Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference

Boston, Massachusetts, USA
June 25–30, 2001

A Practical Scripting Environment for Mobile Devices

Brian Ward
Department of Computer Science
The University of Chicago
bri@cs.uchicago.edu

Abstract

Software development for small mobile devices concentrates on cross-compiling C into platform-specific object code. Each platform has its own idiosyncrasies in setting up an application, memory management, display management, and so on. We present a new programming approach based on dynamic content generation that simplifies development of common applications for mobile devices. This paper introduces new software development tools intended for mobile devices; our approach utilizes a compiler and a multiplatform runtime environment. In the text, we first note a few problems in mobile code development, then briefly analyze a typical mobile application. From this, we draw a similarity to a familiar programming model, and formulate what we would like to borrow from that and apply to mobile code development. Finally, we present the current status of the project and its future plans.

1 Introduction

When a programmer gets a new toy like a Palm handheld, one of the first things that comes to mind is "I'd like to write some code for this." "I'd like to port *foo* to this" also comes to mind. However, the harsh reality of mobile devices is that software development really isn't very convenient. A typical situation is that one writes the code in C on a workstation, runs a cross-compiler on it, loads the resulting binary into the device, then crashes the device without knowing very well what happened. For the more popular devices, there are some additional tools such as device emulators, but of course, this doesn't stop the code from crashing. These facilities also do little to help figuring out the obscure ritual required to print "`Hello, World`" nor do they help in porting the code to other mobile platforms.

Graphical display on mobile devices, as with many environments, isn't convenient to do on a pixel-by-pixel basis. Palm development in C addresses this in part with 'forms,' [8] which are specifications for widget placement on the screen which the programmer pre-specifies separately. Unfortunately, the consequence is that it impedes any dynamic flavor of the display. This is fairly serious, since a mobile device isn't usually used to do any real computation–it is usually there only to extract and display information.

There are some alternatives to the C model, such as Java2ME[3], but these have not been terribly popular because, for the most part, they have many of the same problems (only the language has changed).

Most programmers would probably think highly of a language like Python being available on the Palm. However, there must be more to it than this, since that has already been done[10]. Not only is there is an acute lack of support for display techniques, but since they were written for a much larger environment such as Unix, they often have to be considerably stripped down, with additional limitations placed on what they can actually do.

Moreover, there is a lack of a cross-platform mobile software development 'for the rest of us.' Most people write very simple programs and don't want to have to worry about memory management, heap and record size limitations, and so on.

2 Application Analysis

Perhaps some illumination on this problem lies not in the means to the end, but in the end itself. One should look at the actual applications which run on these devices and see how they work and what they do.

Certain scripting languages like Python and Perl are popular in part because they rather fit in quite well with the shell interface of a Unix system. When at a full-size machine, command-line batch processing prevails. As we develop scripts, we run them over and over again at our shell prompt, often redirecting the output to other files, where we may look at them with a pager program, a web browser, etc. When satisfied with the output, we either continue to run the scripts at a shell prompt or automate them in some way (using cron jobs, window manager macros, dynamic web pages, and so on).

Our observations show that mobile applications have a much different interface model than those of the typical Unix desktop. When a user runs an application on a handheld, some sort of full-screen display usually appears- often, a menu of sorts. Then one selects something that looks interesting and the page either redisplays, or a new page appears to take the place of the current. In a sense, it's a "tap, read screen, tap, read screen" sort of model. Because this sounds a lot like "point and click," we have an idea of where to head to next.

3 We've Seen This Before

This interface model is remarkably similar to a popular strategy for larger web sites, that of so-called 'dynamic content.' While the actual implementation language and detail vary, most of these web sites boil down to code like this:

```
<? print_header(); ?>
Here's some stuff:<br>
<?
  $fd = db_open("server", "dbname");
  $result = db_query($fd,
   "select * from foo where bar = baz");
  while ($row = db_fetchrow($result)) {
    print $row[2] . " " . $row[1];
    print "<br>\n";
  }
  db_close($fd);

  print_footer();
?>
```

This particular segment uses the PHP[9] language between the `<?  ?>` delimiters. The `db*` functions are database access functions.

Most popular mobile applications do the same sort of thing; this likeness is reinforced by the number of personal organizer and directory web sites available (some of which can synchronize with mobile devices).

Though they differ in syntax, implementation, and style, the languages used on the web have one thing in common. In addition to a set of built-in functions, they each offer an interface to add additional functions written in C. This model has proved popular enough to inspire additional tools (e.g. SWIG[2]) meant to further bridge the gap. Due to this level of support, the 'core' set of functions grows as the language matures. This leaves the question of what the core functions are. For example, Python aims for a smaller core, while a PHP build attempts to compile and include as many functions as possible up front. Perl lies somewhere in between.

4 Intent

We therefore asked a question on what it take to bring this sort of convenience to mobile platforms. Previous research shows that mobile devices are inherently poorly-connected and resource-poor[11], so we cannot rely on proxy servers to do the computation for us in real-time (the WAP model[12]), nor can we simply start up a web server on the device and then run a web browser (at least not yet).

An important aspect is ease of presentation. A "Hello World" program should not be much more the text itself. Augmenting this are the usual string manipulation functions available in most scripting languages, as well as interfaces to important system functions (such as database access). This is one of the most important advantages to Unix-based languages such as Python and Perl in web servers–they not only have handy access to common library calls to get the data that they but they also have significant power and ease in producing any kind of output required.

5 Design Overview

A system for doing just this is currently under development. It consists of two parts: a parser/compiler and virtual machine-interpreter. The former takes representation descriptions along with a C/PHP-like language (currently called HHL) and produces a bytecode for the virtual machine-interpreter (named VL).

The VL execution model is similar to the dynamic web server example, which runs scripts once per click and sends them to the web browser as a simple document. When the user runs a program, the code is fed into VL. The immediate output of this is a description of the on-screen content, which VL then draws on the screen, in the same way that a web browser might display HTML. If a whole new page is desired (for example, we're in an address book listing we tap on an entry to bring up the full entry), the whole process repeats for the new page. However, in a mobile environment, we can change a few of the rules. Since the machine which generated the content also happens to be the one displaying the content, all of the information about is still in place. This allows us to update portions of the screen without a full redisplay.

6 The HHL Language

A procedural language, HHL resembles PHP and C. Its flow control constructs include `if-then-else` and `while`. Variables begin with a dollar sign (e.g. `$stuff`).

Functions are defined as in the following example:

```
function add($arg, $another_arg) {
    global $scale;
    return($scale * $arg * another_arg);
}
```

If the function returns no useful value (like a C `void` function), `return` is not necessary. In addition to these user-defined functions, the runtime environment provides a number of built-in functions, such as `atoi()` for converting strings to integers, and `cellp()`, used display modification. It is acceptable to replace a builtin function with a user-defined function within a single program.

There are two basic data types, strings and integers. In addition, the list type provides array- and list-like functions.

7 Display

There are two stages to the display of information. The first is a display specification which determines the layout of the screen and what initially appears. This corresponds to HTML in the web model, and it employs SGML/HTML-like tags. As in a system such as PHP, one can provide a static markup, with HHL code, or a mixture of both.

Instead of using a complete markup language such as HTML, we only employ a small number of tags. In analyzing mobile applications, one HTML-like element stands out among others: the table. From listboxes to buttons, almost anything can be reduced to a table with a user-defined action for taps on the element. This is a pure code segment, making the display a 2x2 matrix of numbers, 1 to 4:

```
<table c=2>
<c>1</c><c>2</c>
<c>3</c><c>4</c>
```

One can insert HHL code to create dynamic content:

```
<table c=2>
```

```
<?
  $i = 1;
  while ($i < 5) {
    print "<c>$i</c>"
  }
?>
```

An important element of the display system is that HHL code can interact with the display device and use the information not only to adapt the current screen geometry, but also to detect how much room is left on the screen given the amount of information already present. This example employs the `rows_left()` function:

```
<table c=2>
<c>Row Number</c><c>Rows Left</c>
<?
  $n = 0;
  while ($i = rows_left()) {
    print "<c>";
    print $n;
    print "</c><c>";
    print $i;
    print "</c>";
    $n++;
  }
?>
```

8 Actions and Display Updates

Now that we've described the display system's basic infrastructure, we must sort out the details of updates to a page. This is where our system most deviates from the web model. We've described the fact that we need to be more aware of our state. A web browser isn't very strong in this regard, nor should it be, for security, consistency, and other reasons. Typically, when the user clicks on something in a page to update it, the page redisplays, sending some state parameters back to the web server. There are some ways to do non-critical browser-side update features with Javascript, but the use of this isn't widespread due to mixed results.

Our system resembles the Tk toolkit's system[7] to a certain degree, but with some modifications for a compiled language. After VL loads and runs an application, it waits for an event, such as a display tap or network activity. The event triggers an event-handling function if applicable.

When defining a cell with the `<c>` tag, you may provide a name and action, as in this example:

```
<table c=2>
<c name=example action=tap_me>Tap me.</c>
<c name=display></c>
<?
  function tap_me($name) {
    cellp("display", "Ow.")
  }
?>
```

When the user taps on the "Tap me" cell, VL calls `tap_me("example")`. When the function completes execution, the VL returns to the idle-event loop.

9 Implementation

9.1 HHL Compiler

The source code for the system has essentially three pieces: the HHL compiler, which is much like the source for any Unix utility, the platform-independent core of VL, and the platform-specific support file sets for each target. Excepting the lex and yacc constructs, all code is ANSI Standard C.

A straightforward tool with few frills, the compiler generates parse trees and massages them, then generates code in two distinct steps. Optimization leans toward eliminating redundancy and space economy. For example, because operations on strings tend to involve multiple function calls and exercise the memory management system, some of those operations are analyzed. Two sequential `print` statements

```
print $str1 . "</c>";
print "<c>" . $str2;
```

would normally generate around ten instructions total in the target code. By condensing this segment into a single `print` statement, we can save three instructions. Because this sort of optimization requires certain semantic knowledge of builtin functions, possibly leading to an unnecessarily complex compiler, we aim only for a select few such tweaks.

At the moment, the compiler runs only on Unix platforms; while in distinct components, its primary use is on a desktop machine.

9.2 VL Core

VL's development has been multiplatform from the start, immediately forcing a distinction between the front-end and computational components. Furthermore, to avoid code riddled with `#ifdef` directives, some modules require separate files for certain functions for each platform; the display and events interfaces are two examples.

At first glance, the VL object code resembles that of a normal microprocessor. It has register-like storage locations, comparisons, branching, and other common items. However, the design has a number of amenities meant to simplify compilation from a higher-level language. For example, in addition to a regular execution stack, it has another stack for use by operations and functions, not unlike the stack in very old hardware architectures [1]. This not only makes compiler implementation easier, but greatly reduces object file size.

Also in the interest of object size reduction, the instructions are of variable length. A further assumption is that much of the object code will consist of function execution. Therefore, there is some emphasis on function call setup size, and the complexity of returning a value. The storage location addressing scheme provides fast mapping from a storage location address to the execution stack. Though it is possible to implement nested functions using traditional methods dating from Algol[6], they are currently absent in HHL.

VL enforces types at run time. Each storage location has a type, regardless of any memory manager's involvement. However, the type enforcement is not overly strict. Function arguments do not undergo typechecking during function execution. For user-defined functions, this is not an issue. Any built-in functions check their arguments as they see fit. For example, since a string operation on a non-string might cause a crash, string manipulation functions should make sure that their arguments are indeed strings. There is also no particular obligation to return run-time errors in the case of a type clash; if `atoi()` gets an integer as its argument instead of a string, it is perfectly acceptable to return that same integer as the result instead of flagging an error.

In addition to the (lack of) obligation for types, a builtin function does not need to check its number of arguments. This has an advantage for functions such as string append. We noted above that we could save some instructions by condensing the two `print` calls into one. In the original form, there are actually four function calls–two `print()` and two string appends. In the condensed form, there are only one of each, where the string append has four arguments instead of two.

The VL core module processes object code. The front end hands the module a memory pointer to the start of the code. The initialization function analyzes the object header, and determines where execution should start. The header includes the names of all functions called in the object code, offset addresses for program-defined functions, and a string constant table. Following the header is the main program code, and finally, the code of all program-defined functions. The initializer makes a note of where the code begins and ends, in the interest of catching illegal program counter access.

The core module includes the instruction executer. After it reads and runs each instruction, it updates the program counter, and leaves an opportunity for an event, such as a tap on the screen or program interrupt. Allowing these interrupts only between instructions simplifies the system not only because we do not have to worry about half-executed instructions, but also cuts down on the complexity of the interrupt handler for each platform.

9.3 VL Functions

Like the core, the VL builtin function framework is also platform-independent. The builtin function vector is a table of HHL function names paired with real function addresses. For example, if the implementation of a builtin function were a C function `vl_builtin_print` and the HHL name were `print`, the entry in the table would be

```
{ "print", vl_builtin_print }
```

When the VL core initializes a program, it identifies all functions that the compiler assumed to be builtin, and attempts to locate them within the function vector. The core stores this location so that it may call a builtin quickly during program execution. We call this dispatching a function; the builtin function dispatcher routine is also in the

To add a builtin function, one needs to write the function with the prototype `Int32 (Int32 argc, Int32 offset)`, and place the function along with its HHL name into the builtin function vector. `argc` is the number of arguments given when the program calls the function, and `offset` is the stack offset. Here is an example which simply adds two arguments together:

```
Int32 vl_builtin_add2
     (Int32 argc, Int32 offset) {
  vl_var i, j;

  /* if num of args not 2, return null */
  if (argc != 2) {
    return(1);
  }
  i = grab_var_with_type(offset);
  j = grab_var_with_type(offset+1);
  s_push(*i.value + *j.value, VL_TYPE_INT)
  return(0);
}
```

We see a number of features from this example. First, we note a safety feature: as long as the dispatcher calls the function with a correct value for `argc`, this function cannot walk off the edge of the execution stack. The assignments to `i` and `j` illustrate how to access a the first and second arguments. Finally, we see that there are two ways to exit from a builtin function; with a return value of 1, indicating that the function doesn't have a meaningful return value (such as `void` in C), or 0, meaning that the function has put its return value on the operation stack mentioned previously. If there are code branches leaving the stack in different states as above, this is not a problem. The dispatcher puts the stack in order if necessary.

Simple builtin functions such as the one in this example do not need to know about their target architecture. However, there are cases in which function implementations are necessarily different, such as the display interface. One may take several approaches in these cases. If the differences are minor, `#ifdef` preprocessor directives may suffice. If there are large discrepancies, it may be necessary to write each implementation separately, possibly placing them in different source files.

10 Stability

10.1 General

One of our primary concerns is to keep programs from causing an crash. In our case, this is an especially serious matter, mainly due to the operating system support on some of our platforms. Since they tend to be on smaller devices, several years behind current processor computational power specifications, operating system designers must make some sacrifices. Memory management is sometimes weaker, in part due to a lack of hardware support.

But if scripting languages on a Unix desktop don't normally experience segmentation faults (other than running dynamically-loadable modules), we should expect to be capable of the same on a handheld. This need not conflict with our interest in performance. For instance, it is not necessary to do a rigorous check on every bit of code upon initialization, seeing if the number of arguments are correct for each instruction is valid, or if functions are called with matching numbers of parameters. A problem such as an invalid instruction pops up soon enough. Other matters cannot be glossed over. For example, each compiled-in string must be checked to see if it appears as it is advertised: length matching string header value, NULL character at end.

10.2 Type System

There are a few particular areas of VL to concentrate on; some rely on the stability of others. First, in the VL core, we have a few basic data structures to worry about. We have the execution and operation stacks. A common set of access functions can assure us that we will not run off the borders of these. If we check the stack at each access, and make sure that the stacks see no other accesses outside of this function, we will not have an illegal memory access.

Our type system presents some obstacles. A language such as ML is strongly typed and doesn't need type information once the code is compiled[5]. Not only is our type system a mainly runtime affair, but we have no reason to trust the compiled code, for someone may have inserted a rogue instruction. Therefore, we build our type security up from a few base rules.

The first rule is that VL instructions may only set the types of a limited set of data, and that only when an instruction loads something into a storage area (these are similar to assembly 'Load Immediate' instructions). Let's assume that these two types are numbers and compiled-in strings. Numbers present no difficulty as far as memory is concerned, and any safe operation or function will know to check to see if a number is cause for concern. Compiled-in strings have one precaution. Each such string is ultimately represented by an index to a table created at program initialization. If that index isn't in the table when a VL instruction sets this type, a runtime error occurs.

Furthermore, there is no other way to set a type from an instruction. All instructions which shift data around (e.g. to and from the stacks) work with both data and type at once, keeping them together.

Therefore, with instructions alone, there is no way to have a type mismatch that may cause an operation or function to get bad data. This leaves two questions: First, while this type system is safe, wouldn't one like to have more types? In addition, can the builtin functions handle this?

Both questions may be answered at once. Builtin functions are the key to more types. We've already talked about compiled-in strings; naturally, we would like to have dynamically-allocated strings as well. We can make a new rule: only builtin functions may return the dynamically-allocated string type. If these functions make certain that the string data that they return is 'real,' there is no danger in passing it back to the VL core. No core instructions may change the data without changing the type, and as we already mentioned, they can't change the type to anything but the simple ones.

Assuming that our builtin functions don't have any bugs (and of course, we know that never happens), this system suffices to keep bad data out of the builtins' hands. But is it enough to assure that our language is useful? With strings, it certainly is: the only way one would ever get any string that's not known at compile time is to use a builtin such as a database access function or string append. If we can do similar things with network sockets, we can achieve the same level of sophistication. Of course, we don't let VL instructions play with pointers. While this can hinder us in some programming environments [4], our system is not intended to be a platform for heavy-duty applications.

With this in mind, it is worthwhile to emphasize that builtin functions must dutifully check their arguments and return a valid data-type pair. If one bad piece of data got into the mix, the result could be catastrophic.

One more note about builtin functions and stability is in order, pertaining to blocking operations. On some platforms, blocked operations can interfere with the operating system. If a program is busy with something and doesn't handle an event from the operating system, application buttons may not work, events may be dropped, the system may get sluggish, or worse. Therefore, builtin functions have something of an obligation to try to avoid blocking operations, or if they must be busy for a period of time, to try to service events if they come in. VL provides an interface for this in its front end through two function calls. One checks for pending events, and the other services any such events.

11 Security Concerns

In a portable development environment, one should explain any security measures because one can easily transfer code (HHL or VL object code) over a network. Our general philosophy towards the matter is the same as for a normal scripting language such as Perl or Python: if you want to run a program you got off the net, you should be a little wary of it. Moreover, this system is not intended as an alternative to the world wide web; it is meant primarily for small application development.

However, there is an extra danger because some potential target platforms do not have the same notions of users and system security as Unix. As a regular Unix user, the most destructive thing you can normally do is to erase all of your own files. The system doesn't care. But on a machine such as a Palm handheld, you can not only erase all of your files, but also the whole system.

One approach to this is just to offer the advice "back up your data." While hard to argue with, not everyone does it, so can we provide more protection? At this point, we offer a proposition on how one might address this issue (no such system has been implemented). Other languages such as Tcl have network safety features when compiled in a certain way; they simply disable builtin functions which may alter the system. For example, we should make databases read-only for an untrusted program, and even turn the network functions off so that the data cannot 'escape' the system.

However, maybe we want to allow selective access, as we'd like our programs to access personal data and do something with it. We can flag any database access and have the user manually confirm it the first time through. Access control lists to certain builtin functions and databases are another possibility. However, this leaves questions of how the user will react to this; they may just develop a habit of confirming everything that comes their way.

12 Sample Applications

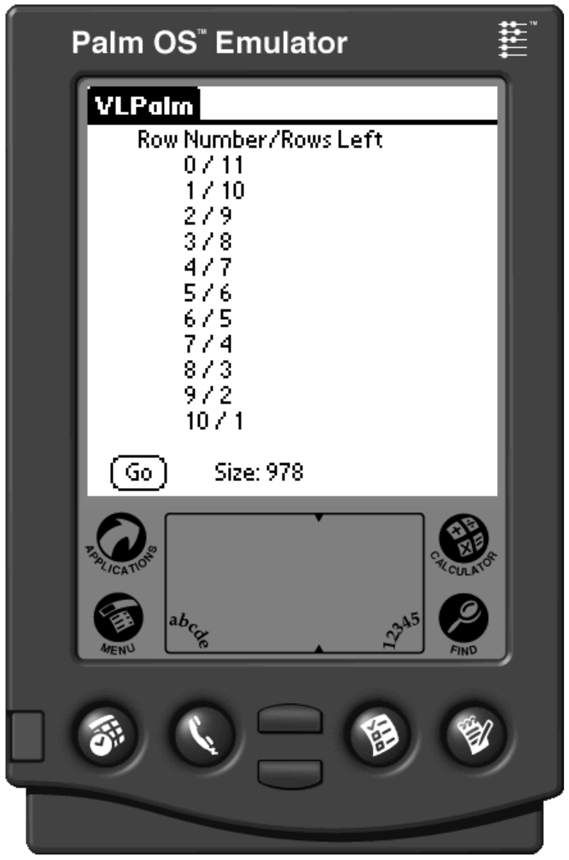
So far we have talked much about the development infrastructure, but little about the actual programs that we intend to write in this system. Therefore, we give some possibilities that are particularly suited to mobile devices.

Custom Schedules. Part of the initial motivation for this project came from the desire for something better than paper train schedules. Various attempts have been made to put train schedules on handhelds, but apparently due to the formatting complexity, these seem just like the paper schedules, only harder to read. An invaluable schedule application would quickly hone in on a certain route, remember it, and perhaps most importantly, automatically find the next trains based on the current time of day.

There are schedules on the web which show some potential features. For example, some transport agencies' sites allow the user to click on a station to bring up information about that station–address, phone, and so on. This would be a simple function for our application to mirror, as it is only a page redisplay.

Network Monitoring. Because there is nothing to stop builtin functions from accessing network data, it should be possible to run network monitoring programs on VL. In addition to the usual socket interface, the creative network administrator may wish to customize their VL, adding specialized builtin functions to filter through a large amount of network data and return only items of interest (remember that a handheld's display isn't large enough to display much).

Simple Games. An unofficial (and not often stated) measure of a development tool is the quality of the games that it can produce. Clearly, VL isn't terribly suited for 3-D animated action games, but it can easily accommodate static games. Examples of these include the minesweeper type games, card games, and the ever-popular DopeWars.



```
VLPalm
    Row Number/Rows Left
         0 / 11
         1 / 10
         2 / 9
         3 / 8
         4 / 7
         5 / 6
         6 / 5
         7 / 4
         8 / 3
         9 / 2
         10 / 1

 Go       Size: 978
```

Figure 1: VL on Palm

13 Operation

The HHL compiler runs on Unix-like systems; it acts like any Unix compiler. VL, on the other hand, runs on the PalmOS platform as well as Unix. The compiler is named `hhlc`. Compiling a source file is simple:

```
$ hhlc file.hhl
```

If there were no errors in the source, `hhlc` creates a VL object file called `out`. Simple debugging may be accomplished with a command-line version of VL, simply called `vl`. This version has no graphical capabilities, therefore, it doesn't require a display or any extra hardware. Its primary function is debugging; it reads text from the standard input, and if the text matches the name of a cell, any defined action for that cell executes. It's therefore possible to run through actions on programs by redirecting standard output.

Unix VL only needs to read the output file that the HHL compiler produces, but because PalmOS is unfriendly to 'alien files,' a Unix conduit is available for the code transfer into a PalmOS database. Once you're happy with the way the program runs on Unix, one may try it out on the Palm. The `install-vlo` program can transfer `out` to a handheld.

Assuming that the `vl_palm.prc` Palm application which comes with the `hhlc` and VL distribution is present, we're ready to run the program there. Tapping on VLPalm, and then the **Go** button starts up the program. Figure 1 shows a simple program similar to one illustrated in the **Display** section, one that displays the number of rows remaining.

14 Current Status

The HHL compiler currently supports all core language features (flow control constructs, user-defined functions, basic types, and so on). Needing more refinement are areas such as diagnostic output and optimization. At the moment, HHL syntax errors are flagged with the line number of the error, but no real effort is made to attempt to zero in on the actual problem. Similarly, not much care is taken to process the parse trees created by `hhl.y`. If these trees were to be massaged, cleaning up the internal representation of statement blocks, it would be make some optimizations simpler.

VL's frontiers lie primarily on one front: builtin functions. Because the general framework for builtins is in place, making it simple to add a new function, adding these fuctions is an incremental process. In particular, more string and display routines are in order, as well as networking functions. While it would be interesting to access certain platform specific features, such as interfacing the IR port on a Palm handheld, this is not a high priority. General functionality (and keeping builtins free of bugs) is the current focus.

In addition to the builtin functions, work is currently underway on the VL ports to some other platforms. Furthermore, one new modification adds a graphic interface to the Unix stdio-based `vl`.

Finally, more work is needed in application management.

15 Future Plans

After the system stabilizes to a certain point, there are two avenues of development, both of which are of particular interest to the open source community. The first is to port VL to as many devices as possible. At the moment, Palm Computing has a lion's share of the market for small mobile devices, but in addition to the prospect of supporting their competitors by having true cross-platform code, there is also an issue of hardware and software upgrades. Bytecode written by the HHL compiler would (presumably) not be affected by these.

Another area of interest is in getting the HHL compiler itself ported to some VL target platforms. The first advantage to this is that it allows for "true mobile development." One would no longer need to run a compiler on another machine to create new programs; if one so desired, they could write the program on the target device. Moreover it also create some interesting opportunity for interaction and debugging. While playing with a program, it would be possible to compile a new function and insert it into the currently-running program on the fly, as well as get more information about the current state of the program.

Our environment currently focuses its attention to hardware on handheld computers, in particular, those that have a stylus. These devices are best suited for our interaction model (based on screen taps, not pressing buttons). Because the input is uniform, we can concentrate on providing a consistent and portable user interface. However, there are other mobile devices capable of a VL port, such as cellular telephones. One area of future work could be adapting the input method to support devices without a stylus or touch pad. Furthermore, the display on mobile phones is much smaller than those on a handheld computer, posing a question of how difficult it is to write programs that work on both display types without too many conditionals around the display size.

16 Concluding Words

Current development tools for mobile platforms tend to the somewhat painful side. Quite a bit of effort is required in writing a program in a language such as C or Java, and the result is often that the emphasis shifts away from the information which is ultimately destined for the screen. The HHL compiler and VL runtime environment offer a content-based approach to code development, with a model more like that of the web, with the additional that platform peculiarities can largely be ignored.

17 Availability

The tools are intended to be distributed as open source. The prototype system, which consists of the simple compiler and virtual machine, is available from

`http://www.o--o.net/comp/`

References

[1] Barton, R. S. "A new approach to the functional design of a computer," *Proc. Western Joint Computer Conf.* (1961), 393-396.

[2] Beazley, D. M. "SWIG: An easy to use tool for integrating scripting languages with C and C++," *Proc. 4th USENIX Tcl/Tk Workshop*, 129-139.

[3] Java 2 Micro Edition.
`http://java.sun.com/j2me/`

[4] Kernighan, B. "Why Pascal is not my favorite programming language," original 1981; appeared in *Comparing and Assessing Programming Languages*, Prentice-Hall, 1984.

[5] Milner, R. "A theory of type polymorphism in programming," *Journal of Computer and System Sciences*, Volume 17, 1978, 349-375.

[6] Naur, P., Backus, J. W., Bauer, F. L., Green, J., Katz, C., McCarthy, J., Perlis, A. J., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J. H., van Wijngaarden, A., and Woodger, M. "Revised report on the algorithmic language ALGOL 60," *Communications of the ACM*, 6(1) 1963, 1-17.

[7] Ousterhout, J. K. *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.

[8] Rhodes, N., McKeehan, J. *Palm Programming*, O'Reilley and Associates, 1999.

[9] PHP Dynamic Hypertext Processor.
`http://www.php.net/`

[10] Python to Palm Ports.
`http://www.endeavors.com/pippy/`
`http://www.isr.uci.edu/`
`projects/sensos/python/`

[11] Satyanarayanan, M. "Mobile information access," *IEEE Personal Communications*, 3(1), February 1996.

[12] Wireless Application Protocol.
`http://www.wap.com/`