

Code Generation for Just-in-Time Compiled Mobile Collector Agents

John G. Allen and Jesse. S. Jin

Biomedical and Multimedia Information Technology (BMIT) Group

School of Information Technologies

Madsen F09, University of Sydney, NSW 2006

{jallen, jesse}@it.usyd.edu.au

Abstract

This paper describes MGEN/x86, a toolkit that simplifies the process of creating dynamic code generators and just-in-time (JIT) compilers for the x86 series of processor. MGEN produces pattern-matching code generators based on a user-defined sequence of rules and semantic action associated with each rule.

MGEN includes a number of useful and interesting features such as a table driven macro compiler and a rule based code generator-generator. The rule-based syntax permits pattern-matching instruction selection with a small degree of intermediate representation (IR) operator look ahead against a compile time stack of states.

Wildcards may be used to reduce the number of required rules for instruction selection and may assist in reducing the complexity associated with introducing new CPU states by providing a default action at each node.

Final year computing students have successfully used the MGEN toolkit to produce syntactically complete just-in-time compilers. The purpose of this paper is to introduce the toolkit as a simple method of performing compilation of mobile collector agents.

Keywords: mobile, collector, JIT, compiler, assemble, x86.

1 Introduction

Due to the large size of multimedia objects (movies, pictures) it is not feasible to perform collection and summarization using the same methods typically used to index plain text content such as hypertext documents.

The concept of mobile collection refers to remotely processing multimedia objects in order to perform feature extraction and summarization. This process alleviates the need to download the actual objects since the document processing is performed at the server in which the object resides.

In this process we need only download those features selected by the mobile collector, which typically has dimensions that are many times smaller than the object itself. To further reduce bandwidth requirements, the collector may choose to compress the features vector using some arbitrary compression scheme.

There exists a need for a remote indexing system that is specialised for the task of performing collection and summarization of multimedia content. Our goal is to produce a system that may compile the collector agents from whatever language specification into native or virtual machine code to be executed at the remote site.

Current generations of computer architectures support instructions that exploit data parallel redundancy by allowing processing of multiple data elements per instruction cycle. Such architectures provide Single Instruction Multiple Data (SIMD) instructions that can improve the performance of multimedia applications by several orders of magnitude versus scalar code.

This work aims to develop an x86 code emission strategy for the compilation of collector agents that supports the use of Streaming SIMD Extensions (SSE) and MMX™ Technology.

2 Background

Previous work on mobile collection and summarization proposed a Java based system called Cyberbroker (Jin and Kurniawati, 2000, 2001) that implemented foreign collection via Remote Method Invocation (RMI) of Java class methods.

Previous work has addressed the issues of authentication and security as well as the overall feasibility of mobile versus centralized collection.

This work has left open the possibility of exploring more explicit compilation of the collector agents that may add features such as video and sound processing and possibly introduce the use of processor enhancement technologies to improve performance.

Pattern-matching code generators map patterns of IR operators to equivalent target instructions for the target machine. They have been used by Fraser and Proebsting (2000) to create small fast generators that fit into an 8KB instruction cache on the x86 processor.

The small memory requirement of these code generators makes them ideal for use in small computers and imbedded devices that have limited memory resources. As a side benefit they are fast, produce good code quickly and are conceptually simpler than tree-based matchers.

Tree-matching code generators such as those described by Fraser, Hanson and Proebsting (1992) require two passes to perform ideal instruction selection based on least cost mappings. The initial pass is bottom up and involves finding a set of rules that cover the tree with

minimum cost. The second pass involves executing the semantic actions associated with minimum cost patterns at the selected nodes in order to emit object code.

Just-in-time (JIT) translation from postfix-based intermediate representation (such as Java bytecode) to target-specific native code has received a degree of attention in recent times due to the popularity of machine independent programming languages like Java (Sun, 2002), and more recently the Microsoft .NET Common Language Infrastructure (Microsoft, 2002).

A number of open source projects now reveal the methods they use to perform JIT translation including Intel (2002), Kaffe (2002) and Ximian (2002). Many recent JIT compiler systems have adopted tree based Bottom Up Rewrite System (BURS) instruction selection (Fraser, Hanson and Proebsting, 1992) and macro based emission of assembly instructions.

3 Dynamic Code Generation

Dynamic assemblers must emit individual instructions quickly and for this reason a macro expansion style approach is favoured. A downside of this method is that we often require multiple macros per instruction to encapsulate the variety of addressing modes and operand sizes for any particular instruction we wish to emit.

A further requirement of our JIT is that the assembly process must support the use of MMX and SSE instructions. There are currently a variety of x86 multimedia processor architectures in use, including Intel's MMX, SSE/SSE2 as well as AMD's 3DNow! technology.

While a macro approach is potentially suitable for handling some of these architecture elements, it is realised that a more sophisticated mechanism is required to support all available assembly instructions in the same capacity as a table driven static assembler.

We have experimented with a number of approaches to dynamic code generation. Our preferred method is a table driven pre-processor that translates a specialised directive into inline macrocode that emits the desired assembly instructions.

3.1 Code Generation Macros

The Intel and Ximian JIT compilation projects utilize code generation macros that originate from the Intel Open Research Platform (Intel, 2002).

We have added support for the basic set of MMX instructions to the existing macros set and in addition have developed a mechanism for inferring new code generation macros quickly where the developer needs only a minimal knowledge of x86 instruction encoding.

3.2 Table-Driven Macro Compiler

The use of programs that write programs is quite common in the context of static as well as dynamic and just-in-time compilation (Ximian, 2002, Hanson and Proebsting, 1992).

We have applied the same principle to the assembly emission component of dynamic compilation by allowing instructions to be encoded using a simple format string. The format strings are translated into pre-processor macros (in the C language) that emit native code for that instruction.

The Kaffe Java virtual machine (Kaffe, 2002) allows higher-level instructions to be declared using a define directive. The definitions are at a more abstract level than individual assembly instructions and are designed to be retargetable to other architectures. In contrast, our toolkit provides functionality that could be incorporated at a lower level to provide assembly services specific to the x86 series of processor.

This mechanism allows the functionality of a table driven approach without the need to store or decode instruction tables at runtime. In practice this method has saved over 500KB of static and dynamic storage for instruction tables and can save several hundred cycles of CPU time per emitted instruction.

The table-driven macro compiler uses a format string concept to eliminate the need to manually write assembly macros. The format string encodes a small set of tokens that direct macro generation for the indicated instruction (Table 1). Some instruction encoding examples are shown with their equivalent MASM syntax in Figure 1.

```
/* add eax, 4 */
x86_emit(c, "add /r /8", EAX, 4);

/* mov ecx, dword ptr [esp+16] */
x86_emit(c, "mov /r /[r+d]", ECX, ESP, 16);

/* paddb mm0, mm1 */
x86_emit(c, "paddb /m /m", MM0, MM1);

/* jeq short label_1: */
x86_emit(c, "jcc /s /imm", CC_EQ, offset);

/* inc dword ptr [var] */
x86_emit(c, "inc /d /[32]", &var);
```

Figure 1: Sample Instruction Encoding

The emit directive specifies a constant format string that is parsed by the pre-processor and replaced by a call to the compiled code generation macro. The resulting macro from the encoding "add /r /8" is given in Figure 2.

```
#define x86_add_reg32_imm8(code,reg1,imm) \
do { \
    *(code)++ = (unsigned char)0x83; \
    x86_reg_emit((code), (0), (reg1)); \
    *(code)++ = (unsigned char)(imm); \
} while(0)
```

Figure 2: A Generated Macro

Token	Description	Argument ⁺
/[32]	Operand is a 32-bit memory immediate memory address	&var
/[r]	Operand is an indirect memory reference with a 32-bit base register	ESP
/[r+d]	Operand is an indirect memory reference with a 32-bit base register and displacement	ESP, 16
/[r+r*s+d]	Operand is an indexed memory reference with a base register, index register and displacement	ESP, EAX, 4, 16
/8 /16 /32	Operand is a 8-, 16- or 32-bit “generic” immediate value	100
/b /w /d /q	Apply a byte, word, double word or quad word specifier to the next operand	
/eax /ax /al /ecx /cx /cl /dx	Operand has an explicit general-purpose register encoding. No argument is required for these special registers however there must be a matching instruction encoding that accepts explicit selection of the indicated register	
/i /imm	Operand is a generic immediate	100
/m /mmx	Operand is an MMX register (MMX instructions only)	MM0
/r32 /r /reg	Operand is a 32-bit general purpose register	EAX
/r8 /r16	Operand is an 8- or 16-bit general purpose register	AX
/s /n /f	Apply a short, near or far specifier to the next operand. Synonyms are /short, /near and /far	
/st /to /st0	Operand is a floating-point stack register selected from the argument list, except for /st0 which encodes ST0 explicitly. The modifier /to may be used to select a target other than /st0	ST1
/u	Select unsigned comparison semantics (conditional instructions only)	
/x /xmm	Operand is an XMM register (SSE instructions only)	XMM1
jcc setcc cmovcc	Instruction is a conditional jump, conditional move, or a set instruction. The first argument in the argument list must be the desired condition code	CC_NE

Table 1: Token Set for the Dynamic Assembler Format String

4 Pattern Matching Code Generation

GBURG (Hanson and Proebsting, 1992) produces a finite state machine (FSM) that maps patterns of linearized IR operators into native instructions that the target can execute.

The FSM method produces small and efficient code generators that accept a subset of operators at any state depending on the input tree grammar. The tree grammar is equivalent in descriptive power to regular expressions and cannot model arbitrarily complex tree structures without an infinite degree of IR look ahead.

4.1 The MGEN Toolkit

Our system produces a code generator based on a lazy code generation scheme with simplified greedy register allocation. The lazy instruction selection takes advantage of IA-32 addressing modes by folding loads of operands against a virtual (mimic) stack of CPU states.

The toolkit is dependant on the host compiler to generate efficient binary-sorted switch statements or jump tables to implement the instruction decoder and thus produces code generators that are less efficient than those implemented as an FSM. This execution behaviour allows greater expressive power in defining the input functionality of the code generator. One

example is simplification at any node by matching patterns of wildcards against the states at the top of the code generation stack. For instance, the rule in Figure 3 simplifies unmatched states for the addition instruction by performing ad hoc register allocation.

```

%ADD+I(REG32, REG32, CNST+I, ADD+I) %{
  x86_emit(c->code, "lea /r [r+r*s+d]",
    $1.u.reg, $1.u.reg, $2.u.reg, 0, @1->v.i);
  rfree(c, &$2);
%}
%ADD+I(REG32, REG32) %{
  x86_emit(c->code, "add /r /r", $1.u.reg, $2.u.reg);
  rfree(c, &$2);
%}
%ADD+I(REG32, CNST32) %{
  x86_emit(c->code, "add /r /32", $1.u.reg, $2.u.i);
%}
%ADD+I(CNST32, REG32) %{
  x86_emit(c->code, "add /r /32", $2.u.reg, $1.u.i);
  $$ = $2;
%}
%ADD+I(REG32, *) %{ ralloc(c, &$2); return(0); %}
%ADD+I(*, REG32) %{ ralloc(c, &$1); return(0); %}

```

Figure 3: Sample Instruction Selection Rules

Register selection is greedy and is performed within the semantic action associated with a selected pattern. Register allocation is not required for non-terminals where an appropriate code-generation rule is found.

⁺ The argument column illustrates an example set of parameters for the given token

Register allocation poses a major difficulty in this type of code generation system because the number of spills and unallocated registers available for temporaries is not known at the time the activation record is configured. Register allocation for JIT compiled systems is well researched with several systems implementing a variety of allocation schemes that allow local variables to be assigned to registers for part or all of their lifetime. Latte (1999) and Ximian (2002) describe some such schemes.

MGEN allows a non-infinite degree of instruction look ahead to be included within pattern matching rules. This typically permits the code generator to perform more optimal instruction selection and permits the exploitation of addressing modes available on the target. The first rule in Figure 3 is shows an example of this feature, with the rule emitting a Load Effective Address (LEA) instruction.

Note that it is the responsibility of the underlying generated macro system to decide what sized immediate to emit for the displacement field. In interpreting the code generation rules, we refer the reader to Table 2.

Token	Description
\$\$	Reference element at top of mimic stack
\$N	Reference matching stack elements
@@	Reference matching node operator
@N	Reference look-ahead node operator
*	Match any stack element (wildcard)

Table 2: MGEN Special Symbols

5 Conclusion

MGEN is an early attempt at a dynamic compilation system that supports multimedia instructions as part of its dynamic code emission strategy. The inclusion of these instructions is seamless and provides the additional benefit of permitting a broader local instruction emission spectrum than traditional macro based approaches.

The code emission strategy employed allows rapid prototyping and development of just-in-time compiler systems and offers the developer many time saving features by encapsulating the basic instruction selection and code emission requirements for dynamic compilation.

Early projects using this system have implemented JIT compilers for the High Order Calculator (HOC) learning language (Kernigan and Pike, 1984) that achieved a performance improvement of between 10-100 times over the original virtual machine.

For most applications, code generated using this system comes within 2-3 times the performance of scalar code produced by the Microsoft Visual C++® compiler. This figure does not take into account the

performance improvement gained when dynamic SIMD (MMX, SSE, 3DNow!) instructions are emitted.

6 Future Work

This work has provided a simple foundation that removes much of the time and complexity associated with producing dynamic compilers and has been used to aid 3rd year teaching. Future work will involve using alternate techniques to perform compilation of mobile collector agents that make use of the dynamic assembler presented in this paper.

References

- MICROSOFT (2002): Microsoft .NET Website <http://www.microsoft.com/net/>.
- SUN (2002): Sun Java Website. <http://java.sun.com>
- XIMIAN (2002): Mono Project - An Implementation of the .NET Development Framework. <http://www.go-mono.org>
- INTEL (2002): Open Source Dynamic Computing Research Platform (ORP). <http://orp.sourceforge.net/>
- KAFFE (2002): Kaffe Java Virtual Machine Implementation. <http://www.kaffe.org>
- JIN, JS. and KURNIAWATI, R (2001). Implementation of an On-Site Web Multimedia Collector. *Computer Communications* 24(7-8): 716-723.
- JIN, JS. and KURNIAWATI, R (2000). Cyberbroker: A Lightweight Web Object Collector. *World Wide Web* 3(3): 185-191.
- YANG, B., MOON, S., PARK, S., LEE, J., LEE, S., PARK, J., CHUNG, Y. and KIM, S (1999): LaTTe: A Java VM Just-in-Time Compiler with Fast and Efficient Register Allocation. *Proc. IEEE PACT99 International Conference on Parallel Architectures and Compilation Techniques*, New Port Beach, CA, USA, 128-129, IEEE, Inc.
- ADL-TABATABAI, A.-R., CIERNIAK, M. LUEH, G.-Y., PARIKH, V.M. AND STICHNOTH, J. M. (1998): Fast, Effective Code Generation in a Just-in-Time Java Compiler. *Proc. SIGPLAN '98 Conference on Programming Language Design and Implementation*, 280-290.
- FRASER, C. and PROEBSTING, T. (1999): Finite-State Code Generation. *Proc. ACM SIGPLAN Conference on Programming Design and Implementation*. 22:207-216, ACM Press.
- FRASER, C., HANSON, D. and PROEBSTING, T. (1992): Engineering a Simple, Efficient Code Generator Generator. *Proc. ACM Letters on Programming Languages and Systems* 1, 3. 213-216, ACM Press.
- KERNIGAN, R. and PIKE, R. (1984): *The Unix Programming Environment*. Prentice Hall Computer Books.