SOFTWARE ENABLEMENT FOR MULTICORE ARCHITECTURES

David Bernstein

IBM Haifa Research Laboratory
The Haifa University Campus
Haifa, Israel 31905
Email: bernstn@il.ibm.com

ABSTRACT

A recent trend of transitioning to multicore architectures in the mainstream market segments creates significant challenges for programming systems. The market need for creating portable multithreaded applications that exploit high performance of chip multiprocessors is not easily supported by existing programming models and languages, compiler technology, performance analysis and testing tools. We overview various research directions in this area and present some partial results achieved by the industry.

1. INTRODUCTION

The computer architecture world is going through a significant change in the past several years. Since the eighties and nineties of the last century, based on the Moore law formulation new generations of superfast processors were coming out every 18 to 24 months, where the new version of hardware had the clock rate usually twice or even more faster than the previous one. So the focus of the computer architects and the compiler people at that time was merely on how to increase the Instruction Level Parallelism (ILP), both in the processor hardware and the compiler optimization techniques and also combination of both. Though it appeared very soon that more and more cycles are being spent not in the processor core execution, but in the memory subsystem which includes the multilevel caching structure, and the so-called Memory Wall problem started to evolve quite significantly due to the fact that the increase in memory speed didn't match that of processor cores. More importantly, a few years ago a new evidence came out showing that it's very hard and not economical to speed the processor clock further due to tremendous increase in power consumption and power dissipation by the advanced computer systems. This phenomenon received the name of the Power Wall problem, and the net result is that most of the industrial manufacturers of advanced processors such as Intel, AMD, Sun and IBM ceased coming out with ever increased clock rate processors and started to consider for alternative ways of providing increased performance in their computer offerings.

Very soon a new direction for increasing the overall performance of computer systems had been proposed, namely changing the structure of the processor subsystem to utilize several processor cores on a single chip. These new computer architectures received the name of Chip Multi Processors (CMP) and allowed to provide increased performance for new generation of systems, while keeping the clock rate of individual processors cores at a reasonable level. The result of this architectural change is that it is possible to provide further improvements in performance while keeping the power consumption of the processor subsystem almost constant, the trend which appears essential not only to power sensitive market segments such as embedded systems, but also to computing server farms which suffer of power consumption/dissipation problems as well. One can argue that the exploitation of multiple processors in computer systems was known already for several decades, but the claim is that in the past it was merely limited to very expensive high-performance systems (HPC) or very specialized embedded offerings which were based on a collection of DSPs. These days, almost every regular desktop or mobile computer you can buy in a store near you will most likely include several processor cores inside their box, not talking about powerful server farms and sophisticated game consoles such as Xbox 360 or Playstation 3. So, the accelerated trend is that CMPs are pervasive in almost every segment of the market, and the question is what does it mean to the hardware and software parts of the computer industry?

The focus of this overview paper is on how CMPs affect the software development processes, i.e. how we program for multicore architectures, how we test and evaluate the performance of multithreaded applications, etc.

Complimentary to this article, a long discussion can be done on how CMPs impacted the computer architecture trade-offs and design decisions. It can be mentioned that the trend is not only "just" having several processor cores on a chip (each of them potentially supporting several parallel threads), but also having specialized computing engines for important market needs (e.g. XML processing, cryptography, graphics), there are questions of how we are going to interconnect a large number of processor cores (e.g. network on a chip), how we are going to provide them sufficient memory bandwidth, how we are going to structure the multilevel caching subsystem, etc. Then the overarching question is how we balance the general purpose computing resources with specialized processing engines and all the supporting memory, caching and interconnect structure, given a constant power budget. Even the evaluation metrics for computer systems recently started to cite also performance per watt figures which present new challenges to making design trade-offs in computer products. Overall it's exciting time for the computer architecture community and a great summary of the recent trends can be found in [1].

Programming multiprocessor systems appear to be much more complex that uniprocessor computers. Most of the existing algorithms in the literature were formulated and proposed several decades ago, when the notion of multiprocessing was very young. It's true that multiprocessing can be used to run several independent programs in parallel without any investment in reprogramming the application software, but this will severely limit the exploitation of multiprocessors, especially when the number of computing cores in the mainstream market will increase beyond the existing 2 to 4 cores on a chip. In addition, taking advantage of specialized processing cores such as in IBM's Cell processor and providing all the needed data movement inside the processor can't be done efficiently using the regular high-level C programming paradigm. So, there are two main directions to pursue programming for multiprocessors: explicit manual programming and exploitation of the combination of the software compiler optimization, the build tool chains, and the run-time subsystems.

In the past both in the HPC and embedded communities, the emphasis was more on explicit manual programming for multiprocessors and special resources by expert programmers. In this explicit programming world the requirement was to allow access to every single resource in the computing system, either by means of assembler level programming or by special directives at the high-level programming languages. Then it was up to the expert programmers to manually parallelize the program, evaluate its behavior, discover the bottlenecks and tune the performance. Usually this resulted in numerous home-grown language directives/extensions, internal tools, obscure run-time systems, etc. The net result of this direction is collection of application software programs that is tuned for a specific architecture and hardly portable to new generations of hardware, long time cycles for porting software to future architectures, strong dependence on expert programming, etc. which makes the transition to CMPs in the mainstream market as unsustainable. Consider, for example, IBM's Cell processor. It was originally created for the gaming consoles market, and today it is the heart of Sony's Playstation 3. In the gaming market, manual programming is a dominating paradigm, and as such, we assume Cell can be programmed successfully to develop a collection of gaming titles for Playstation 3. However, when Cell is considered for broader market of numeric extensive computing such as medical imaging, seismic exploration, graphics acceleration, EDA optical applications, etc., the need for the software enablement ecosystem appears to be crucial for Cell's success.

In this article we would like to overview some of the essential parts of the software enablement system for CMPs and discuss various directions for providing solutions to it. We would like to emphasize that this is an active area of research, and there are only some early results in the academic and industrial worlds in terms of established standards and technology, but much more will evolve in the years to come. In the next section we cover the programming models and the compiler support for CMPs. Then, in Section 3 we discuss the performance evaluation tools, while in Section 4 we present the testing tools. We conclude our observations in Section 5. In terms of references, we don't intend to cover all the great deal of work going on in the world, but rather provide some sample of relevant sources (more extensive list of references can be found in [1]).

2. PROGRAMMING MODELS AND COMPILER OPTIMIZATION

As discussed in the introduction, the main question that is being asked is how to protect the investment in development of software applications for CMPs. The intent is to promote a paradigm in which an application program is written once in a portable way, such that it can be ported/recompiled to new generations of the same computer architecture or even to alternative architectures in a more or less automatic or semiautomatic way. Of course, it's a vision for the future, and currently we are far away from achieving it.

In the past there was a hope that compilers will provide means for automatic parallelization of sequential

programs. There are multiple barriers to this in languages such as C/C++ and Java, but even in FORTRAN for loop intensive programs, the success of automatic parallelization is very limited. Taking into account this great volume of research on automatic parallelization done in the last 3 decades, we postulate that the right approach to programming CMPs has to be based on a symbiosis of programmers providing information on a parallel structure of a program in the right level of abstraction, and the compiler/run-time translating this into the executable code for CMPs. The question then is what are this programming language and/or the programming model in which the programmer will have to express his/her knowledge about the parallel structure of the program?

The evolution of the programming languages demonstrates that less and less new languages were invented in the last 2 decades or so. The success of Java that was released at first in 1995 was quite surprising, but it can be attributed to the success on Internet and Java's value proposition as write-once run-everywhere structure. So, Java is not just a new programming language, but it's a whole new programming environment (including the virtual machine, the interpreter, the garbage collection component, the set of libraries, etc.) and paradigm for developing Internet applications. The question is whether CMPs can spur development of a new language/environment for parallelism like Java did for Internet? There are several attempts to provide new parallel programming models, languages, and libraries for CMPs, where programmers can provide additional information related to the parallelism in their programs. Among these attempts are map-reduce [2], cilk [3], UPC [4], X10 [5] and STAPL [6], but we don't observe at the moment a dominating paradigm evolving from here.

CMP architectures provide multiple types of parallelism, both in terms of thread-level parallelism (TLP) and data-level parallelism (DLP) also known as vectorization or simdization (from single-instruction-multiple-data - SIMD). Due to the larger overhead of spawning, communicating and synchronizing with threads, compared to the use of SIMD instructions, one usually seeks to exploit DLP's fine-grain parallelism across innermost loops or local code segments, whereas TLP's coarse-grain parallelism is looked across outer-loops or large code segments as much as possible. The use of data-level parallelism is provided by several compilers both for explicit programming using vector types [7] and vector intrinsics [8], and by automatic vectorization optimizations focusing on code inside loops and also outside loops ([9] and references within.)

The use of thread-level parallelism is provided by several compilers and supporting libraries, both for explicit programming using OpenMP ([10], the prevalent standard for shared-memory models), MPI ([11], the prevalent standard for distributed-memory models) and explicitly using, for example, pthreads or Java threads, and by automatic parallelization optimizations which often focus on outer loops. Most of the original auto-parallelizing compilers [12] focused on FORTRAN, due to its stronger aliasing rules than C. As we mentioned above, it is still difficult for a compiler to make efficient coarse-grain decisions with auto-parallelization [13], while on the other hand the compiler technology for vectorizing loops appear to be quite mature [9, 14].

To summarize this discussion, our feeling is that OpenMP evolves as an interesting and portable way for programmers to express parallelism in the application programs, leaving the job of distributing the parallel work among CMP cores as well as vectorizing for SIMD for the compiler. One question regarding OpenMP is that it assumes the shared-memory model. As we touched above, there are additional challenges posed by recent multicore architectures which are asymmetric, such as the Cell processor which doesn't assume the shared memory model. For such platforms, software must first be partitioned and compiled for different types of processors on the chip. Fortunately an automation of this process is currently under development inside a compiler [15] which shows this is a problem that can be managed by the compiler. Another much harder question regarding OpenMP is how it can be used for programs in which thread-level parallelism is not embedded inside the loops. This is an open research problem, and one example of an attempt to confront with it can be found in [16] where extensions to OpenMP are proposed for streaming programming model for CMPs like the Cell processor.

Complementary to compiler optimization, there is an important binary level post-link optimization which targets mainly the instruction cache and data cache utilization problems. Since we expect the size of L1 instruction and data cache per processor core not to grow in CMPs, the feeling is that the importance of this post-link optimization technology will increase in the future.

FDPR-Pro [17, 18] is a feedback directed post-link optimizer that is applied directly on program executable files. FDPR-Pro is available for IBM PowerPC platforms (AIX and Linux) and for the Cell processor. The importance of a post-link optimizer such as FDPR-Pro derives mainly from its global view property which considers the entire program executable as a single optimization unit together with accurate and complete profiling information gathered on a representative workload. Consequently, FDPR-Pro is able to perform aggressive global optimizations such as global code reordering, global static data reordering and global code inlining performed on the entire program code.

This global restructuring ability together with the fact that FDPR-Pro is applied at the end of the build chain and thus linkage conventions need not be preserved provides significant additional optimization potential to commercial applications. Global code reordering also provides code straightening that has important synergistic effects when preceded by aggressive function inlining as shown in [19]. Global code reordering of FDPR-Pro is also preceded by many other optimizations such as eliminating spills in procedure boundaries [20], global scheduling, aggressive loop unrolling and others. At the same time, global static data reordering provides room for eliminating TOC load instructions (as shown in [21]) as well as improving data cache locality. As a result of FDPR-Pro's global optimizations, the footprint of the frequently executed code (including referenced data in these areas) is reduced and cache utilization is increased. The improvement, due to better cache utilization, becomes more significant as the gap between the application's code and data footprint and the available cache size, increases. Such a gap exists in very large applications like database engines or applications built on top of them, e.g. ERP systems, as well as smaller size applications that run on limited cache environment, e.g. in embedded systems. The current trend of multicore platforms, where parallel execution is preferred at the expense of cache size, suggests that the benefit of global code and data optimizations will be vital. One example of additional effects of multiprocessing is the possible increase of false sharing that produces unneeded cache misses, thereby creating yet new challenges for post-link global optimization technology like FDPR-Pro.

3. PERFORMANCE ANALYSIS TOOLS

The task of performance analysis and evaluation of complex programs was challenging, even when they were running on single core processors. Porting these programs and evaluating their performance on CMPs appears even more difficult. Program instrumentation and tracing are the usual techniques to obtain program execution profile for analysis of program's behavior and its performance bottlenecks. The FDPR-Pro technology mentioned above is capable to instrument and profile a wide range of application programs. The profiling information includes records of execution counts at the fine grain of basic block level including the control flow between them. Such profiling can also provide an important means to obtain coverage information that is needed for testing large sub-systems. Post-link code coverage turns out to be very useful to programmers. Both profiling and coverage can be visualized and presented to programmers in a convenient way, relating the execution information to the original source of the program at hand. Two useful tools are built on top of the FDPR-Pro technology: Code Analyzer [22] that enables GUI based analysis of binaries for the purpose of performance debugging, and BProber [23] which enables user provided instrumentation and patching of executable binaries. These technologies are part of a more comprehensive performance toolkit, called VPA [24]. Another post-link instrumentation technology is the Diablo open source framework [25]. Diablo is an API based framework that enables the user to build tailored tools that perform binary editing. Diablo currently supports ARM, i386, MIPS, IA64 and Alpha architectures. Several tools were built over the Diablo framework such as compactors (aka Diablo), FIT (a Flexible Instrumentation Toolkit) [26] and Lancet (A Nifty Code Editing Tool) [27].

Comparing the performance evaluation task of single-threaded and multi-threaded applications, we notice that the latter performance is heavily influenced by thread interaction, as opposed to the former systems, which have only one or few threads active at the same time. A single-threaded application may need to be rewritten as multi-threaded to utilize the new hardware. As the level of parallelism increases, resource locks get hotter, possibly requiring additional application redesign. Generally, correct design of communication and synchronization protocols is critical for achieving good multi-threaded performance. Finally, even in the new multi-process, multi-threaded world performance is heavily influenced even when there is no communication, i.e. processes running in parallel can affect each other's performance, for example, by cache and TLB thrashing. Typically, there's no single thread that can be blamed for poor performance; threads replace one another on the critical path, and affect other threads of whose existence they have no knowledge.

One area where high parallelism has long been a familiar feature is scientific computing. A number of performance tools were developed to serve the needs of this community, some of which can also be used in other contexts. Examples of such tools include HPCToolkit [28], TAU [29], and Paraver [30]. In the non-HPC arena, Intel's VTune [31] is probably the best performance tool for multi-threaded programs, with its critical path analysis being a particularly useful feature. A number of tools provide (albeit limited) information on certain aspects of thread communication for Java programs, such as deadlock detection.

We argue that trace-based analysis and visualization can help performance investigation of multi-threaded

programs running on CMPs [32]. One common feature of those trace-based tools is the popularity of timeline views and data. The problem with the traditional approaches, such as profiling, is that data aggregation makes it impossible to understand precisely those communication issues that become so important nowadays. However, a serious issue with timeline data is its lack of scalability, in two dimensions. Except for very short kernels, traces tend to grow large and unwieldy, making it difficult to manipulate and visualize, moreover, making it difficult for the user to quickly grasp where to focus the investigation of performance bottlenecks. With the growing numbers of cores, h/w threads, and s/w threads created by the applications, it also becomes difficult to visualize all the threads or cores at the same time. In the HPC context, those issues could be dealt with due to the regularity of the workloads, by selecting arbitrary subset of cores/threads and arbitrary time intervals; this solution is rarely applicable for general-purpose applications. Other issues have to do with parallel programs' lack of determinism. Tracing disturbs parallel program's behavior much more significantly than profiling affected that of a single-threaded one. Additionally, it may be difficult to match two different executions, which means that even reproducing a problem or identifying whether a particular problem was resolved becomes an issue in itself. In summary, performance evaluation and visualization of multi-threaded applications is a topic for future research.

4. DEBUGGING AND TESTING TOOLS

The transition to CMPs affects the software testing processes as well, namely it requires the programmers to understand concurrent and multi-threaded programming. While multi-threaded programming was previously done by selected experts, now everyone will need to start doing it. This poses huge new challenges in debugging and testing of concurrent programs.

The second impact is that the compilers and JVMs are being modified to take advantage of CMPs. For example, under all JVMs that we are familiar with the Java memory model [33] implementation was very restricted. There was no reason to implement the complicated Java two layer memory model. As a result many bugs that existed in the code were not exposed. The move to multicore is manifesting many bugs that were hidden in the old JVMs.

In the last five years a new community of concurrent testing was created (see PADTAD workshop [34]). The transition to concurrent programming and the new challenges it poses was one of the central topics in PADTAD discussions, and it turned out that concurrent problems constitute about 10% of the bugs according to internal information we have been able to collect. We actually estimate that the number is probably higher as many of the concurrent problems manifest as irreproducible bugs that are never tracked down. Bugs such as crashes (races) or freeze (deadlocks) stay in the application reducing the up-time and causing a lot of grief. One of the reasons that concurrency problems rate so high is that under the current development approach the first testing phase that looks for concurrency problems is load testing. This is very late in the process and consequently very expensive as well as not terribly efficient. In addition, when a bug is found recreating it is a very difficult problem due to the nondeterministic characteristics of the problem.

We have been working on a tool supported methodology for improving the engineering of concurrent programs. The approach has a number of phases and when implemented has yielded clean programs. The main idea is to try to find the concurrency issues as early as possible. The approach is composed of the following steps:

- Teach developers how to write concurrent code. The focus is not on concurrent algorithms or data structures but on practical issues needed to get the application to work. We teach concurrent bug patterns, explain the concurrent programming constructs, and teach general concurrency design patterns. An example of such tutorial can be seen in [35].

- Once the code is written it needs to be reviewed. Regular review/inspection is meant for sequential code. We have developed a specialized review technique for concurrent code [36] which we teach in review workshops in many places in IBM.

- Teach developers how to do unit testing. Without proper tool unit testing can not find multi-threaded issues as there is no enough contention. First we developed synchronization coverage, a tool supported method for measuring contention [37]. Then we show how to use ConTest [38, 39] to do proper unit testing.

In addition to this early bug detection and removal techniques we also enhance the later phases of testing (function, system, and stress) by making the tests a lot more likely to exhibit bugs that exist in the code [40]. This is done by changing the internal timing of the executions in a way that is more likely to show abnormal behaviors.

We are continuing to work in this field and develop our technologies and methodologies in a number of different directions. We are developing tools for pinpointing locations of bugs [41]. The idea there is that if we have a test that we can cause the application to fail some of the time (depending on the timing, we can search on what causes it to fail, and then with further analysis guess the location of the bug). Another avenue of research is healing bugs so that the impact will not be seen [42]. Assume that a program exists in the field that sometimes locks. We have technique that will either eliminate the bug causing interleaving or make them a lot less likely to happen. The techniques are not very complicated but ensuring that we do not introduce new bugs. We are also working on architectures which will enable people in the academia to make use of our infrastructure [43]. Research is constrained by the fact that complicated infrastructure is needed to check new ideas. We would like people to be able to build on our infrastructure so that they can try out different ideas. In summary, the problem of testing concurrent programs is ten fold more difficult than the sequential ones, but the feeling is we have a good handle for it.

5. CONCLUSION

The recent trend of transitioning computer architectures to CMPs poses significant challenges to software programming systems. In order to provide good productivity in developing multithreaded applications and supporting their portability to a range of CMP architectures, programming systems have to evolve quite significantly. This requires raising the level of abstraction in which applications needs to be programmed, developing and maturing compiler optimization technology, providing advanced tools for debugging, testing and performance analysis of concurrent programs. There are partial research results cited in this paper which are encouraging, but there is still a long way to go to achieve efficient programmability for CMP architectures. IBM Research is a major player in this domain, heavily investing in many promising directions mentioned above.

6. ACKNOWLEDGMENT

I would like to thank my colleagues Marina Biberstein, Moshe Klausner, Gadi Haber, Shmuel Ur, Eitan Farchi, and Ayal Zaks for contributing to this article.

7. REFERENCES

[1] Krste Asanovic et al., "The Landscape of parallel computing research: a view from Berkeley", http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html .

[2] labs.google.com/papers/mapreduce-osdi04.pdf.

[3] supertech.csail.mit.edu/cilk/.

[4] http://upc.gwu.edu/.

[5] www.research.ibm.com/x10.

[6] http://parasol.tamu.edu/groups/rwergergroup/research/stapl/.

[7] http://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Vector-Extensions.html.

[8] http://developer.apple.com/hardwaredrivers/ve/model.html.

[9] http://gcc.gnu.org/projects/tree-ssa/vectorization.html and references within.

[10] http://www.openmp.org/drupal/.

[11] http://www-unix.mcs.anl.gov/mpi/.

[12] http://ieeexplore.ieee.org/iel5/9818/30954/01437326.pdf?arnumber=1437326.

[13] http://www.spscicomp.org/ScicomP4/Presentations/Blainey/scicomp1001.pdf.

[14] D. Nuzman and R. Henderson, "Multi-platform Auto-vectorization," in *CGO-4 (The 4th Annual International Symposium on Code Generation and Optimization)*, Manhattan, New York, March 26-29, 2006.

[15] www.research.ibm.com/cellcompiler/.

[16] Harm Munk et al., "The Acotes Project", http://www.hitech-projects.com/euprojects/ACOTES .

[17] G. Haber, E. Henis, and V. Eisenberg, "Reliable Post-link Optimizations Based on Partial Information," in *Proceedings of the 3rd Workshop on Feedback Directed and Dynamic Optimizations*, December 2000.

[18] E. Henis, G. Haber, M. Klausner, and A. Warshavsky, "Feedback Based Post-link Optimization for Large Subsystems," in *Second Workshop on Feedback Directed Optimization*, Haifa, Israel, November 1999, pp. pp. 13–20.

[19] "Aggressive function inlining: Preventing loop blockings in instruction cache," in *To be published at the HiPEAC 2008 proceedings.*

[20] G. Haber, M. Klausner, B. Mendelson, and V. Eisenberg, "Light Weight Optimizations for Reducing Hot Saves and Restore of Callee-Saved Registers," in *Proc. FDDO4 Workshop*, Dec. 2001.

[21] G. Haber, M. Klausner, V. Eisenberg, B. Mendelson, and M. Gurevich, "Optimization Opportunities Created by Global Data Reordering," in *Proc. First International Symposium on Code Generation and Optimization (CGO'2003)*, San Francisco, California, March, 2003, pp. pp. 228–241.

[22] Code Analyzer - http://www.haifa.il.ibm.com/projects/systems/cot/analyzer/index.html.

[23] BProber - http://www.haifa.il.ibm.com/projects/systems/cot/bprober/index.html.

[24] Virtual Performance Analyzer (VPA): http://www.alphaworks.ibm.com/tech/vpa .

[25] L. Van Put, D. Chanet, B. De Bus, B. De Sutter, and K. De Bosschere, "DIABLO: a reliable, retargetable and extensible link-time rewriting framework," in *Proceedings of the 2005 IEEE International Symposium on Signal Processing and Information Technology.* IEEE., 2005, pp. pp. 7–12.

[26] FIT - http://www.elis.ugent.be/fit/ .

[27] Lancet - http://diablo.elis.ugent.be/lancet_book .

[28] HPCToolkit, IBM, contact: David Klepacki.

[29] TAU, University of Oregon, LANL, and RCJ ZAM, http://www.cs.uoregon.edu/research/tau/home.php.

[30] Paraver, UPC, http://www.cepba.upc.es/paraver/.

[31] VTune, Intel, http://www.intel.com/cd/software/products/asmo-na/eng/vtune/239144.htm, http://www.intel.com/cd/software/products/asmo-na/eng/286406.htm (Thread Checker).

[32] P. Sweeney et al., "Understanding performance of multicore systems using trace-based visualization," in *STMCS workshop*, March 2006.

[33] http://www.cs.umd.edu/ pugh/java/memoryModel/jsr-133-faq.html .

[34] http://www.haifa.ibm.com/projects/verification/padtad/index.html .

[35] http://europar05.di.fct.unl.pt/tutorials.html#tut1 .

[36] A. Hayardeny, S. Fienblit, and E. Farchi, "Distributed desk checking," *Concurrency Computat.: Pract. Exper. 2007*, vol. 19, no. 3, Mar. 2007.

[37] A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur, "Applications of synchronization coverage," in *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming.* Chicago, IL, USA: ACM Press, June 15 - 17, 2005, pp. 206–212.

[38] http://w3n.haifa.ibm.com/softwaretesting/ConTest/ .

[39] http://www.ibm.com/developerworks/java/library/j-contest.html .

[40] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur, "Multithreaded Java program test generation," in *JGI '01: Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande.* Palo Alto, California, United States: ACM Press, 2001, p. 181.

[41] R. Tzoref, S. Ur, and E. Yom-Tov, "Instrumenting where it hurts: an automatic concurrent debugging technique," in *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis ISSTA '07.* London, United Kingdom: ACM Press, July 09 - 12, 2007, pp. 27–38.

[42] B. Krena, Z. Letko, R. Tzoref, S. Ur, and T. Vojnar, "Healing data races on-the-fly," in *PADTAD '07: Proceedings of the 2007 ACM workshop on Parallel and distributed systems: testing and debugging.* London, United Kingdom: ACM Press, 2007, pp. 54–64.

[43] Y. Nir-Buchbinder and S. Ur, "ConTest Listeners: a Concurrency-Oriented Infrastructure for Java Test and Heal Tools," in *to be presented at SOQUA 2007.*