Issues and Support for Dynamic Register Allocation*

Abhinav Das, Rao Fu, Antonia Zhai, and Wei-Chung Hsu

Department of Computer Science,
University of Minnesota
{adas, rfu, zhai, hsu}@cs.umn.edu

Abstract. Post-link and dynamic optimizations have become important to achieve program performance. A major challenge in post-link and dynamic optimizations is the acquisition of registers for inserting optimization code in the main program. It is difficult to achieve both correctness and transparency when software-only schemes for acquiring registers are used, as described in [1]. We propose an architecture feature that builds upon existing hardware for stacked register allocation on the Itanium processor. The hardware impact of this feature is minimal, while simultaneously allowing post-link and dynamic optimization systems to obtain registers for optimization in a "safe" manner, thus preserving the transparency and improving the performance of these systems.

1 Introduction

The dynamic nature of languages and dynamic program behavior has increased the importance of post-link and dynamic optimization systems. Many such systems have been proposed in the past[4][5][6][11][12]. To deploy optimizations at post-link or run time, these systems need registers. Register acquisition, which, in the context of post-link and dynamic optimization broadly includes obtaining extra registers for optimization, is challenging due to several reasons: (i) compiled binaries have already performed traditional register allocation that tried to maximize register usage; (ii) control and data flow information, which is necessary for performing register allocation, may not be accurately known from analysis of binary. At runtime when code is seen incrementally, flow analysis is more restricted. Thus, there is no efficient software solution for acquiring registers for post-link time optimization, and even more so for dynamic optimization. Software support, such as compiler annotations, and architecture/hardware support, such as dynamic stack register allocation, can potentially ease post-link and runtime register acquisition.

The requirements of register acquisition for post-link and dynamic binary optimization systems are different from traditional and dynamic compilation models. Since register allocation has already been performed, such systems have

* This work is supported by a grant from NSF EIA-0220021.

C. Jesshope and C. Egan (Eds.): ACSAC 2006, LNCS 4186, pp. 351–358, 2006.

to make very conservative assumptions about register usage. Dynamic binary optimization systems face the additional burden of finding registers with minimal runtime overhead. Post-link time optimization systems do not have a time constraint, since analysis is done off line. Traditionally, these systems rely on binary analysis to find registers that are infrequently used. These registers are freed for optimization by spilling them. For architectures that support variable-size register windows, these optimization systems increase the size of register window to obtain registers. Unfortunately, these software-based schemes make assumptions about code structure that can be easily broken. In this paper, we make the following contributions: (i) Briefly describe existing register acquisition schemes in post-link time and dynamic optimizers, such as Ispike[5] and ADORE[2][4] (ii) Present an architecture feature that enables the use of variable-size register windows to dynamically obtain required registers. In the context of this paper, register allocation means register acquisition as described above.

2 Software-Based Register Allocation

2.1 Fixed-Number Register Allocation

Registers can be allocated statically with help from the compiler or from hardware. The compiler can be instructed to not use certain general purpose registers, which can be later used by dynamic optimization systems. Similarly, hardware can be implemented to allow the use of certain registers only for specific purposes. If the compiler is used for static register allocation, compiler support must be available from compilers and some form of annotation must be provided for the dynamic optimizer to differentiate between supported and unsupported binaries. Hardware can support a fixed number of registers (shadow registers) for optimization. [1] presents a detailed explanation of the limitations involved in using a fixed number of registers. Lu et al. in [4] showed that allocating registers dynamically can significantly improve the performance of runtime optimization systems.

2.2 Dynamic Register Allocation

Register Spilling: Registers used for dynamic optimizations can be obtained by first scanning the optimized trace (a single-entry multiple-exit region of code) to find unused registers, then spilling and restoring these registers at trace entry and exit, respectively. The main challenge in register spilling is where to spill registers. Possible choices are (i) on stack top (ii) on the heap (thread shared or thread private). Each of these has limitations detailed in [1].

Dynamically Increase Variable Register Window Size: The size of frame (of the function to be optimized) is incremented by executing a copy of the *alloc[15]* instruction for the current frame with an increased frame size. Figure 1, shows how extra output registers are dynamically allocated for optimization.

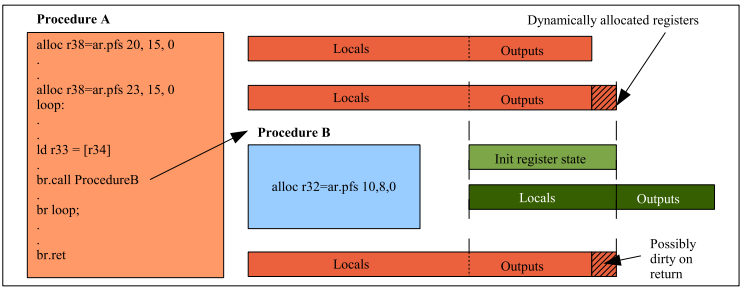

Fig. 1. Mechanism of dynamic register allocation using *alloc* instruction on the IA64 architecture[15]

However, there are some limitations to this approach. It is difficult to find the *alloc* instruction for the register frame of code to be optimized. Scanning the binary may lead to detection of an incorrect *alloc* instruction due to compiler optimizations, such as function splitting. Aggressive compilers use multiple *allocs* to allocate different number of registers down different paths. The presence of multiple *allocs* may lead code scanning in finding the wrong *alloc*. Leaf routines may not have an *alloc* instruction at all, and they use registers from the caller's output frame and global registers. These are described in detail in [1].

Since static scanning has limitations, a dynamic mechanism is needed to find the state of the current register stack. This state information is encoded in an architecturally invisible register called *current frame marker* (CFM). On a function call contents of this register are copied to an architecturally visible application register called *previous function state* (ar.pfs). To determine the register stack state dynamically, we can inject a function call just before trace entry. The injected function call then reads the *ar.pfm* register and passes this value to the dynamic optimizer. Possible methods of injecting a function call are (i) inserting a call instruction before trace entry and (ii) generating a trap by inserting an illegal instruction, for example. For reasons discussed in [1] we need to generate a trap, which has a high overhead (as much as 1789%).

3 Architecture Support

Since the main limitation of existing support is the absence of fast access to current register state information, an easy architecture extension is to expose the CFM register and thus providing the current state information. Doing so, reduces the overhead of finding current state, but does not reduce the overhead of determining if the state is the same as when the trace is generated. If the state has changed, traces need to be regenerated that can result in substantial overhead. With this limitation in mind, we sought features that were free from the above limitations and provided a fast and easy way to obtain additional registers.

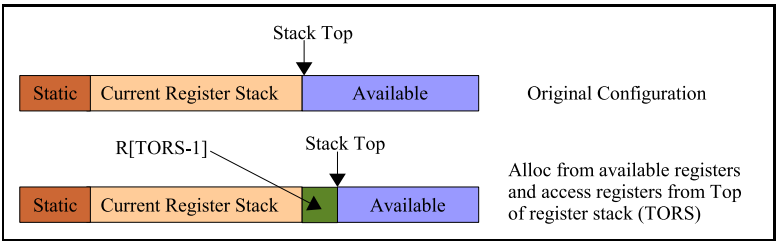

Fig. 2. Main idea of dynamic register allocation

3.1 Main Idea

The main requirements of allocating registers dynamically is a scheme that uses relative addressing to access registers. If we use a fixed method to access registers, compilers can use those registers, and we will have no guarantee of using those registers post-compilation. A disadvantage of fixed register schemes is that optimizations cannot be stacked onto one another. A scheme that can allocate registers on top of existing allocation is better suited to runtime optimization, or in general, incremental post-link optimization. Figure 2, shows the main idea of this approach. In this figure, a new instruction can allocate some number of registers from available architectural registers by only specifying the number of registers needed rather than specifying complete frame information (as is needed in the current implementation of *alloc* instruction). The other aspect shown in the figure is the access mechanism for these registers. Registers are accessed relative to the Top of Register Stack (ToRS). Let us consider some cases and explain how dynamic register allocation will work with such a mechanism, even if the compiler uses this scheme for allocating registers. Suppose that post-link, we want to optimize a loop that already addresses some registers using the top of register stack. Let us assume that optimization requires 2 dynamic registers. We can use a new instruction to increment the size of frame by 2 and adjust the relative offset of instructions already using relative addressing by 2. Note that only those instructions that are in the trace need to be modified, as the state of register frame would be restored upon trace exit. We need another instruction to restore the register frame. Thus, a stack-based approach coupled with relative addressing from top of stack can be effectively used for dynamic register allocation.

3.2 Relative Addressing

In this section, we will discuss implementation details of the relative addressing scheme proposed. There are 128 registers in the IA64 ISA and a 7-bit field is used to address these registers. In the simplest form, a bit can be added to each 7-bit register entry that distinguishes regular access from relative access. We would add an extra bit for each register field thereby increasing the instruction width from 41 bits to 44 bits (maximum of 3 register fields in an instruction).

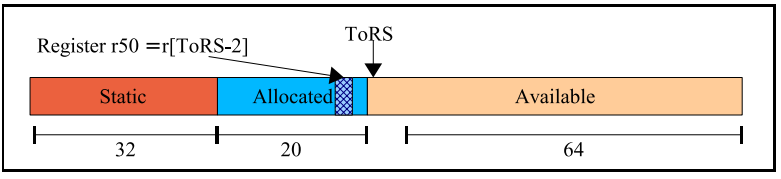

Fig. 3. Case example showing stack allocation of 20 registers highlighting the case where total number of registers accessed is less than 64

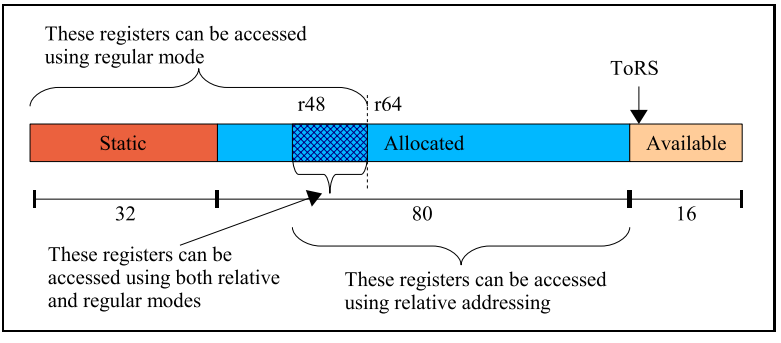

Fig. 4. Example with 80 registers are allocated on the stack highlighting the case where total number of registers accessed is greater than 64

However, we can use a clever trick to ensure that the register address field width is not increased. In this scheme, the 7^{th} bit is used to distinguish between regular access and relative access. Since we are left with only 6 bits for indexing into registers, let us discuss how 128 registers can be accessed using this addressing scheme.

Case 1: Number of Registers allocated by compiler $\leq$ 64: In this case (Figure 3) all registers can be accessed by both relative and regular access. The number of registers (static and stack) total to less than 64. Thus, they can be accessed by both modes. For regular access the 7^{th} bit is set to zero. If the compiler wants to use relative accessing it is free to do so. In the example shown, 20 stack registers are allocated along with 32 static registers. Register *r51* is the last allocated register which can be accessed as *r51* or as *ToRS[-1]* as the ToRS points to *r52*. The compiler should encode this field as 1000000_2.

Case 2: Number of Registers allocated by compiler > 64: In the example shown in figure 4, the compiler has to use regular mode for registers *r0* to *r47*. Registers *r48* to *r63* can be accessed using regular mode and ToRS mode (*ToRS[-64]* to *ToRS[-49]* respectively) and registers *r64* onwards have to be accessed using relative addressing (*ToRS[-48]* onwards).

Thus, the addressing mode is implemented such that the width of register field remains the same. In the extreme case, when all 128 registers are allocated,

```
inc_alloc imm7
  imm7 - a 7-bit immediate field used to specify the amount to increment the frame by
Operation:
  if(cfm.sof+imm7) > 96 then Illegal_operation_fault() else cfm.sof += imm7

dec_alloc imm7
  imm7 - a 7-bit immediate field used to specify the amount to decrement the frame by
Operation:
  if(cfm.sof-imm7) < 0 then Illegal_operation_fault() else cfm.sof -= imm7
```

Fig. 5. Instruction format for inc_alloc and dec_alloc instructions

r0 to *r63* are accessed using regular mode and *r64* to *r127* are accessed using *ToRS[-64]* to *ToRS[-1]*, respectively. Since, some registers can be accessed by both the modes, we must be careful when we increase the size of register frame, as it may so happen that some register that was accessed using ToRS mode now has to be accessed via direct mode. As an example, let the initial frame have 64 stack registers and the first stack register (*r32*) is accessed using ToRS mode. If the size of frame is increased by, say, 5 registers, then the access of this register would have to converted into direct mode. Since this involves knowing the current frame state, optimizers can choose to bail out when such conditions exist.

Since a register access on Itanium already performs an indexing operation to access the correct physical register, we believe our implementation does not add to the hardware cost. To reduce the cost of subtracting the offset, the top of register stack can be maintained as part of the current frame in the CFM register.

3.3 New Instructions

Some new instructions must be added to manage dynamic register stack in a way which is slightly different from the *alloc* instruction. The aim is to leverage existing hardware. We add two new instructions for increasing and decreasing the register stack. The first instruction is *inc_alloc* that increments the current register frame size (cfm.sof) by a number specified in the instruction. The second instruction is *dec_alloc* that decrements the sof value in cfm. The format and operation of these instructions are shown in Figure 5.

4 Related Work

Dynamic binary translation poses similar challenges to register acquisition. Register allocation in translation involves mapping source (i.e. the binary to be translated) registers to the target (i.e. the host) machine's registers. Shade[9][10] is a dynamic translation and tracing tool for the SPARC platform. It translates SPARC (v8 and v9) and MIPS 1 instructions for SPARC v8 systems. For native translation, the virtual and the actual number of registers are the same. Since some registers are needed by SHADE, registers in the translated program are

remapped to different physical registers and registers are spilled lazily. PIN [13] is a instrumentation tool for 4 architectures IA32, EM64T, IA64 and ARM. It performs native translation for each of these architectures and does register reallocation and liveness analysis. PIN builds register liveness incrementally as it sees more code. When traces are linked, PIN tries to keep a virtual register in the same physical register whenever possible. If this is not possible, it reconciles differences in mapping by copying registers through memory before jumping to another trace. Probst et al. [8] discuss a technique for building register liveness information incrementally for dynamic translation systems.

Post-link optimizers such as SPIKE[6] and Ispike[5] optimize binaries by analyzing profile information. SPIKE needs registers for inserting instrumentation code. It uses register usage information collected by scanning the binary to find free registers, so that register spills can be minimized. Ispike collects profile from hardware counters on the Itanium platform and thus it does not require registers for collecting profile. However, data prefetching optimization requires registers. Ispike uses either free registers by liveness analysis, increments register stack (by looking for *alloc* instruction) or uses post-increment/decrement in prefetch and load operations.

Dynamic optimizers can be similar to dynamic translators if they use translation to build traces. Dynamo [11] is a dynamic optimizer for PA-RISC binaries. When emitting traces to be optimized, Dynamo tries to create a 1-1 mapping between virtual and physical registers. For registers that cannot be mapped, it uses the application context stored in the translator to store the physical registers. DynamoRIO [12] is a system based on Dynamo for x86 systems. ADORE [2][3][4] as described earlier uses *alloc* instruction or spills registers to obtain registers for optimization. Saxena in [14] describes various issues of register allocation for the ADORE system and presents data for finding dead registers in a trace. Jesshope in [17] uses register access by relative addressing to communicate dependencies between variables in dynamically parallelized code. Relative access to registers is not a new idea, but one that is already implemented in Itanium. Our contribution is to provide another base for accessing registers, to tackle the specific problem of register acquisition for post-link and dynamic optimization.

5 Conclusion

Register allocation for post-link and dynamic optimization systems poses interesting challenges as correctness, overhead and transparency are important concerns. In this paper, we have presented a modest hardware addition to the IA64 architecture, as an example, to illustrate how such a feature would simplify dynamic register acquisition. The proposed hardware support ensures correct execution while imposing no performance overheard and transparency limitations. When multiple post-link and dynamic optimizations are present, the proposed hardware allows optimization systems to stack their optimizations on top of each other. The architecture feature described, leverages existing hardware on the Itanium processor, thus will likely be feasible. We believe that given the

performance delivered by post-link and dynamic optimization, it will be cost-effective to devote more hardware resources for this purpose.

References

1. Das, A., Fu, R., Zhai, A., Hsu, W.-C.: Issues and Support for Dynamic Register Allocation. Technical Report 06-020, Computer Science, U. of Minnesota, 2006
2. Lu, J., Das, A., Hsu, W-C., Nguyen, K., Abraham, S.: Dynamic Helper-threaded Prefetching for Sun UltraSPARC Processors. MICRO 2005.
3. Jiwei Lu, Howard Chen, Rao Fu, Wei-Chung Hsu, Bobbie Othmer, Pen-Chung Yew: The Performance of Runtime Data Cache Prefetching in a Dynamic Optimization System. MICRO 2003.
4. Jiwei Lu, Howard Chen, Pen-Chung Yew, Wei Chung Hsu: Design and Implementation of a Lightweight Dynamic Optimization System. Journal of Instruction-Level Parallelism, Volume 6, 2004
5. Chi-Keung Luk, Robert Muth, Harish Patil, Robert Cohn, Geoff Lowney: Ispike: A Post-link Optimizer for the IntelItaniumArchitecture. CGO, 2004.
6. R. Cohn, D. Goodwin, P. G. Lowney and N. Rubin: Spike: An Optimizer for Alpha/NT Executables. Proc. USENIX Windows NT Workshop, Aug. 1997.
7. David W. Goodwin: Interprocedural dataflow analysis in an executable optimizer. PLDI, 1997
8. M. Probst, A. Krall, B. Scholz: Register Liveness Analysis for Optimizing Dynamic Binary Translation. Ninth Working Conference on Reverse Engineering(WCRE'02).
9. Cmelik, B. and Keppel, D. 1994: Shade: a fast instruction-set simulator for execution profiling. SIGMETRICS Perform. Eval. Rev. 22, 1 (May. 1994), 128-137.
10. Robert F. Cmelik and David Keppel: Shade: A fast instruction-set simulator for execution profiling. Technical Report 93-06-06, CS&E, U. of Washington, June 1993
11. Bala, V., Duesterwald, E., and Banerjia, S: Dynamo: a transparent dynamic optimization system. PLDI, 2000.
12. Bruening, D., Garnett, T., and Amarasinghe, S: An infrastructure for adaptive dynamic optimization. CGO, 2003.
13. Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., Hazelwood, K.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. PLDI, June 2005.
14. Saxena, A., Hsu, W.-C.,: Dynamic Register Allocation for ADORE Runtime Optimization System. Technical Report 04-044, Computer Science, U. of Minnesota, 2004
15. Intel®Itanium®Architecture, Software Developer's Manual, Volume 1, 2 and 3: http://www.intel.com/design/itanium/manuals/iiasdmanual.htm.
16. UltraSPARC™III Processor User's Manual: http://www.sun.com/processors/-manuals/USIIIv2.pdf
17. Jesshope, C. R.: Implementing an efficient vector instruction set in a chip multiprocessor using micro-threaded pipelines. Proc. ACSAC 2001, Australia Computer Science Communications, Vol 23, No 4., pp80-88