Profile Directed Offline Binary Optimization

Kerry Wong and Weiping Liao

Abstract

This paper describes a framework for profile directed offline binary optimization. Most existing binary optimization systems perform optimizations dynamically. In these systems, program profiles are gathered during the execution of the program and then used to optimize the instruction execution stream dynamically. Optimizing at machine code level at runtime, dynamic binary optimization can exploit far more optimization opportunities than most static optimization methods, and can be adapted to program's changing execution patterns. Since optimization is done at runtime, the time used for optimization is counted as part of the runtime of the program. Due to this fact, we often see programs actually running slower under such dynamic optimization environments. Our framework is intended to eliminate the runtime overhead of dynamic optimization while still exploit program's runtime statistics. Under the framework we propose, a program is first instrumented, program profiles are obtained by running the instrumented program. A faster version of the binary program is then created using the program's runtime information. To obtain more accurate program runtime characteristics (e.g. cache misses), we also incorporate hardware performance counters found on most modern processors to perform instrumentations. We implemented a simple tool that automates the instrumentation and optimization. The current version of our tool consists approximately 2,500 lines of JAVA code and is tested with a 1GHz Pentium III machine running Microsoft Windows 2000. Preliminary experiment results suggest that even with simple optimization methods, the optimized binaries can achieve speedups up to 2%.

I. INTRODUCTION

Recent trends in software and hardware technologies make static compilation harder and harder to improve on performance. The use of object-oriented languages and the dynamic code generation environment like Java virtual machine and dynamic binary translator make the applicability of heavy-weight static compiler optimization techniques almost impractical [1].

Static optimization has other drawbacks. Most static optimizer requires source code to be available. As programs become more and more complicated, it is more and more unlikely that all the source code are available to the programmers. This is especially true for programs developed nowadays. The growing use of dynamic linking libraries excludes a significant portion of source code from being available to the programmers. Even for static libraries, most of the code are proprietary and the source code are hard to find.

As a result, dynamic optimization [1], [2] has gained much focus in computer science community for the past few years. Dynamic optimization can utilize runtime information to optimize programs, and therefore, it can overcome the common obstacles to traditional static compiler analysis methods and improve program performance substantially.

Dynamo [1] is software dynamic optimization system that improves the performance of a native program (binary program) by dynamically optimizing the instruction streams during the execution of the program. It first interprets the instructions until a hot path is found. Then it optimizes the hot path and stores the optimized code in its fragment cache. The next time when the same instruction sequence is encountered, the optimized code can then be loaded from fragment cache and executed directly. By doing this Dynamo can improve program performance dynamically.

IBM Jalapeno [2] performs dynamic optimization when it translates Java bytecode to native machine code. It utilizes ideas that are similar to Dynamo: dynamically monitoring the instruction stream for hot paths, optimizing hot paths and caching them for later reuses. The main difference between the dynamic optimization scheme used in Dynamo and Jalapeno is that in Dynamo, native instructions are interpreted and optimized during an execution session, while in Jalapeno, Java byte code is first translated into an intermediate representation and dynamic optimization is interfaced directly with the intermediate representation.

However, these dynamic optimization methods have a sever common drawback: the optimization time is counted into the runtime overhead. To get benefit from this kind of optimizations, the optimization time has to be short enough, which prevents sophisticated optimization schemes being used. One disadvantage particular to Dynamo is that it has to interpret the native instructions before hot paths can be optimized, the overhead due to interpretation resulted in much slower instruction execution than instructions being executed directly. This in fact, limits the performance of Dynamo to a great extent. In many cases, when Dynamo cannot stay above the break-even point, it has to bail out and resort to the execution of the original binary program.

We believe that constantly monitoring the running binary code as in the dynamic optimization approach is neither efficient nor necessary. In this paper, we propose a different approach for dynamic optimization. Instead of performing optimizations during program's execution time , optimization will be done offline. The key difference between our approach and static offline optimization is that our offline optimization is based on the runtime profiles obtained during the execution of the program. Under this framework, the dynamic optimization overhead becomes invisible to the optimized program.

Since we perform optimizations offline, the profile obtained during a program's execution is an averaged version of the dynamic profile which resembles to a snapshot of a short execution period of the program. This necessarily assumes that the program's internal statistics (e.g. which paths are hot) do not vary much over time and over different runs. The assumption that applications should have rather stable workloads seems to be reasonable and much existing research also provides evidence to this assumption.

To obtain more accurate program runtime characteristics (e.g. cache misses), we also incorporate hardware performance counters found on most modern processors to perform instrumentations.

Our idea is similar to what is done in FX!32 [3], a binary translator which translates Intel X86 instructions to DEC Alpha instructions. In FX!32, some optimized code is stored on harddisk and is loaded when needed. However, there is much difference between FX!32 and our approach. First, we propose to optimize the whole binary executable off-line and store the entire executable containing the optimized code to harddisk, while in FX!32, only parts of the program, called the translated images, are stored on the harddisk. FX!32 does not really modify the original executable, while in our approach, the executable is changed to a new one. Second, we focus on the optimization within one ISA, namely Intel's x86 platform, not the inter-architectural instruction set translation. Fx!32 has to translate Intel x86 instructions to DEC Alpha instructions.

The remaining parts of the paper are organized as follows. Section II gives a high-level overview of how our proposed offline binary optimization methods work. Section III discusses the profiling phase in detail. Section IV talks about the optimization phase. Section V shows some preliminary results based on our tools. Some issues still need to be further explored are covered in Section VI and Section VII concludes.

II. Overview

We chose Intel x86 binary as our target binary format. Our tools are tested on an 1GHz Intel Pentium III machine. Due to the difficulty to translate and optimize x86 binary code directly, we selected the Intel compiler[1] generated X86 assembly code as our target. Instrumentation and optimization are done at the assembly level. The advantage to use assembly code is that, first, we can avoid the trouble of decompiling and decoding binaries, which can be a difficult and tedious task for x86 architecture. Given the time constraint of this project, decompiling x86 binaries would take too much of our time. Using assembly code, we can easily find basic blocks and construct the control flow diagrams; second, we do not have to worry about the branch/jump offsets since target labels are just symbols in assembly code. If we move a basic block for a different code layout, we only need to keep the labels consistent. And finally, the layout of assembly code will closely match that of the machine code. This is extremely important since we need the effects of optimization and instrumentation be accurately reflected in the binary code. Other representations (e.g. high-level language, or Java bytecode) do not provide this luxury, at least it would be extremely difficult to match the machine code to that in the source program.

Using assembly code for our experiments means that we need to write our own parser. Given the relatively simple assembly code format Intel compiler provides, our task becomes much easier. The parser we use is able to parse an assembly file generated from a C++/C source file by the Intel compiler and build an abstract syntax tree accordingly. The parser provides a pleasant interface for adding, removing and rearranging assembly procedures, basic blocks, instructions and definitions.

As discussed briefly in Section I, our proposed profile directed binary optimization consists

[1] Version 4.5

two phases—profiling phase and optimization phase. We discuss these two phases in detail here.

1. **Program Profiling Phase**. During this phase, instrumentation points are added to the target Intel C++ Compiler generated assembly file that needs to be optimized. This step is done offline. Instrumentation is based on the required profiles that need to be obtained (e.g. path profiles used to identify hot paths). The instrumented program is then run with typical working sets (it is possible to feed multiple inputs to the instrumented program and gather profile information over different runs and then use the combined profile to optimize the program. So far we have only implemented optimization based on a single profile.) and the program profile is gathered (e.g. dumped to a file).

2. **Binary Code Optimization Phase**. We then use the program's dynamic profile to optimize the original target program offline. The optimized version of the binary code is stored on the disk for future use. Since program profile is gathered in a separate phase, there is no need for the new binary file to contain any instructions for instrumentations, overheads due to instrumentation can be totally eliminated.

The performance of the optimized binary program is then compared to that of the original program. Figure 1 shows the overview of our proposed platform.

The advantage of our approach is that there is no run-time optimization overhead, so sophisticated optimization methods can be implemented. Such expensive optimization methods cannot be easily implemented in a traditional dynamic optimizer without an efficient cost-benefit model. Further, because no profiling is required every time the targeted program runs (assume that the working set is stable), the profiling overhead is insignificant.

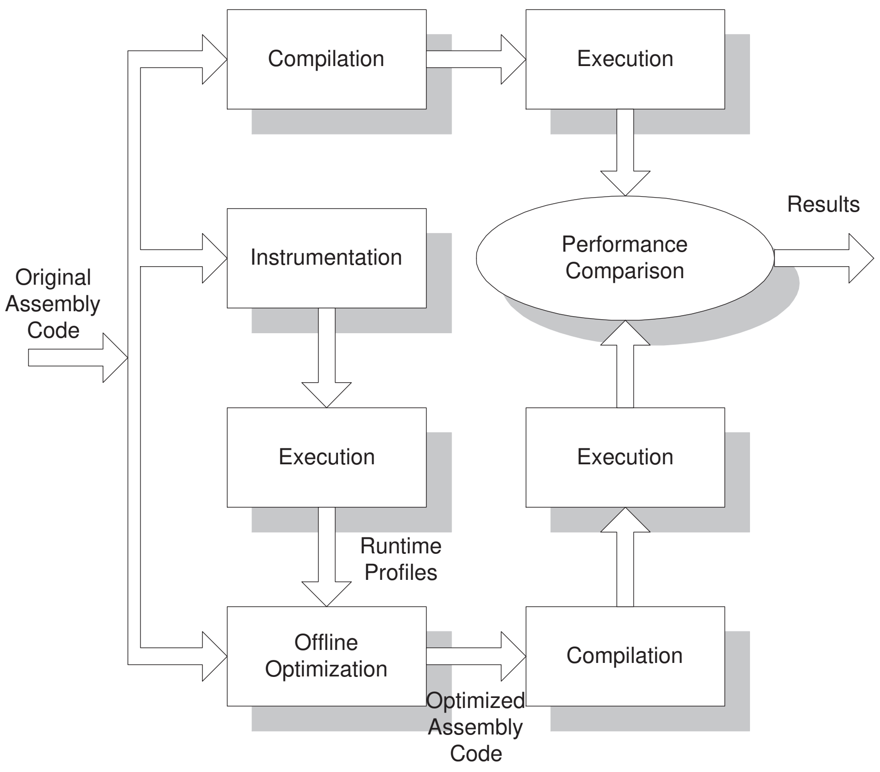

Fig. 1. Overview of the platform

III. Profiling Phase

In our approach, profiling is done once using typical inputs. Our profiling methods include two categories. The first category includes using the instrumentation counters we created and inserted to the program. These counters count the frequencies of each block or edge, or path, so that we can locate hot spots in the program. Section III-A discusses the implemented profiling routines in detail. The other category includes using the performance counters provided by hardware. Various hardware related performance information can be obtained through these performance counters, Section III-B gives more detail on these hardware performance counters.

A. Implemented Profiling Routines

Our current implementation allows one kind of profiling methods to be used at a time (e.g. path profiling). It should not be difficult to extend our current program to simultaneously provide multiple types of profiles. We have so far implemented block, edge and path profiling. Programs can be instrumented to obtain the above three kinds of dynamic profile separately.

Here is a simple example illustrating how instrumentation instructions are inserted into the target assembly code. Suppose the user wants to instrument a program with block profile routine. The parser first scans through the Intel compiler generated assembly code and build the AST. During the first path, the labels and total number of eligible basic blocks[2] are recorded. The AST is then traversed for a second time. During this traversal, instructions to allocate counter memory, increment counters and output the dynamic contents of counters are added. The counter number and its corresponding basic block label are paired up and output to screen so that the future profile results can be related to the instrumented assembly code. Figure 2 shows a piece of assembly code taken from one of our benchmark programs (`prime`), with instrumentation instructions added. The bolded lines ending with ++`AsmOPT`++ are the new code added by our profiling tool to the original assembly code.

In this example, 6 blocks are instrumented. Counters are incremented at the beginning of an eligible basic block (see block `$B1$13` for example). To obtain the counter values at runtime, a print counter routine is added to the last basic block of the original program. Note that a new block `$PrintCounter` is added to initialize the loop. The print counter routine works by walking through all the elements in the counter array and outputting their values to screen.

[2]Most of the basic blocks in a Intel C++ Compiler generated assembly file end either with a jump or a call with only a few exceptions. Our current work only consider intra-procedure jumps so basic blocks that end with a jump instruction are eligible for instrumentations. There are a total of 38 conditional/unconditional jump instructions on Pentium III, and we consider all of these jumps.

```
_DATA  SEGMENT PARA PUBLIC USE32 'DATA'
     __String__PF_BBFreq_Counter DB "counter %d = %d",10,0  ; **AsmOpt**
     _PF_BBFreq_Counter DD 6 DUP(0) ;  **AsmOpt**
     ALIGN 010H
_DATA  ENDS

. . .

_main     PROC NEAR

. . .

$B1$13:
     inc _PF_BBFreq_Counter[5*4] ;  ++AsmOpt++
     mov eax, DWORD PTR [ebp-60]
     add eax, 2
     mov DWORD PTR [ebp-60], eax
     mov eax, DWORD PTR [ebp-60]
     cmp eax, 10000
     jl $B1$3
$PrintCounter:
     xor ecx, ecx ; ++AmsOpt++
$B1$14:
     push ecx ;      ++AmsOpt++
     add esp, -16 ; ++AsmOpt++
     mov DWORD PTR [esp], OFFSET FLAT: __String__PF_BBFreq_Counter ; ++AsmOpt++
     mov eax, ecx;   ++AsmOpt++
     mov DWORD PTR [esp+4], eax ;   ++AsmOpt++
     mov eax, DWORD PTR _PF_BBFreq_Counter[ecx*4] ;  ++AsmOpt++
     mov DWORD PTR [esp+8], ea x ; ++AsmOpt++
     call _printf;   ++AsmOpt++
     add esp, 16;    ++AsmOpt++
     pop ecx ;       ++AmsOpt++
     inc ecx ;       ++AmsOpt++
     cmp ecx, 6 ;    ++AsmOpt++
     jne $B1$14 ;    ++AsmOpt++
     mov esp, ebp;   ++AsmOpt++
     pop ebp ;       ++AsmOpt++
     ret ;           ++AsmOpt++
     ALIGN 4;        ++AsmOpt++
_main ENDP
```

Fig. 2. Sample instrumentation code

The implementation of edge profiling is similar. We used the method proposed by Ball and Larus [4] for efficient path profiling. In path profiling, the counter array element is indexed by an unique path number generated by instrumentation code in blocks along each path and the corresponding array element incremented at the end of each path. For edge and path profiling, the correspondence between counter numbers and edge/path labels (the basic blocks along an edge or a path) are also outputted during the instrumentation phase for future runtime reference.

B. x86 Performance Counters and Prefetching Mechanisms

In our implementation, we also explore the benefit of incorporating information obtained from hardware performance counters to facilitate optimization. Traditional dynamic binary optimization methods simply pick hot paths for optimization. The hotness of a path can be obtained from path profile counters. See Ball and Larus's efficient path profiling method [4] for more details. Since in a fairly large program there will be potentially many hot paths, most existing method would optimize most or all the hot paths found by criteria of a threshold. We believe that it is neither necessary or efficient to optimize all the most frequently executed hot paths found in a program. It is not necessary since a program might just spend a very small proportion of its execution time within certain hot paths, the speed gain of optimization these hot paths might be negligible. It might not be efficient since optimization leads to memory footprint expansion, which in turn could cause more cache misses and slow down the optimized program. These issues were not well addressed in the existing literature. We explore these issues here.

Most modern Intel chips (starting from Pentium MMX) provide performance related events monitoring instructions that can be used to monitor the dynamic execution of programs. The inclusion of these performance counters is intended to facilitate understanding detailed program behaviors, finding out bottlenecks that limit the program's potentials, and fine-tuning applications based on the performance data obtained via performance counters. We shall briefly touches upon some details related to Intel performance counter and the prefetching mechanisms in x86 that we exploit in our offline binary optimization tool.

Intel Pentium processor provides two instructions that can be used to monitor various hardware events. RDTSC (Read time stamp counter) can be used to obtain clock cycles elapsed between two instrumentation points and thus provides accurate means to monitor instruction execution speed. On a 1GHz Pentium III machine, the accuracy of the time stamp counter is one nanosecond. Other crucial hardware events can be obtained via RDPMC (Read processor monitor counter), which reads the performance-monitoring counters. The following information can be obtained by invoking RDPMC. We list a few here. More detailed descriptions can be found in Intel Architecture Optimization Reference Manual.

1. L1 cache misses: this event indicates the number of outstanding L1 cache misses at any particular time.

2. L2 cache misses: this event indicates all data memory traffic that misses the L2 cache. This includes loads, stores, locked reads, etc.

3. L2 cache requests: this event indicates all L2 data memory traffic.

4. Data memory references: this event indicates all data memory references to the L1 data and instruction caches and to the L2 cache, including all loads from and to any memory types.

5. Instructions retired: this event indicates the number of instructions that retired or executed completely. This does not include partially processed instructions executed due to branch mispredictions.

6. Resource-related stalls: this event counts the number of clock cycles executed while a resource-related stall occurs. This includes stalls due to register renaming buffer entries, memory buffer entries, branch misprediction recovery, and delay in retiring mispredicted branches.

We have implemented profiling routines to monitor cache behavior as in item [1] and [2] mentioned earlier. These hardware performance counters are used along with any of the profiling routines in Section III-A.

Intel processors with streaming SIMD instruction extension provides four types of prefetch instructions corresponding to four prefetching hints to the processor, one non-temporal (meaning data which are referenced once and not reused in the immediate future) and three temporal (meaning data will be used again soon). We would like to briefly discuss these prefetching instructions here:

1. prefetchnta: non-temporal; fetch data into location closest to the processor, minimizing cache pollution.

2. prefetcht0: temporal; fetch data into all cache levels.

3. prefetcht1: temporal; fetch data into all cache levels except the 0th level.

4. prefetcht2: temporal; fetch data into all cache levels except the 0th and 1st levels.

In our current implementation, we use `prefetchnta` exclusively. Intel's provision of these prefetching mechanisms are intended for using with other SIMD instructions. Our preliminary experiment shows that programs can benefit from these prefetching instructions even without using the other SIMD instructions. We differ this discussion later in Section V.

We use the information provided by these hardware performance counters to set criteria in choosing hot paths to be optimized. The opportunity for optimizing a hot path that encounters a lot of cache misses is better than a hot path that rarely encounters any cache misses.

After we have implemented the hardware performance counter profiling routines, we encountered some problems that prevent the instrumented programs from running properly under Windows 2000. The RDPMC instruction is considered a privileged instruction and Windows 2000 refuses to execute it. According to Intel, RDPMC should be able to run on any platforms even within the least privileged level (the user application level). Looking through Intel's lengthy manuals, we found out that the ability to execute model specific instructions like RDPMC is determined by the bit values in the privileged control registers

(CR0-CR3). For RDPMC to work, the 8th bit in CR3 must be set. We then tested CR3 under Windows 2000 and found out that this bit was cleared. This situation is unfortunate since control registers can only be changed by the kernel. We consulted Microsoft's knowledge base, but failed to find any solutions to get around this problem.

We later tested our instrumentation code (manually changed to NASM assembly format) under Linux using NASM assembler and the code worked perfectly. The counter information obtained by our profiling tool is almost identical to that gathered by Intel's VTune[3]. Our original reason choosing Intel's C++ compiler and Windows 2000 is that we can use Intel's VTune tool to verify our results. This turns out to be a bad choice. Had we chose our NASM and Linux as our platform, we could have avoided this problem.

Since we have already developed our tool under Windows 2000, we decide to keep it as is and use VTune to instrument these hardware counters instead[4]. We use our tool to perform instrumentation and obtain hot paths, we then run the un-instrumented program under VTune, insert hardware counters to the corresponding paths identified during the profiling phase and get the cache-related information from VTune. Most of the instrumentations are still done with our own tool. Another reason that we decide to keep Windows 2000 as our platform is that we can use VTune to verify our designs. The VTune we have runs only on Windows.

IV. Optimization Phase

Our current offline optimization tool focuses on reducing cache misses[5] within hot paths.

[3]VTune simulates some of the instrumentations without actually executing them. Since we have to add instructions to fetch information from hardware counters, these additional instructions result in slightly more cache misses. The additional cache misses due to instrumentation is not significant however, especially for path profiling, since these hardware counter instructions are only added at the beginning and the end of each hot path, which typically span quick a few basic blocks.

[4]Internally, VTune simulates x86 architecture so that there is no concern whether the OS allows the execution of RDPMC.

[5]Both L1 and L2 cache, but we do not make distinction between them due to the prefetching instruction and method we chose.

When a hot path is found to have encountered enormous cache misses, we optimize this particular path by rearranging its basic blocks and performing prefetching.

To enhance instruction cache locality, we change the layout of basic blocks along the hot path so that logically connected blocks (via jumps) are contiguously placed in memory.

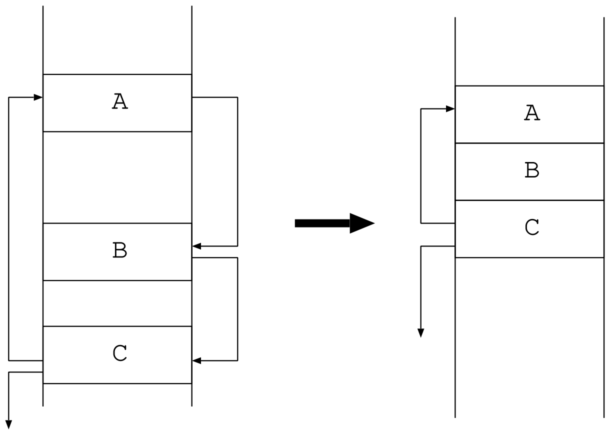

Fig. 3. Change code layout for better locality

Figure 3 illustrates the idea of changing code layout for better instruction cache locality. If the three basic blocks A, B and C are on hot path, they are moved to be physically near each other. This optimization can improve the localities in the execution instruction streams and improve the cache utilization, reduce I-cache miss rate.

For data cache, we focus on prefetching. Program runtime profiles can give valuable information that can be used to guide prefetching schemes that improve data cache performance. we restrict our scope to the basic blocks within hot paths, since cache misses along a hot path cause significant performance degradation. We inspect these basic block along hot paths and use the profile information to locate the load instructions that cause the most cache misses, and we selectively prefetch for that instruction. The advantage of selective prefetching is that it can avoid prefetchings that are unlikely to boost performance, reduce the memory traffic, and still achieve good performance. However, the difficulty of the selective prefetching idea is that it is hard to decide the prefetching address and start prefetching early enough since we are on assembly level.

Determining the prefetching address is a hard issue. Intel's x86 architecture only makes it harder. x86 architecture allows many different ways to calculate addresses. For example, a load or store address can be first calculated and stored in a register and later loaded via `lea`; the calculated address could itself be stored in a memory location as well. It turns out that the way Intel C++ compiler generated assembly code calculates memory addresses is very non-intuitive—the calculation and reference typically span multiple basic blocks. To make things harder, since local variables are stored on the stack, distinguishing a simple variable access and an array access is extremely difficult without the understanding of the source code.

Nevertheless, we have implemented a simple prefetching scheme and the experiment results are dismal (see Section V). In our simple scheme, a load instruction of the form `mov reg, [mem]` is located in basic blocks within a hot path, and the instruction `prefetchnta` [mem+disp] is added to blindly touch the next memory blocks anticipating future usage. The `disp` parameter is decided a priori so that the fetching distance is large enough to ensure the next memory block is loaded into the cache. Since we are using a fixed displacement parameter, programs that access memories with arbitrary large strides are unlikely to benefit from the prefetched block. Figure 4 illustrates the blind prefetching scheme.

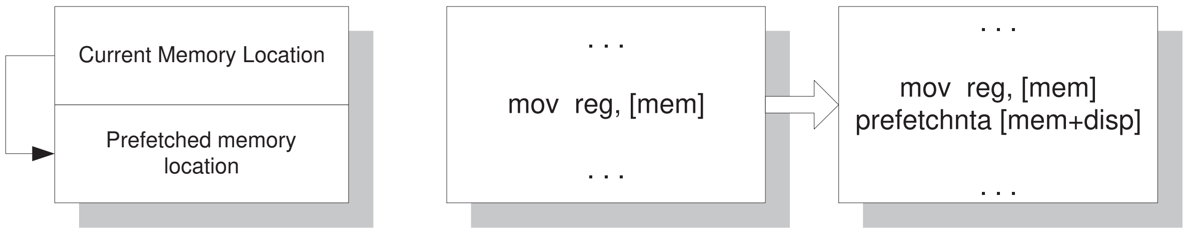

Fig. 4. Blind prefetching

It is always safe to use `prefetcha` to reference a memory location even if the memory location is invalid since `prefetcha` is an idempotent instruction, illegal references are simply ignored by the processor. It turns out that the memory traffic caused by prefetching increased drastically using our simple "blind" prefetching method and the performance of the optimized program using this scheme actually degrades.

It is possible that some more advanced prefetching analysis can be used to make prefetching work better. The profiling information on cache misses within hot paths is still very valuable however, since the hot path identified in the dynamic profile can be easily matched to the corresponding source code locations, a programmer could use this knowledge to perform prefetching at the source level[6]

V. Experimental Results

This section discusses some of the experiments we performed using our profile directed offline binary optimization tool.

To test the performance of our optimization methods, we build three mini-benchmark programs. The first program is a prime number generator (we shall refer to this program as `prime` in the future), which generates all the prime numbers within a given range. The second program is a binary sorting program (`bsort`). Given an array of integers, the program sorts the array elements using binary sort and stores the result to a pre-allocated array. The third program simulates sparse matrix operations (`sparse`), in this program diagonal elements in a 10000x10000 array is fetched at a given constant stride, some lengthy operations are done with the fetched array element.

It is desirable to use a standard benchmark suite instead of creating our own since using a standard benchmark suite could allow us to compare our results with those of others.

[6]prefetchnta can be used in in-line assembly code in C or C++.

The reason we chose to write our own benchmark programs is that our tool currently ignores inter-procedure calls. Since we form hot paths only from intra-procedure jumps, the presence of large amount of library calls in most standard benchmark programs will distort the result of our tests (time spent and cache misses within library calls are not counted in our current implementation. If the time to perform a call is a significant portion with respect to the whole path, speed gains are hard to measure. Further, if the cache misses within a procedure call overwhelms that of the rest part of the path, we would not be able to see any gains of prefetching.). Our benchmark programs do not contain any library calls except `printf` used to output counter values. The inclusion of `printf` does not skew the result however, since it is performed only once at the end of the program and thus does not change the behavior of the program's main body.

We first performed path profiling on these three programs and the counter values for the two hottest paths are shown in Figure 5.

	prime	bsort	sparse
Path 1	937,328	960	100,000
Path 2	4999	730	1000

Fig. 5. Results of path profiling

In these three programs most of the hot paths correspond to inner loops. Since our test programs are quite small, the total number of very hot paths are limited to just a few.

Figure 6 shows the total cache misses on these paths for the three tested programs in Figure 5. The cache miss includes misses resulted from both instruction and data reference.

	prime	bsort	sparse
Path 1	1,105	659	47,460
Path 2	873	314	734

Fig. 6. Hot path cache misses

The above figure supports our conclusion that hardware performance counters are useful in determining program properties. Both `prime` and `bsort` exhibit weak correlations between path frequencies and cache miss rates. This is due to the fact that the code sizes of these two programs are rather small and most values needed can be found in cache. Note that in `prime`, even though the path frequencies for the two hottest paths differ significantly, the cache misses for these two paths are actually quite comparable. We are reassured that path frequency itself cannot be used as the only criteria for optimization.

Figure 7 shows the execution times for these three programs before and after rearranging basic blocks within their hot paths. It would be desirable to show a comparison of the actual cache miss rates between the original and optimized programs. We did not do this since our tool currently can only work with the original assembly file generated by Intel C++ Compiler. Our tool uses some non-standard block layouts (see the print counter block in Figure 2) and makes future modifications to the optimized code difficult. The execution time for these programs however, can be easily measured as a byproduct of the optimization phase. It should not be hard to figure out the effect of optimization from the execution time however. We expect to add more functionalities to our parser so that it can handle more non-standard situations.

	prime	bsort	sparse
Before	15.57 s	0.1845 s	6.13 us
After	15.51 s	0.1844 s	6.13 us

Fig. 7. Execution times before and after rearranging basic blocks within hot paths

None of the tested programs benefited much from rearranging basic blocks. Since these programs memory footprints are quite small, rearranging basic blocks can only help reducing initial instruction cache misses when the programs are executed for the first time. The above figure verifies this view. Particularly, in `sparse` most cache misses are due to data references, rearranging basic blocks has no visible impact on the program's performance. We expect this technique be useful for much larger programs.

As mentioned in Section IV, we have implemented a simple “blind” prefetching method that blindly touches future data references. The results turned out to be poor. Figure 8 shows the blind prefetching results.

	prime	bsort	sparse
Before	15.57 s	0.1845 s	6.132 us
After	15.93 s	0.1856 s	6.129 us

Fig. 8. Execution times for blind prefetching

In this figure, we can see that both `prime` and `bsort` are actually running slower after blind prefetching due to the excessive memory traffic caused by unwise prefetchings that are never referenced later. `sparse` runs slightly faster since the original program is designed in such a way that it encounters a cache miss almost on every other data references. We are reaffirmed that prefetchings need to be accurate enough to be effective.

From the profiling information we obtained, we manually modified `sparse` by inserting prefetching instructions (inline assembly code) into the C++ source code. We see a 2% speed gain and after this tuning, `sparse` finishes execution in 6.001 micro-second.

VI. Future Work

So where do we go from here? The experimental results we got are very encouraging. Even though in most cases the speedups gained by optimization are quite small, we believe this is mainly an artifact of our small code-sized benchmark programs. We expect our tool to be more effective on larger programs. We would like to create some larger programs to test our tool in the future.

As mentioned in the previous section, our current tool does not account for time spent in library procedure calls nor does it taking into account the cache misses within procedure calls, so it is hard to obtain meaningful results with standard benchmark suites. Within our framework, it is possible to add extra functionalities to monitor such behaviors.

We have not really analyzed the sensitivity issue. All of our benchmark programs use the same input sets before and after optimizations. Our assumption is such that the workloads for most programs are rather stable and slight variations to the inputs would not inversely affect the speed gains obtained via optimization. Even though many independent studies have suggested this result, we are yet to verify it on our own.

Our current tool allows only one kind of profiling and optimization to be performed at a time, which limits many potentials that could lead to further performance gains. However, our platform is sound enough to make further extensions possible.

Automatic prefetching remains a hard issue. The core of this issue lies in how to infer future access patterns of a program from its binary code. Since prefetching consumes memory bandwidth and needs extra CPU cycles to compute the potential reference addresses, any mispredicted prefetching will likely to affect program performance inversely. So far, we have not come up with a plausible solution. More work needs to be done to come up with an efficient prefetching algorithm.

VII. Conclusions

This paper presents a framework for performing profile directed offline binary optimization. Compared with various dynamic optimization methods, profile directed offline binary optimization has many advantages. The most significant advantage of offline optimization is that program runtime overhead is minimized. Most programs optimized using our paramilitary methods show a speedup up to a few percent. Our preliminary experiments reaffirmed the advantages of offline optimization method. The only programs that actually run slower after optimizations are due to our ad hoc blind prefetching method. We are quite convinced that offline optimization is promising. Based on our framework, more sophisticated optimization algorithms can be added and more optimization opportunities can be explored to achieve better optimization results.

References

[1] E. D. V. Bala and S. Banerjia, "Dynamo: A transparent dynamic optimization system," in *Proc. of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, 2000.

[2] M. A. et al., "Adaptive optimization in the jalapeno jvm," in *OOPSLA '00*, 2000.

[3] M. A. H. R. J. Hookway, "Digital fx!32: Combining emulation and binary translation," in *Digital Technical Journal*, Jan 2000.

[4] J. L. T. Ball, "Efficient path profiling," in *Proceedings of MICRO-29*, Dec 1996.