Parallel Block Vectors: Collection, Analysis, and Uses

PARALLEL BLOCK VECTORS (PBVS) LINK A MULTITHREADED APPLICATION'S CODE TO THE VARYING DEGREES OF PARALLELISM IT EXHIBITS AT RUNTIME. THIS NEW PERSPECTIVE HELPS USERS REASON ABOUT HARDWARE AND SOFTWARE INTERACTIONS TO IDENTIFY PERFORMANCE-IMPROVEMENT OPPORTUNITIES. THE AUTHORS DEFINE PBVS AND SHOW TWO ARCHITECTURAL APPLICATIONS FOR THE PROFILES. THEY ALSO DEMONSTRATE HOW THE OPEN-SOURCE TOOL HARMONY ENABLES SIMPLE, LOW-OVERHEAD COLLECTION OF PBVS.

Melanie Kambadur
Kui Tang
Martha A. Kim
Columbia University

As multicores began dominating programmable architectures from mobile to the datacenter, both researchers and industry leaders increasingly focused on the efficiency of parallel programs. Application profilers and measurement tools have helped parallelization efforts, often by exposing hard-to-identify performance bottlenecks. Such tools generally collect runtime statistics from the perspective of processes or threads;[1-5] this perspective, while useful to software engineers, does not capture the full picture of a parallel program's execution.

Here, we introduce parallel block vectors (PBVs), a new type of parallel application profile that exposes parallelism from the perspective of a basic block. PBV profiles add a second dimension to conventional basic block vectors[6] to track dynamic parallelism, recording the degree of parallelism exhibited by the application at each basic block execution. Such a profile enables two new kinds of parallel program analyses:

- the serial and parallel portions of an application can be teased apart for individual analysis, and
- users can track the changes in parallelism of fine-grained code regions.

With careful engineering effort, PBVs are neither complex nor expensive to gather, even at the relatively fine granularity of a basic block. We have developed Harmony, an open-source PBV collection tool that uses an LLVM compiler pass to instrument a multithreaded application, thus requiring little more effort from users than recompilation. At runtime, Harmony instrumentation incurs an average of 16 percent slowdown and minimal resource overhead (as measured by register spills and cache miss rates) for eight applications from Parsec,[7] a suite of non-HPC multithreaded benchmarks.

Programmers can use the application characteristics uncovered by PBVs and Harmony to improve multithreaded execution in various ways. Here, we focus on how PBVs can enable advances in microarchitectural design. For example, we used them to separate the parallel and serial portions of several applications and found that the instruction mixes for these subsets of code differ—often significantly—from the overall

Published by the IEEE Computer Society

program instruction mix. In the context of heterogeneous processors, such as those analytically motivated by Hill and Marty,[8] programmers can apply this information to tailor heterogeneous cores to better suit their anticipated parallel and serial workloads. After presenting two such architecturally focused analyses, we conclude with a discussion of future opportunities, including applications of PBVs in other parts of the system and ideas for improving PBV collection.

PBV profiles

A PBV profile consists of one histogram for each basic block, indicating the degree of parallelism exhibited by the application each time the block executed. More formally, for each basic block b in a program, an array of counters $counter_{b,t}$ indicates how many times the block was executed when the application had t threads running. From this profile, it's easy to find blocks that executed at a particular thread count t (all b such that $counter_{b,t} > 0$) or the thread counts each time a particular block b was executed ($counter_{b,t}$ for all values of t).

Figure 1a shows the source code for a simple, unoptimized matrix multiplication program; Figure 1b shows the corresponding PBV profile. Each row in the profile corresponds to a static basic block and each column to a particular degree of parallelism. In this example, the program created four threads, which—in addition to the initial thread—results in up to five parallel threads. Each cell in the profile indicates the number of dynamic executions of a particular block at a particular degree of parallelism. The heatmap shading used in Figure 1b helps visually capture the contents of a profile, which is particularly helpful when analyzing large applications.

PBVs create two new opportunities to better understand parallel programs. First, they let users identify specific basic blocks that run at a particular thread count. For example, examining the first column in Figure 1 reveals that 14 blocks make up the serial phases of matrix multiplication's execution, with main:6 and worker:5 dominating the dynamic mix. Second, a user can monitor any program region of interest to see the phase or phases in which the code executed. For example, blocks worker:4-6 in Figure 1b correspond to the inner multiplication loop, a critical region in terms of performance. As desired, the profile reveals that this code is largely executed at high thread counts.

When tracking an application's parallelism, there are multiple ways to count threads. *Nominal thread count* includes all created threads regardless of whether they're running or blocked. *Effective thread count* excludes blocked threads and counts only runnable threads. *Running thread count* includes only the runnable threads that have actually been granted access to a processor by the scheduler. We collect profiles for nominal and effective thread counts, but we don't count running threads for two reasons. First, running thread count is strongly dependent on the availability of hardware resources and the behavior of the scheduler, thus revealing more about those two system aspects than about the application. Second, counting running threads requires polling the OS, which is likely to substantially slow and perturb the target program's execution.

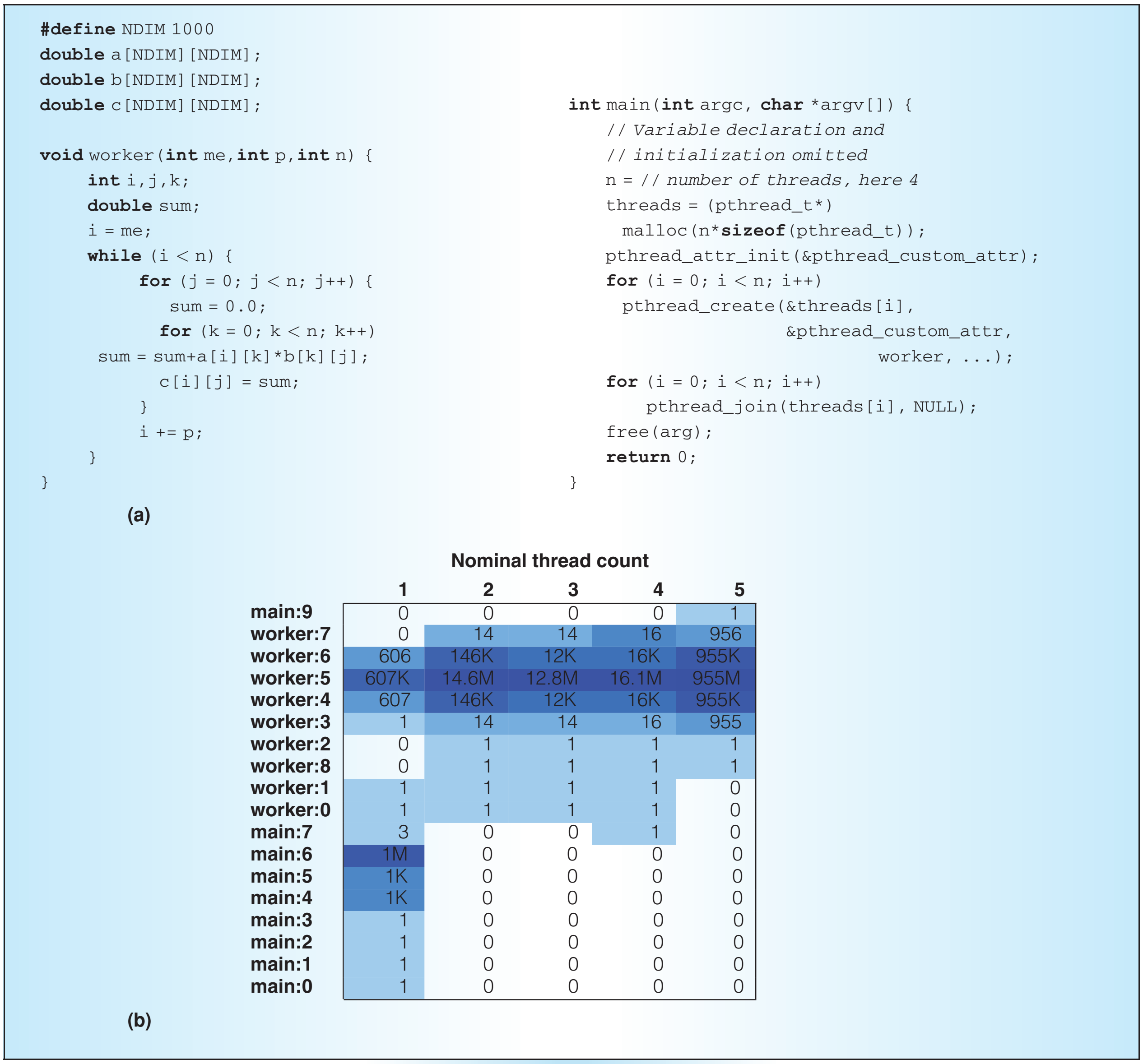
Collecting PBVs with Harmony

Although there are multiple ways to collect PBVs, we designed and implemented a compiler instrumentation pass that inserts PBV collection instructions into multithreaded programs at compile time. Compile-time instrumentation is an attractive option for collecting PBVs because it has low runtime overhead and makes portability trivial: PBVs can be collected on any architecture or language that the compiler supports.

Our Harmony PBV collection tool is an instrumentation pass within the LLVM compiler.[9] Here, we describe the high-level details of Harmony's implementation and evaluation; our paper in the ISCA 2012 proceedings has more detailed information.[10] The latest open source version of Harmony can be downloaded at http://arcade.cs.columbia.edu/harmony.

Harmony design and implementation

Harmony is intended to be the last compiler pass executed, after the program has



```
#define NDIM 1000
double a[NDIM][NDIM];
double b[NDIM][NDIM];
double c[NDIM][NDIM];

void worker(int me,int p,int n) {
     int i,j,k;
     double sum;
     i = me;
     while (i < n) {
          for (j = 0; j < n; j++) {
             sum = 0.0;
            for (k = 0; k < n; k++)
      sum = sum+a[i][k]*b[k][j];
            c[i][j] = sum;
          }
          i += p;
     }
}
```

(a)

```
int main(int argc, char *argv[]) {
    // Variable declaration and
    // initialization omitted
    n = // number of threads, here 4
    threads = (pthread_t*)
      malloc(n*sizeof(pthread_t));
    pthread_attr_init(&pthread_custom_attr);
    for (i = 0; i < n; i++)
      pthread_create(&threads[i],
                         &pthread_custom_attr,
                                    worker, ...);
    for (i = 0; i < n; i++)
        pthread_join(threads[i], NULL);
    free(arg);
    return 0;
}
```

Nominal thread count

	1	2	3	4	5
main:9	0	0	0	0	1
worker:7	0	14	14	16	956
worker:6	606	146K	12K	16K	955K
worker:5	607K	14.6M	12.8M	16.1M	955M
worker:4	607	146K	12K	16K	955K
worker:3	1	14	14	16	955
worker:2	0	1	1	1	1
worker:8	0	1	1	1	1
worker:1	1	1	1	1	0
worker:0	1	1	1	1	0
main:7	3	0	0	1	0
main:6	1M	0	0	0	0
main:5	1K	0	0	0	0
main:4	1K	0	0	0	0
main:3	1	0	0	0	0
main:2	1	0	0	0	0
main:1	1	0	0	0	0
main:0	1	0	0	0	0

(b)

Figure 1. Parallel block vector (PBV) for matrix multiplication. The source code (a). The profile indicates the execution frequency of each basic block at each possible thread count (b).

been fully optimized and the program control-flow graph has been finalized. Harmony inserts instrumentation instructions at several key program events, beginning with instructions to allocate and initialize a profile when the application starts, then with instructions to write the final profile to a file at application exit. At thread creation and exit points, Harmony injects code to increment and decrement the thread count. When tracking effective thread count, the counter is also decremented upon entry and incremented upon exit from any blocking call, such as a lock acquisition. Finally, each basic block is instrumented with an increment of the appropriate entry in the profile matrix. Harmony annotates both the collected profiles and the LLVM assembly file with unique basic block IDs so that the profiles can be cross-referenced with the original application assembly instructions in post-processing. Harmony is currently compatible with any Pthreads application that LLVM can compile.

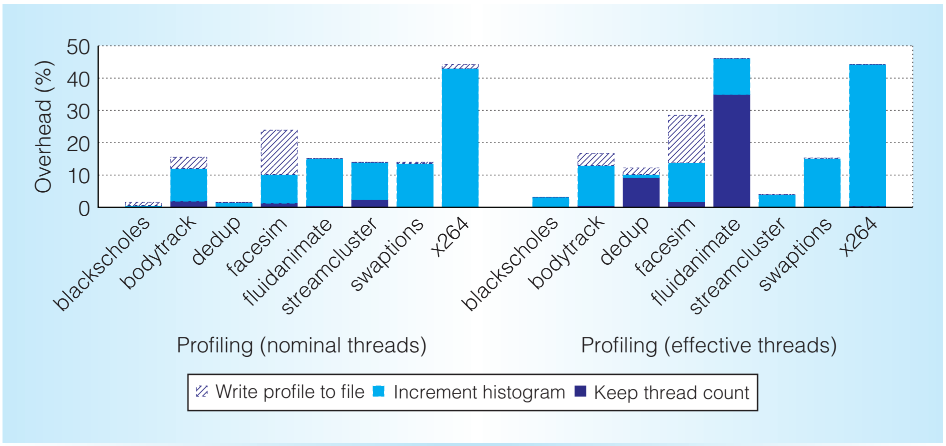


Figure 2. Low overhead of instrumentation. Program slowdown due to profile collection ranges from 2 to 44 percent, with an average overhead of 18 percent.

Two key optimizations reduce the overhead of profile collection and minimize program perturbation:

- The most time-consuming profiling activities (profile setup, aggregation, and cleanup) take place prior to and after a program's regular execution.
- The required runtime instrumentation is minimized: at each block execution, only a read (of the current thread count) and a counter increment (to update the execution count of that block) takes place.

We describe additional overhead minimization strategies in the original conference paper.[10]

Validating Harmony

To check Harmony's impact on parallel program execution, we measured the increase in both execution time and resource pressure added by instrumentation. In our experiments, both Harmony-instrumented and uninstrumented executables were compiled using –O3, the maximum compiler optimization setting. The machine we used had four 2.0 GHz cores and 3.3 Gbytes of RAM, and ran Linux Ubuntu version 8.04. Data was collected for eight Parsec benchmarks.

Figure 2 shows the runtime overhead due to profiling for each of the eight applications averaged across 20 program runs. Nominal thread-count profiling added 16 percent to runtime on average while effective thread-count profiling added 21 percent. The higher overhead for effective thread-count profiling is expected due to the additional thread-counter activity, and is corroborated by the fact that the applications with the most blocking calls (`dedup` and `fluidanimate`) show the largest increases. As Figure 2 indicates, 3 percent of the total overheads are attributable to time spent writing the profile to a file after the program has finished. Thus, the overheads that affect program execution are 13 percent and 18 percent for nominal and effective thread counts, respectively.

Harmony adds little resource pressure to instrumented programs. A 7.5 percent average increase in register spills was measured for instrumented versions of applications, with new spills confined to the profile setup and cleanup activities that occurred prior to and after program execution. According to Cachegrind measurements, instrumented programs experienced negligible cache perturbation: level-one (L1) instruction and data-cache miss rates increased by at most 0.06 percent and 0.2 percent, respectively, and there was no measurable impact on the storage hierarchy beyond the L1 structures.

PBV applications

Figure 3 shows visual representations of the profile for eight Parsec benchmarks. In Figure 3, each row corresponds to a static basic block and each column to a nominal

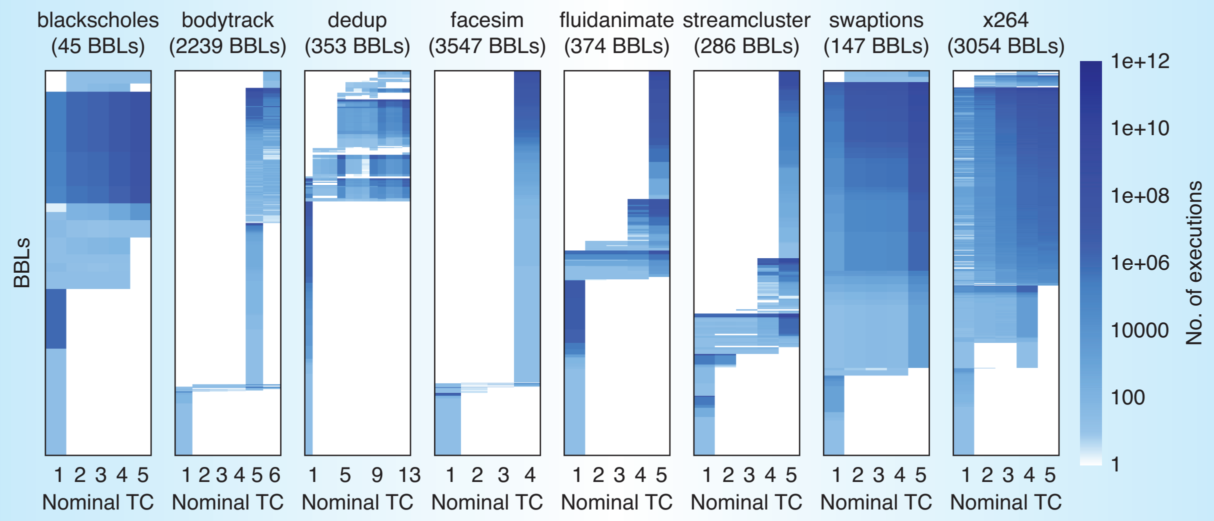


Figure 3. PBVs for Parsec. These heatmaps are a visualization of the Harmony-produced profiles. For the given application, they show the number of times (shading) each static block (row) was executed at each degree of parallelism (column).

thread count. Due to space constraints, the figure shows only the nominal thread-count heatmaps, but the following benchmark analyses use data from both nominal and effective profiles.

As the profiles show, in several applications—`bodytrack`, `dedup`, `facesim`, `fluidanimate`, and `streamcluster`—basic blocks display a strong affinity for either serial or parallel phases. By contrast, large regions of the remaining applications—`blackscholes`, `swaptions`, and `x264`–execute during both serial and parallel phases.

It is well known that an application's serial portions limit parallel speedups,[8] but what exactly do those serial portions look like? Are they amenable to acceleration? We now explore these questions.

Serial and parallel application partitions

Knowing how much of a program runs in parallel and how much runs serially is useful for many purposes. Tools such as Intel's VTune Amplifier XE[1] identify the serial fraction of an application's runtime so that software engineers can improve the parallelization of their programs. These metrics are also useful when estimating the scalability of a particular parallelization according to Amdahl's law.[8]

PBVs make it possible not only to quantify a program's serial portion, but also to map that region back to the specific basic blocks that comprise it. To get this information, we can classify each basic block into one of three categories: *serial* (never executed with a thread count greater than one), *parallel* (always executed with a thread count greater than one), or *mixed* (sometimes executed in serial regions, other times in parallel regions).

From an architect's perspective, the pure serial blocks make natural targets for specialized serial processors or accelerators. The mixed blocks, which run both in parallel and in serial, are likely of interest to all system designers because they can represent application areas with communication overheads or resource contention. Identifying the mixed blocks helps improve their execution through better scheduling algorithms, additional hardware resources, or code transformations.

Figure 4 shows the breakdown of static and dynamic basic blocks by class (serial, mixed, or parallel). Significant portions of several applications are neither purely serial nor purely parallel but rather belong to both regions (the mixed class). This is true of both nominal and effective thread counts. The trends are similar—but even more pronounced—when counting dynamic basic block executions. Thus, when we

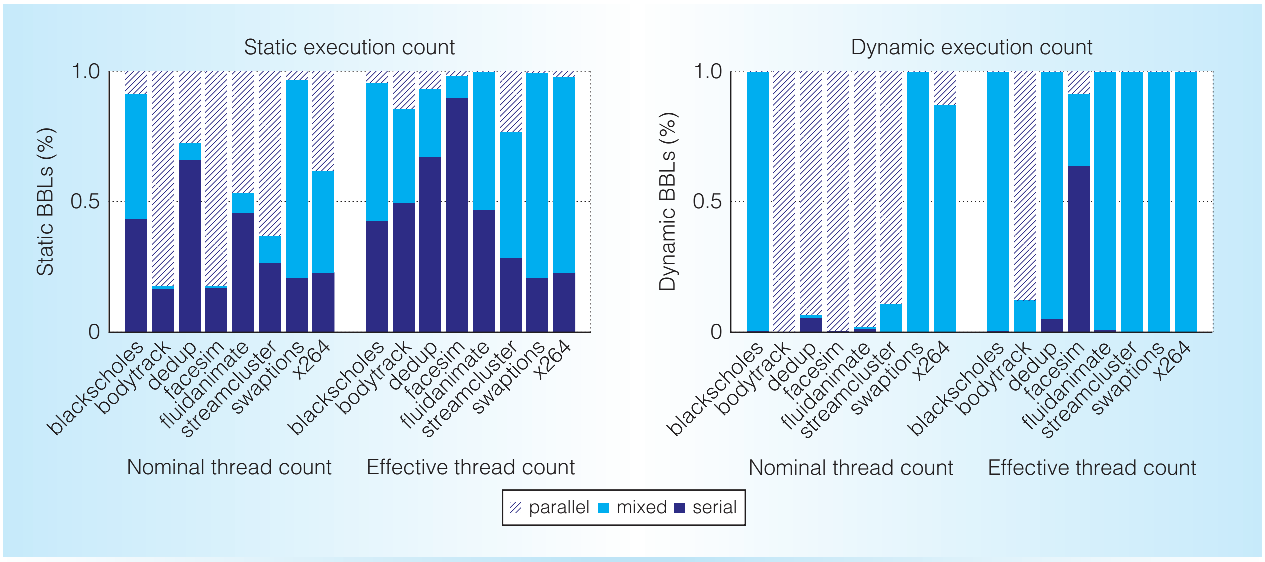


Figure 4. Classifying basic blocks by parallelism. These graphs show the percentage of blocks that execute only serially (serial); blocks that execute both serially and in parallel (mixed); and blocks that execute only in parallel (parallel) for each application, for both nominal and effective thread counting, and for both static and dynamic block executions.

talk about Amdahl's law and a program's serial and parallel phases, those phases often do not correspond to different application portions. One hypothesis is that such blocks result from library code that is called both from the serial and parallel phases.

In terms of the serial, mixed, and parallel classifications, it's easy to identify which applications are well parallelized and which are not. For example, from the static nominal viewpoint, `bodytrack` and `facesim` seem to be equally parallel. However, viewed from the dynamic effective profiles, `bodytrack` has a larger proportion of blocks actually running in parallel, whereas `facesim` apparently suffered from blocking threads and its parallel blocks were less-frequently executed than its serial blocks. To better approximate actual runtime, one can use dynamic instruction counts, obtainable by cross-referencing these dynamic basic block execution counts with the static block sizes found in the Harmony-annotated assembly file.

Having laid the groundwork, we can now examine the content of blocks in each of these classes in greater depth.

Program features by degree of parallelism

Recent interest in heterogeneous multicore architectures spans not only the architecture community but others such as operating systems, high-performance computing, and programming languages.[11,12] The key idea behind heterogeneous processing is specialization: different cores on a heterogeneous machine can address the varied needs of modern workloads while maximizing performance and efficiency. For example, when portions of a program cannot be adequately parallelized, an aggressive, superscalar, out-of-order, no-holds-barred processor can be employed to reduce this serial execution time.

However, if heterogeneous cores are to address the specialized needs of certain application sections, it is important to understand how to specialize these processors. PBVs make it possible to distinguish features of parallel and serial phases. The following analysis uses our earlier classification scheme, wherein each block belongs to exactly one category: always serial, always parallel, or mixed.

Figure 5 compares the dynamic instruction mixes for each category and the program as a whole. All of the x86 opcodes that occurred in the application were classified into one of eight categories: loads and stores, loads of effective addresses, integer arithmetic, floating-point arithmetic, comparisons, conditional control transfers, unconditional control transfers, and synchronization.

For most applications, serial basic blocks show significantly different instruction

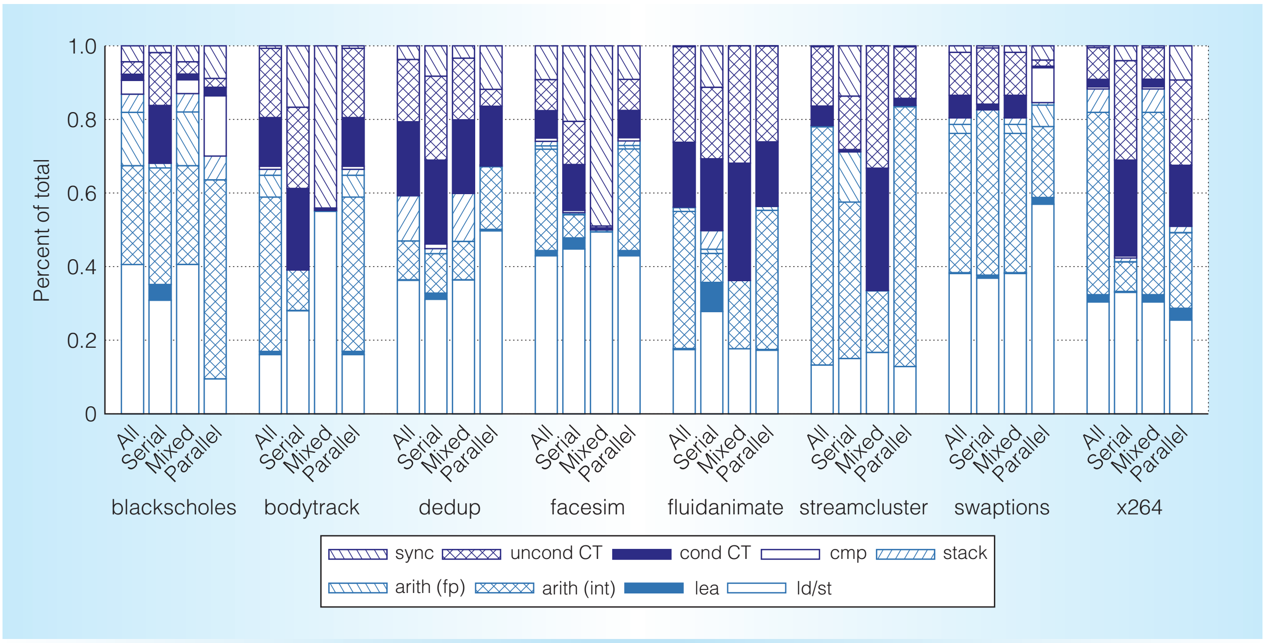


Figure 5. Opcode mix by class. Dynamic instruction mixes for the entire program compared with the mixes for each basic block class (serial, parallel, and mixed). In all applications, the instruction mixes for both purely serial and purely parallel blocks differ significantly from whole program mixes.

mixes than the overall program, indicating a potential opportunity when designing the microarchitecture of aggressive cores for heterogeneous CMPs. Consider the `blackscholes` application. Across the whole program, floating-point operations account for more than 20 percent of the dynamic instructions. If this were the only instruction mix considered—as is currently the case—the aggressive processor for serial regions might waste unnecessary space and expense on floating-point units; as the graph shows, the serial blocks actually require fewer floating-point operations than the program as a whole. Instead, the `blackscholes` serial phases have a higher concentration of control and integer arithmetic, suggesting that resources would be better spent on the branch predictor, for example.

The data in Figure 5 show a similar pattern in each benchmark. In every case, either the serial or parallel portions (and sometimes both) have substantially different instruction mixes than the application as a whole. However, across these applications, there doesn't seem to be a consistent pattern of *how* the instruction mixes change. For example, the serial portions in `blackscholes` reduced the need for floating-point units, while the serial portions of `x264` show increased rates of unconditional control transfers.

Just as opcodes vary, the state upon which the serial and parallel portions of a program operate varies relative to the overall program. Figure 6 shows the memory interactions of the three parallelism classes. As with opcodes, the different parallelism phases display different proportions of memory operations than the overall application.

Future opportunities for PBVs

PBV profiles are lightweight and easy to create. They're also flexible: their block-centric view ties architectural and system events back to the application at multiple levels. Basic blocks can be decomposed into instructions or recomposed into larger code segments such as critical sections, functions, algorithms, files, or even complete applications. This flexibility makes PBVs relevant to performance specialists ranging from computer architects to systems designers to compiler writers and software engineers. We hope to see PBVs applied to a variety of topics beyond the experiments presented here, perhaps enabled by Harmony's release.

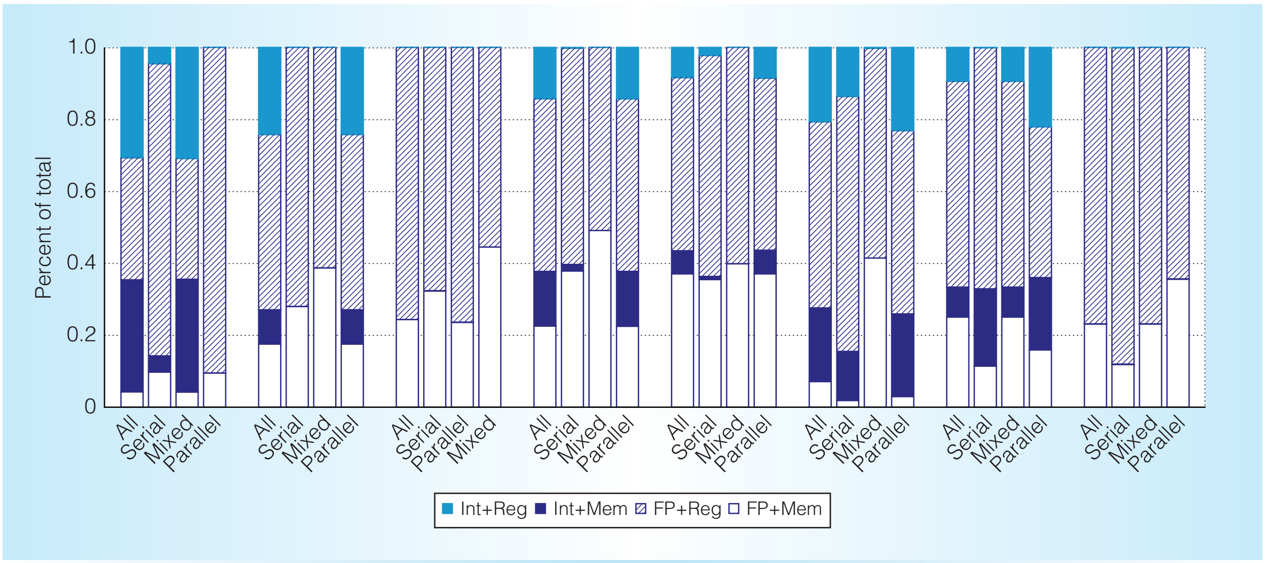


Figure 6. Memory interaction by class. The proportion of memory operations for serial and parallel basic blocks differs from the proportion in the overall program.

Hardware/software interaction

The dynamic collection of PBVs means that the profiles can capture not only algorithmic system effects, but also those of the architecture and operating system. Coupled with the ability to pinpoint fine-grained regions of real-world applications that belong to serial, parallel, or both program phases, PBVs provide important knowledge as hardware platforms increase in parallelism and heterogeneity. Moving forward, PBVs can assist architects in more precisely identifying specialization targets, help schedulers quickly evaluate a set of possible core mappings to select the best, and help programmers debug the performance and potentially even energy of their applications on current and future hardware.

Applications beyond architecture

Although this article highlights architectural applications of PBVs, they could be applied in many other areas of computer science. In operating systems, PBVs could be used to identify concurrency errors, as they reveal the exact number of threads that dynamically execute during each program section. PBVs can also help developers better understand program scalability—for example, by differentiating always-serial program portions from program sections that are only temporarily serial due to resource contention or thread communication. Other applications in compilers include assisting in automatic parallelization efforts helping determine when to apply certain kinds of optimizations in a dynamic compiler.

Strategies for improving PBV collection

We made significant efforts to keep Harmony's overhead low, but there are ways to further reduce collection overheads. In future versions of Harmony, we could analyze block call paths and instrument always-executed-together basic blocks, or *superblocks*, at only one point, instead of at each block. Alternatively, a user (or profiler) could first identify a program's hotspots, and PBVs could then be collected only for blocks of interest.

Harmony represents just one method for collecting PBV profiles. Conceptually, PBVs are applicable to *any* programming language that compiles to basic blocks and can work with any threading model. Although Harmony currently supports only C and C++ programs written using Pthreads, it could be extended—or another tool built—to collect PBVs for programs written in Java, Haskell, or X10, using OpenMP, Cilk, or any number of future languages and threading models.

Like puzzles turned sideways, new perspectives can yield new insights. Unlike existing profiles that examine parallel programs from the perspective of an individual thread or process, PBV profiles show runtime statistics per basic block and laterally by parallelism phase. PBVs reveal which parts of a program belong to the serial and parallel phases of execution and in what proportion. Collecting PBVs is both fast and simple, incurring just 16 to 22 percent profiling overhead in our experiments. Furthermore, collection requires no manual code modification by users, and theoretically, profiles can be collected for any parallel language or library and on any multicore architecture. Their fast collection coupled with detailed dynamic information about program behavior makes PBVs broadly suited to an array of potential uses ranging from compiler optimizations to scalability analysis to microarchitectural design. MICRO

Acknowledgments

The US National Science Foundation partially supported this research through grant CNS-1117135.

References

1. Intel Corporation, ''Intel VTune Amplifier XE,'' http://software.intel.com/en-us/intel-vtune-amplifier-xe.
2. S.S. Shende and A.D. Malony, ''The Tau Parallel Performance System,'' *Int'l J. High Performance Computing Applications,* vol. 20, no. 2, 2006, pp. 287-311.
3. STMicroelectronics, Inc., ''PGProf: Parallel Profiling for Scientists and Engineers,'' 2011; www.pgroup.com/products/pgprof.htm.
4. M. Itzkowitz and Y. Maruyama, ''HPC Profiling with the SunStudio Performance Tools,'' *Tools for High Performance Computing,* Springer, 2009, pp. 67-93.
5. S. Garcia et al., ''Kremlin: Rethinking and Rebooting gprof for the Multicore Age,'' *Proc. ACM Sigplan Conf. Programming Language Design and Implementation* (PLDI 11), ACM, 2011, pp. 458-469.
6. T. Sherwood et al., ''Automatically Characterizing Large Scale Program Behavior,'' *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* ACM, 2002, pp. 45-57.
7. C. Bienia, ''Benchmarking Modern Multiprocessors,'' doctoral dissertation, Dept. Computer Science, Princeton Univ., 2011.
8. M.D. Hill and M.R. Marty, ''Amdahl's Law in the Multicore Era,'' *Computer,* vol. 41, no. 7, 2008, pp. 33-38.
9. C. Lattner and V. Adve, ''LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation,'' *Proc. Int'l Symp. Code Generation and Optimization* (CGO 04), IEEE CS, 2004, pp. 75-86.
10. M. Kambadur, K. Tang, and M.A. Kim, ''Harmony: Collection and Analysis of Parallel Block Vectors,'' *Proc. Int'l Symp. Computer Architecture* (ISCA 12), IEEE CS, 2012, pp. 452-463.
11. E.B. Nightingale et al., ''Helios: Heterogeneous Multiprocessing with Satellite Kernels,'' *Proc. ACM SIGOPS Symp. Operating Systems Principles* (SOSP 09), ACM, 2009, pp. 221-234.
12. J. Gummaraju et al., ''Twin Peaks: A Software Platform for Heterogeneous Computing on General-Purpose and Graphics Processors,'' *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques* (PACT 10), ACM, 2010, pp. 205-216.

Melanie Kambadur is a PhD candidate in the Computer Science Department at Columbia University. Her research interests include hardware and software performance interactions and compilers. Kambadur has an MS in computer science from Columbia University.

Kui Tang is an undergraduate student in the Applied Physics and Applied Mathematics Department at Columbia University. His research interests include parallel programming models and large-scale machine learning. He is a member of the ACM and SIAM.

Martha A. Kim is an assistant professor in the Computer Science Department at Columbia University. Her research interests include computer architecture, parallel hardware and software systems, and energy-efficient computation on big data. Kim has a PhD in computer science from the University of Washington. She is a member of IEEE and the ACM.

Direct questions and comments about this article to Melanie Kambadur, Department of Computer Science, 450 Computer Science Building, 1214 Amsterdam Ave., MC 0401, New York, NY 10027-7003; melanie@cs.columbia.edu.