This paper should be referenced as:

Cutts, Q.I., Connor, R.C.H., Kirby, G.N.C. & Morrison, R. “An Execution Driven Approach to Code Optimisation”. In Proc. 17th Australasian Computer Science Conference, Christchurch, New Zealand (1994) pp 83-92.

An Execution-Driven Approach to Code Optimisation

Quintin Cutts, Richard Connor, Graham Kirby and Ron Morrison

Department of Mathematical and Computational Sciences, University of St Andrews,
North Haugh, St Andrews, Fife KY16 9SS, Scotland.

Phone: +44 334 63241
Internet: {quintin, richard, graham, ron}@dcs.st-andrews.ac.uk

Abstract

Whether code is created by a programmer or generated by a compiler, its construction is traditionally based on a strategy developed before code execution begins. The strategy determines the conditions most likely to occur during execution and optimises code generation accordingly. Execution is non-optimal when the conditions expected by the code generation strategy differ from those actually experienced.

An optimisation technique is described here which allows an executing system to be incrementally improved as information about the execution conditions become available. Optimisations are performed by recompiling source code using new code generation strategies and then linking the new code segments into the executing system. The body of data collected or constructed during execution is unaffected, making the technique particularly suited to long-lived, data-intensive systems.

This paper describes the technique in general, implementation requirements and a particular instantiation of the technique in which the implementation of polymorphism is optimised. Further examples of the technique are also outlined.

1. Introduction

Code is traditionally constructed according to a statically-determined strategy based on expected execution conditions. Typically the code generator, whether a programmer or a compiler, is unable to determine which of the potentially huge number of possible conditions will occur. Since the construction of optimal code for all possible situations is prohibitively expensive or even impossible, the traditional goal of code generation is to construct code which executes acceptably in all situations and optimally in those situations statically deemed most likely to occur.

The aim of this paper is to describe an optimisation technique that allows systems to dynamically respond to differences between the expected and actual execution environment. Response takes the form of code segment recompilation using strategies based on the execution conditions experienced dynamically. The compilation strategies allow optimal code to be constructed for the dynamic conditions.

Examples of the optimisations under consideration here include:

- automatic self-generation of new code to handle execution conditions not statically allowed for,
- optimisation in place of the method code of objects in object-oriented systems, whilst maintaining existing data within the system, and
- dynamic restructuring of the queries in a database system to operate optimally over the particular data discovered during execution.

Dynamically tuned systems are not new. Such systems traditionally are optimised by manipulating the data over which the system operates. An example is clustering [BDH91] where separate data items that are dynamically discovered to be accessed as a group are clustered in storage so as to optimise access speed.

Tuning code in a system is unusual but is found in, for example, query optimisation where database access algorithms are updated as the body of data within the database evolves [CD91].

The optimisation technique described in this paper extends code tuning by supporting the addition of dynamically constructed code to a running system. Newly compiled code segments are added to the body of code making up the system. The most appropriate code version can then be chosen according to the prevailing conditions.

The technique is particularly applicable to systems where programs are constructed in isolation from the data over which they operate. This occurs in long-lived and data-intensive systems such as CAD/CAM systems, office automation and CASE tools.

The requirements for implementing the technique may be summarised as follows:

- persistence of data including code across system invocations,
- the availability of system source code and a compiler during execution, and
- incremental linking and loading of newly constructed code segments.

These requirements make the technique particularly suitable for object-oriented systems [GR83,Car89,BOP+89] and persistent programming environments [KCC+92,PS88,

Tha86,Car85,SCW85]. In this paper, it will be shown that a persistent programming system provides an appropriate base technology for the technique's implementation.

2. The General Optimisation Technique

2.1 Overview of the Technique

Programs are constructed so that they will run optimally under the conditions expected to occur during execution. This optimal position is usually based on a tradeoff between the speed of execution and the space required for the storage and manipulation of program code and data. The optimisations supported by this technique are required for two reasons: either accurate estimation of the execution conditions is hard to determine statically, in which case the optimisations will be made soon after execution begins, or else conditions are as expected during initial execution of the program but they change later requiring an adjustment of the statically chosen tradeoff between space and time.

The complete optimisation technique may be summarised in the following steps.

- An optimisation strategy for a system is developed. That is, the execution profiles and analysis mechanisms required to identify particular code optimisations are determined. An initial implementation of the system is constructed which is optimised for the expected execution conditions.
- The initial system is augmented with code to record execution profiles.
- Execution of the system is initiated. Execution profiles are recorded during system execution.
- Once a body of profiling data has been gathered, the analysis mechanism uses the profiles to determine whether particular code optimisations should be performed.
- According to the results of the analysis, code segments may be optimised by recompiling the source code under a new compilation strategy. Alternatively, the source code may be transformed followed by recompilation.
- The new equivalent code segments are linked into the running system.
- As execution continues, the new segments will be used when appropriate. Profiling may continue as an on-going process.

Implementation is possible in traditional systems that support files as the only persistent data type and isolate from one another the processes of code construction, compilation, linking and execution, e.g. Unix [RT78]. However, this paper demonstrates that a persistent programming system provides a more appropriate base technology for implementation of the technique. The remainder of this section describes the technique in greater detail and illustrates both its benefits and disadvantages as well as the system characteristics required for a useful implementation.

2.1.1 Determining an optimisation strategy

Code segments within a system are identified which cannot be implemented statically in such a way as to ensure optimal execution. The dynamic information that must be collected in order to effect an improved implementation is determined. Analysis and optimisation techniques are designed which operate over both statically and dynamically gathered information to produce an optimised implementation.

There are two distinct styles of optimisation considered here that both use the same technique, as follows:

- Optimisations constructed by the system programmer to be used to improve the algorithms encoded in the system source code.
- Optimisations constructed by the compiler writer to improve the code generated by the compiler for given language constructs.

2.1.2 Recording execution profiles

Profiling code is added to the system at positions where the dynamic information required for the optimisation can be discovered. The code records the information in associated profiling data structures. There are two situations in which profiling code is added, according to the two distinct styles of optimisation described above:

- The programmer may add profiling routines as part of the original source code.
- The compiler may automatically augment compiled code with profilers to gain information about the execution of particular language constructs.

The profiling data structures must be available to all stages of the technique, as follows.

- They are directly or indirectly included in the source code during the construction process.
- They must be retained across multiple executions of the code that invokes the profiles to ensure that a wide range of execution conditions are recorded.
- They must be available for analysis.
- They may be required during the optimisation process.

The efficiency of the profiling operation should be considered to avoid significant degradation of system performance, a requirement that restricts the class of profiles that can successfully be recorded.

2.1.3 Discovering potential optimisations

Possible optimisations are discovered by analysing the profiles according to the strategy developed at the start of the process. The analysis is performed using a cost function which trades the cost of an improvement in terms of construction and storage of the new code segment against the benefits gained during its subsequent execution.

The cost function requires a certain amount of information in order to make an accurate assessment on possible optimisations. This information may consist of both data collected during execution and data gathered statically by the programmer or compiler. The static information may be bound into the cost function; the dynamic data is passed as a parameter on the function's invocation. A good optimisation strategy maximises the amount of information gathered statically whilst minimising the requirement for dynamically gathered data.

Another important consideration affecting the cost function is the time at which the system is checked for possible optimisations. There are two possibilities here.

- During system execution. Analysis may be triggered by conditions experienced during execution. The trigger code contains or makes bindings to the appropriate cost function and profiles. The expense of cost function execution is critical as it takes place during system execution.
- During quiescent points in execution. Enhancer programs may browse over profiles, passing them to the associated cost functions for analysis. Using this technique the expense of the cost function is not critical.

2.1.4 Optimising code segments

As described earlier, optimisations are based on a recompilation of the source code. To do this, the source code for the executable code segment in question must be found. There are two techniques for recompilation, as follows.

- The same compilation strategy may be used against the source code which has undergone some transformation.
- A new compilation strategy may be used against unchanged source code.

The exact optimisation may be known statically; otherwise the transformation or the new strategy to be used is dependent on data recorded in the execution profiles.

The original executable code segment may be handled in two ways, as follows.

- If the newly constructed code is always more efficient than the original code then the latter may be discarded.
- If the newly constructed code is more efficient than the original only in certain conditions then both versions should be retained. The source code is augmented prior to compilation with switching code which chooses the appropriate code version according to the prevailing conditions. Repeated optimisations may produce many versions: the switching code must be extended as new versions become available. Such a situation is shown in Figure 1.

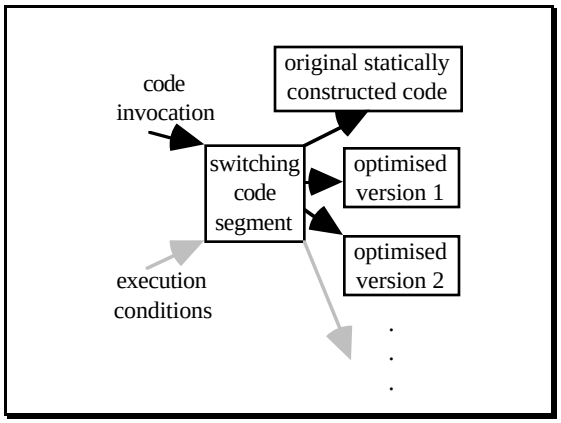

Figure 1 **Switching between a number of equivalent code segments**

2.1.5 Linking and executing new segments

Optimised code segments are linked back into the application. According to the time of optimisation, this may occur either during or in between executions of the application.

Depending on the strategy used for combining old and new code versions, the new code segment will be invoked when the execution conditions are appropriate. Mechanisms for incremental loading of individual code segments will be required as the body of application code is repeatedly extended with new code segments.

Some optimisations may require no further profiling of the associated code. This will occur when no further improvements can be made to the code. Otherwise the profiling code remains to provide continuing records of execution conditions; the optimisation cycle is ongoing, initated by off-line browsing mechanisms or by trigger code within the system.

2.2 Assessing the Technique

The advantages of the technique may be summarised as follows. An executing system can be incrementally improved as knowledge is gathered about its execution characteristics. Analysis

concerning the requirement for and the type of optimisation is based on data collected during execution. Optimal code may be generated for frequently occurring execution conditions.

Unlike traditional optimisation mechanisms which restrict or replace segments of the code, the technique extends the body of code making up a system with new segments which execute optimally in conditions not statically provided for.

Optimisations are made to the executable code of running systems, either during active execution or during quiescent phases. Complete system shutdowns and reconstruction of executable images to include the optimised components are not required.

The optimisation technology may be used both by the system programmer or the compiler writer to optimise the code they construct. Both have the same problems: they are unable to ascertain which of the potentially huge number of possible conditions will occur during execution; they do not have the resources to optimally provide for all of them.

Potential disadvantages of the technique are as follows. The operation of the system will be slowed down by the profiling code. The complexity of the profilers should be minimised by maximising the static information that can be included in the cost functions. In addition the profilers may be switched off to allow information to be gathered on a regular but not continuous basis.

Optimisation during active execution will also affect system performance. This must be traded against the immediate benefits that the optimisation may provide.

Switching between code versions may become expensive as the number of versions increases. This extra cost could become part of the cost function that determines whether new optimisations should be made. There may be occasions where a particular code version is not frequently used and can be removed from the switching loop until conditions change.

The choice of optimisation is critical to the success of the technique - the cost of profiling and choosing between code versions must be small relative to the savings obtained from the optimisation.

2.3 Requirements for Implementation

In order to implement this technique, profiles and new code forms must persist across invocations of individual code fragments [AM85], otherwise the knowledge gained during execution and the benefits of optimisations will be lost. Both profiles and code forms are potentially highly structured objects: the profiles must be structured to permit fast access to and recording of profiling information; the code forms will in general be structured due to the requirements for incremental linking and loading.
The source code of individual code segments must be available during execution. Source code is represented as a data type in the system which may be manipulated as required. Retaining source code is not trivial in the presence of first-class code objects where references to free identifiers must be represented [KCC+92].

A compiler is used during execution by the optimiser to construct new executable representations. For some optimisations, the optimiser can adjust the code generation strategies used by the compiler according to dynamically gathered data. The compiler must be able to construct new code segments which operate over the same data as did the original segments.

New code segments are incrementally linked into a running system. In general this requires structured executable forms [ABC83], although mechanisms do exist for limited incremental linking over flat executable forms [QL91].

Code segments are incrementally loaded in situations where there is a choice between different code versions. This becomes a requirement where the number of code versions from which to choose is large.

To summarise, implementation of the technique will be simplified by the availability of two key features: an integrated program construction, compilation, linking and execution environment as well as the persistence of data values including source and executable code. These features are typically found in persistent programming systems and in object-oriented systems. Such systems are therefore well-suited to implementing the technique, a particular example of which is now described.

3. Implementation in an orthogonally persistent system

A persistent environment is a type safe repository for structured data including source and executable code objects. All objects have a unique identity. The referential integrity of the environment ensures that links between objects are maintained [MBC+93]. The contents of the environment may persist for as long or short as required, irrespective of their type.

The software phases of code construction, compilation, linking and execution may all be supported within the persistent environment. The compiler is a first class procedure available within the environment. Incremental linking to executable code objects can be supported using typed persistent locations as described in [DCC92]. The underlying persistent storage mechanism supports incremental loading [Bro89].

The mechanisms used in a persistent system to implement each stage of the technique are presented in this section. The following example which optimises the implementation of polymorphism will be used to illustrate a particular style of optimisation within the wide range of possibilities offered by the technique. The description of the optimisation is restricted in places to fit the scope of the paper. Full details may be found in [Cut92].

3.1 Optimising polymorphism

3.1.1 Designing a strategy for optimising a polymorphism implementation

Polymorphism in a programming language is the ability to write programs that are independent of the form of the data values that they manipulate. Thus it provides an abstraction over the form of the data which is often categorised by type. The code generated for a polymorphic context must execute correctly irrespective of the format of the data over which it operates.

In the persistent polymorphic language Napier88 [MBC+89], a polymorphic bubblesort function may be written and called as follows:

```
let sort = proc[ t ](
        v : *t ;
        lessThan : proc( t,t -> bool ) )
begin
  let lower= lwb[ t ]( v )

  for outer = lower+1 to upb[ t ]( v ) do
    for inner = outer to lower+1 by -1 do
      if lessThan( v( inner -1 ),v( inner ) ) do
      begin
        ! Swap the two values over
        let temp = v( inner - 1 )
        v( inner - 1 ) := v( inner )
        v( inner ) := temp
      end
end

let numbers = vector @1 of
                        [ 2,6,4,7,3,1,8,0 ]
let intLessThan = proc( a,b : int -> bool )
                        a < b
sort[ int ]( numbers,intLessThan )
```

Figure 2 Napier88 polymorphic identity procedure.

The polymorphic function *sort* takes a vector of any type *t* and a comparison function over values of type *t* and sorts the elements of the vector in place. In the final line of Figure 2, *sort* is called with the quantifier *t* specialised to *int* and the specialised procedure applied to a vector of integers and a comparison procedure operating over integers. Polymorphic values are manipulated inside the procedure at the point of comparison and when values are swapped between locations.

Implementations of polymorphism [MDC+91] trade speed of execution against the space required to store code implementing polymorphic functions. The fastest execution speed is gained when different executable code is constructed for each different specialising type. This gives the efficiency of the equivalent monomorphic procedure. Ada generics are implemented using this technique [DOD83]. The drawback is that there may be a very large number of specialising types and hence a correspondingly large number of code versions. All specialisations are known statically in Ada and so the technique is feasible; in the presence of anonymous procedure values it may be impossible to determine statically the number and type of code versions required.

At the other end of the space/time tradeoff, a single executable version of a polymorphic procedure may be generated in a system which manipulates all data in one uniform format. This technique is used to implement the polymorphism of ML [MCP88]. Whilst efficient in terms of space, speed of execution is reduced by the enforcement of a uniform representation; in particular, arithmetic operations may be affected as they depend on efficient data representations.

The problems of polymorphism implementations are well suited to the optimisation technique described in this paper, since they involve tradeoffs between space and execution speed that may change over time as knowledge of a system is gathered. Optimisation of polymorphism within the Napier88 system will be examined here. The fastest implementation, where the monomorphic form is constructed, cannot be used in general as the particular monomorphic forms required are not known statically and it is too expensive to construct them all. Instead, an implementation is initially used which operates under all specialisations with slightly reduced space efficiency compared with the ML technique but with improved speed. The implementation is described in detail in [MDC+91]. In this context it is sufficient to note that the implementation represents an good tradeoff between space and time at compilation when the specialising types for a given procedure are unknown

The Napier88 implementation is augmented with profiling code to record the particular specialisations that occur during execution. When a representative body of profiling data has been gathered, analysis will indicate which specialisations occur frequently enough to merit construction of the appropriate optimal monomorphic form. Switching code is added to the procedure to choose and call the appropriate monomorphic versions or if none is available to call the less efficient polymorphic version.

Figure 3 shows the positions of the three implementations used in Ada, ML and Napier88 on a graph of space efficiency against speed efficiency. As Napier88 polymorphic code is optimised using the technique described here, its space efficiency is reduced whilst its speed efficiency for the affected specialisations attains that of the Ada implementation.

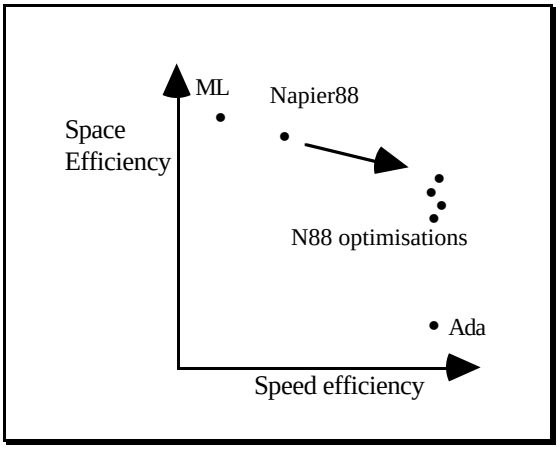

Figure 3 **Comparing different implementations.**

3.2 Implementing the polymorphic optimisation

Each phase of the optimisation technique will now be described with reference to implementation in a persistent environment. The implementation of the polymorphism optimisation will also be described at each stage.

3.2.1 Adding execution profilers

Source code is augmented either with code that directly updates profiling data structures or with calls to profiling procedures. The profiling structures are language data types accessible to any program. They are held in the persistent store and so persist across invocations of the programs that access and update them. Profiling code may be directly linked to profiling structures and so there is no requirement for additional code to make bindings during execution between the running program and the persistent profiling structures.

The same method is used both by the compiler and the programmer. The compiler additions are automated and included with every instance of a particular language construct under consideration for optimisation. By contrast, those manually added by the programmer are specific to the particular code under construction.

The polymorphism optimisation affects the code generation strategy used by the compiler. Code is therefore automatically added by the compiler to each polymorphic procedure to record the number and type of calls.

The data associated with each kind of specialisation is recorded in instances of the data type shown in Figure 4. An instance of type *profile* is created for each different specialisation of a particular procedure. A representation of the specialising types is held in the *types* field in an instance of the type *typeReps*, the details of which are not of interest here. Each time the same call is made the *calls* field is incremented.

```
type profile is structure(
        types   : typeReps ;
        calls   : int )
```

Figure 4 **Data type to hold profiling information.**

The code of Figure 3 is transformed into that of Figure 5.

```
let profiles = mkList[ profile ]()

let compiledSort = proc[ t ](
      v          : *t ;
      lessThan   : proc( t,t -> bool ) )
begin
  countCalls( profiles,mkTypeReps[ t ]() )

  ! remainder of the procedure
  ! as given in Figure 2
end
```

Figure 5 **Polymorphic procedure augmented with profiling code.**

profiles is initialised to be an empty list of element type *profile*. The procedure *countCalls* has been statically constructed and records the appropriate details about execution of the procedure. If *countCalls* fails to find a *profile* structure for the given specialisation then a new one is constructed and inserted into the list.

Lists are used here for simplicity; a faster access data structure is used in practice.

3.2.2 Finding and examining execution profiles

An enhancer procedure is constructed for each optimisation within the system. The enhancer contains the information required to find relevant profiles and the associated code. In the case of a manual optimisation included by the programmer, the enhancer contains direct links to both code and profiles. The mechanism for finding the locations of automatically generated profiles and code constructed by the compiler is encoded into the enhancer for that optimisation.

The enhancer also contains links to the appropriate cost function and optimisation procedure. With access to the code, the profiles, the cost function

and the optimiser, the enhancer is in a position to check for and initiate optimisations when required. The type of the enhancer may be written as

proc(database, costFn, optimisationFn)

where *database* is the part of the environment over which the enhancer will operate.

The enhancers may be called during quiescent periods of system execution. In this case enhancer procedures may be registered with a central enhancer mechanism which organises the execution of each individual enhancer when appropriate. The enhancers may also be called directly from system code during execution. Calls may be added to the source code of the system using the same techniques as for the addition of profiling code. The call to the enhancer may depend on an evaluation of the conditions experienced during execution.

Cost functions are first class procedures in the language that are generated during construction of code, either automatically by the compiler or manually by the programmer. They may contain direct links to information known statically about the code to be optimised or else such information may be passed as parameters.

Polymorphic procedures are marked during compilation. Where polymorphic optimisations take place during quiescent times, the enhancer for the polymorphic optimisation browses the persistent environment using object store browsing technology [DB88] looking for these marked procedures. The profiling data structures may be retrieved from the closure of the procedures and passed to the cost function of the polymorphic optimisation.

The cost function determines which monomorphic versions of the procedure are worth constructing, according to a tradeoff over the following costs:

- The extra time spent executing polymorphic specialisations in comparison to their monomorphic counterparts.
- The extra space used to store monomorphic versions.
- The expense incurred in constructing a monomorphic version.

A new monomorphic version is constructed when the cost in execution speed incurred using the polymorphic version is shown to be significant in comparison to the space and time required to construct and store a monomorphic version.

The extra expense of executing a polymorphic version in comparison to the equivalent monomorphic version is caused by the execution of operations that depend on the type information passed as a tag to the encapsulating procedure. A record of the number of executions of these operations is required in order to determine the difference in efficiency between the polymorphic and monomorphic versions. Records of this kind may only be taken during execution of the program being measured and would significantly affect the performance of the program.

By making judicious choice of which execution profiles to collect and which static information to bind into the cost function, a good approximation to the exact cost of a polymorphic specialisation can be made without incurring a significant expense. The static information available to the cost function is the number of operations in the code that manipulate polymorphic values. This may be used to good effect when combined with the number of executions, but is heavily dependent on the flow of control within the program.

The cost function combines the static and dynamic information to estimate the total cost of the associated specialisation and compares this cost with a threshold level to determine whether optimisation should take place.

3.2.3 Optimisation and relinking

The technique optimises code by performing source code transformations followed by recompilation. In a persistent system, the smallest manipulable unit of code is typically the procedure. The granularity at which an executable system may be manipulated is therefore the procedure level. Procedure source code may be attached to executable procedure values: one format for the source is a list of lexemes and associated symbol tables for outer scope references.

In a persistent programming environment a compilation function is available to be used by any program. Where compilation strategies must be altered, the compiler may itself be available for modification.

Type-safe incremental linking is supported using the architecture described in [DCC92], which is based on the use of shared typed persistent locations.

The code transformation process may allow the combination of old code with new in order to support the style of optimisation where new code is an alternative to and does not replace the original code.

In the polymorphism optimisation, when the cost function indicates that a particular optimisation should be performed, the optimiser procedure is passed the source code of the procedure along with the *profile* structure for the relevant specialisation.

The optimiser needs to know how to change the source code: this is given by the *typeReps* encoding contained in the *profile* structure. Based on the tag, the optimiser can transform the polymorphic procedure into an equivalent monomorphic procedure. For example, given the polymorphic *id* procedure and an indication that an integer monomorphic form should be constructed, the

source code transformation of Figure 6 would be performed. In this particular example, the body of the procedure is unaffected by the transformation; were the body to contain references to the quantifier type *t*, they would also be transformed.

```
proc[ t ]( v : *t ; lessThan : proc( t,t -> bool ) )
begin
...
end
=>
proc( v : *int ;
  lessThan : proc( int,int->bool ) )
begin
...
end
```

Figure 6 **A code transformation.**

The transformed source is compiled to give a new executable form optimal for integer specialisations. To link the new code into the running application, the profile type and the transformed procedure given in Figure 5 must be adjusted. The earlier versions were simplified to avoid confusion. The profile structure is extended with a field to hold a procedure code vector, as shown in Figure 7. Before a monomorphic optimisation for the associated type format this is a nil vector. Afterwards it holds the specialised monomorphic code form.

```
type profile is structure(
  tags          : tagCombination ;
  calls         : int ;
  monoVersion   : codeVector )
```

Figure 7 **A data structure to hold profiles and optimised code forms.**

```
let profiles = mkList[ profile ]()
let compiledSort = proc[ t ](
        v          : *t ;
        lessThan   : proc( t,t -> bool ) )
begin
  let monoVersion =
      countCalls( profiles,mkTypeReps[ t ]())
  if monoVersion = nilCodeVector then
  begin
    ! The initial implementation  of
    ! the procedure is used
    .........
  end
  else monoVersion( v,lessThan )
end
```

Figure 8 **Polymorphic procedure augmented with profiling and switching code.**

The transformed polymorphic procedure is extended with code to choose between monomorphic versions as shown in Figure 8.

The procedure *countCalls* is extended to check whether a monomorphic version has been constructed for the given type representations. If one exists then it is returned, otherwise a fail value, *nilCodeVector* is returned. Profiling information is recorded as usual.

3.2.4 Future execution

Future execution will benefit from the optimisations made by the enhancers provided that the conditions during execution do not change radically. Continued profiling will take account of changing conditions. The values in the profiling structures may be reinitialised each time around the enhancement loop. This depends on whether optimisation is dependent on cumulative records or just the experience of the last enhancer time segment.

In the polymorphism optimisation, it may be sensible to reinitialise the profiles on each enhancement cycle to prevent infrequently-used specialisations from eventually being optimised. The time between enhancer invocations may be an input to the cost function. Continued profiling will indicate when new data types are used against existing polymorphic code allowing the system to adjust accordingly. If monomorphic forms are shown to be falling into disuse they may be deleted if space consumption is important.

3.2.5 Measuring the polymorphism benefits

An extended version of the implementation of polymorphism has been implemented in Napier88 and is fully reported in [MDC+91]. The version given here is restricted in order to keep the description of polymorphism and its implementation in Napier88 within the scope of the paper. Full details of the optimisation mechanism are found in [Cut92].

The mechanism appears overly complex when applied to a procedure as simple as the sort procedure; however using that procedure kept the example short. The implemented version of the mechanism requires less switching code to be executed in general. Additionally, in typical polymorphic code there are a number of operations which manipulate polymorphic values, each of which is more expensive that its monomorphic equivalent. Examples are the conversion of data between polymorphic and monomorphic formats, access and update of data structures that have fields of an abstracted type and equality operations, all of which require knowledge of the underlying data's shape.

Figure 9 shows some timings of the sort procedure measured in units of processor time allocated to the

Napier88 process when running on a SUN Sparc 10. The exact timings are not important; it is the percentage speed increase that is of interest. The measurements compare the polymorphic implementation when specialised to integer over a number of random integer vectors against the same procedure once it has been optimised for the integer specialisation. The increase is of the order of 20-25%.

vector size	polymorphic version	optimised version	% reduction
100	57	42	26
150	132	98	19
200	233	173	26

Figure 9 Percentage speed increases over *sort*

A detailed analysis of the polymorphism optimisation mechanism in action is beyond the scope of this paper. However in general use over a range of polymorphic procedures, speed increases for polymorphic code have typically been in the range 25-35%. This is obviously dependent on the number of polymorphic operations that occur within polymorphic procedures. In the sort example above, the number of polymorphic operations is low in comparison to the required switching and profiling. Further experimentation is required to tailor the cost function to the optimal trade-off between speed improvements and the cost of constructing monomorphic code.

4. Other examples of the technique

Query optimisation

Query optimisers traditionally operate during compilation of the query on the basis of the contents of the database which is statically available. In situations where queries are constructed independently of the data over which they operate or where executable versions of queries are long-lived, query execution will become non-optimal when the contents of the database change significantly.

The optimisation technique may be used here to optimise queries on an on-going basis. Periodically the database may be examined to determine whether its contents affect the efficiency of existing queries. Such information may be gathered using profilers embedded in the query code. Where query efficiency is deteriorating, the query code may be restructured and recompiled according to the changed conditions in the database.

Optimisation of method code in object-oriented systems

The objects of object-oriented systems consist of a number of methods operating over encapsulated data. The state of a running system is recorded in the data items encapsulated within each object. Optimisations to executing object-oriented systems must ensure that this data is retained.

The optimisation technique may be applied to these systems. Profiling code is added to method code in order to record details of system execution. According to analysis of the profiles, method code may be transformed and recompiled without affecting the encapsulated data held in the objects.

Data optimisation

The emphasis in this paper has been on the optimisation of code segments. The general mechanism underlying the optimisation technique is also suited to dynamic data optimisations. In order to support a data optimisation such as clustering, the system may be augmented with profiling code to record those data items that are accessed together. The enhancer program for such an optimisation communicates with the store manager, giving an indication of those data items that are frequently accessed at the same time. Care must be taken to avoid an effect similar to thrashing in paging systems, where each section of a system requires a different clustering over the data.

5. Conclusions

This paper has described an technique for incrementally optimising executing systems. The particular benefits of the technique are as follows:

- Executable systems can be incrementally improved as knowledge is gathered about their execution characteristics.
- The technique allows a body of executable code to be expanded with new code segments to handle conditions not provided for during construction of the original code.
- Optimisations are made to running systems. System shutdowns for the installation of optimised code segments are not required.
- Optimisations affect only the code in a system. This is particularly useful in long-lived data-intensive systems or in systems where the code is constructed in isolation from the data over which it operates.

Care must be taken when applying the technique to ensure that system performance is not significantly affected by the introduction of profiling code and on-line optimisations.

Instances of the technique may be implemented in systems supporting the persistence of structured data including code across the activities of software construction, compilation, linking and execution. Persistent programming environments and object-oriented systems are particularly suited to the technique.
The optimisation of polymorphism described in Section 3 has been implemented in the persistent

programming environment of Napier88 [KCC+92]. Typical speed-ups are of the order of 30%. Further optimisations for the language implementation are currently under consideration.

References

[AM85] Atkinson, M. P.; Morrison, R. "Procedures as Persistent Data Objects," *ACM TOPLAS* **1985**, *7* iv, 539-559.

[BDH91] Benzaken, V.; Delobel, C.; Harrus, G. "Clustering Strategies in the O_2 Object-Oriented Database System," ESPRIT BRA Project 3070 FIDE, 1991.

[BOP+89] Bretl, B.; Otis, A.; Penney, J.; Schuchardt, B.; Stein, J.; Williams, E. H.; Williams, M.; Maier, D. "The GemStone Data Management System," In *Object-Oriented Concepts, Applications, and Databases*; W. Kim and F. Lochovsky, Ed.^Eds.; Morgan-Kaufman: 1989.

[Bro89] Brown, A. L. Ph.D. Thesis, St Andrews, 1989.

[Car85] Cardelli, L. "Amber," AT&T Bell Labs, Murray Hill, 1985.

[Car89] Carey, M. "The Exodus Extensible DBMS Project: An Overview," In *Readings in Object-Oriented Databases*; Morgan-Kaufmann: 1989.

[CD91] Cluet, S.; Delobel, C. "Towards a Unification of Rewrite Based Optimisation Techniques for Object-Oriented Queries," ESPRIT BRA Project 3070 FIDE 91/19, 1991.

[Cut92] Cutts, Q. I. Ph.D. Thesis, St Andrews, 1992.

[DB88] Dearle, A.; Brown, A. L. "Safe Browsing in a Strongly Typed Persistent Environment," *Computer Journal* **1988**, *31*, 540-544.

[DCC92] Dearle, A.; Cutts, Q. I.; Connor, R. C. H. "An Application Architecture Using Type-Safe Incremental Linking," University of St Andrews, 1992.

[DOD83] "Reference Manual for the Ada Programming Language," U.S. Department of Defense, 1983.

[GR83] Goldberg, A.; Robson, D. *Smalltalk-80: The Language and its Implementation*; Addison Wesley: Reading, Massachusetts, 1983.

[KCC+92] Kirby, G. N. C.; Connor, R. C. H.; Cutts, Q. I.; Dearle, A.; Farkas, A. M.; Morrison, R. "Persistent Hyper-Programs," In *Persistent Object Systems*; A. Albano and R. Morrison, Ed.^Eds.; Springer-Verlag: 1992; pp 86-106.

[KCC+92] Kirby, G. N. C.; Cutts, Q. I.; Connor, R. C. H.; Dearle, A.; Morrison, R. "Programmers' Guide to the Napier88 Standard Library, Edition 2.1," University of St Andrews, 1992.

[MBC+89] Morrison, R.; Brown, A. L.; Connor, R. C. H.; Dearle, A. "The Napier88 Reference Manual," University of St Andrews, 1989.

[MBC+93] Morrison, R.; Baker, C.; Connor, R. C. H.; Cutts, Q. I.; Kirby, G. N. C. "Approaching Integration in Software Environments," University of St Andrews, 1993.

[MCP88] MacQueen, D.; Cardelli, L.; Paulson, L. "Polymorphism" In *ML/LCF/Hope Newsletter*; 1988.

[MDC+91] Morrison, R.; Dearle, A.; Connor, R. C. H.; Brown, A. L. "An Ad-Hoc Approach to the Implementation of Polymorphism," *ACM Transactions on Programming Languages and Systems* **1991**, *13*, 342-371.

[PS88] "PS-algol Reference Manual, 4th edition," Universities of Glasgow and St Andrews, 1988.

[QL91] Quong, R.; Linton, M. "Linking Programs Incrementally," *ACM TOPLAS* **1991**, *13*, 1-20.

[RT78] Ritchie, D. M.; Thompson, K. "The UNIX Time-Sharing System," *The Bell System Technical Journal* **1978**, *63*, 1905-1930.

[SCW85] Schaffert, C.; Cooper, T.; Wilpot, C. "Trellis Object-Based Environment Language Reference Manual," DEC Systems Research Center, 1985.

[Tha86] Thatte, S. M. In *ACM/IEEE International Workshop on Object-Oriented Database Systems*; Pacific Grove, California, 1986; pp 148-159.