A Compiler-Enabled Model- and Measurement-Driven Adaptation Environment for Dependability and Performance

Vikram S. Adve[1] Adnan Agbaria[1] Matti A. Hiltunen[2] Ravi K. Iyer[1] Kaustubh R. Joshi[1]
Zbigniew Kalbarczyk[1] Ryan M. Lefever[1] Raymond Plante[1] William H. Sanders[1]
Richard D. Schlichting[2]

[1]University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{vadve,adnan,rkiyer,joshi1,kalbarcz,lefever,rplante,whs}@uiuc.edu
[2]AT&T Labs Research
Florham Park, NJ 07932, USA
{hiltunen,rick}@research.att.com

Abstract

Traditional techniques for building dependable, high-performance distributed systems are too expensive for most non-critical systems, often causing dependability to be sidelined as a design goal. Nevertheless, systems are expected to be dependable, and if dependability could be provided at a lower cost, many applications would stand to benefit. We believe that compiler techniques can be used to create novel and enhance existing dependability mechanisms to create a wider range of cost/dependability tradeoffs than is currently available. Similarly, compilers can assist in the area of error detection by expanding the range of errors that can be detected. New compiler techniques, combined with model-driven adaptation and control mechanisms, can be used to dynamically guide a system as it makes choices, with cost, dependability, and performance tradeoffs, in response to the occurrence of faults and changes in the environment. This paper reports on a new project that is exploring the approach. The broad goal of the work is to create a powerful yet flexible run-time environment for dependable and high- performance systems that operate within much lower cost constraints than is currently possible.

1. Introduction

Traditional approaches for constructing dependable distributed systems using commodity hardware typically involve an expensive combination of replicated software execution, redundant computation and storage resources. With abundant resources, those primarily static application designs are sufficient to provide a specified level of performance and dependability. Such approaches are important for certain safety-critical (e.g., aircraft control) and mission-critical (e.g., military decision support) systems, both because static designs are inherently simpler than more complex dynamic approaches, and because the high costs of the underlying systems may be justifiable. Such solutions are clearly not acceptable, however, for the growing number of distributed applications with inter-operating compute- and data-intensive components that must meet more stringent cost constraints and operate at a specified performance level. Such applications will dominate computing applications of the future, and are likely to span all major computing domains, including commercial (e.g., large-scale Web services), scientific (e.g., data-driven Grid computing), governmental (e.g., emergency response systems), and military (e.g., operational coordination). Moreover, applications in those areas are often highly variable in their computing, bandwidth, and storage demands, and can incur component failures, local network failures, and demand spikes. In spite of the challenges, the common requirement for those important but complex applications is that they perform as expected despite faults that may occur.

Highly flexible, adaptive system management is needed to achieve this requirement. In particular, in order to meet application performance and dependability constraints in a cost-effective way, applications must be able to reorganize their computations and/or reallocate resources in response to changing operating conditions. Achieving solutions of the greatest flexibility requires the combination of information from applications (via compiler analysis) and runtime environments (via measurements) to enable systematic prediction and control of application behavior (via online models). In order to build such adaptive high-performance and dependable distributed systems, we believe fundamental new techniques in three multidisciplinary areas will be needed: compiler-driven flexible dependability mechanisms, model-based runtime prediction and control, and compiler-guided measurements for early error detection.

This paper gives an overview of a project, supported by the National Science Foundation, that aims to create a compiler-enabled model- and measurement-driven adaptation environment for dependability and performance, by developing and integrating new techniques in each of the three areas. If we are successful, the developed environment will make it dramatically easier to create highly flexible, high-performance, and dependable distributed systems subject to prescribed cost constraints.

We have identified several important ways in which integrated techniques from the above three areas can be more flexible than uncoordinated ones. They include examples from compiler-based replication strategies, compiler-driven measurement, measurement-guided online models, and compiler-assisted model construction on online models. In the rest of this section, we first discuss the limitations of current approaches to adaptive system management and explain the motivation for integrating work in the three areas identified. We then summarize our main research goals. In doing so, we also briefly summarize our plan for using two concrete, realistic applications to guide this work: one from the scientific domain (the CARMA image pipeline, which is a data-intensive Grid application for radio astronomy) and one from the commercial domain (iMobile, an enterprise mobile services platform). Due to their complementary requirements, the examples will challenge us to create a generic, tunable solution.

1.1. The Need for Integrated Approaches to Adaptive System Management

The primary dependability mechanisms applied in distributed systems have been computation replication and checkpointing. Traditional replication schemes consume significant computational resources, but have been used in dependable middleware and runtime systems because they do not require significant application-specific knowledge. Compiler transformations can be used to develop new replication techniques that could provide a much richer set of dependability mechanisms while remaining fully automatic. While there has been extensive work on compiler-supported checkpointing schemes (which we will use and extend, where possible), there has been little previous work on compiler-enabled replication schemes, or on compiler-based schemes that integrate replication with checkpointing. Finally, modern dynamic code generation techniques (and the increasing speed of hardware) can allow practical, online generation of replicas, further reducing the performance penalties of replication and greatly increasing the flexibility of adaptive, dependable systems.

Measurement techniques for detecting errors and performance anomalies are also crucial because they can help minimize the performance impact of such events, which can be relatively frequent in large distributed systems. Again, however, the applicability of measurement techniques may be significantly limited in the absence of compiler support, because many programming errors are either difficult or very expensive to detect purely through runtime monitoring. By building on recent advances in compiler technology for a wide range of static error-checking problems and combining them with runtime monitoring techniques, we could greatly expand our error detection capabilities for complex software.

It is necessary to use online models to guide runtime adaptation in order to respond to changing environmental conditions and changes in requirements. Recently, there has been significant interest in modeling techniques for guiding runtime adaptation (e.g., [1, 2, 7, 20, 21, 24, 25]), but most of that work has focused on adaptation for performance alone, not considering faults and failures that may occur. Initial work by one of the authors of this paper (i.e., [15, 16, 17, 18]) suggests that simple online models, solvable in a few hundred microseconds at the time of remote method invocation, can be used to make intelligent decisions about how to serve a call to meet performance requirements while simultaneously providing dependability.

Another key opportunity related to online models is the possibility of simplifying the process of constructing dependability models by using compiler techniques to partially automate it. Much of the past work on compiler-supported modeling techniques (including our own, e.g., [2, 3, 11, 12]), has focused solely on performance modeling (because compilers have traditionally focused on optimization and parallelization). Because there are significant differences between dependability properties/models and their performance counterparts, extending those techniques or developing new ones to support combined performance/dependability modeling will require significant new research.

1.2. Summary of Research Goals

The challenges discussed above motivated four specific goals for our joint research. Specific approaches and potential solutions we will explore are discussed in more detail in later sections of the paper.

Providing Compiler-Based Flexible Dependability Mechanisms: We will use program analysis and transformations to develop novel dependability mechanisms that are more powerful and higher-performing than those that operate without compiler support. In addition, because the space of possible transformations is very large, we will develop dynamic code generation approaches for generating and optimizing code embodying the developed mechanisms at runtime, under the control of the runtime adaptation controller.

Creating Efficient Model-Based Runtime Prediction and Control: We will develop techniques to predict the impact of a changing environment and generate controlling actions to meet dependability, performance, and resource demands. We will do so using 1) offline modeling tools that help create models of the relationships between system and environment, 2) a compiler that facilitates automatic generation of models, 3) online and/or offline decision procedures that combine these models with measured data, and 4) runtime environments that implement the chosen control actions.

Measurement-Driven and Compiler-Enabled Early Error Detection: We will combine error measurements of operational systems with online analysis to extract error symptoms for early error detection (e.g., error patterns and execution patterns), and thus minimize performance impact of failures [14, 13]. Such analysis will provide the parameters and distribution characteristics needed to enable adaptive system models and thus ensure effective online control. We will use compiler support to develop application-specific, preemptive detection techniques to improve coverage and minimize error propagation while maintaining performance.

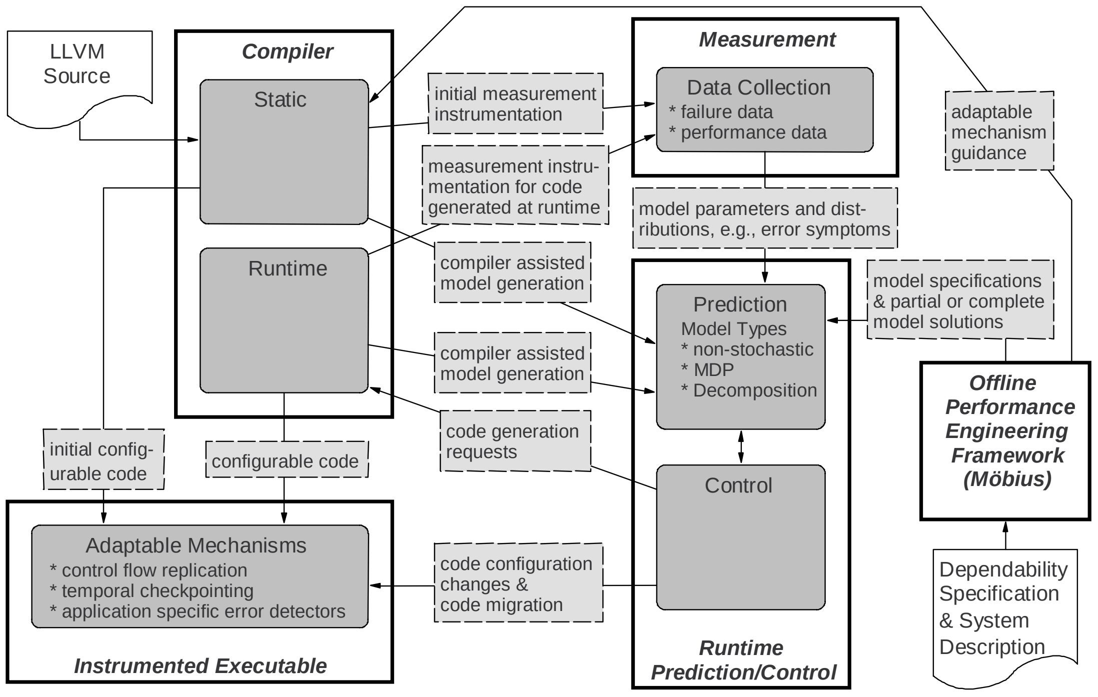
Validate and Apply Developed Techniques on Significant Scientific and Commercial Applications: To provide a concrete focus and a basis for experimental evaluation, we will work with realistic distributed applications from two different domains: one from the scientific domain (a data-intensive Grid-based application) and one from the commercial domain (an enterprise mobile services platform). Together, the two applications present a diverse set of application requirements, operating environments, and system configurations, providing a good test for the developed environment.

2. Integrated Solution Overview

In this section, we present a high-level view of our solution to the needs introduced in Section 1. Our solution integrates performance- and dependability-aware compilation with a runtime environment supporting prediction, control, and measurement. Figure 1 illustrates the interaction among the five logical components of our design, namely the compiler, the instrumented executable, runtime prediction and control, measurement, and the offline performance engineering framework (e.g., Möbius [10]). The inputs to our environment are source code for an application program (compiled with the Low Level Virtual Machine (LLVM) instruction set [19]). A dependability specification for the application that can be used to manage dependability and performance requirements, and a description of the available system resources.

We propose several adaptation mechanisms. They include compiler-generated code versions embodying different forms of code replication (including some novel ones), integrated checkpointing and replication schemes, and application-specific error detectors. Adaptive management can also be used to control many traditional fault tolerance mechanisms (e.g., ways choose the number and location of replicas) and to choose between alternative mechanisms. To initialize the process, the static compiler instruments an application with an appropriate initial set of adaptive mechanisms before deployment, with aid from the offline performance engineering framework.

Adaptive decision-making is performed by the runtime control unit and guided by the runtime prediction unit. The control unit queries the prediction engine to determine the application's dependability/performance needs, e.g., information about dependability-critical regions of code. Prediction is supported by modeling the application and its environment. We consider three modeling approaches: non-stochastic models, Markov decision processes (MDPs), and decomposition. Models are obtained in two ways. First, the offline performance engineering framework is used for manual model



LLVM Source
Compiler
Static
Runtime
initial measurement instrumentation
measurement instrumentation for code generated at runtime
compiler assisted model generation
compiler assisted model generation
code generation requests
Measurement
Data Collection
* failure data
* performance data
model parameters and distributions, e.g., error symptoms
adaptable mechanism guidance
model specifications & partial or complete model solutions
Prediction
Model Types
* non-stochastic
* MDP
* Decomposition
Control
Runtime Prediction/Control
Offline Performance Engineering Framework (Möbius)
Dependability Specification & System Description
initial configurable code
configurable code
Adaptable Mechanisms
* control flow replication
* temporal checkpointing
* application specific error detectors
Instrumented Executable
code configuration changes & code migration

Figure 1. Adaptation Environment

specifications and also to provide partial, and in some cases complete, model solutions. Second, the compiler enables partially automatic model construction by generating model components for aspects of a program.

The runtime control unit uses the predictions to choose a policy that meets the application's requirements. Adaptation requests may be either configuration change requests, which are made directly to the instrumented application through hooks provided by the compiler, or code generation requests to the runtime compiler for a code version not currently available for a relevant code segment. Upon generation of the requested code, the runtime control unit performs migration of the new code to the application.

The final component of the design is the measurement engine. It serves 1) to derive error symptoms for predicting the onset of major dependability problems, 2) to provide insight on why errors escape, and 3) to assess coverage and performance of individual techniques. The data collected by the measurement component is provided as input to the runtime prediction unit to guide adaptation decisions. Both the static and runtime compilers assist in measurement by instrumenting the application for data collection.

In the following three sections, we examine the three major aspects of the above solution, namely, compiler-based dependability mechanisms, measurement-driven and compiler-enabled early error detection, and model-driven decision making.

3. Compiler-Based Dependability Mechanisms

Replication and checkpointing are often used to provide dependability, and there exist several algorithms and protocols with varying dependability and performance characteristics that implement these basic mechanisms. Our research aims to strengthen rather than replace the existing choices, in three ways: 1) new low-cost variants or alternatives for replication and checkpointing that can detect or tolerate a wider range of fault types through the use of compiler support; 2) ways to reduce the potential cost of both traditional and new replication/checkpointing strategies, by extracting information about program behavior via compiler analysis; and 3) mechanisms that allow a distributed application to switch between different performance- and dependability-enhancing techniques dynamically in an efficient and safe manner.

The fundamental insight we aim to exploit is that compiler analysis and transformations can be used to create multiple versions of a program or program component that differs from the original, and to do so automatically. We envisage two orthogonal classes of such versions, which in a dependable system can serve different purposes that are complementary to existing replication strategies.

One class of compiler-generated versions we propose is that of transformed replicas, i.e., ones that use data layouts and computation sequences that are different from those in the original code, but perform equivalent computations. Transformed replicas can be used to detect a class of runtime errors that are not caught by identical replicas. The second class of variant replicas is that of partial replicas, i.e., replicas that perform only a subset of the computations of the original code, using a subset of the data. The key advantage relative to transformed or exact replicas is that the partial replica may have much lower time and resource costs, offering a new set of choices to the runtime controller in a dependable system.

To accomplish the second and third goals of improving the efficiency of existing dependability techniques and enabling dependability mechanism switching during runtime, we rely on the insight that compiler techniques could also create relatively simple models that predict the relative cost of a variant replica or code fragment compared with the original. Such models could not only be used to guide static or dynamic decisions on the use of different combinations of variant/partial replicas for replication-based solutions, but could also be used to reduce the cost of existing dependability techniques such as checkpointing or passive replication.

For example, when checkpointing is being used, compiler analysis could be used to identify subsets of the code for which the volume of data generated is large but the time to recompute that data from existing data is relatively small. (Simple and common examples are the initialization or creation of a copy of a large data structure.) Instead of checkpointing this data, we can use a partial replica of the code to recompute that data from other checkpointed values (analogous to rematerialization as an alternative to spilling during register allocation [5]). The replica can be executed either at the storage end or after reading back the checkpointed data to reconstruct program state.

Similarly, for passive replication, the compiler could identify state values that could be recomputed quickly from other values. Then, only the latter values would be transferred to the passive replica at every state transfer point, while the former would be computed from the latter if an error occurred. That approach, i.e., recomputation on error, has the significant benefit that it saves cost in the common case (state saving) at the expense of added cost in the uncommon case (error recovery).

4. Measurement-Driven Compiler-Enabled Early Error Detection

Error detection is essential when making dependability and performance predictions. Measurement-based studies have shown that it is possible to predict future failures based on current and historical online error information. Our preliminary investigations have shown that instruction sequence checkers, combined with very simple program analysis and instrumentation, can be used to detect classes of errors that are not detected by other checkers. Our goal is to explore how such low-level detection techniques (e.g., at the instruction level) can be augmented with more sophisticated compiler-driven program analysis support to generate application-aware detectors for rapid detection of programming errors.

Towards that end, we are developing a theoretical framework and concrete algorithms for automated generation of system and application signatures (e.g., addresses, values, or instruction sequences) that enable detection of and recovery from runtime errors. Such errors can arise from application bugs (e.g., dangling references, data races, or numerical instability), compiler bugs, operating system bugs, system intrusions, or transient hardware errors. The idea is to use compiler support for 1) extraction of logical predicates (reflecting dependencies between state variables and instructions) that, if violated, can be used as indicators of errors, and 2) conversion of the predicates into runtime assertions embedded into the application code or hardware frameworks. Examples of specific checks (assertions) include sequences of dependent instructions, integer overflow or underflow on values used in array index expressions, legal ranges or relationships for system call parameters, and clusters of values traversed by selected variables in a program [23, 22].

Our initial study demonstrated the feasibility of combining static program analysis with hierarchical simulation to search the program state space for discovering dependencies and enabling generation of runtime assertions. We have constructed an instruction sequence checker module, which verifies whether runtime execution of instructions preserves dependency chain(s) identified via program analysis (implemented using GCC). Failure of the processor to respect the dependencies (due to transient hardware errors, for example) may produce incorrect results and should be considered an error. Initial fault injection results indicate that while coverage of this simple technique is in the range of only 15%-20%, it detects unique errors that escape other techniques, e.g., control flow checkers.

The key challenge in this work is that sophisticated compiler techniques can extract a very large number of program properties from a program. We must first determine which properties can be translated into compact and efficient checkers, and then evaluate which of these checkers are most effective for different classes of applications. The LLVM compiler system provides an effective basis for the compiler analysis in this work, because 1) it is language-independent, and yet exposes high-level type, control flow, and data flow information, and 2) it permits transparent dynamic code generation, which could be used to insert or customize checkers at runtime and to vary the set of checkers used over different time intervals for long-running applications.

5. Model-Driven Decision Making

The compiler-assisted dependability mechanisms described in Section 3, along with more traditional dependability mechanisms, offer the application a wide choice in terms of performance and dependability. However, the correct choice of dependability mechanism depends not only on the application's performance and dependability requirements, but also on factors such as application characteristics (e.g., the amount of state in the application), the environment (e.g., computing vs. communication costs), knowledge of what kinds of faults are most likely to occur (or are already occurring), and the cost of switching dependability mechanisms (perhaps requiring that the application be stopped for some time). In addition, many dependability mechanisms can be further tuned to change their performance/dependability tradeoffs (e.g., varying the number of replicas in replication schemes).

In order to quantify the effects of those factors on system performance and dependability and to choose an appropriate dependability mechanism or appropriate parameters for the chosen dependability mechanism, models of the application, its environment (including the fault environment), and the appropriate dependability mechanisms are needed. The models must take as inputs the information obtained through system instrumentation and error detectors, and their solution must indicate which mechanism is to be used in order to maximize performance, dependability, or a combination of the two.

A number of the factors listed above are nondeterministic. They include imperfect error detectors (i.e., those that give false positives or false negatives), changing workloads, and varying system resources. Therefore, we use stochastic models (e.g., queuing networks or Markov chains) of the system as a basis for our decision-making. Because there is a close relationship between decision procedures and models, a key research issue we are exploring is the applicability of various decision procedures for different kinds of applications, and different kinds of dependability and performance properties.

For stochastic systems or systems with discrete adaptation actions that introduce radical changes in system dynamics (such as mechanism switching), we believe that Markov decision processes (MDPs) [26] with finite state spaces provide a suitable mathematical framework. Although there has been some work on using MDPs for adaptive power management [24, 27, 28], there has been very little work on using MDPs in the performance and dependability domain [29]. We are investigating how dependability and performance properties can be represented in terms of MDP rewards, how the corresponding mechanisms can be represented in terms of the MDPs, and how optimal decisions can be generated to achieve desired levels of these properties.

Because MDPs are extensions of Markov chains, we will use the considerable body of research on modeling the performance and dependability of systems using Markov chains to our advantage. In addition, there exist a wide variety of languages based on Markov chains for describing complex systems in a compact high-level representation. We have implemented several of the languages in the Möbius framework. We will extend the languages (and the Möbius framework) to allow representation of adaptation choices and facilitate the automatic generation of MDPs.

Unfortunately, MDPs that model the combined effects of both performance and dependability can lead to state-space explosion. We propose several techniques to help mitigate this problem. The key insight we will rely on is that the goal of online decision-making is not to compute very accurate values of required measures, but to generate enough insight to make a choice between alternative decisions. That means that approximate models for online decision-making may be good enough. An open research question that we intend to address is that of how accurate the models have to be to generate good control decisions.

Finite Horizon Approximation: One way to reduce solution time is to project MDP trajectories to only a finite horizon in the future. It is an approximation technique, because we must have a value function that identifies the quality of different states at the finite horizon (since we cannot look beyond it).

Reduced State-Space Computation: Another way to reduce state space is to consider only subsets of the possible state space reachable from the current state.

Model Hierarchies: In constructing small (but possibly approximate) models, hierarchy and decomposition have been shown to be very effective tools [Yu97, Li99]. We believe that compiler-based techniques can be very helpful in generating such hierarchical models. In particular, we have done research in automatic generation of task graph models of programs using compilers [3]. The task graphs could be used in conjunction with stochastic models of resource allocation and consumption to predict execution times [4]. The execution times could then be fed to higher-level state-based models of the behavior of dependability and performance mechanisms in order to make decisions for adaptation.

Offline Solution and Symbolic Techniques: For systems and models in which large state spaces are unavoidable, we propose to generate controllers using MDP solutions offline. Research done on symbolic state-space representation and solution techniques for Markov chains (e.g., [9]) indicates that extremely large state spaces and controller functions on them can be represented compactly using symbolic data structures such as Binary Decision Diagrams (BDDs) and Multi-valued Decision Diagrams (MDDs) (e.g., [6, 8]). Although that approach would not reduce solution time, it would remove model solution from the runtime control path, thus making online control decisions very rapid.

6. Conclusion

In this paper, we have reported on our vision for a compiler-enabled model- and measurement-driven adaptation environment for dependability and performance. The project aims to provide a design and runtime environment that will help distributed systems designers construct systems that meet their performance and dependability goals, but at a much lower cost than traditional static dependability mechanisms. In order to achieve the vision, we proposed three complementary approaches that are all assisted by compiler techniques. The first approach was the use of compiler transformations to 1) build variant and partial replicas that have dramatically different cost/benefit tradeoffs than existing replication mechanisms and 2) improve the efficiency of existing checkpointing and replication mechanisms. The second approach was to use compiler analysis to automatically build application-aware error detectors that can detect classes of errors (e.g. programming errors) not easily detectable by the current state of the art. Finally, the third approach involved the use of stochastic models, based on Markov decision processes, of the system, its environment, and dependability mechanisms to build controllers that dynamically change/adapt the dependability mechanisms to achieve the best cost/benefit tradeoff under the existing conditions and fault environment.

Acknowledgements This material is based upon work supported in part by the National Science Foundation under Grant No. 0406351. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] T. F. Abdelzaher, K. G. Shin, and N. Bhatti. Performance guarantees for web server end-systems: A control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems*, 13(1):80–96, 2002.

[2] V. Adve, A. Akinsanmi, J. C. Browne, D. Buaklee, G. Deng, V. V. Lam, T. Morgan, J. R. Rice, G. J. Rodin, P. J. Teller, G. Tracy, M. Vernon, and S. Wright. Model-based control of adaptive applications: an overview. In *Proceedings of the Workshop on Next Generation Software*, April 2002.

[3] V. Adve and R. Sakellariou. Compiler synthesis of task graphs for parallel programs. In *Proceedings of the 13th International Workshop on Languages and Compilers for High Performance Computing (LCPC)*, pages 208–216, August 2000.

[4] V. Adve and M. K. Vernon. Parallel program performance prediction using deterministic task graph analysis. *ACM Trans. on Computer Systems*. to appear.

[5] P. Briggs, K. D. Cooper, and L. Torczon. Rematerialization. In *Proc. ACM SIGPLAN Conf. Prog. Lang. Design and Implement.*, pages 311–321, 1992.

[6] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comp.*, 35(8):677–691, August 1986.

[7] D. Buaklee, M. K. V. G. Tracy, and S. Wright. Near-optimal adaptive control of a large grid application. In *Proceedings of the 16th Annual ACM International Conf. on Supercomputing (ICS 2002)*, pages 315–326, June 2002.

[8] J. R. Burch, E. M. Clarke, and K. L. McMillan. Symbolic model checking: 1020 states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[9] G. Ciardo and A. Miner. Efficient reachability set generation and storage using decision diagrams. In *Lecture Notes in Computer Science: Proc. 20th Int. Conf. Application and Theory of Petri Nets*, volume 1630, pages 6–25, June 1999.

[10] D. D. Deavours, G. Clark, T. Courtney, D. Daly, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. G. Webster. The Möbius Framework and its Implementation. *IEEE Transactions on Software Engineering*, 28(10):956–969, 2002.

[11] E. Deelman, R. Bagrodia, R. Sakellariou, and V. Adve. Improving lookahead in parallel discrete event simulations of large-scale applications using compiler analysis. In *Proceedings of the 15th Workshop on Parallel and Distributed Simulation*, pages 5–13, May 2001.

[12] B. J. Ensink, J. Stanley, and V. S. Adve. Program control language: A programming language for adaptive distributed applications. *Journal of Parallel and Distributed Computing*. to appear.

[13] R. Iyer, Z. Kalbarczyk, and M. Kalyanakrishnam. Measurement-based analysis of networked system avilability. *LNCS*, 1769, 2000.

[14] R. K. Iyer, L. Young, and P. V. Iyer. Automatic recognition of intermittent failures: An experimental study of field data. *IEEE Trans. on Computers*, 39(4):225–237, 1990.

[15] S. Krishnamurthy, W. H. Sanders, and M. Cukier. A dynamic replica selection algorithm for tolerating timing faults. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN-2001)*, pages 107–116, July 2001.

[16] S. Krishnamurthy, W. H. Sanders, and M. Cukier. An adaptive framework for tunable consistency and timeliness using replication. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN-2002)*, pages 17–26, June 2002.

[17] S. Krishnamurthy, W. H. Sanders, and M. Cukier. Performance evaluation of a probabilistic replica selection algorithm. In *Proceedings of the 7th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2002)*, pages 119–127, January 2002.

[18] S. Krishnamurthy, W. H. Sanders, and M. Cukier. Performance evaluation of a qos-aware framework for providing tunable consistency and timeliness. In *Proceedings of the 2002 10th IEEE International Workshop on Quality of Service (IWQoS 2002)*, pages 214–223, May 2002.

[19] C. Lattner and V. Adve. Llvm: A compilation framework for life-long program analysis & transformation. Technical Report UIUCDCS-R-2003-2380, University of Illinois Department of Computer Science, September 2003.

[20] B. Li and K. Nahrstedt. A control-based middleware framework for quality of service adaptations. *IEEE Journal on Selected Areas in Communications*, 17(9):1632–1650, Sept. 1999.

[21] Y. Lu, T. F. Abdelzaher, C. Lu, and G. Tao. An adaptive control framework for qos guarantees and its application to differentiated caching. In *Proc. Tenth International Workshop on Quality of Service (IWQoS)*, pages 23–32, May 2002.

[22] N. Nakka, G. P. Saggese, Z. Kalbarczyk, and R. K. Iyer. An architectural framework for detecting process hangs/crashes. In *Proc. of EDCC-5*, 2005.

[23] N. Nakka, J. Xu, Z. Kalbarczyk, and R. K. Iyer. An architectural framework for providing reliability and security support. In *Proc. of DSN*, 2004.

[24] G. A. Paleologo, L. Benini, A. Bogliolo, and G. D. Micheli. Policy opti-mization for dynamic power management. In *Proceedings of the 35th Design Automation Conference*, pages 182–187, June 1998.

[25] S. Parekh, N. Gandhi, J. Hellerstein, T. J. D. Tilbury, and J. Bigus. Using control theory to achieve service level objectives in performance management. In *Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management*, pages 841–854, May 2001.

[26] M. L. Puterman. *Markov Decision Processes*. Wiley Interscience, 1994.

[27] Q. Qiu and M. Pedram. Dynamic power management based on continuous-time markov decision processes. In *Proceedings of the Design Automation Conference*, pages 555–561, June 1999.

[28] Q. Qiu, Q. Wu, and M. Pedram. Dynamic power management of complex systems using generalized stochastic petri nets. In *Proceedings of the 37th Design Automation Conference (DA'2000)*, pages 352–356, June 2000.

[29] K. Yu. *Reduced State Space Markov Decision Process and the Dynamic Recovery and Reconfiguration of a Distributed Real-Time System*. PhD thesis, University of Massachusetts, 1997.