The Kremlin Oracle for Sequential Code Parallelization

The Kremlin open-source tool helps programmers by automatically identifying regions in sequential programs that merit parallelization. Kremlin combines a novel dynamic program analysis, hierarchical critical-path analysis, with multicore processor models to evaluate thousands of potential parallelization strategies and estimate their performance outcomes.

Saturnino Garcia
Donghwan Jeon
Christopher Louie
Michael Bedford Taylor
University of California, San Diego

Parallelization of existing sequential software is a complex task that typically requires intensive manual effort to achieve optimal performance. Researchers have sought to build parallelizing compilers that would completely automate the parallelization process.[1-3] These compilers typically struggle to exploit available parallelism because of the difficulty of proving parallel transforms' correctness and profitability. This difficulty is demonstrated in a comparison we performed between the output of Intel C++ Compiler (icc), a state-of-the-art, commercial parallelizing compiler, and manual parallelization efforts on the Rodinia benchmark suite.[4] The icc compiler could parallelize only 17 percent of the regions that were manually parallelized and none that were not. Similarly lackluster results have been shown with the NAS Parallel Benchmarks (NPB) benchmark suite.[5] The limited performance is a result of the key limitations of static program analysis: many dependencies can't be resolved statically, so the parallelizing compiler must be conservative to ensure correctness. (For more information on other approaches to parallelization, see our "Related Work in Parallelization" sidebar.)

Driven by the immediate need of programming emergent multicore chips, engineers have adopted a more manual approach that leverages tools such as OpenMP, OpenCL, and Cilk[6] to specify which program regions should run in parallel and to manage parallel execution across multiple cores. In the future, we must strive for tools that will address the remaining parts of the parallelization process. First, we need tools that identify which program regions are most promising for parallelization. Second, we need tools that suggest the transformations necessary to parallelize these regions. The University of California, San Diego, Kremlin project focuses on creating tools that achieve these goals. In this article, we report our recent success in meeting the first goal, as implemented in the UC San Diego Kremlin open-source programming tool.

By extending prior work on critical-path analysis (CPA)[7] to incorporate real-world constraints, we've enabled the Kremlin tool to implement a practical oracle that predicts outcomes for sequential-code parallelization. The tool takes in an unmodified serial program and a few representative inputs, and outputs an ordered list of the regions that are likely to be the most productive for the user to parallelize. Additionally, Kremlin outputs an upper bound on the expected speedup, informing the user

Related Work in Parallelization

Parallelism profiling techniques can be classified into critical-path analysis (CPA)[1,2] and dependence testing.[3,4] CPA quantifies the total amount of parallelism in the program. However, it doesn't localize parallelism to a specific region, limiting its practical use. Dependence testing aims at checking the dependencies between different regions in a program, but it doesn't quantify the amount of parallelism available in each region, and it is highly sensitive to superficial program structure. In contrast, Kremlin's novel hierarchical critical-path analysis (HCPA) quantifies the amount of parallelism in each part of the program, providing a more holistic picture, and works well even if extensive program transformation is required to unlock parallelism.

Several parallel-performance estimation tools share the motivation of Kremlin's speedup estimation—notably, Cilkview,[5] Parallel Prophet,[6] and Intel Parallel Advisor's Suitability Tool (http://software.intel.com/en-us/articles/intel-parallel-advisor). However, these tools differ from Kremlin in that they require either already parallelized code or user annotation for parallelization. Kremlin is especially useful in the early stage of parallelization, then these tools can be used for finer performance tuning.

Researchers continue to examine fully automatic parallelization of serial programs.[7,8] These approaches work well with specific forms of loop-based parallelism that have traditionally been difficult to exploit on multicore processors, but they lack Kremlin's generality in locating exploitable parallelism. Although this article mainly discusses Kremlin's use in manual parallelization, Kremlin could be used in concert with automatic methods to improve the quality of parallelization. Kremlin provides strong evidence that a region will be profitable if it can be safely parallelized. An automatic parallelizing compiler could use this information to limit the scope of heavier-weight, but more precise, static analyses. Conversely, Kremlin could analyze the compiler's output and provide additional hints to the user as to how to improve the quality of the parallelization.

Prior publications on Kremlin's planning[9] and prediction[10] discuss more detailed implementation issues and provide additional experimental results.

References

1. T.M. Austin and G.S. Sohi, ''Dynamic Dependency Analysis of Ordinary Programs,'' *Proc. 19th Ann. Int'l Symp. Computer Architecture* (ISCA 92), ACM, 1992, pp. 342-351.
2. M. Kumar, ''Measuring Parallelism in Computation-Intensive Scientific/Engineering Applications,'' *IEEE Trans. Computers,* Sept. 1988, pp. 1088-1098.
3. M. Kim, H. Kim, and C.-K. Luk, ''Prospector: A Dynamic Data-Dependence Profiler to Help Parallel Programming,'' *Proc. USENIX Workshop Hot Topics in Parallelism* (HotPar 10), USENIX, 2010, http://static.usenix.org/events/hotpar10/poster.html.
4. J.R. Larus, ''Loop-Level Parallelism in Numeric and Symbolic Programs,'' *IEEE Trans. Parallel and Distributed Systems,* July 1993, pp. 812-826.
5. Y. He, C.E. Leiserson, and W.M. Leiserson, ''The Cilkview Scalability Analyzer,'' *Proc. 22nd ACM Symp. Parallelism in Algorithms and Architectures* (SPAA 10), ACM, 2010, pp. 145-156.
6. M. Kim et al., ''Predicting Potential Speedup of Serial Code via Lightweight Profiling and Emulations with Memory Performance Model,'' to be published in *Proc. 26th IEEE Int'l Parallel and Distributed Processing Symp.* (IPDPS 12), 2012.
7. S. Campanoni et al., ''HELIX: Automatic Parallelization of Irregular Programs for Chip Multiprocessing,'' *Proc. 10th Int'l Symp. Code Generation and Optimization* (CGO 12), ACM, 2012, pp. 84-93.
8. G. Ottoni et al., ''Automatic Thread Extraction with Decoupled Software Pipelining,'' *Proc. 38th Ann. IEEE/ACM Int'l Symp. Microarchitecture,* IEEE CS, 2005, pp. 105-118.
9. S. Garcia et al., ''Kremlin: Rethinking and Rebooting Gprof for the Multicore Age,'' *Proc. 32nd ACM SIGPLAN Conf. Programming Language Design and Implementation* (PLDI 11), ACM, 2011, pp. 458-469.
10. D. Jeon et al., ''Kismet: Parallel Speedup Estimates for Serial Programs,'' *Proc. ACM Int'l Conf. Object-Oriented Programming Systems Languages and Applications* (OOPSLA 11), ACM, 2011, pp. 519-536.

as to whether speedups will be too small to merit the effort.

Our results show that Kremlin can accurately predict the parallel speedup of a wide range of benchmarks. Moreover, it is effective at identifying the regions that should be parallelized, even when doing so requires complex transformations. Compared to codes parallelized by third-party experts, codes parallelized using Kremlin have on average 1.57× fewer user-parallelized regions and, in about 18 percent of the cases, implemented a more effective parallelization strategy than was employed by the experts, resulting in significant improvements in performance. We are preparing a public open-source release of the Kremlin tool later this year.

Introducing Kremlin

Kremlin guides a programmer through parallelization by presenting a list of program regions that should be parallelized. This list

```
$> make CC=kremlin-cc
$> ./srad 100 0.5 502 458 image.pgm
$> kremlin srad -model=openmp -num cores=4

Cores           1    2    4*    8    16      32      64
Speedup (est.)  1    2    4     8    15.89   31.58   62.35

   File (lines)        Coverage (%)   Self-Parallelism    Time reduction (%)
1  srad.c (262-296)    70.25          458.0               52.67
2  srad.c (306-325)    24.25          458.0               18.17
3  srad.c (247-251)    5.29           502.0               3.95
4  srad.c (226-227)    0.09           229916.0            0.07
5  srad.c (342-343)    0.04           229916.0            0.03
..                     ...            ...                 ...
```

Figure 1. Interacting with Kremlin. Only three steps are required to get results from Kremlin: first, compile the program with `kremlin-cc`; second, execute the program with its normal inputs; and third, run the parallelism planner (`kremlin`) with desired planning options. The planner lets users specify system-specific constraints (such as OpenMP on a four-core processor); it orders regions according to decreasing expected parallelization benefit, letting software engineers prioritize the most important regions. Each entry in the plan lists the coverage (percentage of serial execution time spent in that region), self-parallelism (the amount of parallelism in a region and not in its children regions), and the estimated upper bound on time reduction (as a percentage of serial execution).

constitutes a plan that, when followed, will minimize the number of regions that must be parallelized in order to maximize parallel performance.

A sample Kremlin session

Kremlin uses a simple usage model inspired by the GNU profiler (*gprof*). Figure 1 demonstrates the user's interaction with Kremlin. First, the programmer compiles a program with `kremlin-cc`, a special drop-in replacement for the machine's C or C++ compiler. `kremlin-cc` creates a compiled program binary that is augmented with instrumentation code that performs parallelism measurements during the program's execution.

Users then run the instrumented binary as they normally would. In addition to the normal program outputs, the instrumented binary creates a small log file that summarizes the parallelism identified across the program's execution. To analyze this profile, users will run the `kremlin` analysis tool with target-specific constraints such as the number of available cores and the target parallel runtime (for example, Cilk++, OpenMP, or OpenCL). Kremlin combines these target-specific constraints with the parallelism profile and examines many candidate parallel implementations of the program. As output, it produces a postparallelization speedup upper bound estimate and a detailed parallelization plan. The plan consists of an ordered list of regions that should bring the highest speedup after parallelization. Each entry in the plan provides important details to the programmer, such as the fraction of serial execution time spent in that region (*Coverage*), the amount of parallelism found solely in that region and not in its children regions (*Self-Parallelism*),[8] and the estimated upper bound on the time reduction from parallelizing that region (*Time reduction*).

Typically, a user is initially the most interested in the estimated speedup. If the estimated speedup is too low, the user might simply skip parallelization and save precious development time. If Kremlin reports that the expected speedup doesn't scale with the number of cores, the user can target fewer cores or change the core algorithm to make program performance scale better after parallelization.

If the user decides to parallelize the code, the user can examine the detailed parallelization

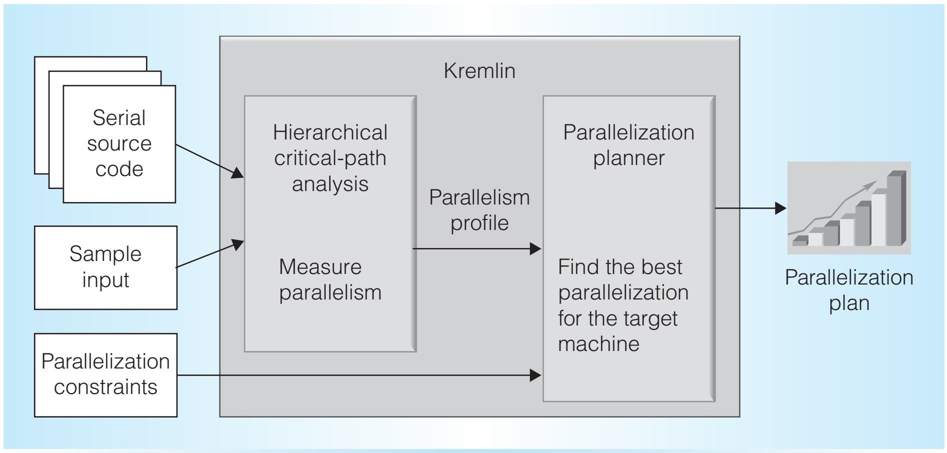


Figure 2. Kremlin's structure. Kremlin consists of two main components: hierarchical critical-path analysis (HCPA) and a parallelization planner.

plan, an ordered list of code regions that must be parallelized to achieve the estimated speedup. For each code region, the plan specifies the region's location, the expected benefit after parallelization, the parallelism type, and the approximate number of cores to be allocated. This list lets the programmer focus solely on the parts of the program that will be profitable when parallelized.

How the Kremlin oracle works

At the core of the Kremlin system is a heavyweight analysis of a sequential program's execution that is used to create predictions about the structure of a hypothetical, optimized parallel implementation of the program. These predictions incorporate both optimism and pessimism to create results that are surprisingly accurate. The pessimism comes from fundamental limitations imposed by the program's dynamic dependencies, as well as fundamental target machine constraints such as synchronization costs and core counts. The optimism comes from the assumption that, subject to these constraints, the programmer will discover a way to manage other performance issues such as cache locality. We use the term *practical oracle* to describe this process, because Kremlin is a real-life implementation of a seemingly omniscient tool that can predict complex outcomes.

At the heart of Kremlin is an extension of CPA, which quantifies a program's average parallelism.[7] Historically, practitioners found limited utility in CPA because it tends to be too optimistic to make accurate predictions. (See the "Critical-Path Analysis" sidebar for more discussion.)

Kremlin extends CPA to capture the impact of key parallelization constraints such as the number of available cores on the target, the exploitability of parallelism, and parallelization overhead. This extension comes in two parts. First, Kremlin greatly improves CPA's resolution by introducing a new hierarchical dynamic program analysis called *hierarchical critical-path analysis* (HCPA). Second, Kremlin employs a parallelization planner that compares the benefits of potential parallel implementations using real-world system constraints.

System architecture. Figure 2 illustrates Kremlin's system architecture. Like CPA, HCPA profiles a program's parallelism from its dynamic execution with a sample input. This runtime information lets HCPA capture input-dependent parallelism characteristics and frees Kremlin from relying on conservative static pointer analysis to disambiguate memory dependencies. Unlike CPA, which measures the whole program's parallelism, HCPA measures the localized parallelism of each region (typically, functions and loops) by modeling the program execution as a hierarchical region tree. Kremlin uses a new metric called self-parallelism to separate a region's parallelism from its children's parallelism.

After HCPA profiles parallelism, the parallelization planner finds the parallelization plan with the largest speedup. Because the

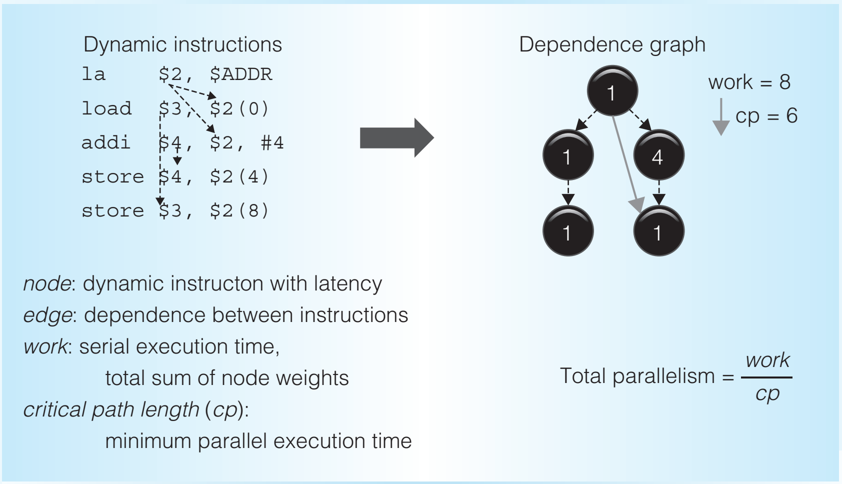
Critical-Path Analysis

Critical-path analysis (CPA) is a dynamic program analysis that computes the longest dependency chain through the dynamic execution of a serial program.[1] CPA is typically used to approximate the upper bound on parallel speedup without parallelizing the code. Figure A shows an example of CPA. First, CPA builds a dependence graph, wherein each node represents a dynamic instruction with latency, and each edge represents a register-, control-, or memory- dependence between instructions. Once CPA builds the dependence graph, it finds the length of the longest path—the critical path length (*cp*)— through this graph. This length represents a program's ideal parallel execution time. Finally, CPA calculates the program's total parallelism by computing the ratio between serial-execution time (*work*) and critical path length. Total parallelism quantifies the ideal speedup of the program when parallelized for a machine with infinite resources and zero communication and synchronization delay.

CPA has seen limited utility outside of research projects because it tends to be wildly optimistic: typical parallelism numbers far exceed the number of available cores and are often uncorrelated with actual speedups attained. Two main factors lead to a large gap between CPA-based estimation and measured speedup. First, CPA assumes a data-flow execution model that doesn't map well to von Neumann machines and imperative programming languages. Second, CPA naively assumes an ideal execution environment in which any parallelism is exploitable, unlimited cores are available, and no parallelization overhead exists—factors that severely limit the speedup on real systems.

Reference

1. M. Kim et al., ''Predicting Potential Speedup of Serial Code via Lightweight Profiling and Emulations with Memory Performance Model,'' to be published in *Proc. 26th IEEE Int'l Parallel and Distributed Processing Symp.* (IPDPS 12), 2012.


Dynamic instructions
Dependence graph
work = 8
cp = 6
1
1
4
1
1
node: dynamic instructon with latency
edge: dependence between instructions
work: serial execution time,
total sum of node weights
critical path length (cp):
minimum parallel execution time
Total parallelism = work / cp

```
la     $2, $ADDR
load   $3, $2(0)
addi   $4, $2, #4
store  $4, $2(4)
store  $3, $2(8)
```

Figure A. Example of critical-path analysis. CPA calculates the ideal parallel speedup without parallelizing the code by constructing the dependence graph from dynamic instructions.

achievable speedup greatly depends on target-specific parallelization constraints, the planner accepts key parallelization constraints (such as the OpenMP platform and an eight-core processor) and incorporates them in its planning. The planner explores many ways to parallelize the program, evaluates them, and picks the one with the highest speedup. Because Kremlin's hierarchical region model is expressive enough to represent a parallelization strategy, the planner also employs it: a parallelization plan consists of <region, allocated core count> tuples. A plan considers a region parallelized if it assigns more than one core to the region.

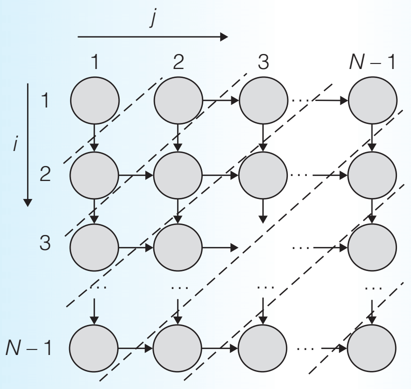
Kremlin's strengths. One of Kremlin's greatest strengths is its ability to find parallelism in many forms: task-based parallelism, pipeline parallelism, skewed parallelism, data parallelism, and many forms of loop-based parallelism (including DOALL and DOACROSS) are recognized, even if the code isn't currently structured to express it.

Figure 3 shows an example of Kremlin's power in detecting parallelism, even when this parallelism is masked in the current implementation. The code in Figure 3a presents two challenges to the parallelizing compiler. First, the 2D array has been implemented as an array of pointers to arrays. Second, the dependence structure between updates of values in the arrays creates cross-iteration dependencies in both loops. Parallelizing this code requires two key analyses. First, the compiler must recognize that a loop transformation technique called *loop skewing* can be applied, which restructures the loop so that execution traverses the array "diagonally" (as shown by the dotted lines in Figure 3b). Second, the compiler must prove, possibly using shape analysis, that none of the pointers in the first level of the array point to the same array in the second level (that is, there is no aliasing).

Some research compilers have implemented shape-analysis passes that could potentially decipher that the data structure is equivalent to a 2D array. Similarly, some research compilers can automatically infer loop skewing of static arrays. More generally, unlocking the parallelism latent in sequence programs could require that an arbitrary number of difficult analyses and transformations be composed. Because of complexity and runtime issues, modern compilers can't compose all these tasks simultaneously into one coherent analysis and transformation framework.

However, using runtime information, HCPA can easily identify and quantify the parallelism that's latent in the nested loop structure, alerting Kremlin and the user to the possibility that restructuring the code would result in large speedups. The user can work to iteratively transform the code sufficiently so that the compiler or runtime system can take it the rest of the way. In contrast, weaker dynamic-dependence testing-based frameworks would typically report no available parallelism because they can't see past the existing structure of the nested loops.

```
void calc_array(int** a)
{
    for(i = 1; i < N; ++i)
        for(j = 1; j < N; ++j)
            a[i][j] = a[i-1][j] + a[i][j-1];
}
```
(a)



(b)

Figure 3. Uncovering masked parallelism: loop with unexpressed parallelism (a); iteration dependency graph (b). Kremlin's underlying use of critical-path analysis (CPA) lets it uncover parallelism even when masked by a serial implementation. The code in (a) shows a nested loop operating on a 2D array with cross-iteration dependencies over both loops, making it appear serial. The iteration dependence graph in (b) shows that iterations can be grouped into independent sets, allowing parallel execution if loop skewing and interchange are used. Techniques relying on dependence testing would overlook this parallelism. Furthermore, the 2D array in (a) is represented as an array of pointers to arrays, thwarting a parallelizing compiler's attempt to statically analyze this section of code.

HCPA

CPA is a well-studied technique for quantifying the amount of parallelism in a program. It has, however, had limited utility in software engineering tools because of two main factors. First, the parallelism it reports is not indicative of the potential parallel speedup. Second, it calculates only a single parallelism number for the whole program, providing little actionable information for parallelization. To counter the limitations of CPA, we introduce HCPA, which allows realistic modeling of parallel program execution, aiding in the creation of a parallelization planner.

CPA versus HCPA

CPA assumes data-flow style execution that is hard to map onto conventional architectures, a problem illustrated in Figure 4a.

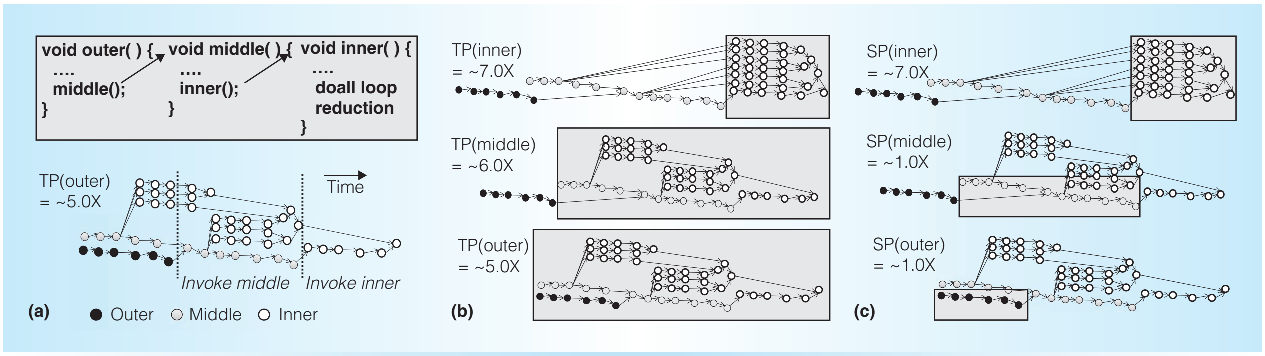


Figure 4. HCPA. CPA calculates total parallelism assuming an unrealistic data-flow style execution. Instructions from all three functions are smeared to their earliest point at which their inputs are available (a). Recursively applying CPA localizes parallelism. However, a child region's parallelism is counted toward its parent's parallelism, blurring the origin of parallelism (b). Self-parallelism captures a region's parallelism excluding its children's, further localizing parallelism to a region (c).

Each node represents a dynamic instruction, and its color shows where the instruction came from among three functions. CPA assumes any instruction can be executed as soon as all of its dependencies are resolved. As Figure 4a shows, instructions from `middle()` and `inner()` are already scheduled when `outer()` invokes `middle()`.

HCPA localizes parallelism to a specific region by independently applying CPA to each region. Figure 4b shows the result when HCPA applies CPA to each region and reports each region's total parallelism. However, CPA measures parallelism derived from both a region and its children; the total parallelism of `middle()` includes the parallelism from `inner()`, making it unclear how much parallelism `middle()` contains.

HCPA localizes parallelism to specific regions using self-parallelism. Figure 4c shows each region's self-parallelism. Because self-parallelism eliminates the parallelism originating from child regions, it's now clear that the `outer()` and `middle()` functions don't contain parallelism. By localizing parallelism to a region, HCPA allows realistic speedup estimation and parallelization planning in the parallelization planner.

Computing a region's self-parallelism

HCPA computes self-parallelism from each region's CPA results. A region's self-parallelism is conceptually the region's total parallelism divided by its children's total parallelism. Kremlin uses the following equation to compute self-parallelism of a region R ($SP(R)$) from the results of CPA:

$$SP(R) = \begin{cases} \dfrac{work(R)}{cp(R)} & R \text{ is a leaf} \\[2ex] \dfrac{\sum_{k=1}^{n} cp(child(R,k))}{cp(R)} & R \text{ is a nonleaf} \end{cases}$$

A leaf region's self-parallelism is the same as its total parallelism because there are no children that can affect its total parallelism. Kremlin can calculate a nonleaf region's self-parallelism by comparing its critical-path length against the sum of its children's critical-path lengths. Intuitively, the sum of the children's critical-path lengths represents the region's serial execution time when all descendant regions are fully parallelized.

Parallelization planner

Kremlin's second phase is parallelization planning. This phase finds the parallelization plan with the highest speedup and provides it to the user. A plan consists of <region, allocated core count> tuples, sorted by each region's potential speedup after parallelization.

Although self-parallelism and the serial execution time (*work*) provide the basis for estimating parallel performance, many other constraints significantly impact performance. For example, the target platform might support only a specific form of parallelism (for example, data parallelism in GPUs), the number of available cores can limit the achievable speedup, and the overhead from scheduling and synchronization might negate the benefit of parallelization.

Estimating parallel-execution time

To find the best parallelization plan, Kremlin must be able to evaluate different plans. Kremlin uses a target-specific parallel-execution time model to evaluate a given parallelization plan's performance. This model leverages both the hierarchical region model and the parallelism profiling data from Kremlin's first stage, working in a bottom-up fashion to estimate the time for all regions of the program. Kremlin starts by estimating the parallel-execution time of leaves in the region hierarchy because their estimated times don't depend on any other regions. Next, Kremlin calculates the parent regions' execution time on the basis of their already-calculated children.

To estimate a region's parallel-execution time, Kremlin uses key parameters gathered in the profiling phase and incorporates target-dependent constraints. Kremlin uses the following equation for calculating the estimated parallel-execution time of a region R ($ET(R)$):

$$ET(R) = \begin{cases} \dfrac{\sum_{k=1}^{n} ET(child(R,k))}{\min(SP(R), A(R))} + O(R) & \text{nonleaf} \\[2ex] \dfrac{work(R)}{\min(SP(R), A(R))} + O(R) & \text{leaf} \end{cases}$$

For leaf regions, the serial-execution time (*work*) is profiled in the profiling phase. After parallelization, the speedup is limited by either self-parallelism ($SP(R)$) or the allocated core count ($A(R)$) specified in a parallelization plan. The parallelization overhead ($O(R)$) is also added to the execution time to account for synchronization and scheduling costs. The parallelization overhead term favors coarse-grained parallelization over fine-grained parallelization. The parallel-execution time of a nonleaf region is similarly calculated, but its serial-execution time is propagated from its children's parallel-execution time ($ET(child)$). The estimated execution time of the root region is the whole program's parallel-execution time. The target-dependent time estimation model can be easily extended. For example, we integrated a memory model for the target platform to incorporate superlinear speedups due to cache effects.[9]

Finding the best parallelization plan

Once a time estimation model is in place, the next step is to use the estimation model to traverse the search space of possible parallel implementations for the program in an efficient manner, picking the best candidate. The parallelization system's constraints and capabilities play a significant role in pruning this space to parallelization plans that can actually be implemented. See our earlier work for more details.[8,9]

Evaluating Kremlin

Our evaluation consists of two main components: evaluating the accuracy with which Kremlin estimates parallel speedup, and determining the efficiency with which Kremlin selects which regions to parallelize.

Parallel-speedup estimation

We first evaluated Kremlin's speedup estimation on an increasing number of cores, modeling conventional multicore processors. In the evaluation, we compared Kremlin's predicted speedups against "real" speedup for OpenMP implementations. We selected programs from SpecInt2000 and NPB.[10] The former benchmarks contain little parallelism, whereas the latter tend to have higher levels of parallelism. The real results were gathered from measurements (ep, cg, bt) or published results of hypothetical parallelization systems (gzip, twolf, vpr).

Figure 5 shows that Kremlin provided tight speedup upper bound estimates across all benchmarks. For low-parallelism benchmarks, these results match the general intuition that those benchmarks lack sufficient levels of parallelism to warrant intense parallelization. We also targeted a different target platform (the MIT Raw processor) to see Kremlin's flexibility in speedup prediction; the results for this platform had a similar accuracy level.[9]

Parallelization planning

Next we evaluated how well Kremlin's parallelism planner worked in conjunction

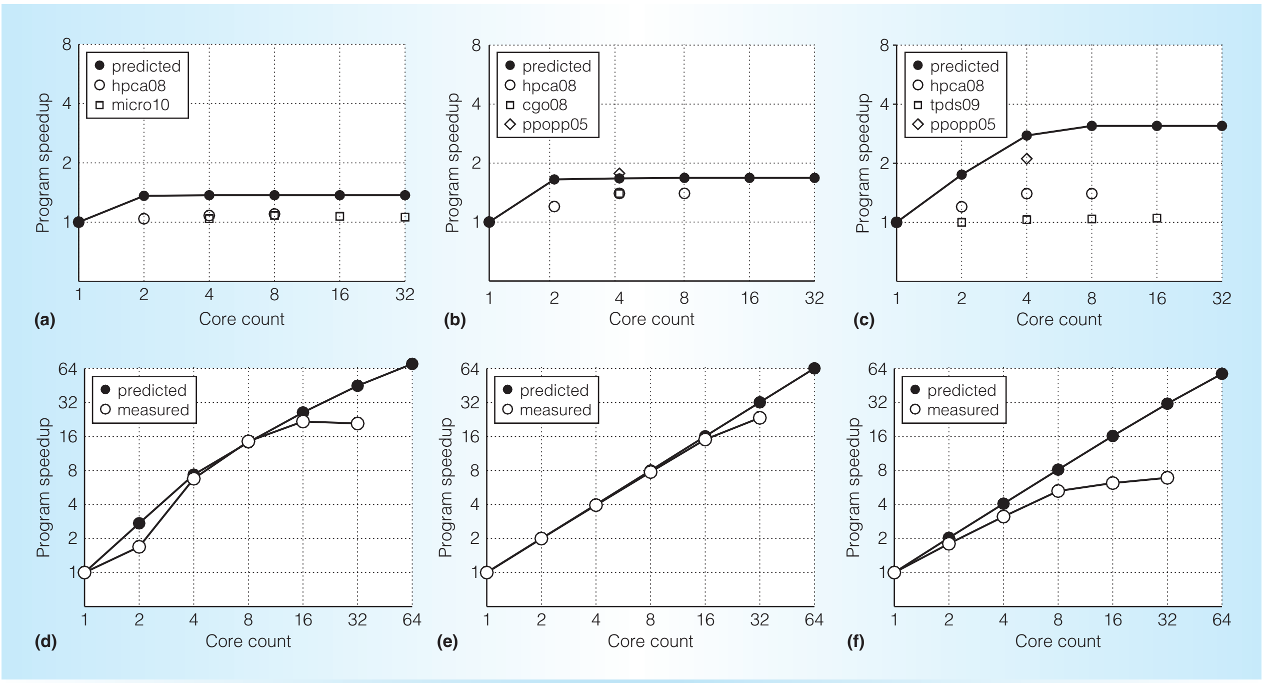


Figure 5. Predicted and reported speedup in low-parallelism SpecInt2000 and high-parallelism NAS Parallel Benchmarks (NPB): gzip (a), twolf (b), vpr (c), cg (d), ep (e), and bt (f). Kremlin automatically provides tight speedup upper bound estimates for serial programs, helping programmers set realistic parallelization goals. SpecInt2000 speedup numbers are from multiple sources that applied aggressive hardware and software techniques to extract parallelism from these benchmarks.[9] NPB speedup numbers are measured on 32-core AMD systems, using a third-party OpenMP-parallelized version.[11]

with the HCPA analysis. We examined three key metrics:

- *Did Kremlin suggest a reasonable number of regions to parallelize compared to third-party experts who had to manually plan parallelization?* Our results in Figure 6 show that Kremlin greatly reduced the number of regions that needed to be parallelized. Overall, the drop was 1.57×, but was more pronounced on benchmarks that required the most parallelized regions.
- *Did the regions that Kremlin excluded but that were parallelized by the third party contain significant speedup opportunities?* Performance in about 20 percent of cases was much better (1.8× and 1.4×) than the third-party version. This resulted from parallelization opportunities that Kremlin found but that the third party didn't implement. Excluding these cases, the performance came within an average of 4 percent of the third-party performance. Taking the time saved by Kremlin's succinct plan and applying it to serial optimization could easily eliminate this gap.
- *Did Kremlin order the regions well? In other words, were regions with the highest potential speedup recommended before less profitable regions?* Kremlin's ordering proved highly effective. Only 13.6 percent of the benefit came from the last 50 percent of regions, with only 4.4 percent of speedup from the final quartile. These results indicate that Kremlin does well at prioritizing the most important regions.

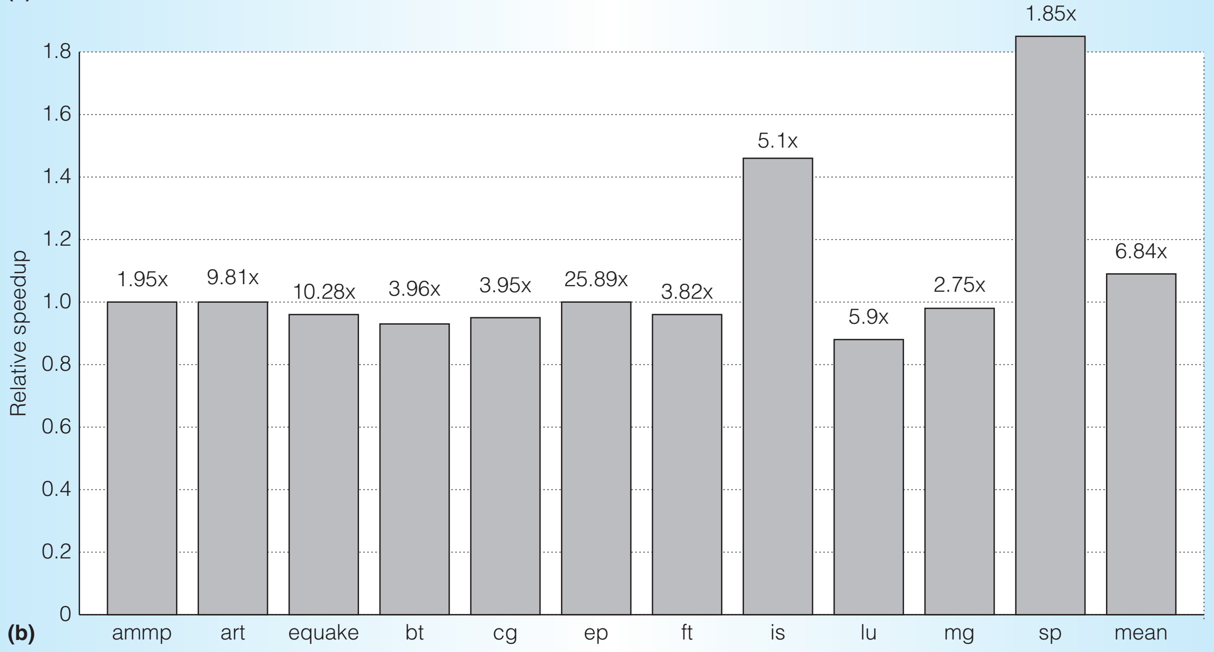
Our evaluation indicates that Kremlin is well suited as a parallelization oracle. Kremlin produces succinct parallelization plans that help the user quickly reach high levels of parallel performance.

GPU support: Early results

GPU-based computing, programmed by CUDA or OpenCL, is rapidly becoming a workhorse of scientific computing because

Benchmark	Manual	Kremlin	Overlap	Reduction
ammp	6	3	2	2.00×
art	3	4	1	0.75×
equake	10	6	6	1.67×
bt	54	27	27	2.00×
cg	22	9	9	2.44×
ep	1	1	1	1.00×
ft	6	6	5	1.00×
is	1	1	0	1.00×
lu	28	11	11	2.55×
mg	10	8	7	1.25×
sp	70	58	47	1.21×
Overall	211	134	116	1.57×

(a)



(b)

Figure 6. Evaluation of parallelization based on the Kremlin oracle: plan size comparison (a); relative speedup of Kremlin compared to third party with absolute speedup (b). The table shows that plans created by third-party experts for OpenMP without the use of Kremlin are significantly larger (1.57× on average). The majority of regions in Kremlin plans are found in the third-party plans. Even though Kremlin proposes fewer regions to the user, (b) demonstrates that the resulting performance is very good, ranging from 12 percent slower to 85 percent faster. This is surprising, because in contrast to the third-party versions, Kremlin doesn't have the luxury of doing trial-and-error measurements of various intermediate parallelized versions.

of the high computational density that these systems offer.

We recently prototyped support into Kremlin for OpenCL-based GPUs. Table 1 shows our initial results on the Rodinia benchmark suite, a set of applications meant to test heterogeneous computing environments. Kremlin's recommendations exactly matched those of the third-party version on eight of the 12 (66 percent) analyzed benchmarks. In two cases (cfd and srad), Kremlin found regions of low coverage that had been turned into OpenCL, potentially leading to wasted programming effort. In another benchmark (nw), Kremlin identified a missed opportunity for exploiting data-level parallelism on a high-coverage region. Follow-on research will parallelize this

Table 1. Kremlin GPU results. Preliminary results show that Kremlin is effective at selecting parallelization regions for code in the OpenCL-based parallelization of the Rodinia benchmark suite.

Benchmark	OpenCL kernels (Manual)	Kremlin recommended	Overlap (%)
backprop	2	2	100
bfs	2	2	100
cfd	4	3	75
heartwall	1	1	100
hotspot	2	2	100
kmeans	2	2	0
lavaMD	1	1	100
nn	1	1	100
nw	2	3	66
pathfinder	1	1	100
srad	5	3	60
streamcluster	1	1	100
Total	24	22	79

benchmark according to Kremlin's recommendations to establish whether they were correct for regions that did not match.

Kremlin strives to quantify the benefits of parallelizing sequential code, and provides a step-by-step list of regions for programmers to focus their time on. At the heart of our approach is the creation of a practical oracle that combines HCPA and a planning algorithm to estimate the outcome of a programmer's parallelization efforts. As we move forward with releasing Kremlin as an open-source tool (http://kremlin.ucsd.edu), we are perhaps a bit closer to the dream of "easy" parallelization. MICRO

Acknowledgments

This research was funded in part by the US National Science Foundation (NSF) under CAREER Award 0846152; by NSF Awards 0725357, 0846152, and 1018850; and by a gift from Advanced Micro Devices.

References

1. W. Blume et al., ''Parallel Programming with Polaris,'' *Computer,* Aug. 2002, pp. 78-82.
2. M. Hall et al., ''Maximizing Multiprocessor Performance with the SUIF Compiler,'' *Computer,* Aug. 1996, pp. 84-89.
3. W. Lee et al., ''Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine,'' *Proc. 8th Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* ACM, 1998, pp. 46-57.
4. S. Che et al., ''Rodinia: A Benchmark Suite for Heterogeneous Computing,'' *Proc. IEEE Int'l Symp. Workload Characterization* (IISWC 09), IEEE CS, 2009, pp. 44-54.
5. G. Tournavitis et al., ''Towards a Holistic Approach to Auto-Parallelization: Integrating Profile-Driven Parallelism Detection and Machine-Learning Based Mapping,'' *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation* (PLDI 09), ACM, 2009, pp. 177-187.
6. M. Frigo, C.E. Leiserson, and K.H. Randall, ''The Implementation of the Cilk-5 Multithreaded Language,'' *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation* (PLDI 98), ACM, 1998, pp. 212-223.
7. M. Kumar, ''Measuring Parallelism in Computation-Intensive Scientific/Engineering Applications,'' *IEEE Trans. Computers,* Sept. 1988, pp. 1088-1098.
8. S. Garcia et al., ''Kremlin: Rethinking and Rebooting Gprof for the Multicore Age,'' *Proc. 32nd ACM SIGPLAN Conf. Programming Language Design and Implementation* (PLDI 11), ACM, 2011, pp. 458-469.
9. D. Jeon et al., ''Kismet: Parallel Speedup Estimates for Serial Programs,'' *Proc. ACM Int'l Conf. Object-Oriented Programming Systems Languages and Applications* (OOPSLA 11), ACM, 2011, pp. 519-536.
10. D.H. Bailey et al., ''The NAS Parallel Benchmarks—Summary and Preliminary Results,'' *Proc. 1991 ACM/IEEE Conf. Supercomputing,* ACM, 1991, pp. 158-165.
11. ''NAS Parallel Benchmarks 2.3; OpenMP C''; www.hpcc.jp/Omni.

Saturnino Garcia is a visiting assistant professor in the Department of Mathematics and Computer Science at the University of San Diego. His research interests include program analysis, parallel software engineering, and computer science education. Garcia has a PhD in computer science from the University of California, San Diego.

Donghwan Jeon is a software engineer at Google. He completed the work described in

this article while pursuing graduate studies at the University of California, San Diego. His research interests include software engineering tools for parallelization, dynamic program analysis, and parallel software systems. Jeon has a PhD from the University of California, San Diego.

Christopher Louie is a server and systems engineer at Gazillion Entertainment. He completed the work described in this article while pursuing graduate studies at the University of California, San Diego. His research interests include tools for software engineering and parallel systems. Louie has an MS in computer science from the University of California, San Diego.

Michael Bedford Taylor is a professor in the Department of Computer Science and Engineering at the University of California, San Diego, and he jointly leads the Kremlin and GreenDroid projects. His research interests include dark silicon, multicore processor design, and software tools for leveraging parallelism. Taylor has a PhD in electrical engineering and computer science from the Massachusetts Institute of Technology.

Direct questions and comments about this article to Michael Bedford Taylor, EBU 3B 3110 MC 0404, 9500 Gliman Dr., La Jolla, CA 92093-0404; mbtaylor@ucsd.edu.