`make world`

Christopher Smowton *Steven Hand*
University of Cambridge Computer Laboratory
15 JJ Thomson Avenue
Cambridge, United Kingdom

1 Introduction

The world is full of computers doing repetitive, redundant work. Every computer which starts up enumerates its hardware, just in case it's changed. Every time you start `python` it searches anew for extension modules, just in case you've installed any since last time. Every time you run your web browser it makes a similar check for extensions, fonts, the widget toolkit you prefer...

The work happens every time, but the result is almost always the same: your hardware is almost certainly the same as it was yesterday, and the same goes for your web browser configuration. It wouldn't be difficult to produce the result once, record it and use that on each subsequent occasion. It would, however, be a tedious task for the software's maintainers. They have to write the code that determines the state of the world and describes it as they wish anyway, and they'd need to devote man-hours to writing and debugging extra code that saves and restores that description. So they don't: they theorise that starting your operating system, or web browser, doesn't happen that often, so we can live with taking the slow path, re-examining the world from scratch, every time.

However, that's not the whole story: assuming that the world is in the same state as last time doesn't just save the time required to examine it. It also presents the opportunity to specialise the rest of the program according to those assumptions, producing a program which could be orders of magnitude smaller [2], in terms of code size and memory footprint, and faster at runtime. This acceleration could stem from information about command-line parameters, the environment, files on disk, other (possibly remote) processes, or even runtime user input.

We argue that opportunities for specialisation have not been maximally exploited because doing so on a program-by-program basis is time-consuming, places a high maintenance burden on programmers, and is difficult to get right. Therefore if repeated work like this is to be eliminated it will have to be done using an automated or semi-automated tool: something which makes it easier for developers to write code that does the least possible work.

In this paper we describe an architecture in which optimisation is lifted beyond compile-time to encompass knowledge about program-external system state, aiming to eliminate both costly operations such as I/O from the critical path, and to aggressively optimise programs based on that state. We also describe a prototype implementation that can entirely evaluate small programs written in C using the standard Unix filesystem-related system calls.

2 Pull and Push Dataflow

In a dataflow graph consisting of consumers and producers, where the consumers are interested in data yielded by producers, there are two potential ways the system can function: the consumers can pull information from producers when they need it, or the producers can eagerly produce results and push them to interested consumers as they become available.

Pulling data makes sense when that data might never be used, saving redundant computation, or when we can't predict what the consumers will need, saving needless work. Pushing data makes sense when data is likely or certain to be needed, and the producers can effectively predict who needs what.

Contemporary operating systems contain many examples of both forms of data flow. In a typical GNU/Linux OS we can see pull dataflow occurring whenever programs read from disk or perform IPC to check the system state – for example, most utilities read an `rc` or `conf` file, and may contact persistent daemons such as the X server or `gconf`, in order to determine how they should function. Here the producer is the user or another program which last wrote the file, or participates in IPC, and consumer is the newly started program. Not only does the consumer pull the desired information, but it often dis-

cards the results between runs and pulls again next time, analogous to polling a device or remote service for its state.

Examples of push dataflow include the `lilo` boot-loader [7], which stores disk block offsets which must be updated manually if a relevant kernel image moves on the disk: the producer (the user or script which moved or updated the kernel) must push that state into `lilo`'s desired form. Other examples include TEX [12], which stores indexes in certain key directories which must be manually updated whenever those directories change, and SELinux [16] policies, which can be specified in human readable form and which are then compiled to machine-readable form, doing the parsing work once per update rather than per use. Note however that all these examples are both manually executed by the user and do not push results to the maximum extent: whilst, for example, we push from a human readable `lilo` configuration to a machine readable list of disk blocks, we don't precompute the menu it shows on screen at boot time, or eliminate dead code our configuration doesn't need.

A more fully realised example of push dataflow occurs in compilers. By performing compilation once per source edit, rather than every time the program is run, we potentially waste time if the program is never run, but save time each time it is invoked. Systems for coordination of compilers and compiler-like tools, such as `make`, extend this push dataflow to a tree of dependencies, performing each translation when its predecessors have changed and storing the result long-term.

Many compilers and related tools also specialise the program to its circumstances; for example to the particular instruction set available on the local machine, or the availability of certain libraries. However, even this doesn't specialise programs, and therefore push information, as far as possible: they could be specialised with respect to other aspects of the state of the system on which they will run, such as the versions of libraries dynamically available at runtime, the machine's hardware setup, or the contents of files controlling the program's behaviour.

There are two important reasons why programs are typically not maximally specialised: firstly because if it were specialised according to the version of some library, a user upgrading that library would need to re-specialise the program – it might be less convenient to use the specialised version. Secondly, it is often easier to write code which pulls state from files, the hardware setup or other processes than to write code-generating scripts which would perform appropriate specialisation.

We propose an architecture for aggressive specialisation based on *partial evaluation* [5], a technique which can automatically specialise programs with respect to arbitrary facts known about the world. Because partial evaluators can function automatically, this architecture will permit maximal exploitation of push dataflow, which we term pervasive specialisation, without significant cost in developer man-hours.

3 Partial Evaluation

Partial evaluation is a program transformation technique which simplifies programs in the light of known facts. As a trivial example it could take a simple function of two arguments, such as `fn x, y -> if x then y else 5`, and given the known fact `x = true` could reduce that into the simpler function of one argument `fn y -> y`. In terms of push and pull dataflow it serves to push the information that `x` is in fact `true` into the program, converting zero-or-more tests of `x` at runtime to exactly one test at specialisation time.

The core challenge of partial evaluation is to automatically improve a program as much as possible given certain facts about its explicit or implicit parameters [5], including persistent state which the program can access. The basic techniques are commonly used in ordinary optimising compilers; they include unrolling and peeling loops, inlining functions and performing constant propagation [1] with the basic goal of executing program instructions once at specialisation time rather than each time the program is run.

Early partial evaluators were able to specialise code written in functional languages according to known-constant parameters [6, 2], and later work could specialise imperative programs written in C according to their explicit parameters [3, 1]. However, to our knowledge whilst certain limited kernel specialisation has been attempted [13, 14], particularly for embedded systems, nobody has brought full partial evaluation to bear on implicit program parameters derived from persistent system state, and in particular the results of system calls. We argue that everyday systems should routinely and aggressively use these techniques to improve their programs as much as possible given the current state of the system by detecting changes to state which programs use and performing automatic respecialisation as and when it is necessary.

Optimising compilers do their best when the maximum amount of information is available [10]: when parameters are known to be constant, array or allocation dimensions are known, and when external functions' effects are known. Partial evaluation with respect to known results of system calls can help in all these ways: the elimination of the calls themselves means that optimisers no longer need to assume the worst about the callout's memory effects. Meanwhile the known results can feed other optimisations with more information: in particular constant propagation is likely to resolve branches

```
# Original:
fd = open("/etc/foo.conf")
for line in fd:
  if line.contains("bar"):
    return true
return false

# After replacing open calls:
fd = <"/etc/foo.conf", pos 0>
# ...

# After peeling the loop once:
line1 = "baz"
fd = <"/etc/foo.conf", pos 4>
if line1.contains("bar"):
  return true
for line in fd:
  # ...

# Evaluating constant expressions...:
fd = <"/etc/foo.conf", pos 4>
for line in fd:
  # ...
```

Figure 1: Pseudocode example of simple partial evaluation of a file-reading routine

in the program which depend on system state, leading to dead code elimination of untaken branches and thus opportunities for further optimisation downstream of the branch. Figure 1 shows this process evaluating a simple file access function: note that after the final transformation the function is now an ideal candidate for inlining, permitting simplification of other functions.

One of the first consequences of partially evaluating with respect to disk contents in particular is likely to be the elimination of parsing work. Not only do programs often read configuration data each time they run, but the data is often specified in a human-readable format for convenient editing. The program will then parse the file, storing interesting data in an internal format which is useful at runtime. Partial evaluation with respect to that file's contents could perform the parse operation at specialisation time, storing the configuration data in the format actually required at runtime and thus effectively compiling the configuration. This will be most effective if the parse is expensive (perhaps because we're using a very general library like an XML parser), or only a subset of the file's information is retained at runtime.

If eliminating the parse turns out to be difficult due to dependence on unknowns such as user input or random numbers, partial evaluation can at least eliminate expensive disk operations. Many programs start by reading configuration data from several possible stores, such as files in `/etc`, reading dot files in the current user's home directory, or consulting with a persistent configuration daemon such as gconf. Even if the parse can't be decided, storing these as constants rather than external files could subsume existing techniques that minimise seeks by colocating programs and their data on disk.

If on the other hand the parse can be eliminated, then the most important benefits are likely to come from improvements in hot functions which make reference to the processed file data, rather than from eliminating the parse itself. The former could speed up the entire program, whereas the latter is likely to be limited to speeding up startup.

We have written a simple prototype which evaluates programs with respect to specified file contents, making a start towards realising these benefits.

4 An LLVM Partial Evaluator

We base our partial evaluation system on the LLVM compiler infrastructure [9] because it already possesses a wealth of analysis and optimisation passes, designed for use in an optimising compiler but equally useful in a partial evaluator. Our design works by *interposing* upon system calls, replacing them with userspace implementations with may implement the call or defer to the real system call. Interposed `open` calls whose filename can be statically determined are translated to symbolic file descriptors which are propagated through the program using ordinary constant propagation techniques. When a known file descriptor reaches a `read` call, the call is replaced by a memcpy from a static string. Because other LLVM passes understand the `memcpy` intrinsic they are able to optimise the rest of the program much more aggressively than for a `read` call.

By itself this mechanism for forwarding constants struggles with typical filesystem code, as a `read` call is frequently part of a loop body, and the symbolic file descriptor includes the current file offset. This means that the file descriptor reaching a read call is often overdefined, being derived of a fresh file descriptor coming from the loop entry edge and a file descriptor with unknown position coming from the loop's back-edge.

However, the presence of a fake file descriptor provides a strong hint that the loop should be *peeled* – a technique which precedes the loop with a copy of its own body which is not part of the loop. The peeled body may be statically evaluated, in part or in its entirety, providing a cue to peel the loop again. Iterating this procedure can effectively statically perform the complete reading of the file. This technique in effect performs a path-sensitive constant propagation, in which a function is re-analysed per potential control flow.

On a similar basis the presence of an `open` call can provide a strong hint to inline the containing function, because this is likely to inline a generic file-handling routine and so concretise the name of the file being dealt with. The same goes for functions containing `read` calls – they're likely to yield useful information because inlining is likely to reveal the file descriptor they're operating

on. This is analogous to performing a context-sensitive analysis, in which a function is analysed once per call-site. Combining the two techniques of adaptive peeling and adaptive inlining yields both path- and context-sensitive analysis.

4.1 Results

To date this work is able to fully reduce simple programs which perform tasks such as counting occurrences of a certain character within a statically named file. This is performed non-adaptively, expanding loops and inlining functions exhaustively, and requires LLVM bitcode available at specialisation time. Figure 2 shows the results of applying this prototype to a simple program that counts occurrences of a given character in a file. The same program was compiled with and without userspace interposition on its system calls, with and without classical optimisation by LLVM, and finally using our partial evaluator. We show results for the program at different read buffer sizes in order to give an idea of the benefits of optimisation and of partial evaluation for programs with differing ratios of kernel-mode to user-mode execution time, with the left of the graph representing a typical program that spends most of its time in the kernel, whilst the right-hand side mostly runs in user mode, as it only makes a system call every 1024 bytes. The results show that for system call-heavy programs, simply executing calls in userspace (termed "usercall" on the graph) yields a 5x speedup. Here the `open` and `read` system calls have been patched using stubs which mock up a file descriptor table and provide data from a static array. These stubs are available to LLVM's optimisers, so classical optimisation is able to offer a further 4x (optimised versions are tagged "w/O3" on the graph). The system call variants are much less amenable to optimisation because the optimiser assumes that the call has arbitrary side-effects. When run with large buffer sizes the cost of computation comes to dominate and the four approaches have similar costs. The partial evaluator, meanwhile, is able to completely eliminate the program; it consists of a single `return` instruction and so has constant runtime.

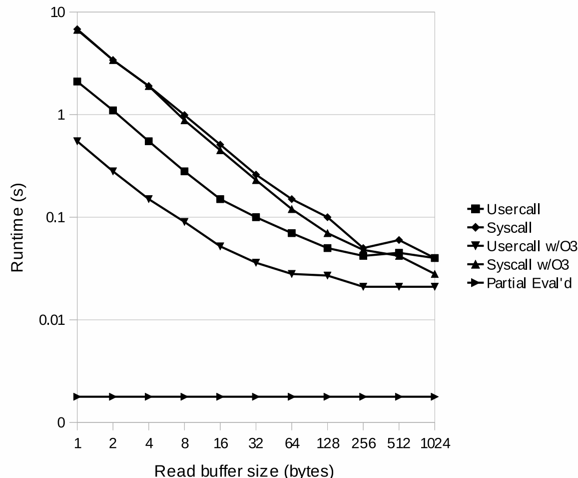

Figure 2: Runtime of a simple program counting a specified character in a file compiled with differing read buffer sizes. See section 4.1 for label definitions and discussion.

5 Future Work

The partial evaluator we have developed so far realises only a tiny fragment of the possibilities for speeding up day-to-day program execution by eagerly pushing known facts about the world.

A key implementation challenge will be automating the re-specialisation process. For on-disk files it will be easy to determine when we should re-specialise using file-watching APIs such as `inotify`. Monitoring other sources of information such as IPC will be more difficult, and may mean instrumenting IPC producers. A further challenge will be dealing with changes to file subject to specialisation whilst a consumer program is running; techniques permitting hot replugging of specialised or optimised functions, used in both previous specialisation work [13, 15] and in JIT compilers [11], may prove useful here.

Another challenge will be determining when partial evaluation is likely to be profitable, both in terms of effort invested at specialisation time vs. gains at runtime and in terms of whether specific code transformations are likely to accelerate the program at runtime. Existing compiler infrastructures [9] (and previous partial evaluators [1]) typically use crude, conservative heuristics, but alternatives include profiling and detailed system modeling.

In the long term we plan to investigate other, more challenging sources of information. One important source is inter-process communication. As we mentioned above, many programs obtain world state by consulting with a persistent daemon such as `gconf`, `nscd` or an X server. In order to effectively integrate this state into a target program we will need to determine what the current state is, possibly by analysing socket or pipe-related calls in the server, before integrating the information into the client by partially evaluating with respect to pipe and socket calls similarly to our current treatment of VFS calls. The most obvious challenge here is that it is easy to determine when VFS results will change, and therefore when specialised programs will need re-specialising, simply by watching the file. It is harder to determine when IPC results will change. Conventional analysis could still help, however: if we are certain that

the server program listens on a given socket, and that we have found the listening routine, we can conclude that it will definitely write certain bytes to that socket on the same grounds that we could conclude that certain bytes would be written to memory. By employing LLVM's existing analyses and transformations, which are already bound to be conservative, we can inherit its safety properties.

Information might also come from remote servers. This is similar to IPC, but with the constraint that the source code of the remote service is not available for analysis. However, by writing a model implementation serving as a proxy sitting between the application we wish to specialise and the server it may still prove possible to integrate common results, using the code of the model implementation rather than the true server. The model can then asynchronously spot when the server's response changes and cue the re-specialisation of the client program. This is essentially the imposition of a caching proxy taken to its logical extreme in which the cached result is used to pervasively optimise the client. This will complicate the procedure required when the server's response changes; however existing dynamic re-specialisation techniques should be applicable.

The most challenging information comes from user input. Naturally there will always be a limit to program specialisation imposed by user input which cannot be known in advance or other unpredictable sources of information such as the time of day or a random number generator. However there are still routes to program improvement despite this lack of knowledge. Firstly we could profile the program to determine likely inputs and generate *guarded specialisations*. These are optimised based on suspected rather than known data, and are guarded by checks that the input does in fact meet required preconditions. Secondly we could use either profile data or ordinary static analysis to determine profitable routes for *speculation*. This would mean finding program points at which a branch based on unknown data leads to a block of expensive computation without further reference to unknowns; effectively an analysis similar to that investigating the profitability of loop peeling: one which asks the question *"if we knew X, how much work could we statically perform?"* Existing work has looked at speculating ahead of both system calls [4] and user input [8]; by employing extensive static analysis we could improve on this with better *informed speculation*.

6 Conclusion

We have proposed that we should employ pervasive specialisation: the aggressive use of seldom-changing facts about the entire system's state to maximally specialise programs to the current state of the world, and some of its inherent challenges. We have also described a simple prototype implementation. In future work we will extend its scope to fully realise the possibilities of whole-system specialisation.

Acknowledgements: We are indebted to the workshop reviewers for their comments on our work.

References

[1] ANDERSEN, L. O. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, 1994.

[2] BERLIN, A., AND WEISE, D. Compiling scientific code using partial evaluation. *Computer 23* (1990), 25–37.

[3] CONSEL, C., HORNOF, L., MARLET, R., MULLER, G., THIBAULT, S., AND VOLANSCHI, E.-N. Tempo: Specializing systems applications and beyond. *ACM Comput. Surv. 30*, 3es (1998), 19.

[4] FASER, K., AND CHANG, F. Operating system I/O speculation: How two invocations are faster than one. In *USENIX Annual Technical Conference, General Track* (2003), pp. 325–338.

[5] JONES, N. D. An introduction to partial evaluation. *ACM Comput. Surv. 28* (September 1996), 480–503.

[6] JONES, N. D., SESTOFT, P., AND SØNDERGAARD, H. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation 2*, 1 (1989), 9–50.

[7] KROAH-HARTMAN, G. *Linux Kernel in a Nutshell*. O'Reilly Media, Inc., 2006.

[8] KROEGER, T. M., AND LONG, D. D. E. Predicting future file-system actions from prior events. In *USENIX Annual Technical Conference* (1996), pp. 319–328.

[9] LATTNER, C., AND ADVE, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. International Symposium on Code Generation and Optimization* (2004).

[10] MUTH, R., DEBRAY, S., WATTERSON, S., BOSSCHERE, K. D., AND INFORMATIESYSTEMEN, V. E. E. alto: A link-time optimizer for the compaq alpha. *Software - Practice and Experience 31* (1999), 67–101.

[11] PALECZNY, M., VICK, C., AND CLICK, C. The Java Hotspot server compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1* (Berkeley, CA, USA, 2001), JVM'01, USENIX Association, pp. 1–1.

[12] PARTL, T. O. H., HYNA, I., AND SCHLEGL, E. The not so short introduction to Latex 2e.

[13] PU, C., AUTREY, T., BLACK, A., CONSEL, C., COWAN, C., INOUYE, J., KETHANA, L., WALPOLE, J., AND ZHANG, K. Optimistic incremental specialization: Streamlining a commercial operating system. In *Symposium on Operating Systems Principles (SOSP), Copper Mountain* (1995), pp. 314–324.

[14] RAJAGOPALAN, M., PERIANAYAGAM, S., HE, H., ANDREWS, G., AND DEBRAY, S. Binary rewriting of an operating system kernel. In *Proc. Workshop on Binary Instrumentation and Applications* (2006).

[15] SHANKAR, A., SASTRY, S. S., BODÍK, R., AND SMITH, J. E. Runtime specialization with optimistic heap analysis. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2005), OOPSLA '05, ACM, pp. 327–343.

[16] SMALLEY, S., VANCE, C., AND SALAMON, W. Implementing SELinux as a linux security module. Tech. rep., 2002.