Language Run-time Systems: an Overview

Evgenij Belikov

Heriot-Watt University, School of Mathematical and Computer Sciences
Riccarton, EH14 4AS, Edinburgh, Scotland, UK
eb120@hw.ac.uk

Abstract

The proliferation of high-level programming languages with advanced language features and the need for portability across increasingly heterogeneous and hierarchical architectures require a sophisticated run-time system to manage program execution and available resources. Additional benefits include isolated execution of untrusted code and the potential for dynamic optimisation, among others. This paper provides a high-level overview of language run-time systems with a focus on execution models, support for concurrency and parallelism, memory management, and communication, whilst briefly mentioning synchronisation, monitoring, and adaptive policy control. Two alternative approaches to run-time system design are presented and several challenges for future research are outlined. References to both seminal and recent work are provided.

1998 ACM Subject Classification A.1 Introductory and Survey, D.3.4 Run-time Environments

Keywords and phrases Run-time Systems, Virtual Machines, Adaptive Policy Control

Digital Object Identifier 10.4230/OASIcs.ICCSW.2015.3

1 Introduction

Programming languages have evolved from assembly languages, where programmers have to specify a sequence of low-level instructions for a specific target platform, towards higher levels of expressiveness and abstraction. Imperative languages enabled *structured programming* by simplifying the expression of nested conditionals, iteration, and by allowing the grouping of instructions to named procedures. More recently, object-oriented and declarative languages with sophisticated type systems and support for generics, polymorphism, concurrency, and parallelism, among other advanced features, further improved *portability* and *productivity*.

It is widely known that given a set of M programming languages and N target platforms, using a *platform-independent intermediate representation*, which abstracts over the instruction set of the target architecture, reduces the number of necessary translation components from $M*N$ to $M+N$ facilitating the implementation of and interoperability between programming languages across a wide range of diverse hardware architectures [37, 34].

As the advanced language features, such as automatic memory management, generics and reflection, require support beyond the static capabilities of a compiler, a *run-time system* (RTS, also referred to as *high-level language virtual machine* – a process-oriented sub-class of virtual machines [31, 11, 20, 41]) was introduced to dynamically manage program execution whilst maintaining high performance, which program interpreters sacrifice in most cases [38]. The added flexibility of delayed specialisation and additional context information available at run time allow for dynamic optimisation for many applications with no hard real-time requirements, e.g. by detecting program hot spots and recompiling relevant program fragments just-in-time (JIT) [4, 2]. Moreover, recent trends towards increasingly hierarchical and heterogeneous parallel architectures [35, 9] and the promise of Big Data Analysis [30] have

© Evgenij Belikov;
licensed under Creative Commons License CC-BY
2015 Imperial College Computing Student Workshop (ICCSW 2015).
Editors: Claudia Schulz and Daniel Liew; pp. 3–12

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

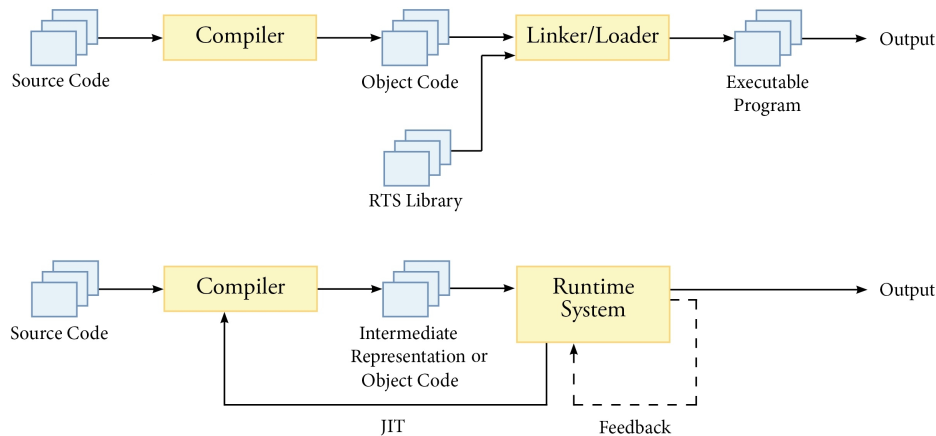

Figure 1 RTS in the Software Life Cycle.

reinvigorated the interest in language support for concurrency, parallelism, and Distributed Computing, based on advanced compiler and RTS-level functionalities [7].

This paper provides an overview of a set of representative language run-time systems including both established systems and research prototypes, presents the key components of a generic RTS and discusses associated policies as well as two more radical alternative RTS designs. Finally, challenges and opportunities for future research are outlined. While the discussion remains rather brief and high-level, references to technical literature are provided.

2 The Architecture of a Generic Language Run-time System

Here, a *run-time system* is defined as *a software component responsible for dynamic management of program execution and of the resources granted by the OS*. A RTS is distinct from and embedded into a more broad *run-time environment*, which is defined to include the OS that is responsible for overall resource management, and RTS instances managing other running applications unaware of each other's existence and resource demands. In other words, an RTS mediates interaction between the application and the OS whilst managing language-specific components in user space to avoid some context switching overhead [39].

2.1 Run-time System in the Software Life Cycle

The term *software life cycle* describes the stages of the software development process and their relations. Here the focus is on the compilation and execution stages. The RTS is primarily used during the application run time, consisting of the startup, initialisation, execution and termination phases, in addition to potential dynamic linking, loading and JIT-compilation as well as ahead-of-time (AOT) compilation.

Figure 1 illustrates the possible roles of the RTS in the software life cycle (excluding design, construction, and maintenance). For instance, C's minimalistic RTS (`crts0`) is added by the linker/loader and initialises the execution environment (e.g. the stack, stack/heap pointers, registers, signal handlers), transfers control to the program by calling the `main` function, and eventually passes on `main`'s return value to the run-time environment (e.g. the shell). However, it is platform-specific and implemented in assembler so that recompilation is required on different architectures, thus the *abstract machine* functionality is not provided.

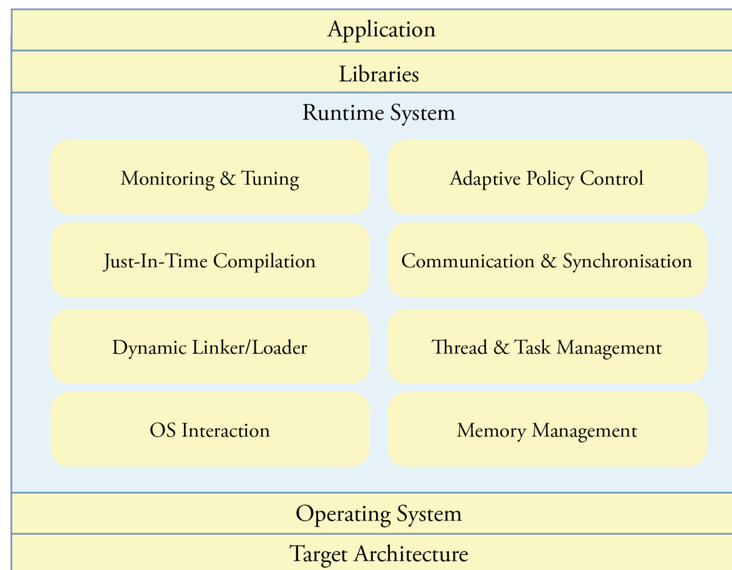

Figure 2 RTS Components in the Software Stack.

More sophisticated RTSes are either implemented as a separate library and linked to the application object code to create an executable program (e.g. GUM RTS for Glasgow parallel Haskell [42]) or are themselves applications which receive program code as an input (e.g. Java Virtual Machine (JVM) [26], and .NET Common Language Runtime (CLR) [33]). The input is usually in an *intermediate representation form* (IR, virtual instruction set) that decouples symbolic programming language representation from the native instruction set of the target architecture, which is then either interpreted or JIT-compiled to native code. Ahead-of-time compilation can be used to compile IR to native code to avoid initially running the interpreted IR code until the JIT phase is complete to reduce start-up time.

2.2 Run-time System Structure and Function

We focus on the general components corresponding to key RTS functions and associated *policies*, i.e. *sets of rules that define choices in the behaviour of the system.* We omit the discussion of RTSes for embedded platforms as strict real-time requirements, often associated with embedded applications, preclude the use of an RTS. The main function of the RTS is the management of program execution on the implemented abstract machine based on a specific execution model. The policies include interaction with the OS, memory management, thread and task management, communication, and monitoring, among others.

Execution Models. A common model is that of a *stack machine* corresponding closely to the Von-Neumann architecture [18], where a stack is used to store function arguments and intermediate results returned from function calls. This sequential model is often employed in RTSes for imperative and object-oriented languages along with *register machines*, where a stack is replaced by a set of virtual registers, which are mapped to the machine's registers.

Another prominent model is *graph reduction* commonly used with functional languages based on the lambda calculus [22, 5]. This model is suitable for parallelism as parts of the graph can be reduced independently [17]. Another notable model is *term-rewriting* that is used

with languages based on predicate logic [23], such as Prolog [43], where facts are represented by terms and new facts can be deduced from available facts. This model implements a variant of tree search and has proven particularly suitable for Artificial Intelligence applications.

OS Interaction. Commonly, an RTS relies on the OS for resource allocation, I/O and networking and provides a layer hiding OS-specific details and offering an *architecture-independent* API to systems programmers. RTS often directly implements higher-level language features and lower-level language primitives that are difficult to implement efficiently in the language itself. Hiding low-level details from the programmers is beneficial for productivity by providing abstractions and by transparently handling boilerplate initialisation, clean-up and error checking. For instance, channels in Erlang [1] offer a higher level communication mechanism than sockets [39] and in some cases communication and synchronisation are *fully implicit*, as in the distributed GUM RTS for parallel Haskell.

Error and Exception Handling. Error handling is often fairly crude such as a global `errno` variable that can be explicitly checked along with the error code returned by the function. More advanced language features include custom signal handlers and exception handling using the `try`/`catch` mechanism which is explicit and often results in complex control flow. Sophisticated RTSes can transparently handle errors at run time as demonstrated by Erlang's supervision mechanism, where failed processes can be restarted and resume execution.

Memory Management. Most high-level languages require the RTS to support *garbage collection* (GC) to automatically manage allocation and reclamation of heap objects. There are several common GC schemes [24]. One mechanism is *reference-counting* where a counter indicates the number of references to an object which can be recovered once this count drops to zero. By contrast, *mark-and-sweep* GC walks through the heap and marks all objects to be collected which is more disruptive but is able to collect reference cycles. Moreover, GC can be *compacting* to avoid fragmentation by periodically grouping the used objects together, and *generational* where objects reachable for a long time are promoted to a less often GC'd heap region, based on the insight that long surviving objects are likely to survive longer and most objects expire after a short period of time. A potential scalability issue on multicores is the use of a *stop-the-world* GC which suspends all execution when GC is performed, suggesting the use of private heaps across cores to enable *concurrent* or *distributed* GC.

Thread and Task Management. OS-level threads and RTS-level tasks (or *lightweight threads*) are at the heart of support for concurrency and parallelism, which is a key source of application performance on modern hardware. Often the RTS manages a thread pool and a task pool multiplexing tasks onto a set of OS threads for scalability. Many languages provide explicit threaded programming model, however, this model is deemed a poor choice due to observable non-determinism and notoriously difficult to detect and correct *race conditions* and *deadlocks* [25]. Thus higher-level deterministic programming models such as Transactional Memory [19] and semi-explicit or skeleton-based models have gained increasing attention [7].

Furthermore, explicit *synchronisation* involves a granularity trade-off that is detrimental to portable performance: a global lock would be safe but sacrifices parallelism, whereas fine-grained locking may result in prohibitive overheads. Coordination of execution, including scheduling and work distribution across heterogeneous architectures substantially increases the complexity of RTS decisions. This is the reason why *optional* tasks or threads are preferable to *mandatory* ones: the RTS only executes a task in parallel if it appears worthwhile [42, 13].

Two main families of work distribution mechanisms are *work pushing* (or work sharing) where work is offloaded eagerly and *work stealing* [8], a decentralised and adaptive mechanism where work distribution is demand-driven.

Communication. Large-scale applications require scalability beyond a single node mandating distribution of work across a cluster, a Grid, or in the Cloud. Two common schemes include *shared memory* (which can be distributed, requiring an additional abstraction layer) and *message passing*. Serialisation of computation and data is required if communication occurs across a network. A low-level library such as MPI [16] can be used to implement communication functionality. More sophisticated protocols such as publish/subscribe [12] may be beneficial for distributed cooperative RTSes which share contextual information.

Monitoring. Profiling, debugging, and execution replay rely on tracing (logging of relevant events), whereas dynamic optimisations rely on system information obtained through monitoring at run time. Common profiling strategies are, in order of increasing overhead, cumulative summary statistics, census-based statistics (collected at times of disruption, e.g. GC), fine-grained event-based statistics (using a sampling frequency or recording all events).

3 A Qualitative Run-time System Comparison

Table 1 provides an overview of several mainstream and research RTSes regarding their support for the features introduced above. We observe that a stack machine implementation is most common for mainstream languages whilst graph reduction is popular among declarative implementations. CLR provides support for tail call optimisation, so that declarative languages can be supported (e.g. F#), whereas JVM lacks direct support. JVM and CLR are compared in more detail elsewhere [15, 40], and although superficially similar they have many distinctive features (e.g. unlike JVM, CLR was designed for multi-language support).

As concurrency and parallelism support grow more important we observe that such support was added as an afterthought and often as a non-composable library to the mainstream languages. By contrast, Erlang was designed for concurrency and natively supports channels which facilitate communication among lightweight processes. Additionally, declarative languages often are a better match for parallelism due to immutability and graph reduction execution model [17]. For example, Haskell provides support for lightweight threads and data parallelism, whereas GUM supports adaptive distributed execution of GpH programs [29, 42].

Most of the RTSes support some kind of GC but only in few cases a more scalable design is used (e.g. Erlang uses per-process GC, X10 [10] and GUM use distributed GC). However, an empirical study is necessary to judge on the relative merits of different schemes. In mainstream languages communication and synchronisation are often explicit, increasing the complexity of concurrent and parallel programming. From this angle GpH appears as a very-high-level language in which all such aspects are implicit, whereas communication in X10 is implicit using the *Partitioned Global Address Space* (PGAS) abstraction whereas locality settings for the distributed data structures are explicitly provided by the programmer.

Many of the RTSes also employ adaptive optimisations of some kind such as JIT, work-stealing, or self-adjusting granularity control to throttle parallelism depending on available resources (e.g. in GUM). Like memory management, this is an area of ongoing research [2].

Table 1 A High-Level Overview of Ten Representative Run-time Systems.

RTS/ Language	Exec. Model	Concurrency/ Parallelism	Memory Management	Commun./ Synchron.	Adaptive Policies
crt0 C	stack machine	libs: fork, Pthreads	explicit malloc/free	sockets, MPI/ mutex, locks	user-defined
JVM Java/Scala/..	stack machine	libraries Threads, ..	impl. gen. mark&sweep	sockets, RPC/ synchronized	JIT
CLR C#/F#/..	stack machine	async/futures PLINQ, ..	implicit gen. m&s	sockets/ locks, wait	JIT
GHC-SMP Haskell	graph reduction	forkIO, STM, Par monad, ..	implicit gen. m&s	libraries/ MVars,TVars	work stealing
GUM Haskell ext.	graph reduction	par, Eval Strategies	impl. distr. ref. count.	both implicit (graph nodes)	work stealing
Manticore CML ext.	graph reduction	expl. tasks/ impl. data	impl. gen. local/global	expl. mesg./ sync	lazy tree splitting
X10 RTS X10	stack machine	async, futur./ foreach, ..	impl. distr. ref. count.	PGAS/ atomic	work stealing
BEAM Erlang	register machine	spawn primitive	per process compact./gen.	expl. mesg./ mesg. boxes	hot code swapping
Cilk C extension	stack machine	spawn, cilk func.	impl. cactus stack	explicit/ sync	work stealing
SWI Prolog ISO Prolog	term rewriting	threads/ concurrent	impl. low prio. thread	sockets/ mutex, join	assert/ retract

4 Alternative Views on the Role of the Run-time System

The proliferation of Cloud Computing as an economical approach to elastically scaling IT infrastructure based on actual demand and of Big Data Analysis in addition to Scientific High-Performance Computing applications have lead to an increased interest in parallel and distributed RTSes capable of managing execution across distributed platforms at large scale. Additionally, cache coherence protocols exhibit limited scalability and many novel architectures (e.g. custom System-on-Chip architectures) resemble distributed systems to some extent and may be non-cache-coherent [36].

Moreover, as many programming languages were not designed with parallelism and distribution in mind, their RTSes provide only minimal support for rather low-level coordination. This is an issue since most programmers are not parallelism and distributed systems experts and explicitly specifying coordination is deemed prohibitively unproductive, favouring flexible and safe RTS approaches. Convergence in RTS and OS functionality can be observed in addition to the aforementioned architectural trends that lead to promising alternative views.

4.1 Holistic Run-time System

A Holistic RTS [28] would move from separate RTS instances running on top of a host OS to a distributed RTS which takes over most of the OS functionalities and provides support for multiple languages simultaneously. The RTS would manage multiple applications at the same time and in a distributed fashion to avoid interference and coordinate distributed GC across isolated heaps. Additional benefits arise from the ability to share libraries and RTS components as well as to better utilise higher-level information for holistic optimisation.

4.2 Application-Specific Run-time System

At the other end of the spectrum, an alternative approach is based on a generic distributed OS [36] (e.g. based on the *Multikernel* approach [6]) offering a standardised interface for RTSes to communicate their demands, facilitating cooperative execution and helping avoid pathological interference. This way RTSes could be more lightweight and tailored for specific applications taking additional context information about application characteristics into account. To achieve this, a novel standard OS-RTS-interaction interface is required that would allow the OS to schedule RTSes and mediate the negotiations among them.

5 Challenges and Opportunities for Future Research

Recent trends towards heterogeneous and hierarchical parallel architectures and increased demand for high-level language features pose substantial challenges for the design and implementation of efficient language run-time systems offering opportunities for research.

5.1 Adaptation and Auto-Tuning for Performance Portability

As argued above, manual program tuning is deemed infeasible for most application domains, thus rendering automatic compiler and RTS optimisations critical for performance portability and scalability. Alas, many decisions have been shown to be NP-hard in general, thus requiring heuristics and meta-heuristics to facilitate heuristics choice based on available context information. Machine Learning has been shown to improve compiler flag selection [14] and it appears that similar techniques may be applicable for RTS-level adaptation. To achieve performance portability, the RTS must provide some performance guarantees based on applications' requirements and available architectural and system-level information.

5.2 Fault Tolerance

Traditionally, language RTS designs have focused on language features and performance, yet mechanisms to transparently detect, avoid, and recover from failures are becoming more important, as hardware failure is rather the norm than exception in modern large-scale systems [3]. In particular, the mean time between failures is rapidly decreasing following the exponential growth of the transistor budget in accordance with Moore's Law.

5.3 Safety and Security

One of the key benefits of using RTSes is that untrusted code can be executed safely in a sandboxed environment so that other applications are safeguarded from many negative effects that could be caused by malicious code. Additionally, intermediate code can be verified and type-checked by the RTS (e.g. as done by JVM's bytecode verifier) to eliminate the possibility of whole classes of common errors. Furthermore, automatic GC eliminates the source of memory corruption errors and avoids the buffer overflow vulnerability. It is an open issue to ensure safe and secure execution of sensitive workloads on Cloud-based infrastructures, which is the reason why many companies are reluctant to use new technologies or prefer investing into private data centres. Support for dependent and session types is another research direction focused on improving the safety of high-level languages [32, 21].

5.4 Unified Distributed Run-time Environment

As described above, both alternative RTS approaches require tighter integration of RTS and OS functions favouring cross-layer information flow. Moreover, a unified and extensible RTS needs to support diverse execution models for multi-language interoperability, as has been demonstrated by the CLR. The holistic view would facilitate avoidance of interference and enable cooperative solutions for resource management. Ultimately, sophisticated RTSes would hide coordination and management complexities from programmers increasing productivity and avoiding explicit overspecification of coordination to achieve high performance portability across diverse target platforms. On the other hand, it may be worthwhile to spatially and temporally separate applications and grant exclusive access to some resources on massively parallel architectures [27].

6 Conclusion

Based on the current trends, modern programming languages are likely to increasingly rely on a sophisticated RTS to manage efficient program execution across diverse target platforms. We have reviewed the role of the RTS in the software life cycle and the key RTS components and associated policies. Additionally, a qualitative comparison of mainstream and research RTSes illustrated the breadth of the design choices across the dimensions of execution models, memory management, support for concurrency and parallelism, as well as communication and adaptive policy control. The overview is complemented by references to influential historical results and current research outcomes alongside a discussion of alternative RTS designs followed by an outline of several promising areas for future research.

Acknowledgements. The author is grateful to Greg Michaelson and Hans-Wolfgang Loidl for thought-provoking discussions, to Julian Godesa, Konstantina Panagiotopoulou and Prabhat Totoo for useful comments, and to Siân Robinson Davies for designing the figures, as well as to the three anonymous reviewers for valuable feedback that improved this paper.

References

1. J. Armstrong. Erlang – a survey of the language and its industrial applications. In *Proceedings of INAP*, volume 96, 1996.
2. M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. A survey of adaptive optimization in virtual machines. *Proc. of the IEEE*, 93(2):449–466, 2005.
3. A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.
4. J. Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35(2):97–113, 2003.
5. H. Barendregt. *The lambda calculus*, volume 3. North-Holland Amsterdam, 1984.
6. A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: a new OS architecture for scalable multicore systems. In *Proc. of 22nd ACM Symp. on Operating systems principles*, pages 29–44, 2009.
7. E. Belikov, P. Deligiannis, P. Totoo, M. Aljabri, and H.-W. Loidl. A survey of high-level parallel programming models. Technical Report MACS-0103, Heriot-Watt University, 2013.
8. R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, September 1999.
9. A. Brodtkorb, C. Dyken, T. Hagen, J. Hjelmervik, and O. Storaasli. State-of-the-art in heterogeneous computing. *Scientific Programming*, 18(1):1–33, May 2010.

10 P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *ACM SIGPLAN Notices*, 40(10):519–538, 2005.

11 P. Deutsch and A. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proc. of the 11th ACM Symp. on Principles of Programming Languages*, pages 297–302, 1984.

12 P. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, 2003.

13 M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Implicitly threaded parallelism in Manticore. *Journal of Functional Programming*, 20(5-6):537–576, 2010.

14 G. Fursin, Y. Kashnikov, A. Memon, et al. Milepost GCC: Machine learning enabled self-tuning compiler. *International Journal of Parallel Programming*, 39(3):296–327, 2011.

15 J. Gough. Stacking them up: a comparison of virtual machines. *Australian Computer Science Communications*, 23(4):55–61, 2001.

16 W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT Press, 1999.

17 K. Hammond. Why parallel functional programming matters: Panel statement. In *Proceedings of Ada-Europe*, volume 6652 of *LNCS*, pages 201–205, 2011.

18 J. Hennessy and D. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 5th edition, 2011.

19 M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. of the 20th Annual International Symposium on Computer Architecture (ISCA'93)*, pages 289–300. ACM, 1993.

20 U. Hölzle and D. Ungar. A third-generation SELF implementation: reconciling responsiveness with performance. *ACM SIGPLAN Notices*, 29(10):229–243, 1994.

21 K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. *ACM SIGPLAN Notices*, 43(1):273–284, 2008.

22 J. Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.

23 J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proc. of the 14th ACM SIGPLAN Symposium on Principles of programming languages*, pages 111–119. ACM, 1987.

24 R. Jones, A. Hosking, and E. Moss. *The garbage collection handbook: the art of automatic memory management*. Chapman & Hall/CRC, 2011.

25 E. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, 2006.

26 T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java virtual machine specification*. Pearson Education, 2014. First published in 1996.

27 R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanovic, and J. Kubiatowicz. Tessellation: Space-time partitioning in a manycore client OS. In *Proceedings of the 1st USENIX Conference on Hot Topics in Parallelism*, pages 10–10, 2009.

28 M. Maas, K. Asanovic, T. Harris, and J. Kubiatowicz. The case for the holistic language runtime system. In *First International Workshop on Rackscale Computing*, 2014.

29 S. Marlow, S. Peyton Jones, and S. Singh. Runtime support for multicore Haskell. *ACM SIGPLAN Notices*, 44(9):65–78, 2009.

30 V. Mayer-Schönberger and K. Cukier. *Big Data: A revolution that will transform how we live, work, and think*. Houghton Mifflin Harcourt, 2013.

31 J. McCarthy. History of LISP. In *History of programming languages*, pages 173–185, 1978.

32 J. McKinna. Why dependent types matter. *ACM SIGPLAN Notices*, 41(1), 2006.

33 E. Meijer and J. Gough. Technical overview of the Common Language Runtime. *Language*, 29:7, 2001.

34 K. Nori, U. Amman, K. Jensen, and H.-H. Nägeli. *The Pascal-P Compiler: Implementation Notes*. ETH Zurich TR 10, 1974.

35 J. Owens, D. Luebke, N. Govindaraju, et al. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

36 S. Peter, A. Schüpbach, D. Menzi, and T. Roscoe. Early experience with the Barrelfish OS and the Single-Chip Cloud Computer. In *MARC Symposium*, pages 35–39, 2011.

37 M. Richards. The portability of the BCPL compiler. *Software: Practice and Experience*, 1(2):135–146, 1971.

38 T. Romer, D. Lee, G. Voelker, A. Wolman, W. Wong, J.-L. Baer, B. Bershad, and H. Levy. The structure and performance of interpreters. *ACM SIGPLAN Not.*, 31(9):150–159, 1996.

39 A. Silberschatz, P. Galvin, and G. Gagne. *Operating system concepts*. AddisonWesley, 1998.

40 J. Singer. JVM versus CLR: a comparative study. In *Proc. of 2nd Intl Conf. on Principles and Practice of Programming in Java*, pages 167–169. Computer Science Press, 2003.

41 J. Smith and R. Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, 2005.

42 P. Trinder, K. Hammond, J. Mattson Jr., A. Partridge, and S. Peyton Jones. GUM: a portable parallel implementation of Haskell. In *Proc. of PLDI'96*, 1996.

43 J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.