Virtual Machine and Bytecode for Optimization on Heterogeneous Systems

Kerry A. Seitz, Jr. and Mark C. Lewis
Department of Computer Science
Trinity University
San Antonio, Texas 78212-7200
Email: {kseitz, mlewis} @ trinity.edu

***Abstract*—We present a description of a virtual machine and bytecode that have been designed around the goal of optimized execution on highly variable, heterogeneous hardware, instead of having goals such as small bytecodes as was the objective of the Java Virtual Machine. The approach used here is to combine elements of the Dalvik virtual machine with concepts from the OpenCL heterogeneous computing platform, along with an annotation system so that the results of complex compile time analysis can be available to the Just-In-Time compiler. We provide a flexible annotation format so that the set of annotations can be expanded as the field of heterogeneous computing continues to grow.**

Keywords-heterogeneous computing; annotation; JIT; OpenCL; Dalvik

I. Introduction

Computing hardware is in the middle of a dramatic shift from the single-core, generally homogeneous model that dominated prior to roughly 2005 to a many-core, generally heterogeneous future. This shift began as a simple move from single-core to multi-core processors. More recently, many-core processors in the form of Graphics Processing Units (GPU) from NVIDIA® and AMD®, as well as Many Integrated Core (MIC) Architectures from Intel® have moved more into the mainstream. This move is probably best illustrated by the line of Fusion® chips from AMD that include GPU circuitry on the same piece of silicon as the CPU. The ubiquitous nature of this development is illustrated by the fact that the first Fusion chips to be released were intended for laptops, not workstations or servers.

In many ways, hardware has gotten ahead of software. Most developers are still used to working in a single-threaded, homogeneous environment. Not only do we have the legacy of developers being trained for that world, but also most of the languages and tools that are in use were designed for that environment. Tools and languages are improving, but this type of development is harder and more costly than single-threaded development.

If development is not yet prepared for multi-core environments, it is certainly far from ready to tackle the challenges of many-core, heterogeneous environments. There are standards for doing this type of programming such as OpenCL® and CUDA®. However, these standards force the programmer to interact with the machine at a low-level, which tends to increase development cost and require more effort from the programmers. To make these systems more accessible, we need higher level tools that allow the developer to spend his/her time focusing on the logic of the application instead of the details of the platform on which it will be running. This is likely to become increasingly important as the options for platforms grow. If making a portable program requires separate development for the different options from Intel (CPU, CPU/GPU, MIC), AMD (CPU, Fusion, GPU), NVIDIA (GPU, Tegra), and the host of other combinations that are likely to appear in the near future, then producing optimized code that can cover the various options will be prohibitively expensive.

The productivity of developers on single-threaded environments was greatly assisted by the development of platforms that support higher level programming in a manner that is generally machine independent. For example, the Java® Virtual Machine (JVM) environment allows programmers to compile to bytecode which is then optimized by a Just-In-Time (JIT) compiler. Unfortunately, these environments were made with design goals that did not include optimizability or support for many-core, heterogeneous environments. Indeed, in the case of the JVM, the design goals were small bytecodes and fast emulation. As a result, it uses a stack based architecture which causes some problems for producing optimal performance even on single-threaded hardware [1], [2].

With all of these things in mind, we have set out to develop a bytecode that has a primary design goal of being able to support optimization for performance on heterogeneous platforms. Our approach is to borrow ideas from one of the newest register based virtual machine platforms, Dalvik®, and combine those with aspects of the OpenCL standard for heterogeneous computing.

II. Related Work

Arguably the biggest goal of heterogeneous computing is increasing code efficiency. Almost all modern compilers perform optimizations to increase code performance, but these optimizations sometimes slow down the compiler. Dynamic compilation and optimization that occurs in a virtual machine (VM) must execute quickly, otherwise the time spent optimizing might negate the speed-ups provided by the optimizations. In order to aid the VM's optimization efforts, Krintz and

Calder developed an annotation framework for the JVM to reduce compilation overhead so that complex optimizations can more efficiently be performed in the VM [3]. Vallée-Rai et al. found that because JVM class files can often be optimized statically, adding annotations to these files may reduce the number of optimizations that the JIT compiler must perform, thereby allowing the compiler to focus on more expensive optimizations. Furthermore, some runtime checks, such as certain array index bounds checks, can be performed at compile time. If these checks are performed at compile time, annotations can be used to forgo the checks at runtime [4].

The idea of providing more high-level concurrency support in VMs is not a novel concept. Marr et al., recognizing the transition to many-core processors, argued that VMs need to abstract concurrency to take advantage of this shift. In addition, Marr et al. surveyed 17 VMs to determine the current support for concurrency models and found only limited explicit concurrency support [5]. While some current VMs support models of concurrency on multi-core processors, few provide abstractions for running code on GPUs. Peercy et al. developed a VM for GPUs focused on performance. The Data Parallel Virtual Machine (DPVM) that they developed provides a simplified method to write code for GPUs yet still reveals low-level functionality for when it is needed [6].

Another example of a VM that utilizes the GPU is GViM, "a system designed for virtualizing and managing the resources of a general purpose system accelerated by graphcis processors." Although GViM incurs a performance cost over non-virtualized solutions, the flexibility and reduction in programmer effort resulting from this VM model helps to justify the virtualization of hardware other than simple CPUs [7]. Indeed, virtualized execution will likely always perform worse than code written and compiled for specific devices. If a programmer knows the intricacies of the hardware for which he/she is writing code, then he/she can write extremely efficient code. However, we suspect that very few programmers know these minute details, and likely even fewer are willing to spend the time and effort to write fine-tuned code for perfectly efficient execution. Doing so would substantially increase development costs, and for most applications, the effort is not worth the result. Moreover, even when the most efficient code is written, this code is not portable to other devices. By compiling source code to a bytecode and allowing a VM to optimize the code at runtime, we simultaneously get portability and efficiency, while allowing programmers to write in high-level languages without concern for low-level concepts.

While the examples provided above allow easier access to the GPU, they do not utilize code execution on both the CPU and the GPU. The MapCG framework provides a means to write code for both the GPU and the CPU using a high-level programming model. This framework allows programmers to write code in the high-level MapReduce framework [8]. The MapCG runtime then compiles the source code for execution on the CPU and the GPU [9]. Along with providing access to both the CPU and the GPU in a high-level environment, this framework also allows for code portability to other CPU and GPU architectures.

New technology is constantly evolving, and predicting the next big advancement in hardware development is challenging at best. Therefore, we need systems that can adapt to new technology, while maintaining backwards compatibility with older code. Devices such as field-programmable gate arrays (FPGA) provide a promising avenue for advancement in hardware efficiency [10], and we would like to have old code utilize this hardware as effectively as possible without requiring rewrites or recompiles from source. In a world where computers are becoming more heterogeneous, we need a way to provide programmers with a programming model that is independent of the hardware. The key to this model is virtualization [11]. Because technology is ever-changing, we present a flexible model for storing optimization information that can be easily adapted as new technology and techniques arise.

III. Dalvik

The Dalvik Virtual Machine is an integral part of Google's Android® operating system. Code for the Dalvik VM is typically written in Java, but the Dalvik bytecode differs significantly from the Java bytecode. First and foremost, the Dalvik VM is register-based, rather than stack-based like the JVM [1], [12]. A register-based VM and bytecode offer a few advantages over stack-based architectures. Because modern CPUs are register-based, this type of bytecode more closely mimics the hardware on which it is running. Therefore, if the number of registers implemented in the VM matches the number of registers in the actual hardware, then register allocation can be performed during the original compilation rather than in the JIT. Furthermore, Shi et al. found that a register-based VM is faster than a stack-based VM when implemented as an interpreter [2]. Because many JIT compilers, such as Oracle's Hotspot VM, initially use an interpreter to decrease start-up time [13], Shi's findings suggest that a register-based bytecode will improve performance in modern VMs.

A register-based bytecode also has some disadvantages. Because it does not make any assumptions about the number of registers, a stack-based bytecode is simpler and possibly better suited for JIT compilation [2]. To mitigate these disadvantages, the bytecode presented in this paper is neither register- nor stack-based. Instead, it is more high-level in that it preserves the concept of variables. The bytecode is based on the Dalvik bytecode, but instead of referencing registers, our bytecode references variables. Type information is also preserved for these variables. Use of type information in low level languages has been explored in other contexts before [14], [15].

IV. OpenCL

The OpenCL standard provides a way to utilize any piece of hardware in the computer, as long as the appropriate drivers are installed. A program written in OpenCL can query the machine to determine which devices are available and then select the device or devices best suited for running each section of code. Because this functionality is in line with our goal of exploiting a heterogeneous environment, we chose to implement the virtual machine for this bytecode using OpenCL.

While the bytecode does not contain any instructions specifically for the OpenCL implementation, the OpenCL computation model influenced some of the information preserved in the bytecode from the original source code. This additional information is stored in annotations, and provides optimization hints to the JIT compiler. An OpenCL program consists of two distinct sections: the host and the kernels. The host serves only to set up kernels and queue them for execution, while the kernels are where the real computation happens. Part of the host set up includes querying for and deciding which devices the program will use [16]. This execution model works well with annotations that specify on what type of device a particular block of code will execute most efficiently (e.g., executing a data parallelizable block on a GPU).

Another annotation that OpenCL helped to motivate is an annotation that states the variables used in a certain section of code. Because the OpenCL memory model requires that data be copied to each device where it is needed, this annotation allows the VM to optimize memory moves to different devices. Moreover, because memory move operations are computationally expensive, annotations such as whether an object is immutable or functionally immutable will tell the VM whether certain memory moves are actually necessary.

V. Basic Assumptions

When developing this bytecode, we made several assumptions about its future uses. Unlike the Java and Dalvik bytecodes, we did not prioritize making the bytecode small. As the cost of memory continues to decrease, the size of compiled bytecode will become less significant. As such, and because a larger bytecode can preserve more information important for optimizations, we decided to focus on providing each instruction with relevant information rather than worry about bytecode size.

In addition, we decided that most of the annotations should be designed so that they can be ignored when executing the code. Thus, the JIT compiler can initially disregard most annotations in order to decrease start-up time, then later recompile code sections using the annotations to improve performance. Certain annotations, such as the annotation preserving type information when type erasure would normally occur, may or may not be ignored, depending on the original source language (for example, a language assuming type erasure, such as Java, can ignore these annotations until optimization).

VI. Bytecode Design

A. *Instructions*

The bytecode contains two distinct types of information: the instructions and the annotations. The instructions are the actual operations that must be performed when executing the code. Each instruction is 128 bits long. The first 16 bits denote which instruction to perform, and the last 16 bits are reserved for annotations associated with the instruction. The other bits are used to store references to variables, type information, or literal values, as determined by the individual instructions, using 16 bits for each variable or type and groups of 16 bits for literals. Figure 1 shows a pictoral representation of the instruction format.

Instructions refer to variables by using a 16 bit integer. Each stack frame contains a new set of variables. Each variable also has a type associated with it, which is specified using a 16 bit reference when the variable is initialized. The first eight types are reserved for the primitives that are built into most modern hardware (boolean, char, byte, short, integer, long, float, and double). The rest of the types are defined by user and language/library defined types. The use of a 16 bit field to represent a type has already been established in the JVM and Dalvik; however, associating annotations with variables and types has only been explored in a limited context [1], [12].

Along with type information for individual variables, the types used in generics will also be preserved in the bytecode. This information will be located in annotations to variables that take a generic parameter. Thus, unlike the JVM, this bytecode will not be limited by type erasure.

Like Dalvik's dex files and the JVM class files, this bytecode also has type files which contain methods and variables associated with objects. The headers for these type files are similar in form to the headers used in Dalvik and the JVM [1], [12]. However, the headers also contain annotation information that pertains to the type as a whole. In addition, annotations will accompany the fields and methods to provide specific details for each. For example, an object may or may not be immutable as a whole, so this information is stored in the header. In mutable objects, methods that do not change the object can be classified as non-mutating so that if only these methods are used within a program, the object can be considered immutable and take advantage of optimizations associated with immutable objects.

When a function call instruction executes, a few key steps occur. First, the stack frame shifts so that the function will have a new set of variables. The first n variables contain the arguments to the function, where n is the number of arguments to that function and $n \leq 4$. If a function requires more than four variables, the first three are passed using variable indexes, and the remaining variables are stored in memory through instructions preceding the function call. The address to this memory location is then stored in a variable and passed to the function in the fourth argument slot. The function is responsible for extracting the variables from memory as needed, using bytecode instructions inserted by the compiler.

The instruction format is rather long, and many instructions will not use the full 128 bits available. For some instructions, the bytecode could be modified to contain variations that perform multiple operations on different sets of data. This idea echoes the sentiments of the Very Long Instruction Word (VLIW) architectures [17]. For simplicity, we chose not to include these types of instruction in the current bytecode design.

instruction	var1	var2	var3	var4	var5	var6	annotation

Fig. 1. The basic format of the instruction for our bytecode. Each box represents 16 bits. Some vars may be unused in certain instructions. The var slots may also be replaced by types or literals, depending on the instruction. See the **Sample Instructions** section for examples of different instruction types.

B. Annotations

The annotations in this bytecode come in two forms. The first form is a set of annotations that are expressed in individual instructions, referenced by a 16 bit integer in the last 16 bits of the instruction. The meaning of the annotation depends on the instruction with which it is associated. For some instructions, each combination of 16 bits will represent a unique annotation. For others, however, the 16 bits will serve as bit flags so that multiple pieces of information can be associated with a single instruction. For example, knowing whether a variable is constant, escapes the current thread, and/or escapes the current stack frame provide important optimization information, so the 16-bit annotation for the variable declaration instruction is implemented as a set of bit flags. Thus, a variable can be flagged with all three of these annotations, and more, within the declaration instruction itself.

The other annotation form in this bytecode is the annotation file. This file contains all of the annotations associated with the code other than the annotations specified in the last field of the instructions themselves. A special annotation instruction is used to reference these annotations, and these instructions are inserted by the compiler at the appropriate locations to specify information for potential optimizations. The annotation instruction uses the last 16 bits to describe the annotation type. The 64 bits after the instruction field encode an offset in the annotation file for the full annotation data. The location in the annotation file must contain the length of the annotation followed by the specific information for that annotation. The format of the information is defined by the annotation type that was in the bytecode instruction. Each type of annotation can have an entirely different format. This flexible annotation format allows for new annotations to be easily added in the future. The creators of those annotations can decide how best to arrange the information associated with them. If a VM identifies an annotation that it is not aware of, it will simply ignore the annotation. Thus, programs compiled with new annotations are backwards compatible with older VMs, but some potentially useful optimization information will go unused.

VII. Sample Instructions

Page limit restrictions prevent the entire set of bytecode instructions from being reproduced in this paper. Instead, we have included a few representative instructions to demonstrate the general format. Empty boxes indicate unused fields. Unless otherwise stated, all fields in the following figures are 16 bits.

The first group of instructions is shown in Figure 2 and deals with variables and object creation. As with most instructions, after the 16 bit op-code, the next 16 bit field stores a destination register. The third 16 bit field encodes a type, except in the assignments, where the type will be known from the initialization of the destination variable. The variable creation instructions then provide an initial value, either from a source variable or from a literal that can be up to 64 bits. A type casting instruction also specifies a source variable for the cast as the static types of the source might not be valid for the cast.

The annotation segment in each instruction uses a full 16 bits to provide for future additions; program analysis is an area of significant current work, and new languages might have constructs that provide particularly useful information for optimization. In some cases, there are clear and obvious uses for some of the bits. For variable declaration, a bit is reserved for telling if the variable is constant after declaration. For object creation, two bits are reserved for the results of escape analysis to see if the object can escape the current stack frame or the current thread. This type of analysis can allow for stack allocation of objects or the removal of synchronization code [18].

In Figure 3 you can see the samples of numeric operations for unary and binary operations. The binary operations come in forms where both operands are variables as well as a form where one operand is a literal.

Figure 4 shows the different array instructions. While arrays can be modeled in languages as data types that sit at the library level [19], the bytecode needs to have a way to set aside and access larger chunks of continuous memory. The virtual machine implementation is given freedom in how it stores objects in an array.

The last group of instructions shown is in Figure 5 where you can see the flow control instructions. These include required and conditional branches as well as method invocations. The branch to a specific instruction will be the most commonly used form, but branching to variable locations is also allowed to provide greater flexibility (i.e., replacing the *dest* literal with a variable that contains the address to which to jump).

VIII. First-Cut Annotations

The cornerstone of the proposed VM design is that compilers can do costly and complex analysis to discover information that is useful for optimization, but that the information will only be useful if it is retained in some form in the bytecode so that it is available to the JIT. Some of this information can be placed in the instructions themselves, as noted in the previous section. However, additional flexibility is provided through a separate annotation file and instructions that can reference into it. In this section, we consider some of these annotations.

At the class level, there are obvious advantages for annotations that tell if a given class is immutable or functionally immutable. An immutable class is one for which the values stored in instances can not change in any way after instantiation. A functionally immutable class relaxes this

Instantiates variable *destVar* with the value from *sourceVar*

init-var	*destVar*	*type*	*sourceVar*				annotation

Instantiates variable *destVar* with the value of *literal*

init-var-lit	*destVar*	*type*	*literal* (64 bits)				annotation

Creates a new object of type *type*, storing a reference to it in *destVar*

obj-create	*destVar*	*type*					annotation

Fig. 2. This is a list of the instructions related to variable and object creation and assignment.

Unary operators

neg	*destVar*	*sourceVar*					annotation

Binary operators on two variables. The *op* includes add, mult, etc. The *type* is an appropriate primitive type.

op-type	*destVar*	*sourceVar1*	*sourceVar2*				annotation

Fig. 3. These are the basic math instructions that are part of the bytecode.

Creates a new array of size *sizeVar* and type *type*

new-array	*destVar*	*type*	*sizeVar*				annotation

Stores *sourceVar2* in array *destVar* at position *sourceVar1*

array-put	*destVar*	*sourceVar1*	*sourceVar2*				annotation

Fig. 4. These are the instructions that are intended to work with arrays in the VM.

Jumps to the bytecode instruction at location *dest*

goto	*dest* (64 bits)						annotation

Branch to *dest* if *sourceVar1* and *sourceVar2* compare as specified (e.g., eq, ne, lt, le, gt, ge)

if-**test**	*sourceVar1*	*sourceVar2*	*dest* (64 bits)				annotation

Invokes method number *methodNo* on object *objVar*

invoke	*objVar*	*methodNo*	*argVar1*	*argVar2*	*argVar3*	*argVar4*	annotation

Fig. 5. These are the instructions that support flow control.

restriction so that any changes are not visible to outside code by changing the results of method invocations. An instance of a functionally immutable class might have a mutable field that is not directly accessible from the outside and which is used only for optimization. Instances of such a class are not as thread safe as a truly immutable class, but they can be safely copied and distributed without needing to communicate changes back to the original. This fact is particularly beneficial in a heterogeneous environment where devices often have separate memory spaces.

The class level annotations can also include the results of analyses such as shape analysis [20]. This annotation can provide the VM with information that is significant for memory allocation or the use of preloading instructions to prevent pipeline stalls. The memory allocation aspect is particularly useful in heterogeneous systems because many devices do not support dynamic memory allocation or have cached memory hierarchies that are standard in CPUs.

Another use of annotations is the specification of full types for generics/parametric types. This information has to be stored in annotations because the types can be arbitrarily long. The optimization benefits of having access to full types is well known from their use in C++ in techniques such as expression templates [21]. Putting the basic types that result from type erasure into the primary bytecode allows the JIT to get running quickly using the limited space in standard instructions. The complete types will be available in annotations for optimizations or for languages that allow run time matching on those types.

The true power of allowing arbitrary annotations in a separate file is realized when we look at the impact on heterogeneous computing. This is an area that is still in rapid development and will inevitably experience significant change in the coming years. Annotations aimed at heterogeneous computing specify when blocks of code are particularly well suited for being run as kernels or when they would work particularly well on certain types of devices.

One of the other major benefits of JITs is that they can take runtime information into account. This feature is particularly significant in parallel and heterogeneous processing. Deciding whether a particular process should be split across threads or run as a kernel on a particular device can often only occur at runtime when the value of some size parameter is known. Compilers can add annotations estimating how large a parameter needs to be in order for these approaches to be beneficial, but the decision for whether any particular call should be handled in one way or another requires knowing the run time values.

IX. Conclusions

In this paper, we have presented a bytecode and virtual machine design that prioritizes optimization in a heterogeneous environment. By storing annotations with the bytecode, the JIT compiler can make runtime optimization decisions, taking into account the devices available in the computer. This design provides code portability to different architectures and different combinations of devices, and because many optimization decisions are made at runtime, the application can adapt to the hardware and provide optimizations specific for each platform.

The field of heterogeneous computing is in a state of constant, rapid evolution. In order to incorporate new advances in this field, the annotation format for this bytecode is extremely flexible and designed for annotations to be added over time. New annotations can be formatted in a way that best organizes the information within, and because most annotations only provide optimization information, code compiled with new annotations is backwards compatible with older VMs. Preserving backwards compatibility is important for many applications, but this requirement should not limit future advances in optimizability.

In the future, we plan to implement the VM for this bytecode to execute code on both a CPU and a GPU. We will initially write an interpreter in OpenCL, with the intention of later using it with a JIT compiler. In addition, we will write a compiler to the bytecode for a simple high-level language in order to explore program analysis techniques and annotation generation. We also plan to design more annotations to add to the bytecode to improve optimization potential.

Acknowledgment

This work was supported by the Trinity Mach Fellowship.

References

[1] T. Lindholm and F. Yellin, *Java Virtual Machine Specification*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[2] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg, "Virtual machine showdown: Stack versus registers," *ACM Trans. Archit. Code Optim.*, vol. 4, pp. 2:1–2:36, January 2008. [Online]. Available: http://doi.acm.org/10.1145/1328195.1328197

[3] C. Krintz and B. Calder, "Using annotations to reduce dynamic optimization time," in *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, ser. PLDI '01. New York, NY, USA: ACM, 2001, pp. 156–167. [Online]. Available: http://doi.acm.org/10.1145/378795.378831

[4] R. Vallée-Rai, P. Lam, C. Verbrugge, P. Pominville, and F. Qian, "Soot (poster session): a Java bytecode optimization and annotation framework," in *Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)*, ser. OOPSLA '00. New York, NY, USA: ACM, 2000, pp. 113–114. [Online]. Available: http://doi.acm.org/10.1145/367845.368008

[5] S. Marr, M. Haupt, and T. D'Hondt, "Intermediate language design of high-level language virtual machines: towards comprehensive concurrency support," in *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages*, ser. VMIL '09. New York, NY, USA: ACM, 2009, pp. 3:1–3:2. [Online]. Available: http://doi.acm.org/10.1145/1711506.1711509

[6] M. Peercy, M. Segal, and D. Gerstmann, "A performance-oriented data parallel virtual machine for GPUs," in *ACM SIGGRAPH 2006 Sketches*, ser. SIGGRAPH '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1179849.1180079

[7] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan, "GViM: GPU-accelerated virtual machines," in *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, ser. HPCVirt '09. New York, NY, USA: ACM, 2009, pp. 17–24. [Online]. Available: http://doi.acm.org/10.1145/1519138.1519141

[8] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, January 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[9] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin, "MapCG: writing parallel program portable between CPU and GPU," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, ser. PACT '10. New York, NY, USA: ACM, 2010, pp. 217–226. [Online]. Available: http://doi.acm.org/10.1145/1854273.1854303

[10] M. Lin, "The amorphous FPGA architecture," in *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, ser. FPGA '08. New York, NY, USA: ACM, 2008, pp. 191–200. [Online]. Available: http://doi.acm.org/10.1145/1344671.1344700

[11] D. F. Bacon, "Virtualization in the age of heterogeneous machines," in *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, ser. VEE '11. New York, NY, USA: ACM, 2011, pp. 1–2. [Online]. Available: http://doi.acm.org/10.1145/1952682.1952684

[12] D. Bornstein, "Dalvik VM internals," in *Google I/O Developer Conference*, 2008.

[13] S. Meloan, "The Java HotSpot performance engine: An in-depth look," Oracle's Sun Developer Network, Tech. Rep., 1999. [Online]. Available: http://java.sun.com/developer/technicalArticles/Networking/HotSpot/

[14] G. Morrisett, D. Walker, K. Crary, and N. Glew, "From system F to typed assembly language," *ACM Trans. Program. Lang. Syst.*, vol. 21, pp. 527–568, May 1999. [Online]. Available: http://doi.acm.org/10.1145/319301.319345

[15] G. Morrisett, K. Crary, N. Glew, and D. Walker, "Stack-based typed assembly language," *J. Funct. Program.*, vol. 12, pp. 43–88, January 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=968425.968427

[16] A. Munshi, *The OpenCL Specification*, Khronos OpenCL Working Group, 2011. [Online]. Available: http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

[17] J. A. Fisher, "Very long instruction word architectures and the ELI-512," in *Proceedings of the 10th annual international symposium on Computer architecture*, ser. ISCA '83. New York, NY, USA: ACM, 1983, pp. 140–150. [Online]. Available: http://doi.acm.org/10.1145/800046.801649

[18] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff, "Escape analysis for Java," *SIGPLAN Not.*, vol. 34, pp. 1–19, October 1999. [Online]. Available: http://doi.acm.org/10.1145/320385.320386

[19] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala: A Comprehensive Step-by-Step Guide, 2nd Edition*, 2nd ed. USA: Artima Incorporation, 2011.

[20] M. Sagiv, T. Reps, and R. Wilhelm, "Parametric shape analysis via 3-valued logic," *ACM Trans. Program. Lang. Syst.*, vol. 24, pp. 217–298, May 2002. [Online]. Available: http://doi.acm.org/10.1145/514188.514190

[21] T. Veldhuizen, *Expression templates*. New York, NY, USA: SIGS Publications, Inc., 1996, pp. 475–487. [Online]. Available: http://dl.acm.org/citation.cfm?id=260627.260749