Trustless Grid Computing in ConCert *

Bor-Yuh Evan Chang, Karl Crary, Margaret DeLap, Robert Harper, Jason Liszka, Tom Murphy VII, and Frank Pfenning

Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA, USA
{bechang,crary,mid,rwh,jliszka,tom7,fp}@cs.cmu.edu

Abstract. We believe that fundamental to the establishment of a grid computing framework where all (not just large organizations) are able to effectively tap into the resources available on the global network is the establishment of trust between grid application developers and resource donors. Resource donors must be able to trust that their security, safety, and privacy policies will be respected by programs that use their systems.

In this paper, we present a novel solution based on the notion of *certified code* that upholds safety, security, and privacy policies by examining *intrinsic* properties of code. Certified code complements authentication and provides a foundation for a safe, secure, and efficient framework that executes native code. We describe the implementation of such a framework known as the ConCert software.

1 Introduction

In recent years, numerous organizations have been vying for donated resources for their grid applications. Potential resource donors are inundated with worthwhile grid projects such as discovering a cure for AIDS, finding large prime numbers, and searching for extraterrestrial intelligence. Part of the difficulty in obtaining resources is establishing trust between the grid application developer and the resource donors. Because resource donors often receive little or no direct reward for their contributions, they demand assurances of safety, security, and privacy to protect themselves from malicious as well as simply unreliable software. In an ideal grid framework, as proposed in Legion [18], users are provided the abstraction of a single virtual machine that automatically distributes work and gathers results. In such a framework, this issue is even more salient because the exchange of code happens automatically.

Most current grid frameworks, such as the Globus toolkit [13], focus on *authentication* to provide the basis for security on computational grids. Authentication provides a means for one entity to verify the identity of another. If this is combined with some form of *access control*, then resource donors are able to control *who* can access their resources. In essence, this seeks to address the

* The ConCert Project is supported by the National Science Foundation under grant ITR/SY+SI 0121633: "Language Technology for Trustless Software Dissemination".

question “how do I identify those I trust?”; however, there seems to be a more fundamental question: “whom can I trust?” Currently, resource donors are limited to relying on the reputation of the developer or possibly the reputation of some quality assurance team that endorses him. While this is a reasonable solution for large well-known projects, we would like a more automated means that would enable more people to utilize the grid.

To address this issue, the ConCert project [9] enforces safety, security, and privacy policies by verifying *intrinsic* properties of code. This is realized through the use of *certifying compilers* that allow software developers to produce efficient machine code along with checkable certificates that can be easily and automatically verified on behalf of code recipient. Our vision is to empower *all* programmers to utilize donated resources by establishing trust via rigorous, mathematical proof of the intrinsic safety, security, and privacy properties of software.

Our report proceeds as follows. We compare several existing techniques for code certification, briefly introduce the specific technologies on which our framework is built, then explain the properties that can be certified. We then discuss the design and implementation of our peer-to-peer architecture, and finally describe our first application, a grid ray tracer.

2 Certification and Security

In this section, we discuss current means of ensuring safety and security without authentication, provide an overview of the certified code technologies we employ, and discuss properties that can be certified.

2.1 Related Safety and Security Ideas

Mechanisms for safeguarding against faulty code produced by trusted users have been needed in systems long before the concern over malicious mobile code. For example, one method to safeguard memory between threads of computation is to simply place the processes in separate address spaces, as is done in most modern-day operating systems. This and other operating system mechanisms provide a very coarse level of safety or fault isolation both for individual systems and any grid framework. The problem with basing safety solely on these coarse mechanisms is that there is very little control on what properties are enforced and there is often a significant runtime overhead associated with these mechanisms (*e.g.* frequent context switching and copying between addresses spaces). Several technologies have been developed to lower the cost of fault isolation and provide finer control over which safety, security, and privacy properties are ensured. This, in essence, is also the goal of certified code. In this section, we describe a few of these related technologies, namely virtual machine techniques and methods that interpose between the process and the operating system, along with some shortcomings that we believe certified code addresses.

Virtual Machine Methods. The most prominent use of a virtual machine environment for providing a secure execution space is in the Java Development Kit (JDK). At the core of the JDK 1.2 security model [15] is the ability to type-check Java Virtual Machine (JVM) bytecode [21]. It is critical that untrusted code cannot bypass the runtime checks that enforce the high-level security policy. In addition, the JDK has authentication-based mechanisms for identifying the origin of code and restricting access based on this information.

Several grid frameworks have been developed using Java as a secure host for mobile code, such as IceT [17], Javelin [5], and Bayanihan [33]. The IceT project aims at allowing processes running in the IceT environment to freely flow among available sites. IceT code primarily consists of Java *application* bytecode that is dynamically loaded into a governing environment with some optional policy-based means to link with native code. Javelin focuses on creating a grid framework using Java-enabled web browsers to minimize the technical expertise needed to participate in grid computing. Code for Javelin must be in the form of Java *applets* rather than Java applications. Project Bayanihan also aims to utilize Java applets to make it easier to participate but also supports the loading of Java applications. Further, it investigates preventing *sabotage* by nodes who submit erroneous results.

At the same time, despite the additional safety and security guarantees afforded by a virtual machine environment for loading code, a number of grid frameworks load native code for performance reasons. Although concerns about the performance of interpreted Java bytecode are somewhat relieved by just-in-time (JIT) compilers, JIT compilation to native code occurs *after* bytecode verification, so errors in the JIT compiler may lead to security holes. In addition, since the JIT compiler is run at execution time, its compilation process must be fast, which limits the quality of code it is able to produce.

Interposition Methods. Even prior to grid computing, interposition mechanisms between an untrusted process and the operating system were used to provide finer control of safety properties. Interposition gives control of execution to a watchdog whenever the untrusted process performs a possibly unsafe operation. One interposition method is known as Software-based Fault Isolation (SFI) [35], which is exhibited in the Omniware system for mobile code [22]. One striking issue with interposition mechanisms is that they are limited in what properties they ensure. For example, SFI only ensures memory safety. As with virtual machine methods, performance of interposition methods is often also a concern. In grid applications where we seek to make the best use of donated resources, avoiding the overhead of these methods is desirable.

Proof-Carrying Code. *Proof-carrying code* (PCC) is a form of certified code with arbitrary certification properties written as logic statements [28]. In PCC, a *certifying compiler* generates a proof along with the object code, and this proof is verified by a simple proof-checker on the code recipient's computer. PCC has very little runtime overhead because many properties can be verified before the program is ever run. For instance, Necula and Lee show that PCC handily outperforms SFI on a network packet filter application [29].

Because the certified properties are, in principle, arbitrary, PCC is highly flexible. However, this flexibility also means that it can be difficult for programmers to know when the certifying compiler will be able to generate proofs. One response to this problem is to encode desired safety properties in a *type system* for the source programming language. In this sense, a type system is simply a syntactic realization of specific properties such that a well-formed program is guaranteed to have those properties. Verification then consists of type-checking rather than proof-checking. A well-designed type system makes it easy for the programmer to understand which properties are certifiable by a compiler.

Many properties can be encoded in a type system; however, all type systems rely on the basic notion of *type safety*. Type safety means that the program will not "go wrong" by violating abstraction boundaries, accessing memory outside its address space, or branching into unsafe code. Therefore, for our first version of a grid framework with certified code, we choose languages with just this basic property.

2.2 Enabling Technologies

In this section, we provide some background on the enabling technologies that allow us to develop a grid framework based on the notion of certified code. The implementation of our grid framework builds on the TALx86 [24] realization of the Typed Assembly Language (TAL) [26, 25] developed by Morrisett *et al.* A certifying compiler for a type-safe C-like language called Popcorn that compiles to TALx86 has also been developed as part of the TAL project. These tools serve as a foundation for an implementation of the ConCert grid software and applications that run on it.

An Overview of Popcorn. Popcorn is a programming language similar to C, except that unsafe features like the unrestricted address-of operator, pointer arithmetic, and pointer casts have been left out. At the same time, it has a number of advanced features akin to modern high-level programming languages like Java [16] and Standard ML (SML) [23], such as exceptions, garbage collection, tagged unions, and parametric polymorphism that mitigate the need for these unsafe features.

The primitive types supported by Popcorn include `bool`, `char`, `short`, `int`, `float`, `double`, `string`, and `unsigned` forms of the numeric types. Unlike C, arrays carry their size for bounds checking, and strings are not null-terminated. Instead, strings are treated like arrays of `chars`, and a special `size` construct is used to extract the size of an array or string. Popcorn's basic control flow constructs (`if`, `for`, `while`, `do`, `break`, and `continue`) behave identically to their C counterparts except that test expressions must have type `bool`.

The aggregate data structures in Popcorn are similar to ones in high-level languages like Java and SML. First, Popcorn supports tuple types in addition to structs and unions. Tuples (as well as structs and unions) are created using the new construct and projected using .1, .2, ... as shown in Fig. 1A. There are two forms of structure definitions: struct and ?struct. Values of types defined with struct cannot be null (a primitive in Popcorn) whereas values of types defined with ?struct may. A value of a type defined with ?struct is checked for null upon access to a field. If it is null, the program aborts immediately. Unions in Popcorn resemble SML datatypes more than C unions in that each variant has a tag and an associated type. In Fig. 1B, we declare a full binary tree with integer data at the leaves.

```
(A)     *(int,int) p    = new (0,1);
        int        sum = p.1 + p.2;
(B)     union tree { int Leaf; *(tree,tree) Node; }
(C)     int numleaves(tree t) {
          switch (t) {
            case Leaf(x):      return 1;
            case Node *(l,r):  return numleaves(l) + numleaves(r);
          }
        }
(D)     *(t2,t1) swap<t1,t2>(*(t1,t2) x) {
          return new (x.2,x.1);
        }
```

Fig. 1. Popcorn examples

Notice that union types may be recursive. We often utilize values of a type defined with union by switching on them. For example, we can write a function that counts the number of leaves in a binary tree (Fig. 1C).

Parametric polymorphism provides a means to write data structures or algorithms where the use of values is independent of the types. The syntax of parametric polymorphism in Popcorn resembles that of templates in C++ or generic types in GJ [4]. In Fig. 1D, we use parametric polymorphism to write a generic swap of the components of a pair. The symbols t1 and t2 are type variables that represent arbitrary types.

From this overview, we see the similarity of Popcorn to Java in that it has the look and feel of C but with strictures that enable the demonstration of safety properties. However, Popcorn differs from Java in that the output of the Popcorn compiler is machine code able to be run at full speed rather than bytecode that is interpreted or that requires just-in-time compilation upon execution.

An Overview of TALx86. TALx86 is a statically-typed assembly language for the Intel IA-32 architecture. Since it is not feasible to fully describe TALx86 here, we simply provide an overview of TALx86 in order to appreciate our grid framework's underlying technology. Further details about TALx86 can be found in Morrisett *et al.* [24] as well as about its theoretical basis in related works [26, 25].

From the high-level source language, the Popcorn compiler (or potentially other certifying compilers) produce .tal files that contain TALx86 assembly with all the typing annotations for each source file along with a number of other files

Table 1. TALx86/Popcorn Files for `main`

File	*Produced By*	*Description*	*Shipped*
`main.pop`	developer	the Popcorn source file for `main`	
`main.tal`	`popcorn`	typed assembly language output	
`main_i.tali`	`talc`	`main`'s imports (any `extern` declarations)	√
`main_e.tali`	`talc`	`main`'s exports (non-static types and values)	√
`main.to`	`talc`	binary file with the typing annotations	√
`main.o`	`talc`	native object file in ELF	√

that are generated by the TALx86 assembler `talc`. Some of these files are shipped to the code recipient. These contain information such as the imports, exports, and typing annotations that are used for link-time verification. A `.tal` file is a realistic assembly language that is, in fact, compatible with the Microsoft Macro Assembler (MASM). Table 1 summarizes these files for an example program called `main` that is written in Popcorn.

The TALx86 assembler `talc` translates a `.tal` file into a native object file in either COFF (for Windows) or ELF (for Unix variants) and a `.to` file that contains typing annotations from the original `.tal` file in a binary format. The `talc` assembler is actually composed of a TALx86 type-checker, a link-verifier, a code assembler, and a code linker. Upon reading of `.tal` files, `talc` first type-checks each file individually. This type-check should never fail provided that the implementation of the certifying compiler is correct; however, by type-checking, we no longer need to assume the correctness of the certifying compiler. Contrast this with using the JVM with just-in-time compilation for ensuring type-safety where we have to assume that the JIT compiler is implemented correctly. Before creating the native object file, the link-verifier ensures that the multiple `.tal` files are safe after being linked together by verifying that they have the same assumptions about the values and types that they share. Technical details about link verification can be found in Glew and Morrisett [14].

The ConCert grid software simply ships the native object, `.to`, `_i.tali`, and `_e.tali` files. Before dynamically loading the code on the donor host, the grid software type-checks the code using a TAL verification library on the received files.

Certifiable Properties. Type safety is the key property that allows us to control how a program behaves. As a baseline, a type-safe program must be memory safe (no illegal reads or writes) and control-flow safe (no jumps to illegal addresses). In addition, it cannot violate abstraction boundaries that could lead to such errors. We are further able to provide the distributed code with an arbitrary "safe" subset of the system library. It is simple, for instance, to allow the code to modify or delete only files that it created (without run-time checks), or to simply not allow access to any files at all. The grid volunteer can be given a choice of several policies on these sorts of issues, each of which is enforced by verifying that the candidate code type-checks with the supplied view of the system library.

There are, however, other properties that we may wish to certify that cannot be enforced this way. These properties are a subject of current research. To express these, we enhance our type systems so that type safety implies adherence to the property in question. For instance, one primary concern for grid computing is the resource usage of the distributed code. Much work is in progress regarding *resource bound certification* [30, 10], which allows the type system to include bounds on how much CPU, memory, and disk resources a piece of code may use.

Finally, users may wish to give code access to private information (such as the computer's configuration), so long as this information does not, for instance, make its way back onto the network. Such requirements are known as *information-flow* properties, and work is being carried out on type systems that certify them [19, 36, 27].

3 ConCert Architecture

A chief property of our architecture is decentralization. A decentralized grid allows us to use idle resources while avoiding bottlenecks that could overload hosts or networks and degrade performance—possibly even for users who are not involved in grid computation. Another driving principle is fault tolerance; we expect nodes in the network to fail frequently, and this has a significant impact on our programming and scheduling model.

Our current design is entirely peer-to-peer and symmetric (each node both serves code and runs it). Every participating node has three components: a *locator*, a *worker*, and a *conductor*. The *locator* finds a host's peers at runtime as is necessitated by the decentralized architecture. The *conductor* component keeps track of what work is available to be done and bundles files as necessary to provide work to peers. The *worker* acquires code from a peer (perhaps itself), verifies it using the TAL verifier, and runs it. We now provide an overview of each of these components. Further details can be found in DeLap's honors thesis [12].

Node Discovery. Because of the system is decentralized, participants should find their peers dynamically. In our current implementation of the *locator* component, we use a protocol similar to Gnutella's [8] to discover nodes. Using this protocol, hosts build tables of peers as they run. A host that wishes to participate joins the network by sending an initial ping message to some number of predetermined peers. If any of these peers are participating, they will forward that message to the peers of which they are aware, and so forth within a limited number of hops. Anyone receiving a ping message notifies the originating host that it is alive by sending a response.

We do not share all of Gnutella's privacy goals, so we are able to reduce network traffic for certain exchanges. Nonetheless, since it is known that Gnutella may have scalability issues [31], we may wish to change our method of node discovery in the future. Components other than the locator remain unaware of how the list of contacts is found. Therefore, it would be feasible for us to change our entire network topology without affecting the serving and running of work itself.

Parallelism Model. Since the grid is likely to contain slow, unreliable network links and hosts, we do not support the use of either fine-grained, high-communication threads or shared memory, both for performance reasons and for failure tolerance. At the same time, our model of parallelism should allow for inexpensive scheduling. We therefore base our model of parallelism on that of Cilk-NOW [3] and, more generally, dataflow computer architectures [11]. In our model, programs are split into segments called *cords*, on which we impose certain invariants to simplify scheduling tasks.[1]

First, once a cord is ready to run, it is able to execute continuously to completion without waiting for data from other cords or otherwise blocking. In other words, a cord does not communicate with other cords while executing. This restriction simplifies scheduling greatly.

Second, any execution of a cord is, as far as the developer is concerned, "as good as" any other. This is evident for deterministic cords. Nondeterministic or randomized algorithms may also be acceptable, as long as a re-execution (due to the failure of a node) still produces a valid result.

Finally, cords do not produce outside effects that other cords rely on. For example, it would be unsound for one cord to write to a file on a participating machine and another cord to depend upon the contents of that file. Such effects are forms of "out-of-band" communication with other cords. As such, they would be hidden from the scheduler; if any cords were later rescheduled, they would almost certainly not behave correctly.

How, then, does information travel between the parts of a program? Each cord's I/O consists of the entire set of *arguments* it needs and the *result* it produces. We may therefore represent a program as a graph in which nodes denote cords and edges indicate the data flowing into them as arguments. A cord is not ready to run until its inward edges have the necessary data available. Fig. 2(A) gives a simple example of this model. This form of "communication" may seem rudimentary, but by creating new cords at run-time with appropriate dependencies we are able to implement more sophisticated forms of control flow. For instance, with a process similar to Continuation-Passing Style [1], we are able to implement fork-join parallelism.

To make our model more powerful, we collect sets of dependencies into groups of *and*- and *or*-dependencies. In an *and*-dependency set, all of the dependencies are required—they must all contain data before the set of dependencies is considered complete. Alternatively, in an *or*-dependency set, only one dependency result is necessary. As soon as any one of the dependencies is filled, that set is considered completed. Such sets can be chained into trees of dependencies that are to be simplified and collapsed as dependency data (results) are filled in. Fig. 2(B) provides a simple example of a dependency tree.

[1] Adherence to these invariants is not presently certified, however, programs that do not meet them do not pose any danger to the resource donor; the programs simply do not work properly. It would be desirable that a grid programming language assist the programmer in verifying these properties.

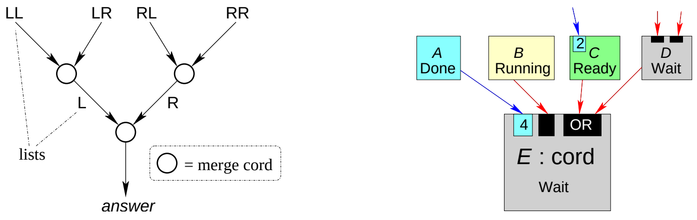

Fig. 2. (A) A simple cord graph example. This graph represents an execution of a program that mergesorts four sorted list segments into one sorted list. The list segments, which are themselves results of cords, are arguments to the cords. Merged list segments travel downward in the graph until the final result (the entire sorted list) is obtained. (B) Dependency resolution. Here, cord A has recently terminated; its result, 4, becomes one of the arguments to E. Cord E now awaits results from B and either C or D. Meanwhile, cord C is ready to run, having received the result 2 from the cord on which *it* depends. To summarize, the set $C + D$ is a set of *or*-dependencies for E; $AB(C + D)$ is its overall set of *and*-dependencies.

Work Stealing. Whenever a participating workstation becomes idle, it may pull work from its peers. Hosts never actively foist work on others. We choose to use the work-stealing model for three reasons. First, the worker node knows best when the host it is running on is idle, and when it is in need of more work. Second, the worker is likely to know better *how much* work it will need (depending on its number of processors, etc.). Third, the worker is able to send with its request a description of its security policy. This allows the code producer to find or generate a matching cord and send only the certification information that the worker needs to be satisfied.

Work stealing has major implications for scheduling. In our case, it means the conductor's scheduler can run *on demand only*, that is, whenever it receives a work request or result from a worker. Since users may wish to serve cords even when their machines are not idle, reducing the scheduler's use of resources is especially advantageous.

To steal work from a conductor, a worker initiates a sequence of messages to acquire the appropriate cord and its arguments. It then verifies and runs the code, sending the result back to the conductor. If, on the other hand, the code fails to verify, the worker does not run it.

Failure Tolerance and Recovery. Of course, computation will not always proceed smoothly on a failure-prone network such as the grid. Given that grid computing attempts to solve large problems on possibly unreliable networks and often on non-dedicated, consumer-level hardware, tolerance of failure is critical. In particular, we need to be able to checkpoint programs at some reasonable granularity and to restart them with partial results, so that earlier computation is not lost if some part of the program fails (*e.g.* due to a downed node). Since cords do not communicate among themselves as they run, restarting them is not especially difficult provided the code for the cord in question is kept. With respect to checkpointing, results are cached on conductors so that if a cord *is* rescheduled, preceding cords on which it depends do not have to be re-run.

Failure detection is also an issue. If a worker notices and notifies the corresponding conductor that its cord has died or failed to match its host's security policy, the conductor can simply reschedule it and hand it off to any subsequent requester. Otherwise, we need to decide if and when to reschedule apparently failed cords. In some cases, grid programmers may wish to approximate the result of a failed cord rather than reschedule the cord itself. We expect that failure detection will involve some form of heartbeat between cords and originating conductors, but further work on implementation is required. Related work has already been done on failure detection for distributed computing, including [6].

Our framework also entails a new mode of failure, where a binary does not pass the type-checking phase once transmitted to the worker's hosts. There are several reasons such failure may occur. For example, it may fail because the certificate does not match the code, due to corruption of the files, a bad copy of the certifying compiler, malice, or similar causes. Second, it may fail because, while correct, it does not match the host's security policy. Presumably this would occur only if the conductor serving the code did not check the worker's policy against it in advance, or if the worker host's policy changed in the midst of the negotiation. In those cases, however, explicit failure notices can be generated to reduce the impact on the overall grid performance.

Grid Clients. External clients can connect to conductors on local machines to submit work and view or interpret results. To do this, they use a socket interface to send messages to the conductor specifying the cord and the arguments with which to run it. If all goes well, the conductor will return a handle to that cord, which may be used to query its status and retrieve its result through the same socket interface. Using a socket allows for easy interaction with clients written in various languages—we have written demonstration clients in C and Standard ML. The ConCert *protocol* software is also language-independent. Note, however, that the cords themselves must be available in the TAL-produced format of an object file and typing annotations, and the verifier clearly must check this format.

4 Sample Application: Ray Tracer

Because of the strong interplay between certification technology and the source programming language, we are interested in how applications are developed for the grid and especially in what programming issues arise. Therefore we have developed a sample application for our grid framework: a parallel ray tracer called Lightharp. The ray tracer back-end, which does the actual tracing work (and is distributed across the grid), is written in the Popcorn language described in section 2.2. The front-end, which divides the scene into small chunks, submits the tracing jobs, and displays the results, is a separate program written in Standard ML.

Lightharp implements the specification for the ICFP 2000 Programming Contest [20]. Scenes to be rendered are described by a simple stack-based language called GML. GML supports basic diffuse and specular lighting, constructive solid geometry, and several advanced features such as procedural textures. The GML representation of a scene is typically quite compact because it can create named objects and duplicate them to populate the scene. Therefore, rather than create an intermediate representation of the scene we communicate a (slightly) modified version of the original GML code on the grid.

The back-end is a function of type `string -> string`, which takes a GML program as input, and returns the rendered scene as a sequence of RGB colors. The color of a particular pixel is computed by shooting a "ray" from the point of view through the image plane and recursively tracing its path through the scene using standard ray-tracing algorithms. Note that the color of each pixel or block of pixels can be calculated independently, which is the only source of parallelism in our implementation.

The front-end works by parsing the GML scene and modifying it to instruct the back-end to render only a small part of the image, that is, only trace rays through a certain range of pixels. It then submits cords into the grid to cover the entire image, and simply waits for the results to arrive (currently by polling the ConCert software).

For an application with such a simple parallelism model as a ray tracer, we found that this implementation strategy was adequate. Popcorn is a powerful enough language to implement the back-end tracing functions without much pain. It is also relatively easy to manage the cords from the front-end and only mildly tedious to manually marshal between strings and RGB data.

Our ray tracer application's parallel behavior is extremely simple; for instance, the back-end cords have no need to spawn other cords, nor do they have any dependencies. Work is currently under way to tackle some more difficult parallel computations, such as parallel game tree search in chess. We hope to push the limits of what is possible with cords, and discover interesting research problems in programming techniques for the grid.

5 Conclusion

We have presented a framework and technologies for grid computing in a trustless setting based on the idea of certified code. Certification compares favorably to similar technology; it verifies rich properties of native code with low run-time overhead. We have also described a peer-to-peer network for fetching and running work, and presented a sample ray-tracing application that runs on the grid.

We believe that our trustless peer-to-peer strategy is the most effective way to lower the barriers to universal utilization of the grid, while maintaining a secure and robust network. However, code certification can be applied to other mobile code scenarios in grid computing. For instance, manually-installed native code applications like used for SETI@home [34] could benefit from the improved security and potential for automatic updates afforded by certification. Untrusted systems based on virtual machines could use certified native code to improve performance.

Future Work. At present, we certify only type safety (which entails memory safety and control flow safety). In section 2.2, we discussed other properties that might be certified. When more certification options are available, we will need to encode policies in such a way that code can be tested against them easily and efficiently. We also plan to provide an accessible interface for users to specify their security policies in a transparent manner.

Our framework protects cycle donors from broken or malicious code, but it does not protect developers from false answers. Malicious workers might, instead of actually running the code presented, return a fabricated result. We plan to investigate how techniques proposed elsewhere [32, 33] can be adapted to the ConCert architecture.

So far, we have concentrated on certification of safety properties. In practice, this should be combined with methods for peer-to-peer authentication. Here, again, the ideas behind proof-carrying code provide novel solutions [2]. We plan to investigate how they may be applied in the ConCert framework.

Although we have implemented a ray tracer application using the ConCert grid software, we noted in section 4 that the parallel structure of Lightharp is exceedingly simple. To support more sophisticated uses of parallelism, we will need a high-level programming model for supporting some notion of starting remote computation and gathering of results *within* a grid application. Consequently, we will need the means to map high-level programming languages to the simple low-level interface provided by ConCert. Although it has yet to be demonstrated, we believe we can leverage several programming language techniques to achieve this goal. Preliminary work in this direction is discussed in Chang's honors thesis [7].

Acknowledgments. Our implementation and protocols, especially the work-stealing protocol and code to support TAL verification, are based largely on Joshua Dunfield's initial implementation of the ConCert framework. We would also like to acknowledge the other members of the ConCert project and Guy Blelloch for their helpful comments.

References

[1] Andrew Appel. *Compiling With Continuations.* Cambridge University Press, Cambridge, 1992.

[2] Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In G. Tsudik, editor, *Proceedings of the 6th Conference on Computer and Communications Security*, pages 52–62, Singapore, November 1999. ACM Press.

[3] Robert D. Blumofe and Philip A. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *USENIX 1997 Annual Technical Conference on UNIX and Advanced Computing Systems*, pages 133–147, Anaheim, California, 1997.

[4] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java™ programming language. In *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 183–200, Vancouver, British Columbia, October 1998.

[5] Peter Cappello, Bernd Christiansen, Mihai F. Ionescu, Michael O. Neary, Klaus E. Schauser, and Daniel Wu. Javelin: Internet-based parallel computing using Java. In *ACM Workshop on Java for Science and Engineering Computation*, Las Vegas, Nevada, June 1997.

[6] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[7] Bor-Yuh Evan Chang. Iktara in ConCert: Realizing a certified grid computing framework from a programmer's perspective. Technical Report CMU-CS-02-150, Carnegie Mellon University, June 2002. Undergraduate honors thesis.

[8] Clip2 Distributed Search Services. The Gnutella protocol specification v0.4, September 2000. URL: http://www.gnutella.co.uk/library/pdf/gnutella_protocol_0.4.pdf.

[9] ConCert. Certified code for grid computing, project webpage, 2001. URL: http://www.cs.cmu.edu/~concert.

[10] Karl Crary and Stephanie Weirich. Resource bound certification. In *Twenty-Seventh ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 184–198, Boston, Massachusetts, January 2000.

[11] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, San Francisco, California, 1999.

[12] Margaret DeLap. Implementing a framework for certified grid computing. Technical Report CMU-CS-02-151, Carnegie Mellon University, June 2002. Undergraduate honors thesis.

[13] Ian Foster and Carl Kesselman. The Globus toolkit. In Ian Foster and Carl Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*, chapter 11, pages 259–278. Morgan Kaufmann, San Francisco, California, 1999.

[14] Neal Glew and Greg Morrisett. Type-safe linking and modular assembly language. In *Twenty-Sixth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 250–261, San Antonio, Texas, January 1999.

[15] Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *USENIX Symposium on Internet Technologies and Systems*, Monterey, California, December 1997.

[16] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java™ Language Specification*. Addison-Wesley, second edition, 2000.

[17] Paul A. Gray and Vaidy S. Sunderam. Metacomputing with the IceT system. *International Journal of High Performance Computing Applications*, 13(3):241–252, 1999.

[18] Andrew S. Grimshaw and William A. Wulf. Legion: The next logical step toward the world-wide virtual computer. *Communications of the ACM*, 40(1):39–45, January 1997.

[19] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Twenty-Fifth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 365–377, San Diego, California, January 1998.

[20] ICFP. The third annual ICFP programming contest, 2000. URL: http://www.cs.cornell.edu/icfp/.

[21] Tim Lindholm and Frank Yellin. *The Java^{TM} Virtual Machine Specification*. Addison-Wesley, second edition, 1999.

[22] Steven Lucco, Oliver Sharp, and Robert Wahbe. Omniware: A universal substrate for web programming. In *Fourth International World Wide Web Conference*, pages 359–368, Boston, Massachusetts, December 1995.

[23] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Massachusetts, 1997.

[24] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, Georgia, May 1999.

[25] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. *Journal of Functional Programming*, 12(1):43–88, January 2002.

[26] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

[27] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *Twenty-Sixth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 228–241, San Antonio, Texas, January 1999.

[28] George C. Necula. Proof-carrying code. In *Twenty-Fourth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, January 1997.

[29] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Second Symposium on Operating Systems Design and Implementation*, pages 229–243, Seattle, Washington, October 1996.

[30] George C. Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. In Giovanni Vigna, editor, *Special Issue on Mobile Agent Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 61–91. Springer-Verlag, October 1997.

[31] Jordan Ritter. Why Gnutella can't scale. No, really., February 2001. URL: http://www.darkridge.com/~jpr5/doc/gnutella.html.

[32] Luis F. G. Sarmenta. Bayanihan: Web-based volunteer computing using Java. In *Second International Conference on World-Wide Computing and its Applications*, pages 444–461, March 1998.

[33] Luis F. G. Sarmenta and Satoshi Hirano. Bayanihan: Building and studying web-based volunteer computing systems using Java. *Future Generation Computer Systems*, 15(5-6):675–686, 1999. Special Issue on Metacomputing.

[34] SETI@home. The search for extraterrestrial intelligence, 2001. URL: http://setiathome.ssl.berkeley.edu.

[35] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, December 1993.

[36] Steve Zdancewic and Andrew C. Myers. Confidentiality and integrity with untrusted hosts. Technical Report 2000-1810, Cornell University, 2000.