

Why Should I Rewrite My Software When Dynamic Compilation Can Be Good Enough?

Nathan Clark
College of Computing
Georgia Institute of Technology
ntclark@cc.gatech.edu

1. FUTURE MANY-CORE SOFTWARE

Trends in device manufacturing and processor development have made it abundantly clear that heterogeneous many-core processors are going to be the dominant computational platform for executing tomorrow's applications. Projections are that these platforms will soon contain thousands of cores, many of which will be highly customized to provide as much computational capacity as possible within a fixed power budget.

Clearly there are many challenges associated with engineering such complex systems, but software development is the most critical one. Looking at the problem from an economic perspective, people purchase computers because of the valuable services they provide. Almost universally, these services are developed in software. New services are constantly being enabled by more powerful hardware, which drives consumers to purchase that new hardware, and fuels the computing industry in a positive feedback loop [11]. Current developers are used to creating sequential applications, and the move to parallel applications is a daunting challenge. In order to maintain the pace of software innovation and growth of the computing industry as a whole, we must provide simple ways for software developers to leverage the computational power of heterogeneous many-core architectures.

Developing software for many-core systems is certainly not a new problem. The supercomputing community has devised several different strategies to tackle the problem, as they have been creating applications that use hundreds or thousands of cores for decades. Unfortunately many of these techniques are ineffective in the hands of the average developer, and thus there have been several recent proposals for how mainstream developers should create parallel software. I will briefly discuss the strengths and weaknesses of the classes of proposed solutions.

New Languages: Much of the difficulty in writing parallel software stems from the fact that programmers typically specify the computation to be executed in relatively old, sequential languages (e.g., C), which were not designed with parallelism in mind. Specifying the desired computation in new languages that better express dependencies (e.g., by eliminating pointers and using strict typing) or by explicitly describing parallelism that exists in the code enables compilers to do a very good job parallelizing applications automatically. There are several drawbacks with proposing new languages, however. First, developers using a new language often must think in parallel to explicitly write parallel code, and they must also debug the parallel applications they write. The move from understanding a single thread of execution to many threads of execution is a tremendous cognitive leap, and significantly raises the difficulty of software engineering. Second, in most markets legacy code is still critically important, and the cost of rewriting legacy applications using new languages is tremendous. Lastly, the historical adoption rate of new programming languages is very poor. Despite the many benefits available to developers who use modern programming languages, convincing developers to use those languages has generally not been successful.

Libraries: The use of libraries is a popular method for parallel programming, particularly in the supercomputing community. In this approach, a group of skilled library writers identify common functionality used in many applications and manually design par-

allel software to effectively execute that functionality. When the underlying hardware evolves, a new library must be developed, but the software developers who use the library do not have to change their applications. As with new languages, software developers who utilize libraries must rewrite their legacy code, however using libraries moves the burden of parallel thinking and debugging to a small set of library writers. Another issue with libraries is that the entire set of functionality needs to be defined a priori; if developers need parallel functionality that is not implemented in the library, then they must implement it themselves. Library interfaces also hinder program optimization; for example, if an application had consecutive calls to $\sin(x)$ and $\cos(x)$, there is a significant amount of redundant work that could be eliminated if the compiler could see beyond the interface boundaries.

Advanced Compilation: Recent work [4, 14] has shown that automated compiler parallelization of sequential code is surprisingly effective for many applications. Clearly, automated parallelization of legacy codes falls short of what expert human developers are capable of, but this technology presents a low-cost path for many software developers to create parallel applications that perform "good enough" without sacrificing the sequential programming model. Another major benefit of this approach is that legacy applications can be parallelized simply through recompilation. Among the drawbacks of compiler parallelization is that increasingly the compiler's scope is being limited by dynamically loaded libraries and several advanced language features, such as virtual functions and reflection. If the compiler cannot see significant portions of the application statically, then its ability to parallelize that code greatly diminishes. A second issue is that, even though it is simpler than rewriting applications, recompilation of legacy code is still a significant cost that has prevented the adoption of many innovations in the past. Perhaps the most important drawback of compilation based parallelization is that the thread decomposition produced by the compiler is static and does not adjust based on the underlying architecture or the current state of the system. Clearly the best thread decomposition for an application can change depending on factors such as number of cores on chip, whether a heterogeneous processor (e.g., a GPU) exists in the system, or how overloaded a shared resource is.

New languages, libraries, and automated compiler parallelization have all been used successfully in various circumstances, but they all leave a lot to be desired. What software developers *really* want is the ability to use the sequential programming model they know and understand. They do not want to learn new programming languages and they want their legacy applications to "just work" when new hardware is developed.

My position is that *dynamic compilation can make this possible* in many situations. A well engineered dynamic compiler can perform the sophisticated analyses already demonstrated in parallelizing static compilers, but unlike static compilers, the dynamic compiler can adjust its task decomposition based on the underlying hardware and system state. Dynamic compilation also has no limitations on code visibility, and legacy binaries can be executed without recompilation. Expert human programmers will always be able to generate better code than the automated parallelization, however, providing a system that can parallelize legacy code and applications

developed using old, sequential programming models will go a long way toward helping sustain the rapid innovation we've come to expect in the software industry. The remainder of this paper will try to convince the reader that automatic parallelization through dynamic compilation is both feasible and a fruitful area of future research.

2. FEASIBILITY OF A PARALLELIZING DYNAMIC COMPILER

In order to estimate the feasibility of a parallelizing dynamic compiler, it is necessary to examine successful static parallelization techniques and evaluate their effectiveness in a dynamic context.

Work by Ryoo et al. [14] cited several analyses that were essential in automatically parallelizing media applications. The most critical analyses include interprocedural context-, heap-, and field-sensitive pointer analysis, and value constraint analysis (i.e., discovering certain variables only take on a small number of values during execution). Bridges et al. [4] add to that list user-inserted annotations for commutative functions and hardware support for speculative thread execution. Commutative functions are functions that have dependences between calls, but violating these dependences does not change program semantics. For example, hash table insertions can be executed in any order without changing the result of subsequent hash table lookups. These modern analyses and speculation support have overcome many of the granularity problems associated with traditional automated parallelism.

One potential concern is that many of these analyses are very computationally intense to perform statically (such as context sensitive pointer analysis), and may lead to dynamic compilation overheads that outweigh any potential performance improvements. Two factors mitigate this problem. First, analyses can be performed in parallel of executing the input code. Second, the vast majority of program execution happens in very few "hot spots". It is common for dynamic compilers to monitor execution frequency to identify hot spots, and perform progressively more optimization only when it is clear the code is important [15]. With this in mind, let's examine these analyses in the context of dynamic compilation.

Pointer Analysis: There is a significant amount of work in Java virtual machines discussing pointer analysis. For example, work by Hirzel et al. [10], presented a very fast dynamic algorithm for Andersen's style pointer analysis with field sensitivity and partial context sensitivity. While this is not quite as precise as the pointer analysis used in [14], this algorithm took less than 0.1 seconds to execute on average and provides a concrete example that sophisticated pointer analysis can be performed dynamically. Other work [8] noted several instances where performing pointer analysis on low level code, accessible at runtime, is actually more accurate than pointer analysis on high level code, where static compilers traditionally perform it.

Value Constraint Analysis: Determining that variables take on a small number of values during program execution is invaluable for removing both control and memory dependencies in applications. As with pointer analysis, there has been a significant amount of work in this area in the Java community; the primary application being dynamically removing spurious array bounds checks when the array index is provably within the array boundary. One paper [3] on this topic demonstrates that this analysis takes only 4 milliseconds, on average, per application. More extensive dynamic invariant analysis techniques [7] have been implemented in the software engineering community, and again prove this analysis is possible in a dynamic compilation environment.

Commutativity Analysis: Unlike the previous two analyses, I am unaware of any work on dynamically identifying commutative sections of code in applications. Several papers have explored identifying commutative sections of code in static compilation [13], and these static algorithms typically run in less than 1 second. Static

compilation algorithms almost always have a dynamic compilation analog, and dynamic compilation should be able to, at the very least, identify probable commutativity for use in automatic parallelization by adapting the static technique.

Speculation Support: Proving thread independence in any compilation system can be quite difficult, and so support for speculation of probable thread independence is necessary for high performance automatic parallelization. Beyond performance, speculation support also frees the dynamic compiler from having to analyze the entire application. The compiler merely has to detect likely program properties, parallelize assuming those properties hold, and correct or update the parallelization when incorrect. Speculation support has been commercially deployed using both software-based [1] and hardware-based [5] detection and recovery, and has been used in several research projects for speculative parallelization.

There is significant amount of previous research demonstrating many of the critical analyses needed for effective static compiler parallelization have been implemented in dynamic compilers. There is even work showing that a subset of these optimizations has provided useful automatic parallelization in the context of a dynamic compiler [12]. *Building a dynamic compiler to effectively and automatically parallelize binaries is possible for many domains.*

Beyond parallelization, there many other benefits to using dynamic compilation, such as dynamic thread specialization. This transformation retargets individual threads for heterogeneous processors. One excellent example of this technique is the PeakStream toolset, which can take an application written once and dynamically generate code for diverse platforms such as multi-core x86 systems, GPUs, or Cell processors. Stitt et al. [17] have also demonstrated a system that dynamically retargets threads for FPGAs. The ability to dynamically specialize code for heterogeneous processors is well within the realm of possibility.

There are also many instances in the literature of adaptive dynamic compilation systems that monitor the system and continually improve the code running on it [16]. In one example, Hazelwood and Brooks describe a dynamic compiler that monitors when programs cause supply voltage swings through fluctuating current draw, and applies program transformations to ameliorate the problem [9]. Applying adaptive principals to automatic parallelization will certainly be challenging, but gives automatic parallelization the potential to outperform manual parallelization by adjusting to the system at runtime.

Equally important, the applications predicted to be most important in the future are very amenable to automatic parallelization using dynamic compilation. Looking at media processing applications, RMS benchmarks [6], and Berkeley's "13 Dwarfs" [2], one can observe that many of these applications have easily identified parallelism. These benchmarks are also long running, giving the dynamic compiler ample time to identify patterns and perform the necessary optimizations.

The compiler field has advanced to the point where we understand many of the analyses needed to provide effective automatic parallelization for many classes of applications. There is certainly much research left to be done, but many of these analyses have already been implemented in dynamic compilation systems, and the ones that have not are ripe research targets. Threads discovered by an automatic parallelization system can leverage heterogeneous processing resources, and the can be dynamically adapted to improve overall system performance. A parallelizing dynamic compiler would enable software developers to hold onto the sequential programming model they understand, and maintain the pace of software innovation. I believe the pieces to make this system work are within reach. Let's try to put the them together and see what can be accomplished before rewriting all of our legacy code and completely abandoning the sequential programming model that has brought us so far.

3. REFERENCES

- [1] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. A survey of adaptive optimization in virtual machines. *IEEE Proceedings of*, 93(2):449–466, June 2005.
- [2] K. Asanovic et al. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, Dec. 2006.
- [3] R. Bodik, R. Gupta, and V. Sarkar. Abcd: eliminating array bounds checks on demand. In *Proc. of the SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 321–333, June 2000.
- [4] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August. Revisiting the sequential programming model for multi-core. In *Proc. of the 40th Annual International Symposium on Microarchitecture*, pages 69–84, 2007.
- [5] M. K. Chen and K. Olukotun. The Jrpm system for dynamically parallelizing Java programs. In *Proc. of the 30th Annual International Symposium on Computer Architecture*, pages 434–446, 2003.
- [6] P. Dubey. A platform 2015 workload model recognition, mining and synthesis moves computers to the era of tera, 2005. <ftp://download.intel.com/technology/computing/archinnov/platform2015/download/RMS.pdf>.
- [7] M. Ernst et al. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, Dec. 2007.
- [8] B. Guo et al. Selective runtime memory disambiguation in a dynamic binary translator. In *Proc. of the 15th International Conference on Compiler Construction*, pages 65–79, 2006.
- [9] K. Hazelwood and D. Brooks. Eliminating voltage emergencies via microarchitectural voltage control feedback and dynamic optimization. In *Proc. of the 2004 International Symposium on Low Power Electronics and Design*, pages 326–331, 2004.
- [10] M. Hirzel, D. von Dincklage, A. Diwan, and M. Hind. Fast online pointer analysis. *ACM Transactions on Programming Languages and Systems*, 29(2):11, Apr. 2007.
- [11] W. Hwu et al. Implicitly parallel programming models for thousand-core microprocessors. In *Proc. of the 44th Design Automation Conference*, pages 754–759, June 2007.
- [12] L. Rauchwerger. *Runtime Parallelization: A Framework for Parallel Computation*. PhD thesis, University of Illinois at Urbana-Champaign, 1995.
- [13] M. Rinard and P. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 19(6):1–47, Nov. 1997.
- [14] S. Ryoo, S. Ueng, C. Rodrigues, R. Kidd, M. Frank, and W. Hwu. Automatic discovery of coarse-grained parallelism in media applications. *Transactions on High Performance Embedded Architectures and Compilers*, 1(1):194–213, Jan. 2007.
- [15] J. E. Smith and R. Nair. *Virtual Machines*. Morgan Kaufmann Publishers, 2005.
- [16] M. Smith. Overcoming the challenges to feedback directed optimization. In *Proc. of the 2000 ACM Workshop on Dynamic and Adaptive Compilation and Optimization*, pages 1–11, 2000.
- [17] G. Stitt and F. Vahid. Thread warping: A framework for dynamic synthesis of thread accelerators. In *Proc. of the 2007 International Conference on Hardware/Software Co-design and System Synthesis*, pages 93–98, Sept. 2007.