A Model-Driven Performance Analysis Framework for Distributed, Performance-Sensitive Software Systems

Swapna S. Gokhale
Dept. of Computer Science and Engineering
University of Connecticut
Storrs, CT
{ssg@cse.uconn.edu}

Aniruddha Gokhale
Dept. of Electrical Engineering and Computer Science
Vanderbilt University
Nashville, TN
{a.gokhale@vanderbilt.edu}

Jeffrey Gray
Dept. of Computer and Information Science
Univ. of Alabama at Birmingham
Birmingham, AL
{gray@cis.uab.edu}

I. Introduction

Large-scale, distributed, performance-sensitive software (DPSS) systems from the basis of mission- and often safety-critical applications. DPSS systems comprise of many independent artifacts, such as network/bus interconnects, many coordinated local and remote endsystems, and multiple layers of software. DPSS systems demand multiple, simultaneous predictable performance requirements such as end-to-end latencies, throughput, reliability and security, while also requiring the ability to control and adapt operating characteristics for applications with respect to such features as time, quantity of information, and quality of service (QoS) properties including predictability, controllability, and adaptability of operating characteristics for applications with respect to such features as time, quantity of information, accuracy, confidence and synchronization. All these issues become highly volatile in large-scale DPSS systems due to the dynamic interplay of the many interconnected parts that are often constructed from smaller parts.

Although it is possible in theory to develop these types of complex DPSS systems from scratch, contemporary economic and organizational constraints, as well as increasingly complex requirements and competitive pressures, make it infeasible to do so in practice. To address these challenges there is a paradigm shift in the way DPSS systems are being developed. This new paradigm is called *Software Factories* [7] and it envisions the *industrialization* of software development, similar to industrialization in other domains, such as the avionics and automotive industry. Software factory architectures and processes determine how DPSS systems will be assembled and deployed using patterns (which codify recurring solutions to particular problems occurring in certain contexts [6]), models (which are higher levels of abstractions of the domain), tools (which automate mundane and repetitive tasks), and product line architectures (which are a collection of a large number of reusable building blocks, and their configuration and customization options).

A key enabler to the Software Factories paradigm has been

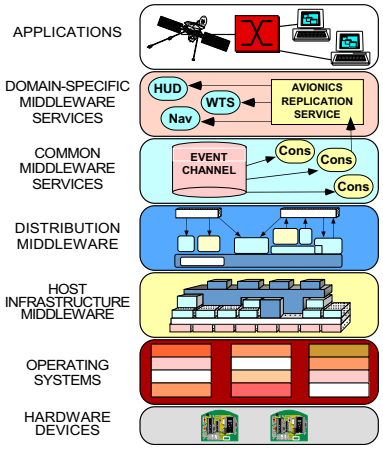

Fig. 1. **Middleware Stack**

middleware [13], as shown in Figure 1. Middleware comprises software layers that provide higher-level execution semantics and reusable services that coordinate how application components are composed and how they interoperate. The primary role of middleware is to (1) functionally bridge the gap between application programs and the lower-level hardware and software infrastructure in order to coordinate how parts of applications are connected and how they interoperate, (2) enable and simplify the integration of components developed by multiple technology suppliers, and (3) provide a common reusable accessibility for functionality and patterns by factoring out artifacts from applications into reusable code.

Middleware addresses key commonality and variability [3] concerns of DPSS systems by providing product line architectures. These building blocks embody good design practices called patterns [6], [14] and are used to compose and customize large systems end-to-end. To enhance configurability and customization, moreover, middleware product lines often comprise several building blocks that provide similar functionality (*e.g.*, request demultiplexing), but which are meant to be used in different operational contexts (*e.g.*, as in concurrent

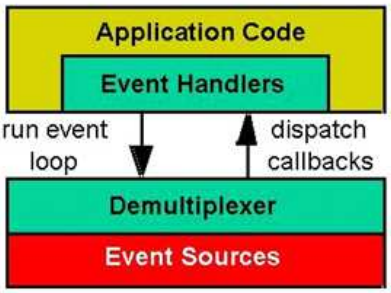

Fig. 2. **Event Demultiplexers in Middleware**

versus iterative request handling). An arbitrary choice of a building block in the software factory processes and architectures for DPSS system may lead to significant performance overheads and can impact DPSS system correctness. The magnitude and complexity of DPSS systems demands that the software factory processes and architectures driving the composition of DPSS systems from these building blocks be functionally correct and provide the necessary performance assurance.

II. Research Challenges

In Section I we discussed the emerging paradigm shift towards industrialization of software systems and the need for performance analysis of the composed infrastructure earlier in the development lifecycle. In this section, we describe the challenges in conducting such performance analysis in the face of significant variability manifested in patterns-based reusable building blocks provided by contemporary middleware technologies and the need to compose them in different ways to suit the requirements of the applications. There are two dimensions of these variabilities discussed below.

A. Horizontal dimension of variability

Middleware developers provide numerous configuration options to customize the behavior of individual building blocks, such as Reactor and Acceptor. This flexibility incurs significant variability in the behavior and performance of a building block and hence on the design choices for architecting an application. Since the impact is on a per building block basis – as opposed to a composition – we refer to this as the *horizontal dimension of variability*.

Figure 2 shows an example of this type of variability involving an event demultiplexing and dispatching mechanism codified by the Reactor pattern [14]. In this pattern, a DPSS application registers an event handler with the event demultiplexer and delegates to it the responsibility of listening for incoming events. On occurrence of an event, the demultiplexer dispatches the event by making a callback to the right application-supplied event handler.

The Reactor pattern could be configured in many different ways depending on the event demultiplexing capabilities provided by an OS and the concurrency requirements of an application. For example, the demultiplexing capabilities of a Reactor could be based on the `select()` or `poll()` system calls provided by POSIX-compliant operating systems or `WaitForMultipleObject()` available on Windows. Moreover, the handling of the event in the event handler could be managed by the same thread of control that was listening for events giving rise to a single-threaded Reactor implementation. Alternately, the event could be delegated to a pool of threads to handle the events giving rise to a thread-pool Reactor (other variants are also possible). These different configuration choices are dictated by the functional and performance needs of DPSS systems, which means that DPSS developers are responsible for handling this horizontal dimension of variability.

B. Vertical dimension of variability

When designing DPSS systems, developers must decide the set of patterns to use in the system composition and also ensure that the customizations to individual building blocks are compatible with each other in the vertical composition. This approach ensures that the desired end-to-end performance and functional requirements are met. The choice of patterns selected are driven by various factors, including the context in which the DPSS system will be deployed, the concurrency and distribution requirements of the application, and other concerns, such as end-to-end latency and timeliness requirements for real-time DPSS systems, or throughput for other DPSS systems (such as telecommunications call processing). We refer to the incurred design space variability as the vertical dimension since it occurs during a vertical composition of configurable patterns.

Figure 3 illustrates a family of interacting patterns forming a pattern language [1] for middleware designed to support DPSS systems. Patterns such as the *Active Object* (not shown) and *Leader-Follower* [14], provide alternate approaches to concurrency with each solution having its own advantages and disadvantages. For example, although the Active Object pattern is simple to implement, it incurs an additional performance penalty due to thread context switching and message queueing. Conversely, the Leader-Follower does not incur these drawbacks, but its implementation is more complicated to implement and analyze, so it may cause race conditions if not implemented properly.

The *Reactor* [14] pattern shown in Figure 3 provides solutions to synchronous event demultiplexing and dispatching. The Reactor pattern can be customized by composing it with a *Strategy* [6] to use the *Leader-Follower* pattern for request demultiplexing and dispatching. Similarly, a reactor could be composed with the *Connector* (client-side) or *Acceptor* (server-side) pattern. Developers of DPSS systems need to make the right choices of patterns and their configurations to use in their solutions, so that the end-to-end performance and functional requirements of the DPSS systems are met, *i.e.*, they are responsible for handling the vertical dimension of variability.

C. Proposed solutions

Sections II-A and II-B describe the two dimensions of variability in contemporary middleware technologies that are

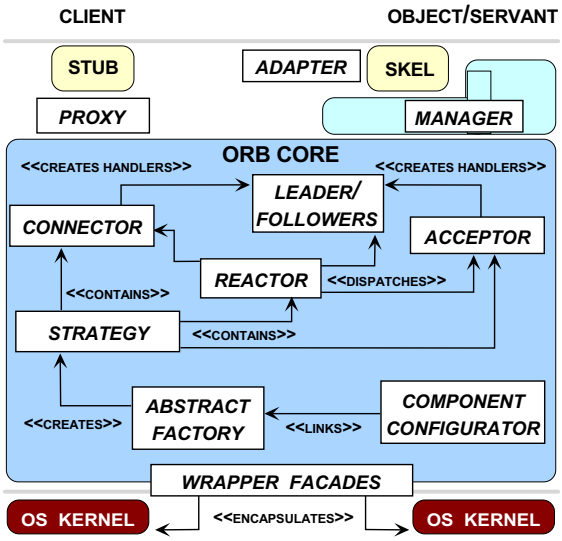

Fig. 3. **Middleware Patterns and Pattern Languages**

manifested in the form of highly configurable reusable building blocks. Since there are no mechanisms available today to estimate the expected end-to-end performance provided by the different customizations to a building building block and also the composition of these middleware blocks, DPSS developers must rely on rigorous testing much later in the lifecycle to determine if their design choices are appropriate. Unfortunately, these decisions occur very late in the lifecycle of DPSS systems, which adversely impact costs, schedules, and quality. Mechanisms are therefore needed that can enable performance estimation of composable systems at design time since changes made earlier in the lifecycle are easier/cheaper to incorporate. Moreover, the higher-levels of abstraction and formalism in the design phase makes it possible to formally verify DPSS system correctness, *i.e.*, to help enforce the notion of "correct by construction" for software factory processes and architectures.

Analyzing the performance of DPSS systems composed of pattern-based reusable building blocks at design-time requires resolving the following technical needs:

- Constructing and validating a set of base performance models that capture the common characteristics of individual patterns-based building blocks,
- Customizing and specializing these base models to account for the horizontal dimension of variability,
- Composing the customized models of the patterns according to the software factory processes and architectures for DPSS systems to account for the vertical dimension of variability,
- Conducting whole-system performance analysis of the composed system.

Our solution to the first aspect of design-time performance analysis is based on the application of well-established performance analysis techniques including analytical/numerical techniques such as Stochastic Reward Nets (SRNs) [12] and simulations such as hybrid systems [8] simulation [2]. To eliminate the need to manually develop performance analysis models, we propose to use Model-Driven Software Development (MDSD) [11], [10], [7], which is used to model individual building blocks and the assembly of DPSS systems, and Generative Programming [4], which is used to synthesize artifacts required to drive the performance evaluation of the end system. Aspect-Oriented Software Development (AOSD) [5] is used to address the two dimensions of variability.

III. Preliminary Results

In this section we present the process of developing a SRN model of the Reactor pattern. We then illustrate how the SRN model could be used to estimate the performance metrics of the Reactor.

A. Characteristics of the Reactor pattern

We consider a single-threaded, *select*-based Reactor implementation, with the following characteristics:

- The system is moeled with two types of input events with one event handler for each type of event registered with the Reactor.
- Each event type has a separate queue, which holds the incoming events of that type. The buffer capacity for the queue of type one events is denoted N_1, and of type two events is denoted N_2.
- Events of type #1 are serviced with a higher priority over events of type #2.
- Event arrivals for both types of events follow a Poisson distribution with rates λ_1 and λ_2, while the service times of the events are exponentially distributed with rates μ_1 and μ_2.

B. Performance metrics

The following performance metrics are of interest for each one of the event types in the Reactor pattern model described above.

- **Expected throughput** – which provides an estimate of the number of events that can be processed by the single threaded event demultiplexing framework. These estimates are important for applications. such as telecommunications call processing.
- **Expected queue length** – which provides an estimate of the queuing for each of the event handler queues. These estimates are important since it is possible to develop appropriate scheduling policies for applications with real-time requirements.
- **Expected total number of events** – which provides an estimate of the total number of events in a system. These estimates are also tied to making scheduling decisions. In addition, these estimates will determine the right levels of resource provisioning required to sustain the system demands.
- **Probability of event loss** – which indicates how many events will have to be discarded due to lack of buffer space. These estimates are important particularly for safety-critical systems, which cannot afford to lose events. Alternately, these also give an estimate on the desired levels of resource provisioning.

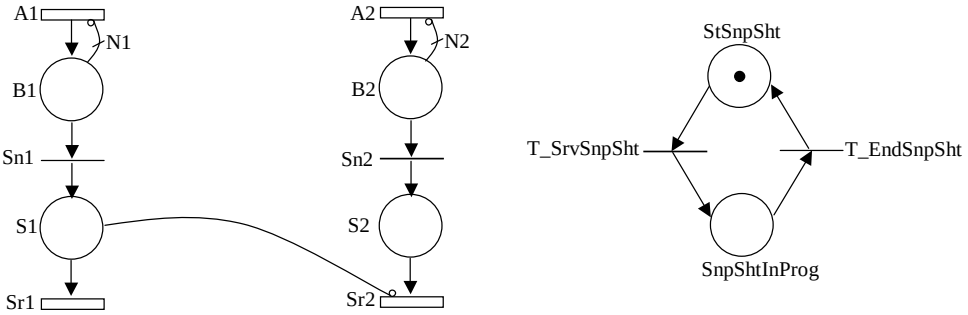

Fig. 4. **SRN model for the Reactor pattern**

TABLE I
GUARD FUNCTIONS

Transition	Guard function
$Sn1$	$((\#StSnpShot == 1)\&\&(\#B1 >= 1)\&\&(\#S1 == 0))?1 : 0$
$Sn2$	$((\#StSnpShot == 1)\&\&(\#B2 >= 1)\&\&(\#S2 == 0))?1 : 0$
$T\_SrvSnpSht$	$((\#S1 == 1)\|\|(\#S2 == 1))?1 : 0$
$T\_EndSnpSht$	$((\#S1 == 0\&\&(\#S2 == 0))?1 : 0$

C. SRN model

Figure 4 shows the SRN model for the Reactor pattern described in Section III-A. In the figure, transitions $A1$ and $A2$ represent the arrivals of the events of types one and two, respectively. Places $B1$ and $B2$ represent the queue for the two types of events. Transitions $Sn1$ and $Sn2$ are immediate transitions which are enabled when a snapshot is taken. Places $S1$ and $S2$ represent the enabled handles of the two types of events, whereas transitions $Sr1$ and transition $Sr2$ represent the execution of the enabled event handlers of the two types of events. An inhibitor arc from place $B1$ to transition $A1$ with multiplicity $N1$ prevents the firing of transition $A1$ when there are $N1$ tokens in the place $B1$. The presence of $N1$ tokens in the place $B1$ indicates that the buffer space to hold the incoming input events of the first type is full, and no additional incoming events can be accepted. The inhibitor arc from place $B2$ to transition $A2$ achieves the same purpose for type two events. The inhibitor arc from place $S1$ to transition $Sr2$ prevents the firing of transition $Sr2$ when there is a token in place $S1$. This models the prioritized service for the events of type of one over events of type two.

Places $StSnpSht$, $SnpInProg$, $T\_SrvSnpSht$ and $T\_EndSnpSht$ are introduced to model the process of taking successive snapshots. Transition $Sn1$ is enabled when there is a token in place $StSnpSht$, at least one token in place $B1$ and no tokens in place $S1$. Similarly, transition $Sn2$ is enabled when there is a token in place $StSnpSht$, at least one token in place $B2$ and no tokens in place $S2$. Transition $T\_SrvSnpSht$ is enabled when there is a token in either one of the places $S1$ and $S2$, and the firing of this transition deposits a token in place $SnpShtInProg$.

The presence of a token in the place $SnpShtInProg$ indicates that the event handles that were enabled in the current snapshot are being serviced. Once these event handles complete execution, the current snapshot is complete and it is time to take another snapshot. This is accomplished by enabling the transition $T\_EndSnpSht$. Transition $T\_EndSnpSht$ is enabled when there are no tokens in both place $S1$ and $S2$. Firing of the transition $T\_EndSnpSht$ deposits a token in place $StSnpSht$, indicating that the service of the enabled handles in the present snapshot is complete which marks the initiation of the next snapshot. Table I summarizes the enabling/guard functions for the transitions in the net.

We now explain how the process of taking a single snapshot is modeled by the SRN model presented in Figure 4 with an example. We consider the scenario where there is one token in each one of the places $B1$ and $B2$, and there is a token in the place $StSnpSht$. Also, there are no tokens in places $S1$ and $S2$. In this scenario, transitions $Sn1$ and $Sn2$ are enabled. Both of these transitions are assigned the same priority, and any one of these transitions can fire first. Also, since these transitions are immediate, their firing occurs instantaneously. Without loss of generality, we assume that transition $Sn1$ fires before $Sn2$ depositing a token in place $S1$.

When a token is deposited in place $S1$ transition $T\_SrvSnpSht$ is enabled. In addition, transition $Sn2$ is already enabled. If transition $T\_SrvSnpSht$ were to fire before transition $Sn2$, it would disable transition $Sn2$, and prevent the handle corresponding to the second event type from being enabled. In order to prevent transition $T\_SrvSnpSht$ from firing before transition $Sn2$, transition $T\_SrvSnpSht$ is assigned a lower priority than transition $Sn2$. Since transitions $Sn1$ and $Sn2$ have the same priority, this also implies that the transition $T\_SrvSnpSht$ has a lower priority than transition $Sn1$. This ensures that in a given snapshot, event handles corresponding to each event type are enabled when there is

at least one event in the queue.

After both the event handles are enabled, transition $T\_SrvSnpSht$ fires and deposits a token in place $SnpShtInProg$. The event handle corresponding to type one event is serviced first which causes transition $Sr1$ to fire and the removal of the token from place $S1$. Subsequently, transition $Sr2$ fires and the event handle corresponding to the event of type two is serviced. This causes the removal of the token from place $S2$. Once both the events are serviced and there are no tokens in places $S1$ and $S2$, transition $T\_EndSnpSht$ fires which marks the end of the present snapshot and the beginning of the next one.

The performance measures described in Section III-B can be computed by assigning reward rates at the net level as summarized in Table II. The throughputs T_1 and T_2 are respectively given by the rate at which transitions $Sr1$ and $Sr2$ fire. The queue lengths Q_1 and Q_2 are given by the average number of tokens in places $B1$ and $B2$, respectively. The total number of events E_1 is given by the sum of the number of tokens in places $B1$ and $S1$. Similarly, the total number of events E_2 is given by the sum of the number of tokens in places $B2$ and $S2$. The loss probability L_1 is given by the probability of $N1$ tokens in place $B1$. Similarly, the loss probability L_2 is given by the probability of $N2$ tokens in place $B2$.

TABLE II
REWARD ASSIGNMENTS TO OBTAIN PERFORMANCE MEASURES

Performance metric	Reward rate
T_1	return rate($Sr1$)
T_2	return rate($Sr2$)
Q_1	return ($\#B1$)
Q_2	return ($\#B2$)
L_1	return ($\#B1 == N1?1 : 0$)
L_2	return ($\#B2 == N2?1 : 0$)
E_1	return($\#B1 + \#S1$)
E_2	return($\#B2 + \#S2$)

The SRN shown in Figure 4 can be solved using tools such as SPNP [9] to obtain the performance metrics described in Section III-B for various values of the input parameters. It can also be used to establish performance bounds via sensitivity analysis.

IV. FUTURE RESEARCH DIRECTIONS

Section III outlined our approach to developing design-time performance analysis tools to address the horizontal and vertical dimensions of variability in contemporary middleware that must be addressed to realize the goals of software industrialization through the example of the Reactor pattern. To date we have developed the basic performance model for the single-threaded Reactor pattern. Single-threaded, event-driven reactive systems form only a small subset of a large class of distributed applications that need scalability and high performance. These features are most often supported by introducing high levels of concurrency and different messaging paradigms, such as publish-subscribe or peer-to-peer asynchronous request-response. Many of these features therefore require many different patterns and their composition.

Existing *ad hoc* techniques based on manually selecting the building blocks to construct DPSS systems are error-prone and adversely impact performance, system costs, and schedules since most errors are caught late in the lifecycle of DPSS systems development. Model-driven development (MDD) techniques, such as the Model Driven Architecture (MDA) [11], Model-Integrated Computing [15], and Software Factories [7], provide the foundation for resolving many of these challenges. To make the industrialization of software development a reality, however, will require MDD tools that can model the software factory processes and architectures and analyze the performance of the composed infrastructure much earlier in the lifecycle, thereby significantly lowering system testing costs and improving the correctness of fielded systems.

The next research steps therefore include the following:

- **Performance models for building blocks** – this step involves developing base models and customizing these models to capture the horizontal dimension of variability in different patterns. Examples of these patterns include *Half-sync/Half-async*, *Proactor*, *Active Object*, and *Broker* among many others shown in Figure 3.
- **Performance models for compositions** – this step requires developing a suite of performance models for a set of pattern compositions shown in Figure 3 suited for different product-line architectures to capture the vertical dimension of variability.
- **Model validation** – this step requires developing techniques to validate the performance models. We envision using simulations and empirical benchmarking to validate the performance models for different workloads.
- **Tool-based design** – this step requires building a tool chain that will automate the three tasks outlined above.

V. CONCLUDING REMARKS

Technological advances in hardware and networking are driving the unprecedented growth in next generation of distributed, performance sensitive software (DPSS) systems. Today's state of the art in software tools and processes however fall short in meeting the increasing demand for newer families of DPSS systems. Software factories has been proposed as a concept to make the industrialization of software feasible.

Middleware has been a key enabler to realize the goals of software factories by providing a large number of reusable and configurable building blocks that can be composed to form platforms to host these DPSS systems. However, the very nature of reusability, configurability and composability gives rise to tremendous variability in how middleware building blocks can be configured and composed, and the resulting performance offered to applications. This paper describes a methodology using stochastic reward nets and a set of tools that DPSS developers can use at design-time to analyze effects of different configurations and compositions of middleware building blocks on resulting performance.

REFERENCES

[1] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language*. Oxford University Press, New York, NY, 1977.

[2] S. Bohacek, J.P. Hespanha, J. Lee, and K. Obraczka. A hybrid systems modeling framework for fast and accurate simulation of data communication networks. In *Proceedings of ACM SIGMETRICS '03*, June 2003.

[3] James Coplien, Daniel Hoffman, and David Weiss. Commonality and Variability in Software Engineering. *IEEE Software*, 15(6), November/December 1998.

[4] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Boston, 2000.

[5] Robert Filman, Tzilla Elrad, Mehmet Aksit, and Siobhan Clarke. *Aspect-Oriented Software Development*. Addison-Wesley, Reading, Massachusetts, 2004.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[7] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. John Wiley & Sons, New York, 2004.

[8] T. A. Henzinger and S. Sastry, editors. *Hybrid Systems: Computation and Control - Lecture Notes in Computer Science*. Springer Verlag, New York, NY, 1998.

[9] C. Hirel, B. Tuffin, and K. S. Trivedi. "SPNP: Stochastic Petri Nets. Version 6.0". *Lecture Notes in Computer Science 1786*, 2000.

[10] Gabor Karsai, Janos Sztipanovits, Akos Ledeczi, and Ted Bapty. Model-Integrated Development of Embedded Software. *Proceedings of the IEEE*, 91(1):145–164, January 2003.

[11] Object Management Group. *Model Driven Architecture (MDA)*, OMG Document ormsc/2001-07-01 edition, July 2001.

[12] A. Puliafito, M. Telek, and K. S. Trivedi. "The evolution of stochastic Petri nets". In *Proc. of World Congress on Systems Simulation*, pages 3–15, Singapore, September 1997.

[13] Richard E. Schantz and Douglas C. Schmidt. Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications. In John Marciniak and George Telecki, editors, *Encyclopedia of Software Engineering*. Wiley & Sons, New York, 2002.

[14] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley & Sons, New York, 2000.

[15] Janos Sztipanovits and Gabor Karsai. Model-Integrated Computing. *IEEE Computer*, 30(4):110–112, April 1997.