Can Computers be Programmed Productively in the Post-Dividend Era?

Rastislav Bodík, Justin Bonnar University of California, Berkeley
Doug Kimelman IBM T.J. Watson Research Center

We hold the future in our hands. If Bell's Law is correct, continuing miniaturization will further shrink the personal computer. Handhelds will bring new usage modes, making laptops obsolete.

This transition to a new computer class has been expected; what has not been anticipated is that Moore's Dividend has run out. Exponential hardware speedups have disappeared, leaving handhelds too power-constrained to run programs written in productive languages. The need to close this performance gap gives the programming languages community the opportunity, perhaps for the first time ever, to dramatically accelerate the ubiquity of a new computer class.

The new usage modes suggest that many new applications will be written. The desire to accelerate their development motivates our technical challenge — making programming efficient handheld applications productive.

On the laptop, the roots of productive programming can be traced to 1996 when the dynamically typed JavaScript language was added to the web browser. Soon, programmers were enjoying the convenience of the flow layout engine and high-level frameworks. The browser grew successful by spending Moore's Dividend, ultimately enabling Web 2.0. In this post-Dividend era, it will be sustained by clever software engineering and familiar compiler optimizations, which will be sufficient thanks to laptops relative abundance of CPU cycles.

In contrast, evolutionary improvements alone may not enable productive programming on the handheld. The tiny battery and lack of cooling restricts the handheld CPU to about 0.5W, which makes it 12-times slower than a 20W laptop CPU. Because browsers are CPU-intensive, they run about 7-times slower on a handheld. As a result, a typical web page renders in 15 seconds, and that's too slow for handhelds to break through as a new computer class.

To counteract the 12x performance penalty, we sacrifice productivity. While laptop programmers develop for the browser, handheld programmers reach for somewhat less productive frameworks such as Java-based Android, or by escaping to C on the iPhone. Historically, handhelds are the first emerging class that is unable to use the best software tools of its predecessor.

Hardware will continue to improve, of course, but at a much slower pace. While future transistors will switch faster, on-chip heat density will keep clock speeds constant, leaving us with improvements in energy efficiency instead. These improvements will eventually lead to fast handheld processors, but only when today's 20W laptop CPU can be run at 0.5W. However, this is predicted to take twenty years (see Section 6.2.1 in the Exascale report) – four-times longer than it took to bridge the same 12x performance gap during the 90's.

It is tempting to address the problem by making the handheld a thin client connected to the cloud. Unfortunately, there are fundamental limits to what we can offload. First, the latency of wireless communication will often exceed the 100ms user interface perception threshold. Second, radio communication may consume more energy than client-side computation. Finally, disconnected operation, likely to remain a problem, precludes relying on the cloud as a co-processor. The bottom line: the handheld must be as standalone as the laptop.

Making handhelds productively programmable involves other problems, such as designing new domain abstractions, but we focus here on the problem of efficient execution of high-level abstractions. Our goal is to reduce the *abstraction tax*, or the runtime overhead of supporting productive abstractions. Is the abstraction tax large enough to merit our attention? An experiment by Chris Jones *et al* showed that a browser application using the Google Maps API is about 100-times slower than its C counterpart. It is revealing to realize that only one in a hundred instructions delivers essential functionality, while the rest merely build the "scaffolding" for convenient software engineering.

There are many ways to make high-level programs more efficient, but many of us in academia and industry are betting on two technologies in particular: parallelization and specializing virtual machines. Parallelism is popular because scaling in energy efficiency (about 25%/2 years) will double the number of cores roughly every four years. Specialization is popular because it eliminates repeated sub-computations; presumably the abstraction tax scaffolding is among them. These techniques are technically sound but the goal of this paper is to point out their inherent limitations and ask whether they are the best directions for our research.

To evaluate their potential, we need performance metrics; the most important on the handheld are *responsiveness* (speed of the computation) and *battery life* (energy efficiency of the computation). Perfect *parallelization* over *n* processors improves responsiveness *n*-times. The total work remains unchanged, so the battery life is not improved. *Abstraction tax reduction* in general, and specialization in particular, improves responsiveness as well as the battery life. If the program is sped up *m*-times, both are improved *m*-times because the program performs *m*-times less work. Let us also consider *voltage scaling*, which improves battery life at the cost of responsiveness, and can be used to translate the speedup of parallelization into an improvement in battery life. The idea is to slow down the clock frequency and correspondingly also the supply voltage, improving energy efficiency; the improvement is linear up to 2- to 3-fold frequency reduction after which returns are diminishing.

After we plug in the constants, we see this quantitative lesson: Considering the large (100x) abstraction tax and the modest number of cores (about 10 in 10 years on the handheld), it is more profitable to aim for a 10-fold tax reduction than at 10-fold parallelization. Both appear feasible to achieve, but compare their benefits: assume that rendering of a web page takes r=10 seconds and consumes so much energy that it allows b=1 hour worth of web page downloads. Achieving 10-fold parallelization improves responsiveness to an acceptable r=1 second but the battery life stays at the unacceptable b=1 hour. To improve the battery life, let us distribute the benefits of parallelization with 3x voltage scaling, obtaining r=3sec and battery life b=3hours. In contrast, 10-fold tax reduction yields r=1 second and b=10 hours. The difference becomes clearer once we realize that the speedup of tax reduction, s_t=10, roughly equals the square of the parallelization benefit, s_p≈3, that is $s_t \approx s_p^2$. Combining the two compounds the benefits, of course; parallelization should perhaps be applied once the code has little abstraction tax.

Abstraction tax reduction is effective, but can it be performed automatically? Specialization, also known as partial evaluation, is a powerful transformation shown to be capable of compiling away interpretation overhead. It does so by statically evaluating computations observed to depend only on runtime-static values. Recently, specialization has gained new power through embeddings into dynamic optimizers, leading to hopes that modern VMs will remove most of the abstraction tax. We have examined three tax-heavy scenarios from the web browser and show that specialization is unlikely to be as profitable as we might hope.

Passing arguments as strings. When a JavaScript program wishes to modify the style attribute of a HTML element, it does so by communicating with the layout engine through string values, for example with the expression `tile.height = x + "px"`. This interface is convenient because programmatic manipulations use the familiar (textual) syntax of CSS stylesheets. However, the interface is inefficient because it converts the integer value `x` into a string and concatenates `"px"`, only to perform the inverse operation within the layout library. Unfortunately, when the value `x` is not constant the string manipulations cannot be specialized away. Furthermore, half of this inefficient computation occurs inside of the C++ layout library, outside of the reach of a VM-resident specializer. Hence, even a perfect specializer would be quite limited: because browsers spend only 15% of their time executing JavaScript, it would reduce the abstraction tax from the current 100x penalty to 85x. The lessons are that (i) the tax of a productive construct may be spread across components written in different languages; (ii) the specialization of the tax requires semantic reasoning that is higher-level than the classical constant propagation based on runtime static values.

Page layout. Web applications create their user interface by manipulating a document tree, which the browser then lays out and renders. While convenient, the indirection is much less efficient than drawing to the screen directly, particularly for animations and other dynamic HTML effects that repeatedly modify the document causing frequent re-layouts. Because most of the document does not change between re-layouts, specialization should be able to reduce the tax of interpreting the tree. Again, this is hard to do, at least on existing implementations of layout engines. First, consider the calculation $\texttt{top}_k\texttt{=((top}_1\texttt{+height}_1\texttt{)+height}_2\texttt{)+}$ $\texttt{…+height}_{k-1}$ which computes the x-coordinate of the kth stacked element. Assume that only $\texttt{height}_1$ is dynamic. While only one addition should be needed at runtime, all of subcomputations are marked dynamic. It may seem trivial to reassociate this computation and specialize them away, but the subexpressions are spread across the document tree, possibly obscured by a complex code base. Furthermore, the intermediate results may be needed to render the other stacked boxes, preventing any expression reassociation. If we are unable to specialize coordinate calculations, can we at least specialize away the tree traversal, so that unchanging subtrees need not be revisited? In principle yes, but programs frequently modify the document structure, contaminating layout calculations up to the root node. Finally, we need to know which portions of the document are static, but applications provide no hints as to what changes are going to occur next. Solving this problem seems to require a reformulation of the layout process as well as the interface with the scripting language.

Specializing embedded DSLs. Dynamic languages like JavaScript enable the creation of high-level frameworks/DSLs, such as the jQuery language for document manipulation and animation. These productive languages incur significant interpretation overhead, which specialization could in principle remove. The challenge is illustrated by the following idiomatic example: `loop { nodes = selectnodes(tree); …modify tree…; }`. Often the value of `nodes` in successive iterations remains constant even though the tree has been modified; we would like to specialize away subsequent calls to `selectnodes`. However, doing so requires proving that changes to the tree did not affect the set of selected nodes. A specializer that reasons about the static nature of individual memory locations does not seem powerful enough. A more tractable approach seems to involve raising the level for reasoning about animation abstractions with declarative primitives.

It seems that we currently cannot offer programming techniques that are both productive and efficient enough for handhelds. Parallelism does not sufficiently improve battery life and classical specialization reasons at a prohibitively low-level of abstraction. Our recommendation is to raise the level of efficient abstractions, and do so with compilers rather than with layers of unspecializable interpreters. It is also interesting to realize that the ML-family languages are efficient and productive in the hands of experts, but have not been widely adopted. Designing an approachable language that is both efficient and productive remains an open problem.

We wish to thank Elad Alon, Krste Asanovic, Joel Galenson, James Ide, Chris Jones, Tsu-Jae King, David Mandelin, Leo Meyerovich, and the reviewers.