Optimized Register Usage for Dynamic Instrumentation

Nick Rutar

December 6, 2005

Abstract

Dynamic instrumentation of programs presents an interesting problem for register usage. Since program functionality can be changed at any given instruction in the program, it is imperative that register state is the same before and after the instrumentation code. The trivial solution is to spill all registers before instrumentation code is generated. This paper discusses more efficient methods for register allocation and discusses the performance gains on different architectures when using these methods.

1 Introduction

Dynamic instrumentation of programs involves the insertion of code into a running program. A primary concern when doing these operations is that the state of the registers is the same when you return from your instrumentation code. Overwriting a register in your instrumentation that is used within the original program will change the functionality of the instrumented program in unpredictable ways.

This paper examines ways to improve register usage for Dyninst[4], an instrumentation tool that provides a C++ library for dynamic instrumentation. Currently, Dyninst does a blind spill for all registers that may be affected by instrumentation. This includes registers that may be explicitly used by instrumentation, but also all caller-saved registers. In the cases where the instrumentation code is minimal, this leads to a high number of wasted memory references due to the spilling.

The paper is organized in the following matter. Section 2 gives a short overview of the DyninstAPI. Section 3 discusses the main method we have used to make the register usage more efficient, liveness analysis. Section 4 covers other, often platform specific, optimizations. Section 5 discusses performance results from these methods. Section 6 discusses related work. We summarize our conclusions in Section 7.

2 Dyninst Overview

The DyninstAPI is a means to insert code into a running program. It does this through an API that allows platform independent C++ calls that allow the programmer to create tools and applications that use runtime code patching. Implementations of Dyninst are currently available for Alpha (Tru64 UNIX), MIPS (IRIX), Power/PowerPC (AIX), SPARC (Solaris), x86 (Linux and Windows NT), AMD64 (Linux), and ia64 (Linux).

Figure 1 shows shows the structure of Dyninst. A mutator process generates machine code from the high-level instrumentation code, which was created by the calls from the API. This machine code is transferred to the application process. The new code is inserted by creating dynamic code patches, called trampolines, at the the point where the new code is

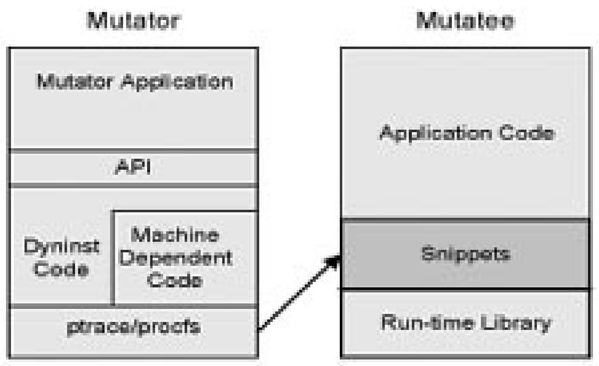

Figure 1: Structure of Dyninst

to be inserted.

The actual insertion of the code and the trampoline structure is shown in Figure 2. When a given instruction is to be instrumented, the entire basic block containing the instruction is moved to a multi-tramp structure. A basic block is a sequence of consecutive instructions where control enters at the beginning and leaves at the end.

In the original program, the first instruction of the basic block is replaced by the jump to the multi-tramp, and all other instructions are replaced by noops. In the multi-tramp, any instruction within the basic block that has been instrumented (multiple instructions may have had instrumentation added) is contained within a base tramp structure. Thus, a multi-tramp is a combination of base tramps and uninstrumented instructions that have been moved over from the original program's affected basic block. A base trampoline contains the relocated instruction from the application address space, and has slots for calling mini-trampolines before and after the relocated instruction. Before any mini-tramp call the registers are spilled and after the mini-tramp call returns the registers are restored. The mini-tramp is where the actual code generated by the mutator (a code snippet) is stored.

3 Liveness Analysis

The primary technique we are using to increase the efficiency of the register usage for Dyninst is liveness analysis at the point of instrumentation. Liveness analysis is a data flow analysis that has been used in compilers for register allocation[3]. It calculates, for a given program point, whether a variable is read before the next write update. For instance, given these 3 lines in a program,

```
1: foo = 1;
2: bar = 2;
3: foo = baz + bar;
```

At the exit of line 1, the set of live variables is $\{baz\}$. This is because bar is updated in line 2. The set of live variables at exit of line 2 is then $\{baz, bar\}$ since there are no writes to baz and bar between lines 2 and 3.

The above example performs liveness analysis on a series of instructions that would be present in one basic block. However, liveness analysis on a full program involves multiple basic blocks with registers that are live across basic block boundaries. Therefore, you need a control flow graph to look at which registers are live out for each basic block. One of the reasons that liveness analysis is an attractive choice for Dyninst, is that control flow graphs are already generated for the instrumented program for other features of Dyninst. This allows liveness analysis to be done with minimal additional overhead.

The primary task that needed to be implemented for liveness analysis to work for these platforms was the parsing and interpretation of the instructions in the program. Specifically, each instruction that reads or writes to a register needed to be correctly parsed and information about whether a register is read or written, needed to be recorded for that instruction. Limited parsing had already been established in Dyninst to handle loads, stores, and control

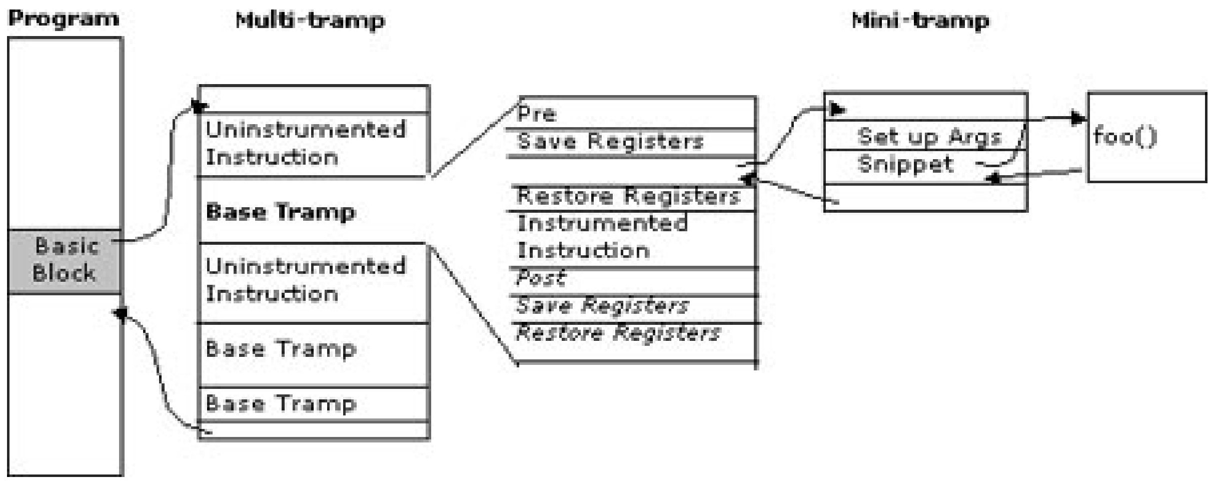

Figure 2: Instrumentation Code Insertion into a Program

instructions such as jump and branch. However, all potential instructions in the ISA (Instruction Set Architecture) needed to be successfully parsed to assure correctness for liveness analysis.

We examine the point of instrumentation and determine the registers that are live going out of the instruction(live out). If a register is dead at the instrumentation point then we know we do not have to save that register in the base trampoline. Our analysis assumes that the compiler follows the platform ABI (Application Binary Interface). The ABI for a given hardware platform is a specification for the low-level elements of the application program. It specifies such things as calling convention rules and system call details.

An important caveat to only saving live registers at the instrumentation point is that this only applies to registers that are caller saved registers. This means that the ABI for that architecture requires that when making a function call that if the register information should be maintained when the function returns, the caller of the function (as opposed to the callee) needs to spill those registers before the function call. Callee saved registers, on the other hand, are the responsibility of the called functions and any register used must be saved by the called function.

The callee saved registers, unfortunately, cannot use the liveness information at the instrumentation point to determine whether or not they need to be spilled. This is simply because, short of doing a whole program analysis, we cannot be sure of the register state of every function that calls the function that contains the instrumentation. For this reason, we must always save any callee saved register that is potentially used in our code generation.

It should be noted that every platform has a different subset of their general purpose registers available for use in the code generation of the snippet. For most platforms, this includes all of the caller saved registers, and in some cases some callee saved registers as well. The x86 architecture, for instance, has only eight GPRs (general purpose registers) so all of these registers must be available to potentially be used in code generation.

Instructions using FPRs(floating point registers) and SPRs (special purpose registers) are not explicitly generated by Dyninst at this point. However, due to the nature of instrumentation, it is possible to generate a snippet of code that calls a function that does clobber the values from floating point and special purpose registers. For this reason, all non-GPR caller saved registers must be saved on a given platform. This means that they are also available to be examined by liveness analysis.

Liveness analysis can obviously be used on any of the platforms that Dyninst currently supports. Certain architectures see more of an immediate gain than others. We've already discussed the limited number of GPRs for x86, which means that for any given instrumentation point the number of dead registers is minimal. Among the Dyninst supported architectures, Power and AMD64 made the most sense to be the first choices for the improved register use because they have relatively standard spilling procedures, and a solid number of caller saved GPRs that could be more efficiently managed in our code generation.

3.1 Power

Power was the first platform liveness was implemented on, and it was done for both the GPRs and the FPRs. The set of caller saved GPRs are $\{r0, r3 - r12\}$ and $\{r0 - r13\}$ for the FPRs[8].

3.2 AMD64

Liveness analysis was also performed on AMD64. For this platform, it was only performed on the GPRs. The set of caller saved GPRs is $\{rax, rcx, rdx, rsi, rdi, r8 - r11\}$[1]. The handling of FPRs is handled a little differently than Power, and calculating liveness for FPRs wasn't as advantageous. On AMD64, there are eight 64-bit MMX registers and sixteen 64-bit XMM registers, all of which are caller save registers. However, Dyninst does not spill each of these registers individually but instead uses the FXSAVE and FXRSTOR instructions. FXRSTOR Instructions These instructions are used to save and restore the entire 128-bit media, 64-bit media, and x87 instruction-set environment. [2] Because of this, it does not make sense to look at the individual FPRs to try to determine their liveness information. However, looking at the group as a whole and only calling the FXSAVE/FXSTOR instruction if a floating point operation occurs is an acceptable alternative as we will discuss in the next section.

4 Further Optimizations

There were other optimizations besides liveness analysis that were performed for Dyninst. Some of these are in place now and will be included in the performance results. Other features have the core functionality in place, but cannot be utilized until fundamental changes in Dyninst are implemented.

4.1 Usage based spilling

One of the more intuitive optimizations for register usage is only spilling those registers that are actually used during our code generation. Those registers that weren't clobbered during the execution of the instrumentation code would be guaranteed to have the same value as they did before the call to the trampolines.

The reason this functionality isn't already in place has to do with the way instrumentation is handled in Dyninst. Dyninst allows multiple snippets to be placed for any given instrumentation point. This is implemented by having a linked list of mini-tramps, one for every inserted snippet. The problem with this implementation from a register standpoint is that the base tramp (where the spilling takes place) is generated before the mini-tramps, so the ability to generate code on the fly multiple times at one instrumentation point limits the plasticity of the base trampoline. One option to alleviate this problem is to place the saves and restores in each mini-tramp, but in the cases where there are multiple mini-tramps linked together, the instrumentation code is going to have many redundant spills.

The next version of Dyninst is moving to an inline tramp model. Under this model, the mini-tramp(s) are going to be inlined within the base tramp as opposed to the current model which has the base tramp jump to the first mini-tramp in the linked list. With the inline model, any new snippet insertion to a point that already contains instrumentation will mean that the entire tramp structure will have to be regenerated. Since regeneration is going to occur anyway, the information about which registers were used in the code generation can be utilized by the regenerated base trampoline to only spill those necessary registers. This model trades a cleaner code generation with fewer jumps for ease of multiple instrumentation changes to the program on the fly.

The register allocation scheme for the Dyninst code generation didn't need to concern itself with which registers were allocated first when all the registers were being spilled. Spilling only the registers that are used is optimized by allocating the dead registers for the code generation first. Therefore, it is more correct to say that with this model the registers we need to spill are the union of the live caller saved registers that are clobbered, and all callee saved registers that are clobbered.

It is important to define what it exactly means for a register to be clobbered for Dyninst. The most obvious case, and the one that we have been discussing the most is the case where the code generated for a snippet explicitly uses a register. There is also the indirect case where a snippet calls a function. In that case, if we do not examine the called function all caller saved registers must be assumed to have been clobbered. Our current implementation does a linear scan of any called functions from a snippet to learn which registers have been clobbered. In the case that the called function makes another function call, we then take the conservative approach and declare all of the caller saved registers have been clobbered. The final case we have to look at for clobbering registers is internal code generated for managing the tramps. For instance, in base tramps for multithreaded applications a function call is made to assign a thread index to that trampoline. This call, thought not in a user defined snippet, has to be treated similarly to a function call from a snippet that we have just discussed.

Currently, everything discussed in this section has been implemented on both platforms we discussed for liveness analysis, and is just awaiting the inlined tramp model in the next release so the base tramp can have the appropriate information to spill the needed registers. For Power, this affects the GPR and FPR registers. For AMD64, this affects the GPR registers. Additionally, x86 (32 bit) and AMD64 will use information about the use of floating points to determine whether the FXSAVE operation is needed in the base tramp. The other platforms will follow as they are considered for developing improved register usage.

4.2 Special Purpose Registers

We touched on the fact that sometimes SPRs (Special Purpose Registers) can be caller-saved. These registers, and their usage, are often even less frequent than the floating point operations and have more complex rules involving them since there can be many instructions that implicitly affect the state of these registers. For this reason, many times Dyninst will take these registers on a case-by-case basis and develop solutions for those whose spilling can cause severe penalties.

Platform	Mutator Time not Optimized	Mutator Time Optimized	Mutatee Time no Inst.	Mutatee not Optimized	Mutatee Optimized
Power	4.38	4.38	1.17	94.95	7.62
AMD64	0.74	0.73	0.25	6.69	5.85

Table 1: Performance Results (in seconds)

One special purpose register which Dyninst explicitly addresses is the MQ register for Power (also referred to in documentation as SPR0). It mainly deals with multiply and division operations. For multiply, it stores bits $32-63$ of the result. For division, the MQ register is used to store the remainder from the operation[8]. There are additional operations that use the MQ register explicitly but they are rarely used. Each spill of this register can take multiple cycles so it was important to only spill it when necessary.

We didn't perform full liveness to determine the use of this register since its use is fairly rare in standard applications. Instead, we examine the basic block that contains the instrumentation point to see if there are any instructions that modify the MQ register. If one of these instructions are present, we save the state of the register prior to instrumentation.

5 Performance Results

For the Power results, the experiments were performed on an IBM p670 with 8 POWER4 processors, running AIX 5.2 with 8.0 gigabytes of memory. For the AMD64 results, the experiments were ran on dual AMD Opteron processors running Linux 2.4 with 2.0 gigabytes of memory.

These tests were ran on a small program that calculates a sum based on the control flow path that was dependent on the loop index. The instrumentation was placed in the first instruction from each basic block in the control flow graph of the program. This means literally every instruction in the program is relocated to respective multitramps even though there are not representative base tramps for every instruction. The snippets for each instrumentation point were a few instructions each, and had the functionality of incrementing a global variable found in the mutatee program.

The instrumented portion of the small program we tested against is listed below.

```
int a, x = 0;
for (int a = 0; a < 0xffffff; a++)
{
        x = x +a;
        x += 5  * a;
        if (x > 6000)
              x /= 2;
        else
              x *= 4;
}
```

The small program above is meant to be simple display of the amount of time that can be saved in a program with a majority of its instructions instrumented. Obviously, larger programs with less instrumentation might see less of a percentage improvement in time because the uninstrumented portions of the mutatee could outweigh the instrumented portions (and overshadow any improvement).

Table 1 displays the results of the test. For Power, the optimized results include using liveness analysis for GPRs, FPRs, and limited usage of the MQ

SPR. For AMD64, the optimized results include using liveness analysis for the GPRs.

The mutator time is the amount of time it takes for Dyninst to load the run-time library and do the actual instrumentation on the original program. The mutator time is relatively unchanged regardless of the implementation. This is because Dyninst has always calculated the control flow information that liveness takes advantage of so most of the work has already been done for us.

The mutatee time is the length of time the instrumented program takes to run. We have given the running time of the program before instrumentation and the time after instrumentation with the optimized and unoptimized approaches. The mutatee time sees a significant speedup for Power because of all the different types of registers that are examined. Much of the speedup can be attributed to the use of the MQ register sparingly. AMD64 also sees a noticeable speed up even though it is only examining GPRs.

6 Related Work

Various instrumentation tools have their own methods for register usage. EEL[6] is a library for building tools to modify an executable program, and regenerate a new executable that contains those modifications. EEL differs from Dyninst in the fact that it does offline instrumentation, while Dyninst attaches to a running program. Because of these differences, EEL is able to generate code much like a compiler, without the issues Dyninst runs into with its dynamic approach. While designed for multiple platforms, EEL was never implemented on anything other than the SPARC architecture. EEL uses liveness analysis and register scavenging[5] to try to utilize unused registers in the snippets. If no unused registers are available, it spills the needed number of registers.

Another instrumentation tool is ATOM[9], which does binary instrumentation on Alpha. ATOM takes the conservative approach of saving all caller-save registers at each instrumentation point. At each instrumentation point, it makes calls to analysis routines. However, instead of going straight to the analysis routine the call goes instead to a wrapper routine that saves the needed registers, calls the analysis function, restores the proper registers, and returns back to the instrumentation point in the original code. This cuts down on code size added by instrumentation, but also means that there is no optimizations based on which registers are actually used since the full set of caller-save registers are always spilled.

Finally, Etch[7] is an instrumentation tool which operates on Win32/x86 binaries. The literature on this tool does not include specifics of the register usage in its instrumentation. Etch claims that instrumentation ”should not change program correctness” which means at least a conservative spilling would have to be done, if not a more detailed analysis.

7 Conclusions

Improved register usage is an excellent way to improve performance for instrumentation in programs. This paper has examined different methods for improving register usage in one such instrumentation program, Dyninst, which is a platform independent API for the dynamic instrumentation of running programs. The primary method examined was using liveness analysis on architectures where there could potentially be a number of dead registers that would not need to be saved. Other methods were examined based on platform specific register usage, such as certain special purpose registers that have high penalties for spilling. Also discussed were features that would need core changes to the way Dyninst inserts instrumentation in order to be integrated.

The preliminary results based on the architectures that these optimizations have been implemented on have been promising. Architecture specific register usage improvements will continue to be looked at, as will the propagation of the techniques discussed in this paper to the rest of the architectures supported by Dyninst.

References

[1] *AMD64 Architecture Programmers Manual Volume1: Application Programming*, September 2003.

[2] *AMD64 Architecture Programmers Manual Volume2: System Programming*, September 2003.

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

[4] Bryan Buck and Jeffrey K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[5] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. Technical Report CS-TR-92-1083, Madison, WI, USA, 25 March 1992.

[6] James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–300, 1995.

[7] Ted Romer, Geoff Voelker Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian N. Bershad, and J. Bradley Chen. Instrumentation and optimization of Win32/Intel executables using etch. pages 1–8.

[8] *AIX Assembler Language Reference, Version 4*, October 1996.

[9] Amitabh Srivastava and Alan Eustace. Atom: a system for building customized program analysis tools. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 196–205, New York, NY, USA, 1994. ACM Press.