Living with the Law: Can Automation Give Us Moore for Less?

Celina Gibbs[1], Jennifer Baldwin[1], Nieraj Singh[2],
Maja D'Hondt[3] and Yvonne Coady[1]
[1]University of Victoria, [2]IBM Canada, [3]IMEC

Abstract— **Multi-core programming presents developers with a dramatic paradigm shift. Whereas sequential programming largely allowed the decoupling of source from underlying architecture, it is now impossible to develop new patterns and abstractions in isolation from issues of modern hardware utilization. Synchronization and coordination are now manifested at all levels of the software stack, and developers currently lack the essential tools to even partially automate reasoning techniques and system configuration management.**

As a first stage to addressing this problem, this paper proposes a framework for a tool suite designed to partially automate the acquisition and management of static system visualization in a feedback loop with dynamic execution properties. This model enables developers to find a best fit system configuration, potentially reconciling resource contention and utilization tensions that are critical to multi-core platforms. The application of a prototype of this suite, Déjà View, demonstrates how tool support can aid reasoning about causally related sets of changes across system artifacts.

Keywords- Visualization; multi-core; configuration.

I. INTRODUCTION

Until recently, Moore's law enabled developers to reap benefits of faster hardware without having to revamp their work [17]. Running these same applications on multi-core architectures exposes subtle new relationships between software and hardware, and these interactions are now painfully exposed at all levels of the software stack. Parallelization of an application means developers must explicitly consider sets of causally related changes across multiple artifacts: source, build/runtime configuration, profiling infrastructure, input data, and of course dynamic characteristics as the system is executing.

Transforming applications from sequential to parallelized versions requires changes that are rarely well contained. Instead, they causally cascade through multiple artifacts—a change in hardware causes new development of source, causing the corresponding build to change. Subsequent changes in dynamic characteristics must be tracked, which may call for a change in profiling infrastructure, which may impact memory footprint with an unwanted observer effect [10], and so on.

Could a general approach to automation allow developers to more effectively customize their applications for parallel architectures with less manual labour than is now currently required? We believe a first step to answering this question is directly associated with the degree to which ASE can help developers reason about causal relationships between changes to artifacts.

This paper proposes a general framework to automate the acquisition and management of causally related changes. After a brief survey of related work in Section II, Section III proposes the framework for a tool suite designed to separate familiar content from incremental and causally linked changes that need to be highlighted. The framework leverages automation to aid viewing, navigating, integrating, and cataloging fine grained program transformations in a way that can be easily combined with customized tools for specific sets of changes. A prototype of the tool suite, *Déjà View*, along with its application to the NASA Parallel Benchmarks [5, 6] is provided in Section IV.

II. PROBLEM OVERVIEW AND RELATED WORK

Current trends suggest that future hardware platforms will house thousands of cores, as those that contain millions are actually already actively undergoing testing, such as IBM's project KittyHawk [3]. Inevitably, specific patterns for best practices for resource utilization will emerge over time, as programmers get more experience with the precise tradeoffs involved. Once these practices are established, automation will be key to supporting stable and efficient implementations in mainstream development.

In practice, the popularity of Message Passing Interface (MPI) [16], an API that includes both point-to-point and collective (global) operations, has provided an important language-independent approach for programming parallel computers for many years. More recently, MapReduce [7], a robust software framework designed to support parallel computations over massive data sets, has achieved high reliability and also offers potential mainstream appeal.

Open Multi-Processing (OpenMP) [15], an API that supports multi-platform shared memory in a language-independent way, consists of a set of compiler directives, library routines, and environment variables for multi-core applications. This approach lends itself well to the transformation of existing sequential applications to their parallelized counterparts in that it is particularly flexible and noninvasive in terms of source code modifications required.

978-1-4244-2188-6/08/$25.00 ©2008 IEEE

More specifically in terms of true concurrency on a multi-core system, there are several contenders. The debate between events, threads, and hybrid approaches has perhaps reached a crescendo with a hybrid approach suggested by Staged Event Driven Architectures (SEDA) [18]. In this approach, event-driven software applications are decomposed into a set of stages connected by queues, avoiding overheads of thread-based concurrency models, and decoupling concurrency from application logic. Another promising approach includes both Software Transactional Memory (STM) [8] coupled with corresponding Hardware Transactional Memory (HTM) [9]. Transactional memory allows a group of instructions to execute atomically, analogous to database transactions. Finally, commercial solutions are also starting to emerge, including RapidMind [14], a framework for executing data-parallel computations in C++ on multi-core processors.

The daunting task of efficient programming for highly parallel systems is currently receiving much attention from several perspectives within computer science [4]. This offers an opportunity for researchers to rethink programming models, system software, and hardware architectures from the ground up. However, the question of which part of the system to parallelize—the application, the infrastructure, or both—remains an open issue. In the meantime, developers need tool support to bridge this multifaceted transition from serial to parallel systems.

III. PROPOSED FRAMEWORK

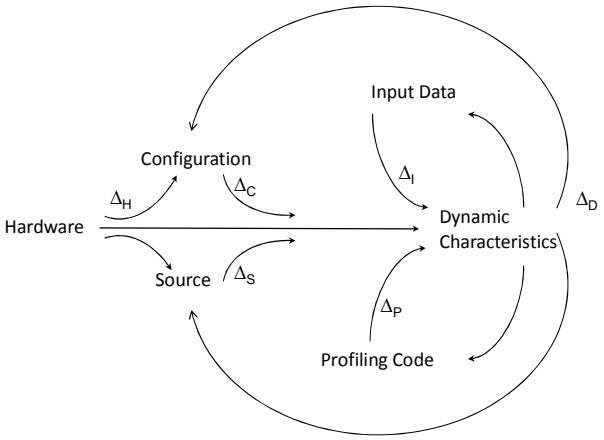


Figure 1. Key possible causal relationships between changes in artifacts.

A lofty long-term goal from an ASE perspective might be to have an expert system operating, perhaps within a virtual machine, to guide the transformation from sequential to the most efficient parallel implementation on a given architecture. In the short-term however, we would like to make it easier to explore subtle feedback loops and causal relationships between changing artifacts in this intense and precarious transformation process. To this end, it is useful to identify and enumerate a core set of related changes as interaction between specific artifacts, as highlighted in Figure 1. Though this list is not exhaustive, we start by considering the following sets of representative changes imposed by one artifact onto other artifacts, each denoted by a unique Δ_i:

Δ_H A change to hardware that may cause a change to configuration, source, and/or one or more dynamic characteristics.

To somewhat simplify the problem space, we then consider the following in terms of their causal impact on one or more dynamic characteristics (such as performance, memory usage, resource utilization and contention):

Δ_C A change in build or runtime options.

Δ_S A change to source code.

Δ_I A change in test input.

Δ_P A change in profiling infrastructure.

Finally, completing the feedback loop:

Δ_D A change in a dynamic characteristic that in turn may precipitate many more related Δ_C, Δ_S, Δ_I, and Δ_P.

The development of a general purpose framework that takes into account each feedback loop between sets of changes, $\Delta_i \Leftrightarrow \Delta_j$, will most likely require its own application or system specific tool. That is, the variability in everything from build systems to profiling strategies suggest that customized tools should be integrated into a framework in order to facilitate a common strategy for viewing and cataloging changes to sets of artifacts. Figure 2 highlights the common component for change relationships, coupled with the extension points that need to be system specific. For example, though even simple profiling in C based systems may range from conditional compilation to more modern techniques [13], viewing their respective transitions may share a common representation.

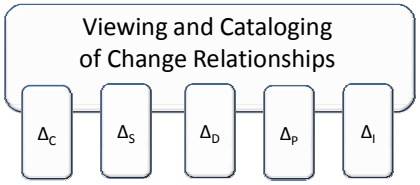


Figure 2. Structure of proposed framework.

Though we envision policies to automate cataloguing support, perhaps triggered by time or number of changes, we currently rely on manual practices to identify a semantically significant point in development, and a standard version control system for storage. Figure 3 highlights the ways in which the relative differences ($\Delta_{i,j}$) between two or more snapshots (SS_i, SS_j) can be acquired. These transitions can viewed in the general viewer, and further investigated with application specific extensions to the framework.

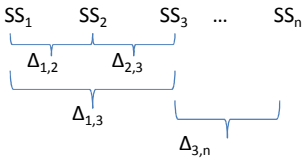
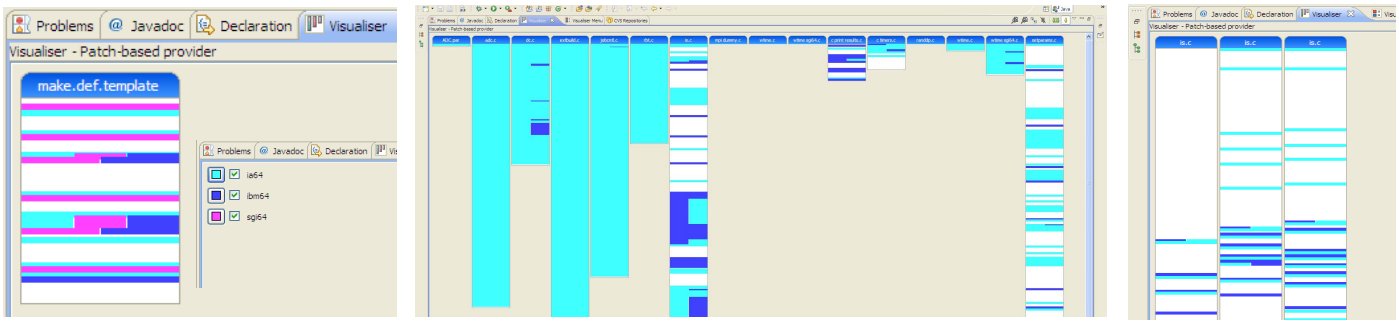


Figure 3. Snapshots (SS_i) of all system artifacts are cataloged, promoting further reasoning support for changes ($\Delta_{i,j}$) between any two (SS_i, SS_j).



(a) Δ_C (b) Δ_S (c) Δ_P

Figure 4. (a) Déjà View of the differences between ia64, ibm64, sgi64 configuration files relative to the template provided, (b) course-grained view of OpenMP (dark) and MPI (light) parallelization code relative to a serial implementation, and (c) three levels of profiling relative to a uninstrumented file.

IV. Case Study: Déjà View Applied

The initial prototype of this framework, *Déjà View*, partially leverages ASE techniques to make specific causal relationships easier to reason about, such as feedback loops associating changes between dynamic characteristics and source ($\Delta_D \Leftrightarrow \Delta_S$), configuration ($\Delta_D \Leftrightarrow \Delta_C$), profiling ($\Delta_D \Leftrightarrow \Delta_P$), and input data ($\Delta_D \Leftrightarrow \Delta_I$). This section considers the application this prototype to the NASA Parallel Benchmarks [5, 6]. The benchmarks consist of 8 programs designed to evaluate performance on parallel architectures. Collectively, they represent essential computation and data movement characteristics that typically create resource contention in parallel systems. Results of running the benchmarks on a wide variety of platforms over several years are reported for over 25 different platforms [6]. Each benchmark has three coarse grained configuration options designed to capture various aspects of sequential and parallel execution: *Serial*, *OpenMP*, and *MPI*. Both the OpenMP and MPI benchmarks can be reasoned about as incremental transitions relative to the serial implementation.

The general framework component of Déjà View leverages the an Eclipsed based viewer originally designed for AspectJ [1, 12]. The viewer helps to uncover the differences within any given artifact associated with the characteristically tight feedback loops within a parallelized implementation. The goal is to facilitate the understanding of relationships between changes, and to support reasoning about alternative configurations and implementations.

A. Δ_C: Configuration

Figure 4(a) provides a visualization of the three 64 bit architecture-specific configuration files for OpenMP benchmarks with respect to the template configuration file provided by the benchmark suite. This view indicates, at a high-level, the points where the three configuration files differ. Two of the three configuration files are shown to differ from the template file in roughly the same places. Further support for navigation and visualization of the details of these fine-grained differences are provided in a standard Eclipse editor view, where gutter annotations indicate the places where a specific configuration differs from the template. This navigation facilitates a developer's ability to experiment with different subtle combinations of flag settings. The importance of this type of fine-grained experimentation is indicated in the OpenMP ReadMe file:

```
For some buggy OpenMP compilers, you may
have to play with the optimization flag, for
instance, use "-O2" over "-O3".
```

B. Δ_S: Source

With the development of the two variations of parallel benchmarks relative to the serial implementation, it is beneficial to view their implementation in terms what was parallelized. Figure 4(b) provides a high-level perspective of where the parallel implementation is introduced by both OpenMP and MPI with respect to the serial implementation. This view shows the larger impact of the MPI implementation, as well as the locations of overlap between the two implementations. Again, from this high-level perspective there is direct navigation to the source for the associated implementation details.

C. Δ_P: Profiling

Current strategies for profiling include a mix of print statements, a dummy file added to the current working directory to turn on timing, and C pre-processor directive conditionally compiled statements:

```
#ifdef  TIMING_ENABLED
        timer_start( 3 );
#endif
```

Emphasis is placed on profiling to provide catalogue information for tracing relevant configuration options:

```
This is a gross hack to allow the benchmarks
to print out how they were compiled.
```

Figure 4(c) provides the visualization of a naïve implementation of profiling which builds on existing profiling support provided by the `TIMING_ENABLED` flag. The implementation introduces two additional levels of profiling: `PROFILE_ENTER` and `PROFILE_LEAVE`, where the first traces entry to a function, and the second provides a combination trace of function entry and exit. The visualizer provides a high-

level view of these profiling levels (each in a different colour) as it applies to the base system, as well as the ability to configure the profiling through the editor view.

D. Δ_I and Δ_D: Input and Dynamic Properties

Input files for the NASA Parallel Benchmarks are small—on the order of few lines of code. Though these input files are small, the results vary greatly with respect to dynamic characteristics of benchmarks, and even configuration options. For example, different size inputs require different environment settings. The `README` for the OpenMP configuration specifies that an increase in input size may precipitate the need for a subsequently larger stack size and gives the following suggestions for two specific architectures:

```
SGI Origin3000: setenv MP_SLAVE_STACKSIZE 50000000
SGI Altix Intel compiler: setenv KMP_STACKSIZE 50m
```

Additionally, in terms of dynamic characteristics, there is a critical link between input size and performance. As previously mentioned, the benchmarks consist of 8 programs run on a wide variety of platforms, with hundreds of sets of test results collected over time. These results are typically categorized based on overall comparisons between *Class A* versus *Class B* results, which differ in the size of their principal arrays, shown in Figure 5. As these causal relationships are fundamentally cross-artifact, tool support akin to a Mylyn [11] approach will be most effective in further linking this context.

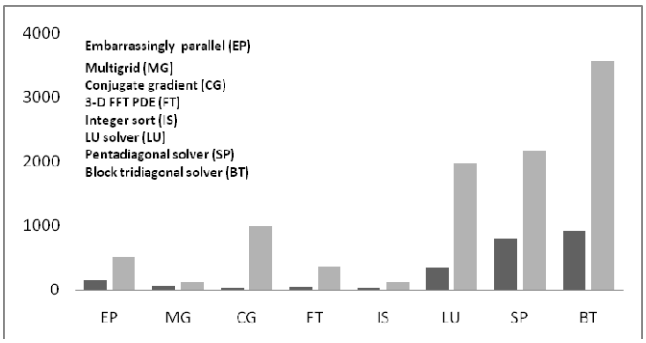


Figure 5. Comparison of Class A (dark) and Class B (light) execution results in seconds on a single processor of a Cray Y-MP.

V. Conclusions and Future Work

ASE will be essential to making the currently labour intensive process of parallelization more unified, consistent and systematic. As a first step we have been able to unify perspectives on changes to multiple artifacts that are typically casually linked. By allowing developers to more easily catalogue a comprehensive system snapshot, and compare any two snapshots over time, we hope to establish a more global perspective on causal relationships and develop further strategies where ASE can be applied. The question of how well this approach will scale remains.

Related studies on co-evolution of build structure and the Linux kernel [2] hold promise for more ASE techniques for integrating this evolution as further application specific tool extensions. Future work also includes incorporating support for querying change sets in snapshots, as well as incorporating dynamic analysis tools into what are now only static reports of dynamic characteristics.

References

[1] AJDT: AspectJ Development Tools, The Visualizer, December 2005. http://www.eclipse.org/ajdt/visualiser/.

[2] B. Adams, K. De Schutter, H. Tromp, and W. D. Meuter. Design recovery and maintenance of build systems. In Proc. of the 23rd International Conference on Software Maintenance (ICSM), October 2007.

[3] J. Appavoo, V. Uhlig, and A. Waterland. Project Kittyhawk: Building a Global-Scale Computer. Published in in ACM SIGOPS Operating System Review, January 2008.

[4] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, K. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. No. UCB/EECS-2006-183, December 2006.

[5] D. Bailey, E. Barscz, J. Barton, D. Browning. R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, S. Weeratunga. The NAS Parallel Benchmarks. NAS Technical Report, RNR-94-007, NASA Ames Research Center, Moffett Field, CA, 1994.

[6] D. Bailey, T. Harris, W. Saphir, R. Van der Wijngaart, A. Woo, M. Yarrow. The NAS Parallel Benchmarks 2.0. NAS Technical Report NAS-95-020, NASA Ames Research Center, Moffett Field, CA, 1995.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In Operating Systems Design and Implementation (OSDI), pages 137-150, 2004.

[8] T. Harris and K. Fraser. Language Support for Lightweight Transactions. Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA) pp.388–402. 2003.

[9] M. Herlihy, and J. Moss. Transactional memory: architectural support for lock-free data structures. SIGARCH Comput. Archit. News 21, 2 pp.289-300. May 1993.

[10] W. Heisenberg, "Über den anschaulichen Inhalt der quantentheoretischen Kinematik und Mechanik", Zeitschrift für Physik, 43 1927, pp. 172-198. English translation: J. A. Wheeler and H. Zurek, Quantum Theory and Measurement Princeton Univ. Press, pp. 62-84. 1983.

[11] M. Kersten and G. C. Murphy. Using Task Context to Improve Programmer Productivity. In Proceedings of the Foundations of Software Engineering (FSE), 2006.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. Griswold. An Overview of AspectJ, Proceedings of 15th European Conference on Object-Oriented Programming (ECOOP), 2001.

[13] B. McCloskey and E. Brewer. ASTEC: a new approach to refactoring C. SIGSOFT Softw. Eng. Notes, 30(5):21–30, 2005.

[14] M. Monteyne. RapidMind Multi-Core Development Platform. RapidMind White Paper, www.rapidmind.net/product.php, 2008.

[15] OpenMP Architecture Review Board. OpenMP application program interface. Technical Report 2.5, OpenMP Architecture Review Board, May 2005. http://www.openmp.org/specs.

[16] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra. MPI: The Complete Reference. MIT Press, 1996.

[17] H. Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software, Dr. Dobb's Journal, 30(3), March 2005.

[18] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In Proceedings of the Eighteenth Symposium on Operating Systems Principles (SOSP-18), 2001.