Targeting Dynamic Compilation for Embedded Environments

Michael Chen and Kunle Olukotun
Computer Systems Lab
Stanford University
mikey@hydra.stanford.edu, kunle@ogun.stanford.edu

Abstract

A generally held notion is that high quality code comes with high compilation cost. As a result, previous efforts at minimizing dynamic compilation costs have focused on designing fast, lightweight compilers that sacrifice code quality for compilation speed, and resource intensive approaches that combine multiple engines to limit expensive optimizations to critical sections. In this paper, we show one possible way fast compilers can be constructed to generate high quality code. We have implemented microJIT, a small and portable just-in-time (JIT) compiler for Java that can produce high quality code 2.5x faster than a comparable dataflow-based compiler and 30% faster than a compiler that performs only limited optimizations. We use dataflow techniques, but speed up compilation by minimizing the number of major compiler passes given the number of optimizations performed. Architectural features of our compiler also allow it to perform instruction set dependent optimizations efficiently. microJIT achieves these high compilation rates while still maintaining small static and dynamic memory requirements. This compiler can be highly effective in an embedded system where computing and memory resources are highly constrained and where multiple target platforms must be supported.

1. Introduction

Dynamic code generators differ from traditional compilers because they must carefully consider their impact on runtime performance. It is widely believed that generating optimized code comes with high compilation times. Two approaches to reducing dynamic compilation overheads have been popularized.

The first approach uses techniques that focus solely on generating code quickly [1][9]. This usually involves focusing only on simple optimizations, or relying on imprecise analysis or heuristics to speed up compilation. This helps alleviate compilation times, but penalizes long-run performance through poor code quality.

The second approach is through lazy compilation. A lazy compilation system usually combines a fast, non-optimal compiler or interpreter with an expensive compiler [10]. The key goal of this system is to minimize expensive optimizations. Runtime profiling selects which portions of code to interpret and which portions to recompile and optimize aggressively.

In this paper, we show the gap between high quality code and fast compilation may be bridged a third way: by improving the compilation performance of the dynamic compiler. We have implemented microJIT, a small and portable dataflow compiler for Java that produces optimized code 2.5x-10x faster than comparable dataflow-based compilers and 30% faster than a compiler that performs limited optimizations. These compilation rates are achieved while maintaining small static and dynamic memory requirements. The major contributions of this paper are:

- Architecture and implementation of a fast, portable, and small optimizing compiler. Compilation speedup was achieved by minimizing major compiler passes for the number of optimizations performed: global and local code optimizations are applied as intermediate expressions are generated, registers are allocated concurrent with code generation, and dataflow information is efficiently communicated between compiler passes. Our compiler also implements an extension infrastructure to help support fast machine dependent optimizations.

- Experimental results that show using this fast compiler alone can compete against systems using interpreters, lazy compilation, and fast compilers on long and short run applications, with less total system cost.

Our compiler can be highly effective in embedded systems like PDAs, tablet PCs, and thin clients targeted by Sun's J2ME (Java 2 Micro Edition) platform, using the CDC and CVM configuration (http://java.sun.com/j2me/). These environments have highly constrained computing (30Mhz – 200MHz) and

memory resources (2MB – 32MB RAM and ROM) relative to modern desktop machines, and must support many target platforms. The current standard for these devices is to interpret bytecodes, which results in at least 10x slowdown relative to native code. Our fast compiler may alleviate tradeoffs that would otherwise need to be made in this environment. Overheads for expensive code optimization can be significant for slower devices, and a fast, poorly optimizing compiler ultimately sacrifices long run performance for code generation time. Alternatively, using multiple compilation and/or execution engines required for lazy compilation add to code ROM size and require more development effort to target different machine platforms.

The rest of our paper is organized as follows. We describe related dynamic compilers and fast compilation techniques in Section 2. We outline the major optimizations performed by our compiler in Section 3 and discuss how they relate to traditional dataflow implementations in Section 4. In Section 5, we compare compilation times and code performance with other dynamic compilers. Finally, we present our conclusions in Section 0.

2. Related Work

The Sun Hotspot Java virtual machine (JVM) is the most widely known system that implements lazy compilation [10], although similar research and commercial systems by Intel, IBM and SNU exist [5][19][16]. We compare the performance of these compilers to microJIT in Sections 4 and 5. An interesting extension of lazy compilation is adaptive optimization, implemented in the Jalapeño VM [2]. Written mostly in Java, this VM moves the Java / non-Java boundary below the VM, providing more opportunities for optimization.

A critical component of fast dynamic compilation is fast register allocation. The Intel JIT compiler uses lazy code selection in the context of on-the-fly register allocation to speed up compilation [1]. Previously encountered bytecode sequences that generated the current value in a register are cached, as well as possible bytecode aliases, so that future equivalent sequences can simply be replaced by an instantiation of the register. In linear scan register allocation [17][4], a prepass computes lifetimes and lifetime holes and then directs global allocation of registers with a simple linear sweep over the program being compiled. The LaTTe JIT compiler uses another potentially effective fast register allocation scheme that requires a prepass and two sweeps [19]. In Section 4.2, we discuss how these allocators relate to the one implemented in microJIT.

There are several well-known research projects that use dynamic compilation. The SimOS project includes the Embra execution engine that uses dynamic translation to speed up architecture simulations [18]. `C (“tick-c”) implements language extensions to C to support dynamic compilation of critical sections of code [14][7][13]. The Digital FX!32 project uses dynamic translation to run x86 binaries on the Alpha architecture [15]. The Dynamo project optimizes and recompiles binaries according to statistics collected from runtime profiling [3].

3. Compiler Architecture

3.1 Overview

Our compiler only makes three major passes over

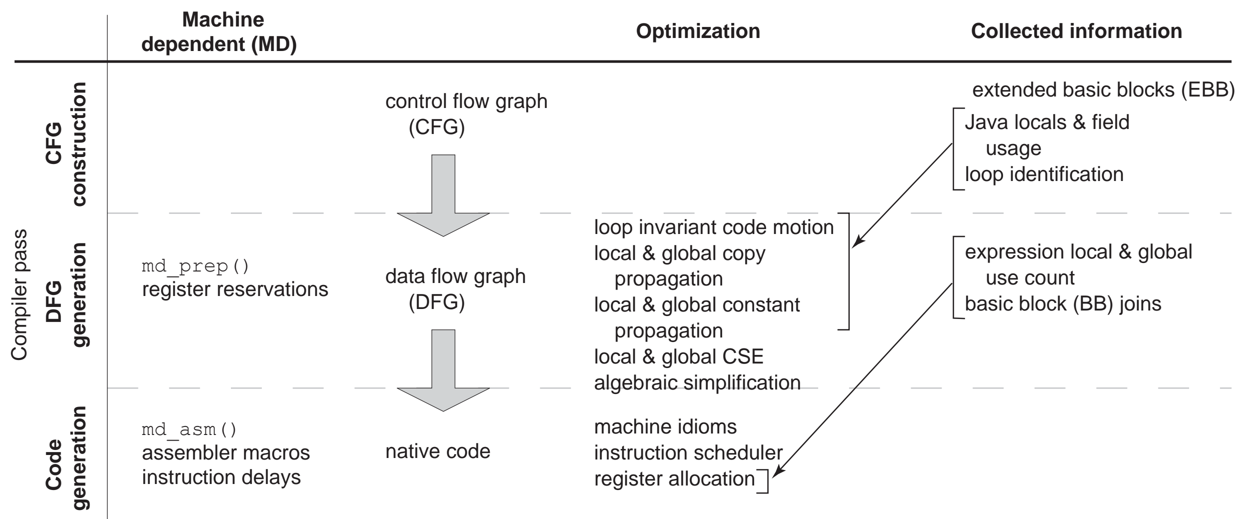

Figure 1. Architectural overview of microJIT detailing flow of information through major components and passes.

the code: a fast scanning pass to build the control flow graph (CFG), a major pass to generate the data flow graph (DFG), and then the final pass to generate the code. We found that organization represents a minimal pass configuration that gathers enough good information for the following pass to compile effectively. Figure 1 graphically illustrates the high-level architecture of our dynamic compiler, showing specific work done by each pass, ISA specific components, and flow of information between the major blocks. The following sections (3.2 – 3.5) detail the major compiler passes, and steps we take to minimize compilation costs without sacrificing generated code quality.

3.2 CFG Construction

We generate a CFG of basic blocks (BB) from a single pass of the bytecode. In our compiler, a method starts as one block representing all the bytecodes. Blocks are split as branches and associated targets are found. During a block split, appropriate control flow arcs between blocks are added and adjusted. In this fashion, we can generate the CFG without touching a bytecode more than once.

This scan executes very quickly. Most bytecodes, except for control flow instructions, are not decoded during this scan, and most next pc and stack pointer offsets can be found simply by indexing the bytecode into a static lookup table. At the end of CFG construction, arcs are also added to any defined exception handlers for the BB.

From the CFG, we compute extended basic blocks (EBB) and dominator blocks using standard algorithms [12]. An EBB is a maximal sequence of BBs with one entry and possible multiple exits. Subsequent passes operate on EBBs in order to allow local optimizations to be applied to the largest possible region.

Table 1. Example IR expressions.

Class	Example expression	bytecode equivalent(s)	Arguments
Load/ store	load	`getfield` `getstatic`	1 “
Unary op	`not`	`not`	1
Binary op	`add`	`add`	2

Branches	`cbr_eq`	`ifeq`
		`if_icmpeq`
		`if_acmpeq`
			2 + target
			“
			“

Auxiliary	`call`
	
	`prologue`
		`invokevirtual`
		`invokestatic`
		`invokespecial`
		`none`
			variable
			“
			“
			0

Dominator block information is immediately used to detect loops. Since `goto` statements with arbitrary target labels are not allowed in the Java programming language [8] (although it is not formally excluded from the VM specification [11]), we can expect all loops to have one entry point and be found with a natural loop detector [12].

Basic load/store statistics on local (e.g. `iload`, `astore`) and field (e.g. `getfield`, `putstatic`, `iaload`, `aastore`) accesses are recorded for each BB during CFG construction. The local and field accesses for each BB in a loop are then merged to compute their definitions and uses within the loop. This is used in the next pass for global optimizations.

3.3 DFG Generation

3.3.1 Intermediate Representation

Within BBs, we use triples to represent IR expressions. Triples are similar to quadruples used by most compilers, except that results are not named explicitly [12]. In our implementation of triples, pointers are used to refer to source argument

Java bytecode

```
bpc
 0  aload_0
 1  getfield count      (3)
 4  newarray char
 6  astore_1
 7  getfield count      (3)
10  iconst_1
11  iadd
12  putfield count
15  aload 1
16  areturn
```

Equivalent C

```
L1 = new char[count];
count += 1;
return L1;
```

Intermediate representation

```
bpc eid
    [1]  prologue                    (1)
 0  [2]  load @( [L0]+16 ) r:%o1
 4  [3]  const 5 r:%o0 (1)
 |  [4]  const 0x48a9c
 |  [5]  call newarray [4]
 v         ( [3] [2] )
 6       (2) -> [L1] r:%o0 (1)
10  [6]  const 1
11  [7]  add [2] [6] (3)
12  [8]  store [7] @( [L0]+16 )
16  [9]  returnarg [5] r:%i0
                    (4)
```

Pointer assignments (@ bpc=16)

```
           locals.in   locals.out
local0_p:  [L0]        [L0]
local1_p:  null        [5] (2)

stack1_p:  null
stack0_p:  [5] (4) <---- stack_p
           stack
```

Key points:

1. argument and return registers are reserved by `md_prep` for call expression (`r:%x` refers to SPARC ISA registers)
2. local1_p is assigned pointer to call expression
3. CSE opt matches getfield count @ pc=1 (safe to match through newarray() call since it is an internal VM function)
4. load local 1 on stack to be returned by this method; return argument reg also reserved

Figure 2. Example showing how bytecodes are translated to our intermediate representation

expressions. Basic expression classes of our IR are listed in Table 1. Our IR expressions more closely resemble basic machine instructions than Java bytecodes to make mapping to machine code more straightforward. Constants are represented as individual expressions so they can be properly manipulated on the Java stack as bytecodes are processed. Because we implement triples, expressions have no explicit destination.

3.3.2 Local Optimization

Our compiler is designed to perform local optimizations quickly. This is important because bytecodes sequences often reoccur, a side effect of using a stack to hold temporaries, and of complex bytecodes such as array accesses that hide repeated computations.

```
EXPR_add( expr * e, basic_block * bb )
{
    // CONSTANT OPTIMIZATION
    if( constant_arguments( e ) ){
        e->const_opt();
    }
    // ALGEBRAIC SIMPLICIATION
    else{
        e->algebraic_opt();
    }
    // LOOP OPTIMIZATION
    if( in_loop ){
        (success, match_e) = bb->loop_opt( e );
        if( success ){
            return match_e;
        }
    }
    // CONSTANT SUBEXPRESSION ELIMINATION
    (success, match_e) = bb->cse_opt( e );
    if( success ){
        return match_e;
    }
    else{
        (success, match_e)
            = bb->global_cse_opt( e );
        if( success ){
            return match_e;
        }
    }
    // SETUP MACHINE DEPENDENT INFO
    e->md_prep();
    // ADD TO BASIC BLOCK
    bb->add( e );
}
```

Figure 3. `EXPR_add()` pseudo code (simplified code).

Figure 2 illustrates an example of how bytecodes are translated into expressions in our IR. A local expression pointer array and a stack pointer array are maintained as we interpret bytecodes in a BB. Java locals and stack assignments simply move expressions between these two arrays and update the stack pointer. Using pointer assignments to mimic the VM stack and locals effective performs copy propagation as no intermediate copy expressions are generated for these operations. Expressions that are assigned to Java locals on BB entry and exit are listed in expressions pointer arrays for each BB (`locals.in[]` and `locals.out[]`).

Only when we actually encounter a true operation is an expression generated for it. When an expression is created, it is submitted to the function `EXPR_add()` (see Figure 3), which checks the expression for possible optimizations before adding it to the expression list for the BB. Basic optimizations performed include constant propagation, and algebraic simplifications and reductions. Assuming a non-constant expression, local common-subexpression elimination (CSE) is applied to the new expression. Our local CSE is implemented as a backward search within the EBB for an available matching expression, with expressions hashed by operation to eliminate searching through expressions that cannot possibly match the new expression.

Most bytecodes that perform simple operations simply map to a single corresponding IR expression. Complex bytecodes, like array accesses, branches and method calls, are decomposed into IR expressions that are more representative of the instructions that must be executed on the underlying hardware. Consider the array access sequence shown in Figure 4. Decomposing allows us to optimize an array bounds check with another access to the same array, or allows us to use the same index computation for access to a different array.

3.3.3 Inlining and Specialization

Our inliner supports fast inlining of small methods (< 20 bytes) rather than full, and potentially more expensive, integration of a callee method into a caller method. Expressions in the inlined method are added to the caller context, but maintain a separate environment.

Small methods represent the most common opportunities to inline, and result in big performance gains by eliminating large calling overheads relative to work performed within the inlined call. Our inliner handles nested inlining (e.g. for optimizing subclassed object constructors like `class.<init>`), and specialization of virtual and interface methods (e.g. for optimizing object accessor methods). The inliner is also responsible for inlining fast, common case handlers for the `checkcast` and `instanceof` bytecodes, which must execute a costly class hierarchy search if the object class is not equivalent to the requested class.

Java bytecode

```
bpc
0  aload_0
1  iload_1
2  iaload
3  iconst_1
4  iadd
5  aload_0
6  iload_2
7  iastore
```

Equivalent C

```
L0[L2] =
   L0[L1]+1;
```

array object layout: 0 header, 8 length, 12 a[0], 16 a[1], 20 a[2]

Unoptimized intermediate representation

```
bpc eid
 2 [1]   load @([L0]+8)
   [2]   cmp [L1] [1]
   [3]   cbranch_ult [2]
           --> [5]
   [4]   call bad_array_idx
   [5]   target
   [6]   const 2
   [7]   sll [L1] [6]
   [8]   const 12
   [9]   add [L0] [8]
   [10]  load @([7]+[9])
 3 [11]  const 1
 4 [12]  add [10] [11]
 7 [13]  load @([L0]+8)        (1)
   [14]  cmp [L2] [11]
   [15]  cbranch_ult [14]
           --> [17]
   [16]  call bad_array_idx
   [17]  target
   [18]  const 2               (1)
   [19]  sll [L2] [18]
   [20]  const 12
   [21]  add [L0] [20]         (1)
   [22]  store @([19]+[21])
```

Optimized intermediate representation

```
bpc eid
 2 [1]   load @([L0]+8)
   [2]   cmp [L1] [1]
   [3]   cbranch_ult [2]
           --> [5]
   [4]   call bad_array_idx
   [5]   target
   [6]   const 2
   [7]   sll [L1] [6]
   [8]   const 12
   [9]   add [L0] [8]
   [10]  load @([7]+[9])
 3 [11]  const 1
 4 [12]  add [10] [11]
 7 [14]  cmp [L2] [11]
   [15]  cbranch_ult [14]
           --> [17]
   [16]  call bad_array_idx
   [17]  target
   [19]  sll [L2] [18]
   [22]  store @([19]+[9])
```

(1) instructions removed in second array access

Figure 4. Array accesses bytecodes are decomposed so that optimizations can be performed on their components.

3.3.4 Global Optimization

We perform global optimizations non-iteratively, but produce results that are equivalent to a traditional iterative dataflow analysis. This is possible because IR expression generation for EBBs is processed in reverse post-order traversal (an EBB must be processed before any of it sucessors have been). In this fashion, we can propagate forward flow information to successor EBBs before IR expressions are generated for them.

At EBB headers, we merge flow information for global copy and constant propagation before generating IR expressions in the EBB. Loop invariant code motion and global CSE are handled within the `EXPR_add()` function introduced in the previous section.

If the current BB exists within a loop, a check is made to see if the new expression can be hoisted to the loop preheader. For most expressions, this involves determining if their arguments are constants or locals that are not redefined in the loop. This later property is queried from the loop locals and field access statistics computed during the CFG generation pass (Section 3.2).

Global CSE is performed on a new expression only if local CSE fails and its arguments are locals or globals that we have created. The global CSE optimizer searches BBs backwards toward the method entry for matching available expressions using the same routines used for CSE within an EBB. We terminate the search if one of the arguments is redefined along any of the backward paths toward method entry.

We compare our local and global optimizer to other implementations in Section 4.1.

3.3.5 Data Flow Statistics

During DFG generation, we collect additional information that will be utilized by the register allocator. Each IR expression includes a local and global use counter. The local use counter is incremented whenever a given expression is used as a source in another expression. We also compute a flag called `expr_spans_call` which is set if a call occurs between an expression definition and use.

The global use counter is accumulated towards the expression definition after all expressions have been generated using a post-order traversal from method exit BBs to the entry BB. The global use count for a given local expression at the exit of a BB is equal to the sum of the local and global uses for the local expression at the entry to immediate successor BBs. We do not consider this scan a major pass as only expressions in Java locals at BB exits and entries are considered. This computation is equivalent to a live variable dataflow analysis but also includes relative weighting of uses. This computation must be iterated when loops are present to compute correct liveness values within the loop.

Also computed concurrent with DFG generation is a structure we call a BB join. A BB join is the union of all adjoining BBs entries or exits, with each BB identified as whether its entry or exit is part of the join. Example BB joins are shown in Figure 6. A BB may be in at most two BB joins, or it may be in only one BB join (its entry and exit share common successors and predecessors). A BB join is used to link register assignments between dependent BB entry and exit points, described in more detail in Section 3.4.2.

3.4 Code Generation Pass

3.4.1 Code Generation

For the most part, code is generated in place using single pass of the expressions in a BB. Like expression generation, EBBs are processed in reverse post-order when generating code. A patching system is used to fix unknown values likes branch targets and variable sized method prologues and epilogues (for certain ISAs) after the primary code generation pass. Selective code buffering and movement is supported for block-level code scheduling. This facility is currently used to move array out-of-bounds throw clauses out of the critical code path and to implement loop inversion.

3.4.2 Register Allocation

We do not allocate registers in a separate pass, but assign registers as code is generated. Use counters, the `expr_spans_call` flag, and register pre-assignments are all critical to achieving good register allocation. Pseudo-code for the register allocator is shown in Figure 5 and an example allocation pass is shown in Figure 6.

```
REG_alloc( expr * e )
{
    reg * r; int r_type;

    // HANDLE REGISTER RESERVATIONS
    if( e->reserved_reg ){
        r = e->reg;
        if( r->is_assigned ){
            r->spill_expr();
            r->free();
        }
        r->assign( e );
    }
    // NORMAL ALLOCATION
    else{
        // CLASSIFY REG ASSIGNMENT
        if( e->uses.other > 0
            || e->spans_next_call ){
            r_type = R_caller_saved;
        }
        else{
            r_type = R_temp;
        }
        // ASSIGN REGISTER
        if( r = get_free_reg( r_type ) ){
            r->assign( e );
        }
        else if( r = any_free_reg() ){
            r->assign( e );
        }
        else if( r = min_cost_live_reg() ){
            r->spill_expr();
            r->free();
            r->assign( e );
        }
    }
}
```

Figure 5. `REG_alloc()` pseudo code (simplified code).

Register allocation starts with any register allocated arguments at a method entry. When a register is needed for an expression being generated, allocation occurs as follows. If a register has not been pre-assigned, we must choose an appropriate register class assignment (e.g. temporary or call-preserved register). Accounting for whether an expression must survive a future call (`expr_spans_call` flag), whether it will survive past the BB it is defined in (global use counter), and potential conflicts with registers allocated at the BB exit, we can select an appropriate register to minimize future moves or spills.

As each expression is processed and code is generated for it, we decrement the local use counter of the source arguments to reflect that the argument has been "used." When the local use counter and the global user counter both are zero for a register allocated expression, the register can be freed as it can be guaranteed that this expression will never be used again.

Once code has been generated for a BB, the BB linker is responsible for properly linking register assignments between BBs. The register assignments at a BB exit are compared to the assignments in the BB join. Register assignments are added to the BB join or a register move or spill is generated match a previous BB join register assignment.

We will compare our register allocator to other schemes in Section 4.2

3.4.3 Instruction Scheduler

We use a standard list scheduler for low-level instruction scheduling. To simplify the implementation, scheduling is currently limited to a given BB. Despite this limitation, we believe BB regions have enough instructions to sufficiently target the biggest benefactors of scheduling, filling load and branch delay slots. We schedule instructions only after they have been generated because some IR expressions may expand into more than one instruction while others may not even generate one. Additionally, the loads and spills to the runtime frame of Java locals, which are only generated as needed, are not represented as explicit expressions in our IR.

3.5 Fast Optimization of Machine Idioms

Machine idioms are instructions or instruction sequences for a specific ISA that execute more efficiently than a similar sequence of instructions targeted for a more general architecture. Common machine idioms include immediate arguments, auto-increment arguments, call argument specifics, leaf

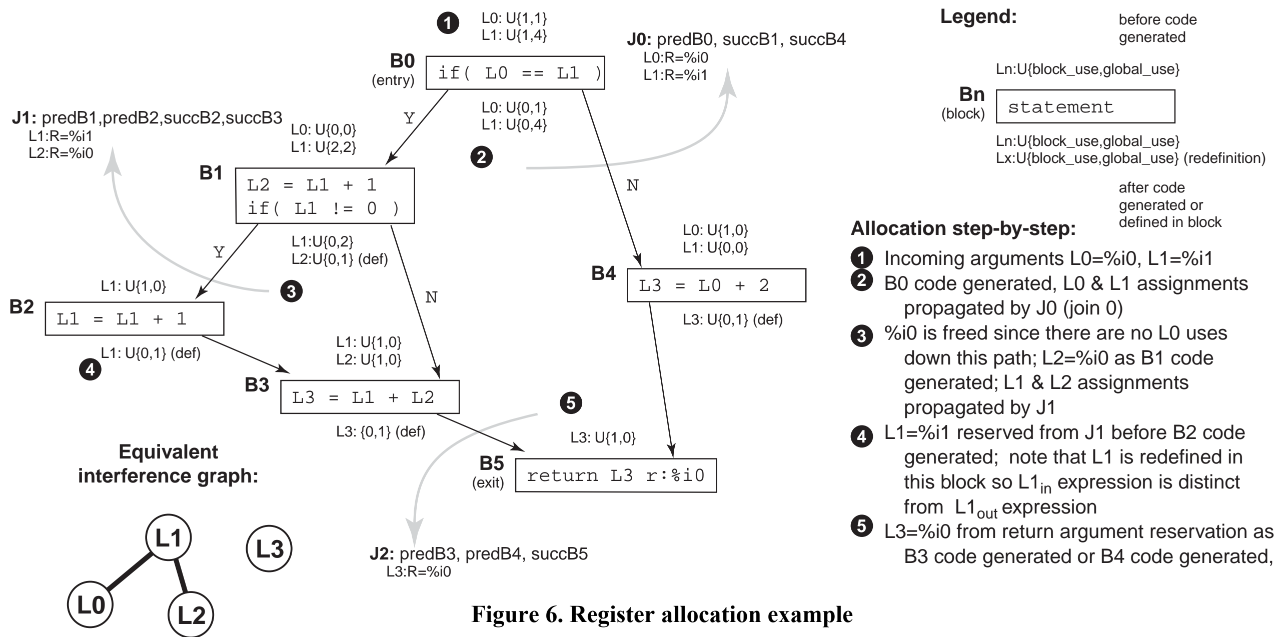

Figure 6. Register allocation example

procedure optimization, and condition code (CC) usage.

Optimizing for machine idioms is often left to the end of the compilation process, after one has generated machine specific code, using a peephole optimizer [12]. A peephole optimizer searches code for instruction patterns it knows how to replace with an equivalent sequence that requires less instructions or cycles. Using peephole optimization is expensive because it usually requires at least one additional pass across all instructions generated with a pattern matcher.

Java bytecode

```
bpc
0  aload 0
1  iload 1
2  iaload
3  istore 3
4  aload 0
5  iload 2
6  iaload
7  istore 4
```

Equivalent C

```
L3 = L0[L1];
L4 = L0[L2];
```

array object layout: 0 header, 8 length, 12 a[0], 16 a[1], 20 a[2]

Intermediate representation 1

```
bpc eid
(bounds check omitted
   for clarity...)
 2 [1]  const 12
   [2]  add [L0] [1]
   [3]  const 2
   [4]  lsl [L1] [3]
   [5]  load @( [2]+[4] )
 6 [6]  lsl [L2] [3]
   [7]  load @( [2]+[6] )
```

SPARC code

```
add  %i0, 12, %g1
sll  %i1, 2, %g2
ld   [%g1+%g2], %g3
sll  %i2, 2, %g2
ld   [%g1+%g2], %g4
```

ARM code

```
add  r3, r0, 12
ldr  r4, [r3, r1, LSL#2]
ldr  r5, [r3, r2, LSL#2]
```

1 ARM and SPARC support [r1 + r2] addressing

Intermediate representation 2

```
bpc eid
(bounds check omitted
   for clarity...)
 2 [1]  const 2
   [2]  lsl [L1] [1]
   [3]  add [L0] [2]
   [4]  load @( [3]+12 )
 6 [5]  lsl [L1] [1]
   [6]  add [L0] [2]
   [7]  load @( [3]+12 )
```

MIPS code

```
lsl  $t1, $a1, 2
add  $t0, $t1, $a0
lw   $s0, 12($t0)
lsl  $t1, $a2, 2
add  $t0, $t1, $a0
lw   $s1, 12($t0)
```

2 MIPS only supports [r + offset] addressing; note that offset can be merged directly into the expression since these cannot change during compilation

Figure 7. Idiomatic optimizations can be performed more quickly by having different IRs for different ISAs.

In our dynamic compiler, we handle machine idioms as well as other miscellaneous opportunities to reduce instruction sequences by providing machine dependent (MD) code an opportunity to access an expression as soon as it is created. This allows us to perform preliminary analysis and set flags which can be accessed at code generation time to help generate machine idioms. A common use of this facility is for generating immediate arguments. In the `md_prep` access, the MD code can flag constants that can fit into immediate fields for that ISA. When the actual pass to generate code occurs, only constants that cannot fit into immediate fields will be register allocated.

Another important use of this facility is for satisfying calling argument conventions. We implement a register reservation system where register assignments can be made before the code generation pass. Using this system, we can pre-assign registers to expressions that will be used as a call argument, which prevents a possible expression assignment to an unknown register and then an extra move to the correct argument register during code generation.

For more complex idioms, we can make minor adjustments to the IR to simplify certain optimizations. For example, we provide a special branch IR representation to accommodate

architectures that set CCs. Also useful is altering array access decompositions to target addressing modes available for the ISA, as illustrated in Figure 7.

Using this approach, we do not need to make an additional pass to optimize for machine idioms. This system appears sufficiently robust, as we have been able to accommodate all the low-level optimizations that we have wanted to perform on the ISAs we have targeted so far.

Initial design of the compiler was done on a MIPS IV ISA. So far, we have ported the compiler to the SPARC v9 and StrongARM ISAs. An ISA port requires defining a register file, assigning register classes and coding `md_prep` (when required) and `md_asm` functions for each IR expression type (see Figure 1). MD code represents about 1/4 – 1/3 of the total binary size of our dynamic compiler.

As a demonstration of the portability of our design, each port, with some MD optimizations, only took about 2 man-weeks to complete. As RISC style machines, the SPARC and MIPS ports are similar, but there are still significant differences in the register file and calling convention models. The ARM architecture provides for unusual source argument arrangements, load/store addressing modes, and full instruction predication. The current port does not use the more exotic aspects of the ARM architecture, but support could be added in future revisions. A x86 port is planned, though not currently implemented.

4. Comparisons To Other Compilers

4.1 Dataflow Analysis

Our compiler implementation of dataflow algorithms differs significantly from most modern optimizing compilers. In this section, we discuss how our approach compares against traditional implementations.

Optimizing compilers like the Sun-server compiler [6] and LaTTe JIT compiler [19] implement traditional dataflow analysis using lattices and flow functions [12]. Traditional implementations can be computationally expensive for several reasons. Setting up a problem for dataflow analysis often requires scanning the entire method to set up initial conditions. Additionally, some optimizations require more involved auxiliary data structures than bit vectors alone, may be iterative, or may require solving more than one dataflow problem.

Rather than setting up separate dataflow problems, we apply several optimizations concurrently as IR expressions are generated. Forward flow information required by these optimizations is communicated by processing EBB in reverse post-order when generating IR expressions

When performing traditional iterative dataflow analysis, blocks are also processed in reverse post-order to minimize required iterations to reach a fixed point [12]. If A is the maximal number of loop back edges in the CFG, the bound on the maximum number of times a block may be visited before reaching a fixed point is $A+1$. Logically, this is required to propagate forward flow information through loops so a fixed point can be found at the loop header. In our implementation, we can use the per loop Java local load and store usage statistics collected in the CFG generation pass (Section 3.2) to compute forward flow information through loop back edges without iterating. For example, a local V can be copy or constant propagated to successor blocks outside the loop if V is not redefined within the loop (no stores to V within the loop).

Our loop invariant code motion optimizer is also non-iterative. Normally, a loop has to be scanned multiple times as loop invariant code motion of one expression inevitably can cause other expressions dependent on it to become loop invariant [12]. We generate IR expressions in a BB in order, and predecessor loop BBs are always processed first. As a result, when loop invariant code motion is applied to an expression, this change is immediately communicated to successive, dependent instructions of the loop.

What might be considered a major limitation to our approach is that it is largely restricted to optimizations that rely primarily on forward flow information. Since most basic optimizations are based on forward flow information (e.g. reaching definitions, available expressions, copy propagation, constant propagation) [12], we do not consider this a serious restriction.

4.2 Register Allocation

In this section, we compare our on-the-fly register allocation scheme with graph coloring and other proposed fast allocation schemes. Most register allocation algorithms work with liveness information, the span between a variable's definitions to all its uses.

While graph coloring usually generates the best results, it can be expensive. Register coalescing and spill points cause the algorithm to iterate, which can result in high register allocation times, particularly for methods that are large or are not initially colorable [12].

Table 2. Features and characteristics of compilers evaluated.

JIT	Sun - Client	Sun - Server	SNU LaTTe	MicroJIT
Source	C++	C++	C	C
64b ops	Full	Full	Some	Some
Intermediate representation	Simple	SSA dataflow	Dataflow	Dataflow
Major compiler passes	4	Iterative	7	4
Optimizations	Block merging/elimination Simple constant propagation Inlining & specialization	Loop invariant code motion Global value numbering Conditional constant propagation Inlining & specialization Instruction scheduling	EBB value numbering EBB constant propagation Loop invariant code motion Dead code elimization Inlining & specialization Instruction scheduling	CSE Copy propagation Constant propagation Loop invariant code motion Dead code elimization Inlining & specialization Instruction scheduling
Register allocation	1-pass dynamic	Graph coloring	2-pass dynamic	1-pass dynamic
Garbage collection	Incremental copying	Incremental copying	Incremental mark & sweep	Incremental mark & sweep
Compiler size	700KB	1.5Mb	325KB	200KB
Interpreter size	220KB	220KB	65KB	None

All the fast register allocation schemes share more with each other than with graph coloring. The most important common characteristic is that they consider allocation using a limited view of interference. Additionally, these algorithms handle spills dynamically at points in the program where there are no free registers.

LaTTe's register allocation system probably bears the closest resemblance to our implementation [19]. Their algorithm uses three passes for each block: a scan that computes estimates of live variables and their last uses, a backward sweep that computes preferred register assignments, and a final forward sweep which allocates registers and removed unnecessary copies.

Compared to the LaTTe JIT, our on-the-fly allocator requires one less pass over the code. We compute liveness (derived from our local and global use counters) and perform register preallocation in the same step. Additionally, we integrate these two computations into the DFG generation stage so that only one pass is required for register allocation (concurrent with code generation). At each allocation point, the LaTTe allocator also has less information to make good spill decisions and register class selections.

Linear scan allocators direct global allocation of register using a linear sweep of the program being compiled [4][17]. The basic linear scan allocator uses a simple view of liveness known as a lifetime interval, which spans from a variable definition to where it is last live in linear program order. Each step of the algorithm tracks the active lifetimes at a given program point. When there are more active lifetimes than available registers, the longest active lifetime is spilled.

Although the basic linear scan algorithm is probably the fastest allocator we describe here, its representation of liveness is probably the most imprecise. An algorithm has been proposed to improve the precision of lifetime intervals [17], at the cost of an additional dataflow analysis pass, and a scan to resolve assignment conflicts at basic block boundaries.

5. Experiment Results

5.1 Setup

The microJIT was developed on a commercial version of the open source Kaffe virtual machine (http://www.transvirtual.com). We compared our JIT compiler against three other compilers that target the SPARC ISA. Characteristics of these compilers are shown in Table 2. The SPARC architecture was chosen because it had the largest availability of good compilers for which source code could be found, and because we wanted performance results from a neutral RISC architecture. We could have also chosen the x86 platform, but we were concerned that its small register file might skew the results by eliminating most possibilities for register allocation.

We included the two dynamic compilers from the Sun JDK, the client and server compilers [9][6]. The server compiler uses the powerful, but expensive static-

single assignment (SSA) representation internally. This compiler is not optimized for fast compilation times, but generates extremely good code through traditional dataflow analysis. The client compiler does not perform any advanced analysis, but focuses on basic register allocation and inlining optimizations. Both of these compilers run under the HotSpot VM, which only compiles frequently called methods and interprets otherwise.

The other compiler included in our experiment is the LatteVM [19]. This relatively fast dataflow compiler implements many of the optimizations performed by the Sun-server compiler. This VM also supports lazy compilation, although it appears to use very little interpretation during exeuction. This compiler was not designed to be ported to different ISAs as the IR maps closely to the SPARC ISA.

We used perfmon (http://web.cps.msu.ed/~enbody/perfmon.html), a library interface to the UltraSparc2 hardware counters, to time compilations. This was necessary to get accurate times for the compilation of smaller methods. The UNIX time command was accurate enough for code execution times. All VMs ran on the same machine (200MHz UltraSparc2 w/ Solaris 8).

Table 3. Method bytecode sizes by benchmark.

Benchmark	Method bytecode size				
	<50B	50-250B	250B - 1KB	1K – 5KB	>5KB
mp3 – mp3 decoder	131	70	17	4	1
mtrt – raytracer	128	34	13	2	1
jess – expert system	289	60	11	0	0
compress – compression	35	8	4	0	0
db – database	16	9	0	0	0
jlex – parser gen.	47	33	23	3	3
deltablue – planner	52	15	1	0	0
richards – task simulator	306	54	4	0	0
java_cup – parser gen.	122	30	7	0	0
moldyn – particle simulation	13	15	3	1	0
search – alpha beta search	15	20	4	0	0
h263dec – video decoder	40	24	20	11	3
pizza – java compiler	327	194	40	9	0
euler – fluid dynamics	14	14	4	5	0
jpeg – image compression	237	75	54	16	0
mips_sim – cpu simulator	25	25	9	2	0
scimark2 – fp loops	14	13	2	0	0
jbytemark – int & fp loops	47	64	15	0	0

Table 3 lists the benchmarks used to evaluate the performance. In choosing benchmarks, we tried to include a variety of programs to represent both numerical and object-oriented programs. Most the larger benchmarks are part of the Spec JVM98 (http://www.spec.org/) and Java Grande (http://www.epcc.ed.ac.uk/javagrande/javag.html) benchmark suites. Scimark2 and jBYTEmark are benchmarks suites comprised of smaller kernels.

5.2 Compilation Time

Compilation times, decomposed by method bytecode size, are shown in Figure 8. We normalized to bytecodes processed per 1k cycles to accommodate varying method bytecode sizes within each bin. In this figure (and subsequent figures), two bars show microJIT's compilation performance: advanced optimizations (scheduling, inlining, loop opt) are enabled for microJIT1 and disabled for microJIT2. For methods <1000b, microJIT's compilation times are always better, averaging over 2.5x faster than the closest dataflow compiler, LaTTe, and 12x faster than the Sun-server compiler. Relative compilation times may be even better against the LaTTe compiler because it is heavily optimized for SPARC, and may incur additional overheads to support multiple target ISAs. While we are only about 30% faster than the Sun-client compiler, the client compiler performs fewer optimizations than the microJIT, LaTTe, and Sun-server compiler.

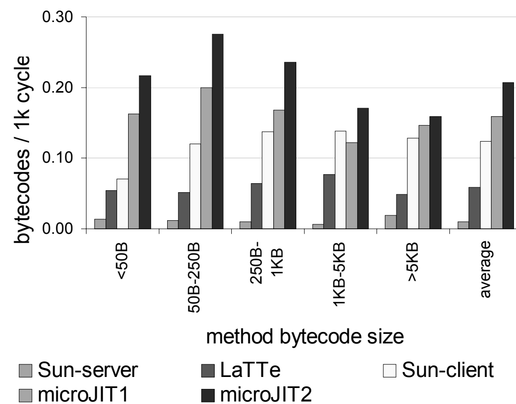

Figure 8. Compilation rate for given method bytecode sizes in bytecodes processed per 1k cycles.

Figure 9 shows the effect of various optimizations on compile time (CSE is always on by default). Inlining results in about a 10% slowdown and scheduling caused an average 15% penalty. The cost of loop optimizations are relatively cheap expect for large methods, where it adds over 20% to the compilation time.

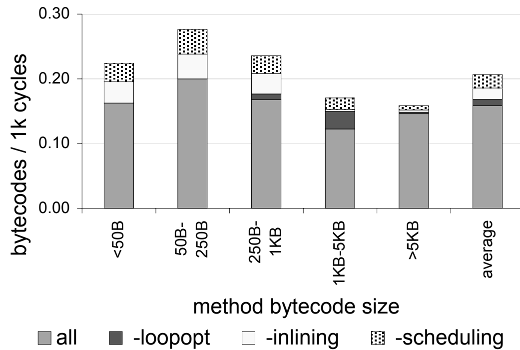

Figure 9 Compilation rate for microJIT with different optimizations disabled, in bytecodes processed per 1k cycles.

Figure 10, also decomposed by method bytecode size, breaks down time spent in each pass of our compiler. For very large methods (>1000b), time spent in DFG generation dominates almost 70% of compilation time. We attribute this shift to the high cost of CSE. As methods get large, we expect regions searched for CSE will grow, along with the number of expressions on which CSE is applied, resulting in non-linear computational cost. We believe this is also the primary cause of decreasing compilation speeds for larger methods. If this effect is undesirable, one possible fix is to limit the depth of backward searches performed by CSE.

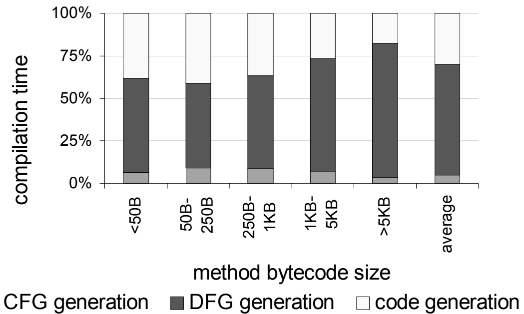

Figure 10. Times spent in each pass of microJIT.

5.3 Generated Code Performance

Performance of the code generated by the compilers is shown in Figure 11 (long running applications, large data sets) and Figure 12 (short running applications, small data sets). Benchmarks included in both graphs (like compress, db, jess, mp3, and mtrt) are run with different input data set sizes. Reported performance times are for total running time, including compilation, interpretation (if any), garbage collection, and native execution. For comparison, performance of the original Kaffe JIT and a Sun JDK using only interpretation are also included. The performance in Figure 11 is expressed as speedup normalized to the JIT compiler used in the JDK1.1 to compensate for the different sizes of each benchmark. This JIT compiler is a suitable baseline because it is a relatively conservative dynamic compiler that does not attempt any advanced optimizations. Short run execution times in Figure 12 are represented in seconds and have been divided into interpretation, compilation

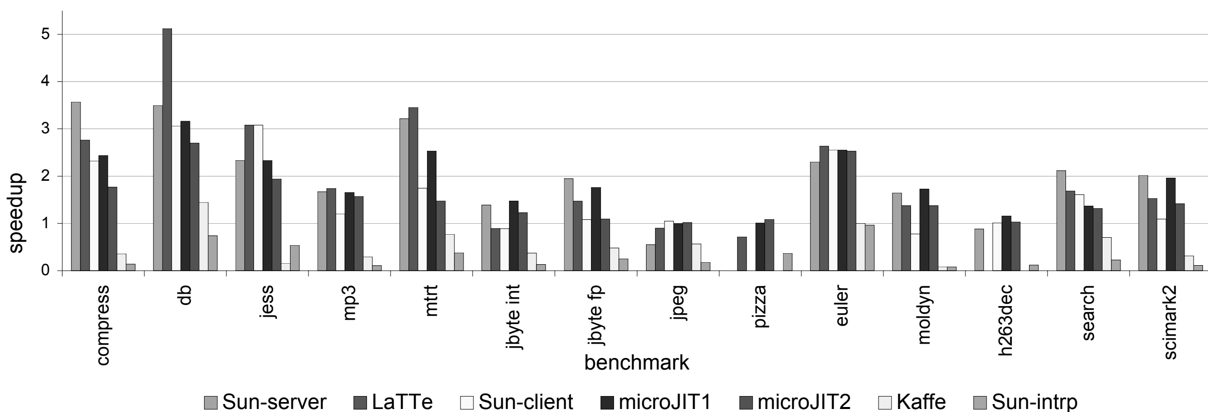

Figure 11. Speedup of large benchmarks relative to JDK1.1 (Sun's pre-Hotpot JIT).

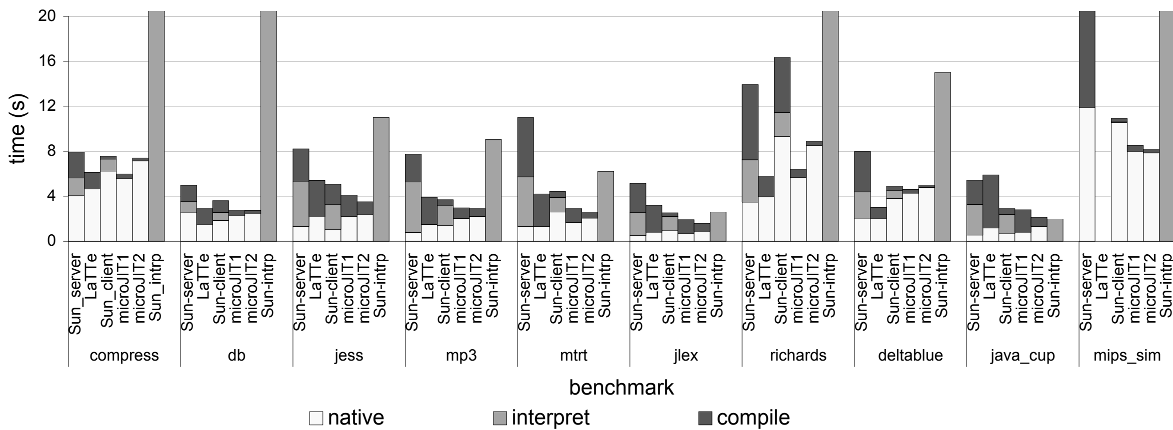

Figure 12. Performance on short running benchmarks.

and native execution times. (Note: missing bars in the graphs represent programs that we could not collect results for)

For long running applications, microJIT performs well on numerically intensive applications (e.g. mp3, euler, moldyn, jpeg, h263dec, jBYTEmark, and scimark2). While the LaTTe and Sun-server compilers still produce better code, microJIT is able to outperform the Sun-client compiler on many applications. On short running applications, microJIT's low compilation times allow it to keep total execution time small relative to the other systems.

Overall, we are most disappointed by our performance on object-oriented applications like db and jess. db's execution time is largely dominated by a loop nest within a shell sort routinue. For this benchmark, we suspect aggressive array bounds check elimination within the loop nest allows LaTTe to perform particularly well on this program.

To understand further the quality of code generated by our compiler, we also decomposed execution times of some of the long running benchmarks, given in Figure 13. Results are normalized to the Sun-server compiler and include garbage collection times. This graph suggests that one factor limiting performance of microJIT code is inefficiencies in our garbage collector. On applications that allocate memory intensively, our system spends a larger percentage of time in collection than other virtual machines, deteriorating its relative performance.

Another performance limitation could be from our naïve implementation of specialization. The Sun-client and server compilers support a particularly fast form of specialization using class hierarchy analysis (CHA) and deoptimization [9]. Non-final, public virtual and interface calls that have only one target class can be inlined directly without a check to verify the correct target object class. If dynamic class loading causes this virtual or interface call to have more than one target class, the methods can be recompiled with these optimizations removed, including those on the current

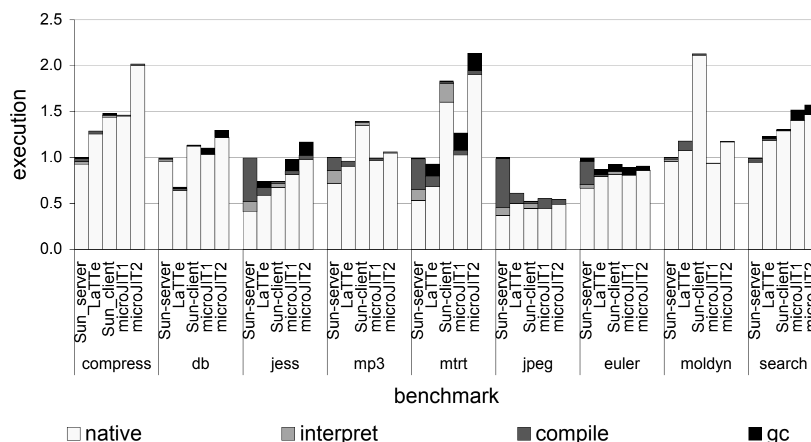

Figure 13. Performance of large benchmarks normalized to Sun-server.

call stack, so that the program will run correctly.

5.4 Static Memory Usage

Total size of the compilers and associated interpreter (if any) are shown in Table 2. These numbers were obtained by taking associated object files and applying the UNIX `strip` to them to remove unnecessary symbols. At 200KB, microJIT's static memory requirements are smaller than the other compilers. Total static memory requirements are further improved by omission of an interpreter in our system. While Sun-server and Sun-client may be larger because they are written in C++, and support the profiling (JVMPI) and debugging (JVMDI) interfaces, we believe these differences should not dramatically affect static memory comparisons.

5.5 Dynamic Memory Usage

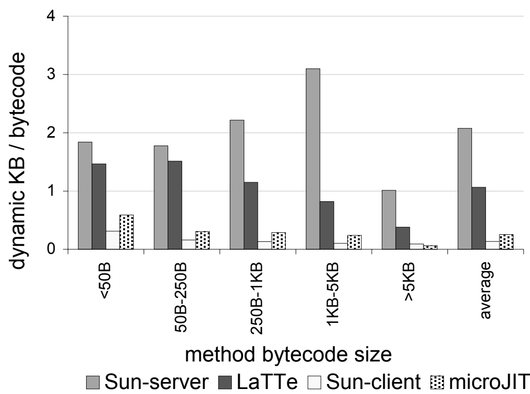

Figure 14. Dynamic memory required during compilation.

An important consideration for dynamic compilation in an embedded system is the limited dynamic memory available to the compiler. Figure 14 shows dynamic memory used by the compilers during compilation. On average, microJIT uses 25% of the memory required by the LaTTE compiler and 12.5% of the memory required by the Sun-server compiler, but it uses twice the memory required by the Sun-client compiler. These numbers suggest a 250KB buffer is sufficient memory for microJIT to compile method bytecodes less than 1KB.

To limit the dynamic memory required by the compiler for larger method bytecodes (> 1KB), microJIT could be amended to support partial compilation. In this mode, microJIT's first pass (CFG construction) would execute normally and generate the CFG of all the BBs in the method. The DFG generation and code generation passes would then execute as before, but only on sections of the CFG at one time (e.g. one EBB or loop nest). This relies on the observation that the bulk of dynamic memory used by the compiler is for the intermediate representation of the bytecodes. By only generating the intermediate representation for subsections of a method at one time, we can reduce total dynamic memory requirements. This possible improvement to microJIT would reduce dynamic memory requirements at the cost of limiting some global optimizations for large bytecode methods.

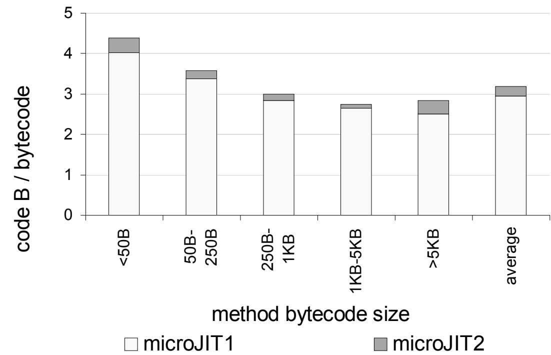

Figure 15. Code expansion of native code after translation of bytecodes.

The other important dynamic memory consideration in an embedded system is memory used to store translated code. The effects of a limited code buffer on total system performance were beyond the scope of our study (e.g. choosing which translated methods to discard and factoring the cost of recompilation when the code buffer is full), but we did collect statistics on code expansion resulting from translation. Figure 15 shows the average number of bytes of native code (code and data segments) generated per bytecode translated by mciroJIT. microJIT1 (with all optimizations enabled) generated less code primarily due to filling of branch delay slots by the instruction scheduler. We also did not observe any dramatic differences in code expansion between the compilers evaluated. We found the largest benchmarks evaluated here (jpeg and pizza compiler) generated at most 300KB of native code.

6. Conclusions

We have demonstrated how a fast dynamic optimizer can be constructed that includes advanced optimizations without incurring high compilation costs or having high memory requirements. This was accomplished by minimizing compiler passes while optimizing aggressively and by efficient communication and representation of flow information. Unlike traditional dataflow compilers that solve

dataflow equations and apply optimizations successively, we perform local and global optimizations as the IR expressions are generated. Additionally, we allocate registers concurrently with code gneration using an on-the-fly allocator that utilizes local interference, liveness, and register classes when making allocation and spill decisions.

Our experiment shows that the tradeoff between short compile times and high code quality may be less pronounced than commonly believed. This result suggests that we can incorporate small dynamic compilers into resource-constrained environments where high compile times, poor code quality, and the cost of more expensive systems cannot be tolerated.

7. Acknowledgements

The authors wish to acknowledge Tim Wilkinson and Transvirtual Technologies for their support of the Kaffe JVM used in this study. This work was supported by DARPA contract MDA904-98-C-A933.

8. References

[1] Adl-Tabatabai, A.R. et al. Fast, Effective Code Generation in a Just-In-Time Java Compiler. In SIGPLAN'98, Montreal, Canada, 1998.

[2] Alpern, B. et al. Implementing Jalapeño in Java. In OOPSLA'99, Denver, Colorado, November, 1999.

[3] Bala, V., Duesterwald, E., and Banerjia, S. Dynamo: A Transparent Dynamic Optimization System. In PLDI'00, Vancouver, BC, Canada, June, 2000.

[4] Blickstein, D.S. et al. The Gem Optimizing Compiler System. Digital Equipment Corportation Technical Journal, 4(4):121-135, 1992.

[5] Cierniak, M., Lueh, G. Y., and Stichnoth, J. Practicing JUDO: Java Under Dynamic Optimizations. In PLDI'00, Vancouver, BC, Canada, June, 2000.

[6] Click, C. High-Performance Computing with the Server Compiler for the Java HotSpot Virtual Machine. In JavaOne 2001, San Francisco, CA, June, 2001.

[7] Engler, D.R. vcode: a retargetable, extensible, very fast dynamic code generation system. In PLDI'96, Philadelphia, PA, May, 1996.

[8] Gosling, J., Joy, B., and Steele, G. The Java Language Specification. Addison Wesley, Reading, MA, 1996.

[9] Griessemer, R. and Mitrovic, S. The Java HotSpot Virtual Machine Client Compiler: Technology and Application. In JavaOne 2001, San Francisco, CA, June, 2001.

[10] Holzle, U. et al. Java On Steroids: Sun's High-Performance Java Implementation. In Hot Chips '97, Stanford, CA 1997.

[11] Lindholm, T. and Yellin, F. The Java Virtual Machine Specification. Addison Wesley, Reading, MA, 1997.

[12] Muchnick, S. Advanced Compiler Design Implementation. Morgan Kaufmann Publishers, San Francisco, CA, 1997.

[13] Poletto, M. Language and Compiler Support for Dynamic Code Generation. PhD thesis, MIT, 1999.

[14] Poletto, M., Engler, D.R., and Kaashoek, M.F. tcc: A System for Fast, Flexible, and High-level Dynamic Code Generation. In PLDI'97, Las Vegas, NV, June, 1997.

[15] Rubin, N. and Chernoff, A. Digital FX!32: A Utiliity for Fast Transparent Execution of Win32 x86 Applications on Alpha NT. In Hot Chips '97, Stanford, CA, Auguest, 1997.

[16] Suganuma, T. et al. Overview of the IBM Java Just-in-Time Compiler. In IBM Systems Journal, Vol. 39, No. 1, 2000.

[17] Traub, O., Holloway, G., and Smith, M.D. Quality and Speed in Linear-scan Register Allocation. In SIGPLAN'98, Montreal, Canada, 1998.

[18] Witchel, E. and Rosenblum, M. Embra: Fast and Flexible Machine Simulation. In ACM SIGMETRICS '96, Philadelphia, PA, 1996.

[19] Yang, B.S. et al. LaTTe: A Java VM Just-in-Time Compiler with Fast and Efficient Register Allocation. In PACT'99, New Port Beach, CA, October, 1999.