POWER: Parallel Optimizations With Executable Rewriting

Nipun Arora, Jonathan Bell, Martha Kim, *Vishal K. Singh, Gail E. Kaiser
Computer Science Department **NEC-Labs America*
Columbia University *Princeton, NJ*
$\{$*nipun,jbell,martha,kaiser*$\}$*@cs.columbia.edu* *vishal@nec-labs.com*

Abstract

The hardware industry's rapid development of multi-core and many core hardware has outpaced the software industry's transition from sequential to parallel programs. Most applications are still sequential, and many cores on parallel machines remain unused. We propose a tool that uses data-dependence profiling and binary rewriting to parallelize executables without access to source code. Our technique uses Bernstein's conditions to identify independent sets of basic blocks that can be executed in parallel, introducing a level of granularity between fine-grained instruction level and coarse-grained task level parallelism. We analyze dynamically generated control and data dependence graphs to find independent sets of basic blocks which can be parallelized. We then propose to parallelize these candidates using binary rewriting techniques. Our technique aims to demonstrate the parallelism that remains in serial application by exposing concrete opportunities for parallelism.

1 Introduction

The proliferation of multi-core architectures over the last 10 years offers a significant performance incentive for parallel software. Unfortunately, such software is substantially more complicated than serial code to develop, debug and test, resulting in a large number of applications that cannot exploit today's abundant hardware resources. There has thus been significant research interest in auto-parallelization - techniques that automatically transform serial code into a parallel equivalent [2].

Auto-parallelization approaches can broadly be divided into two categories, each with its own drawbacks. *Static auto-parallelization* techniques are applied when the program is not running, simply by examining a program's code or a disassembled binary. These techniques add little to no overhead during execution, but suffer from the usual drawbacks of static alias analysis [5], a well known and unresolved problem within the compilers community. Thus, in practice, static analysis techniques are limited to structured DO-ACROSS and DO-ALL style parallelizations [2, 4, 15].

There have also been a few attempts at purely *dynamic auto-parallelization*. These approaches are typically rooted in Bernstein's conditions, which allow two statements to run in parallel if they do not produce a read-write or write-write data conflict [3]. Dynamic parallelization [7, 17] occurs while a program is running and can provide more opportunities to parallelize, but comes at a cost of run-time overhead.

In this paper we present *POWER: Parallel optimizations with executable rewriting*, a tool that transforms sequential binary executables into parallel ones, using a hybrid of static and dynamic analysis techniques. The POWER tool chain captures run-time profiles over test executions of the code and then analyzes them offline to determine potential parallelizations of basic blocks. POWER is designed to make some parallelization easily exploitable, without requiring additional development time to understand and parallelize the code.

POWER is designed with three principle goals::

Generality: The parallelized binary should run on all operating systems and architectures on which the original serial binary ran.

Transparency: In many situations, the application source code will not be available. POWER must not require source code or any changes in underlying execution environment like OS or hardware.

Performance: POWER must exploit more parallelisms than other tools, to give improved performance for the application.

We believe that POWER offers a significant improvement over current auto-parallelization techniques, as follows:

- Exploitation of basic block level parallelism using dynamic profiles
- Profile driven auto-parallelization using a hybrid technique (static/dynamic)
- A source-free, binary only approach to parallelism

One of our key innovations is to explore parallelism at

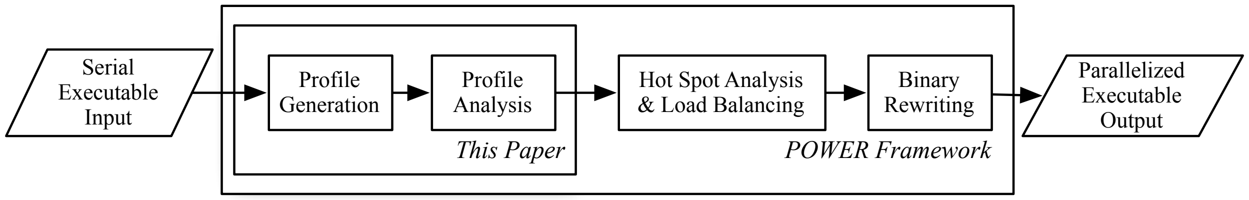

Figure 1: POWER Architecture Overview: POWER takes in an unmodified serial binary, profiles it, analyzes the static program structure in conjunction with the dynamic profile information, then rewrites the binary to expose basic block-level parallelism

the basic block level, based on profiles collected during program execution. The intuition behind parallelism at the basic block granularity is that most parallelism hot spots occur in loops, recursions, and similar structures. A basic block representation provides natural boundaries for detecting parallelism candidates, as loop/function recursions all start with a jump instruction to the head of a basic block. Basic block level parallelism, introduces a new level of granularity between instruction level and task-level parallelism.

During program execution, POWER collects both data and control flow profiles. By generating data dependency profiles, we are able to observe actual data-dependence as the program executes. Static parallelizers, however, are limited in their applicability because of pointer aliasing problems. Using these profiles simplifies aliasing concerns and provides us with additional candidates for parallelism.

Operating purely at the binary level allows us to present a generic solution that can be applied irrespective of the language/compiler of the target application. Our approach can optimize applications regardless of the availability of their source code, a potentially valuable feature when supporting legacy systems.

2 POWER: A profile-guided auto-parallelization system for executables

Our approach is a hybrid of static and profile guided approaches, and is presented in Figure 1. POWER utilizes information gathered by running the target program several times on representative inputs, generating a number of profiles. After generating profiles, we use a novel combination of the apriori algorithm [1] and Djikstra's shortest path algorithm to identify sets of basic blocks that can be parallelized. Next, we analyze potential parallel schedules based on hot spots and load balancing constraints. Finally, we propose to apply the parallelizations determined above to the binary and verify the correctness of the rewritten binary. Unlike classical compiler approaches, where transforms must to be correct under all conditions, our transforms are somewhat optimistic, and they fallback to the original code in case the actual execution doesn't match the profile. This is similar to optimistic optimization approaches where code segments are executed out of order on an otherwise idle processor and the results are flushed if a dependency arises later. This section gives a detailed view of the work flow of POWER. Later in sec. 3, we explain the working of the tool-chain using an example.

2.1 Serial Executable Input

POWER takes a sequential binary as input, which is then used for profile generation in the subsequent steps. The system requires no source code information or special compilers. This allows us to employ POWER to any sequential application, irrespective of original programming language or the compiler that produced it, including, importantly, legacy applications.

2.2 Profile Generation

The first phase of our approach generates a cumulative profile of control flow and data dependencies over several runs of the program. The goal of profiling is to get a realistic idea of the control flows in the application binary. By observing executions of the target application at the binary level, we create a disambiguated control and data dependence flow of the program. We instrument each basic block and variable access with the PIN Instrumentation Tool [16]. The PIN instrumentation tool framework provides an opportunity to dynamically insert code anywhere necessary, including before specific instructions and functions.

Since the profile is dependent on the input set provided by the user, the quality of the profile depends entirely on the coverage and representativeness of the profiled inputs. Regressive test-suites and benchmarks are often provided with applications for testing with real-world loads, and these can be used to generate profiles for our tool.

2.2.1 Control Flow Analysis

The control flow is simply a trace of instructions executed, represented as a set of basic blocks, addressed by their address offset from the image load address. Instructions are identified by image names and offsets rather than instruction pointers to control for different image load addresses across different profile runs. We annotate each basic block with the number of times it was executed and the number of instructions of the block.

One of the advantages of control flow analysis is detecting hot-spots. Hot-spot's are region of the code that is executed frequently. This helps us to focus our efforts only on hot-spot regions in the binaries which are relatively smaller in size, but offer significant performance benefits if parallelized.

2.2.2 Data Flow Analysis

During profiling we collect a complete data dependence graph [2] by instrumenting the instructions that have been executed. Using basic block boundaries (i.e.heads and tail pointers) we analyze the data dependence trace to identify data sharing between basic blocks. For the purpose of this paper, we say that two basic blocks are in conflict if either block has instructions dependent upon the other. This also includes self conflicts between the same basic block.

Hence, if instruction L in basic block B_i has a dependency on instruction M in basic block B_j, we define B_i and B_j to be in conflict. We aggregate these dependencies across several runs with different input sets to find other data dependencies that may arise Coalescing data-dependencies at instruction level to basic block level reduces the size of the dependency graph, reducing the size of the input to our analysis algorithms when compared to at the instruction level.

2.3 Profile Analysis: Conflicting Blocks

Our goal in this section is to generate parallel sets of basic blocks. A parallel set is a set of basic blocks which may be executed together in sequence, however, can be executed in parallel with other parallel sets. We hypothesize that (1) all basic blocks with potential data conflicts can be put into the same parallel set, and (2) all basic blocks having no data conflicts can be parallelized with any other blocks. Blocks that have no data dependencies can trivially be parallelized - given the control flow of basic blocks $A \rightarrow B \rightarrow C$ where none of A, B, or C conflict, A, B, and C can run at the same time. The problem becomes somewhat more complicated when basic blocks have data dependencies, and even more so as those dependencies become more complex.

We view parallelization of basic blocks as a subset problem. For each basic block B_i in the program, we collect a set of conflicts S_i. Our goal is to build the maximal subset of blocks such that we can run the most blocks in parallel, given the conflict sets and control dependencies.

Unfortunately, analyzing dynamic traces generated at binary level brings about it's own sets of problems. One of the largest difficulties is dealing with very big trace files, creating an intractably large search space. In order to attack this subset problem, we use the apriori algorithm, ranking each possible set by total number of instructions parallelized [1]. The apriori algorithm is a well known algorithm used in data mining to create association rules between data sets. We create associations based upon conflicts, and then generate sets of conflicts, maximizing the number of supporting basic blocks. We also filter out blocks that share the same conflict set but are not reachable using a depth-limited Dijkstra's algorithm.

Another important task is detecting false dependencies. There are often false negatives in dependencies for some registers which are independent across blocks. For example, stack pointers often have dependencies carrying over which realistically can be parallelized. These dependencies need to be handled as special cases, to realize truly independent basic blocks.

2.4 Hot Spot Analysis and Load Balancing

Thanks to the annotated profiles generated in the previous step, POWER is able to perform an effective cost-benefit analysis of each potential parallelism. By capturing both the number of times that each basic block is executed and the number of instructions in each block, we can detect hot spots: parts of the program that may benefit most from parallelism. We plan to use this information to guide our search to determine if the cost of adding the parallelism (from potential overhead) is worth the gain.

Additionally, if there is more than one possible parallel workflow, we can use different heuristics to balance allocation of basic blocks to parallel sets. One such heuristic, for example, can be the number of instructions in each thread. Load balancing is useful to avoid any skew in assigning basic blocks to parallel sets. For example if in the first workflow, parallel set 1 has 20 blocks, and parallel set 2 has 1, and in workflow 2, parallel set 1 is assigned 10 blocks and parallel set 2 is assigned 11, the second workflow will be chosen as it is more balanced and will give better performance benefit.

2.5 Binary Rewriting

Binary rewriting transforms an executable into a different but functionally equal program. Binary rewriting has been applied to various fields, including computer security [21] and code compaction [6] optimization.

After discovering candidates for parallelization, the next step is to apply these in the binary. We propose to utilize binary rewriting to parallelize the selected control flows. We are currently exploring static binary analysis tools including IDA Pro [11], Diablo [8], the SOLAR Project [20], ETCH [9], and hex binary editors to rewrite and parallelize the binaries.

3 An Example: How is parallelism discovered?

In this section we show how potential parallelism can be found using our technique for a simple control and data flow example. Figure 2 shows a small portion of the basic blocks from a sample control flow trace. A typical execution may yield hundreds or thousands of basic blocks. We focus our efforts on hot-spots: regions that are heavily executed. We present one such hot spot below:

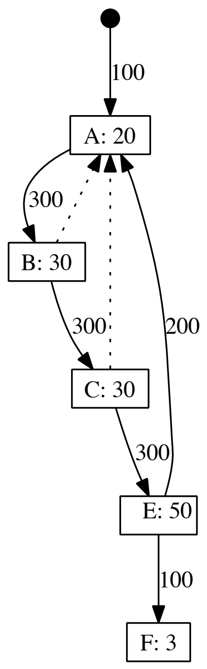

Figure 2: A Simple Profile

Basic blocks are labeled with an identifying letter, followed by the number of instructions in the block. Solid edges are labeled with the number of times that it is traversed, and dotted edges indicate data dependencies. Hence, the edge A-B with annotation 300 means that there has been 300 jumps from A to B. In this profiled execution, we can always run blocks B, C, and E at the same time (or B and C followed by E and A). The control flow is simple enough that we can intuitively tell that we can always run our parallel schedule without interfering with program correctness, thanks to Bernstein's conditions.

Consider, a somewhat more complex control flow, such as the one represented in Figure 3. In this case, we can again run B, C, and E concurrently. However, we cannot assume that block C will be executed after B, as D may also be executed afterwards. In this case we must be more conservative in our parallelization strategy. However, if the system supports speculative parallelization such parallelizations are still useful.

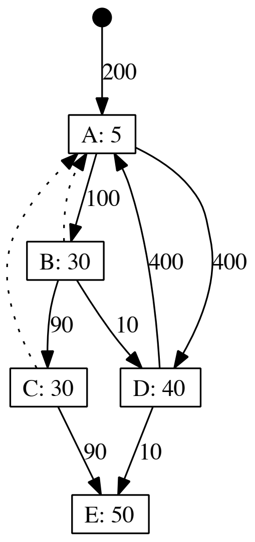

Figure 3: A Complex Profile

4 Related Work

The past few years have seen renewed interest in parallelizing applications. There are a wide range of approaches towards parallelization, including speculative, automatic and programming language oriented parallelization. To the best of our knowledge, the nearest work similar to our approach has been done by Yardimci and Franz who explored a dynamic recompilation approach (recompiling the binary to a parallel version at run-time) towards parallelizing binaries [23], whereas our approach focuses on a static binary re-writing hence avoiding the overhead of dynamic recompilation of the binary.

Programming language based approaches to assist in parallelization range from parallelization primitives such as Intel TBB's [13], X10 [18], Cilk-5 [10], ATOM or parallelizations provided in FORTRAN, and parallel function libraries extremely restrict practical usage (for example restricting the use of pointers, or in case of a previous existing implementation of sort/search which cannot easily be modified/applied etc.).

Parallel programming paradigms such as Open MP etc. are slowly being adopted, but require a large amount of expertise. Additionally these techniques cannot help to port legacy sequential applications. On the other hand auto-parallelization techniques(such as our own) can be more easily adopted and applied to legacy sequential applications.

Pure compiler solutions to auto-parallelization employ static analysis [2, 15] techniques to recognize and execute independent regions of code in parallel, but are extremely limited due to the challenge of scaling pointer analysis in large and complex applications. These issues are further compounded by the inability of compilers to recognize complex control flows and irregular flow patterns, or to properly judge costs and benefits of parallelizing different portions of code. Profile information, however, provides the clues that compilers are missing - exact number of times that blocks are executed, what they conflict with, etc. Dynamic auto-parallelization [7, 17] techniques are also used in some places, but their use cases are extremely limited as they delay the parallelization work to the run-time of the program, hence adding run-time overhead. In contrast our approach avoids this by analyzing and transforming the binary offline, resulting in minimal run-time overhead.

In the last few years there have been other profile-driven approaches. Tournavatis et al. [22] identified potential parallelisms using machine learning to analyze cumulative profiles. Another profile driven approach was proposed by Kim et al. [14], as an extension to existing Intel Parallel Studio [12]. Both of these approaches do not actually parallelize programs, but provide assistance to programmers in optimizing their applications by hand. Additionally, the profiles are generated at an intermediate level and require special compiler infrastructures. In contrast we focus on the binary, and do not require the source code.

Finally, there have been several attempts to use user feedback to help programmers parallelize their applications. These approaches do not necessarily ensure correctness of proposed transformations, as that is left as an exercise to the programmer. The most noteworthy amongst these is the Intel Parallel Studio [12] Suite, an IDE provided by Intel to help programmers analyze target applications by suggesting potential candidates for parallelism. The authors have also worked on COMPASS, a collaborative and interactive IDE to assist programmers in parallelizing applications [19]. COMPASS is an ongoing research project which uses a wisdom of the crowd approach to gather parallelizations for legacy sequential software.

5 Ongoing and Future work

We have tested our approach on small examples and are implementing a tool to apply our parallelizations to application binaries and are investigating tools such as IDA Pro [11](a static binary disassembler and analyzer) to assist in the development of such a prototype.

Our approach relies on generating cumulative profile information. This has a possibly significant drawback, that we may classify a code segment wrongly as parallelizable. There can always exist a data input for which our parallelization transform fails. We are currently exploring techniques, to ensure the correctness and verification of the parallel transforms that POWER will apply.

We also plan to explore how transformations that are typically used by compilers, can be used in POWER. For example, typical loop splitting (ie, placing half of the executions of a loop in one thread and half in another) requires directly modifying instructions, and not just parallelizing blocks. As we have defined it, POWER can not make these optimizations, but we are working to adapt it to support these transformations. A lot of work has been done in auto-parallelization techniques in compilers, we wish to explore how these algorithms/techniques can be applied to the executable

6 Conclusion

In this paper, we have presented POWER a generic, executable-only, profile-driven solution for finding candidates of parallelism. We proposed basic block level parallelism, whereby we parallelize basic block's within an executable. This technique allows us to maintain the integrity and structure of code segments of programs, while introducing parallelism. Our profile-driven approach counters well known aliasing problems faced by auto-parallelizing compilers. Moreover, POWER is generic and is programming language independent. We believe our tool will provide a new enhancement to the auto-parallelization techniques currently available to developers. The work presented is part of an ongoing project at the PSL lab at Columbia University.

7 Acknowledgments

The Programming Systems Laboratory at Columbia is funded in part by NSF CNS-0717544, CNS-0627473 and CNS-0426623, and NIH 2 U54 CA121852-06.

References

[1] AGRAWAL, R., AND SRIKANT, R. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1994), VLDB '94, Morgan Kaufmann Publishers Inc., pp. 487–499.

[2] ALLEN, R., AND KENNEDY, K. *Optimizing Compilers for Modern Architectures:A Dependence-Based Approach.* Morgan Kaufmann, 2002.

[3] BERNSTEIN, A. Program analysis for parallel processing. In *IEEE transactions on Electronic Computers, EC 15* (2010), pp. 757–762.

[4] BURKE, M. G., AND CYTRON, R. K. Interprocedural dependence analysis and parallelization. *SIGPLAN Not. 39* (April 2004), 139–154.

[5] CHAKARAVARTHY, V. T. New results on the computability and complexity of points–to analysis. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2003), POPL '03, ACM, pp. 115–125.

[6] CHANET, D., DE SUTTER, B., DE BUS, B., VAN PUT, L., AND DE BOSSCHERE, K. Automated reduction of the memory footprint of the linux kernel. *ACM Transactions on Embedded Computing Systems 6*, 4 (9 2007), 23/1–23/48.

[7] CHEN, M. K., AND OLUKOTUN, K. The jrpm system for dynamically parallelizing java programs. In *Proceedings of the 30th annual international symposium on Computer architecture* (New York, NY, USA, 2003), ISCA '03, ACM, pp. 434–446.

[8] Diablo, *A Better link time analyzer*. http://diablo.elis.ugent.be/.

[9] Etch, *Instrumentation and Optimization of Win32/Intel Executables*. http://etch.cs.washington.edu/overview/index.html.

[10] FRIGO, M., LEISERSON, C. E., AND RANDALL, K. H. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation* (New York, NY, USA, 1998), PLDI '98, ACM, pp. 212–223.

[11] Ida pro, *Disassembler and Debugger Suite*. http://www.hex-rays.com/idapro/.

[12] Intel corporation, *Intel Parallel Studio*. http://software.intel.com/en-us/intel-parallel-studio/home.

[13] Intel corporation, *Intel Thread Buiding Blocks*. http://software.intel.com/en-us/articles/intel-tbb/.

[14] KIM, M., KIM, H., AND LUK, C.-K. Prospector: A Dynamic Data-Dependence Profiler To Help Parallel Programming. In *2nd USENIX Workshop on Hot Topics in Parallelism* (2010).

[15] LIM, A. W., AND LAM, M. S. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1997), POPL '97, ACM, pp. 201–214.

[16] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2005), PLDI '05, ACM, pp. 190–200.

[17] PETERSEN, P., AND PADUA, D. A. Dynamic dependence analysis: A novel method for data dependence evaluation. In *Proc. Fifth Workshop on Languages and Compilers for Parallel Computing, LNCS 757* (1992), Springer Verlag, pp. 64–81.

[18] SARASWAT, V. A., SARKAR, V., AND VON PRAUN, C. X10: concurrent programming for modern architectures. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, NY, USA, 2007), PPoPP '07, ACM, pp. 271–271.

[19] SETHUMADHAVAN, S., ARORA, N., GANAPATHI, R. B., DEMME, J., AND KAISER, G. E. Compass: A community-driven parallelization advisor for sequential software. In *Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering* (Washington, DC, USA, 2009), IWMSE '09, IEEE Computer Society, pp. 41–48.

[20] Solar, *Software Optimization at Runtime and Linktime*. http://www.cs.arizona.edu/solar/.

[21] SONG, Y., AND FLEISCH, B. D. Utilizing binary rewriting for improving end-host security. *IEEE Transactions on Parallel and Distributed Systems 18* (2007), 1687–1699.

[22] TOURNAVITIS, G., WANG, Z., FRANKE, B., AND O'BOYLE, M. F. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2009), PLDI '09, ACM, pp. 177–187.

[23] YARDIMCI, E., AND FRANZ, M. Dynamic parallelization and mapping of binary executables on hierarchical platforms. In *Proceedings of the 3rd conference on Computing frontiers* (New York, NY, USA, 2006), CF '06, ACM, pp. 127–138.