NetBuild: Automated Installation and Use of Network-Accessible Software Libraries †

Keith Moore, Jack Dongarra, Shirley Moore

Innovative Computing Laboratory, University of Tennessee

Eric Grosse

Computing Sciences Research Center, Bell Labs

ABSTRACT

NetBuild is a suite of tools designed to aid users in making use of computational software libraries that are stored on the network, without needing to have those libraries preinstalled on each user's computer. Instead, the NetBuild client determines which libraries are not installed, identifies suitable versions of those libraries that are accessible from the network, downloads those libraries, and links them into the user's program.

This report describes the current status of the NetBuild project, recent progress, and future plans.

1. Introduction and Overview

1.1 Problem Description

NetBuild's goal is to make it easier for people to use high-performance computational software libraries by ridding them of the need to install and maintain current libraries on each computing platform that they use. NetBuild attempts to achieve this by determining which libraries needed to compile or link a program are not installed locally, downloading the missing libraries, and supplying appropriate arguments to the compiler or linker to cause those libraries to be linked in with the user's program.

We hope that this approach will result in less burden on users because they need only install the NetBuild client rather than each of the libraries they want to use. NetBuild can match against fine-grained attributes of a user's computing platform making it possible to choose the most efficient version of a library available for the platform without the user needing to explicitly configure and build that library. NetBuild can also serve as a means to keep libraries up-to-date, as it will always use the most recent version of a particular library that it knows about. Finally it is believed that NetBuild can correctly configure and install libraries more reliably than individual users. In effect, NetBuild leverages expertise of its library maintainers to benefit a potentially much larger number of users.

1.2 How NetBuild Works

The user invokes NetBuild's services via the ‘‘nb’’ program. This program is used to run the compiler, or ‘‘make’’, or compilation script, or any other command that would be used to compile the user's program. So for instance, instead of typing

```
f77 program.f −llapack −lblas
```

the user types

† This material is based on work supported by the National Science Foundation under Grant No. 0130499. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

```
        nb f77 program.f -llapack -lblas
```

Or instead of typing

```
        make program
```

the user types

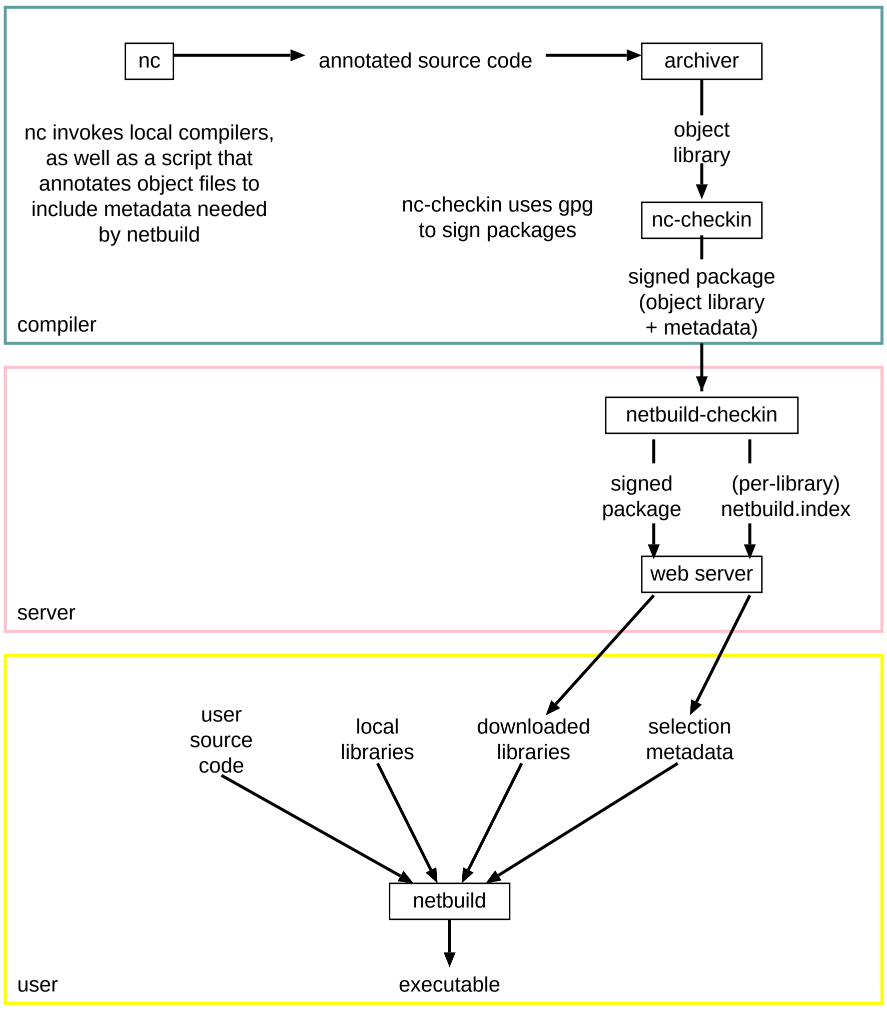
```
        nb make program
```

nb then runs the supplied command in an altered environment, which has its PATH variable modified to have a directory prepended to it. That directory contains ‘‘shims’’ which have the same names as the compilers and linkers which nb needs to intercept.

Whenever one of those compilers or linkers is invoked − either directly from nb, by make, or by some other compilation tool, the shim is run instead of the real compiler. The shim then parses the compiler's arguments looking for names of libraries that need to be linked in. If those libraries are not installed on the system, the shim then downloads them, verifies their signatures, and extracts them into an empty directory. Finally the real compiler or linker is run with a modified argument list that causes the newly-downloaded libraries to be linked in along with the user's program and any native libraries that are used.

1.3 NetBuild Toolset and Structure

This figure illustrates most of the components of the NetBuild system and the relationships between those components:


nc
annotated source code
archiver
object library
nc invokes local compilers, as well as a script that annotates object files to include metadata needed by netbuild
nc-checkin uses gpg to sign packages
nc-checkin
signed package (object library + metadata)
compiler
netbuild-checkin
signed package
(per-library) netbuild.index
web server
server
user source code
local libraries
downloaded libraries
selection metadata
netbuild
executable
user

1.3.1 nc

There are two ways to construct libraries for use with nb − one in which metadata for the libraries are manually supplied by a human expert, and another in which some or all of the metadata are automatically derived when the library is built. nc is the program that is invoked to build a library while automatically supplying the metadata.

Whoever is building the library uses the normal compile command prefixed by ‘‘nc’’. For example, if the compile command is **`make`**, the user would type **`nc make`**. nc intercepts calls to compilers and other tools that are used to compile libraries. It determines what target platform the code is being compiled for and which features of the target platform will be required for the code to run. In some cases it can also determine for what platform variant(s) the code is being optimized. nc then annotates the compiled object files with metadata that can be used in selection of an appropriate library for the user’s platform.

The nc command is actually a small shell script that prepends a directory to the user’s search PATH variable and then calls the command that the user specified. The directory that was added to the PATH contains aliases for compilers that are installed on the system. Each alias is linked to a program called nc-shim which does most of the work.

nc-shim parses the command line arguments of the compiler in order to determine the names of the object files which a successful compile will produce, and also some of the the characteristics of those object files. For instance, on gcc on an IA32 platform the option **`−msse`** tells gcc to optimize for a processor with the SSE extension. Other options that nc-shim needs to know about include those that change the procedure calling mechanism, or the sizes or alignment of data types, used by the generated code. These options need to be reflected in the metadata with which the resulting code is annotated. In order to anticipate the compiler’s actions nc-shim needs to know a fair amount of detail about the compilers installed on a system.

Unless nc-shim thinks it is invoking a cross-compiler, nc-shim also consults native system facilities to obtain metadata for inclusion in the resulting object files. For instance, the operating system, version, and CPU family of the compile platform can be determined from the uname system call.

Once nc-shim has determined what metadata to bundle with the object file, it calls the native compiler to produce the object file, then invokes a separate program (nc-annotate-object) to add metadata to the object file. The implementation of nc-annotate-object varies slightly from one system to another, but in general it first produces a separate object file that contains the metadata as text, and then uses the linker to create a single object file which contains the contents of both the compiled code and the metadata. That resulting object file is renamed to replace the object file that contained compiled code. Within the resulting object file, the metadata is encoded in such a way as to enable it to be extracted from the object file by other tools.

1.3.2 nc-checkin

The purpose of nc-checkin is to accept libraries that consist of object files produced using nc, package them up in a form that can be downloaded by nb, and install them on a server where they can be accessed by nb.

nc-checkin starts by extracting the metadata that are included in the object files produced by nc. This metadata from individual object files are unified to produce a single set of metadata that describes the entire object file. These metadata are then stored in a text file named ‘‘metadata’’. The MD5 hash function [MD5] is then applied to produce hashes for both the metadata file and the library file; these are stored in a file named md5sums. The md5sums file is signed using GNU privacy guard (an encryption and digital signature program) [GPG]. The resulting signature is stored in a file named md5sums.gpg, and the public key of the signer is extracted into pubkey.gpg. The tar program is then used to create a file containing the library, metadata, md5sums, md5sums.gpg, and pubkey.gpg. The result is compressed using gzip to create a package file.

The package file is then transmitted to the NetBuild server using ssh. An MD5 fingerprint of the package file is transmitted along with the package file to serve as an integrity check.

A similar program is used to check in libraries that were not produced by nc, but which have manually-generated metadata.

1.3.3 netbuild-checkin

The netbuild-checkin program resides on the server which makes NetBuild libraries available to NetBuild clients such as nb. It receives a new package from nc-checkin, sanity checks it, stores the package in a location where it can be accessed by a NetBuild client, and updates the metadata that is used by the client to select from among the variants of that package available for that target platform.

netbuild-checkin is invoked via ssh. It is passed the package as standard input, and the filename of the package and the MD5 hash of the package on the command-line. It sanity checks its filename argument (so that netbuild-checkin can't be used to modify files in other portions of the file system), creates any necessary directories, and stores the package in a temporary file. The MD5 hash of the temporary file is then compared with the MD5 hash supplied on the command-line. An different hash value indicates that the library was corrupted in transit, in which case the file is removed. Otherwise the file becomes part of the network-accessible collection.

Each library is stored in a directory (relative to the root of the NetBuild package collection) named *package-name*/*platform-name*, where *package-name* is something like ''lapack'' or ''blas'' and *platform-name* is of the form *cpu-opsys*. (For example: ia32-linux, sparc-solaris, powerpc-aix) A file in that directory named netbuild.index serves as a list of implementations of a particular library. It is built by extracting the metadata from each of the libraries in the directory every time a library is added to or removed from the directory.

1.3.4 nb

The nb program is the primary NetBuild ''client program''. It is the user interface that causes programs to be linked with NetBuild's libraries instead of, or in addition to, the libraries installed on the user's system. Instead of typing **`make`** or whatever command would normally be used to compile a program that used mathematical software libraries, the user merely types **`nb make`**. This causes ''make'' or whatever command to be run in an environment in which linker calls are intercepted by nb. When nb intercepts such a call, any libraries that are not installed locally are downloaded from the network and automatically linked into the user program. When multiple versions of a library are available for the target platform, nb attempts to select the most optimal version available. Also, nb uses digital signatures to ensure that downloaded libraries are not corrupted in transit and that they are not modified by miscreants.

The nb program does two things, depending on how it was invoked:

1. If invoked as `nb`, it prepends a ''shim directory'' to the user's path and interprets the rest of the command-line as a command to be run with the modified path.
2. Otherwise, it assumes it is intercepting a call to a compiler or linker. It parses the command-line options to determine whether an executable file is being produced and whether any libraries are to be linked in. For any libraries that are needed, nb determines whether they are installed locally. If not, nb consults the Netlib repository [Netlib] to determine whether there are downloadable libraries available. If so, nb consults the list of available libraries to determine the best version available for the target platform, selects that one, downloads it, sanity checks it (e.g. for tar pathnames that contain the root directory or '..'), verifies its signature, and if it's valid, calls the linker with appropriate options to cause that library to be linked into the executable.

nb's parser for compiler command lines is driven by configuration files which allow it to support new compilers without recompiling.

nb uses GNU Privacy Guard (GPG) to verify signatures on library packages. It maintains its own GPG keyring (separate from the user's normal GPG keyring) to determine whether to trust a package's signature. At setup time nb is configured to trust any package whose signature has been signed by netbuild-master@netlib.org. In this way the NetBuild maintainers can delegate authority to sign code to a small number of trustworthy individuals, without compromising the netbuild-master key. However a user can change the trust settings by using GPG to manipulate the trust parameters on nb's keyring.

NetBuild does not currently have a way to revoke signatures. However libraries deemed suspect or untrustworthy can be removed from the list of libraries made available to nb, and this list is checked each time nb is run.

A library with a valid signature and signed by a trusted key is intended to signify three things:

- ‘‘This library has not been corrupted in transmission.’’
- ‘‘NetBuild developers and library maintainers do not have reason to believe that this library poses a security risk to your computing environment or data’’, and
- ‘‘To the best of the knowledge of the NetBuild developers and library maintainers, this library produces correct output when given well-formed input, according to the specifications for the library.’’

On the other hand, the signature **does not indicate any warranty** on the part of NetBuild developers or library maintainers that the library will produce correct output given well-formed input, nor that the library is free from security risks to users’ computing environments or data. While we believe that having signed libraries does ameliorate some of the security risks associated with using downloaded code, it clearly does not eliminate all security risks.

nb’s library selection currently works as follows: it scans the current list of candidate libraries, eliminating those that are ineligible. Currently ’ineligible’ means that the library was compiled for a different CPU type or operating system, or that the OS version on which the library was compiled is more recent than the OS version of the target platform, or that the library has a target.constraint expression that is incompatible with the target platform. Of the remaining candidates the one with the highest valued library.precedence expression is chosen. If there is more than one of these, the candidate with the recent OS version is chosen. (There is a special comparison function for OS version numbers which treats each facet of the version number as an integer. So 3.0 is greater than 2.10, 2.10 is greater than 2.1, 2.10.1 is greater than 2.10. On some platforms it is necessary for nc to alter the syntax of the version number supplied by the operating system so that such comparisons will work.)

2. Recent Changes: New Features, Problems Solved, Lessons Learned

2.1 Changes to nb

- The name of the NetBuild client program was changed from ‘‘netbuild’’ to ‘‘nb’’ as it was confusing to have the same name for the client program and the project.
- Support for constraint expressions and precedence expressions. nb now selects from a set of libraries based on constraints and precedence expressions. Constraint expressions dictate whether a particular library is eligible to be used on a particular target platform. For example, the following expression matches an AMD Athlon with an L2 cache size of 256K or greater and an L1 cache size of 64K or greater

  ```
  cpu.ia32.vendor=="AuthenticAMD" && cpu.ia32.model>=4 &&
  cpu.ia32.l2.dcache.size>=256*1024 && cpu.ia32.l1.dcache.size>=64*1024
  ```

  Precedence expressions define an ordering among the set of eligible libraries according to estimated performance. Most precedence expressions are simple integers; however the same expression evaluator used for constraint expressions is also used for precedence expressions, making complex precedence relationships possible.
- Support for fine-grained selection criteria on IA32. On IA32 platforms constraint and precedence expressions can reference fine-grained characteristics of the target platform − including any parameter available through the CPUID instruction. [IA32]
- A nb installation may now be shared between all users on a system. Each user is required to run **`nb −setup`** before using nb for the first time. This creates the necessary per-user directories and copies default configuration files into those directories. Previous versions of nb had locations of configuration files determined at compile time, which caused problems when multiple users tried to share GPG trust parameters or cached libraries.
- nb can now support packages that use threads and multiple CPUs (on Linux, NetBSD, and MacOS X)

2.2 Changes to the Package Format

- The package format now supports GPG-signed libraries. By default, any package that is signed by the netbuild-master key, or which is signed by a key that is trusted by netbuild-master, will be trusted. Users can change this to trust whomever they choose.
- Support for post-install shell scripts has also been added. Such a script, if present, is executed once the package has been downloaded, verified, and extracted. This provides a great deal of flexibility. For instance, since redistribution of Goto BLAS [GotoBLAS] files is discouraged by that library's author, the NetBuild Goto BLAS package uses the post-install script to download the actual libraries from the official web site, and to verify those libraries against predetermined MD5 hashes for integrity checking.
- Packages can now explicitly specify args to pass to the linker, rather than nb assuming that link argument is derived from the package name. This allows nb to provide a more uniform interface between packages. For instance, both the ATLAS and the Goto BLAS packages require multiple libraries to be linked. Also, on some systems the NetBuild BLAS package might recognize the existence of a vendor-supplied BLAS library with a different name, and use this if it is present. So for example on a Sparc Solaris system the user might type **`-lblas`** but a NetBuild BLAS package could recognize the presence of an equivalent vendor-supplied ''sunperf'' library using the constraint expression **`have_file(/opt/SUNWspro/lib/libsunperf.so)`**. If this package were selected it would cause the linker to use the local vendor-supplied BLAS library using the argument **`-xlic_lib=sunperf`**.

2.3 Changes to Package Building Tools

- We have developed tools and methodology for building library packages for each of multiple platform variants from the same source (without modifying the source for each variant). Currently the build tools accept a list of platform variant descriptions along with instructions for taking a pristine copy of the library source code and configuring and/or modifying that code so that when compiled it will produce a library that is suitable for a particular platform variant.
- We have developed tools for maintenance of the NetBuild package server, which support library checkin and replacement, and automatic maintenance of index files (both machine and human-readable).

2.4 Changes to Libraries

More libraries are now supported, including ARPACK [ARPACK], BLAS [BLAS1..BLAS5], EISPACK [EISPACK], LAPACK [LAPACK], LINPACK [LINPACK], NetCDF [NetCDF], and SuperLU [SuperLU].

NetBuild's BLAS package on IA32 Linux platforms is an attempt to supply the ''best of breed'' version of BLAS for each platform variant. A total of thirteen different libraries are currently used. Since the Goto implementations of BLAS seem to be the fastest on the platforms for which these are available, NetBuild uses the Goto BLAS if there is an implementation that matches the target platform. An ATLAS BLAS [ATLAS] is used for the Linux Athlon processor as there is no Goto BLAS for this processor. (Additional ATLAS-based BLAS implementations could be supplied for other platforms if we had an example of that platform on which to compile the library.) Other IA32 processors use a GEMM-based BLAS [GEMMBLAS]; there are different versions of this which are compiled for different versions of the IA32 instruction set.

3. Issues

3.1 Difficulty of Supporting Fine-grained Target Attributes

In order to achieve near-peak performance with some compute-intensive libraries it is necessary to select a library according to fine-grained attributes of the target platform such as the sizes of various caches, which can vary considerably from one processor to another of the same type. A library that is selected on the basis of incomplete target information can be several times slower than one which is optimal for the target platform.

On IA32 platforms the (unprivileged) CPUID instruction provides a great deal of detail about the precise model of the processor, set of features supported, cache sizes, etc. On most other platforms less information is available, and such information as is available usually requires the ability to execute privileged instructions.

Possible workarounds include:

- On some platforms detailed information is available but only in a format which is specific to the operating system and which may not generalize to use of the same processor on other operating systems.
- Mapping of limited information (e.g. CPU serial number) which is available via operating system services, onto more detailed information about processor characteristics.
- Empirical determination of some processor characteristics at link time, or when nb is installed.
- Link-time performance measurement of each of several alternative libraries.
- Use of user-supplied configuration information, as users may know the characteristics of their computing platforms via means unaccessible to the nb program.

3.2 Difficulties With Maintaining Large Collections of Libraries

In order for NetBuild to be viable it must be able to provide access to most of the libraries that are needed by its user community. This further implies that, at least initially, NetBuild's developers must provide a large collection of libraries that are pre-compiled for the target platforms that are supported.

This is difficult for several reasons:

- Some libraries need to be built for a large number of platform variants. For libraries that empirically determine compilation parameters to maximize performance (e.g. ATLAS) it is necessary to build each variant library on the specific target platform for which it is intended. This in turn requires that the library be built separately on each of several slightly different compilation platforms. While there is no intention to provide optimal implementations of libraries for truly obsolete hardware, there is still considerable variation among the hardware that in widespread use and which provides adequate performance for computationally intensive tasks.
- Some libraries have sophisticated Makefile builders or compilation programs which automatically select specific optimizations and compiler flags. Many of these Makefile builders or compilation programs assume that the library is being built on the exact platform on which it will be used. If the library is somewhat dated, or if it is being ported to a platform or platform variant not originally supported by the author, these optimizations and flags may be anachronistic or inappropriate for a particular target platform. In these cases it is necessary to alter the compilation program to support new target platforms and platform variants, and to test the resulting libraries for correct operation and robustness. It is also desirable that any changes be made in a way that doesn't alter the source code of the library itself (so that a single code base may be used to support all NetBuild platforms and platform variants) but this makes such testing and debugging more time-consuming.
- Some libraries (e.g. SuperLU) require hand-tuning of various compiled-in parameters in order to provide good performance. A library which is hand-tuned for one platform variant may not perform acceptably on a different variant of the same platform.
- Even if one variant library will work satisfactorily on several platforms, it can be difficult to predict exactly which variant will work best on a particular platform without testing each variant on that platform, and even then the performance can vary a bit according to the actual conditions in which the library is used.
- Compiler bugs may cause a library compiled for one platform variant to fail even though the same compiler produces correct code for other platform variants. Compiler bugs have also been observed that produce suboptimal code on some platform variants while producing acceptable code on others. Multiple compilers, or multiple versions of a compiler, may therefore be needed to produce all of the variant libraries required for a particular platform.

- Many libraries lack validation routines that would permit us to verify that they operate correctly. When libraries do have validation routines it may still be necessary for a human domain expert to examine their output before it can be determined that the library is operating correctly.
- Currently the nc-checkin routines require that each library be signed and checked in separately. This is tedious as it is occasionally necessary to rebuild large numbers of libraries, for instance, to accommodate a new platform variant or a new operating system version.

3.3 Difficulties With Multiple Implementations of the Same Library

Some libraries (e.g. LAPACK, BLAS) have standardized APIs that are more-or-less shared across different implementations. It is often desirable for NetBuild to be able to select between different implementations of the same library API. For example, in the case where a vendor's BLAS outperforms a freely-available one, but the vendor's BLAS is not freely available, you want nb to use the vendor BLAS if it is installed, otherwise to use a free one.

However there are nearly always some differences between the implementations. For instance, the Goto BLAS (which is supplied in binary form only) also implements some routines from LAPACK, which can potentially produce symbol conflicts if the library is linked with an object file that includes its own implementation of one of those routines. Similarly, if a program was originally written and tested with one BLAS, it might reference routines that are not present in a different BLAS.

3.4 Resolution of Conflicting Selection Criteria

The introduction of dependencies in NetBuild libraries will introduce the possibility that selection of one library required to build a program will preclude selection of a different library that is also required to build that program. The implication is that it will no longer be sufficient to select each library separately − NetBuild must generate sets of libraries which are mutually consistent and then select the best of the available sets.

3.5 Licensing

Many software packages have licenses that impose (to NetBuild) arbitrary restrictions on use, or require explicit consent to terms by users, even if monetary compensation is not required. Such licenses are impediments to NetBuild's goals of providing transparent access to those libraries.

One possible workaround is to allow users to examine each of several license agreements at the time NetBuild is installed, or at other times (say, by visiting a web page), and to declare their assent (or objection) to some, none, or all of those agreements. NetBuild can then record that assent or objection (e.g. ‘‘user XXX has consented to agreement with MD5 hash YYY’’) and use this information as input to the library selection process. This would permit NetBuild to use software whose licenses permit unfettered use once conditions are agreed to, but would not permit NetBuild to support licenses imposing time, place, or manner restrictions on use.

3.6 Incompatibilities Between Different Compilers on the Same Platform

In some cases different compilers on the same platform may produce libraries that cannot be used with other compilers on that platform, or that may not be mixed with other libraries compiled with other compilers, or that require different run-time support than libraries compiled with other compilers. In some cases the same compiler called with different options (say, to change the sizes of fundamental data types or the convention by which functions are called) can have the same effect. Sometimes the differences are subtle, as in FORTRAN routines that only introduce such dependencies when they do formatted input or output (because they call different run time libraries to implement formatted i/o). Similar issues result from mixing routines written in multiple languages.

It is sometimes, but not always, feasible for NetBuild to determine at link time whether object files and libraries are compatible. For instance, if object files and libraries are linked using a compiler, the name of that compiler can provide a clue as to the API expected by object files that it generates. However if the object files and libraries are linked by calling the linker, this clue is not available. Metadata can be embedded in objects compiled using nb and libraries compiled using nc, and nb may be able to use this metadata to determine whether or not libraries and objects are mutually compatible. However, it seems unrealistic to expect that all objects and libraries be compiled using these tools. It also seems unrealistic to expect nb and nc to reliably determine exactly what constraints are imposed. Should nc check to see whether a FORTRAN routine uses formatted input/output if these are the only constraints on the ABI? If a C routine is specifically written to be called from a particular FORTRAN compiler, with its function names and parameters arranged accordingly, how can nc automatically determine this without requiring changes to the library source code?

3.7 User-defined Trust Parameters

By default nb trusts any library that is signed by the netbuild-master key or by a key that was certified by the netbuild-master key. This essentially limits use of nb to libraries supplied by the NetBuild maintainers. It is desirable to allow users to invest trust in other library sources. This is possible in the current version of nb but it requires users to understand subtleties of GPG configuration. Creating a user interface to trust parameters which is both easy to understand and has predictable behavior is a challenge.

4. Future Plans

Our intention is to release the NetBuild 1.0 client in March 2004. Before we do this we need to:

- Add metadata support for representing compiler dependencies and data-format dependencies,
- Add support for software licensing (at least for ‘‘freeware’’ licenses),
- Rationalize metadata parameter names. This in turn will require rebuilding all libraries.

The initial client release may have only a few libraries that are supported on only a few platforms, but we expect to be able to add many more libraries without changing the client code. Release of other tools, including ‘‘nc’’ and checkin tools that would allow users to build their own libraries, will be delayed until we have more experience with building libraries with those tools.

Future NetBuild releases will support the ability to compile libraries from source code if no suitable pre-compiled libraries are available. Consideration is being given to allow the NetBuild client to support other package formats (e.g. pacman, rpm). Fine-grained target attributes will be supported on as many platforms as possible.

5. Conclusions

By far the most difficult and time-consuming aspect of NetBuild development has been creating and maintaining a current and comprehensive set of libraries. This is surprising given that we are not attempting to fix bugs in the libraries themselves, but only to configure and compile libraries on a large number of target platforms. More work is needed to understand how to streamline this process. It is hoped that this will result in improved tools for automatically (re)building libraries and managing the complexity associated with supporting multiple platforms.

In particular the approach taken by ‘‘nc’’ has not been as effective as anticipated, because platform-specific optimizations are often performed by conditional compilation or by generating code to fit the target platform, rather than by merely specifying platform-specific compiler optimization. It is therefore necessary to specify the metadata for many libraries ‘‘by hand’’. Work continues on tools to ease the generation of libraries for multiple platform variants.

netbuild-master@netlib.org key fingerprint

The fingerprint for the netbuild-master@netlib.org GPG key is:

```
7313 ABD7 6CA2 3572 C231  A39B 54C9 E5BC 28EE ABDC
```

References

[ARPACK]
R. B. Lehoucq, D. C. Sorensen, and C. Yang. *ARPACK Users’ Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*. SIAM, 1998. (also available at

`http://www.caam.rice.edu/software/ARPACK/UG/ug.html`)

[ATLAS]
R. Clint Whaley, A. Petitet, J. J. Dongarra. ‘‘Automated Empirical Optimization of Software and the ATLAS Project’’. *Parallel Computing*. 27 1−2, pp 3-35. (2001) (Also available as University of Tennessee LAPACK Working Note #147, UT−CS−00−448, 2000.
`http://www.netlib.org/lapack/lawns/lawn147.ps`).

[BLAS1]
C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh. ‘‘Basic Linear Algebra Subprograms for FORTRAN usage’’. *ACM Trans. Math. Soft.*, 5 (1979), pp 308−323.

[BLAS2]
J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. ‘‘An extended set of FORTRAN Basic Linear Algebra Subprograms’’ *ACM Trans. Math. Soft.* 14 (1988), pp 1−17.

[BLAS3]
J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. ‘‘Algorithm 656: An extended set of FORTRAN Basic Linear Algebra Subprograms’’, *ACM Trans. Math. Soft.*, 14 (1988), pp 18−32.

[BLAS4]
J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. ‘‘A set of Level 3 Basic Linear Algebra Subprograms’’. *ACM Trans. Math. Soft.*, 16 (1990), pp 1−17.

[BLAS5]
J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling, ‘‘Algorithm 679: A set of Level 3 Basic Linear Algebra Subprograms’’, *ACM Trans. Math. Soft.*, 16 (1990), pp 18−28.

[EISPACK1]
B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, C. B. Moler. ‘‘Matrix Eigensystem Routines − EISPACK Guide’’. in G. Goos and J. Hartmanis, eds. *Lecture Notes in Computer Science* 6. Springer-Verlag, 1976.

[EISPACK2]
B. S. Garbow, J. M. Boyle, J. J. Dongarra, C. B. Moler. ‘‘Matrix Eigensystem Routines − EISPACK Guide Extension’’ in G. Goos and J. Hartmanis, eds. *Lecture Notes in Computer Science* 51. Springer-Verlag, 1977.

[GEMMBLAS]
B. Kagstrom, P. Ling, and C. Van Loan. ‘‘Level 3 BLAS tuned for single processors with caches’’ *High Performance Computing* II North-Holland, 1991.

[GotoBLAS]
K. Goto and R. van de Geijn. ‘‘On Reducing TLB Misses in Matrix Multiplication’’. FLAME Working Note #9, The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-2002-55. Nov., 2002. (Also available at
`http://www.cs.utexas.edu/users/flame/pubs/FLAWN9.ps.gz`)

[GPG]
J. M. Ashley. *The GNU Privacy Handbook*. 1999.
`http://www.gnupg.org/gph/en/manual.html`

[IA32]
Intel Corporation. *IA-32 Intel Architecture Software Developer’s Manual*. 1, 2A, 2B, 2003.
`http://developer.intel.com/design/pentium4/manuals/25366513.pdf`
`http://developer.intel.com/design/pentium4/manuals/25366613.pdf`
`http://developer.intel.com/design/pentium4/manuals/25366713.pdf`

[LAPACK]
E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen. *LAPACK Users’ Guide, Third Edition*. SIAM, 1999.

[LINPACK]
J. Dongarra, C. B. Moler, J. R. Bunch and G. W. Stewart. *LINPACK Users’ Guide*. SIAM, 1979.

[MD5]
R. Rivest. ‘‘The MD5 Message-Digest Algorithm.’’ RFC 1321, April 1992.
`ftp://ftp.rfc-editor.org/in-notes/rfc1321.txt`

[NetCDF]
R. Rew, G. Davis, S. Emmerson, and H. Davies. ‘‘NetCDF User’s Guide for FORTRAN’’. June 1997.
`http://www.unidata.ucar.edu/packages/netcdf/guidef/`

[Netlib]
Netlib software repository. `http://www.netlib.org/`

[SuperLU]
J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. Liu. ‘‘A Supernodal Approach to Sparse Partial Pivoting’’. *SIAM Journal on Matrix Analysis and Applications*, (20) 3, pp 720−755. 1999.
`http://http.cs.berkeley.edu/˜xiaoye/simax95.ps.gz`