

Clarity MCode: A Retargetable Intermediate Representation for Compilation

Brian T. Lewis
L. Peter Deutsch
Theodore C. Goldstein

SMLI TR-95-43

May 1995

Abstract:

The Clarity C++ programming language is a dialect of C++ being developed in Sun Microsystems Laboratories to support the development of reliable systems and application software, and especially distributed software. We have developed a high-level, machine-independent intermediate representation to compile Clarity that we call MCode (for "middle code"). We use MCode to compile Clarity programs at program runtime (i.e., on-the-fly) into SPARC[®] code for the Solaris[™] operating system. The runtime code generator produces good quality machine code and is designed to be easily retargetable to new machines. We also support an interpreter for MCode that supports full interoperability with C code and existing C libraries. The choice of whether to compile or interpret MCode is done at runtime on a procedure-by-procedure basis. MCode is significantly more compact than native machine code. This fact, plus our ability to selectively interpret seldom-executed code instead of compiling it, means that MCode programs have a smaller working set and run better with less memory than the corresponding native machine code programs. In order to support systems programming, we store MCode in platform-standard object files. This enables MCode programs to fully interoperate with existing C libraries and code. It also allows programmers to use standard linkers and other program development tools with MCode object files.

 *Sun Microsystems*
Laboratories, Inc.
A Sun Microsystems, Inc. Business
M/S 29-01
2550 Garcia Avenue
Mountain View, CA 94043

email addresses:
brian.lewis@eng.sun.com
lpd@aladdin.com
ted.goldstein@eng.sun.com

Clarity MCode: A Retargetable Intermediate Representation for Compilation

Brian T. Lewis
Theodore C. Goldstein
Sun Microsystems Laboratories
2550 Garcia Avenue
Mountain View, California 94043

L. Peter Deutsch
Alladin Enterprises
203 Santa Margarita Avenue
Menlo Park, California 94025

1 Introduction

The *Clarity* C++ programming language is a dialect of C++ in development at Sun Microsystems Laboratories. Clarity shares many features with C++, but is less complex and has a more consistent syntax and simpler semantics without loss in expressiveness. It is influenced by the interface definition language IDL [OMG 91] in that it strictly separates the interface to a software component from its implementation. It encourages the development of reliable software by incorporating ideas from Modula-3 [Nelson 91] and the formal interface specification language ADL [Sankar 94]. Clarity is intended to be a wide-spectrum language suitable for both systems and application programming, particularly of distributed software.

To support the compilation of Clarity, we have developed a high-level, machine-independent intermediate representation that we call *MCode* (for “middle code”). We use MCode to compile Clarity programs at execution time (i.e., on-the-fly) into SPARC[®] code for the Solaris[™] operating system. This code generator is designed to be largely machine-independent: besides the SPARC code generator, an Intel[®] x86 version is being developed. MCode includes a small amount of optimization information that enables the runtime code generator to produce good quality code. Our SPARC code generator produces code about as good as that produced by the SunPRO C compiler at the -O2 optimization level. A significant

advantage of MCode over native machine code is that it can be represented more compactly; MCode is stack-based, and the encoding of most instructions can be a single byte. We also support an interpreter for MCode that supports full interoperation with C and existing C libraries. Although MCode includes instructions and data structures needed to implement some Clarity language-specific constructs such as its exceptions and method calls, the core of MCode is suitable for representing code for C and many other languages.

Runtime generation of machine code offers many advantages. A runtime code generator can take advantage of the information about a particular target platform to generate better code. For example, different implementations of the SPARC architecture have different instruction pipeline properties. In the case of one new SPARC implementation, code generated specifically for this new processor can run up to 25% faster than code generated for a “generic” SPARC processor. A runtime code generator can also take advantage of the specific values used in a program to generate machine code customized for those values. One of our goals is to explore the use of on-the-fly code generation for systems programming within Sun Microsystems, Inc. (SMI).

Our representation of MCode object files is unusual in that we use platform-standard object files instead of some Clarity- or MCode-specific representation. This enables us to fully interoperate with existing C and SPARC ABI (Application Binary Interface) code. Object files containing MCode (which we call *Linkable MCode* files) are processed by standard linkers and other tools in the same way as other object files. As an example, our Solaris SPARC implementation supports complete interoperation with all SPARC Application Binary Interface (ABI) compliant code [SPARC ABI]. In particular, interpreted or compiled MCode programs can call C programs and vice versa; addresses of MCode procedures can be passed to C code and later called, and all C data types can be exchanged. This level of interoperability is a requirement for systems programmers at SMI who need to use all the capabilities of existing tools. For example, programmers often *interpose* on calls to system library routines by linking first against specialized implementations of those routines before linking against the system libraries. We need to ensure that interposition routines can be implemented using Clarity.

Our experience with MCode has convinced us that:

- A stack-based intermediate representation (IR) is easy to generate but still enables the runtime generation of good quality machine code.
- A stack-based IR significantly reduces the size of object files. This size reduction is valuable even on machines with large physical memories because of the reduced paging.

- Interoperation with existing libraries and code in other languages (especially C) is essential for a systems programming language. Furthermore, it is essential to support existing linkers and other tools since programmers exploit all of their capabilities. Encapsulating MCode in platform-standard object files lets us support standard tools and enables full interoperation with existing code.

The next section gives some background about MCode and describes related work. Section 3 then discusses the MCode system in more detail with an emphasis on these three points.

2 Background

2.1 Why we chose MCode

MCode has its basis in unpublished work done by L. Peter Deutsch at Sun Microsystems Laboratories in 1992-93. This work consisted of an implementation in Smalltalk of the core of a portable, on-the-fly compiler for a subset of the C language; we will refer to this system as “CCore.” Similar to Deutsch's compilers and code generators for the Smalltalk-80TM system (described in [Deutsch&Schiffman 84]) which has been retargeted to a wide range of machine architectures and operating systems. CCore consisted of a front end that produced a simple stack-based IR, and a back end that ran at program execution time to generate machine code on demand for a target machine. The CCore back end consulted a machine description that defined all the necessary structures, parameters, and customized procedures needed to generate machine code for a particular target machine such as the SPARC. Several points about CCore are worth noting:

- Since the CCore front end could not make any assumptions about the target machine, the CCore IR contained explicit type and size information as defined by the C language semantics.
- The CCore IR instruction set was RISC-like, consisting of relatively few instructions, each designed to be easy to implement with one or a few machine instruction(s) on a RISC processor.
- The CCore IR instruction set and back end were designed from the start to be easily ported to the SPARC and several other RISC architectures, as well as the Intel x86.

At the time CCore was being built, we were starting to build both a compiler and interpreter for Clarity. We wanted a common IR and we needed portability to multiple architectures, as well as interoperability with existing libraries and source

code. CCore was attractive because of its careful design for portability. Our work since then has mostly consisted of modifying its IR for a language substantially more complex than C, finishing the code generator's design, and reimplementing the system in C++ as part of the Clarity program development environment. We also developed an interpreter and support for interoperability with existing code and libraries.

2.2 Overview and status of the MCode system

This subsection presents an overview of the Clarity MCode compilation system; more details are given in Section 3. There are two major parts to the MCode compilation system: a program development-time part and a runtime part. The development-time portion consists of an MCode generator for the Clarity language and a Linkable MCode converter; see Figure 1. The MCode generator reads semantically-decorated Clarity ASTs stored in the Clarity programming environment's database and produces platform-independent MCode. The Linkable MCode converter then wraps a compact encoding of the MCode into a standard object file. Linkable MCode object files are then combined by standard linkers with other object files to produce executables and shared libraries.

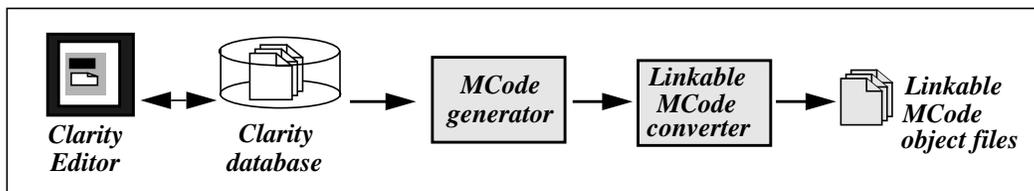


Figure 1. The development time portion of the Clarity MCode system

The runtime component of the MCode system is illustrated in Figure 2. The MCode runtime in an MCode-containing executable internalizes the MCode for each procedure as needed when the procedure is first called. It also implements an interpret/code generate policy separately for each MCode procedure. This policy chooses for each procedure whether to interpret it, generate code, interpret then later generate code, or generate better code. The code generator currently produces SPARC code of approximately the quality of the SunPRO C compiler at the default -O2 optimization level. A port of the code generator to the x86 is underway. The MCode interpreter interoperates with all SPARC ABI code. Like the compiler, it is reentrant and supports multithreaded programs. It also does extensive checking during program execution, which makes it especially useful for

uncovering errors in Clarity programs that are otherwise difficult to detect.¹ The interpreter will also be used by the Clarity debugger that we are developing to evaluate Clarity statements and expressions.

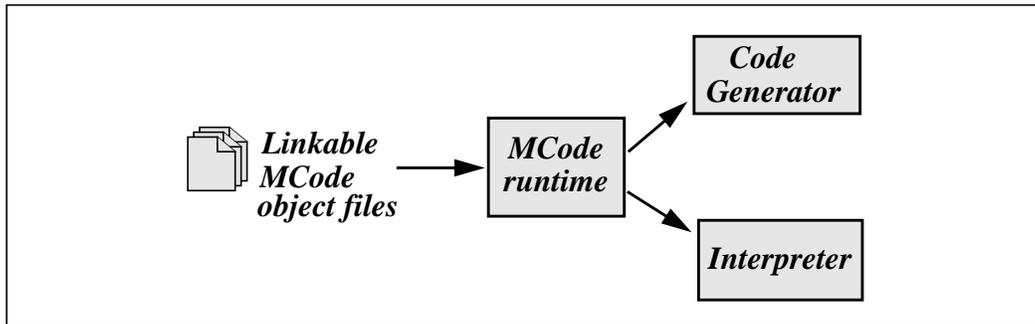


Figure 2. The runtime portion of the Clarity MCode system

The MCode compilation system has been under development for about eighteen months by two people, one of whom had other major responsibilities. All components illustrated in the two figures above are implemented. However, the system has not been optimized and is constantly being improved. As a result, we cannot give definitive performance figures. Hand examination of the machine code generated for Clarity shows code quality comparable to the SunPRO C compiler. For example, one million executions of the Clarity array shell sort procedure shown in Figure 3 requires 5 seconds for a six-element array on a 40MHz SuperSPARCTM processor; the corresponding C version takes 8.7 seconds at optimization level -O2 and 2.8 at level -O4. Performance of the MCode interpreter varies depending upon the code being executed. For example, the same million executions of the shell sort procedure using the MCode interpreter requires more than 200 seconds. Much of this extra time is due to the interpreter's extensive runtime checking. On the other hand, an interpreted Clarity version of Peter Buhr's test program for the μ C++ language [Buhr et al 92] runs at a respectable one-tenth the speed of the machine code version. This program is heavily multithreaded and makes extensive use of Solaris threads.

1. Errors in *unsafe* portions of Clarity code. Like Modula-3, Clarity requires that programmers state explicitly which portions of a program might possibly damage the program runtime state. Other, safe regions are checked by the compiler or language runtime.

```

// Sort v[0]..v[n-1] into increasing order using a shell sort.
shell_sort: proc (v: in pointer to array of int)
{
  n: int = v.number;    // set n to the number of elements in the array
  for (gap: int = n/2; gap > 0; gap /= 2) {
    for (i: int = gap; i < n; i++) {
      for (j: int = i-gap; j >= 0 && v[j] > v[j+gap]; j -= gap) {
        temp: int = v[j]; v[j] = v[j + gap]; v[j + gap] = temp;
      }
    }
  }
};

```

Figure 3. The Clarity array shell sort procedure

2.3 Related work

As described earlier, the MCode compilation system resembles ParcPlace Systems Smalltalk-80 implementation. Both systems generate a platform-independent IR used at runtime to generate platform-specific code. However, encapsulating MCode in platform-standard object files allows us to interoperate fully with C and existing libraries. This is significantly different from the ParcPlace implementation in which Smalltalk *bytecodes* are entirely encapsulated inside a single memory image; there is no support for linking. The bytecodes only implement the Smalltalk-80 virtual machine as well, Smalltalk bytecodes are incapable of supporting the full C language including its *switch* statements, addressing modes, and data formats. In Smalltalk, these constructs are implemented using other means.

The MCode system also resembles the PCode system supported by recent versions of the Microsoft[®] C and C++ development environments. PCode is a stack-based instruction set generated by Microsoft's C and C++ compilers, and then interpreted at program runtime. It is much smaller than native x86 code (about 40% of the size) and helps to reduce program memory requirements. It is especially useful for user interface code where speed is not essential. PCode is well integrated into the standard Microsoft tools. However, PCode is interpreted and not compiled (the goal is reduced memory). Also, PCode is very x86-specific.

The Java compilation system [Gosling 95] strongly resembles the MCode system in that it also supports a machine-independent intermediate representation that is

either interpreted or compiled on-the-fly. Like MCode, the Java IR is stack-based and contains a substantial amount of type information. However, Java instructions are more concrete. There are specific instructions for operating on particular sizes and kinds of the primitive data types. MCode instructions, on the other hand, are higher-level and refer to MCode type information for operand size and other instruction properties. Also, MCode's control instructions are left abstract, while the Java control constructs are represented in terms of jumps. Java is intended for building application programs, while Clarity, as a systems programming language, must support full SPARC ABI interoperation and the use of existing tools.

Michael Franz [Franz 94] has developed a compact, machine-independent IR used for on-the-fly code generation in the Macintosh[®] implementation of Oberon. This IR is a very compact encoding of the semantically-decorated abstract syntax tree for Oberon program modules. This IR has the advantage that it contains all the information in the original source, unlike MCode which does lose some information. It is an open question whether code can be generated as easily from this near source-level IR as from MCode.

Implementations of the Self programming language [Hölzle 94] use runtime code generation. Like Smalltalk, Self benefits greatly from dynamic translation because more information is known about a program's types and behavior is known at runtime. Clarity's MCode implementation already has information about program types, but it could benefit from some Self ideas for dynamically improving the generated code based on runtime program behavior.

Another related system is Jack Davidson's C compilation system [Davidson 87, 88]. His C front-end generates machine-independent CCode. CCode is interpreted by a fast portable interpreter. It is also used by an optimizing C compiler. The compiler generates CCode, which is translated into a lower-level register-transfer list (RTL) form, which is then passed to a portable optimizing code generator. The interpreter can easily be ported to new platforms. However, support for external function calls (i.e., to non-interpreted procedures) beyond a limited set requires recompiling the interpreter.

3 Some Details About the Clarity MCode Compilation System

3.1 The MCode intermediate representation

The MCode for each procedure in a code unit is essentially independent from other procedures. This enables procedure-at-a-time processing and code genera-

tion, which is useful since many procedures in a typical program (especially library procedures) are never called. MCode information shared among procedures primarily consists of information about types.

MCode's types currently include integer, real, pointer, array,² procedure, bit field, struct, union, interface, implementation, and void. Type information includes explicit size information for all primitive types. It does not include platform-specific alignment information. Only an MCode machine code generator or interpreter for a particular platform can compute the actual alignment requirement for each type, and hence, the offsets of the individual members of aggregate types and the total size for aggregates.

MCode assumes byte addressing to avoid having to distinguish byte pointers from other kinds of pointers. It also assumes a 2's complement integer representation.

MCode instructions are RISC-like. Each instruction is straightforward to implement. There are currently eighty-seven instructions, although two or three will be added when we implement Clarity exceptions; see Figure 4. One unusual feature of MCode's instruction set is that control instructions are left abstract. That is, the MCode generator does not attempt to implement control operations such as *while* and *switch* in terms of simpler instructions, but instead leaves them high-level: e.g., *BeginSwitch* and *EndSwitch*. This allows different MCode back ends to choose the best implementation for those control constructs on each target platform. Making these structures abstract aids in the generation of the most efficient code for a particular construct. For example, it allows the implementation to make the trade-offs between indexing, binary search, and hashing for *switch* statements based on the relative costs of branches and other instructions. MCode includes some optimization information. This includes variable aliasing information and a boolean for each procedure that indicates whether it is a leaf procedure or not. This boolean is especially useful on the SPARC since leaf procedures do not require a register window and faster code can be generated for them. We intend to add additional optimization information in the future, such as variable lifetime information.

2. Unlike C and C++, Clarity includes true arrays which contain a length. Accesses to arrays are checked at runtime.

LoadInt(type index, int constant)	RealToReal(target type index)
LoadReal(type index, real constant)	RealToInt(target type index)
LoadNil(nil type index)	IntToReal(target type index)
LoadInner()	IntToInt(target type index)
LoadOuter()	
	AddInt()
LoadGlobal(global ref index)	SubInt()
LoadArg(arg index)	Mullnt()
LoadLocal(local index)	DivInt()
LoadIndirect(ref type index)	ModInt()
LoadMember(type index)	CompareInt(relation code)
LoadMemberIndirect(member type index)	
LoadObjMember(member type index)	And()
	Or()
StoreGlobal(global ref index)	Xor()
StoreArg(arg index)	BitCompl()
StoreLocal(local index)	ShiftLeft()
StoreIndirect(ref type index)	ShiftRight()
StoreMember(member type index)	
StoreMemberIndirect(member type index)	AddReal()
StoreObjMember(member type index)	SubReal()
	MulReal()
StoreGlobalSave(global ref index)	DivReal()
StoreArgSave(arg index)	CompareReal(relation code)
StoreLocalSave(local index)	
StoreIndirectSave(ref type index)	AddPtr()
StoreMemberSave(member type index)	SubPtr()
StoreMemberIndirectSave(member type index)	ComparePtr(relation code)
StoreObjMemberSave(member type index)	
	PrefixInc(type index)
LocalAddr(local var index)	PrefixDec(type index)
ArgAddr(arg var index)	PostfixInc(type index)
GlobalAddr(global ref index)	PostfixDec(type index)
MemberAddr(member type index)	
ObjMemberAddr(member type index)	ProcCall(proc type index)
SizeOf(type index)	ProcReturn(return type index)
Pop()	SkipThen(begin else tag)
Dup()	SkipElse(end if tag)
Exch()	BeginElse(begin else tag)
	EndIf(end if tag)
AllocItem(item type index)	BeginLoop(loop start tag)
DeleteItem(type index)	EndLoop(loop start tag)
AllocArray(array type index)	BeginSwitch(end switch tag)
ArrayIndex(array type index)	EndSwitch(end switch tag)
ArrayLength(array type index)	BeginExprCase(int)
	BeginDefaultCase()
InvokeOuter(method type index)	EndCase(end switch tag)
InvokeDelegated(method type index)	
Widen(base interface number)	DoBreak(end tag)
Narrow(target type index)	DoContinue(end tag)

Figure 4. The MCode instruction set

Figures 5 and 6 give an example of generated MCode. The Clarity method *startup* in Figure 5 produces the MCode instructions shown (in part) in Figure 6.

```

ThreadedSimulation: module
{
  work_mutex: Threads::Mutex;
  work_per_worker: int;      // protected by work_mutex
  extra_work: int;          // protected by work_mutex

  Worker: type = interface inherits Threads::Thread {};
  // an unusual interface: no methods beyond Thread::startup and the other Thread methods

  WorkerImpl: type = implementation of Worker
  {
    implement startup: method (our_workers: in int)
    { // executed when the thread is started; delegates most of its work to forked sibling workers
      within work_mutex { // acquire work_mutex for duration of the within statement
        our_work = work_per_worker;
        if (extra_work > 0) {our_work += 1; extra_work -= 1;}
      }
      delegated_to_workers: int = (our_workers - 1);
      if (delegated_to_workers > 0) {
        left_workers = delegated_to_workers/2;
        right_workers = delegated_to_workers - left_workers;
        if (left_workers > 0) left_sibling = new WorkerImpl(left_workers);
        right_sibling = new WorkerImpl(right_workers);
      }
      do_work(our_work);
    };

    // the following declarations are private to the WorkerImpl implementation
    do_work: method (work_to_do: in int) { /* elided */ };

    our_work: int = 0;
    left_workers: int = 0;
    right_workers: int = 0;
    left_sibling: Worker;      // left delegatee; manages "left_workers" workers
    right_sibling: Worker;    // right delegatee; manages "right_workers" workers
  };
  ...
};

```

Figure 5. Part of the Clarity version of the μ C++ test program

3.2 The MCode generator

The MCode generator is straightforward in its operation, largely because of the simplicity of generating MCode. It does a recursive walk of the AST and usually

generates code for each AST node as it is visited. Our strategy is to do as much work as possible at development time to make execution as fast as possible. In keeping with this, the MCode generator does some limited optimization and includes a peephole optimizer. The generator can emit straightforward, if sometimes inefficient, code for a node because the peephole optimizer will improve the code later. The MCode generator also produces the optimization information mentioned above.

3.3 Linkable MCode

Linkable MCode object files contain a machine-independent *pickle* of an MCode code unit. This is a compact, platform-independent encoding of the MCode information into a sequence of bytes. This pickle can later be internalized or unpickled to reconstruct the original MCode. The MCode for each procedure is pickled separately to support procedure-at-a-time processing. The current encoding is not especially compact although the Linkable MCode object files are still smaller than object files containing machine code. We intend to replace the current pickle format with a more compact one.

Linkable MCode object files are platform-standard object files that are processed in the usual way by the platform's standard linker. This means that they need to include platform-dependent definitions of global variables and procedures, and descriptions of referenced symbols. We currently encode ("mangle") symbol names in order to ensure that the resulting executables or shared libraries are type-safe with respect to the Clarity language. Eventually, this type-safety will be checked by a Clarity *prelinker*. Besides symbol definitions and references, our Solaris Linkable MCode object files also contain a few machine language instructions for each procedure's entry code. This entry code allows C code to call the MCode procedure. On the SPARC, this entry code consists primarily of a three instruction "trampoline" that redirects the call to the appropriate target procedure chosen by the interpret/compile strategy module in the MCode runtime. The SPARC entry code also has three words used when atomically updating the trampoline. Despite this platform-specific information, the contents of a Linkable MCode file are mostly platform-independent. The Linkable MCode converter itself is also mostly platform-independent. We currently execute the Linkable MCode converter during program development, before a program is distributed. It could also be executed when the program is installed on a particular platform.

```

startup: method (our_workers: in int)
{ // executed when the thread is started; delegates most of its work to forked sibling worklers
  within work_mutex { // acquire work_mutex for duration of the within statement
    LoadGlobal(0)           work_mutex
    InvokeOuter(0x00010004)  Thread::enter
    our_work = work_per_worker;
    LoadGlobal(1)           work_per_worker
    StoreGlobal(5)          our_work
    if (extra_work > 0) {...
      LoadGlobal(3)         extra_work
      LoadInt(0, 0)         integer constant 0, type 0
      CompareInt(>)         extra_work>0
      SkipThen(cond_false_tag_2)  if(extra_work>0){...
      our_work += 1;
      GlobalAddr(5)         our_work
      Dup()
      LoadIndirect(0)
      LoadInt(0, 1)         integer constant 1, type 0
      AddInt()              our_work+=1
      StoreIndirect(0)      our_work
      extra_work -= 1;
      GlobalAddr(3)         extra_work
      Dup()
      LoadIndirect(0)
      LoadSigned(0, 1)      integer constant 1, type 0
      SubInt()              extra_work-=1
      StoreIndirect(0)      extra_work
      SkipElse(end_if_tag_3)  if(extra_work>0){...
      BeginElse(cond_false_tag_2)
      EndIf(end_if_tag_3)
    }
    LoadGlobal(0)           work_mutex
    InvokeOuter(0x00010005)  Thread::exit
    delegated_to_workers: int = (our_workers - 1);
    LoadArg(0)              our_workers
    LoadInt(0, 1)           integer constant 1, type 0
    SubInt()                 our_workers-1
    StoreLocal(0)           delegated_to_workers
    ...
    do_work(our_work);
    LoadGlobal(4)           do_work
    LoadGlobal(5)           our_work
    ProcCall(7)             do_work(our_work)
  };
  ProcReturn(1)             method(our_workers:...

```

Figure 6. MCode instructions generated for the μ C++ test program's startup method

3.4 MCode runtime

The MCode runtime library supports the unpickling of MCode and implements the selection of the interpret/code generate policy for each MCode procedure. The unpickling support provides a type-safe interface to the MCode information; it hides details about how that information is represented in the pickle. We expect to replace the current MCode pickle format with a more compact one without changing the source for our compiler and interpreter. To minimize program execution time, the unpickling package does as little memory allocation as possible.

Currently, our interpret/code generate policy initially interprets each procedure. When a specified call threshold is reached, it arranges for the machine code generator to be run the next time the procedure is called. This is done by rewriting the procedure's trampoline instructions to jump to the on-the-fly compiler instead of the MCode interpreter. When the compiler is finished, it rewrites the trampoline to jump to the generated machine code, and then jumps to the code itself. This rewriting is multithread-safe and is done atomically. Eventually we plan to add support for further optimizations to the code generator and extend the compile/interpret policy to allow recompilation of procedures that are called especially often.

The MCode runtime also includes an incremental garbage collector. This collector is mostly non-conservative: that is, it scans all objects on the collected heap using precise information about types and offsets, and it only scans thread stacks and static areas conservatively (i.e., a value there that is a pointer to the heap is treated as if it is a heap pointer). To support the collector, the runtime computes platform-specific information about MCode types such as the offsets of pointers within aggregates.

3.5 The machine code generator

The object-oriented architecture of the code generator has significantly simplified our implementation. The MCode machine code generator is designed to be retargetable to a new machine architecture (especially a RISC machine) with relatively little effort. It defines two key C++ base classes that must be subclassed to port the code generator. The first class, CGMachine, represents a target machine for code generation and a code stream for that machine. The basic machine model is a generic, non-windowed RISC processor. CGMachine subclasses may define variations such as windowed RISC and CISC. These subclasses implement virtual methods that describe the target machine's registers, data types, and instruction

properties. CGMachine methods then use those descriptions to generate machine code from MCode. Part of the CGMachine class definition appears in Figure 7.

```

class CGMachine {
public:
    CGMachine(CGPool<CGIntRegister>*, CGPool<CGRealRegister>*, CGModule*, CGInstructionStream*);

    CGPool<CGIntRegister>*get_int_pool()      { return int_pool; };
    CGPool<CGRealRegister>*get_real_pool()    { return real_pool; };
    CGModule*             get_module()        { return module; };
    CGInstructionStream*  get_instruction_stream() { return code; };
    CGValueStack*        get_stack()         { return &stack; };

    //----- Methods for testing machine properties -----
    // return the register holding the frame pointer for local variables
    virtual CGIntRegister* frame_register() = 0;
    // does the machine support 3-operand arithmetic instructions?
    virtual bool can_arith_3();
    // are there instructions for doing arithmetic directly from memory to a floating point register?
    virtual bool can_arith_mf();
    // should pointers be widened and compared as signed values?
    virtual bool pointers_are_signed();
    ...

    //----- Register-related methods -----
    virtual CGIntRegister* assign_int_register(int width, CGValue*) = 0;
    virtual CGRealRegister* assign_real_register(machine_data_types, CGValue*) = 0;
    ...

    //----- Methods to return machine opcodes for various MCode operations -----
    virtual CGOp::Code op_load_real(int width) = 0;
    virtual CGOp::Code op_store_real(int width) = 0;
    virtual CGOp::Code op_load_int(bool signed, int width) = 0;
    virtual CGOp::Code op_compare(CGRelationEnum op_index) = 0;
    virtual CGOp::Code op_add_real() = 0;
    virtual CGOp::Code op_mul_real() = 0;
    virtual CGOp::Code op_sub_real() = 0;
    virtual CGOp::Code op_add() = 0;
    ...
};

```

Figure 7. Part of the machine code generator’s CGMachine base class

The second C++ base class, CGValue, describes values during compilation. The code generator “executes” MCode instructions in order to maintain a running simulation of the MCode machine’s stack. Concrete subclasses of CGValue represent the state of the individual entries on the simulated stack. These entries include constants, variable references, previously “executed” subexpressions, and proce-

cedure or method calls. The simulated stack records information about operands until the MCode instructions that use them are encountered. Machine code for (sub)expressions is only generated when the value of those expressions is needed. Good code can be generated at that time because the destination (a register or memory) is known. CGValue and most of its subclasses are machine-independent. The machine-dependent subclass is the one for procedure and method calls. This is because calls are so machine-specific: for example, the way in which aggregate values are passed and returned on the SPARC is highly SPARC-specific.

The code generator includes a peephole optimizer, completes dead code elimination, and generates “leaf procedure” calls on the SPARC. However, little further optimization is done at this time; our immediate concern is generating correct code. Despite this, the code generator generates good code for the SPARC. In the future, we plan to use the aliasing information in MCode to produce higher quality code. We are looking into adapting the code generator for doing processor-specific optimizations in order to take advantage of different RISC pipeline architectures and cache protocols.

In order to do high-quality code generation, we need to rebuild the expression trees from the stack machine. The code generator defers generation until the final target for an expression is known. Much of the process is similar to that of TNBIND algorithm documented in [Wulf 75]. This algorithm gives excellent results and executes extremely efficiently. A final optimization does instruction reordering to minimize RISC processor pipeline execution conflicts. By organizing the code generator into a series of cascading object streams, we are able to consume MCode and generate native machine code in one pass. Our object-oriented architecture provides an efficient way to trade increased memory for speed.

We have not finished the Intel x86 port of the machine code generator yet, but our experience to date shows its design, even though oriented towards RISC processors, works well for the latest generations of the Intel x86 family. Its structure as a collection of object-oriented classes has proven very successful. Subclassing and encapsulation are powerful tools for organizing the components of a code generator.

3.6 The MCode interpreter

While the MCode interpreter is mostly platform-independent, about 20% of its code is platform-specific. For example, in order to fully support procedure interposition and other ABI capabilities, the SPARC MCode interpreter does not directly interpret MCode ProcCall or Invoke instructions, but instead implements them as SPARC ABI calls. Even MCode calls to other MCode procedures are

implemented using SPARC instructions and execute the procedure's machine language entry code. This is necessary because the interpreter can never know whether a called procedure is actually implemented in MCode or in C. (For example, a programmer might have replaced the called procedure using interposition.) This means the interpreter must fully handle all the details required for ABI calls. If a called routine will return an aggregate value, the interpreter must generate a sequence of machine instructions at runtime (a *thunk*) to support the SPARC ABI's calling convention that the returned aggregate's length must be encoded into a SPARC UNIMP instruction just after the call. The interpreter also stores all program values in memory as SPARC values, since this is required for ABI interoperation.

Recently, a second MCode interpreter has been developed by Mick Jordan. This interpreter executes *system models* written in the Clarity language. These system models precisely describe how a software system is built: the exact versions of its component parts, all options and build parameters, and how the component parts are assembled. This system modeller is intended to replace the UNIX[®] *make* tool and to eliminate some of its problems: e.g., the inability to exactly reproduce the construction of a software system. The system modeller's MCode interpreter is specialized to executing these models and to interacting with the Clarity program database. It does not need, for example, to support SPARC ABI interoperation.

4 Summary

We have described an intermediate representation MCode that is compact, easy to generate, and supports the on-the-fly generation of good-quality machine code. Linkable MCode is an encoding of MCode in platform-standard object files that enables full interoperation with C and existing libraries, as well as the full use of all capabilities of standard linkers and other programming tools.

5 References

[Buhr et al 92] P. A. Buhr, Glen Ditchfield, R. A. Stroosbosscher, B. M. Younger, and C. R. Zarnke, "µC++: Concurrency in the Object-Oriented Language C++." *Software-Practice and Experience* 22:2, February 1992.

[Davidson 87] Jack Davidson, "Cint: A RISC Interpreter for the C Programming Language." Proc. of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques, *SIGPLAN Notices* 22:7, July 1987.

[Davidson 88] Jack Davidson, "A Portable Global Optimizer and Linker." Proc. of the SIGPLAN '88 Conference on Programming Language Design and Implementation, *SIGPLAN Notices* 23:7, July 1988.

[Deutsch&Schiffman 84] L. Peter Deutsch and Alan Schiffman, "Efficient Implementation of the Smalltalk-80 System." Proc. of the 11th Symposium on the Principles of Programming Languages. 1984.

[Franz 94] Michael Franz, "Technological Steps Towards a Software Component Industry." J. Gutknecht (ed.), *Programming Languages and System Architecture*. Springer Verlag #782, March 1994. ISBN 0-387-57840-4.

[Gosling 95] James Gosling, "Java Intermediate Bytecodes." *SIGPLAN Notices* 30:3, March 1995.

[Hölzle 94] Urs Hölzle, *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming*. Ph.D. dissertation, Stanford University, August 1994. Also *Sun Microsystems Laboratories Technical Report SMLI TR-95-35* (May 1995).

[Nelson 91] Greg Nelson, ed., *Systems Programming with Modula-3*. Prentice Hall, New Jersey. 1991.

[OMG 91] Object Management Group, "The Common Object Request Broker: Architecture and Specification." OMG Document Number 91.12.1, Object Management Group, 1991.

[Sankar 94] Sriram Sankar and Roger Hayes, "ADL - An Interface Definition Language for Specifying and Testing Software." Proc. of the Workshop on Interface Definition Languages, Jeannette Wing editor, *SIGPLAN Notices* 29:8, August 1994. Also *Sun Microsystems Laboratories Technical Report SMLI TR-94-23* (April 1994).

[SPARC ABI] *AT&T System V Application Binary Interface: SPARC Processor Supplement*. Prentice Hall, New Jersey. 1990.

[Wulf 75] William Wulf, Richard K. Johnson, Charles B. Weinstock, Steven O. Hobbs, Charles M. Geschke, *The Design of an Optimizing Compiler*, Elsevier Publishing Company. 1975.

About the authors

Brian T. Lewis is currently a Senior Staff Engineer at Sun Microsystems Laboratories. His interests include distributed programming, programming language and programming environment implementation, and user interface software. He previously developed monitoring and debugging tools as part of the Spring distributed object-oriented operating system project. He also has worked at Olivetti and Acorn Research, where he developed application runtime support software and a user interface toolkit. At Xerox, he developed a publication management system, software configuration and version management tools, and helped to implement the Mesa programming language and environment. He received a Ph.D. in Computer Science from the University of Washington.

L. Peter Deutsch received a Ph.D. in Computer Science from U.C. Berkeley in 1973. Prior to the Ph.D., he was one of the key designers and implementors of the SDS 940 time-sharing system, the first commercial, general-purpose time-sharing system using paging hardware. In his subsequent 13 years at Xerox PARC, where he attained the position of Research Fellow, he was one of the key designers and implementors of the research prototype of the Interlisp-D system; a significant contributor to the design of the Cedar Mesa language and the Smalltalk-80 programming environment; and the principal creator of PS, the first high-performance implementation of the Smalltalk language and programming environment on microprocessor-based hardware. From 1986 to 1991, Dr. Deutsch was Chief Scientist at ParcPlace Systems, where he was a principal designer of a high-performance Smalltalk implementation that is also highly portable across processors, operating systems, and window systems. From 1991 to early 1993, Dr. Deutsch held the position of Sun Fellow at Sun Microsystems, where he helped define future corporate strategy and technology in a variety of areas. Dr. Deutsch currently is President of Artifex Software Inc., which commercially licenses a highly portable high-performance implementation of the PostScript language that is also available for non-commercial use from the Internet under the name Ghostscript. In 1993, Dr. Deutsch was a co-recipient of the ACM Software System Award and was also named a Distinguished Alumnus of the U.C. Berkeley Computer Science program. He has served on numerous program committees for both ACM-sponsored and international conferences. Dr. Deutsch is a member of ACM, IEEE, CPSR, and the League for Programming Freedom.

Theodore C. Goldstein is a Principal Investigator at Sun Microsystems Laboratories. His research includes programming tools and methodologies for distributed operating systems and object-oriented programming languages. Previously, Ted has been with Whitesmiths, Ltd., Visicorp, Xerox PARC, and ParcPlace Systems. He holds a Bachelor of Arts degree in Computer and Information Science from the University of California at Santa Cruz.

© Copyright 1995 Sun Microsystems, Inc. The SML Technical Report Series is published by Sun Microsystems Laboratories, a division of Sun Microsystems, Inc. Printed in U.S.A. This paper was originally published in *ACM SIGPLAN NOTICES* 30, 3 (March 1995).

Unlimited copying without fee is permitted provided that the copies are not made nor distributed for direct commercial advantage, and credit to the source is given. Otherwise, no part of this work covered by copyright hereon may be reproduced in any form or by any means graphic, electronic, or mechanical, including photocopying, recording, taping, or storage in an information retrieval system, without the prior written permission of the copyright owner.

TRADEMARKS

Sun, Sun Microsystems, and the Sun logo are trademarks or registered trademarks of Sun Microsystems, Inc. UNIX and OPEN LOOK are registered trademarks of UNIX System Laboratories, Inc. All SPARC trademarks, including the SCD Compliant Logo, are trademarks or registered trademarks of SPARC International, Inc. SPARCstation, SPARCserver, SPARCengine, SPARCworks, and SPARCcompiler are licensed exclusively to Sun Microsystems, Inc. NFS is a registered trademark of Sun Microsystems, Inc. Solaris is a trademark of Sun Microsystems, Inc. Intel is a registered trademark of Intel Corporation. Smalltalk-80 is a trademark of ParcPlace Systems. Microsoft is a registered trademark of Microsoft Corporation. Macintosh is a registered trademark of Apple Computer, Inc. All other product names mentioned herein are the trademarks of their respective owners.

For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <jeanie.treichel@eng.sun.com>.

For distribution issues, contact Amy Tashbook, Assistant Editor <amy.tashbook@eng.sun.com>.