Exploiting Hardware Resources: Register Assignment across Method Boundaries

Ian Rogers, Alasdair Rawsthorne, Jason Souloglou
The University of Manchester, England
{Ian.Rogers,Alasdair.Rawsthorne,Jason.Souloglou}@cs.man.ac.uk

Abstract

Current microprocessor families present dramatically different numbers of programmer-visible register resources. For example, the Intel IA32 Instruction Set provides 8 general-purpose visible registers, most of which have special-purpose restrictions, while the IA64 architecture provides 128 registers. It is a challenge for existing code generators, particularly operating within the constraints of a just-in-time dynamic compiler, to use these varying resources across a number of architectures with uniform algorithms. This paper describes an implementation of Java using Dynamite, an existing Dynamic Binary Translation tool. Since one design goal of Dynamite is to keep semantic knowledge of its subject machine localized to a front-end module, the Dynamite code generator ignores method boundaries when allocating registers, allowing it to fully exploit all hardware register resources across the hot spots of a Java program, regardless of the control graphs represented.

1. Introduction

Current microprocessor families present dramatically different numbers of programmer-visible register resources. For example, the Intel IA32 Instruction Set [1] provides 8 general-purpose visible registers, most of which have special-purpose restrictions, while the IA64 architecture [2] provides 128 registers. In the former case, register renaming and out-of-order issue is used in the microarchitecture to exploit a richer resource set than indicated by the instruction set, but the paucity of the visible instruction set remains a severe constraint on a just-in-time code-generator. In particular, it is difficult to pass method parameters efficiently in registers in x86 implementations, since register pressure ensures that the lifetimes of values in registers are so short.

In the case of RISC, and particularly EPIC [3] architectures, the visible instruction set more closely models the register hardware resources provided in an implementation. Register assignment becomes viable using a conventional approach, such as defining a method calling sequence, involving caller-saved, callee-saved and parameter registers, and allocating local variable and temporary registers within individual methods.

In the distant future, it is possible that unconventional CPU architectures may provide extremely high levels of performance without any significant number of registers.

This disparity of hardware resources has been addressed by designers of current Java Virtual Machines by providing different register allocation algorithms for (e.g.) IA32 and RISC machines. In this paper, we describe a single register allocator appropriate to register-rich and register-poor architectures, and we explain how this allows us to set optimization boundaries independently of method boundaries, giving performance advantages.

Conventional static compilers, and existing JIT compilers, use method-inlining, more or less aggressively, to uncover a number of optimization possibilities, with register assignment among them. Inlining may have its limitations, however, and we have found a number of cases where inlining (particularly leaf-method inlining) does not completely address hot regions of an application.

In this paper, we present some techniques we use to run Java byte-coded programs on Dynamite, our existing dynamic binary translation environment. As will be described below, our techniques rely on optimizing regions of code without reference to the semantic boundaries of methods. Broadly, we claim to gain the performance benefits of inlining, without its limitations.

We describe the Dynamite binary translation system, its interfaces and its approach to optimization. In section 4, we show how Java programs are implemented using the Dynamite facilities, and section 5 discusses our preliminary results. Previous work in register assignment for Java programs is introduced in section 6.

2. Dynamite

Dynamite is a reconfigurable Dynamic Binary Translation system, whose aims are to extend the "Write Once, Run Anywhere" paradigm to existing binary applications, written and compiled for any binary platform. To enable the quick configuration of a particular translator, Dynamite is constructed as a three-module system, as shown in figure 1.

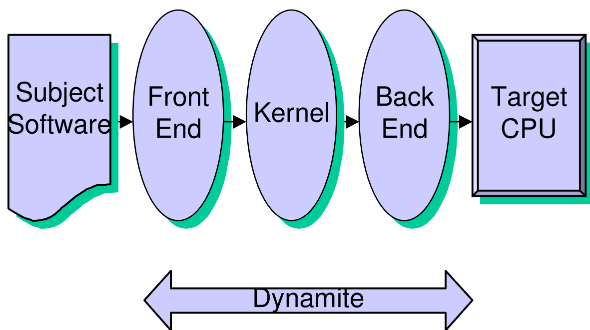

Figure 1. Dynamite Structure

The function of the major components is almost self-explanatory: the Front End transforms a binary input program into an intermediate representation (IR), which is optimized by the Kernel, and the Back End generates and executes a binary version on the target processor. The Front End interface supports a number of abstractions convenient for efficient Front End implementation, such as the "abstract register", which holds intermediate representations of the effects of subject instructions. This interface is configured by an individual Front End module, since different subject architectures require different numbers of registers.

Two features of this front end interface are relevant to the current discussion: firstly, the interface resembles a RISC-like Register Transfer Language, containing no peculiarities (such as condition codes or side-effects) adapted for particular subject architectures. Secondly, procedure (method) calling is achieved at a primitive level, typically by having the front end create IR to compose a link value in subject state space, and then branching or jumping to the callee. Return from a procedure is similarly implemented by having the Front End create IR which causes a jump to be made to the return address. Parameter passing and stack-frame management is implemented by having the Front End create the appropriate IR to model the subject architecture's requirements.

The Kernel contains about 80% of the complexity of the translator system. It creates and optimizes IR in response to Front End calls, and invokes the Back End to code generate and execute blocks of target code. Register assignment for the target machine code generator currently takes place within the Kernel, again using a Back End interface that is parameterized for specific target architecture. Optimization is performed adaptively at a number of different levels, starting with initial translation as described below.

To achieve its performance goals, Dynamite operates in an entirely lazy manner. An instruction is never translated until that instruction must be executed, either because it is a control (jump, branch, exception or call) target, or because the immediately preceding branch has fallen through. As instructions are decoded by the Front End, their IR is combined until a control transfer is encountered. During this process, the kernel performs optimizations such as value forwarding and dead code elimination. When the block of IR is complete, it is code generated, executed by the back end, and cached for subsequent reuse. After a block of target code is executed, its successor location may either be found within the cache, or may need translation using the same actions.

In efficiency terms, target code blocks generated using this initial scheme leave something to be desired. The benefit is that the initial translation is quick, taking only a few thousand instructions per subject instruction. Register usage is determined by an individual Back End, largely as a result of the method calling sequence mandated by the static compiler used to compile Dynamite. Target registers are used to store temporary results within the block, but existing Back Ends preserve all subject register values in target memory at the boundary of basic blocks.

More optimization and higher quality code generation are triggered when an individual target block is executed more frequently than a dynamic execution threshold. This event causes the kernel to create a group containing this and related blocks in a hot region, and to optimize this Group Block as a single entity. Group Blocks may span arbitrary boundaries in the subject machine: indeed, in other applications, Dynamite optimizes across programs and their procedures, static and dynamically-linked libraries, OS Kernels, and across different virtual machines.

Within a Group Block, the Dynamite kernel examines the existing control flow of the region, identifying certain blocks as entry and exit blocks, and performing value propagation and dead-code elimination across the entire group. The control flow is used to straighten the conditional branches and eliminate jumps within the group, so that frequent cases fall through, minimizing taken branches and maximizing I-cache utilization.

Code generation for a Group Block occurs next. To avoid the expense of an iterative algorithm, a very simple incremental register allocation algorithm is used. Starting with the target block, operands are allocated to registers (if the target machine architecture requires), and operation results are allocated to registers if they are to be reused. As the register set is exhausted, spill code is generated to relinquish previously allocated registers for new operations. Register allocations are carried across basic block boundaries, and the act of code generating from the most- to the least-frequently executed blocks within the group ensures that spills are minimized.

We emphasize that during this code generation process, all abstract registers and target registers are treated symmetrically. We do not distinguish between

registers used to pass parameters, those used to carry visible results, and those used to hold temporary values. In this way, we can code generate a region containing multiple procedure call and returns as efficiently as one containing just a portion of a large procedure.

The final stage of code generation is to generate stubs for the entry and exit blocks, which need to load abstract register values into target registers and compute and store exit values from the group block.

This group-block creation phase can be invoked and re-invoked any number of times during program execution, creating larger and smaller groups of basic blocks, always independent of method boundaries, as the subject program proceeds through its execution.

3. Implementing Java

The critical design decisions when implementing Java using Dynamite are the mapping of JVM registers, local variables, and the stack to the relevant Dynamite objects, namely abstract registers .

To allow Dynamite to optimize across different Java methods, we need to map multiple stack frames simultaneously to different abstract registers. Two schemes were considered for doing this.

3.1 Sliding frame

On entering a method the front-end would create a frame within the abstract registers. The arguments to the method (stored at JVM local variable 0 upwards) become the base for the frame. After the local variables the return address is held in the next available abstract register. The JVM stack is held at the end of the frame. However, studies [4] show that the stack is usually empty on basic block boundaries. The purpose of the stack in the frame is therefore to hold onto stack values that occasionally span basic blocks and to pass arguments to called methods. The arguments could become part of the next frame by overlapping the stack part of the caller's frame and the local variable part of the callee frame.

Unfortunately, the problem with this scheme is that the IR for an individual method needs to refer to specific abstract registers. This fixes its translation to a particular stack depth. If the same method is called at a different stack depth, we need to re-translate it for this new depth. This is particularly expensive for recursive methods. We could possibly generate special case translations for recursive methods and fall back on a scheme that saves the frame to memory on a method call. Otherwise, for methods that are called from multiple stack locations we could avoid re-translation if the method's frame is at a greater abstract register location than the current frame. We would, however, still have to copy the arguments to the method from the caller's frame to that of the callee.

Our research has shown that around 90% of execution time is spent in methods called from more than 16 call sites. These methods are typically utility functions which are prime candidates for optimization. Expensive optimizations would be prohibitive for these methods as the optimization would need repeating many times.

We conclude that using a sliding frame is therefore undesirable.

3.2 Fixed frame

The drawback with the "sliding-frame" scheme is that it is necessary to recompile methods called with different frame base pointer values. If we fix the address where a method's frame lives in abstract registers we remove this problem. To do this, we allocate a new, unique frame from a large pool of abstract registers the first time a particular method is invoked.

We do, however, still need to pass arguments to the called method from the stack of the calling method. On encountering a method call, the arguments to the method are held in intermediate representation ready to be written to registers. At this point, they can be written directly to the called method's local variables avoiding any copy operation.

The first penalty for this scheme is that we need to retranslate these abstract register assignments for different methods called from the same call site. A study using Harissa [5] shows that at least 40% of method calls can be accurately statically predicted for 100% of the time, and dynamic statistics are even better than this.

As in the "sliding-frame" design, recursion needs to be handled differently, as two invocations of a method cannot share a single frame. A simple scheme to handle recursion is to save a frame to memory before using it, if it is active, and to restore it on exit. Alternatively, we may find some circumstances in which it is advantageous to generate special-cased versions of recursive methods, each of which uses a different frame of abstract registers. This special-casing will be triggered by a heuristic monitored by code planted in the initial translation of a (potentially recursive) method.

Finally, assigning unique abstract registers to every method presents a problem when the static pool is exhausted. For programs studied to date, fewer than 8000 abstract registers would be sufficient. If greater numbers were required, the front-end could start re-using frames. For example, all leaf methods can share the same frame, and more generally, methods that occur only on disjoint subtrees of the call graph can share frames. In the pathological case, frames of abstract registers can be reused by planting code that spills a number of frames to memory and refills them when necessary.

4. Discussion

To evaluate this scheme before its implementation, we carried out a number of experiments by instrumenting

Kaffe [6] to log information about the dynamic behaviour of Java applications. We keep sufficient information to create the dynamic method call tree of the application. We create a call tree, in which methods appear once for each call site, to assess our implementation alternatives. For each method occurrence, we keep the number of byte codes executed in this invocation, and its local variable requirement, including parameters.

To estimate the number of target registers needed in optimization regions of different sizes, we identify hot spots on this call tree (methods with high contributions to overall instruction counts), and successively add them to optimization regions, counting the total number of local variables required at each step. This approximates to code generating by hottest method first, then by successively cooler region. As each successive method is added to the optimization region, we require more local variables. This gives us the characteristics we show below. For these experiments, we monitored "javac", the Java compiler in Java, since it is the largest Java application we can find.

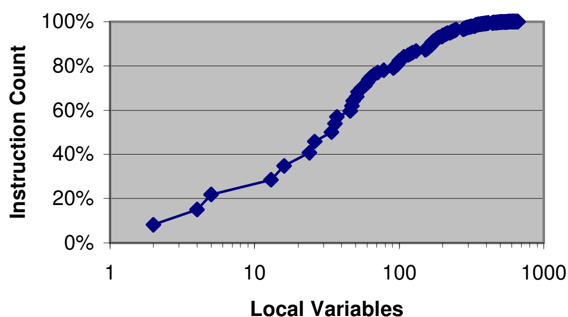

Figure 2. Instruction Count against Local Variables

In figure 2, we show how a straightforward "hottest first" selection algorithm can use varying numbers of target registers to code generate "spill-free" regions of different sizes. As we intuitively expect, as we allocate more and more local variables into target registers, we can encompass larger and larger regions, contributing to ever increasing fractions of total instruction count. For example, with 5 target registers, we can code generate a region that contributes 22% to total execution count, and with 26 registers, 46% of execution count.

In Figure 3, we use a slightly different heuristic to select methods to include within our selection region. Here, we select methods based on their run-time contribution per local variable. That is, comparing methods with similar run-time contributions, we

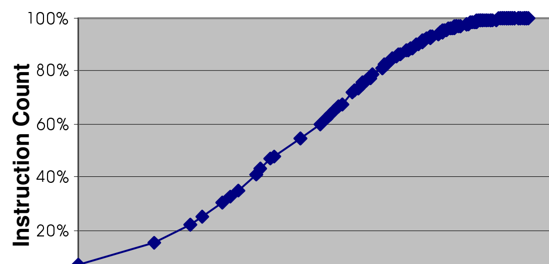

Figure 3. Instruction Count against Local Variables

Local Variables

preferentially select the method with the smaller requirement for local variables. This gives better results when there are fewer target registers available: for example with 8 registers, we can cover 30% of total instruction count, and with 25 registers, we can cover 54% of the instructions.

5. Previous Work

The success of Java has resulted in many JVM implementations. Some implementations such as Harissa [5], and J2C translate Java to C code. They then rely on a C compiler to perform register allocation within and over method call boundaries. Register mappings within C programs are beyond the scope of this paper.

In this section we examine how other JVM implementations perform register mapping and allocation and compare these to Dynamite.

5.1 Register allocation

Cacao [4] initially maps the JVM stack and local variables to pseudo-registers, which are then allocated to CPU registers. Each mapping and allocation begins at the start of a basic block and builds on the mappings and allocations of previous basic blocks. When CPU registers are exhausted a register is spilled to memory and filled by a pseudo register.

On method call boundaries Cacao pre-allocates registers. It uses CPU registers to pass arguments and to receive return values. On machines without register windows pre-allocation of arguments is only possible for leaf methods.

Pre-allocation can tie in with existing compiler method call conventions: for example, in the DAISY JVM [7] arguments and return values are passed and received using the Power PC's C compiler calling conventions, which uses standard registers for passing arguments.

5.2 Comparison with Dynamite

Method invocation creates a new frame on a call stack. Cacao avoids unnecessary accesses to this frame by pre-allocation, utilizing register windows or potentially by using the machine's standard calling convention. However, these static mappings take no account of run-time information on register usage. Therefore registers could be allocated and then subsequently unused. Cacao would also calculate any parameters even if they were unused. Cacao would also have to copy from one register to another if it repackaged arguments to another method. Dynamite on the other hand can avoid this by value forwarding and dead code elimination within a group block.

Also, when registers are spilled only the surrounding and previous basic blocks are considered. This means that a pseudo register could be spilled in one basic block and then filled back again in the next, and Cacao wouldn't know it could spill different registers which are unused in subsequent basic blocks. Dynamite's runtime information about register usage can provide a better register allocation in this case.

6. Conclusion

In this paper, we have introduced Dynamite, an environment for creating dynamic binary translators. We have shown how the run-time concepts of the Java Virtual Machine are mapped onto the Dynamite front end interface and its internal register allocation algorithms. This mapping necessarily discards the concepts of methods and their local variables.

Preliminary investigations show that "method-free" register allocation shows promise for efficient code generation across architectures providing wide ranges of hardware register resources. We look forward to presenting more definitive numerical results at the workshop in October.

7. References

[1] Intel Corporation, "Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture," Order Number 243190, 1999.

[2] Intel Corporation, "IA-64 Application Developer's Architecture Guide," Order Number 245188, 1999.

[3] Trimaran consortium, "Trimaran Project Homepage," http://www.trimaran.org

[4] Andreas Krall, "Efficient JavaVM Just-in-Time Compilation," *International Conference on Parallel Architecture and Compilation Techniques (PACT98),* Paris, France, October 13-17, 1998.

[5] G. Muller, B. Moura, F. Bellard, C. Consel, "Harissa: A Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code," *Third USENIX Conference on Object-Oriented Technologies (COOTS-97),* Portland, Oregon, June 16-20 1997.

[6] Transvirtual Technologies Inc., "Kaffe Product Architecture," http://www.transvirtual.com/products/architecture.html

[7] K. Ebcioglu, E.R. Altman, E. Hokenek, "A JAVA ILP Machine Based on Fast Dynamic Compilation", *IEEE MASCOTS International Workshop on Security and Efficiency Aspects of Java,* Eilat, Israel, January 9-10, 1997.