Abstract—Dynamic Binary Translation (DBT) has been used as an approach to transparently run code on different architectures and has generally made use of runtime information to perform effective dynamic compiler optimization. Dynamic optimization techniques are hard to design, as they have to improve code performance under stringent runtime constraints. In this paper we present preliminary work on a code optimization technique called Hole Allocation. Our goal with this paper is to introduce and highlight its potential, and to point new directions for improvement. Hole Allocation uses runtime information, collected by DBTs, to identify free register ranges (holes) which could be target of register promotion. Preliminary experiments conducted with the SPEC CPU2000 benchmark, using only one memory access pattern, shows that Hole Allocation can achieve, for some program runs, moderate speedup-ups, but also reveal that performance can suffer in some cases, and needs to be addressed.

I. INTRODUCTION

Dynamic Binary Translation (DBT) has been largely used as a tool to support a number of applications like the migration of legacy code and the transparent execution of programs across different architectures [5], [11], [19], [20], [22], [26]. DBT techniques have also been used to collect runtime information for dynamic optimizing compilers, so as to improve code quality.

Efficient DBT systems can automatically convert code to run on a new architecture without source code recompilation, providing code portability between different architectures. Moreover, DBT may enable legacy code to use new architecture features (e.g. a larger number of available registers) which were not available at the time the program was written and compiled [2].

DBT can also collect runtime information, thus enabling new optimization opportunities which were not present at compile time. Though code optimizations can improve performance, applying them at runtime can have a considerable impact on the program overall execution time. The performance of the optimized code, combined with the time spent on executing the specific code fragment, must compensate the overhead of running the optimization. To achieve that, DBT selects heavily executed program traces (i.e. hot regions) as good targets for optimizations. Optimizing traces can also be used as a promising way to overcome the overhead of translating instructions to a new ISA.

This paper is divided as follows. Section II presents DBT systems that use dynamic optimizations to improve performance. Section III describes the DBT environment used in this work, and the extensions that enable offline optimization. Section IV introduces the hole allocation optimization and show how it can be used to reduce the amount of load instructions. Section V discuss the results of applying hole allocation to the collected traces. Finally, Section VI concludes the work.

II. RELATED WORK

Dynamo [6] translates PA-8000 code to PA-8000. The program is interpreted until a hot region is found (traces, loops, etc). Optimizations are then performed on the code and the resulting trace is stored into a code cache. Optimizations performed by Dynamo achieve speedups up to 1.22x for some benchmarks. Dynamo implements redundancy removal, copy and constant propagation, strength reduction, loop invariant code motion and loop unrolling.

In UQDBT [30] a set of specifications define the binary code format, instruction syntax and semantic. The tool is also capable of identifying hot paths and to apply optimizations. UQDBT slowdown ranges between 2.5x and 7.0x when compared to native code. According to the authors, when optimizations are turned on, the slowdown is reduced by 15%.

ADORE [26] is a dynamic binary optimization framework with hardware mechanisms to help code analysis at runtime. ADORE tries to identify the most frequently executed traces, which are built with the help of dedicated registers that store the last four taken branches. Once the trace is built, several optimizations can be applied: register allocation, data cache pre-fetching and trace layout. Due to its hardware support, ADORE has a low overhead, 2%, and can achieve speedups of up to 2.56x.

Saxena and Hsu [29] developed a technique to identify more available registers so they can be used for code optimization in ADORE. Their concept of dead register is similar to our notion of dead hole. While their work is focused in gathering more registers, at the expense of inserting some overhead instructions, our work only select those registers which add no extra overhead, and re-allocate them to reduce memory accesses.

DAISY [20] is a dynamic emulator target to VLIW architectures. It is capable of translating PowerPC code to VLIW...
primitives, which is optimized to take full advantage of the target architecture. Optimizations performed by DAISY are ILP scheduling with data and control speculation, loop unrolling, alias analysis, load-store telescoping, dead code elimination among other. The final code shows good instruction level parallelism [20].

Cooper and Dasgupta [15] adapted graph coloring register allocation for JIT systems. Their goal was to minimize the time spent on building the interference graph with some impact to the allocation quality, as they generate more spill code than regular graph coloring register allocation. Their approach is complementary to our technique. Even if we apply their technique to allocate registers on DBT hot traces, we can apply our optimization technique later, given that spilling still moves a whole variable/register live range to memory. Hole allocation can revert some of these memory references as it looks for fragments of a spilled live range. In this particular case, hole allocation could bring Cooper and Dasgupta algorithm closer to regular graph coloring, helping it to revert some spills, in linear time.

III. WORKING ENVIRONMENT

In order to evaluate the Hole Allocation optimization on a DBT system, we used a modified version of StarDBT [31], [8] that runs on Linux and translates code from IA32 to IA32. StarDBT is implemented as a shared library that is loaded by the OS when programs are started. The StarDBT library takes control and executes the program via dynamic binary translation.

The StarDBT library has three main modules: the frontend, the runtime and the backend. The frontend module translates the application instructions and stores them into the code cache, controlling program execution from the code cache, and selecting hot traces for runtime optimization by the backend. The runtime module performs the communication between application, DBT and the OS, providing interfaces to I/O and system calls, handling system signals, dynamic shared objects loads and self-modifying code. Figure 1 shows the overall structure of StarDBT.

The backend module is responsible for hot trace optimizations. In order to isolate the code optimization benefits from the optimization overhead, we modified the backend module to dump hot traces into files, and to load them back in future runs [10]. The dumping process occurs at the end of the first run. StarDBT translates the program capturing all the hot traces, and, at the end of the execution, it stores the hot traces into a file. On a second run, StarDBT loads the traces from the file into the code cache, performs checksums for consistency checking, and links the traces together. This time, the StarDBT library performs little or no translations, since the hot code is already translated and placed into the code cache. Dumped traces can be optimized off-line, what enabled us to validate the optimization potential without incurring in optimization overhead.

In our experiments with the Hole Allocation optimization, we compare the execution time of the benchmarks when loading optimized and non-optimized traces. In this way, we can account for the trace loading overhead in both cases.

A. Trace Construction

In order to be worthy, code optimizations need to have a low execution overhead and generate very good code quality. Moreover, the quality of the selected traces is central in the final performance of the optimization, making trace detection techniques essential to the success of an dynamic optimization set [23]. To enable good optimization results, the desired trace should have three main features: (a) high number of instructions; (b) high Execution Coverage (EC); and (c) high Completion Rate (CR).

Trace length is important because if a trace has only one basic block it will be almost impossible to find optimization opportunities, as the static compiler has certainly applied many optimizations on that single basic block. Moreover, longer traces pass through many procedure calls, making it possible to reduce the overhead of context switching.

Execution Coverage (EC) is the percentage of time spent by the program inside the trace, that is, the trace relevancy with respect to 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 a 10% EC trace will result in a modest 5% on the overall program execution time.

A trace with many basic blocks also has many side exits (e.g. conditional branches) which define 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 its 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 executed).
Our DBT constructs three groups of traces: MRET2 [32] traces, loops by tail and loops by most executed side exit.

**MRET2:** The Most Recent Executed Tail (MRET) technique [5] associates a counter to hot spots (instructions) of the program, which are usually target at backward branches. When the counter reaches a certain threshold, the trace begins to be constructed. Trace construction lasts until a jump back to the trace entry or a stop criteria is reached. This technique is based on the idea that once an instruction is defined as a hot spot, the subsequent instructions will be hot too. In MRET counters are placed only on hotspot instructions, and once the trace is constructed, the counter can be removed. MRET2 was used to generate the traces studied on this paper. As the name says, MRET2 [32] is based on the MRET. Contrary to MRET, in MRET2, after a trace is identified, its instruction counter 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, and the result of this intersection is selected as the program hot trace. Since many programs have a lot of control flow in their hot regions, it is likely that MRET traces achieve a low CR. On the other hand, the idea of making the intersection of two runs, as in MRET2, significantly increases the CR at a cost of reducing trace length.

**Loops by Tail:** While working with MRET2 traces, we noticed that by joining some traces we could create loops. Such loops were constructed by connecting the target of the final branch of a trace to the beginning of another trace, and repeating this behavior until a loop of traces is formed. Therefore all MRET2 traces were analyzed to form loops among them.

**Loops by Most Executed Branch:** Even MRET2 traces are susceptible to low CR. In such cases, it is likely that a side exit is taken more times then the loop tail. So, the third group of traces was obtained by constructing loops as before. Nevertheless, here we look for the target address of the most executed branch (side exit and the trace tail) to form a loop. Figure 2 shows a loop constructed this way. Consider Trace 180 as the loop header. Its tail points to Trace 160, which points to Trace 161. Trace 161 has a side-exit which points to Trace 162, and this last trace has a side-exit which goes back to the initial loop header Trace 180. In the loop of Figure 2 (b), labeled Trace 100180, the side exits taken in the previous traces had their branch condition negated. For example, side-exit jge ORG 0x08048797 was turned into jnge ORG 0x0804878b.

Once a loop is found, it is not certain which trace should be considered the loop header. In fact, the constructed cycle should have many entry points. With the goal of achieving the highest EC as possible, loops were replicated, each one with a different loop header. This way, the loop built in Figure 2 was replicated three times, resulting in four loops, each with a distinct loop header (Trace 160, Trace 161, Trace 162 and Trace 180).

**IV. HOLE ALLOCATION**

In a DBT system, optimizations can compensate the overhead of the translation process, and in some cases, they can improve the final code performance.

**Hole Allocation** is an optimization that tries to remove memory accesses in the binary code. Its main goal is to re-allocate spill code to register accesses. Contrary to register coloring [12] hole allocation does not use expensive graph coloring techniques, and unlike Linear Scan [28] it does not use live range length measurements to assign registers. On the contrary, it keeps track of the registers liveness holes created at compile-time and tries to fit memory accesses into them. In some sense, it does live range splitting [16] and reallocation at runtime after register allocation.

---

**Fig. 3. Dead/Live holes in a trace from program 256.bzip2**

As expected, a good compiler tries to do the best possible usage of registers so as to generate the fastest code possible. But even in very optimized code it is possible to find optimization opportunities, in particular when considering execution trace information not available at compile-time.

For example, consider Figure 3. It shows a program trace taken from 256.bzip2 in the SPEC CPU2000 benchmark. This code was generated with the Intel Compiler using the –O3 optimization level coupled with profile information. The figure carries an analysis of the status of the processor registers at runtime. The registers under analysis are: EAX, EBX, ECX, EDX, ESI, EDI and EBP.

In the table of Figure 3, a blank space (□) indicates that register is empty, that is, it has no useful value for the program. We call this situation a dead hole. A space filled by (□□□) means that for that particular instruction, the indicated register is alive, that is, it has a useful program value, though it is not been used by that instruction. Although the register contains a useful value, it could temporarily be stored into memory, leaving the register free to be re-used by some other spilled variable, if the value is not used for a long time. This trade-off should be carefully analyzed by the allocator. We call this type
of opportunity a live hole. Symbol (\(\neg\)), in the figure, shows that the register is being used/defined by the instruction, while (\(\neg\)) means that it is not possible to determine if the register is empty or not for that instruction, but the application of cold code analysis [4] could be used to obtain more information about the liveness for this register.

Intuition says that dead holes should be few and small, mainly in the x86 architecture, as the compile-time register allocator is highly optimized, and the architecture has very few registers. Nevertheless, as shown in the figure, dead holes are not that uncommon in traces. There are dead holes in 5 registers on that figure. The EBX register has a dead hole that covers 14 out of the 22 instructions of the trace. In other words, the register is dead in 63.6% of the instructions of the registered trace. For EBP we can not determine if it is a live hole or a dead hole.

Although dead holes exist in the trace, we need to determine if they can be used for re-allocation of memory accesses. Figure 4 shows a a piece of code from program 256.bzip2 which shows that such opportunities do exist.

In that figure, instructions 6 and 10 access the same memory position. In fact, these two instructions seem to be spill code inserted by the compiler. A closer look at this code reveals an interesting behavior: The lifetime of EAX ends on instruction 3 and restarts at instruction 11, so EAX is dead in between them. This dead hole for EAX overlaps the two memory accesses at instructions 6 and 10.

The main idea behind Hole Allocation is to transform the second memory access into a register usage. To achieve that, we insert the instruction `mov eax,edx` just after instruction 6, so that EAX now has the same value of EDX and obviously the same value of memory position [esp+02ch]. We then change the instruction 10 into `mov edi,eax`. This way, register EDI is assigned the value of the EAX register, thus eliminating a memory load. Any kind of memory access

![Fig. 2. A loop built from four traces, following any possible side exit](image)

![Fig. 4. Repeated memory access](image)

![Fig. 5. Memory access optimization](image)
code. We optimize instructions that read memory, given that changing instructions that write to it may cause inconsistency between the value stored in memory and the value generated by the instruction. Remember that the memory position under optimization can be used outside the trace.

A. The Procedure

Hole allocation is a very simple optimization. First, we scan each trace for instructions writing to memory, then we look forward on the trace for one or more instructions reading this same memory position, and build its live range. The next step is to check if there is a dead hole (empty register) covering all these instructions and to perform the code transformation. If no such register is found, we try to combine several dead holes together and perform the code transformation, moving the values between holes when required, as in Figure 7.

When several dead holes can be used for optimization, we choose the smallest one which covers the desired instruction range, leaving the big ones to other ranges which may be longer. When the number of dead holes is not enough to optimize all the desired memory accesses inside a trace, we choose those which result in the largest removal of memory reads.

For the sake of simplicity in this paper we look only for memory instructions accesses which use the pattern \([\text{esp}+\text{xxxxh}]\).

The most relevant issue to be addressed by the optimizer is aliasing. The optimizer should be able to disambiguate memory references in the code that alias to the same memory position. Some pair of instructions can be easily disambiguated, other unfortunately not. As alias analysis is a very time-consuming task, our approach tries to solve only the simple, and most obvious cases.

B. Hole Allocation Side Effects

As expected, hole allocation can produce side-effects, as new load instructions are inserted into the trace to handle register re-allocation. For example, Figure 6 shows a trace after the execution of hole allocation. The left side of the figure shows the original code and the right side the optimized code.

In this trace, a range of instructions which use register EDI was identified as a dead hole and was employed to remove a memory access. Notice that after the optimization, the following sequence of code was generated: mov edi, esi followed by mov esi, edi. These instructions can be removed by optimizations like copy propagation and dead code elimination [1], [27].

A similar situation is presented in Figure 7, where two registers were used as dead holes. In a first moment, register ESI was used as a dead hole, and later register EDI. Here we use instruction mov edi, esi to move the value from one hole to another.

The optimized instruction occurs after the instruction inserted to move the value between holes. This creates a sequence of instructions that can be removed by copy propagation and dead code elimination, though we are not using them.

Figure 8 shows yet another situation. The register chosen as dead hole was ECX, the same that already contains the value store into memory, generating the instruction mov ecx, ecx. This instruction is obviously dead code and can be removed by dead code elimination.

In general, hole allocation created new opportunities for other optimizations which could result in additional improvement in the trace performance.

V. EXPERIMENTAL RESULTS

In order to evaluate the impact of hole allocation in trace optimization, we have used the SPEC CPU2000 benchmark, compiled with the Intel Compiler with profile information and -O3 optimization level.

It is important to notice that in all experiments we have only used a single hole allocation variation, namely removing memory reads in the form \([\text{esp}+\text{xxxxh}]\). Moreover, no other additional optimizations have been used to remove the extra instructions inserted by the technique.
Every SPEC program was executed and their traces were collected and optimized by hole allocation. We created three sets of traces: \textit{traces}, \textit{loops} and \textit{loops+traces}. Set \textit{traces} contains the traces generated by MRET2. Set \textit{loops} contains all traces generated by the \textit{loops by tail} and the \textit{loops by most executed branch} techniques. Set \textit{loops+traces} is the union of the two former sets.

In a second moment, sets \textit{traces}, \textit{loops} and \textit{loops+traces} were generated again, but without using hole allocation. So each trace set has two versions: one optimized by hole allocation and one not optimized.

With the traces in hand, every SPEC program was executed again, loading the generated traces from files. Each program was executed 10 times for each trace set, that is, 10 times for the \textit{traces} set optimized by hole allocation, 10 times for the \textit{traces} set without optimization, 10 times for the \textit{loops} set optimized by hole allocation, 10 times for the \textit{loops} set without optimization, 10 times for the \textit{loops+traces} set optimized by hole allocation, and finally, 10 times for the \textit{loops+traces} set without optimization. For each execu-
tion, the program execution time was collected, so we could compute the average execution time for the 10 runs of each program.

The average execution time was computed using the geometric mean. We then computed the standard deviation, using the confidence interval of 95% of the generated results. This way we could know for sure when a program achieved a performance improvement, as both the minimum and maximum improvement were above the x axis. The results are shown in Figures 9, 10 and 11.

A. Traces Set

In Figure 9 we have the results generated by the execution using the traces set. Notice that for 7 programs we had a small performance improvement, but for 2 programs we had a performance loss, for the others we cannot assure if there was any improvement or not. The highest performance improvement was achieved by 175.vpr. For this program the minimum improvement was 0.5% and the maximum was 2.1%. We can assume that 95% of the executions of the 175.vpr program with the optimized traces in the traces set will have a performance improvement ranging from 0.5% to 2.1%, though we can not say how much runs near 0.5% or 2.1%.

Program 164.gzip achieved a minimum improvement of -0.18% and a maximum improvement of -0.07%, meaning in fact that no improvement was generated for this program, but only a performance loss because both values are negative. For this program, the hole allocation optimization modified the traces in a way that execution behavior was affected.

Program 181.mcf achieved a minimum improvement of -0.9% and a maximum improvement of 2.21%, so we can not assure if hole allocation affected the execution time of this program.

B. Loops Set

In Figure 10 we show the results for the execution with the traces from the loops set. This set was built expecting to have traces with higher completion rate and execution coverage than traces built only using the MRET2 technique. In this set, we had a performance improvement only in 2 programs. For the other programs, it was not possible to conclude if we had a performance improvement or not, as all bars cross the axis.

C. Loops+Traces Set

The loops+traces experiment was created to evaluate the improvement that could be achieved by all types of traces together. Only in 4 programs we achieved performance improvement, less than in the traces set alone. The best result was in the 177.mesa program with improvements ranging from 0.68% to 2.42%. In 4 programs we had a loss of performance.

The results for the three sets show some improvements in the execution time but also some highly degraded results. The performance loss, mostly on the loops set, could be due to cache issues. The way that the loops are built to form traces may be affecting the instruction cache behavior, generating cache misses where there has no misses before the optimization.

The loops+traces set achieved improvement in 4 programs, while the traces set improved 7 programs. We also noticed that the loops+traces set generated the highest improvement (program 177.mesa), suggesting that constructing traces with loops could be profitable, but the loop construction technique must be chosen carefully.

It is important to notice that the experimental environment was very hostile to hole allocation optimization. Programs were compiled with profile information, translating from IA32 to IA32 (no extra registers) and only a single memory access pattern was under analysis. Even with these constraints, hole allocation generated performance improvement is some programs, and suggesting the existence of space for improvements. Moreover, some real world programs are compiled without profile information, which could eventually benefit from such optimization.

We have shown that when translating from IA32 to IA32, hole allocation could still find some available registers for re-allocation. This also suggests that more performance can be achieved when translating from x86 to x86-64, as more registers are available.

VI. CONCLUSION AND FUTURE WORK

Based on our experiments, we conclude that hole allocation is an optimization that can generate trace performance improvements for some programs. Better results were not achieved mostly because hole allocation is still in development, being an exploratory optimization for now. More work will be needed to understand the long term potential of the technique. A study of the effects of the insertion of new instructions on traces and how it affects the processor execution units will help to understand the results obtained in this work and how to improve them.

The side effects of hole allocation need to be better explored. The aggregation of common code optimizations with hole allocation will, probably, have an impact on the results, as the application of these optimization will remove some inserted instructions and the removal of these instructions could generate more opportunities for hole allocation to optimize other memory accesses not allowed previously, due to a lack of registers or due to short dead holes.

As future work, we intend to extend the types of memory references analyzed by hole allocation, combine it with other optimizations, perform some additional memory disambiguation and explore live holes. Also, we plan to evaluate the performance of the optimized traces in other translation environments, such as in an x86 to x86-64 translation system, which will have more registers available.

REFERENCES
