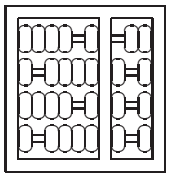

INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Dynamic Optimization Effects on DBT

Daniel Nicácio *Wesley Attrot*
Guido Araújo *Edson Borin*

Technical Report - IC-08-015 - Relatório Técnico

July - 2008 - Julho

Dynamic Optimization Effects on DBT

Daniel Nicácio* Unicamp
Wesley Attrot† Unicamp
Guido Araújo Unicamp
Edson Borin Intel

Abstract

Dynamic Binary Translation can provides code portability between different architectures. The translation process occurs on the fly by translating the binary code from source to target architecture. Since this is a dynamic process, the translation overhead must be minimum.

One way to compensate the overhead cost of binary translation is to dynamically optimize traces of the translated code. This paper discuss the influence of trace quality on dynamic optimization and also presents three dynamic optimizations and shows which gain expectations a dynamic optimization system can have.

1 Introduction

Dynamic Binary Translation (DBT) consists in, given a stream of executing code, dynamically translate it to a target ISA. In the last years DBT has become an interesting subject due to its many advantages, being the research topic of some works [4], [7], [11], [18], [12]. DBT provides code portability between totally different architectures, offering a new level of freedom for the computation environment. It automatically converts code to run on a new architecture without recompiling the source code. Another benefit is to make possible to legacy code to use new architecture technologies which were not available at the time they were written and compiled, like a larger number of available registers [2].

DBT also can provides information collected at runtime. The static compiler does not have this kind of information, so using it, new optimization opportunities comes to surface. In order to have a efficient optimization system, it's essential to know where to apply the optimization, that is, know how to obtain critic regions of code which really sets out gain possibilities. A sequence of instructions, including branches but not including loops, that is executed for some input data is called a trace [19]. DBT selects traces which are heavily executed as good spots to make improvements. Since DBT is made during the program execution, time becomes a critical factor. Optimizing traces is one promising way to overcome the overhead of translating instructions to the new ISA, permitting the new code to run faster then the original one.

*This work is sponsored by CAPES
†This work is sponsored by CNPq and Intel

This paper discuss three building traces techniques, which features a trace must has to be considered a good trace for optimization and how those features affects a dynamic optimization. Then, we use three dynamic optimization to evaluate the effects of optimization on DBT systems.

Section 2 presents the DBT environment used in this work. It also explains the mechanism to dump and load traces from files. This mechanism permits the traces to be modified offline and then different sets of traces can be compared to each other, like non-optimized traces and optimized traces. Section 3 enunciates the features a trace must has to be considered good and why they make then a good candidate for optimization. This section also presents three groups of traces, how they were collected and some experiments using them to demonstrate how the features presented affects the overall optimization quality. In section 4, the three optimizations studied in this paper are presented. There is a brief explanation of each one and why they were selected as promising dynamic optimization. Section 5 shows existing trace detection techniques and some works which intends to develop good dynamic optimization environments. Section 6 presents a massive number of experimental results, providing a solid argument to justify the lack of dynamic optimization environments. Section 7 concludes this work.

2 The Environment

We implemented the optimizations in our DBT translating from IA32 to IA32. Our DBT [6] runs on Linux and has its structure shown at Figure 1. It has a kernel module that is loaded and replaces the system call **execve**. Every time that a program will execute, the module checks for a shared library in the same directory of the application, if this library is not found, them the original Linux execve is called. If the shared library is found, them it is loaded and starts the execution. This shared library is our DBT itself, that will them load the application code and starts the translation process.

The DBT has 3 main modules: *frontend*, *runtime* and *backend*. The *frontend* module translates the application instructions and store them in the code cache and controls the program execution in this cache. The *runtime* makes the communication between the DBT and the OS and the application and the OS, providing interfaces to I/O and system calls, handling system signals, dynamic shared objects loads and self-modifying code. The *frontend* and the *runtime* interact to handle system related features. The *frontend* is also responsible to select hot traces for runtime optimization by the *backend*.

The *backend* module is responsible for trace optimization. The DBT is configurable to make the *backend* module to dump the hot-traces into files or to load them from the same files. The dumping process occurs at the end of a first run. DBT translates the program capturing all the hot-traces, and at the end of it's execution it translates the traces into a intermediate representation and stores it in a file like the one in figure 2. For the load process, the DBT keeps the original address of all traces dumped into the file in a table, so every time the original program would execute an instruction at such address the DBT loads the respective trace instead. This dump/load system permits the trace to be modified, that is, instrumented and optimized offline and then executed. This system permits a valid

time comparison between runs with non-optimized traces and with optimized traces. A first run is made loading the traces without any modification, a second run is made loading the optimized traces and then the time spent at each one is compared. This way all the overhead of translating and optimizing the program is discarded and only the optimization gain is measured.

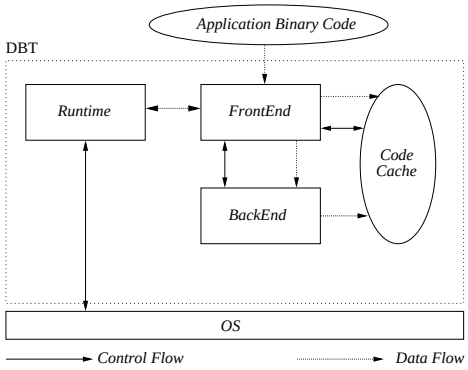

Figure 1: Our Dynamic Binary Translator Structure

Traces are dumped into files like the example in figure 2. This trace was extracted from the 164.gzip program of the cpu2000 SPEC benchmark. This file contains the trace identifier; a header with it's original address in the original code, a checksum number used to verify if it's the correct trace to load, a list of traces which has this one as a branch target, a counter named `ExecutioFrequency` to keep how many times this trace was taken (executed), another counter named `NumberOfBlocks` to tell how many basic blocks are in the trace and `HeadBlockSize` and `RealHeadBlockSize` are used to know how much space is really available at the beginning of the trace to make some code instrumentation. The `Code` segment is composed by `blks`, which are a sequence of instructions (both it's opcode and mnemonic), branch instructions, the original address of each basic block (bb_org_addr) and the times each basic block was executed inside the trace (exec_info). The exec_info is also used to know how many times each Side Exit (SE) was taken, that is, how many times a branch result of the program execution flow left the trace and followed another path.

3 Trace Construction

A set of optimizations may appear to be inefficient in some dynamic optimization systems. Those results may point that dynamic optimization are not worth. But the optimization quality may not be the cause of the bad results, the quality of the traces where the optimizations are been applied has a great hole on the final result, it can maximize good results or turns it into performance loss. Trace detection techniques are essential to the success or not of a dynamic optimization set [14].

To achieve good optimization results on traces, the desired trace should have three main features: High number of instructions, high Execution Coverage (EC) and high Completion Rate (CR). The following subsections explain what are they and why they are so important.

```
Trace 2 {

  ORIGINAL_ADDRESS: 0x0804883a

  CHECKSUM: 0x9e29860b

  Predecessors 2 {
    ORG 0x080487f8
    ORG 0x08048acb
  }

  Stats {
    ExecutionFrequency: 1841987194
    NumberOfBlocks: 3
    HeadBlockSize: 40
    RealHeadBlockSz: 40
  }

  Code {
    exec_info    1841987194
    bb_org_addr    0x0804883a
    blk {
      8b 45 08                  ; mov      eax, DWORD PTR [ebp+08h]
      8d 80 40 21 11 08         ; lea      eax, DWORD PTR [eax+08112140h]
      89 45 f4                  ; mov      DWORD PTR [ebp-12], eax
      8b 45 f4                  ; mov      eax, DWORD PTR [ebp-12]
      8b 55 f0                  ; mov      edx, DWORD PTR [ebp-16]
      0f b6 04 02               ; movzx    eax, BYTE PTR [edx+eax]
      0f b6 c0                  ; movzx    eax, al
      0f b6 55 fc               ; movzx    edx, BYTE PTR [ebp-4]
      0f b6 d2                  ; movzx    edx, dl
      3b c2                     ; cmp      eax, edx
    }
    je           ORG 0x08048862
    exec_info    1676374850
    bb_org_addr    0x08048aae
    blk {
      8b 45 08                  ; mov      eax, DWORD PTR [ebp+08h]
      25 ff 7f 00 00            ; and      eax, 0x7fffh
      0f b7 04 45 00 d5 0e 08   ; movzx    eax, WORD PTR [eax*2+080ed500h]
      0f b7 c0                  ; movzx    eax, ax
      89 45 08                  ; mov      DWORD PTR [ebp+08h], eax
      8b 55 ec                  ; mov      edx, DWORD PTR [ebp-20]
      3b c2                     ; cmp      eax, edx
    }
    jbe          ORG 0x08048ad8
    exec_info    1643523311
    bb_org_addr    0x08048acb
    blk {
      8b 45 dc                  ; mov      eax, DWORD PTR [ebp-36]
      48                        ; dec      eax
      89 45 dc                  ; mov      DWORD PTR [ebp-36], eax
    }
    jnz          ORG 0x0804883a
    exec_info    10629421
    jmp          ORG 0x08048ad8
  }

}
```

Figure 2: A Trace Dumped Into a File

3.1 Trace Length

In case a trace has only one basic block, it will be almost impossible to find optimization opportunities, since the static compiler has already done every thing possible on that single basic block and the program execution didn't add extra information to a single block trace. In other words, the single basic block trace is identical to the correspondent block at the control flow graph (CFG).

A trace with many basic blocks shows which path the program is wiling to follow. This turns out which computation is really needed and which are just wasting time. It is also possible that complete procedure calls are included in a long trace, making possible to reduce the overhead of context saving needed by a procedure call.

To sum up, the longer the trace is, more runtime information is available and more powerful the optimization becomes.

3.2 Execution Coverage

Execution Coverage (EC) is the time percentage of the hole program spent inside the trace, that is, the trace relevancy in the total program time. If a program is completed in 100 seconds, and 10 seconds is spent inside trace Y, we say that trace Y has an EC = 10%.

As the Amdahl's law [3] states, any optimization performance gain will be limited by the trace EC. An extremely good gain of 50% on trace with 10% EC will result in a modest 5% on the program. In short, choosing significant traces in time terms is crucial for an acceptable optimization result. For that reason, EC is one of the most important factors to have in mind when picking a trace for optimization.

This second feature usually walks side by side with the first one, since longer traces means more time consuming. But the third feature shows that it is not necessarily true, and it forms a trade-off between the three features, which makes the task of choosing a trace to optimize even harder.

3.3 Completion Rate

A trace with many basic blocks also has many side exits, conditional branches witch defines if the program execution stays in the trace or leaves it, going to another trace or to a cold code region. If it leaves the trace we say that the side exit was taken. A trace execution is said complete if all of it's instructions are executed, from the trace head to the trace tail without taking any side exit. The Completion Rate (CR) is given by the number of completed executions divided by the total number of executions (number of times the trace was taken).

While optimizing a trace, the trace analysis is made hoping the trace will be totaly executed. As the next example shows, if the trace has a low CR, optimizations will be sub-used, or even worse, they will affect the trace in a negative way.

Let's take for an example the trace in Figure 3, this trace was taken from 300.twolf program from the SPEC cpu2000 benchmark. This trace, composed by 8 basic blocks can be considered a long trace, and consequently, we believe it has a high EC. Meanwhile, if we take a closer look, we see that the side exit in instruction `jnz ORG 0x080c08df` is taken 61.33% times the trace is executed. So, the trace completes it's execution only 38.67%, thus, we say the trace CR = 38.67%.

```
Trace 25 {
  Code {
    exec_info    145591
    blk {
      mov      eax, DWORD PTR [ebp-632]
      cmp      DWORD PTR [ebp-628], eax
      mov      ebx, DWORD PTR [ebp-636]
    }
    jnz          ORG 0x080c08df
    exec_info    56297
    blk {
      mov      edx, DWORD PTR [ebp-628]
      add      edx, edx
      cmp      edx, 0x100h
      mov      DWORD PTR [ebp-632], edx
    }
    jae          ORG 0x080c0b77
    exec_info    56297
    blk {
      mov      DWORD PTR [ebp-632], 0x100h
      mov      ecx, DWORD PTR [ebp-632]
      lea      eax, DWORD PTR [ecx*4+010h]
      sub      esp, eax
      mov      eax, DWORD PTR [ebp-636]
      lea      ebx, DWORD PTR [esp+01fh]
      and      ebx, -0x10
      test     eax, eax
    }
    je           ORG 0x080c08df
    exec_info    0
    jmp          ORG 0x080c0b9b
  }
}
```

Figure 3: Compensation code issue example

Now we apply the dead code elimination optimization (shown in section 3.1) on that trace. We see instruction `mov ebx, DWORD PTR [ebp-636]` is killed by instruction `lea ebx, DWORD PTR [esp+01fh]`, thus it can be removed from the trace. But the value computed by the `mov ebx, DWORD PTR [ebp-636]` can still be used outside the trace if any side exit is taken. So, it is necessary to insert compensation code on every side exit between the two instructions (killed and killer). The compensation code requires an additional jump instruction to be inserted in the trace, so the trace jumps to the compensation code and then it jumps to the original side exit target, figure 9 shows a compensation code example. That way, if we execute the optimized trace, there will be 1145334 instructions executed, and the original trace executed only 1112337 instructions. In this case, the optimization added 32997 more instructions to the program execution (2.97% more instructions), making it worse.

To summarize, a trace with many basic blocks sets out more optimization opportunities, but a further analysis must be made to know if the trace CR is high enough to make it worth.

3.4 Traces

This section provides a detailed analysis of three groups of traces: MRET2 traces, loops by tail and loops by most executed side exit. This subsection explains what are each group of traces and how they were obtained. At the end many results concerning the three features discussed are presented.

The Most Recent Executed Tail (MRET) technique [4] was first used in the dynamic optimization system called Dynamo [4]. The technique consists in associates a counter to hot spots (instructions) of the program, these hot spots are usually targets of backward branches. That way they intend to capture the program loops. When a counter reaches a certain threshold, the trace begins to be constructed. The trace construction last until a stop point is reached. This technique is based on the idea of once a instruction is defined as a hot spot, the subsequent instructions will be hot too. This technique allows to hold counters only on the hot spot instructions, and once the trace is constructed, the counter can be trashed.

The traces studied in this work came from the MRET2(Two-Pass MRET Trace Selection for Dynamic Optimization). As the name says, it is based on the MRET technique. The difference between then consists in, after a trace is identified, the counter of it's instruction start point is reset and a second trace is identified when the counter reaches the threshold a second time. With the two traces at hand, an intersection of these traces is made, the result of this intersection is the final trace selected as a hot trace of the program. Since many programs have a lot of control flow in their hot regions, it is likely that MRET traces achieve a low CR, the idea of make the intersection of two runs is to significantly increase the CR at a cost of reduce the trace length.

With the obtained MRET2 traces, it was possible to make experiments to simulate the result of two new trace detection techniques. These new techniques are explained next.

3.4.1 Loops by Tail

Working with MRET2 traces is possible to notice that the junction of some traces would create a loop. The target of the final branch of a trace is the same address of the beginning of another trace, this fact occurs until a loop of traces is formed. So, all the MRET2 traces were analyzed looking to form trace loops by linking then by the tail to another header.

3.4.2 Loops by Most Executed Branch

A MRET2 trace may have many side exits, and even with those traces, we may find some low CR. In those cases, it is likely that a side exit is taken more times then the loop tail. So, the third group of traces is obtained in a similar way of loops by tail, but here we look for the target address of the most executed branch (side exit and the trace tail) to form a loop. Figure 4 shows a loop formed that way. Taking `Trace 180` as the loop header it's tail points to `Trace 160` which points to `Trace 161`, then, `Trace 161` has a SE which points to trace `Trace 162` and this last one also has a SE, which goes back to the loop header `Trace 180`. Note that in the loop built in Figure 4 (b), labeled `Trace 100180`, the side exits taken in the previous traces had it's branch negated, for example, SE `jge ORG 0x08048797` turned to `jnge ORG 0x0804878b`.

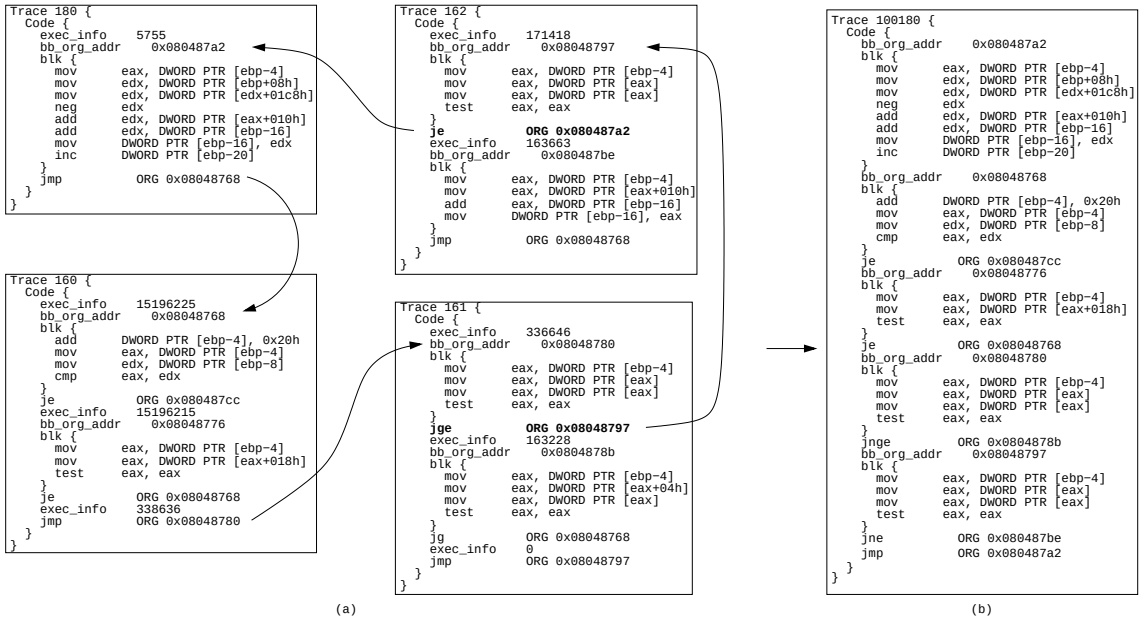

Figure 4: A loop built from four traces, following any possible side exit

Since we find a cycle of traces, it is not certain which trace should be considered the loop header. In fact, the formed cycle should has many entry points. Aimimng to achive the higher EC as possible, the loops were replicated, each one with a different loop header. This way, the loop built in figure 4 were replicated three times, resulting in four loops, each with a distinct loop header (Trace 160, Trace 161, Trace 162 and Trace 180).

3.4.3 Trace Results

Figure 5 shows the average CR of traces for each SPEC2000 program. Despite the high completion rate of traces, it is not necessarily a good result. The MRET2 technique finds a lot of single block traces, which will undoubtly generate a high CR.

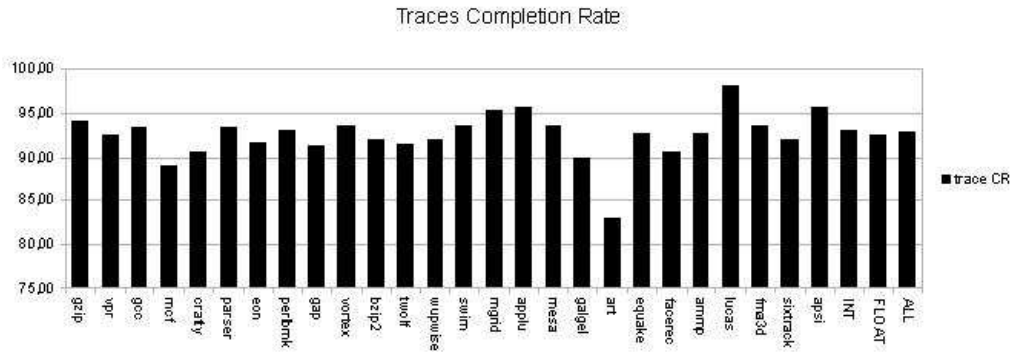

Figure 5: The Completion Rate of Traces

Figure 6 presents the average number of instruction of each trace in the MRET2 traces and in the loops by the most executed branch. Note that the trace lentgh of the loops are 16% greater than the lentgh of MRET2 traces on average.

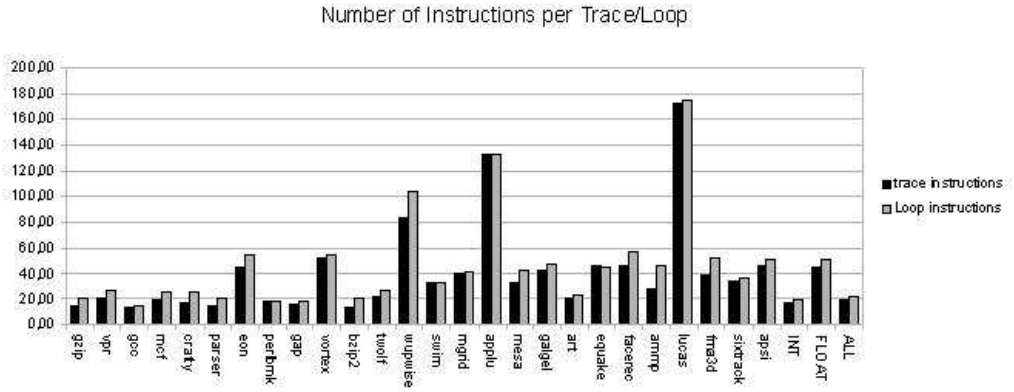

Figure 6: The average number of instructions in each trace/loop

Figure 7 presents the total EC of the MRET2 traces and of the loops by the most executed branch. Figure 8 presents the EC given by the 10 most relevant traces/loops of the MRET2 traces and of the loops by the most executed branch. A small drop on the EC of loops was expected since some trace tails are discarded to form a loop. But the EC of the 10 most relevant traces were suposed to raise, and as the graphics show it did not happen. A possible reason for this result is the fact that some trace tails can be very relevant in

time terms, they are executed just a few times but has a lot of heavy instructions. Cutting those tails may generate the EC drop on loops.

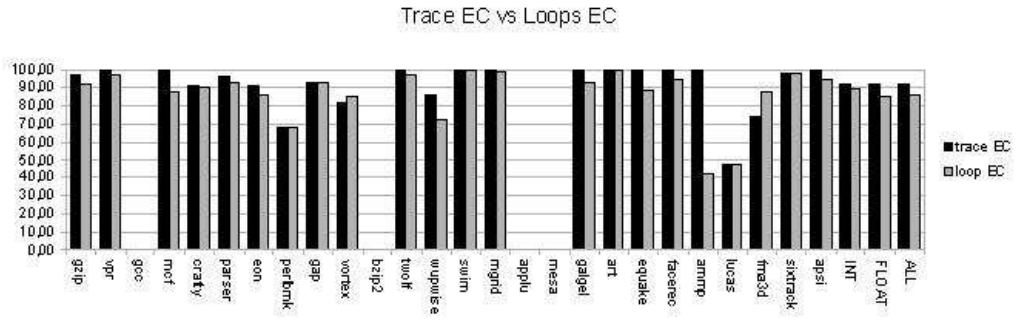

Figure 7: The EC of Traces/Loops

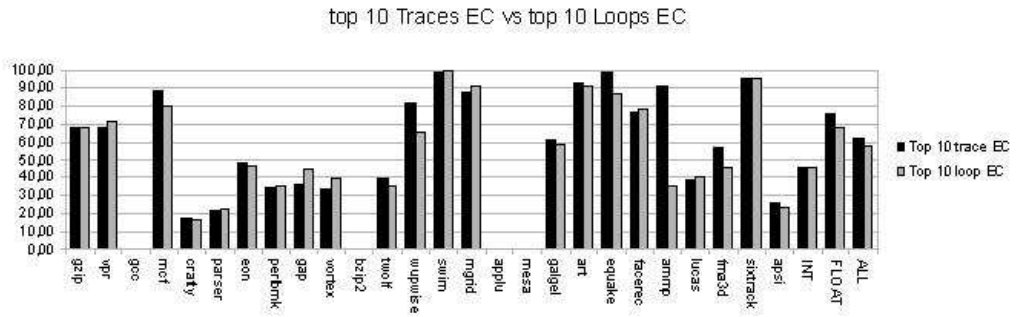

Figure 8: The EC of Trace/Loops for the 10 most significant trace/loop

Unfortunately, it was not possible to measure the EC of some programs. The 176.gcc crashes when running with MRET2 traces generated by the DBT. 173.applu stays in an infinite loop when running with MRET2 traces. 256.bzip2 has a corner case where the loop construction fails and the 177.mesa stays in an infite loop when running loops. Despite of the lack of these programs, the result of almost the entire SPEC cpu2000 benchmark gives a solid base to evaluate the two sets of traces. Both techniques generates traces with few instructions in average (19.19 on traces and 22.30 on loops), this number usually is not high enough for dynamic optimization. The EC of both techniques are fragmented in many traces, since the EC of the ten most relevant traces is around 60% only, this fact also harms the optimization efficiency. The overall CR is high (92.95%), and the loops CR should be greater than that, but the low number of basic blocks in each trace can be the reason for that good result. To sum up, there still space for improvements in trace detection techniques focusing on dynamic optimizations.

4 Dynamic Optimizations

4.1 Dead Code Elimination

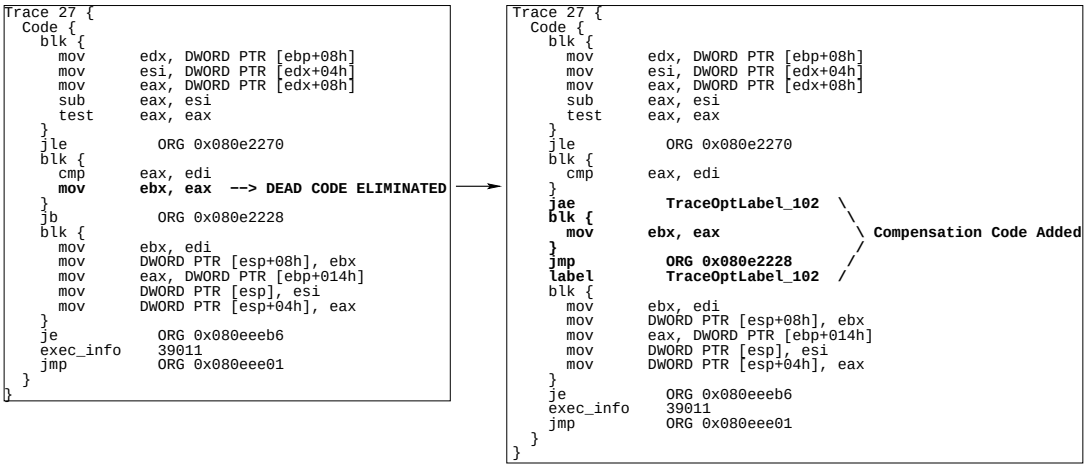

Figure 9: DCE Example with compensation code

Since we are looking at a list of basic blocks as a single big basic block, some dead code previously hidden will come to sight and therefor should be removed from the trace. Besides the dead code already present in the original code, other optimizations, like Hole Allocation, Code Motion, Copy Propagation, Register Remapping, among others, will generate some garbage code. This code must be removed by Dead Code Elimination too. For that reason, Dead Code Elimination (DCE) is a critical optimization to be done in a dynamic environment.

4.2 Issues

4.2.1 General Issues

Traces usually has some or many side exits. The first issue is: what if a instruction is removed and the execution leave the trace before the instruction was really dead? If this happens, the program semantics will be changed and the program will crash or generate a wrong result.

To solve this problem, when a instruction is removed, Compensation Code must be inserted in each SE between the instruction removed and the instruction which killed the instruction removed. Figure 9 shows an example.

4.2.2 ISA Issue

Basically, to remove a instruction, we must see if all it's definitions are dead before someone uses it. In other words if there is no use of a variable between two definitions of this variable. In our case, variables are architectural registers, and flags are also considered registers.

```
mov     ch, BYTE PTR [eax+09h]
and     eax, ecx
shr     eax, 0x4h
add     edx, eax
mov     DWORD PTR [ebp-72], edx
mov     ch, BYTE PTR [ebp-60]
test    ch, 0x01h
```

Figure 10: ISA Issue

In the X86 architecture, one register is composed by others registers. For example, EAX is formed by it's high part and the low part called AX, which is also divide in AH and AL. Lets see the figure 10, at first, we may think that the instruction `mov ch, BYTE PTR [eax+09h]` is dead, cause it is killed by `mov ch, BYTE PTR [ebp-60]` and no one uses CH between them. But instruction `and eax, ecx` uses ECX, and since ECX is formed by CH, is also uses CH, let's say it is a implicit use of CH, and therefor the instruction `mov ch, BYTE PTR [eax+09h]` can't be removed.

To turnover this problem, we set that: if a instruction is defining/using AH or AL, it is also defining/using AX and EAX; and if it is defining/using AX it is also defining/using EAX. This way we guarantee that we don't remove live code and we also don't miss a chance to remove dead code.

4.2.3 Compensation Code Issue

At first, to remove a instruction from a trace, we just see if all it's definitions are killed before it is used. So, in figure 11, instruction SUB define registers EDX and EFLAGS, looking forward in the trace, the CMP instruction kills EFLAGS and the MOV instruction kills the EDX definition; then, the SUB instruction is a dead instruction and can be removed.

But, notice the SE at the instruction `js ORG 0x0804ef5e`, the SUB instruction must be moved to that SE. But the target of that SE is just another branch. Now we realize that the SUB instruction cannot be moved there, because it defines EFLAGS witch is used by the jump instruction, and this jump is expecting to use the EFLAGS set by the CMP instruction. Concluding, dead code cannot be moved thorough instructions which kills one or more of its definitions. In other words, a instruction is only dead if all of it's killers are in a single basic block.

Similar to the logic above, we can't move a instruction thorough a instruction witch defines a register used by the instruction being moved. Let's see the example in figure 11 again , if the instruction `mov eax, esi` is moved to the SE, the definition of EAX will be done using a wrong value of ESI.

4.3 Function Inlining

While building traces in a dynamic environment, many system calls and even procedure calls will be fully incorporated in a single trace. That way, the overhead to save/load the context before and after the call/ret instruction may become unnecessary. The Function Inlining (FI) optimization consists in eliminate this overhead when possible. It is also

```
blk {
  mov      edx, ebx
  mov      eax, esi
  sub      edx, 0x1h
  lea      esi, DWORD PTR [esi+ecx]
  mov      DWORD PTR [ebp-24], esi
  cmp      ecx, ebx
}
js           ORG 0x0804ef5e
blk {
  add      esi, 0x1h
  xor      ecx, ecx
  mov      edx, esi
  movzx    eax, BYTE PTR [ecx+edi]
  add      ecx, 0x1h
  mov      BYTE PTR [edx-1], al
  add      edx, 0x1h
  cmp      ecx, ebx
}
```

```
bb_org_addr    0x0804ef5e
jnz          ORG 0x0804ef90
```

Figure 11: Compensation Code Issue

important because when a pair pop/push instruction is eliminated it underweight the table which tracks then.

Figure 13 (a) and (b) shows a trace before and after the optimization respectively. Since registers EAX and ECX were not used between the call/ret instructions, the overhead of saving it into memory is a waste and can be eliminated.

Note that pop and push instructions update register ESP, this must be taken into consideration if any instruction makes use of it inside the called function. If that is the case, those instructions must have it's displacement field update to keeps the memory/stack consistent.

4.4 Hole Allocation

Hole Allocation is an optimization that will try to remove memory access in the binary program. The main purpose is turn spill code into instructions using registers, but in a aggressive mode, we want to transform generic memory access into registers references. The first step is to identify **dead holes**, that is registers that don't have a live value. In the Figure 13 we have an example of dead hole.

In the example at Figure 13, the register **ebx** is dead at instructions `add eax, [ecx+0e0df39eh]` and `lea ecx, DWORD PTR[esp+014h]`. The set of instructions where ebx is dead are a hole in ebx lifetime, so we call this hole of dead hole.

4.4.1 Using Dead Holes to perform optimizations

Once we have identified dead holes, we try to use it to optimize the code. In the Figure 15 we have a piece of real code taken from SPEC program bzip2.

The instructions `mov DWORD PTR [esp+02ch], edx` and `mov edi, DWORD PTR [esp+02ch]` both access the same memory position, in particular, these two instructions looks like spill code inserted by the compiler. Looking closer in this code, the Figure 15 will present us

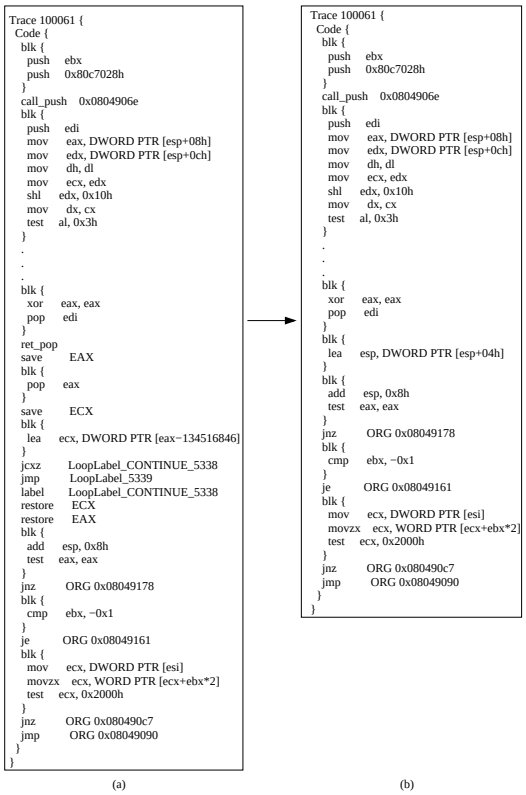

Figure 12: Function Inline Example

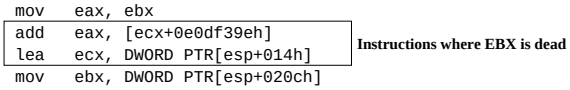

Figure 13: Dead Hole Example

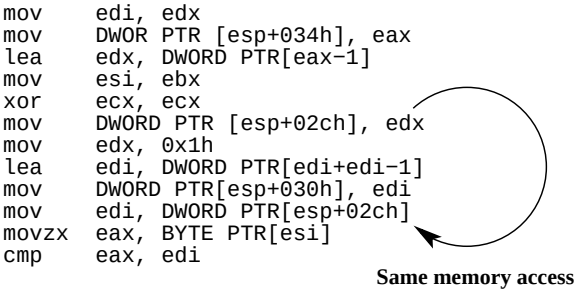

Figure 14: Repeated memory access

an interesting behavior. The lifetime of eax ends on `lea edx, DWORD PTR [eax-1]` and restarts at `movzx eax, BYTE PTR [esi]`, so eax is dead between them. This dead hole for eax overlaps the two memory access `DWORD PTR[esp+02ch]`.

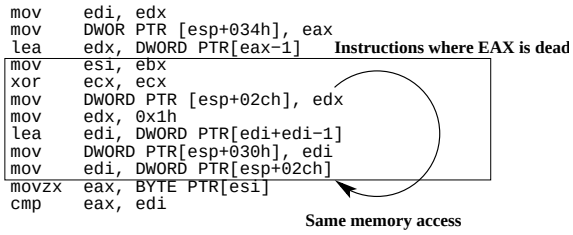

Figure 15: Dead Hole in bzip2

The main idea is to transform the second memory access into a register usage. For that, we introduce the instruction `mov eax,edx` after `mov DWORD PTR[esp+02ch],edx`, so eax now has the value as edx. Now we change the instruction `mov edi, DWORD PTR[esp+02ch]` to `mov edi,eax`. The whole transformation is shown in Figure 16.

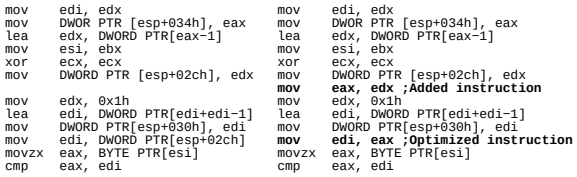

Figure 16: Memory access optimization

As presented in former example, the look for patterns in form `[esp+xxxxh]` to optimize, but the optimization is not limited to this pattern of memory access. Any kind of memory access pattern can be considered for optimization, the only issue to be addressed by the optimizer is alias. The optimizer need to be able to disambiguate memory references in the code that alias the same memory position.

5 Related Work

5.1 Trace Detection Techniques

Trace detection techniques have been studied for a long time now. Chang and Hwu [8] used counters on the edges between basic blocks, that way, the basic block with the most executed exit is known, and a trace is build following that path. The work of Balland Larus [5] used another heuristic, it counts how many times each acyclic path is executed, it reduces the number of required counters and results in the most executed path.

The Next Executing Tail (NET) [10] is a popular technique among dynamic optimization systems, Hiser [15] and Hiniker [14] points that 40,2% of NET traces consists of only one basic block and they have 14,8 instructions in average. Hiser also says that the NET traces CR is only 50,1% on average.

Hiniker et. al. [14] proposes another technique called Last Executed Iteration (LEI). This technique turns to be more efficient to capture cycles then NET. It's traces have 18,3 instructions on average, and they achives an overall 90% EC with fewer traces then NET. Unfortunately, the work does not states the traces CR.

Gal and Franz [13] presented the trace tree technique. It consists on finding hot regions, usually loops, and represent them as a tree structure. The tree structure has many benefits, which facilitates many analysis and optimizations. These benefits come with the cost of memory usage, since the trace tress uses a lot of code duplication.

Bala et. al. created the Most Recent Executed Tail technique (MRET) [4], used in the dynamic optimization system called dynamo. The technique consists in assossiates a counter to hot spots (instruction) of the program, these hot spots are usually targets os backward branchs. That way they intend to capture the program loops. When a counter reaches a certain threshold, the trace begins to be constructed. The trace construction last until a stoppoint is reached. This technique is based on the idea of once a instruction is defined as a hot point, the subsequents instructions will be hot too. This technique allows to hold counters only on the hot spots instructions, and once the trace is constructed, the counter can be trashed.

5.2 Optimization on DBT

Some works on binary translators uses optimization as a way to overcome the translation overhead. FX!32 [16] does a static translation and optimization. CMS (Code Morphing Software) presents the Transmeta Crusoe, focusing on system level optimization. DAISY [12] translates code targeting VLIW architectures and optimize the code so it can use the benefits of the architecture. SIND [20] is a framework for application and techniques development focused on flexibility, it uses branch modification and constant folding as dynamic optimization, but does not present any result.

Dynamo [4] interprets instructions of a PA-8000 to execute on a PA-8000 processor. It interprets instructions until a hot-trace is found. Then this hot trace is optimized and used the next time the program reaches it's address. The main optimization is the branch modification, where the trace branch are transformed so that their fall-through direction remains on the trace. Indirect branches are also transformed in direct branches based on previous profiling. Dynamo also performs redundant instruction elimination, copy propagation, constant propagation, strength reduction, loop invariant code motion and loop unrolling. They claim to have a gain between 3% and 5% due to trace optimization, and an overal gain of 9% on average. Most of it comes from the trace selection and allocation mechanism.

UQDBT [21] is a framework of dynamic binary translation which suport many architectures as source and target. It says it achives a 15% gain with optimizations. But their optimization includes trace locality in cache, brach modification (as in Dynamo) and inlining. It does not show how much gain is obtained from each one.

ADORE [18] uses hardware support (like Itanium 2 profiling registers) to evaluate the code behavior and them translate and optimize it. Their set of dynamic optimization are dynamic Register Allocation (which is based on the IA64 alloc instruction), Data Cache Prefetching and Trace Locality. They achieve some outstanding results as in the applu

program (speedup of 106.75%), but in average, the speedup is not clear.

6 Experimental Results

Our experiments were performed on SPEC2000 benchmark. The programs were compiled with the Intel Compiler using the peak tunning and ref input. For each program the DBT identified the hot traces and at the end of the program execution, the DBT does the dump of the traces to a file. Our DBT performs the dump of the traces into several files. One file called dbt.traces.trc with the hot traces identified by the DBT. Other file, called dbt.opt.traces.trc, is dumped with the traces optimized with the optimizations presented in the paper. Another file, called dbt.loops.trc contains loops built as explained in sections 3.4.1 and 3.4.2, another file called dbt.opt.loop.trc with the the loops optimized with the optimizations presented in the paper, another file called dbt.loops+traces.trc which is a union of dbt.traces.trc and dbt.loops.trc and another file called dbt.opt.loops+traces.trc which is also a union of dbt.opt.traces.trc and dbt.opt.loops.trc.

To obtain the time results, all programs were executed another time, but the DBT was configured to load the traces from files and execute them. This is possible because all spec programs were static compiled, with all libraries into them, so all address for each function/library call are the same each time that the program is optimized. In this way, we can also optimize the libraries that were included into the programs.

During trace generation, the DBT will identify a trace and dump to a file the starting address of this trace. To execute the traces that are in the files, the DBT first reads the file to load into memory the optimized traces. When the program reaches the starting address of the trace, the program execution branches to the trace that was read from file. With this fashion, we can profile traces and also perform offline optimization on them.

This schema of dumping/loading traces were employed to avoid the overhead of optimizing the traces during program execution. Our intention is to measure only the effects of an optimized trace on program execution time.

Each program were executed 10 times for every input file, as we have 6 input files, each program were executed 60 times. The time comparison was done between a set of optimized traces with a set of non-optimized traces in the respective set, so we compared the traces in file dbt.traces.trc with the optimized traces in the dbt.opt.traces.trc and so on. the results are shown as confidence interval, and the confidence is set to 95%.

Figures 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27 and 28 show the optimization results running on two different procesors: a Pentium 4 (SPEC2000 integer only) and a Core 2 Quad.

As in the trace experiment, some program results are missing. On the Pentium 4 processor, the 176.gcc program crashes when running with MRET2 traces and programs 197.parser and 255.vortex crash during the optimization process. On the Core2 Quad, 176.gcc also crashes when running with MRET2 traces. 173.applu stays in an infinite loop when running with MRET2 traces, 256.bzip2 has a corner case where the loop construction fails and programs 168.wupwise, 177.mesa, 191.fma3d and 301.apsi crash during the optimization process.

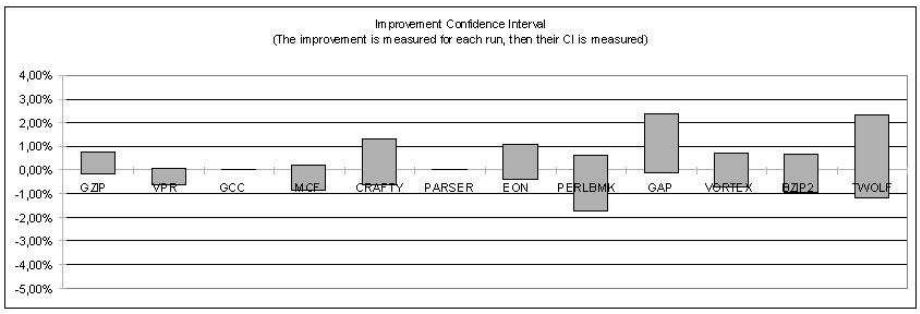

Figure 17: Optimization results on traces running on a Pentium 4

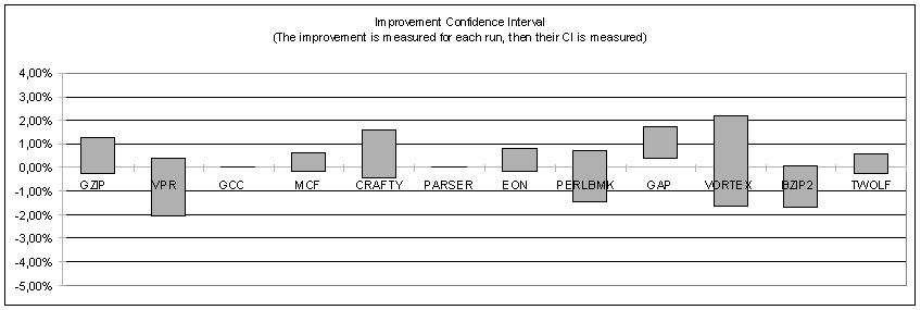

Figure 18: Optimization results on loops running on a Pentium 4

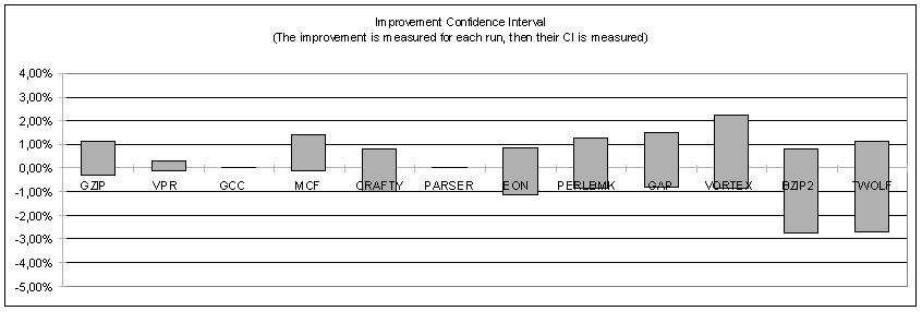

Figure 19: Optimization results on loops+traces running on a Pentium 4

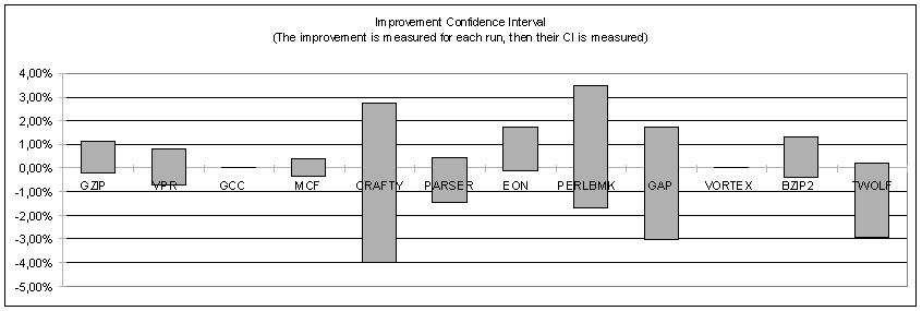

Figure 20: Optimization results on traces previously optimized by a static compiler running on a Pentium 4

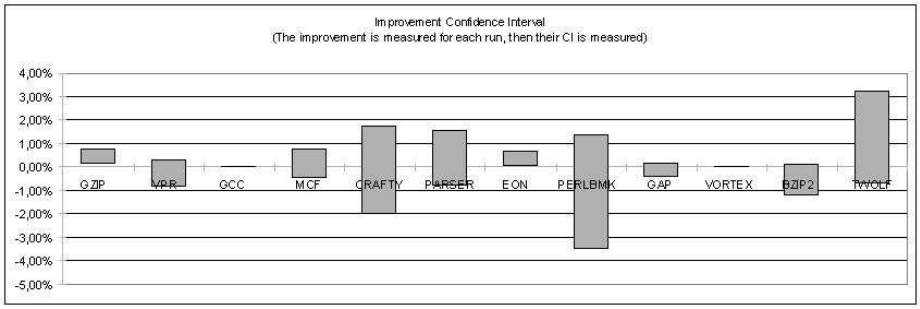

Figure 21: Optimization results on loops previously optimized by a static compiler runninng on a Pentium 4

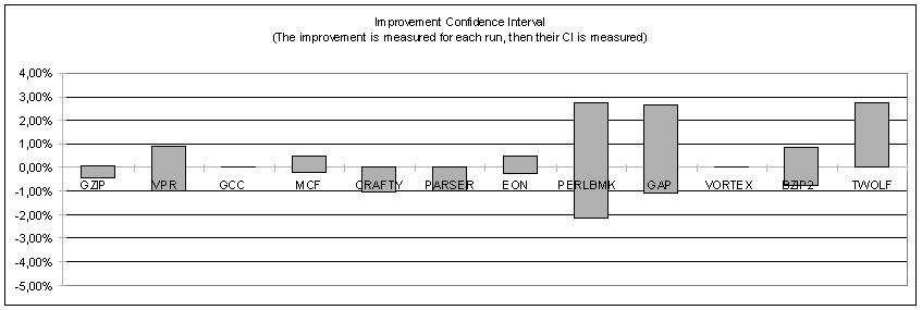

Figure 22: Optimization results on loops+traces previously optimized by a static compiler running on a Pentium 4

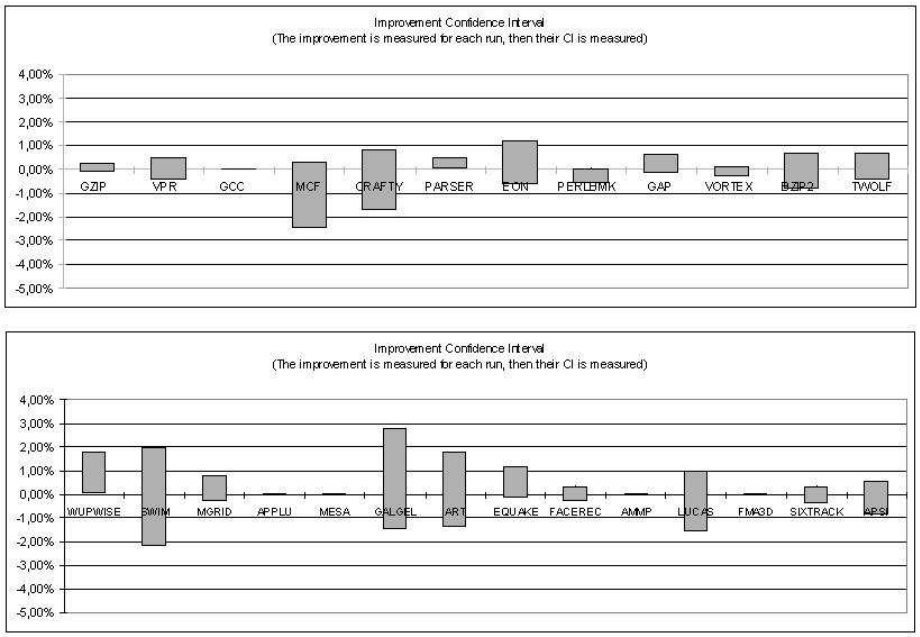

Figure 23: Optimization results on traces running on a Core 2 Quad

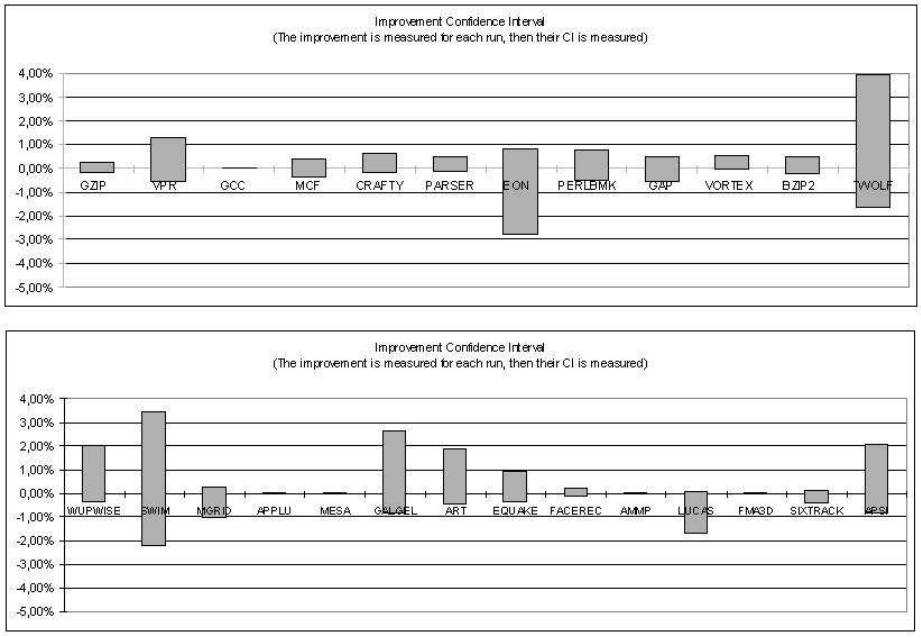

Figure 24: Optimization results on loops running on a Core 2 Quad

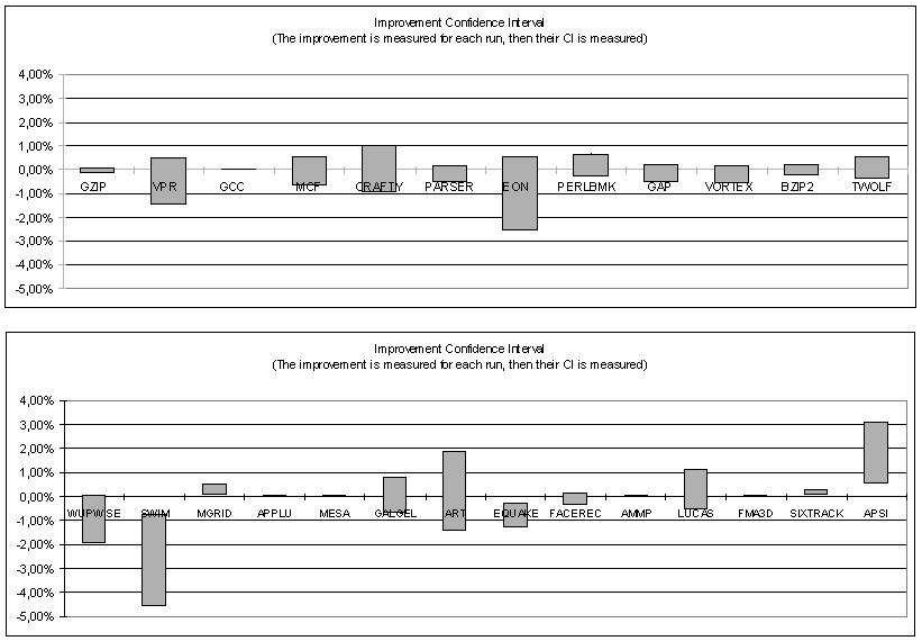

Figure 25: Optimization results on loops+traces running on a Core 2 Quad

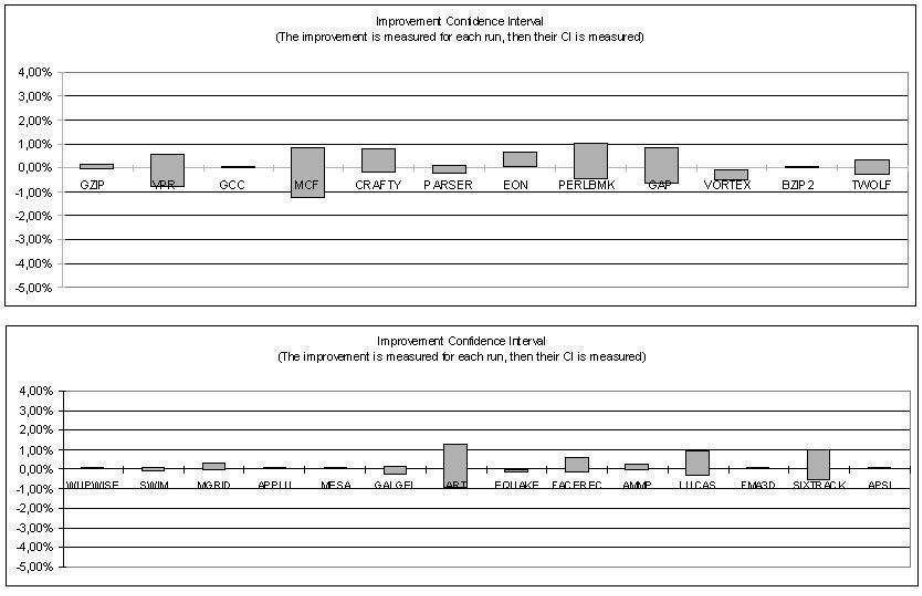

Figure 26: Optimization results on traces previously optimized by a static compiler running on a Core 2 Quad

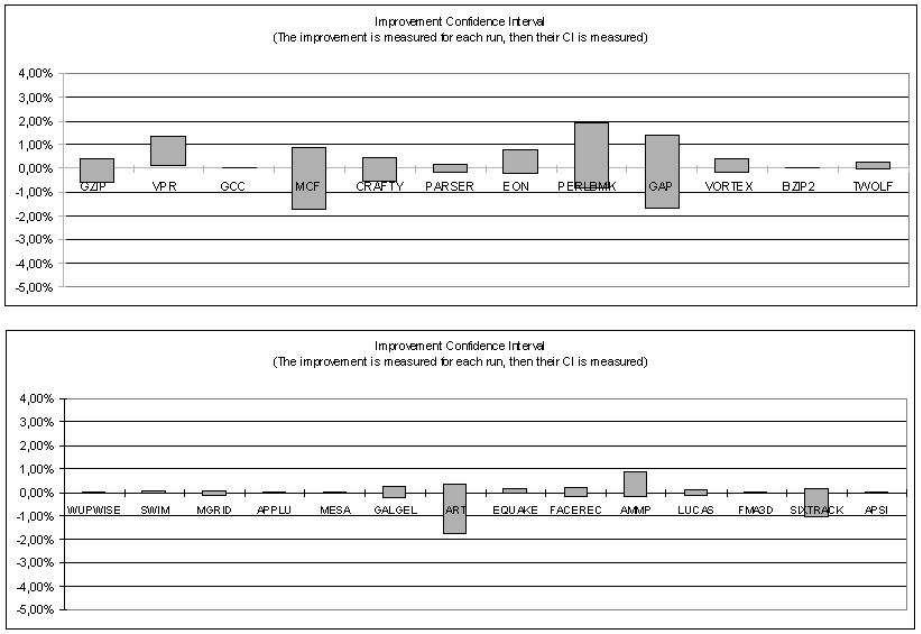

Figure 27: Optimization results on loops previously optimized by a static compiler running on a Core 2 Quad

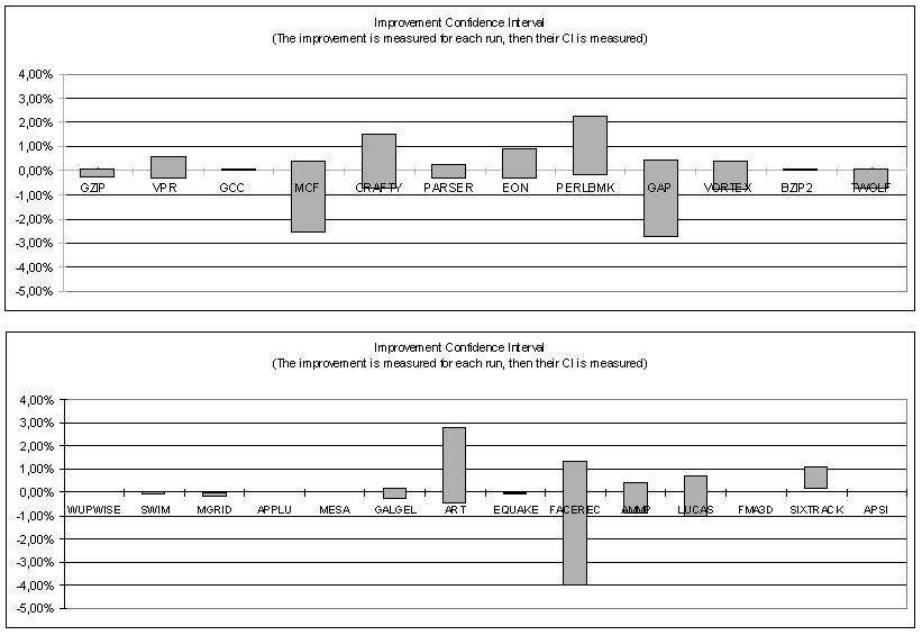

Figure 28: Optimization results on loops+traces previously optimized by a static compiler running on a Core 2 Quad

7 Conclusion

This work presented a DBT enviroment and focused on the trace detection techniques and dynamic optimization implemented on it. It first discussed three features a trace should have to be considered a good candidate for dynamic optimization. It becomes clear that is necessary to balance the trace length, the EC and the CR of a trace. Those three conceps forms the base of a good trace and there is a trade-off between them. Usually, a high lentgh decreases the CR and increases the EC; a high CR will reduce the trace length and therefor the EC; so it is a very diffcult task to build a trace long enough to be optimized: with high CR (so it does not demmand compensation code) and with high EC to be woth the time spent in optimzing it. Based on this fact, we obeserved the traces of the MRET2 technique implemented on our DBT had a low CR, since many SPEC programs has a lot of control flow. Trying to improve the trace quality, a new set of traces were built. It consists in connect traces by it's most executed side exit until a loop is formed. That way, the trace length was increased without compromising the CR, resulting in a higher EC too.

Next, three dynamic optimization implemented on the DBT were explained. DCE is a natural optimization to be implemented, since the trace construction may reveal some unecessary instructions and other optimizations like copy propagation and hole allocation may generate dead code to be removed. Our tests revealed that less than 1% of the traces instructions were removed by DCE. Function inling is also an interesting optimization, since we may have entire function calls into a trace. When it happens, all the calling overhead can be eliminated by the function inlinig. The tests shown that it's a promissing optimization, especially on the set of traces called dbt.loops.trc, where the trace legth is higher. Hole Allocation is an optimization to transform memory access into register usage. It can be very profitable when translating from architectures with different number of registers.

Our DBT enviroment permits the dump/load traces from file. This mechanism permits the evaluation of the dynamic optimization without the interference of any overhead. This is done by comparing two runs where in the first run the set of non-optimized traces are loaded and in the second run the set of optimized traces are loaded. With this metodology, we presented a large number of results. From them, we conclude that dynamic optimization at the binary level is a hard task. Even with a program that was not static optimized it is difficult to find optimization opportunities. Results show that no program had a significant improvement, the reasons to that bad result may be:

- the EC of the programs are spread in many traces, so even if a trace is optimized, it's contribution to the program speedup is minimum
- DCE demands compensation code, which minimizes the optimization gain as explained in section 4
- HA may have it results minimized by the large amount of L2 cache on actual architectures. Except for gcc, every SPEC cpu2000 program fits entirely on the L2 cache

Concluding, no DBT enviroment has shown a good dynamic optimization result until today. The kind and amount of information at the binary level does not seem to be sufficient

for the optimizations implemented in all those systems. Until now, the locality of traces in the cache is the most valuable performance gain of a DBT.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[2] E.R. Altman, D. Kaeli, and Y. Sheffer. Welcome to the opportunities of binary translation. *IEEE Computer*, 33(3):40–45, Mar 2000.

[3] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. pages 483–485, 1967.

[4] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. *SIGPLAN Not.*, 35(5):1–12, 2000.

[5] Thomas Ball and James R. Larus. Efficient path profiling. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

[6] Edson Borin, Cheng Wang, Youfeng Wu, and Guido Araujo. Software-based transparent and comprehensive control-flow error detection. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 333–345, Washington, DC, USA, 2006. IEEE Computer Society.

[7] Derek Bruening, Timothy Garnett, and Saman 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.

[8] P. P. Chang and W. W. Hwu. Trace selection for compiling large c application programs to microcode. In *MICRO 21: Proceedings of the 21st annual workshop on Microprogramming and microarchitecture*, pages 21–29, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[9] J.C. Dehnert, B.K. Grant, J.P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The transmeta code morphing/spl trade/ software: using speculation, recovery, and adaptive retranslation to address real-life challenges. *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pages 15–24, 23-26 March 2003.

[10] Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction: less is more. *SIGPLAN Not.*, 35(11):202–211, 2000.

[11] Kemal Ebcioglu, Erik Altman, Michael Gschwind, and Sumedh Sathaye. Dynamic binary translation and optimization. *IEEE Trans. Comput.*, 50(6):529–548, 2001.

[12] Kemal Ebcioglu and Erik R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *ISCA*, pages 26–37, 1997.

[13] Andreas Gal and Michael Franz. Incremental dynamic code generation with trace trees. *Technical Report No. 06-16, Donald Bren School of Information and Computer Science*, November 2006.

[14] David Hiniker, Kim Hazelwood, and Michael D. Smith. Improving region selection in dynamic optimization systems. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 141–154, Washington, DC, USA, 2005. IEEE Computer Society.

[15] J.D. Hiser, N. Kumar, Min Zhao, Shukang Zhou, B.R. Childers, J.W. Davidson, and M.L. Soffa. Techniques and tools for dynamic optimization. *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 8 pp.–, 25-29 April 2006.

[16] R. Hookway. Digital fx!32 running 32-bit x86 applications on alpha nt. *Compcon '97. Proceedings, IEEE*, pages 37–42, 23-26 Feb 1997.

[17] Bich C. Le. An out-of-order execution technique for runtime binary translators. *SIGOPS Oper. Syst. Rev.*, 32(5):151–158, 1998.

[18] J. Lu, H. Chen, P. Yew, and W. Hsu. Design and implementation of a lightweight dynamic optimization system, 2004.

[19] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 340 Pine Street , Sixth Floor, San Francisco , CA 94104-3205 , USA, 1997.

[20] T. PALMER, D. ZOVI, and D. STEFANOVIC. Sind: A framework for binary translation, 2001.

[21] D. Ung and C. Cifuentes. Optimising hot paths in a dynamic binary translator.