TASNEEM G. BRUTCH

migration to multicore: tools that can help

Tasneem Brutch is a Senior Staff Engineer at Samsung Research in San Jose, CA. She holds a BS in Computer Science and Engineering, a Master's in Computer Science, and a PhD in Computer Engineering from Texas A&M University. She was awarded a USENIX scholarship for her Ph.D. research. She has 12 years of industry experience, working as Senior Engineer and Architect at Hewlett-Packard and Intel.

t.brutch@samsung.com

THE ADVENT OF MANYCORE SYSTEMS requires that programmers understand how to design, write, and debug parallel programs effectively. Writing and debugging parallel programs has never been easy, but there are many tools that can help with this process. In this article I provide a survey of useful tools and resources for multi-threaded applications.

Multiple threads are said to execute concurrently when they are interleaved on a single hardware resource, which limits the overall maximum performance gains from threading. When multi-threaded applications run simultaneously on different hardware, threads in an application are said to execute in parallel. To achieve software parallelism, hardware must be able to support simultaneous and independent execution of threads [1].

Performance gains through parallelism are proportional to effective partitioning of software workloads across available resources while minimizing inter-component dependencies. Performance is impacted by issues such as communication overhead, synchronization among threads, load balancing, and scalability as the number of cores changes.

It is recommended that performance bottlenecks which impact both serial and parallel applications be removed prior to parallelizing an application. This includes optimizing existing serial applications for the multicore memory hierarchy prior to parallelization.

A number of tools can be used to assist with the migration of sequential applications to multicore platforms. This article focuses on tools for C and C++ programming languages in Windows and Linux environments. Most of the tools noted here are open source or built on top of open source tools. The discussion is intended to be a starting point and is not comprehensive of all available tools. Figure 1 provides a high-level view of various categories of tools and the workflow between them [2]. Tool categories identified in the figure are discussed in this article.

Threading APIs

First, I include a brief discussion of threading APIs, as the choice of APIs may affect the selection of tools. A number of open source multi-threading programming APIs are available for both shared memory and distributed memory systems.

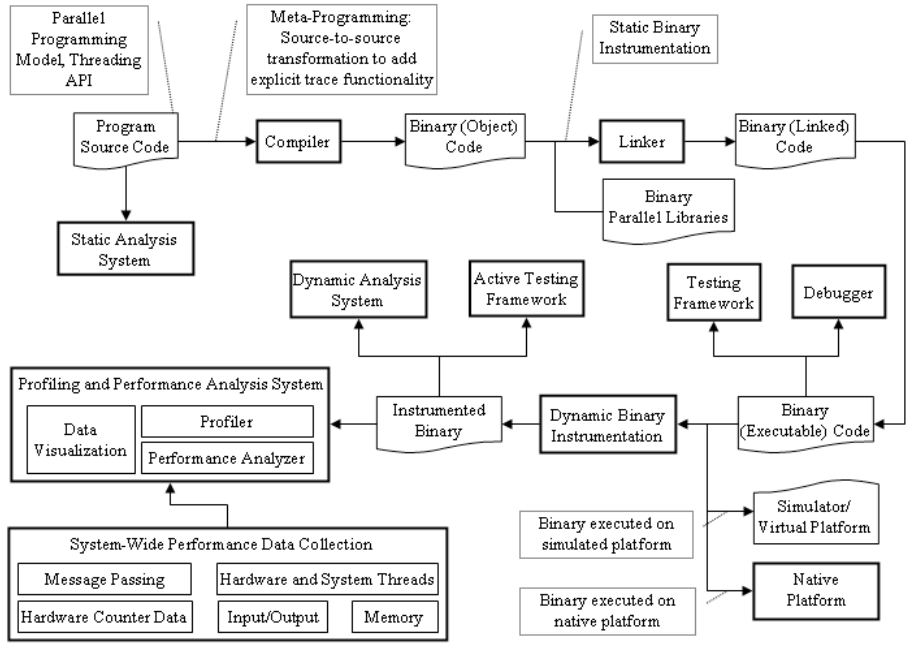

FIGURE 1: CATEGORIES OF TOOLS AND THE WORKFLOW BETWEEN THEM

MULTI-THREADING APIS FOR SHARED MEMORY SYSTEMS

OpenMP (Open Multi-Processing) is a multi-threading API, which consists of a set of compiler directives, library routines, and runtime environment variables, and is available for C, C++, and Fortran. Data may be labeled as either *shared* or *private*. The OpenMP memory model allows all threads to access globally shared memory. Private data is only accessible by the owning thread. Synchronization is mostly implicit, and data transfer is transparent to the programmer. OpenMP employs a fork-join execution model and requires an OpenMP-compatible compiler and thread-safe library runtime routines [3], [4].

Pthreads (POSIX Threads) is defined in the ANSI/IEEE POSIX 1003.1-1995 standard. It is a set of C language programming types and procedure calls which do not require special compiler support. The header file pthread.h needs to be included. Pthreads uses a shared memory model, that is, the same address space is shared by all threads in a process, making inter-thread communication very efficient. Each thread also has its own private data. Programmers are responsible for synchronizing access to globally shared data. Pthreads is now the standard interface for Linux, and pthreads-win32 is available for Windows [5].

GNU Pth (GNU Portable Threads) is a less commonly used POSIX/ANSI-C–based library. It uses non-preemptive, priority-based scheduling for multi-threading in event-based applications. All threads execute in the server application's common address space. Each thread has its own program counter, signal mask, runtime stack, and errno variable. Threads can wait on events such as asynchronous signals, elapsed timers, pending I/O on file descriptors, pending I/O on message ports, customized callback functions, and thread and process termination. A Pthreads emulation API is also optionally available [6].

Threading Building Blocks (TBB) is a C++ template library that consists of data structures and algorithms for accessing multiple processors. Operations are treated as tasks, by specifying threading functionality in terms of logi-

cal tasks, as opposed to physical threads. TBB emphasizes data parallel programming [7].

MULTI-THREADING ON DISTRIBUTED MEMORY SYSTEMS

Message Passing Interface (MPI) is a library specification for message passing on massively parallel machines and workstation clusters which supports point-to-point and collective communication. Operations in MPI are expressed as functions. The MPI standard originally targeted distributed memory systems, but now MPI implementations for SMP/NUMA architectures are also available. The programmer is responsible for identification of parallelism and its implementation using MPI constructs. Objects called "communicators" and "groups" define communication between processes [8] [9].

PLATFORM-SPECIFIC MULTI-THREADING APIS

Open Computing Language (OpenCL) is a C-based framework for pragmas for general-purpose parallel programming across heterogeneous platforms. It is a subset of ISO C99 with language extensions. The specification includes a language for writing kernels and APIs for defining and controlling a platform, and it provides online or offline compilation and build of compute kernel executables. It includes a platform-layer API for hardware abstraction and a runtime API for executing compute kernels and managing resources. It uses task-based and data-based parallelism, and implements a relaxed-consistency, shared memory model [10].

Compilers and Compiler-Based Instrumentation

A number of C and C++ *compilers* are available to programmers for compiling applications using OpenMP and Pthread APIs. Information about selection of appropriate options for OpenMP and Pthreads and inclusion of appropriate include files can be obtained from their documentation.

Figure 1 illustrates various stages of compiler-based instrumentation. Code may be modified by the compiler for generating trace information. Instrumentation may be source-to-source, static binary, or dynamic. *Source-to-source instrumentation* modifies source code prior to pre-processing and compilation. In *static binary instrumentation* the compiled binary code is modified prior to execution [11].

Static Code Analyzers

Static code analyzers help detect issues beyond the limits of runtime coverage which may not have been reachable by functional test coverage. Static code analysis is done on the source code without executing the application, requiring any instrumentation of the code, or developing test cases. Potential errors are detected by modeling software applications using the source code. These models can be analyzed for behavioral characteristics. Static analysis exhaustively explores all execution paths, inclusive of all data ranges, to ensure correctness properties, such as absence of deadlock and livelock. Static analyzers cannot model absolute (wall-clock) time but can model relative time and temporal ordering. A directed control flow graph is developed, built on the program's syntax tree. The constraints associated with variables are assigned to the nodes of the tree. Nodes represent program points, and the flow of control is represented by edges. Typical errors detected by using analysis based on a control flow graph include: illegal number or type

of arguments, non-terminating loops, inaccessible code, uninitialized variables and pointers, out-of-bound array indices, and illegal pointer access to a variable or structure. Due to the large number of possible interleavings in a multi-threaded application, model checking is computationally expensive and limited in its applicability [11].

The use of static code analyzers helps maintain code quality. Their integration in the build process is recommended to help identify potential issues earlier on in the development process.

Berkeley Lazy Abstraction Verification Tool (BLAST) is a software model checker for C programs. It is used to check that software satisfies the behavioral properties of its interface. It constructs an abstract model, which is model checked for safety properties. Given a temporal safety property for a C program, BLAST attempts to statically prove that it either satisfies the safety property or demonstrates an execution path that is in violation of the property. It uses lazy predicate abstraction and interpolation-based predicate discovery to construct, explore, and refine abstractions of the program state space. BLAST is platform independent and has been tested on Intel x86/Linux and Intel x86/Microsoft Windows with Cygwin. BLAST was released under the Modified BSD license [12].

Debuggers

Enhanced complexity of multi-threaded applications results from a number of factors, such as non-deterministic thread scheduling and preemption, and dependencies between control flow and data [11]. Non-deterministic execution of multiple instruction streams from runtime thread scheduling and context switching generally stems from the operating system's scheduler. The use of debuggers themselves may mask issues caused by thread interactions, such as deadlocks and race conditions. Factors such as thread priority, processor affinity, thread execution state, and starvation time can affect the resources and execution of threads.

A number of approaches are available for debugging concurrent systems, including traditional debugging, and event-based debugging. *Traditional debugging*, or breakpoint debugging, has been applied to parallel programs, where one sequential debugger per parallel process is used. These debuggers can provide only limited information when several processes interact. *Event-based* or *monitoring debuggers* provide some replay functionality for multi-threaded applications, but can result in high overhead. Debuggers may display control flow, using several approaches, such as textual presentation of data, time process diagrams, or animation of program execution [13].

The use of a threading API may impact the selection of a debugger. Using OpenMP, for example, requires the use of an OpenMP-aware debugger, which can access information such as constructs and types of OpenMP variables (private, shared, thread private) after threaded code generation.

Dynamic Binary Instrumentation (DBI)

Dynamic binary instrumentation analyzes the runtime behavior of a binary application by injecting instrumentation code which executes as part of the application instruction stream. It is used to gain insight into application behavior during execution. As opposed to static binary analysis, which exhaustively exercises all code paths, DBI explores only executed code paths. DBIs may be classified as either lightweight or heavyweight. A *lightweight DBI* uses architecture-specific instruction stream and state, while a *heavyweight*

DBI utilizes an abstraction of the instruction stream and state. Lightweight DBIs are not as portable across architectures as heavyweight DBIs. Valgrind [22], which is discussed later, is an example of a heavyweight DBI, and Pin [24], also discussed in this article, is an example of a lightweight DBI.

Profiling and Performance Analysis

Profilers are useful for both single- and multi-threaded applications. They facilitate optimization of program decomposition and efficient utilization of system resources by inspecting the behavior of a running program and helping to detect and prevent issues that can impact performance and execution. Issues encountered in multi-threaded applications include:

- Large number of threads, leading to increased overhead from thread startup and termination [1].
- Overhead from concurrent threads exceeding the number of hardware resources available [1].
- Contention for cache usage resulting from the large number of concurrent threads attempting to use the cache [1].
- Contention for memory use among threads for their respective stack and private data structure use [1].
- Thread convoying, whereby multiple software threads wait to acquire a lock [1].
- Data races occurring when two concurrent threads perform conflicting accesses and no explicit mechanism is implemented to prevent accesses from being simultaneous [14].
- Locking hierarchies causing deadlocks, which result in all threads being blocked and each thread waiting on an action by another thread [11].
- Livelocks (similar to deadlocks except that the processes/threads involved constantly change with respect to one another, with neither one being able to progress) can occur with some algorithms, where all processes/threads detect and try to recover from a deadlock, repeatedly triggering the deadlock detection algorithm [1].

Approaches to profiling can be identified as either *active* or *passive*. Compiler-based probe insertion is an example of active profiling, where execution behavior is recorded using callbacks to the trace collection engine. In passive profiling, control flow and execution state are inspected using external entities, such as a probe or a modified runtime environment. Passive profiling may require specialized tracing hardware and, in general, does not require modification of the measured system [11].

Data may be gathered by a profiler using a number of methods. *Event-based profilers* utilize sampling based on the occurrence of processor events. *Statistical profilers* use sampling to look into the program counter at regular intervals, using operating system interrupts. *Instrumenting profilers* insert additional instructions into the application to collect information. On some platforms, instrumentation may be supported in hardware using a machine instruction. *Simulator* or *hypervisor-based data collection* selectively collects data by running the application under an instruction set simulator or hypervisor.

Profilers may also be classified based on their output. *Flat profilers* show average call times, with no associated callee or context information. *Call-graph profilers* show call times, function frequencies, and call chains.

Profilers can provide behavioral data only for control paths that are actually executed. Execution of all relevant paths requires multiple runs of the application, with good code coverage. Code coverage can be improved using care-

fully selected input data and artificial fault injection. Fine-grained behavioral data from a running system can be coupled with offline analysis.

Profilers may not be portable across architectures, as they may require special hardware support. Others may focus only on user-space applications. A profiler may be designed to focus on analyzing the utilization of one or more system resources, such as call stack sampling, thread profiling, cache profiling, memory profiling, and heap profiling. Profilers can include aspects of absolute (wall-clock) time in their analysis [11].

OProfile is a profiling and performance monitoring tool for Linux on a number of architectures, including x86, AMD Athlon, AMD64, and ARM. It provides system-wide profiling, with a typical overhead of 1% to 8%, and includes a number of utilities. It consists of a kernel driver and a daemon for collecting sample data. OProfile uses CPU hardware performance counters for system-wide profiling, which includes hardware and software interrupt handlers, kernel and kernel modules, shared libraries, and applications [15].

DTrace is a dynamic tracing framework created by Sun Microsystems. It is now available for a number of operating systems, including Linux. It can be used to get an overview of the running system and is used for tuning and troubleshooting kernel and application issues on production systems, in real time. It allows dynamic modification of the OS kernel and user processes to record additional data from locations of interest using "probes." A probe is a location or activity with which DTrace can bind a request to perform a set of actions: for example, the recording of a stack trace. The source code for this tool has been released under the Common Development and Distribution License (CDDL) [16].

GNU Linux Trace Toolkit next generation (LTTng) is a static and dynamic tracer that supports C and C++ on Linux (and any language that can call C). It is supported on x86, PowerPC 32/64, ARMv7, OMAP3, sparc64, and s390. LTTng is available as a kernel patch, along with a tool chain (lttcontrol), which looks at process blocking, context switches, and execution time. It can be used for performance analysis on parallel and real-time systems. LTTV is a viewing and analysis program designed to handle huge traces. Tracers record large amounts of events in a system, generally at a much lower level than logging, and are generally designed to handle large amounts of data [17].

CodeAnalyst by AMD is a source code profiler for x86-based platforms with AMD microprocessors that is available for Linux and Windows environments. It has been built on top of the OProfile Linux tool for data collection, and provides graphical and command line interfaces for code profiling, including time-based and event-based profiling, thread analysis, and pipeline simulation [18].

Data Visualization

Visualization of profile data facilitates the comprehensibility of data and enhances its usability. A number of tools provide a standard interface for visualization of different types. *Gnuplot* is a portable, command-line–driven, interactive data and function plotting utility. It is copyrighted but can be freely distributed [19]. *Graphviz* is open source graph visualization software, which can be used to represent structural information as diagrams of abstract graphs and networks [20].

Dynamic Program Analysis

Dynamic program analysis is done by executing programs built either on actual hardware or on a virtual processor. Dynamic analysis checks program properties at runtime, and it generally identifies the problem source much faster than extensive stress testing does. Issues can be detected much more precisely, using code instrumentation and analysis of memory operations. These tools are generally easy to automate, with a low rate of false positives. For dynamic testing to be effective the test input has to be selected to exercise proper code coverage.

Valgrind is an instrumentation framework for building dynamic analysis tools. It is available for x86/Linux, AMD64/Linux, PPC32/Linux, and PPC64/Linux. Work on versions of Valgrind for x86/Mac OS X and AMD64/Mac OS X is currently underway. The Valgrind framework is divided into three main areas: core and guest maintenance (coregrind), translation and instrumentation (LibVex), and user instrumentation libraries [21]. Valgrind tools are used for detecting memory management and threading issues, and for application profiling [22]. Helgrind, Memcheck, Cachegrind, and Massif are some of the tools included in Valgrind's tool suite:

Helgrind is a thread debugger to detect data races in multi-threaded applications. It detects memory locations accessed by multiple Pthreads that are lacking consistent synchronization.

Memcheck is used to detect memory management–related issues for C and C++.

Cachegrind provides cache profiling and simulation of L1, D1, and L2 caches. **Callgrind** extends Cachegrind to provide visualization information about callgraphs.

Massif performs detailed heap profiling by taking regular snapshots of a program's heap, to help identify parts of the program contributing to most memory allocations.

DynInst allows dynamic insertion of code in a running program. It uses dynamic instrumentation to allow modification of programs during execution, without re-compilation, re-linking, and re-execution. DynInst was released by the Paradyn Parallel Tools Project and has been used by applications such as performance measurement tools, correctness debuggers, execution drive simulations, and computational steering. The recent release of DynInst supports PowerPC (AIX), SPARC (Solaris), x86 (Linux), x86 (Windows), and ia64 (Linux) [23].

Pin from Intel is a framework for building program analysis tools using dynamic instrumentation. It is an example of dynamic compilation targeting a VM which uses the same ISA as the underlying host [11]. It is an open source tool and does runtime binary instrumentation of Linux applications, whereby arbitrary C/C++ code can be injected at arbitrary places in the executable. Pin APIs allow context information, such as register contents, to be passed to the injected code as parameters. Any registers overwritten by the injected code are restored by Pin. It also relocates registers, in-lines instrumentation, and caches previously modified code to improve performance. The Pin architecture consists of a virtual machine (VM), a code cache, and an instrumentation API, which can be invoked by custom plugin utilities called Pintools. The VM consists of a Just-in-Time (JIT) compiler, an emulation unit, and a dispatcher. Instructions, such as system calls, which cannot be executed directly are intercepted by the emulator. The dispatcher checks

for the next code region in the code cache. If it is not present in the code cache, it is generated by the JIT compiler [24], [25], [26].

Active Testing

Active testing consists of two phases. Static and dynamic code analyzers are first used to identify concurrency-related issues, such as atomicity violations, data races, and deadlock. This information is then provided as input to the scheduler to minimize false positives from the concurrency issues identified during the static and dynamic code analysis. The tool CalFuzzer uses this approach.

CalFuzzer provides an extensible active testing framework for implementing predictive dynamic analysis to identify program statements with potential concurrency issues, and it allows implementation of custom schedulers, called *active checkers*, for active testing of concurrency-related issues [27].

System-Wide Performance Data Collection

In addition to problem partitioning and load balancing, programmers need access to systemwide resource usage data, as well as the ability to relate it to application performance. Availability of standardized APIs can facilitate access to such low-level system data by profilers and performance analyzers. PAPI is one such attempt at an API for accessing hardware performance counters.

The Performance Application Programming Interface (PAPI) project at the University of Tennessee defines an API for accessing hardware performance counters, which exist as a small set of registers for counting events. Monitoring the processor-performance counters enables correlation between application code and its mapping to the underlying hardware architecture and is used in performance analysis, modeling, and tuning. Tools that use PAPI include PerlSuite, HPCToolkit, and VProf. PAPI is available for a number of environments and platforms, including Linux, on SiCortex, Cell, AMD Athlon/Opteron, Intel Pentium, Itanium, Core 2, and, for Solaris, UltraSparc [28].

Conclusion

The increased complexity of multi-threaded parallel programming on multicore platforms requires more visibility into program behavior and necessitates the use of tools that can support programmers in migrating existing sequential applications to multicore platforms. This article presents a survey of different categories of tools, their characteristics, and the workflow between them. Most of the tools discussed are open source, or built on top of open source tools, for C and C++.

ACKNOWLEDGMENTS

I would like to thank Sungdo Moon from Samsung and *;login:* editor Rik Farrow for their review of the article and valuable suggestions.

REFERENCES

[1] Shameem Akhter, Jason Roberts, *Multi-Core Programming: Increasing Performance through Software Multithreading* (Intel Press, 2006).

[2] H. Wen, S. Sbaraglia, S. Saleem, I. Chung, G. Cong, D. Klepacki, "A Productivity Centered Tools Framework for Application Performance Tuning," *Proceedings of the Fourth International Conference on the Quantitative Evaluation of Systems*, Sept. 2007.

[3] "OpenMP API Specification for Parallel Programming": http://openmp.org/wp/.

[4] "GNU OpenMP (GOMP) Project": http://gcc.gnu.org/projects/gomp/.

[5] Lawrence Livermore National Lab, "POSIX Threads Programming": https://computing.llnl.gov/tutorials/pthreads/.

[6] Ralf S. Engelschall, "GNU Portable Threads," June 2006: http://www.gnu.org/software/pth/.

[7] Intel, "Threading Building Blocks 2.2 for Open Source": http://www.threadingbuildingblocks.org/.

[8] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard, version 2.1," June 2008: http://www.mpi-forum.org/docs/mpi21-report.pdf.

[9] Message Passing Interface (MPI) standard Web site: http://www.mcs.anl.gov/research/projects/mpi/.

[10] Khronos Group, "Open-CL, the Open Standard for Parallel Programming of Heterogeneous Systems": http://www.khronos.org/opencl/.

[11] Daniel G. Waddington, Nilabja Roy, Douglas C. Schmidt, "Dynamic Analysis and Profiling of Multi-Threaded Systems," 2007: http://www.cs.wustl.edu/~schmidt/PDF/DSIS_Chapter_Waddington.pdf.

[12] Thomas A. Henzinger, Dirk Beyer, Rupak Majumdar, Ranjit Jhala, "BLAST: Berkeley Lazy Abstraction Software Verification Tool": http://mtc.epfl.ch/software-tools/blast/.

[13] Yusen Li, Feng Wang, Gang Wang, Xiaoguang Liu, Jing Liu, "MKtrace: An Innovative Debugging Tool for Multi-Threaded Programs on Multiprocessor Systems," *Proceedings of the 14th Asia Pacific Software Engineering Conference*, 2007.

[14] Liqiang Wang, Scott D. Stoller, "Runtime Analysis of Atomicity for Multithreaded Programs," *Proceedings of the IEEE Transactions on Software Engineering*, Feb. 2006.

[15] OProfile Web site: http://oprofile.sourceforge.net/about/.

[16] Sun Microsystems, "Solaris Dynamic Tracing Guide," Dec. 2007: http://wikis.sun.com/display/DTrace/Documentation.

[17] GNU Linux Trace Toolkit next generation (LTTng) Project Web site: http://ltt.polymtl.ca/.

[18] AMD, "CodeAnalyst Performance Analyzer": http://developer.amd.com/CPU/CODEANALYST/Pages/default.aspx.

[19] Gnuplot Web site: http://www.gnuplot.info/.

[20] Graphviz graph visualization software Web site: http://www.graphviz.org/.

[21] Daniel Robson, Peter Strazdins, "Parallelization of the Valgrind Dynamic Binary Instrumentation Framework," *Proceedings of the International Symposium on Parallel and Distributed Processing with Applications*, 2008.

[22] Valgrind Web site: http://valgrind.org/.

[23] DynInst by the Paradyn Parallel Tools Project: http://www.dyninst.org/.

[24] Pin Web site: http://www.pintool.org/.

[25] "Pin: A Framework for Building Program Analysis Tools using Dynamic Instrumentation": http://www.cc.gatech.edu/~ntclark/8803f08/notes/pin.pdf.

[26] Steven Wallace, Kim Hazelwood, "SuperPin: Parallelizing Dynamic Instrumentation for Real-Time Performance," *Proceedings of the International Symposium on Code Generation and Optimization*, 2007.

[27] Pallavi Joshi, Mayur Naik, Chang-Seo Park, Koushik Sen, "CalFuzzer: An Extensible Active Testing Framework for Concurrent Programs," *Proceedings of the 21st International Conference on Computer Aided Verification*, 2009: http://berkeley.intel-research.net/mnaik/pubs/cav09/paper.pdf.

[28] Performance Application Programming Interface (PAPI) Web site: http://icl.cs.utk.edu/papi/.