SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 27(5), 497–517 (MAY 1997)

Achieving Incremental Compilation through Fine-grained Builds

TIM COOPER AND MICHAEL WISE
Department of Computer Science, University of Sydney, NSW 2006, Australia
(email: { timc,michaelw } @staff.cs.su.oz.au)

SUMMARY

Traditional programming environments represent program source code as a set of source files. These files have various 'dependencies' on each other, such that a file needs recompilation if it depends on a file which has changed. A 'build tool' is used to process these dependencies and bring the application 'up-to-date'. An example of a build tool is the UNIX 'make'. This paper examines what happens when the dependencies used by the build tool are expressed between *functions* (or *objects*) rather than between *files*. Qualitative differences arise from the difference in granularity. The result is an effective incremental compilation programming environment, based on the C++ language. It is called 'Barbados', and is fully implemented. The environment resembles an interpreter in that changes to source code appear to be immediately reflected in all object code, except that errors are reported early as in compiled systems. Incremental compilation is not a well-used technology, possibly because the 'fine-grain build' problem is not well understood. Nevertheless, incremental compilation systems do exist. The advantages of the system described here are that it works with relatively standard compilation technology, it works for the C++ language including the preprocessor, it is an elegant solution and it is more efficient than competing algorithms. ©1997 by John Wiley & Sons, Ltd.

KEY WORDS: make; incremental compilation; configuration management

INTRODUCTION

Traditional programming environments represent application source code as a set of source files. One source file is said to 'depend' on another if the result of compiling it depends on the contents of the other file; in other words, it needs to be recompiled whenever the other file changes. A 'build tool' is used to process these dependencies and bring the application 'up-to-date', usually with a minimum of recompilation.

A 'build' tool can operate at the *module* level or the *procedure* level (Table I).

File-level build tools are common and well-understood, for example the UNIX 'make' tool.[1]

Procedure-level build tools are not common (in fact, our system is the only procedure-level build tool we are aware of). They are worth studying because they support incremental compilation. Indeed, the fact that the fine-grain build problem is not well-understood may explain why incremental compilation is not more popular than it is—there are *interpreted systems* and there are *compiled systems*, but to

CCC 0038–0644/97/050497–21 $17.50
© 1997 by John Wiley & Sons, Ltd.
Received 5 February 1996
Revised 4 June 1996 and 9 August 1996

Table I. *File-* and *procedure-level* build tools

A *file*-level build tool	A *procedure*-level build tool
Dependencies are expressed between files	Dependencies are expressed between individual objects (procedures, types, global variables and macros)
Compilations are of files	Compilations are of individual objects

achieve the best of both worlds, the fine-grained build problem must be solved. We are not aware of this problem having been reported in the literature before.

There are incremental compilation systems which work at an even finer grain than procedures, for example statements or expressions.[2,3] We have chosen not to investigate finer granularities, because the procedure-level granularity gives us sufficient performance and is considerably easier to program.

The system to be described has been implemented in an integrated persistent programming environment called 'Barbados'. For an overview of the system, see Reference 4.[4] Barbados is based on C++ and runs on the Windows NT operating system.

Motivation

We are interested in fine-grained builds and incremental compilation because:

(i) It significantly reduces the time that one wastes waiting for a program to compile.
(ii) It means that interactive compilers can be built, so that, for instance, (a) code can be tested with simple calls at a command line, (b) during debugging, commands or queries can be issued using the same language and same compiler and same context as the code being debugged.
(iii) An offshoot of the solution is that makefiles (or at least our equivalent structures) are automatically and transparently maintained.

Scope of the work

We investigate fine-grained build techniques within an integrated program development environment.

We have implemented these techniques within a system called Barbados. Barbados is an 'integrated persistent system', which means that the entire software development process takes place inside a persistent store. However, we propose that the ideas we present are not limited to persistent systems. In fact, if features were added to Barbados to facilitate importing source code and exporting executable programs, we believe Barbados could be used as an integrated development environment for non-persistent systems.

The Barbados C++ language has some extensions to C++, and some omissions,*

* Virtual functions, virtual inheritance and some other features of C++ are not yet implemented, because of limited resources.

however these differences are not relevant to this paper—the languages are very similar.

The system described is intended to replace in part tools such as the UNIX 'make'. However, we have not attempted to provide the generality of such tools—our 'build' system is specifically directed at scheduling the compilation of source code objects in a single language. As such, it can be regarded as just one component of a configuration management system.

The techniques described here were designed to scale up to very large programming projects. We expect that the following results apply to any statically typed, procedural or object-oriented language, with the proviso that languages with nested procedures (such as Pascal) will need to be written without extensive use of nested procedures.

Claims of the work

We claim that:

(i) This system can be applied more generally than other incremental compilation techniques.
(ii) Empirically, it significantly outperforms other incremental/batch C++ compilers.
(iii) It is a relatively simple technique.

THE PROBLEM STATEMENT

We propose that any fine-grained build tool should satisfy the following requirements:

(a) Generate dependencies automatically.
(b) Bring or keep all program components up-to-date transparently.
(c) Ensure that there are never multiple versions of a compiled object (i.e. products of the same source code), in the program development area.
(d) Ensure that no out-of-date code is ever executed.

An example

For an example of what a fine-grained build tool is required to do, consider the following dialogue with an incremental compilation system.

This example consists of Barbados C++ code being entered at the command-line. The command-line interpreter automatically compiles everything given to it, and if an expression or statement is entered, it is automatically executed. If code is to be executed, the build tool is then called on the objects about to be called/accessed. This makes the system look like an interpreter while otherwise having the properties of a compiler.

```
typedef struct {
 float x,y;
} vector;                      // 'vector' is compiled now.

float length(vector v)
{
 return sqrt(v.x*v.x + v.y*v.y);
}                              // 'length' is compiled now.
```

```
typedef struct {
 boolean rel_or_abs;
 float x,y;                 // The old definition of 'vector' is
} vector;                   // replaced with this one.

vector v = { true,3,4 };    // This statement is compiled and executed.

length(v);                  // The build tool automatically
= 5.0;                      // detects that 'length( )' is out-of-date,
(output is highlighted)
                            // so it recompiles it and executes the
                            // new version.
```

Although incremental compilers don't *need* an interactive command prompt, such an interface is an obvious exploitation of the advantages of incremental compilation (so much so that it would be unfortunate to have one and not the other). This means that the build tool will rarely be required to bring a program's 'main()' function up-to-date—it is more important that the build tool works efficiently for the individual functions that are interactively tested.

Some subtleties

The implementation of a procedure-level build tool can be complicated, for the following reasons:

(a) Dependencies must be generated automatically (they are too numerous for the programmer to generate them).
(b) Functions, type/class-definitions, variables and preprocessor macros are all involved in the process.
(c) An entity can change size after a recompilation, and so depending on the implementation, it might change address.
(d) The tool must deal with recursive types and recursive functions.
(e) The tool must deal with thousands of entities (or more) rather than tens of entities.
(f) Objects can be modified at any time, so whereas traditional compilers can rely on the static source code to order the compilation of objects such as structures,* a fine-grained compiler cannot.
(g) When objects are deleted, this could potentially cause dangling pointers in dependency lists.
(h) Objects may become out-of-date actually during the recompilation process, e.g. by a function parameter value changing, which has implications for the ordering of recompilations.
(i) Dependencies can change during a compilation.

Our solution to the fine-grained build problem is presented in two halves: how we generate dependencies, and how we process them.

* e.g. to generate appropriate sizeof( )'s and field offsets.

OUR SOLUTION: GENERATING DEPENDENCIES

The entities that the build tool must deal with

In the C++ language, there are the following types of entities which the build tool must deal with. (Other statically typed, compiled languages will have similar entities.)

A type or class

```
typedef struct { int x,y;
        } Point;
```

Often depends on other types, because of pointer fields and nested objects. Function prototypes can depend on types/classes.

A static variable

```
Point P;
```

Only depends on the type; although other objects can depend on it. If you change its type, this can affect everything that depends on it.

A function

```
Point Point::Reflect( )
        {
         x = -x;
        }
```

Consists of a prototype and a body. If the prototype changes, e.g. you modify a parameter type or return type, this can cause substantial changes in other functions which use it. If the body changes, at most this can change the address (which can be dealt with in various ways).

A preprocessor macro

```
#define min(a,b) \
        ((a < b) ? a : b)
```

Does not depend on anything. On the other hand, any object which uses a macro is said to depend *directly* both on it and the objects generated in its expansion—which can only be determined during the compilation of that other object.

In the remainder of this paper, we will refer to all of these entities collectively as ‘make-objects’, or simply as ‘objects’.

In our system, the member functions of classes are considered to be make-objects in their own right. This means that dependencies can be expressed specifically to the member functions rather than to the class definition. The alternative was to lump them together with the class, for dependency purposes, but this was considered too large-grained to be desirable—it would generate too many recompilations.

In C++, the above entities are precisely those entities which have source-code and can be separately compiled. Therefore, they are the entities used by the build tool. Local variables, even local static variables and 'goto' labels do not participate in the build tool. In our system, immutable system functions such as 'strcpy( )' also do not participate in the build tool. The above taxonomy should also apply to most other modern compiled languages.

Dependencies

In this paper, we say that object 'A' *depends directly* on object 'B' if a change to object 'A' would cause object 'B' to become out-of-date. 'Out-of-date' means that 'B' will change (or might change) if it were recompiled.

Therefore, an object depends on any other object or type-definition that is directly used by the object. This includes every function called, every static variable accessed, every type mentioned. In an object-oriented language, it also means the methods used—for example:

```
X * Y;      /* might use: */ operator*(complex, complex)
```

We shall use the phrase 'dependency list' to refer to the set of objects which a given object directly depends upon. The dependency list for an object can be generated quite easily during its compilation.

The 'Interface' optimisation

In UNIX, an object's time-stamp is updated whenever it is recompiled. This is tantamount to asserting that the object changes each time it is recompiled.

An effective optimisation is to separate an object's *interface* from its *body*. The *interface* would be defined as that part of the object capable of affecting other objects (in the static world of compile-time). In practice, this usually means a function's prototype. Depending on how functions are implemented and how recursion is dealt with, the address of a function might also belong in the interface, since this is part of the view that the outside world sees.

An object would only be said to change if its interface changes. Interfaces change relatively infrequently. Therefore, making this distinction dramatically reduces the frequency of objects being out-of-date and needing recompilation. (In this framework, we like *interface-change-stamps* to be as old as possible and *compile-stamps* to be as new as possible).

Depending on which build algorithm is used, this distinction can be a necessity rather than an optimisation because of recursive types and functions.

Being up-to-date

We give each object a time-stamp which marks the last time its 'interface' changed, as well as a time-stamp marking the last time it was compiled.

An object is out-of-date, i.e. requiring recompilation, *if it depends on an object whose interface has changed since this object was compiled.*

Generating dependencies

The process of generating dependencies comes down to the compiler marking all the functions, static variables, types and macros that it comes across during the compilation of an object. A dependency of the form 'A depends on B' can either be stored in a list associated with 'A' or a list associated with 'B'. Because of the top-down nature of our algorithm, we prefer to store dependencies with the object that depends, rather than with the object that is depended upon.

OUR SOLUTION: MAKE( )

In the remainder of this paper, we will refer to our build tool as 'Make( )'. We will refer to the application programmer as the 'user', since they are the user of Make( ).

When is Make( ) called?

Make( ) is a completely transparent feature, which means that the user is never required to explicitly invoke it and is not informed that it is even operating.

Make( ) provides the abstraction of every object being always up-to-date, and guarantees that no out-of-date code is ever executed. This means that the compiler resembles an interpreter. However, unlike interpreters, errors are reported early, namely whenever the user executes code whose transitive closure includes an object with an error.

Whenever the user enters a compileable entity at the command-line, that entity is compiled. Sometimes that entity also needs to be executed, e.g. expressions and other statements.

Make( ) is called *lazily*, i.e. we call Make( ) only if or when we need an object to execute some code. When it is called, it is called recursively over all the objects accessed in the transitive closure of the code we want to execute. This means that the user can make multiple modifications to various objects, and compilation is only scheduled when the user calls some affected code.

The dependency list for an object is updated each time it is compiled.

Make( ) is never called at run-time—it is purely a 'compile-time' or 'development-time' function. Admittedly, the run-time/compile-time distinction is quite blurred in an interactive system such as this one: what we mean by this distinction is that once user-code begins execution, Make( ) is never invoked again until control returns to the Barbados command-line. For example:

```
typedef struct {          // A type-definition. The compiler is
 int x,y;                 // called, but Make( ) is not, since no
} Point;                  // code is generated.

Point P;                  // A variable declaration. The compiler
                          // is called, but Make( ) is not.

Point Point::Reflect( )   // A function definition. The compiler
{                         // is called, but Make( ) is not, since no
 x = -x;                  // code is generated for immediate
}                         // execution.

P.Reflect( );             // An expression. The compiler is called,
```

```
                        // then Make( ), and then the code itself
                        // is executed.
Point P=Q+C;            // A declaration that includes an
                        // expression. Calls the compiler,
                        // Make( ) and then the code itself.
Point A,B,C;            // A multiple declaration. Only the
                        // compiler is called.
```

The 'root' object

We assume that the purpose of one invocation of Make( ) is to bring a given 'root' object up-to-date. This root object could be `main( )` in an application, or it could be a statement issued at the command-line in interactive mode, for example:

```
A( ) + B( ) + C( );
```

In this case, this command is wrapped up in a function object called '`_top_level`' which has dependencies to all three functions.

The purpose of having a 'root' object is to (a) bring up-to-date only those objects we need for the current command, and (b) to have a terminal object on which everything ultimately depends. In fact, sometimes this root object will need to be compiled twice—once to generate dependencies and then later if Make( ) causes types (and hence methods) to change.

Our algorithm

A first approximation to a build algorithm could be: 'recompile everything that is out-of-date'.

This is not sufficient because objects can become out-of-date during the recompilation of another object, even if they have been compiled once already during this call to the build tool. For example, suppose the type `complex` and the object `complex F(complex)`; are both out-of-date. If the function is compiled first, then subsequent compilation of the type could cause the function to be out-of-date again.

This illustrates the fact that the order of compilations can be important. As a second approximation, therefore, suppose we take the root object and follow the transitive closure of its dependencies. When we reach an object with no dependencies, we rise up the dependency graph, recompiling every out-of-date object as we go.

This has the benefit of compiling an object's antecedents before compiling the object itself. However, this algorithm is insufficient for two reasons:

(a) The first reason is that it doesn't deal with circularities in dependency graphs (circularities arise through recursive functions or types). Complicated recursive types might mean that some objects actually need to be compiled twice or even more times,* so it is not possible simply to ignore dependencies to objects already reached.

* Actually, this is implementation-dependent: if all objects are referenced by pointer-pointers and pointers to classes are distinguished in the dependency lists from nested structs, this situation can be prevented.

(b) The other reason is that a compilation of an object can actually cause that object's dependency list to change. For example in C++, a change in a macro or in a type can cause that function to *depend directly* on different objects. These objects newly-inserted into the dependency list will not have been checked.

Therefore, our algorithm detects circularities (by marking objects reached) and doesn't descend into objects already examined during the current run of Make( ). It also repeats the whole process as many times as it takes before there are no more compilations. This solves both problems.

This may appear at first to be inefficient, and to cause unnecessarily many recompilations because recursive objects will be compiled more than once. (The usual solution is to have a separate back-patching or a linking phase[5]). This algorithm deals with the problem by having multiple compilations of an object so that the successive compilations effectively insert the recursive references. However, this extra amount of recompilation will seldom amount to a significant delay in response, and changes do not propagate around a cycle of recursive references. Only recursive code requires more than one compilation during one run of Make( ), and only very complex recursive code requires more than two compilations during one run of Make( ).

The typical scenario is thus: the transitive closure is performed for the first time, during which the majority of out-of-date objects are recompiled. During the second pass, there may be a few more recompilations necessary because a recursive type or function changed its interface during the last pass or because dependency lists changed and new objects were introduced. In the third pass, no recompilations will be needed or performed, and so there will be no fourth pass.

It is our experience that the time spent doing this graph traversal, even multiple times, is insignificant compared to the time spent on recompilations.

The 'Check-stamp' optimisation

According to the above algorithm, each execution of a program would require a traversal of its transitive closure—every component object it uses. Although this can be quite quick even for very large programs, we have implemented one optimisation to avoid this traversal.

Each object is stamped with a third time-stamp called the 'check-stamp'. This indicates the last time it was examined by Make( ). This stamp is mainly used to avoid cycles when we look at the transitive closure of an object.

We also log the last time there was any change to an object's interface in the whole system.

If an object's check-stamp is more recent than this global 'interface-change' stamp, then it is impossible for that object to be out-of-date. Therefore we can skip even the transitive-closure checking stage. This is handy for long streams of little commands being entered interactively.

Example

To provide an example of our algorithm in action, and also to discuss recursion, consider the following code:

```
struct VALUE {
 int a,b;
 struct TREE *parent;
};

struct TREE {
 struct TREE *left, *right;
 struct VALUE val;
};

VALUE f(TREE t)
{
 if (t->left)
 return g(t->left);
 else return t->val;
}

VALUE g(TREE t)
{
 if (t->right)
  return f(t->right);
 else return t->val;
}

TREE t;
. . .
f(t);                         /* The build tool is first activated here. */
```

The dependency graph is as shown in Figure 1.

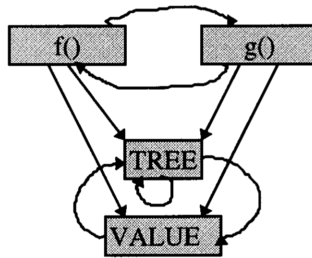

Figure 1. Dependency graph example

The algorithm performs as follows:

1. The build tool starts at ‘f( )’.
2. Suppose it descends to ‘TREE’ first and then from ‘TREE’ into ‘VALUE’.
3. ‘VALUE’ will have dependencies back to ‘TREE’, however these are not followed since ‘TREE’ has already been reached on this pass.
4. Therefore the recursion bottoms out. ‘VALUE’s time-stamps are examined, and since ‘TREE’ was changed after ‘VALUE’ was last compiled, ‘VALUE’ is deemed out-of-date and it is recompiled. Since the size of the type-definition has not changed, it returns to the same location as before.

5. Returning up a level, 'TREE' is then examined. It is likewise out-of-date, and it is recompiled.
6. Similarly for 'g( )' and finally for 'f( )'.
7. Since there were some compilations (which could have affected interfaces or dependency lists), the entire pass is repeated.
8. This time only 'TREE' is recompiled. This is because all the other objects are up-to-date, and yet 'TREE' depends on 'VALUE' and 'VALUE' had a recent interface-change when the reference to the 'TREE' stub was replaced with the new 'TREE' object.

An alternative solution is described below, with reasons why we chose to adopt the above algorithm.

An alternative algorithm

Suppose we separate function bodies and function prototypes. Callers would point to a function header rather than a function body, and the body would be free to change size or implementation without affecting the prototype. A similar approach would then be used for types, for example, a structure definition which includes a pointer to another struct would point to a 'prototype' for that struct which specifies the size of the function and a pointer to the actual structure definition. Dependencies would then be expressed between these objects (mostly from a body to a prototype). Furthermore, instead of a 'compile stamp' and an 'interface-change stamp', there is a single stamp on the 'body' object and a stamp on the 'interface' object.

Incidentally, this kind of approach has a similar effect as having multiple types of dependencies, e.g. Eiffel discriminates between client/supplier dependencies and inheritance dependencies.

For example, the above example would be rearranged as shown in Figure 2. This structure has the advantage that any valid program would be represented by an acyclic directed graph (at least after the initial compilations during which dependencies are generated).

Furthermore, a build algorithm can be written which discovers dependencies on the fly and yet is guaranteed to terminate:

(a) Recursively descend the program graph.
(b) Upon ascent from a recurse, compile an object if it is out-of-date.

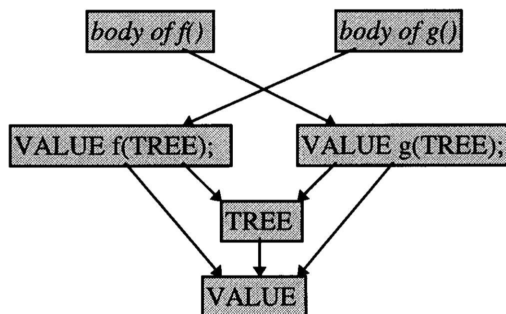

Figure 2. Alternative algorithm example

(c) Compare dependency lists before and after a recompilation. Whenever they change, recurse again down the new dependencies and return to part (b).

This works because, given an existing program graph, a change in dependencies can only affect objects higher in the dependency graph.

Each function call will then require a run-time indirection from the function-object pointer to the code itself, or a linking phase to make these conversions. The linking phase would be applied after all recompilations have been made, and it would be applied to all objects which were recompiled since the last linking phase. Dangling 'function call' pointers from old functions to invalid addresses will never be invoked because the build tool would prevent it.

Our choice of algorithm

Of these two algorithms, we chose to implement the first algorithm in our system because

(a) It was a little awkward for us to separately compile a function body and its prototype. (Similarly for structs, if a body/interface distinction is made.)
(b) We wanted to generate CALL instructions to the physical location of a function's body, whereas the above algorithm would require the use of pointer-pointers to function bodies (so that when an object increases in size and needs to be moved elsewhere, existing references remain valid).

Note that it was important for us to implement a 'lazy' algorithm rather than a 'greedy' algorithm. A 'lazy' algorithm is one in which objects are compiled on a need-to-use basis. A greedy algorithm is one in which objects are recompiled as soon as they become out-of-date. A greedy algorithm would perform a lot of unnecessary compilation, because often an object becomes out-of-date yet does not need to be used.

Deletion of objects

What happens if an object is deleted? Does it cause dangling references in the dependency lists? Here is a typical scenario:

```
int X;          /* X is created */
int f( )        /* Function f( ) is created. */
{               /* It depends on 'X'. */
 return X;
}

delete X;       /* X is deleted. */

f( );           /* Make( ) runs on 'f'. It checks the dependencies.*/
                /* If 'X' has been freed, this will cause all kinds */
                /* of errors. We really just want an error */
                /* gracefully reported. */
```

We solved the problem in the following way. Dependency-lists and other information relating to the build tool are stored separately from the object itself. They are stored

in a system object called a 'make-node' which is linked to the actual object. Dependency lists actually consist of a list of pointers to make-nodes. When we delete an object, we free the object itself but its make-node is recycled. Whenever we examine a dependency from one object to another, the time-stamps provide us with enough information to distinguish between a true dependency and a dependency to an object which has been recycled.

Therefore the above code will correctly generate an error, as follows:

```
f( );
⟨Error: Undefined identifier: 'X'⟩
```

Furthermore, an object which has not been successfully compiled is marked as out-of-date, regardless of its dependencies. Therefore, we can create an object **X** and call **f( )** and it will successfully recompile **f( )** with a reference to the new **X**:

```
int X=10;        /* An object called 'X' will be recreated with a */
                 /* new make-node. */
f( );            /* 'f( )' is now recompiled successfully. */
= 10;
```

If an object is *renamed*, it is deemed to have modified its interface, but otherwise it retains the same make-node. Updating its interface-change time-stamp will cause all the objects that depend on it to be considered out-of-date, so they will be recompiled before being used (although probably generating an error unless a new object with the old name is created).

Make( ) and classes

If a class (or struct or union) is redefined, i.e. a new definition overwrites an old definition, the two must be checked for equality. Two structs are considered equal if and only if they have the same members with the same names, types and offset (or the same name and type in the case of member functions). If the class has changed, then the 'interface-change stamp' of that class is updated and everything dependant on it will immediately be considered out-of-date.

The preprocessor

The C/C++ preprocessor is integrated into Barbados. Macros can alter how a function is compiled in many ways—by inserting braces, by pasting tokens, inserting keywords and so on. However, this does not affect Make( ), since very few assumptions are made about the relationship between two successive versions of an object. (The only problem arises if a macro changes the name of a compiled object. In this case, the system will just generate an error 'Source-code lost for object X'.)

If a macro is written to replace some other identifier, e.g. a typedef name, then the macro will overwrite that identifier. This will cause the old typedef's interface-change stamp to be updated. Therefore, all objects which depend on that identifier will be marked out-of-date and will be compiled before being used again.

Conditional compilation, if limited to the inside of a function, is supported with

no extra effort by the build tool. '#include' files are not supported in Barbados because they are unnecessary.

DISCUSSION

Does Make( ) scale to very large programs?

The algorithm used by Make( ) involves a traversal of the transitive closure (through dependencies) of the program being executed, where this traversal is performed each time code is executed following some modification to the program. It might seem that this graph traversal would be infeasibly slow for large projects which consist of thousands of functions, especially since the aim of an incremental compilation system is to allow the user to make frequent small changes to a program.

For a worst case, consider a menu program which allows the user to select from a menu of large applications—each time the user interactively modifies something, the following call to the menu program will involve a traversal of the dependency graphs for every application.

However, this is not considered a problem for two reasons. Firstly, a program can actually be very large before this becomes a serious problem—a simple graph traversal with infrequent recompilations can be quite quick even for quite large projects. Secondly, Barbados actually organises objects into collections called 'large-grained objects', which among other things can help factor out these dependency-checks. However, large-grained objects are beyond the scope of this paper—see Reference 6.[6]

Indeed, the motivation for incremental compilation comes partly from the problem of developing very large applications.

Does Make( ) reach a unique stable configuration?

Let us define a *stable configuration* as a set of program objects which do not change through recompilations of the same source-code.

There is nothing in the Make( ) algorithm to guarantee that a stable configuration is unique. However, in Barbados with C++, it almost always is. For a counter-example, consider the following case:

```
typedef long A;
typedef A B;
typedef B C;
typedef C A;
```

In this cycle of declarations, the last line overwrites the source for the first line. This cycle will stabilise with all A, B and C all being 'longs', and yet if 'A' had initially been set up as a different type, we would have the same set of source code but with a different configuration.

The authors do not know of any static set of source-code which through luck or ordering can yield alternative stable configurations.

Should Make( ) be an application or an intrinsic feature?

With a fine-grained build tool, the dependencies are too numerous to have the user specify them, and the invocations of the build tool are too frequent to have the user make them. Therefore we consider it necessary for Make( ) to be a transparent feature. This means it must be implemented at the system level as an intrinsic feature.

Could the UNIX 'make' be used in the way outlined here?

'make' is the ubiquitous UNIX build tool. Related to it is the 'makedepend' program, which automatically constructs makefiles (or at least makefile dependency lists) given C source files and some knowledge of the C language. One can consider whether the combination of 'make' and 'makedepend' and a C compiler optimised for compiling small sections of code, could give the benefits outlined in this paper.

However, we don't think 'make' could be used as a fine-grained build tool. It inherently deals with transformations from one 'file' to another (for example, time-stamps are based on files' UNIX time-stamps). The objects which a fine-grained build tool deals with are too small for it to be efficient to treat them as files. Furthermore, 'make' has no concept of the distinction between an object's interface versus its implementation, and therefore does not cope with circular dependencies. Also, to properly maintain fine-grained dependency-lists really requires a very close degree of integration between the compiler and the build tool. (Dependencies are very numerous and therefore need to be maintained transparently, they are very dependent on the language definition, and they change very frequently.)

What other languages/systems would this apply to?

While we have only implemented this system for C++, we consider it a quite general solution to the fine-grained build problem. We therefore anticipate that it could apply to any statically typed compiled languages, object-oriented or otherwise.

To apply this build tool to another language would involve modifying that language's compiler to (a) be efficient at compiling single objects at a time, (b) generate dependency information, and (c) interact with the Barbados persistent store.

How does this solution fit in with configuration management?

Configuration Management (CM) is the broad problem which includes building applications, as well as version control, data evolution and supporting different platforms. Tackling these broader issues is a direction for future research.

Part of our design for supporting CM involves a tool called 'Release( )', which would package up arbitrary programs (program = a function + its transitive closure) into a single object, separate from the source-code. This means that the program can be used independently from changes to the source-code, since the links to source-code have been severed. This would be useful for generating release versions of software as well as supporting teamwork. (At the same time, cross-function optimisations could potentially be employed.)

Our goal of ensuring that there are never multiple compiled versions of the same object should be understood to exclude such 'released' software.

RESULTS

Our system was compared against two other development environments. The experiments were based on comparing how many lines of code each compiler had to compile to bring a program up-to-date. A single large application was used:

Test application	'tt' : A program for generating high-school timetables using AI search techniques
Size and statistics	38 modules, all written in C; 38 header files 20159 lines of code in *.c files 1161 lines of code in *.h files

This application was not originally written for Barbados, and it was not modified in any substantial way for Barbados. It was chosen as the test application on account of these facts and its size. There is a high degree of interaction between the modules, i.e. it has a relatively dense dependency graph. There is a central header file of about 500 lines which contains a number of mutually recursive type definitions and which is included by almost every module.

The environments compared are described in Table II.

In Barbados, the preprocessor is integrated into the lexical analyser. The 'gcc' preprocessor has the property that it outputs the same number of lines as are input through include files and source files, which means that the number of lines that pass through it can be compared with the number of lines passed to the Barbados compiler.

In Visual C++, the system informs the user of which source files are being analysed or compiled. This information, along with the compiler documentation's description of the incremental compilation techniques, was used to calculate how many lines of code were probably compiled.

Various scenarios, consisting of small changes to an application's source-code

Table II. Environments compared

	Barbados	gcc v2.7	Visual C++ v4.0
Description	The system described here	The gnu C compiler, common on UNIX systems, together with the UNIX 'make' command	The Microsoft C/C++ integrated development environment; supposedly a state-of-the-art incremental compilation system
Method of counting # of lines compiled	A special function was added to count lines of source code as they're compiled	The 'makefile' was extended to pass input files through the preprocessor and then through a line-counter	Analysis of the 'build window' messages and a bit of guesswork based on the descriptions of compilation techniques given in the documentation

were tried out in each environment. The number of lines of code each compiler needed to compile are listed in Table III.

These figures show the vast differences between Barbados and the other environments. The systems are compared on 'lines-of-code compiled' to compare the efficiency of the 'build' algorithms, rather than the speed of the compilers.

The traditional environment, represented by 'gcc+make', suffers because (a) each module must separately compile large amounts of header files, and (b) because large units of code must be recompiled when each change is made. The standard C header files comprised about three quarters of the figures below. Of the remainder, about half consists of source files (*.c) and half consists of header files specific to that project.

The Visual C++ environment has the following features available to speed recompilation: (a) precompiled headers; (b) incremental compilation of function bodies; (c) 'minimal rebuild' (described below); and (d) incremental linking. All features were turned on for the experiments, except for precompiled headers of application header files. These features are detailed in Reference 7. Interestingly, features (a) and (b) are turned off by default. The reason given was that the cost of maintaining the necessary information is generally greater than the cost savings for small and medium-sized applications. The same is not true for Barbados: in Barbados, the cost of compiling and building a program increases approximately linearly with the size of the program.

The 'minimal rebuild' feature is the closest feature to the techniques described in this paper. It consists of maintaining a database which relates each source file to the class interfaces given in each header file. If something changes inside a class

Table III. Lines of code needed (multiple numbers mean multiple examples)

Lines of code	Barbados	gcc	Visual C++
To rebuild all modules	20,159	150,231	21,320
Adding a comment to the central header file	4	139,736	34,926
Adding a comment to a type-definition in the central header file	30,37	139,736	1107 lines of analysis (?)
Adding a member to a struct in the central header file	6371, 5448, 6173	139,736	815, 2329, 815 if added to the end of struct; 26,533, 27,822, 26,533 if added to start of struct
Adding a function prototype	4	139736 with the central header file; 21333, 11402, 25,919 with peripheral header files	34926 with the central header file; 9681, 3229, 7594 with peripheral header files
Adding a parameter to a function	539, 380, 326	12699, 13823, 53,272	3662, 5076, 18,139
Changing the implementation of a function	9, 89, 48, 31 for small changes; 310, 336, 51, 34 for large changes	3446, 2952, 3651, 3329	9, 89, 48, 31 with padded object files

interface, it is these 'class' dependencies which are checked with each source file rather than the normal dependencies linking source files with header files. (Our application had to be compiled as C++ source for this feature to be turned on.) However, changes made to header files outside class/struct definitions are not dealt with by this optimisation and they often cause large amounts of recompilation. Such changes can include adding comments, declaring new functions or objects, or adding new classes. (Admittedly, good C++ header files should consist entirely of class definitions, but nevertheless this technique would be a lot more useful if it were more robust.)

Visual C++ has an incremental linker and a full linker. The environment supposedly uses the incremental linker in most situations but occasionally needs to resort to a full link. Barbados does not require a separate linking phase; this is performed as a by-product of compilation.

Even very large functions can be recompiled in Barbados without a noticeable delay, e.g. 300 ms for a 500-line function. This vindicates our decision not to explore finer grains of incremental compilation.

LITERATURE

The closest work we could find to this system is probably the LOIPE system.[8] This was an integrated programming environment for a language called GC, designed for LOIPE, which was a type-safe version of C. This system was quite advanced for its time. It consisted of a number of tools, all of which interacted via a syntax-tree representation of programs (including the editor). LOIPE was part of the Gandalf project.[9] While this system is probably quite efficient at reducing compilation (although empirical comparisons do not seem to be available), the paper did not describe how the system deals with the various subtleties described in section 2.2. Perhaps the language implemented was a very limited subset of the C language (the language was defined in a technical report).

Incremental compilation occurs naturally within an integrated development environment. A number of incremental compilation environments exist, for example Orm,[10,11] Gandalf,[12] The Synthesizer Generator,[13,14] PSG[15] and Mentor.[16]

Most of these systems are based on formal language definitions of one kind or another. For example, the Synthesizer Generator uses attribute grammars, Mentor uses 'METAL': a 'meta-language' for specifying languages, Orm uses 'Door Attribute Grammars' (an extension to attribute grammars) and PSG involves a custom-designed 'formal language definition language'. Our system is different from these because it is based on very standard compilation technology. That is, it achieves incremental compilation using new 'build' technology rather than using new compilation technology. As a consequence, we do not suffer any of the limitations of having to express a language formally. Formal language definitions can suffer limitations such as: (a) not being able to express all classes of grammars; (b) having trouble with complex naming schemes and cross-module links; (c) not being able to handcraft the compiler for better performance or intelligent error-reporting.

All of the above systems perform incremental compilation by manipulating syntax-tree representations of programs. In fact, text versions of program source are not even stored, since the editors, compilers and debuggers in these systems all use the syntax-trees. This makes it possible to implement very fine grain incremental compi-

lation, however it also reduces the scope for optimising the generated code. None of the papers reviewed discuss (explicitly) issues such as propagating changes in type definition throughout programs. The Synthesizer Generator and its precursor, the Cornell Program Synthesizer either make it a non-issue by virtue of the fact that they interpret the syntax trees directly, or they propagate such changes greedily. PSG has a 'lazy compilation' algorithm, whereby the modifications to a syntax tree's attributes are made when execution reaches that point. By comparison, Barbados manipulates ordinary text representations of programs, rather than syntax-tree representations. This is an advantage for users who prefer to use standard text editors to create programs (although the PSG editor does have a 'textual mode'). It is also a much more compact representation. Barbados performs procedure-level incremental compilation, which means that traditional code optimisation techniques can be used. Also, compilation (and therefore error-reporting) occurs just prior to execution of a program. We would argue that this is preferable to propagating all changes as soon as modifications are made to a source object, because such a 'greedy' algorithm would perform much unnecessary processing of code; and also that it is preferable to compiling objects/statements as execution reaches them because such a system would report errors too late.

As a result of our approach, we have been able to implement a reasonably rich subset of the C++ language. Our implementation supports the preprocessor, inheritance, overloading (and some persistence extensions which would tax any language definition language). Furthermore, we have been able to perform a side-by-side comparison of our system with other program development environments, with a large and real software project, to demonstrate the relative amount of code sent through the compiler; with good results. (The advantage of our system is that it achieves incremental compilation not through new compilation technology, but with standard compilation technology and a new 'build' technology.)

Of the above systems, Orm, the Synthesizer Generator, PSG and Mentor are based on statement-level or expression-level incremental compilation. The disadvantage of these systems is that the opportunities for the compiler to allocate registers and optimise code is considerably reduced. The advantage is that more incremental compilation can be achieved.

There are several books and papers on the UNIX 'make' program and its variants, including the original paper by Feldman (Reference 1), plus References 17 and 18.[17,18] However, as discussed above, the UNIX 'make' does not address the problems of fine-grained builds.

A relevant paper is 'Smart recompilation',[19] which discusses a method for minimising compilations of modules by processing source files to separate interfaces from implementations. This is also in the context of traditional systems and compiling at the 'modules' level. Another paper of some relevance is 'Lazy and incremental compilation,[20] which discusses programs that compile fragments as execution reaches an uncompiled fragment. This concept of 'lazy compilation' is an attempt to reduce compilation times by a very different method to ours. While we consider it a worthwhile avenue to investigate, we suspect it could be a very difficult problem in general, and we are satisfied with our results with 'lazy builds'.

Various programming systems and integrated development environments have their own built-in build tool, e.g. all Ada and Eiffel systems. The 'Eiffel' tool shares

some similarities with the build tool as described here,* although the basic algorithm does not generalise to a language such as C++. Unfortunately, there do not seem to be any published sources describing the details of Eiffel's build except Reference 21.

CONCLUSIONS

This paper has described what a fine-grained build tool is, why it is useful, and what issues arise in its construction. A complete fine-grained build tool has been described in this paper (for statically typed, compiled languages), and it has been implemented in the Barbados system.

This system is considerably quicker than traditional environments at bringing applications up-to-date after small modifications. It is also simpler, more general and more efficient than competing incremental compilation systems.

REFERENCES

1. S. I. Feldman,'Make—a program for maintaining computer programs', *Software—Practice and Experience,* **9**(3), 255–265 (1979).
2. M. J. McCarthy, 'Incremental code generation in a distributed integrated programming environment', *PhD Thesis*, Flinders University, South Australia, 1996.
3. J. Earley and P. Caizergues, 'A method for incrementally compiling languages with nested statement structure', *Comm ACM,* **15**(12), 1040–1044 (December 1972).
4. T. Cooper and M. Wise, 'An integrated persistent programming system', unpublished, http://www.cs.su.oz.au/~timc/Barbados, 1995.
5. A. V. Aho, R. Sethi and J. D. Ullman, *Compilers, Principles, Techniques and Tools*, Addison-Wesley, Reading, MA, 1986.
6. T. Cooper and M. Wise, 'The case for segments', *Proceedings of the International Workshop on Object Oriented Operating Systems*, 1995, pp. 94–102.
7. J. Henshaw, 'Inside visual C++'s incremental build technologies', http://www.microsoft.com/VISUALC/V4/V4tech/incrbld.htm.
8. R. Medina-Mora and P. H. Feiler, 'An incremental programming environment', *IEEE Transactions on Software Engineering,* **7**(5), 472–482 (September 1981).
9. D. Notkin, 'The Gandalf Project', *Journal of Systems and Software* (5), 91–106 (May 1985).
10. B. Magnusson, M. Bengtsson, L. Dahlin, G. Fries, A. Gustavsson, G. Hedin, S. Minor, D. Oscarsson and M. Taube, 'An overview of the Mjolner Orm development environment: incremental language and software development', *Proceedings of 2nd International Conference TOOLS (Technology of Object-Oriented Languages and Systems)*, 1990.
11. A. Gustavsson, 'Viewing the evolution of software objects in an integrated environment', *Proc. 2nd ACM SIGSOFT Workshop on Configuration Management*, November 1989.
12. D. Notkin, 'The Gandalf Project', *Journal of Systems and Software* (5), 91–106 (May 1985).
13. T. Reps and T. Teitelbaum, 'The synthesizer generator', *ACM SIGPLAN Notices,* **19**(5), 42–48 (May 1984).
14. T. Teitelbaum and T. Reps, 'The Cornell program synthesizer: a syntax-directed programming environment', *Communications of the ACM,* **24**(9) (September 1981).
15. R. Bahlke and G. Snelting, 'The PSG System: From formal language definitions to interactive programming environments', *ACM TOPLAS,* **8**(4), 547–576 (1986).
16. V. Donzeau-Gouge, G. Huet, G. Kahn and B. Lang, 'Programming Environments based on Structured Editors: the Mentor experience', *Rapports de Recherche 26*, INRIA, Domaine de Voluceau, Rocquencourt, France, 1980.
17. A. Oram and S. Talbot, *Managing Projects with Make*, O'Reilly and Associates, 1991.

* With Eiffel, there are two kinds of dependencies—interface dependencies and implementation dependencies. A client of a class depends on that class's interface, whereas a child of a class depends on that class's implementation. In Barbados, entities depend separately on member functions; and private and public members are lumped together for the purposes of dependencies; so such a distinction is not necessary.

18. P. DuBois, *Software Portability with imake*, O'Reilly and Associates, 1993.
19. W. F. Tichy, 'Smart recompilation', *ACM TOPLAS,* **8**(3) (1986).
20. J. Heering, P. Klint and J. Rekers, 'Lazy and incremental program generation', *ACM TOPLAS,* **16**(3), 1010–1023 (1994).
21. B. Meyer, *Object-oriented Software Construction*, Prentice Hall, Englewood Cliffs, NJ, 1988.