# Using C as a Compiler Target Language for Native Code Generation in Persistent Systems

S.J. Bushell, A. Dearle, A.L. Brown & F.A. Vaughan
Department of Computer Science
University of Adelaide
South Australia, 5005
Australia.
email: jsam, al, fred, francis@cs.adelaide.edu.au

## Abstract

Persistent programming languages exhibit several requirements that affect the generation of native code, namely: garbage collection; arbitrary persistence of code, data and processes; dynamic binding; and the introduction of new code into a running system. The problems of garbage collection are not unique to persistent systems and are well understood: both code and data may move during a computation if a compacting collector is employed. However, the problems of garbage collection are exacerbated in persistent systems which must support garbage collection of both RAM resident and disk resident data. Some persistent systems support a single integrated environment in which the compiled code and data is manipulated in a uniform manner, necessitating that compiled code be stored in the object store. Furthermore, some systems assume that the entire state of a running program is resident in a persistent store; in these systems it may be necessary to preserve the state of a program at an arbitrary point in its execution and resume it later. Persistent systems must support some dynamic binding in order to accommodate change. Thus code must be capable of binding to arbitrary data at a variety of times. This introduces the additional complexity that code must be able to call code contained in the persistent store produced by another compilation. In this paper native code generation techniques using C as a target language for persistent languages are presented. The techniques described solve all of the problems described above. They may therefore be applied to any language with these or a subset of these features.

## 1 Introduction

When orthogonal persistence is introduced to a programming language, several requirements emerge which affect code generation:

a. data in the system may persist,
b. code in the system may persist, and
c. the dynamic state of the system may persist.

Since all data may potentially persist, it must be held in a suitable form. Typically, a persistent object store will support one or more object formats onto which all data must be mapped. For example, objects must be self-describing to support automatic garbage collection and persistent object management. In particular, it must be possible to discover the location of all inter-object pointers contained in an arbitrary object. As a consequence, the code generation techniques employed must ensure that the objects constructed by the code conform to the appropriate object formats.

In languages that support first class functions and procedures, a further consequence of persistence is that these values may also persist. This implies that executable code must be mapped onto persistent objects. This requirement would defeat most traditional code generation techniques since the traditional link phase links together all the procedural values contained in a single compilation unit using relative addresses. If all code resides in relocatable persistent objects then the compiler/linker cannot determine the relative positions of code segments at run-time. Furthermore, facilities such as garbage collection and persistent object management may result in code segments moving during execution.

Persistent systems support potentially long-lived applications whose functionality may evolve over time. To accommodate this, many persistent systems provide facilities to dynamically generate new source code which is compiled and linked into the running system. This facility may be provided by making the compiler a persistent procedure [6, 7] .

In order to provide resilience to failure, many persistent systems periodically take snapshots. A system snapshot contains at least the passive data within the system but may also include the dynamic state of all executing programs. If a failure should occur, the data is restored from the last snapshot and if the dynamic state was saved, the system resumes execution. To support this, it is necessary to automatically preserve the state of a program at some arbitrary point in its execution and resume it later. This can give rise to problems in determining what constitutes the dynamic state of a program. For example, a traditional code generation technique includes a run-time stack containing return addresses, saved register values and expression temporaries. The task of saving state must establish what information on the stack should be saved and how it should be saved in order to support rebuilding the stack when the system is restarted.

In summary, code generation for a persistent programming language must address the following issues:

- mapping generated code onto relocatable persistent objects,
- linking generated code to the necessary run-time support,
- linking generated code to other generated code,
- preserving pointer values, including code linkage, over garbage collections,

- run-time compilation and execution of dynamically generated source, and
- preserving the dynamic state over checkpoint operations.

In this paper, the techniques employed to generate native code for the persistent programming language Napier88 [11] are presented. Napier88 is a persistent programming language which supports first class procedures, parametric polymorphism and abstract data types. The Napier88 system provides an orthogonally persistent integrated mono-lingual programming environment. The techniques described may be applied to any language with the features described above or a subset of these features.

## 2 Choosing A Compiler Target Language

Perhaps the most obvious method of code generation is to generate native code directly. This has the advantage that the writer of the code generator has complete control over:

- the mapping of code onto objects,
- linkage to the run-time support, and
- the location of pointers in data structures and registers.

Generating native code directly is also extremely costly since the compiler produced is architecture-dependent. An alternative to generating native code directly is to utilise existing code generation tools. Some advantages of this approach include:

- reuse of existing code generation technology,
- sophisticated optimisers are available, and
- the compilers can abstract over architecture-specific features.

The ability to reuse existing code generation technology is a significant advantage. For example, even low level tools such as assemblers include optimisers which relieve the compiler of the complexities of generating and backpatching instruction sequences. Higher level tools, such as compilers, incorporate more sophisticated optimisers which have been the subject of considerable research and development effort. Thus, this approach is a potentially cost-effective method of generating high quality code.

The range of tools investigated included assembly language, RTL and C [9]. Register Transfer Language, RTL, is an intermediate form used by the GNU C compiler [14]. RTL provides a rich set of abstract operators to describe a computation in terms of data flow between an arbitrary number of virtual registers. The GNU C compiler parses C source to produce a parse tree decorated with RTL. A range of optimisation techniques are applied to the parse tree which include the allocation of virtual registers to physical registers.

It was originally thought that a Napier88 compiler could generate an RTL representation of a program and have the GNU C code generator produce architecture-specific native code. However, RTL proved to be a poor choice since it does not completely define the program semantics without a parse tree, and it depends on machine specific descriptions. The developers of the GNU C compiler suggested the generation of C code followed by invoking the full GNU C compiler [16]. C is an excellent target language since it is:

- low level,
- easy to generate,
- can be written in an architecture-independent manner,
- highly available, and

- has good optimisers.

The C system chosen was GNU C, since it provides two very useful extensions over ANSI C. Firstly, it allows arithmetic on goto labels. This feature may be used to support saving and restoring state over checkpoints and garbage collections. Secondly, it is possible to explicitly map global variables onto fixed registers. This feature may be used to efficiently link generated code with the run-time support. A further advantage is that GNU C is freely available for most architectures, thus the use of GNU specific C extensions need not limit portability.

## 3 Seven Tricks for Compiling Persistent Languages

In this section, seven tricks are described which may be employed to efficiently solve the problems described in Section 1. They are:

1. the introduction of native code into a running system,
2. the ability to call other native code,
3. linking to persistent data and environment support code,
4. linking to the static environment,
5. reducing memory accesses,
6. the ability to run programs that cope with garbage collection and snapshot, and
7. reducing memory allocation overhead by allocating frames lazily.

### 3.1 The Introduction of Native Code Into a Running System

In order to support both integrated programming environments and run-time reflection, the Napier88 system contains a compiler that is callable at run-time. Various compiler interfaces exist and are described elsewhere [5]. All the interfaces take a description of the source text and environment information as parameters and produce an executable procedure injected into an infinite union type.

This functionality requires that the code generation technology be capable of supporting the dynamic generation of native code and its introduction into the persistent system. This may be achieved in four steps:

1. the compiler generates a C program,
2. the resulting C program is compiled in the normal manner to produce an object or executable file,
3. the executable native code is extracted from the object or executable file, and
4. the compiler creates one persistent object per compiled procedure and copies the instruction sequence for each procedure into the object.

Two techniques may be used to extract native code from the executable file produced by the C compiler: writing a utility based on the C linker or by generating self extracting code.

Object files generated by the C compiler contain linkage information that is used by the C linker. This information could be extracted by other programs capable of reading the object code format. However, the format of object files is operating system and/or architecture-dependent and therefore a separate utility needs to be written for each host environment. A viable alternative is to generate *self extracting* code, i.e. a program which, when run, will output all or some of its executable code. In our system, native code for the compiled program is placed into temporary files in a known format, independent of the host architecture.

The use of self-extracting code makes the compiler independent of the architecture-dependent structure of executable programs. Using this approach, the C compiler is directed to produce an executable program which is immediately executed. This program copies the executable code for each function into a temporary file. Function pointers are used to find the start of each instruction sequence. It is assumed that all memory between function pointers is code. This may result in extra data being copied, but guarantees that all the instructions of each function are copied.

This technique relies on specific assumptions about the C compiler. In particular, each compiled C function is assumed to contain no non-local references and all the local references are relative. That is, the compiled code is individual *pure* code sequences.

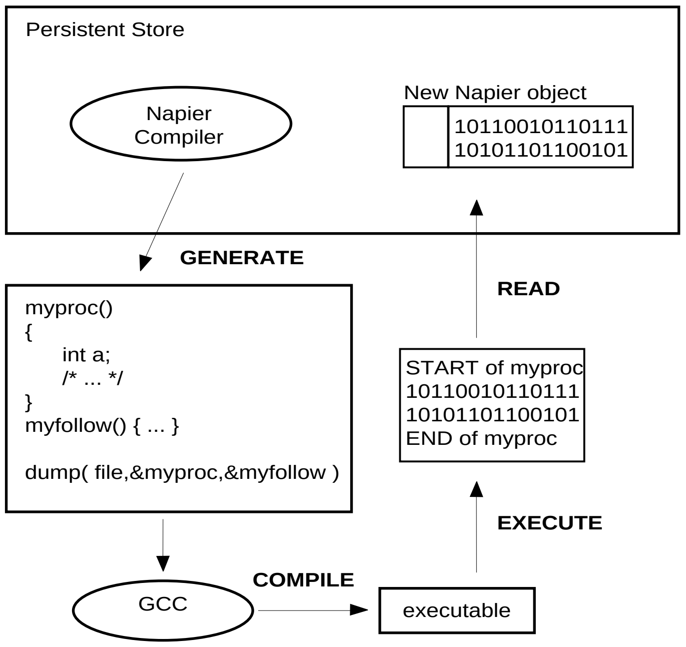

**Figure 1:** Introducing code into the persistent store

## 3.2 The Ability to Call Other Native Code

Since Napier88 supports first class procedures, a piece of native code must be able to call arbitrary compiled Napier88 procedures. These procedures may either be in the static environment of the caller, extracted from a data structure in the store, or passed as a parameter. When C is used as a target language, procedure call conventions may be based on jumps (gotos) or C function calls.

A major reason for using C as an intermediate form is to obtain access to the considerable optimisation technology already in existence. This optimisation technology is given more scope when Napier88 procedures are encoded as C functions with all invocation performed using C function calls. This presents the C compiler with independent compilation units over which its optimisers may operate. However, due to the presence of first class procedures, many global optimisations such as in-line expansions are not possible. For example, the C compiler is unable to trace execution between functions. Indeed, some functions may not exist at the time of compilation.

Utilising C functions has the advantage that calls to and from the run-time support can pass parameters, get results and save return addresses automatically. On some processors such as the SPARC [17], this is optimised by the use of windowed register sets. Native code-generated procedures can also use this mechanism to call each other.

C function calls have one major disadvantage: the C stack contains compiler and architecture-dependent data such as return addresses and saved register values. These addresses include object pointers that may be modified by the garbage collector and addresses which must be rebound over a snapshot and system restart. In both cases, some mechanism is required that allows both pointers and dynamic state information to be accessed and relinked to the appropriate addresses following a garbage collection or system restart. Ideally this mechanism should be architecture independent. This problem is addressed in Section 3.6.

An alternative is made possible by a GNU C extension called *computed gotos*. GNU CC lets you obtain the address of a goto label using the prefix operator "&&". Such an address may be stored in a C variable and later used as the target for a jump instruction with a statement of the form "goto *(*expression*)".

The Napier88 run-time system can call arbitrary generated code by computing an address within the code object and jumping to it. When generated code requires some support code to be executed, it jumps to an address within the run-time system. (How this address is calculated is discussed in Section 3.3.) Before jumping, it saves the address at which its execution should continue in a global variable.

Implementing source level procedure calls using jumps requires the provision of a mechanism to enable parameter passing. This may make procedure calling inefficient since any hardware support for procedure calling cannot be employed. This technique does have two advantages: it is extremely easy to implement and the location of all data is entirely under the control of the code generator.

A final point is that the decision to use jumps or C procedure calls affects the form of the generated code. If jumps are to be used, the code is structured as a collection of blocks, with each block corresponding to a source level procedure. If C function calls are employed, the generated code consists of a collection of C function definitions.

## 3.3 Linking to Persistent Data and Environment Support Code

By definition, programs written in persistent programming languages access persistent data held in some persistent object store. Programs may be statically bound to persistent data, as is the case in languages such as Galileo [1] or DBPL [10]. However, the exclusive use of static binding precludes system evolution. For this reason, many persistent languages, including Napier88, permit dynamic binding to persistent data. This requirement necessitates that native code be capable of dynamically binding to data in the persistent object store. In addition to user level code, compiled code must be capable of invoking the run-time support system. The support system contains functions such as the persistent object management code and garbage collector that would be extremely space inefficient to include in every piece of compiled code. The linking mechanism employed to link to persistent data naturally influences the mechanism used to invoke the run-time support.

Dynamic linking may be achieved in one of three ways:

1. by performing a linkage edit on compiled code,
2. through register indirection, and
3. through parameter passing.

The requirements of dynamic linking may be met by performing a linkage edit on generated code whenever the addresses it contains may have changed. This approach requires that code objects be modified dynamically and that all code objects include symbolic information which are architecture/operating system dependent.

```
/* declare a structure containing */
/* the address of an object creation function */
struct global_data {
     int (*create_object)(int) ;
} the_globals ;
/* declare a  fixed register */
/* %g7 is a sparc global register */
register struct global_data *fixed_register asm( "%g7" )
;

void init_table(void)
{
     /* the store's object creation function */
     extern int SH_create_object( int ) ;

     fixed_register = &the_globals ;
     fixed_register->create_object = &SH_create_object ;
}
```

**Figure 2:** Register indirection.

An alternative strategy is to note that many systems address global data by indexing through a fixed global register. BCPL employed this technique by providing a data structure called the *global vector*. Dynamic linking by register indirection can be easily encoded in GNU C since specific registers can be treated as C variables. The implementation of this technique requires that, upon initialisation, the support code construct a data structure containing the addresses of persistent data and support code. A pointer to this data structure is placed in some register. Whenever generated code wishes to address global data or the run-time support, it simply dereferences the allocated register variable. Figure 2 illustrates the initialisation of a fixed register and a global table with the address of an object creation function. Figure 3 illustrates how a C function could use dynamic linking via the fixed register to create an object of size 7.

```
struct global_data {
     int ( *create_object )(int) ;
} ;
register struct global_data *fixed_register asm( "%g7" )
;

void Cfunction(void)
{
     int object_id ;

     /* create an object of size 7 */
     object_id = fixed_register->create_object( 7 ) ;
}
```

**Figure 3:** Dynamic linking.

The advantage of this technique is that it allows the persistent store and run-time support to be freely re-implemented as long as they retain their specified interfaces. The disadvantages are that it permanently reserves a register for this purpose and cannot be implemented on some architectures.

An alternative to the fixed register approach is available if generated code is structured as C functions as describe in Section 3.2. A pointer to the global data structure may be passed as parameter. The invoked procedure passes this pointer to any procedures it calls and so on. The advantage of this technique is that it is architecture independent. The disadvantage is that a pointer must be passed as a parameter on every procedure call; although this overhead is not onerous on architectures such as SPARC that support register windows.

## 3.4 Linking to the Static Environment

Napier88 is a block structured Algol-like language with nested scope. Although C is block structured, it does not support nested functions and therefore some mechanism must be provided in the generated code to support scope. The interpreted version of Napier88 executes on a machine known as the Persistent Abstract Machine (PAM) [2]. In this implementation, procedure values or *closures* are implemented as a pair of pointers: one to an object containing code, the *code vector*, the other to the activation record of the defining procedure, the *static link*. Since procedures are first class values, they may escape the scope in which they were declared, therefore an activation record may be retained beyond the execution of the procedure which created it. Consequently, some stack activation records may not be deleted on procedure return and may only be reclaimed by garbage collection. Consequently activation records must be allocated on a heap rather than a conventional stack.

This run-time architecture may be reused in a system that generates native code by implementing each closure in the same manner as PAM and presenting each procedure call with a pointer to a heap object containing the activation of the lexicographically enclosing procedure. This pointer is placed in the activation record of the invoked procedure so that it may be dereferenced by the native code in order to access variables that are in scope.

A disadvantage of this technique is that, if C functions are generated, two stacks need to be maintained: a heap based Napier88 stack and the C stack. However, it has the advantage that object pointers are automatically rebound if the objects they address are moved by garbage collection since the pointers to them all reside in the heap.

## 3.5 Reducing Memory Accesses

As described in Section 3.4, in order to support block structure, all Napier88 variables may be placed in an activation record contained in a heap object. In the interpreted system, in order to perform a computation such as “a := a + b”, the PAM pushes the values of *a* and *b* onto the stack, incrementing a stack pointer each time. Then the plus instruction pops both values, adds them and pushes the result. Finally, the result is removed from the stack, the stack pointer decremented and the result written to its destination. Such computation is expensive because it dynamically maintains stack pointers and operates on memory rather than in registers. This expense can be reduced in three complementary ways: the elimination of stack pointers, the transformation of source expressions into C expressions and the use of local C variables.

In an interpretative system, many memory accesses are due to stack top relative addressing. In practice, there are very few cases where the stack top cannot be statically determined by the compiler's internal stack simulation.

In a system that generates C code, many simple sequences of stack-based PAM instructions may be directly replaced by C expressions. In order to implement “a := a + b”, a C sequence may be generated such as:

```
local_frame[ a_index ] =
     local_frame[ a_index ] + local_frame[ b_index ]
```

Such optimisations must be applied with care if the semantics of the high level programming language are to be preserved. In general, C does not guarantee the order of evaluation whereas Napier88 specifies left to right evaluation order. For this reason, expressions which cannot be guaranteed to be free from side-effects (including function calls) are not permitted in the generated C expressions. Nevertheless, this class of optimisation has a dramatic effect on the speed of generated code, largely due to optimisation possibilities.

In Section 3.4 we describe why a heap based Napier88 stack is required in addition to the C stack if Napier88 procedures are mapped onto C functions. Whilst many Napier88 expressions may be translated into C expressions, they still contain a performance bottleneck: all the data is referenced relative to the current Napier88 activation frame base.

A solution to this problem is to declare C variables, local to each generated function, which corresponded to locations in the PAM activation record. However, it is not always possible to represent source-level variables as C variables. For example, local procedures must be able to access variables in outer scope levels. Thus these variables must be stored in PAM objects when other procedures need to see them. On the other hand, leaf procedures (those that do not contain any other procedures) can keep their local variables in C variables with impunity, provided that they are still able to save their data into PAM frames when it is necessary to checkpoint the store or perform garbage collection.

This problem was solved by dividing generated code for procedures into two groups: the easy cases and the harder cases. The easy procedures, like leaf procedures, use C variables and are prepared to copy their values into PAM stack frames when necessary. The harder cases always keep their data in PAM stack frames. Since simple leaf procedures are a common case (akin to class methods in object-oriented languages) this yields a significant performance improvement.

## 3.6 Surviving Garbage Collection and System Snapshots

As described in Section 3.2, some mechanism is required that allows both pointers and dynamic state information to be accessed and relinked to the appropriate addresses following a garbage collection or system restart. The mechanism described in this section explicitly encodes source-level procedures as restartable C functions which are parameterised with a *restart point* and return a scalar *status value*. The restart point is used to indicate where in the procedure the code should start executing. The first call to a Napier88 procedure is performed by a C function call with a 0 restart point. The status value indicates if the procedure executed to completion or encountered some hindrance such as a request to make a system snapshot to invoke a garbage collection.

The PAM stack of persistent activation records, described in Section 3.4, is utilised by the restartable C functions. When a Napier88 procedure is called, a persistent object is created to represent its activation record. This object provides a repository in which data may be saved over garbage collections and checkpoints.

### 3.6.1 Garbage Collection and Checkpointing

In the Napier88 system, all garbage collection is performed synchronously with the computation: that is the computation stops when the garbage collector is running. Napier88 procedures recognise the need for garbage collection when heap space is exhausted.

```
struct global_data{
     int (*create_object)(int);
     int *local_frame;
};
register struct global_data *fixed_register asm( "%g7" )
;

int Nproc(int restart_point)
{
     int x ;         /* the variable x */
     int *S ;        /* object id for structure S */

                     /* restart using label arithmetic */
     goto *(&&start + restart_point);

start:
     x = 7 ;
create:
     S = fixed_register->create_object( SIZE_OF_S ) ;
     if ( S == NULL )       /* did the create fail */
     {               /* YES, garbage collect */

                     /* save x in current stack frame */
        fixed_register->local_frame[ x_offset ] = x ;
                     /* save restart point in stack frame
*/
        fixed_register->local_frame[ ResumeAddress ]
           = &&restart-&&start ;
                     /* return garbage collect request*/
        return unwind_and_continue ;

restart:                    /* restore x */
        x = fixed_register->local_frame[ x_offset ] ;
        goto create ;       /* repeat attempt to create S
*/
     }
     S[ 2 ] = x ;    /* initialise S */
     .....
                     /* update local_frame */
                     /* to point at caller */
     fixed_register->local_frame
        = fixed_register->local_frame[ DLink ] ;
     return OK ;     /* normal completion*/
}
```

**Figure 4:** The C function implementing the Napier88 procedure shown in Figure 5.

When they start executing, all Napier88 procedures register the address of the heap object containing the activation record in a global data structure. Before a garbage collection or checkpoint is executed, the generated code must ensure that their entire dynamic call chain is stored in PAM heap objects over which the garbage collector can operate. This is achieved by each procedure saving its entire state in the corresponding PAM stack frame. The saved state includes a resume address which may be passed to the function when it is restarted to indicate where it should continue computation. After saving its state, each procedure returns a status value to its caller indicating that it too should save its state and return the same status value to its caller.

Eventually, the flow of control returns to the run-time support which services the request by reading the global data structure and applying the appropriate function. Since each executing C function has saved its state in a heap object and returned, there is no data on the C stack. The mechanism is therefore architecture independent. Figure 4 shows a restartable C function for the Napier88 procedure shown in Figure 5.

```
proc()
begin
   let x := 7                     ! declare an integer x
   let S := struct( a := x )      ! declare a structure S
      !intialised using x
   .....
end
```

**Figure 5:** Napier88 procedure.

In the code generation system which we constructed, the status value is an enumeration containing three labels:

**OK**: which indicates that the procedure ran to completion,

**unwind_and_reapply**: which indicates that something abnormal occurred in the procedure so the caller should save state and unwind and that the procedure should be *reapplied* when computation re-commences, and

**unwind_and_continue**: which indicates that something abnormal occurred in the procedure so the caller should save state and unwind, but that the procedure has completed therefore the caller's next instruction should be executed when computation of the caller re-commences.

### 3.6.2 Restarting a Napier88 Program

Restarting a saved Napier88 program execution is performed in 3 steps. The first step is to find the PAM stack frame for the currently executing procedure; the address of this frame is held in the global data structure. Secondly, a pointer to the code object for the currently executing procedure is read from the stack frame. Finally, the C function in the code object is called and passed the restart point saved in the stack frame.

When a restarted procedure returns, it returns to the run-time support rather than its original caller. It will also have copied its Napier88 result, if any, into the caller's stack frame. This frame is found by following the dynamic link information stored in the stack frame. Since the caller's state has been stored in its stack frame together with an appropriate resume address, it can also be restarted.

## 3.7 Lazy Frame Allocation

The final trick employed was to avoid allocating stack frames for procedure unless absolutely necessary. As described in Section 3.5, many procedures are leaf procedures and as such only require their state to be saved in a heap object if a garbage collection or system snapshot is required. A considerable performance increase in performance may be obtained by only creating heap objects for stack frames when required to do so.  This trick needs to be applied with care.  Garbage collections are usually only invoked when the system has run out of space.  The creation of objects at such times can be counter-productive!

In our system, when a Napier88 procedure call is made, space is reserved in the heap for the frame that may be required.  When the procedure returns the reserved space is released.  This ensures that there is always space available for dumping procedure state even when a garbage collection is required.

## 4 History

The techniques described above were not all conceived or implemented at once, but at different stages in continuous development.  The compiler and run-time system to support native code were built from a working compiler that generated PAM code and PAM code interpreter written in C.  The run-time system still contains the interpreter code; interpretative and native code may coexist and call one another in our implementation.

### 4.1 Threaded Code

The first step was to utilise the PAM abstract machine architecture by generating C code which replaced PAM instructions with explicit calls to the C functions that interpret them.  This kept the changes to the compiler and interpreter manageable.  Where the compiler previously generated PAM instruction *n*, it would now generate an instance of a C macro called “Pam_*n*”; this macro expanded into a call to the C function that implemented instruction *n*.

The structure of the code generation mechanism and the communication between generated code and the run-time system was established and tested prior to the development of efficient code generation patterns.

### 4.2 Simple Macros

The second step was to replace calls to the simpler interpreted instructions with equivalent in-lined C code.  This was accomplished by rewriting the C macros for these instructions.

The net effect of this step was to produce a working Napier88 system where most of the interpretative decoding overhead had been removed.  However, the generated code still followed the PAM stack model, explicitly manipulating a persistent activation record through stack pointers.  Since the C compiler cannot determine the global effects of assignments to the frame, it will ensure they are all performed.  This effectively defeats an optimiser since it cannot elide superfluous frame assignments.

### 4.3 C Expressions

The first attempt to diverge from the PAM stack model was to translate simple Napier88 expressions like “(a + b * c) > 5” into isomorphic C expressions.  This optimisation was only peformed where the semantics were guaranteed to be faithful to the defined Napier88 semantics; no side-effects were permitted except for those in assignment statements, and neither were Napier88 procedure calls.

Where appropriate this allowed the C compiler to perform constant folding and avoided unnecessary memory references. This resulted in excellent optimisation of this class of Napier88 expressions.

### 4.4 Removing Run-time Stack Pointers

The next stage was to take advantage of the fact that the locations of the stack tops are statically known. Stack-based operations were rewritten so that they read and wrote directly to known offsets into stack frames rather than incrementing and decrementing stack pointers and performing stack loads. Run-time stack pointers are still needed to handle polymorphism [12], but only during short and well-defined windows of uncertainty. Extra parameters were supplied to the C macros to indicate the relative stack locations.

Although this technique simplified the C code produced, it still encoded calculations as manipulations of stack locations in main memory and so inhibited effective use of an optimiser.

### 4.5 Local Variables

The next major stage was to declare C variables, local to the generated function, which corresponded to locations on the PAM stack. The word at location *n*, previously accessed as Frame[*n*], was now treated as the variable F_*n*. As described earlier, this optimisation is only applied to leaf procedures. Code was also generated to save and restore the variables, where necessary.

### 4.6 Lazy Frame Allocation

Having decided to keep local data in C variables, we realised that many leaf procedures do not actually need to have PAM frame objects allocated at all, and that we could reduce the time overhead of function call by omitting this allocation. However, as described above, it is sometimes necessary to have a frame later in the execution of the procedure – for example, if a garbage collection is imminent. We therefore implemented lazy frame allocation for non-polymorphic leaf procedures. This required that the native code calling mechanism be modified so that the callee allocated the activation record. Arguments to Napier88 procedure calls were passed as C arguments.

## 5 Performance

To indicate the relative merits of some of the above code generation techniques, timing results for the following simple benchmarks are provided:

- nfib – a recursive, function call intensive program, listed in appendix 1.
- dhrystone – an Ada benchmark [19].
- quicksort – sorting a 10,000 element array of integers, listed in appendix 1.

Measurements were made on a Sun Sparcstation 2 with 64 megabytes of main memory. The native code generation experiments were conducted using a Napier88 system based on a single user, page based (CASPER [18]) object store held on a local disk. Computation-intensive benchmarks are used for two reasons. Firstly, the performance characteristics of the persistent object store cannot significantly affect the results. Secondly, a C implementation of the benchmarks can be used to give an upper bound on our performance expectations.

The graph in Figure 6 shows the speedup relative to the CASPER PAM achieved by the incremental application of the code generation techniques. For comparison the graph includes the performance of the benchmarks written in GNU C when optimised and un-optimised. Note that procedure calls are relatively infrequent in the quicksort algorithm; hence the flat spot on the quicksort curve.

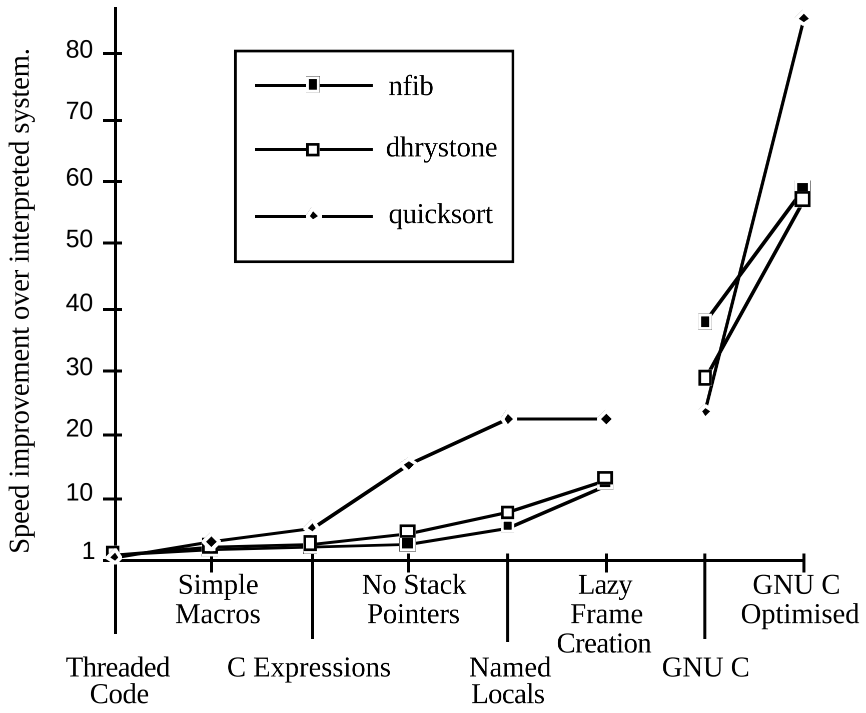

**Figure 6:** Performance increase relative to interpreted system.

One drawback of the generated code is its size. It is only fair to expect that native code be significantly larger than PAM code, since PAM instructions can describe complex operations in a single byte, while the smallest machine instruction available on the SPARC architecture occupies four bytes.

Code files containing native code can exceed the size of their PAM code counterparts by a factor of ten, or more. As the graph in Figure 7 illustrates, code file sizes have varied significantly during our experiments with code generation techniques.

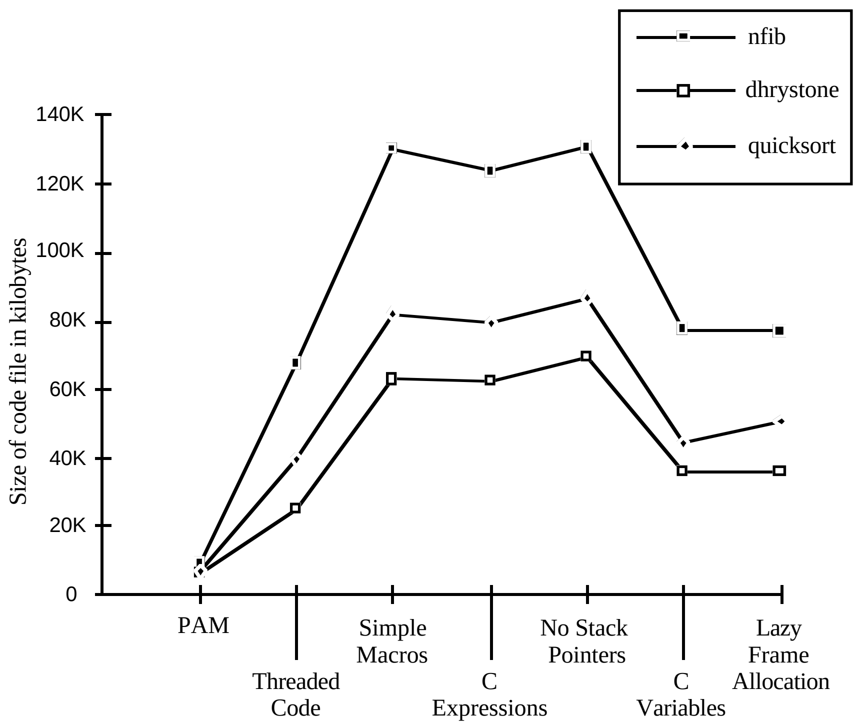

**Figure 7:** Sizes of the code files for the benchmark programs (kb).

The other drawback is that compilation to native code is relatively slow. The native code compiler takes between three and fifteen times as long as the PAM-code generating compiler on which it is based. For instance, compiling the Dhrystone benchmark into PAM code takes about 10 seconds; compiling it with the native code compiler takes about 35 seconds, or 105 seconds with GNU C optimisation level 2.

The extra time is spent almost entirely in GNU CC. This is not surprising, since large amounts of C code are generated. Compiling Dhrystone produces 100K of macros which are expanded into 480K of C source by the pre-processor.

We have endeavoured to reduce code size and compilation time by making the code generation patterns as simple as possible, factoring out common segments of code into new run-time support calls.

## 6 Conclusions

This paper presenta techniques for generating native code for persistent programming languages. C is used as a compiler target language resulting in a portable and efficient code generation technique whose performance approaches that of equivalent C programs. The full functionality of a strongly typed persistent object store is freely available without the undesirable aspects of programming in C. The code generation techniques presented permit:

- the co-existence of interpreted and native code,
- code to be mapped onto relocatable persistent objects,
- linked to the necessary run-time support and other generated code,
- the use of compacting garbage collectors,
- the run-time compilation and execution of dynamically generated source, and

- the preservation of dynamic state over system snapshots.

We have recently re-implemented the code generator's C macros to abstract over address sizes. This enhanced portability permits the code generation technology to be employed on the latest 64 bit RISC architectures such as the DEC Alpha [15]. We expect to have a robust native code generator running on an Alpha platform by the time this paper is published.

## 7 Future Work

We are currently investigating the use of boxed values [13] as implemented in the interpreted Napier88/Octopus system [8]. We believe that the use of boxed values will permit lazy code, as described in Section 3.7, to be generated in many more cases than is currently possible. Whether or not this proves to be an optimisation remains to be seen.

We are currently assembling Napier88 systems that will permit us to compare swizzled and directly mapped store technologies. The system under construction will support independent configuration of the following options:

- whether the store is directly mapped with page granularity, or swizzed with object granularity;
- whether the code in the store is native code or interpreted PAM code;
- whether the code in the store uses the Octopus model [8] or the original PAM frame model [4]; and
- whether the system will run on the Sun SPARC architecture or the DEC Alpha AXP architecture.

We plan to measure the performance of the OO7 benchmark [3] under all combinations of the above options, comparing object-swizzled and page-mapped stores.

## Acknowledgements

We would like thank the POS VI referees for their anonymous comments on this paper. We would particularly like to thank Ron Morrison for his un-anonymous and very helpful comments. Thanks must also go to Karen ‘the comma’ Wyrwas for her stylistic suggestions; any typos are her fault!

## References

1. Albano, A., Cardelli, L. and Orsini, R. “Galileo: a Strongly Typed, Interactive Conceptual Language”, *Association for Computing Machinery Transactions on Database Systems*, vol 10, 2, pp. 230-260, 1985.
2. Brown, A. L., Carrick, R., Connor, R. C. H., Dearle, A. and Morrison, R. “The Persistent Abstract Machine”, Universities of Glasgow and St Andrews, Technical Report PPRR-59-88, 1988.
3. Carey, M., DeWitt, D. and McNaughton, J. “The 007 Benchmark”, *SIGMOD*, vol 5, 3, 1993.
4. Connor, R., Brown, A., Carrick, R., Dearle, A. and Morrison, R. “The Persistent Abstract Machine”, *Proceedings of the Third International Workshop on Persistent Object Systems*, Newcastle, Australia, Springer-Verlag, pp. 353-366, 1989.
5. Cutts, Q. “Deviering the Benefits of Persistence to System Construction and Execution”, Ph.D. thesis, Computational Science, St Andrews, 1994.
6. Dearle, A. “Constructing Compilers in a Persistent Environment”, Universities of Glasgow and St Andrews, Technical Report PPRR-51-87, 1987.
7. Dearle, A. and Brown, A. L. “Safe Browsing in a Strongly Typed Persistent Environment”, *The Computer Journal*, vol 31, 6, pp. 540-545, 1988.
8. Farkas, A. and Dearle, A. “The Octopus Model and its Implementation”, *17th Australian Computer Science Conferenc, Australian Computer Science Communications*, vol 16, pp. 581-590, 1994.
9. Kernighan, B. W. and Ritchie, D. M. “The C programming language”, Prentice-Hall, 1978.
10. Matthes, F. and Schmidt, J. W. “The Type System of DBPL”, *Proceedings of the Second International Workshop on Database Programming Languages*, Portland, Oregan, Morgan Kaufmann, pp. 219-225, 1989.
11. Morrison, R., Brown, A. L., Connor, R. C. H. and Dearle, A. “The Napier88 Reference Manual”, University of St Andrews, Technical Report PPRR-77-89, 1989.
12. Morrison, R., Dearle, A., Connor, R. C. H. and Brown, A. L. “An Ad-Hoc Approach to the Implementation of Polymorphism”, *Transactions on Programming Languages and Systems*, vol 13, 3, pp. 342 - 371, 1991.
13. Peyton-Jones, S. “The implementation of functional languages”, Prentice-Hall, 1987.
14. R. Stallman, R. “Using and Porting GNU CC”, Free Software Foundation, Technical Report 1991.
15. Sites, R. L. “Alpha Architecture Reference Manual”, Digital Press, 1992.
16. Stallman, R. 1993.
17. Sun Microsystems Inc. “The SPARC Architecture Manual, Version 7”, 1987.
18. Vaughan, F., Schunke, T., Koch, B., Dearle, A., Marlin, C. and Barter, C. “Casper: A Cached Architecture Supporting Persistence”, *Computing Systems*, vol 5, 3, California, 1992.
19. Weicker, R. P. “Dhrystone: A Synthetic Systems Programming Benchmark.”, vol 27, 10, Communications of the ACM, pp. 1013-1030, 1984.

# Appendix 1 Benchmark Sources

## A1.1 Nfib Napier88 Source

```
rec let nfib = proc( n: int -> int )
    if n < 2 then 1 else 1 + nfib( n-1 ) + nfib( n-2 )
```

## A1.2 Quicksort Napier88 Source

```
let partition = proc ( A : * int; l,r : int -> int )
begin
        let k = ( l + r ) div 2
        let al = A( l ) ; let ar = A( r ) ; let ak = A( k )
        let t = case true of
        al < ar :
                if ak < al  then al else
                if ak < ar  then { A( k ) := al ; ak }
                            else { A( r ) := al ; ar }
        ak < ar : { A( r ) := al ; ar }
        ak < al : { A( k ) := al ; ak }
        default : al

        let v := l ; r := r + 1
        let notdone := true
        while notdone do
        begin
                repeat l := l + 1 while l < r and A( l ) < t
                if l = r then notdone := false else
                begin
                        repeat r := r - 1 while l < r and t < A(
                      r )
                        if l = r then notdone := false else
                        begin
                                  A( v ) := A( r ) ; A( r ) := A(
                      l )
                                  v := l
                        end
                end
        end
        l := l - 1
        A( v ) := A( l ) ; A( l ) := t
        l
end

rec let quicksort = proc( A : * int ; l,r : int )
while l < r do
begin
        let k = partition( A,l,r )
        if k - l > r - k then {quicksort( A,k + 1,r ) ; r := k
      - 1}
                          else {quicksort( A,l,k - 1 ) ; l := k +
      1}
end
```