ParaShares: Finding the Important Basic Blocks in Multithreaded Programs

Melanie Kambadur, Kui Tang, and Martha Kim

Columbia University, New York, NY
{melanie, martha}@cs.columbia.edu, kt2384@columbia.edu

Abstract. Understanding and optimizing multithreaded execution is a significant challenge. Numerous research and industrial tools debug parallel performance by combing through program source or thread traces for pathologies including communication overheads, data dependencies, and load imbalances. This work takes a new approach: it ignores any underlying pathologies, and focuses instead on pinpointing the exact locations in source code that consume the largest share of execution. Our new metric, *ParaShares*, scores and ranks all basic blocks in a program based on their share of parallel execution. For the eight benchmarks examined in this paper, ParaShare rankings point to just a few important blocks per application. The paper demonstrates two uses of this information, exploring how the important blocks vary across thread counts and input sizes, and making modest source code changes (fewer than 10 lines of code) that result in 14-92% savings in parallel program runtime.

1 Introduction

With massive-scale data to analyze, explosive growth in server and mobile core counts, and multithreading making its way into mainstream language specifications such as C++ [22], parallel software is officially ubiquitous. All parallel applications share the same fundamental goal of making the best use of resources: time, power, money, or some combination of these. To honor this goal, programs must be performant, bug-free, scalable, and not overly difficult to write or debug. Parallel program optimization poses particular challenges, as developers must uncover and address a nearly unbounded catalog of potential inefficiencies arising at any level of the stack, from relatively high level algorithmic and design choices, to program inputs, to source language implementation, to thread library selection, to operating system configurations, and the target hardware platform. Correcting performance inefficiencies requires programmers to have knowledge of, and potentially, take action at, multiple levels of the stack.

Many research and industrial tools have been introduced over the years to help programmers correct parallel performance inefficiencies. Generally these tools employ one of two broad strategies. The first is to look for specific kinds of errors, sometimes within targeted program regions such as a program's critical path. For example, tools may identify load imbalances [4], long waits [16, 8], lock contention [23, 6], I/O blocking [18], or unnecessary I/O [5]. One issue with

this approach is that each type of inefficiency may need its own tool or search procedure. The second general strategy is to troll for multiple or broader types of problems by tracking hardware and system events. Some tools track thread traces and program runtimes to predict which threads will scale poorly in future runs [9, 12]. Other tools take a hardware perspective, monitoring instruction counts, CPU utilization, thread preemption rates, and cache latencies [14, 7, 26, 15, 21, 1]. Unfortunately, linking hardware events back to software can pose a number of challenges. For example, event data may need to be aggregated across parallel threads. Additionally, it is often difficult to connect certain events precisely to software, meaning that areas of code identified as problematic may be large.

This paper utilizes a third strategy for performance debugging. *ParaShares* identify very tiny regions of code that take up the majority of multithreaded execution, agnostic to the type or cause of underlying performance pathologies. Their only goal is to precisely point programmers to the lines in their program that would benefit most from optimizations. A ParaShare is a rankable score that measures each basic block's share of a total parallel program's execution. The rankings are similar to hot block analyses that report the most frequently executed basic blocks and their CPU use. However, ParaShares factor in the degree of program parallelism at each block execution, providing a more accurate reflection of a block's contribution to wall-clock execution time. The weighting scheme downgrades the importance of blocks that execute during highly parallel program phases. As a result, it ranks blocks that mostly run during serial phases relatively higher in importance as they tend to consume a greater fraction of runtime.

Per block parallelism weights are enabled by parallel block vector (PBV) profiling [17], a recent technique which was introduced for the purpose of improving micro-architectural design. In the next section, we explain this new application of PBVs in more detail, comparing ParaShares to existing analyses and motivating the use of such a precise and fine-grained performance debugging tool (Sect. 2). We then present a step by step procedure for collecting and analyzing ParaShares (Sect. 3). Finally, using ParaShares for eight benchmark applications, we examine how the key optimization points move as input size and parallelism vary (Sect. 4.1), and make small, ParaShare-targeted source code changes that, although only a few lines apiece, speed the benchmarks 14–92%(Sect. 4.2).

2 ParaShares

ParaShares are a new way to rank the basic blocks in a parallel program according to their relative multithreaded runtime contributions. This section defines ParaShares, describes how they differ from traditional hot block analyses, gives readers a first look at experimentally collected ParaShares, and makes a case for analyses that focus on fine-grained regions of code.

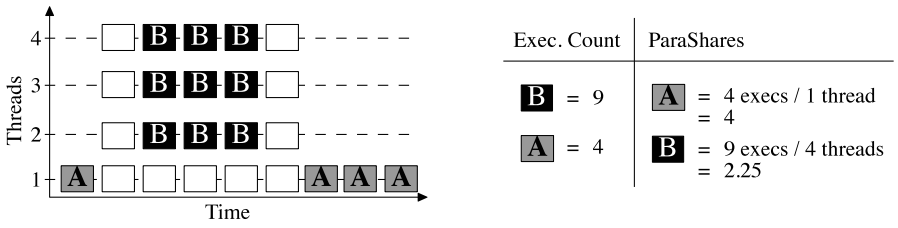

Fig. 1: **ParaShares rank basic blocks to identify those with the greatest *impact on parallel execution*,** weighting blocks by the runtime parallelism exhibited by the application each time the block was executed.

2.1 The Basic Concept

Basic blocks are small program fragments, constrained to be a linear sequence of instructions with a single entry point and a single exit point. As the program executes, some blocks will be executed very frequently, while others may execute rarely or not at all. The frequently executed blocks are called "hot" and are important optimization targets as they constitute a large share of an application's dynamic work. Hot block analysis has traditionally been used for a variety of purposes, including JIT translation [24], garbage collection optimizations [13], simulation points analysis [19], code cache management [20], and parallel performance debugging, for example, in Intel's VTune Amplifier [15].

ParaShares makes a subtle but important twist on traditional hot block analyses, weighting each basic block by the degree of parallelism exhibited by the program when the block was executed. Figure 1 illustrates the significance of this change. On the left is a program trace that highlights the execution patterns of two blocks of interest, A (gray) and B (black). For simplicity, we assume that both blocks have the same number of instructions and equal execution times, though in actual ParaShare computations this unlikely assumption is amended (Sect. 3.1). Simple counting reveals that B executes 9 times whereas A executes only 4, giving B a higher rank of importance. However, A may consume more of the program's execution time because its executions occur during serial phases of the program. To account for this nuance, ParaShares divides the executions by the degree of parallelism at execution time, in this example dividing B's 9 executions by the 4 threads that ran while B executed, and dividing A's 4 blocks by 1 for the single running thread. As a rule, parallelism is counted at the start of a basic block's execution to resolve any overlaps in block executions between threads. The resulting scores capture parallel execution shares more effectively, and in this case rank A and B in the opposite order of importance versus traditional execution counts.

2.2 A First Look At Real Applications

Figure 2 gives a first look at ParaShare block rankings for real applications, eight programs from the Parsec Version 3.0 [3] and Splash-2 [25] benchmark

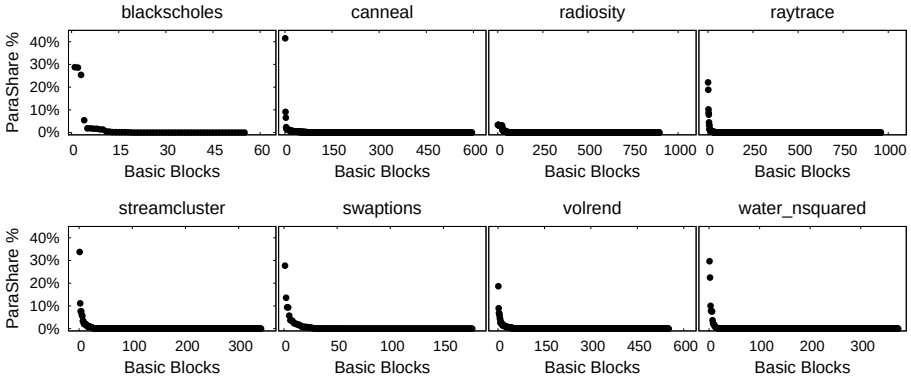

Fig. 2: **ParaShare rankings identify important blocks to target for multi-threaded performance optimizations.** These graphs show the ParaShare percentages (ordered from greatest to least share) of all the basic blocks in eight benchmark applications.

suites, namely `blackscholes`, `canneal`, `radiosity`, `raytrace`, `streamcluster`, `swaptions`, `volrend`, and `water_nsquared`. The Splash2x variant of Splash that is packaged with Parsec was used for its provision of multiple input sets. All of the applications are written in C and C++ and parallelized using pthreads with a variety of design patterns, including a mix of data and task parallelism. Each program was run alone using 24 threads and native input set sizes on a Dell PowerEdge R420 server. The server is dual socket with Intel Sandybridge E5-243 chips, each with six cores and two-way hyper-threading for a total of 24 effective cores. The system has 24GB of DRAM and runs Ubuntu 12.04.2 with the 3.9.11 version of the Linux kernel. The graphs show that just a few basic blocks (on the x-axis) per program dominate the ParaShare rankings (on the y-axis). The small number of important blocks is no surprise. However, ParaShare's ability to highlight blocks that are important in terms of wall-clock time instead of processor execution times combined across threads makes it possible to massively improve program performance with just minor code changes, as demonstrated later in Sect. 4.2.

2.3 Benefits of Fine Granularity

The well known 90-10 rule of thumb says that 90% of program execution time resides in just 10% of code. For our benchmarks, the rule holds: functions that consume roughly 90% of the execution represent 2.3-17.3% of the lines in the overall programs, or an average of 7.7%. Table 1 shows the exact line counts per benchmark, as well as line counts for the functions consuming 90% of the execution based on ParaShare computations.

The table also shows the number of lines of code contained in the basic blocks that are responsible for 90% of the ParaShare execution. Using block-granularity

Benchmark Application	Total Lines	90% Exec By Func Lines	90% Exec By Block Lines	50% Exec By Block Lines
blackscholes	564	68	34	21
canneal	1362	204	70	6
radiosity	11836	276	42	4
raytrace	10963	431	51	8
streamcluster	2539	439	12	5
swaptions	1550	359	28	10
volrend	4227	585	133	89
water_nsquared	2079	338	29	18

Table 1: **A case for fine-grained identification of performance inefficiencies.** To examine the functions that take up 90% of the parallel execution, a programmer must examine an average of 338.5 lines per program. To examine the basic blocks that consumed the same amount, they would need to look at an average of 50 lines per program.

hotspots rather than function hotspots saves programmers from looking at an average of 289 lines per benchmark. In fact, basic block hotspots save enough that we could coin a new 90-2 rule of thumb, because 90% of the parallel execution is taken up by just 2.4% of the program source lines according to our precise ParaShares analysis. The top 50% of program execution could be covered by searching an even more targeted set of code; programmers would need to look at only 20 source lines per application, or 1% of the overall program. The block versus function savings is particularly important in unfamiliar applications with lengthy functions and lots of loops — a feature common to some of the scientific benchmarks used in this study. For example, `volrend` has one function with three sets of doubly nested loops, and we found more than a few instances where a single function contained four or more loops.

3 Collecting and Analyzing ParaShares

This section describes the framework for translating source code to ParaShare rankings, examines the robustness of ParaShare rankings across trials, and experimentally demonstrates that ParaShare weighting can significantly change top blocks' relative importance versus traditional profiling.

3.1 The Collection Framework

From a user's perspective, ParaShares are straightforward to collect. They require recompilation, a single program run with the usual inputs and usual outputs, and the execution of a post-processing script. Under the covers, ParaShares are more complex, as depicted in the framework in Figure 3. The first two steps come from previous work, while the remaining steps are new to this work.

Step 1. Compile the source program with Harmony. ParaShares use parallel block vectors, or PBVs [17], to count how many times each basic block

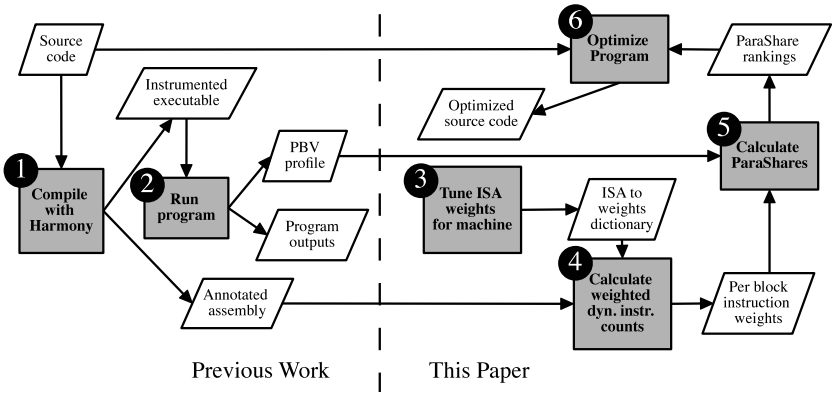

Fig. 3: **The Collection Framework.** To collect ParaShares, programmers re-compile their program with a specialized compiler, then execute it once with normal inputs. Profiling files produced at compile and execution time are analyzed in post-processing to give the programmer a list of ParaShares and corresponding source code locations.

executes at each thread count exhibited over the course of a program's execution. PBVs are collected via compiler instrumentation, requiring source programs to be compiled with Harmony [11], an extended version of LLVM. Compilation with Harmony produces two outputs: an annotated assembly code file and an instrumented executable file.

Step 2. Execute the program once to collect a PBV. After compilation with Harmony, a single program run with normal inputs produces a PBV profile as well as the usual program outputs.

Step 3. (Optional) Tune machine specific parameters. Optionally, ParaShares can incorporate machine specific instruction weights to account for differences in opcode processing or memory access times. If used, these weights should be stored in a dictionary mapping instruction types to latency factors. Opcode dependent latency factors are often already available online; for example, latency factors for our machine are available in [10]. These latency factors suggest multiplying conditional operations by two, add instructions by one, and divide instructions by 30, as well as multiplicative factors for other types of instructions. Due to the overwhelming significance of total instruction count, our applications' ParaShare rankings showed minimal sensitivity to these latency factors. However, latency factors could have more of an effect for other applications and architectures.

Step 4. Calculate per block static instruction counts. Next, the total (possibly weighted) instruction count per basic block is calculated. The instruction contents of each block are available in the annotated assembly file produced earlier by Harmony. With weighting, a sum of the weights of each instruction in the block produces a total block weight ($Weight_b$). As an unweighted alternative, a simple count of the instructions per block suffices.

Step 5. Calculate ParaShare rankings. The ParaShare for each block b is computed using the block's static instruction weight and dynamic thread weight. Specifically, the sum of each block's executions at thread count t ($Execs_{b,t}$) are divided by t. This formula is loosely related to the runtime calculation used in Quartz [2], but we apply it here at a smaller granularity and for a different purpose. The ParaShare of block b is the product of this dynamic thread weight and the static block weight, where max is the maximum number of threads that ever executes concurrently in the program:

$$\text{ParaShare}_b = \sum_{t=1}^{max} \frac{\text{Execs}_{b,t}}{t} \times \text{Weight}_b$$

As necessary for further analysis, the absolute ParaShare for each basic block can be normalized to the program's total ParaShare (the sum of ParaShares across blocks).

Step 6. Use the ParaShare rankings for performance optimizations or other analyses. Finally, ParaShares can be mapped back to the source code via compiler debug information in the assembly code.

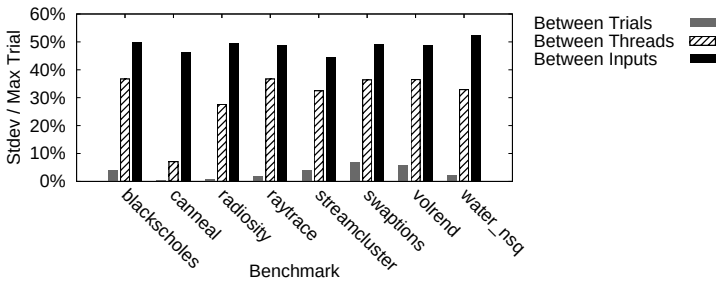

Fig. 4: **Robustness of the metrics.** Runtimes and basic block execution counts can change across program trials, but the differences are small relative to the differences in ParaShares collected across varying thread counts or input set sizes.

3.2 ParaShare Robustness

A program's parallel behavior may be inconsistent across runs, changing block execution counts or overall program runtime. Despite these variations, a single profiling run can produce representative ParaShares, particularly if the purpose of collection is to examine and optimize the hottest blocks with the highest ParaShares. Figure 4 plots the standard deviations of a program's total ParaShares as a fraction of the maximum program total ParaShare across ten trials. Across runs with the same thread count and input, this division was never more than 7% and averaged only 3.2%. The variation is small when compared with variations between trials given different maximum thread counts (31% on

average) or different input sizes (48%). In addition to the magnitude of the overall program ParaShare staying consistent between trials, the ranking of individual basic blocks varies minimally, and changes only in lower ranked blocks with ParaShares of 2% or less. This is not the case across thread counts and input sizes as explored in Sect. 4.1.

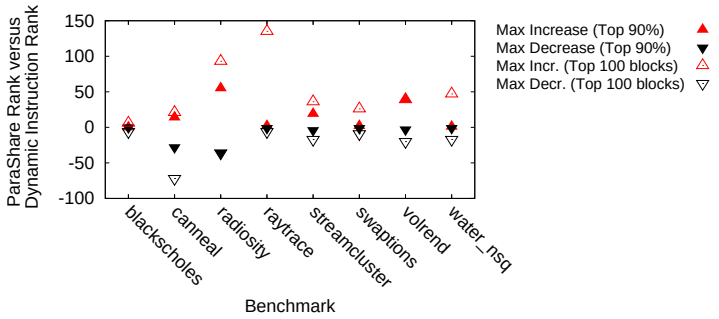

Fig. 5: **ParaShares versus unweighted rankings** for the blocks representing 90% of ParaShare execution and for the larger set of the top 100 blocks per application. ParaShares often significantly impact the relative importance of a block versus dynamic instruction count rankings *not weighted by parallelism.*

3.3 Impact of ParaShare Weights

ParaShare's utility is not just to locate small regions of significant source code, but to locate significant code that other tools may not highlight. Figure 5 shows differences in the top block rankings according to ParaShares versus according to dynamic instruction counting that is *unweighted by parallelism.* The graph shows the minimum and maximum differences between two sets of 'top' ParaShare blocks: first, those blocks representing 90% of the execution of 24-thread count runs (as previously depicted in Fig. 2, this block count varies by application), and second, the top 100 ParaShare blocks per program. From the first set of differences, we see significant changes in four of the eight applications. One block in `radiosity` is ranked 55 spots higher by ParaShares than by dynamic instruction counts, and another is ranked 36 spots lower by ParaShares. In the second, larger set of 100 block differences, rankings change significantly amongst almost all of the applications. Individual blocks (in `raytrace`) jump as many as 135 spots in the ParaShare rankings, and fall as many as 72 spots (in `canneal`).

4 ParaShares in Real Applications

This section uses ParaShares to explore real applications in more detail, examining how important blocks differ across inputs and thread counts and using ParaShares for targeted micro-optimizations.

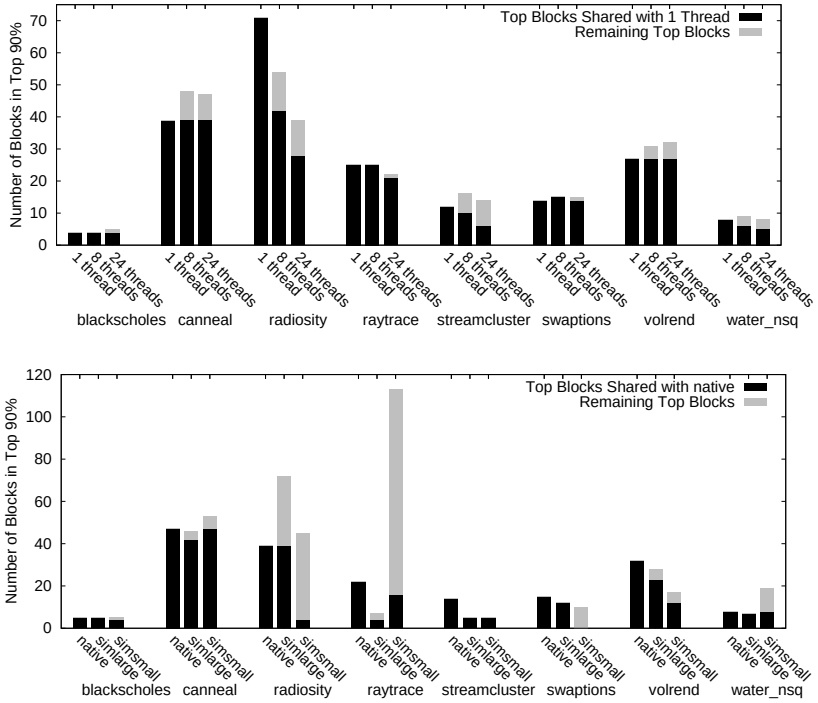

Fig. 6: **Top ParaShare blocks vary across thread counts and input sizes.** These differences suggest that optimizations may need to be targeted to the level of expected parallelism and to input size for maximum effect.

4.1 How Top Blocks Differ

A small handful of basic blocks dominates the ParaShare ranks and overall execution. These top blocks can vary significantly across thread counts and input sizes, suggesting that as environmental circumstances change, optimizations may need to be re-applied or re-targeted for maximum effect. The top of Fig. 6 plots the number of blocks that make up the top 90% of each application when run with 1, 8, and 24 threads. The number of hot blocks can change significantly. For example, when run with 1 thread, 71 blocks comprise the top 90% of `radiosity`, but this number shrinks to 39 when the application runs with 24 threads. The black part of each bar indicates how many of the top 90% were also in the top 90% of a serial run. Thus, the 39 key blocks in 24-threaded `radiosity` include 11 blocks that were not important to single-threaded `radiosity`. While it is not evident in the plot, the ranking of blocks within the top 90% changes as well: the block with the highest ParaShare in single-threaded `radiosity` falls to 26th place in 24-threaded `radiosity`. The highest ranking block in single-threaded `streamcluster` remains atop the list in 24-threaded `streamcluster`, but the second place block falls off of the list entirely, dropping from 19% of the ParaShare to 0.3%, and the third ranked block falls to the ninth spot.

Hotspots shift even more with program input variations. Black portions of the bars in the bottom of Fig. 6 show the overlap of other input sizes with the largest, native input size. `Raytrace` shows the biggest sensitivity to input, with the number of top blocks exploding from 22 to 113 between the native and simsmall inputs. The first two top blocks stay the same across inputs, but their combined ParaShare drops from 40.9% to 28.3%, while the third block drops even more sharply from 10.2% to 2.6%. In `swaptions`, none of the top native blocks appear amongst the top simsmall blocks. These variations indicate the surprising degree to which the internal dynamics of a parallel application can shift depending on simple parameters such as thread count and input size.

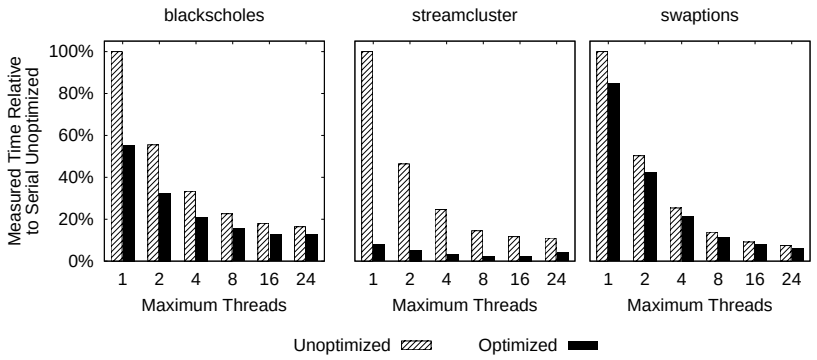

Fig. 7: **ParaShares pinpoint inefficiencies that lead to significant opportunities for optimization.** With the extremely targeted profiling provided by ParaShares, we were able to improve benchmark performance by up to 92% through source code changes less than 10 lines long.

4.2 Performance Tuning

Using ParaShares to target particularly important lines of source code, we made extremely simple and short source code changes to reduce application runtimes 14-92%. Figure 7 shows the effect of optimizations to `blackscholes`, `streamcluster`, and `swaptions`. Both optimized and unoptimized versions were compiled with LLVM's `-O3` optimization set. Our manual optimizations improve computation time, but do not make any algorithmic or parallelization changes. As a result, the savings shrink as thread counts increase, but they remain significant (up to 82%) even at large thread counts.

In `blackscholes`, the top two blocks consume nearly 60% of the overall runtime given 24 threads and native input sizes. These blocks are found in the kernel function which calculates financial option values. By collapsing the original 20 temporary variables in the function to 3, we alleviated register pressure resulting in a 44.6% performance improvement at one thread and 22% at 24 threads. For `streamcluster`, the top blocks are found in the `dist()` function, which computes the squared Euclidean distance between two `Point`s, each of which is a

struct with pointers to arrays of `float` coordinates. Inspecting the line of code in question (the body of a nested for loop), we guessed that the compiler missed an opportunity for common subexpression elimination, then modified the code to force it to do so. This change halved the loop body's original four array lookups and two subtractions and reduced register pressure, saving 92% of the serial runtime, and 64% of the 24 thread runtime. Finally, the top blocks in `swaptions` correspond to a few nested loops within the `HJM_SimPath_Forward_Blocking.cpp` file. We experimentally unrolled these loops one to four times to find the optimum unrolling level for each. In addition to the inability of the compiler to dynamically test a variety of unrolling levels, these opportunities may have been missed because the loops involve nested accesses to custom data structures. In total, our loop optimizations resulted in a 15% savings for a single threaded swaptions execution, and a 19.7% savings for a 24-thread execution.

Given the simplicity of our optimizations, the performance savings are disproportionately large. Across a datacenter or many nodes in a distributed system, the savings could be even more important, and potentially financially significant as well. Best of all, we were able to make the optimizations quickly, because ParaShares allowed us to focus our efforts on just a few lines of code rather than thousands.

5 Conclusions

ParaShares provides a new lens through which to analyze multithreaded application performance. In contrast to most parallel performance optimization techniques, ParaShares do not target a specific type of inefficiency or level of the system stack. Instead, ParaShares track parallelism from the code's point of view, weighting each basic block execution by the whole program's parallelism at the time of the execution. This fine-grained scoring makes it simple to localize important lines of code, even in large or unknown programs. Once important code is localized, more extensive analysis and optimizations can be precisely targeted, leading to small code changes with big performance effects.

References

1. L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCTOOLKIT: tools for performance analysis of optimized parallel programs. *Concurrency and Computation*, 22(6), Apr. 2010.
2. T. E. Anderson and E. D. Lazowska. Quartz: a tool for tuning parallel program performance. *SIGMETRICS*, 18:115–125, 1990.
3. C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011.
4. D. Bohme, F. Wolf, B. R. de Supinski, M. Schulz, and M. Geimer. Scalable critical-path based performance analysis. In *IPDPS*, 2012.
5. M. Chabbi and J. Mellor-Crummey. Deadspy: a tool to pinpoint program inefficiencies. In *CGO*, 2012.

6. G. Chen and P. Stenstrom. Critical lock analysis: diagnosing critical section bottlenecks in multithreaded applications. In *SC*, 2012.
7. K.-Y. Chen, J. Chang, and T.-W. Hou. Multithreading in Java: Performance and scalability on multicore systems. *Transactions on Computers*, 60(11), Nov. 2011.
8. K. Du Bois, S. Eyerman, J. B. Sartor, and L. Eeckhout. Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior. In *ISCA*, 2013.
9. K. Du Bois, J. B. Sartor, S. Eyerman, and L. Eeckhout. Bottle graphs: Visualizing scalability bottlenecks in multi-threaded applications. In *OOPSLA*, 2013.
10. T. Granlund. Instruction latencies and throughput for AMD and Intel x86 processors, Feb. 2012. `http://gmplib.org/~tege/x86-timing.pdf`.
11. Harmony Parallel Block Vector Collection Tool. `http://arcade.cs.columbia.edu/harmony`.
12. Y. He, C. E. Leiserson, and W. M. Leiserson. The Cilkview scalability analyzer. In *SPAA*, pages 145–156, 2010.
13. X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: Improving program locality. In *OOPSLA*, Oct. 2004.
14. Y. Huang, Z. Cui, L. Chen, W. Zhang, Y. Bao, and M. Chen. HaLock: hardware-assisted lock contention detection in multithreaded applications. In *PACT*, 2012.
15. Intel® Corporation. Intel® Parallel Amplifier 2011. `http://software.intel.com/en-us/articles/intel-parallel-amplifier/`.
16. J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt. Bottleneck identification and scheduling in multithreaded applications. In *ASPLOS*, 2012.
17. M. Kambadur, K. Tang, and M. A. Kim. Harmony: Collection and analysis of parallel block vectors. In *ISCA*, June 2012.
18. B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. Bruce, I. Karen, L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn parallel performance measurement tools. *IEEE Computer*, 1995.
19. E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder. Using simpoint for accurate and efficient simulation. In *SIGMETRICS*, volume 31. ACM, 2003.
20. H. Shi, Y. Wang, H. Guan, and A. Liang. An intermediate language level optimization framework for dynamic binary translation. *SIGPLAN Notices*, 42(5), May 2007.
21. STMicroelectronics, Inc. PGProf: parallel profiling for scientists and engineers, 2011. `http://www.pgroup.com/products/pgprof.htm`.
22. B. Stroustrup. C++11 the new ISO C++ standard, 2013. `http://www.stroustrup.com/C++11FAQ.html`.
23. N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield. Analyzing lock contention in multithreaded applications. In *PPoPP*, 2010.
24. N. Topham and D. Jones. High speed CPU simulation using JIT binary translation. In *MOBS*, volume 7, 2007.
25. S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *ISCA*, 1995.
26. W. Yoo, K. Larson, L. Baugh, S. Kim, and R. H. Campbell. ADP: automated diagnosis of performance pathologies using hardware events. In *SIGMETRICS*, 2012.