of branch outcomes is such, that it is difficult to predict the *not-taken* branches using branch histories.

An optimizer can do better branch prediction, if the prediction is based on the data that determines branch outcome, rather than on branch history. Since branch prediction is done by the front-end instruction fetch engine and the data that decides branch outcome is computed in the

back-end, there should be a sufficient distance between the data and the dependent branch. The prediction mechanism of an indirect branch, in Itanium-2, is based on the assumption that the branch target is the value that is stored in the branch register at the time of instruction fetch. If we convert the direct branch into an indirect branch and compute the target of the branch (fall-through or taken path in the original code) in the branch register early enough for the instruction fetch engine to see the correct branch target, branch mispredictions would be eliminated.

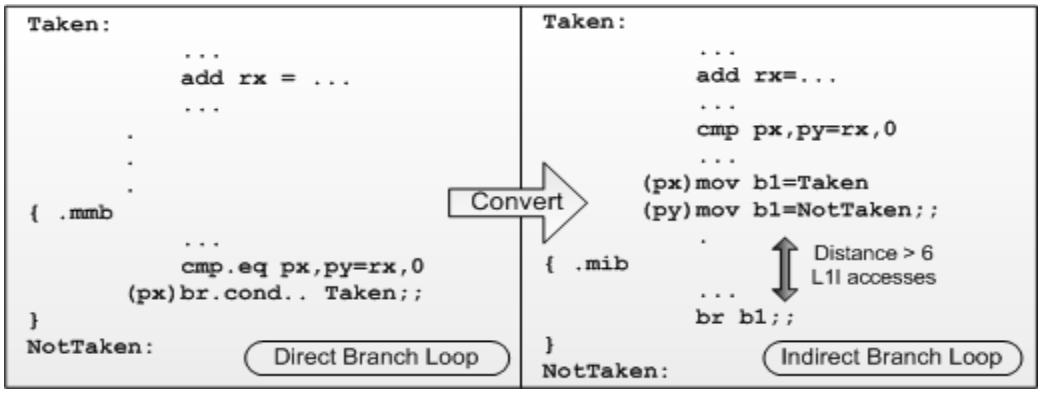

Figure 12: The conversion of direct branch to an indirect branch in a loop for the Itanium-2 processor using pseudo-assembly code.

Figure 12 shows a loop with a conditional direct branch converted to an unconditional indirect branch. The target of the indirect branch is conditionally changed to the *taken* or *not-taken* target. Target address calculation should be a fair distance away (> 6 L1 instruction accesses in Itanium-2) from the branch for the front end to see the correct value of the target register. This is the main constraint in using indirect branches for perfect loop prediction. The benefit of this scheme is that branch prediction will be accurate for branches with unpredictable patterns. In the case for *blastn*, the loop is very tight causing the *mov* instruction to be very close to the branch. Due to this we could not apply this transformation to BLAST. To see if this scheme could be applied to other programs, we analyzed the top five mispredicting branches for programs in the SPECint2000 benchmark suite. We estimated the number of execution cycles between data and dependent branch, by moving the data dependence chain as far away from the branch as possible. Figure 13 shows that dependence chain is very tight, with an average of 2.2 execution cycles between data

and branch, except for eon that has an average distance of 12 execution cycles. A distance of 6 L1I accesses would need at least a distance of 6 execution cycles between data and branch instruction.

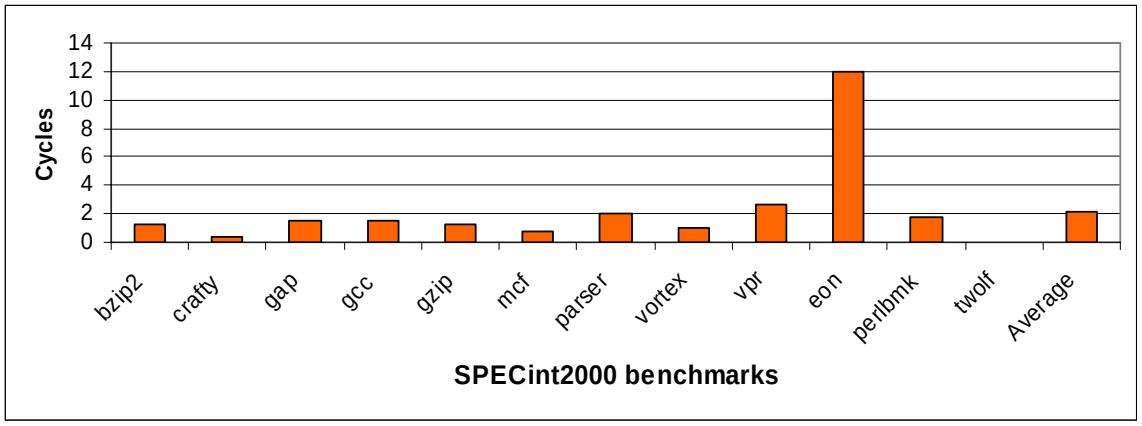

Figure 13: Average number of execution cycles between data and dependent branch for SPECint2000 benchmarks.

Branch misprediction is still a significant bottleneck in real-world applications. The proximity of data and dependent branch reduces the possibility of influencing branch prediction hardware, in time to affect prediction accuracy. However, when such opportunities are available, such as for Eon, a dynamic optimizer may be able to deploy our proposed technique to decrease branch misprediction stalls. As stated earlier, a compiler can deploy this optimization too, but the tradeoff still remains the same. A compiler must ensure that the optimization will provide performance benefit across inputs and across different micro-architectures before applying the optimization.

7. Related Work

The concept of traces is similar to Superblocks proposed by Pohua et Al in [20]. The Superblock aids the application of optimizations such as dead code elimination and some loop optimizations by generating single-entry, multiple-exit code regions. Path profiles are also used to perform new optimizations that improve the locality of instruction cache by re-ordering basic

blocks and splitting procedures. Pettis et al. in [21] discuss profile-guided code positioning and Cohn in [22] discusses Hot-Cold optimizations that make use of path profiles.

Software cache optimization to predict cache misses and issue timely prefetches based on profile information is used by Abraham in [10]. He collected various statistics about loads/stores into a cache profile. This information is used to drive compiler-directed prefetching. Mowry in [23] proposed combining cache profiles with path information to form a correlation profile that provides information about the cache behavior of a load for a specific path. The main idea is that some loads may miss in the cache along certain paths and summarizing this information across all paths can lead to inferior cache profile.

To deal with runtime profile challenges such as inaccurate mapping of profile from an optimized binary to source code, lack of availability of source code (especially for library and legacy code) and necessity of recompilation, applying optimizations post-link was proposed. Post-link time optimizations for inter-procedural dead code elimination and loop invariant optimizations were evaluated in [24]. Luk [25] applied profile information post-link to find strides and initiate prefetching for loads that cannot be statically analyzed for regular patterns.

Reality based optimization [12] applies profiling to real world applications. They collect and combine profiles over time to generate a representative profile. Zilles [26] speculatively pre-computes backward slices of delinquent loads that are identified by profiling. This method is effective if a representative profile is available. The central limitation of profiling is the assumption that runtime characteristics of programs will be insensitive to profiled inputs, which may not be the case for many applications. Dynamic optimization frameworks were proposed as a solution for these problems. Many systems besides ADORE (section 5.1) have been proposed. Dynamo [27] starts running statically compiled executables through interpretation, and generates code fragments into a fragment cache after a set number of code interpretations. Its trace selection policy can lead to improved I-cache performance for benchmarks with stable execution paths. DynamoRIO [29] is an x86 dynamic optimizer based on Dynamo that allows customization of

dynamic tasks through an API. When a hot trace is found, a fragment is formed that is natively executed. Continuous Program Optimization (CPO) [28] compiles an instrumented version of the intermediate binary to continually generate an execution profile of the program. An optimizer runs continuously in the background, using the collected profile information to drive profile based optimization (PBO) in parallel with program execution. Mojo [30] is a dynamic optimization system developed by Microsoft targeting x86 architectures that deploys optimizations through a trace.

8. Conclusion and Future Work

Cache misses and branch mispredictions remain significant bottlenecks in present-day applications. Such performance bottlenecks may move from one region to the other depending on the input data sets. Static and profile-guided optimizations generate one binary and hence are less likely to address these bottlenecks effectively. In this work, we used a real world application, BLAST, which highlights these issues and supports a case for the simplicity and efficacy of runtime optimizations.

In optimizing BLAST, we found that traditional aggressive optimizations do not have a significant impact on performance. BLAST exhibits dynamic behavior for different input sets. Current implementations of profile-guided optimizations could not identify delinquent loads or initiate the needed prefetches, resulting in poor performance. The limiting factor in current profile-guided optimization was the absence of a representative input and other information in the form of cache profiles. We also found that certain optimizations may have secondary effects that are hard to predict at compile time. However, the application of runtime optimization using ADORE was quite simple. It was able to handle a large real-world application and optimize it across different compilers with no change in optimization strategy. It could target critical performance bottlenecks that led to speed-ups of up to 58% for some input sets. Runtime optimization can compliment existing compiler optimizations by deploying optimizations or fine

tuning existing optimizations, to gain better performance. Branch misprediction was found to be a significant bottleneck for some inputs, and we tried to eliminate stalls caused by such mispredictions. We found that the distance between data and dependent branch is generally too small to influence the branch prediction in time. However, the scheme we propose may be used for applications that have a greater distance between data and dependent branch such as Eon.

In future, we plan to study more real world applications, like databases, flow modeling and circuit simulation applications, to understand and evaluate the effectiveness of different runtime optimization techniques. The question we have in mind is "*What optimization strategies are most effective and are relatively simple to apply from the perspective of real world applications, the end user and the developers*"? This work shows that runtime optimization could be successful in reducing stalls for large applications, while existing static optimizations are limited by several factors. With the analysis of more applications, we hope to identify the strengths and weaknesses of static and dynamic optimizations, so that both techniques can be further advanced.

References

[1] Altschul, S.F., Gish, W., Miller, W., Myers, E.W. & Lipman. D.J., Basic local alignment search tool. *J. Mol. Biol*. 215, p.403-410, 1990

[2] Altschul, S.F., Madden, T.L., Schäffer, A.A., Zhang, J., Zhang, Z., Miller, W. & Lipman, D.J. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Res*. 25, p3389-3402, 1997.

[3] Gish, W. & States, D.J. Identification of protein coding regions by database similarity search. *Nature Genet.* 3, p266-272, 1993

[4] Karl Pettis, Robert C. Hansen. Profile guided code positioning. In *PLDI*, p.16-27, June 1990.

[5] A. Ramirez, L. Barroso, K. Gharachorloo, R. Cohn, J. Larriba-Pey, P. G. Lowney, M. Valero. Code layout optimizations for transaction processing workloads. In *ISCA'01*, p.155-164, 2001.

[6] P. P. Chang, W. W. Hwu. Trace selection for compiling large C application programs to microcode. *Proceedings of the 21st annual workshop on Microprogramming and microarchitecture*, p.21-29, 1988.

[7] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, D. M. Lavery. The superblock: an effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, v.7 n.1-2, p.229-248, May 1993

[8] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *MICRO'92*, p.45-54, December 01-04, 1992.

[9] Brad Calder, Peter Feller, Alan Eustace. Value profiling. In *MICRO'97*, p.259-269, December 01-03, 1997.
[10] S. G. Abraham, R. A. Sugumar, D. Windheiser, B. R. Rau, Rajiv Gupta. Predictability of load/store instruction latencies. In *MICRO'93*, p.139-152, December 01-03, 1993.
[11] Todd M. Austin, Gurindar S. Sohi. Dynamic dependency analysis of ordinary programs. In *ISCA'92*, p.342-351, May 19-21, 1992.
[12] Scott McFarling. Reality-based optimization, In *CGO'03*, p.59 – 68, 2003.
[13] J. Lu, H. Chen, P.-C. Yew, W.-C. Hsu. Design and Implementation of a Lightweight Dynamic Optimization System. In *The Journal of Instruction-Level Parallelism*, vol.6, 2004.
[14] H. Chen, et Al. Continuous Adaptive Object-Code Re-optimization Framework. *Ninth Asia-Pacific Computer Systems Architecture Conference*, pp. 241 – 255, 2004.
[15] J. Lu, H. Chen, R. Fu, W.-C. Hsu, B. Othmer, P.-C. Yew. The Performance of Runtime Data Cache Prefetching in a Dynamic Optimization System. In *MICRO'03*, December 2003.
[16] Intel® Itanium® Manuals, http://www.intel.com/design/itanium/manuals.htm
[17] PIN - A tool for software instrumentation of Intel® Itanium® Linux programs, http://www.intel.com/software/products/opensource/tools1/inst/
[18] PerfMon, http://www.hpl.hp.com/research/linux/perfmon/
[19] Cliff Young, Michael D. Smith. Static correlated branch prediction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, v.21 n.5, p.1028-1075, September 1999.
[20] P. P. Chang, S. A. Mahlke, and W. W. Hwu. Using profile information to assist classic code optimizations. *Software-Practice and Experience*, 21(12), p1301-1321, December 1991.
[21] K. Pettis, R. C. Hansen. Profile guided code positioning. In *PLDI'90*, p.16-27, June 1990.
[22] Robert Cohn, P. G. Lowney. Hot cold optimization of large Windows/NT applications. In *MICRO'96*, p.80-89, December 02-04, 1996.
[23] Todd C. Mowry, Chi-Keung Luk. Predicting data cache misses in non-numeric applications through correlation profiling. In *MICRO'97*, p.314-320, December 01-03, 1997.
[24] A. Srivastava, D. W. Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages* 1(1), p. 1-18, March 1993.
[25] C.-K. Luk, R. Muth, H. Patil, R. Weiss, P. G. Lowney, R. Cohn. Profile-guided post-link stride prefetching. In *Proceedings of the 16th international conference on Supercomputing*, p. 167-178, 2002.
[26] C. B. Zilles, G. S. Sohi. Understanding the backward slices of performance degrading instructions, In *ISCA'00*, p.172-181, June 2000.
[27] Vasanth Bala, Evelyn Duesterwald, Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI'00*, p.1-12, June 18-21, 2000.
[28] T. Kistler, M. Franz. Continuous Program Optimization: Design and Evaluation. *IEEE Transaction on Computers*, vol. 50, no. 6, June 2001.
[29] D. Bruening, T. Garnett, S. Amarasinghe. An Infrastructure for Adaptive Dynamic Optimization. In *CGO'03*, 2003.
[30] W.K. Chen, S. Lerner, R. Chaiken, and D. Gillies. Mojo: A dynamic optimization system. In *FDDO-04*, pages 81-90, 2000.