Kremlin: Rethinking and Rebooting gprof for the Multicore Age

Saturnino Garcia, Donghwan Jeon, Chris Louie, and Michael Bedford Taylor

Department of Computer Science & Engineering
University of California, San Diego
La Jolla, CA, USA
{sat,djeon,cmlouie,mbtaylor}@cs.ucsd.edu

Abstract

Many recent parallelization tools lower the barrier for parallelizing a program, but overlook one of the first questions that a programmer needs to answer: *which parts of the program should I spend time parallelizing?*

This paper examines Kremlin, an automatic tool that, given a serial version of a program, will make recommendations to the user as to what regions (e.g. loops or functions) of the program to attack first. Kremlin introduces a novel *hierarchical critical path analysis* and develops a new metric for estimating the potential of parallelizing a region: *self-parallelism*. We further introduce the concept of a *parallelism planner*, which provides a ranked order of specific regions to the programmer that are likely to have the largest performance impact when parallelized. Kremlin supports multiple *planner personalities*, which allow the planner to more effectively target a particular programming environment or class of machine.

We demonstrate the effectiveness of one such personality, an OpenMP planner, by comparing versions of programs that are parallelized according to Kremlin's plan against third-party manually parallelized versions. The results show that Kremlin's OpenMP planner is highly effective, producing plans whose performance is typically comparable to, and sometimes much better than, manual parallelization. At the same time, these plans would require that the user parallelize significantly fewer regions of the program.

Categories and Subject Descriptors D.2.2 [*Software Engineering*]: Design Tools and Techniques; D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming

General Terms Measurement, Performance

Keywords Hierarchical Critical Path Analysis, Self-Parallelism, Parallelism Planner, Parallel Software Engineering

1. Introduction

The emergence of multicore processors has profoundly impacted the way in which future software performance gains will be achieved. In order to take advantage of the resources available in multicore processors, existing sequential software will need to be refactored for parallel execution.

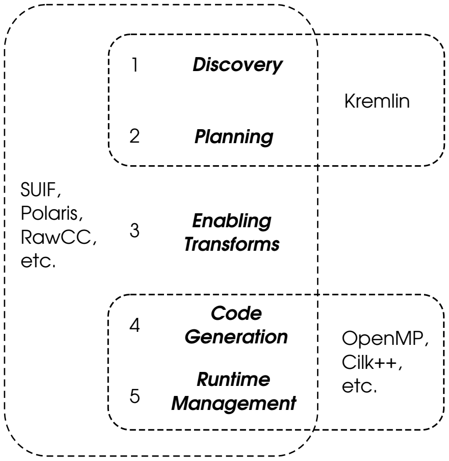


Figure 1. A Taxonomy of Parallelization Tools. The taxonomy categorizes parallelization tools based on which of five fundamental parallelization stages they assist with. Automatically parallelizing compilers like Polaris [7] and SUIF [12] attempt to perform all five without programmer assistance, while tools like OpenMP, Cilk++ [27], and X10 [33] focus on the last two. The paper's tool, Kremlin, targets the first two stages.

The task of parallelizing software has proved to be extremely difficult. To address this difficulty, recent mainstream tools have largely emphasized the ease of expressing parallelism; language extensions such as OpenMP and Cilk++ [27] have greatly reduced the effort involved in expressing opportunities for parallel execution in a program. Other tools guide the programmer in debugging performance or correctness problems in their parallelized code [4, 14, 35]. While these tools increase the efficiency of parallel programmers, there is still a need to assist the earlier stages of parallelization, where programmers must decide how to transform a serial program. The programmer requires new tools for these earlier stages of parallelization that will complement existing tools and form a complete toolflow for parallelization.

This paper examines Kremlin, a new tool that, given a serial version of a program, answers the question: *which parts of the program should I parallelize first?*

This question is not altogether unlike the question that `gprof` attempts to answer for serial programs: which parts of the program should I try to optimize first? In this paper, we rethink the design of a `gprof`-like tool that applies to the parallelization of serial pro-

PLDI'11, June 4–8, 2011, San Jose, California, USA.
Copyright © 2011 ACM 978-1-4503-0663-8/11/06. . . $10.00

grams rather than their serial optimization. Although the motivation, goals, and user interface are quite similar to gprof, we find that pursuing parallelism requires a new approach that is quite different, and leads to a tool whose underlying architecture is a total *reboot*, as it were, of the underlying architecture of a serial performance profiler like gprof.

A Taxonomy of Parallelization Stages To explain Kremlin's role in the spectrum of parallelization tools, we introduce a taxonomy that describes the basic stages involved in parallelizing a program. Figure 1 shows this taxonomy. Parallelization begins with *Parallelism Discovery*, which is the process of identifying regions of a program that have exploitable parallelism. Locating these regions of the program is especially onerous for large, complex programs or when—as is often the case—the parallel programmer is not the original author of the program. The next stage, *Parallelism Planning*, determines which subset of these regions should be parallelized. Ideally, this plan would factor in important constraints such as the number of cores available and the system's ability to support different types of parallelism. The third step is *Enabling Transforms*. These are source-level transformations that the user performs in order to enhance the parallelizability of the code used in subsequent stages. The parallelization process concludes with the final two steps of *Parallel Code Generation*, where parallel or threaded code is generated, and *Runtime Management*, where a runtime system tries to optimize the efficiency of parallel execution.

As shown in Figure 1, automatic parallelizing compilers such as Polaris [7], RawCC [26], and SUIF [12] seek to automate all of the stages in the parallelization taxonomy. Although they eliminate the need for manual intervention, the performance of code generated by these compilers often pales in comparison with code generated manually. This lack of performance stems from the difficulty of many of these steps; in many cases, the compiler cannot effectively perform them without the benefit of runtime information or without otherwise unsound semantic changes by the user. While additional information could potentially improve performance, accomplishing this integration has proven a difficult problem. Without this additional input, parallelizing compilers have an all-or-nothing feel: the compiler either does all of the work or none of it.

An alternative to the fully-automated approach is to separate the parallelization process into the stages shown in Figure 1 and make use of parallel programming tools to automate as much of the work as possible. These tools would relieve most of the burden on the programmer and allow them to focus on the Enabling Transforms stage where automated tools are most limited[1]. Recent tools such as OpenMP, Cilk++ [27], X10 [33], and Fast Track [17] exemplify how user-centered tools can improve the productivity of parallel programmers. These tools provide extensions to standardized languages that allow a programmer to explicitly specify parallel regions and synchronization points, automatically handling the final two stages of parallelization. However, by the time a programmer can utilize these tools, they must have already performed the initial stages of parallelization without assistance.

Kremlin We have developed Kremlin to aid programmers in both the parallelism discovery and parallelism planning stages of parallelization. Kremlin's parallelism discovery phase utilizes critical path analysis [23], or *CPA*, to quantify the amount of innate parallelism that exists in a piece of code. Kremlin extends traditional critical path analysis by automatically incorporating the nested structure of the program's regions (e.g. loops and functions) into its analysis. Kremlin's *hierarchical critical path analysis*, or *HCPA*, performs critical path analysis across many nested regions in a program. By comparing the amount of parallelism that exists in each parent region to the amount that exists in its children, we are able to localize parallelism to specific regions of the program, revealing the amount of parallelism in each code region.

To accomplish this, Kremlin introduces a new metric, *self-parallelism*, which given a parent region, can factor out the parallelism of subregions, much like gprof's *self-time* metric factors out the time spent in subregions. Our results show that HCPA is highly effective in identifying those code regions that are prime candidates for parallelization.

In addition to its support for parallelism discovery, Kremlin performs parallelism planning by incorporating hierarchical data from parallelism discovery as well as additional constraints. The planner supports the concept of *planning personalities*, which incorporates the impact of parallelization system (e.g. OpenMP) and machine properties. As an example of one such personality, Kremlin includes an OpenMP planner which has been validated on a multicore system with 32 cores.

Our results, acquired by applying Kremlin to serial versions of the NAS Parallel Benchmark Suite [6], and all of the C-language Spec OMP2001 [2] benchmarks, show that our approach is quite effective. Kremlin's recommendations, which required no manual intervention, and relied only on information extracted from the serial version of the program, comprise only 3.0% of the original programs' region count. Comparing to a 3rd-party parallelized version, Kremlin required 1.57× fewer regions to be parallelized. In cases where the recommended parallelization plan was similar, Kremlin's performance averaged within 3.8% of manual parallelization, close enough that applying the saved time to serial optimization would likely produce a better outcome. In cases where Kremlin's plan was significantly different, Kremlin's plan exceeded the manual plan's performance by 85%.

To summarize, this paper makes the following contributions:

- We introduce the technique of hierarchical critical path analysis. Traditional critical path analysis techniques are unable to localize parallelism to specific code regions, which is a fundamental limitation that prevents its widespread use in practical end-programmer tools. Unlike dependence-testing approaches that focus on loop bodies, HCPA allows parallelism in serial program regions to be identified even if the program's current form (e.g. loop nesting structure) does not expose it, which can reveal opportunities where non-intuitive code restructuring can yield large benefits.
- We introduce a new metric, self-parallelism, which is the critical step in extending CPA to HCPA. Self-parallelism is able to quantify the parallelism of a parent region independent of its children. We show that high self-parallelism is well-correlated with achieving real parallel speedup. This metric is analogous to *self-time* in the gprof profiling tool.
- We develop the concept of a parallelism planner as a key step in the manual parallelization process; that is to say, because of the complexity of the task, we believe profilers for parallel programming should not only provide self-parallelism, work, and other information about program regions but also combine these factors with Amdahl's Law and target system properties to estimate which regions are worth pursuing.
- We introduce the concept of planning personalities that tailor the planning process based on the target system and language. We describe in detail one such personality, a planner for a multicore processor using OpenMP. Combined with parallelism planners, this is a practical way of bridging the gap between abstract CPA analysis and realistic constraints of today's execution environments.

[1] Nonetheless, some researchers have developed tools which help perform specific refactoring tasks in the Enabling Transforms stage [10, 40].

- We demonstrate Kremlin's OpenMP planner's ability to produce concise plans whose performance is typically comparable to—and sometimes far exceeds—manual parallelization that required many trial-and-error iterations and was able to take advantage of post-parallelization measurements. At the same time, these plans would require that the user parallelize significantly fewer regions of the program.

The rest of this paper is organized as follows. Section 2 motivates the need for improved parallelism discovery and planning tools, and formalizes these concepts. Section 3 presents a high-level overview of Kremlin. Sections 4 and 5 describe the implementation of the profiling and planning aspects of Kremlin. Section 6 demonstrates the capabilities of Kremlin in identifying parallelism as well as creating an effective parallelization plan. Related work is detailed in Section 7, and Section 8 concludes.

2. Examining Parallelism Discovery and Planning

In this section we will first motivate the need for tools that discover and plan for parallelism. We then discuss the challenge of quantifying parallelism in specific parts of a program. Finally, we define the problem of parallelism planning informally and then formally.

2.1 Motivation

Despite recent research into parallel programming tools, many programmers still rely on a relatively painful methodology that employs serial profiling tools such as `gprof` in order to direct their parallelization activities. The process starts with a serial hotspot list, which ranks regions by the amount of time spent inside them. This list effectively becomes the order that they examine the functions to improve their performance.

It is at this point that the process gets especially onerous. The programmer starts leafing through the code trying to puzzle through the dependencies in the code, and the granularity at which to try to exploit it. Since the programmer has no indication of whether a hotspot is parallelizable, they frequently give up before they are able to recognize subtle but large parallelism opportunities, or they spend excessive amounts of time fruitlessly modifying serial parts of the code. Alternatively, even if parallelism does exist, it may not be large enough to yield speedup, or when combined with lower coverage, the overall speedup may not justify the effort. Finally, interference between nested parallel regions may prevent speedup.

Our experience watching graduate students struggle to parallelize serial code led us to realize that the inability to quantify parallelism in these program regions was indirectly responsible for large amounts of wasted time. To our surprise, providing them with a tool that quantifies parallelism was not quite enough. Rather, parallelization also requires planning tools that help process this information and apply both parallel programming system and machine constraints. With the ability to positively identify the existence of parallelism, and also to prioritize regions, users can invest their time more productively, attacking the correct portions of the program.

2.2 Quantifying Parallelism

Amdahl's Law provides guidance to the programmer by defining the basic relationship between parallelizability and speedup. According to this law, two factors directly impact speedup: the percentage of time spent in a section of code and the amount of parallelism within that code. While `gprof` and similar tools offer ready-made solutions for determining the work coverage, quantifying parallelism requires additional tools and techniques.

Promise of Critical Path Analysis Approaches One promising approach for quantifying parallelism is to use a *critical path analysis* [23], or *CPA*. CPA is a dynamic analysis that finds the string of dependencies that forms a lower bound on the execution time (the critical path) of a piece of code. The critical path in turn creates an approximate upper bound on the parallelism available, with the ideal parallel implementation performing all non-critical operations in parallel with the critical path operations. The work and critical path define the average amount of parallelism available according to the equation $p = work/length_{cp}$. We refer to parallelism calculated with this equation as the *total-parallelism*.

```
for(i=win..rows-win) {
  for(j=win..cols-win) {
    currLambda = lambda[i][j];
    ...
    for(k=0..nFeatures) {
      if(features[2][k] < currLambda) {
        ...
        features[0][k] = j;
        features[1][k] = i;
        features[2][k] = currLambda;
      }
} } }
```

Figure 2. Localizing Parallelism. In this nested loop from the fillFeatures function in feature_tracking, only the innermost loop (over k) is parallel. Traditional CPA would erroneously report parallelism in the outer loops because they contain the innermost.

The basic premise behind Kremlin and other parallelism discovery tools that employ critical path analysis [5, 21–23] is to evaluate the application's potential for parallelization under relatively optimistic assumptions based on observation of the program's dynamic execution. Most parallelizing compilers, in contrast, must take relatively pessimistic views because they are responsible for guaranteeing correctness. For example, parallelizing compilers may not be able to prove that two pointers do not alias, while a critical path analysis will at least report that it did not observe such dependencies in the actual execution of the program. The basic idea is to elevate to the user awareness of the at least circumstantial evidence of parallelism in the program, so that users can apply their understanding of the real application constraints (as opposed to what is encoded in program source) and refactor to exploit the parallelism.

Critical Path Analysis's Inability to Localize Parallelism Unfortunately, traditional critical path analysis has not found widespread use as a parallelism quantification tool for parallel programmers because it has one important limitation: it cannot localize the parallelism to a particular level of the nested hierarchy of a program's regions. This limitation is illustrated by a code snippet from the feature_tracking benchmark from the San Diego Vision Benchmark Suite [20], shown in Figure 2. In this example, only the innermost loop is parallel. Traditional CPA would only detect that parallelism exists somewhere among the three loops, not just the innermost.

Localizing Parallelism to Specific Regions Kremlin avoids the aforementioned limitation through two novel contributions: *hierarchical critical path analysis* and a new metric, *self-parallelism*. Hierarchical critical path analysis calculates the amount of parallelism within every nested region in a program. Self-parallelism uses this hierarchical data to factor out the parallelism contributed by a region's subregions, much like `gprof`'s self-time metric factors out the time spent in a region's subregions. These two features combine to help Kremlin localize parallelism to specific regions of a program. Section 4 will define and explore them more formally.

What Is a Region? Kremlin uses the concept of a region to denote a region of code whose parallelism is to be measured from the time that region is entered until the time it is exited. In order for the self-parallelism metric to work, regions must obey a proper nesting structure: regions must not partially overlap, but they may nest or be siblings with the same parent region. Based on this nesting structure, we can define a dynamic region graph which shows the relationship between parent and children regions in the dynamic execution of the program.

Although more arbitrary delineations of regions are possible, Kremlin places regions around all loops and functions, since they tend to correspond well to constructs that users understand well and relate more directly to the process of parallelization.

2.3 Creating a Parallelism Plan

Informally, parallelism planning is the problem of producing a sequence of program regions for the programmer to parallelize, ordered according to their expected impact on program execution. While the self-parallelism and self-time metrics improve our ability to create an effective parallelism plan, factors outside of work coverage and parallelism will also impact planning. One such constraint is the risk of over-parallelizing the program: since parallel execution often incurs some overhead in terms of work and/or resource contention, expressing more parallelism than there are cores available to exploit can result in slowdown. For instance, we found that in OpenMP, on our experimental setup, it was seldom profitable to parallelize a child region of a region that had already been parallelized. Another constraint is that synchronization and data movement costs in the system often affect the smallest parallel region that can attain speedup.

We now formally define the problem of parallelism planning so that we may later solve it algorithmically. Let R_P be the set of parallelizable regions in a program P. A parallelism plan creates an ordering on this set such that the relation defining the ordering, $<$, is true for $A < B$ if A should be parallelized before B.

A set of constraints, C, will help to define the ordering on R_P. Examples of these constraints range from architecture-specific constraints (e.g. the number of cores available), to language-specific constraints (e.g. inability to express pipeline parallelism in OpenMP), and even human factors (e.g. the desire to achieve large speedups as soon as possible). These constraints are combined to form a personality for the planner. A planner personality may range from detailed (e.g. fine-grained parallelism on a 100-core Tilera machine) to general (e.g. coarse-grained parallelism), depending on the goal of the user; detailed planners will have better performance on targeted systems while broader personalities will have more robust performance across a broader range of machines.

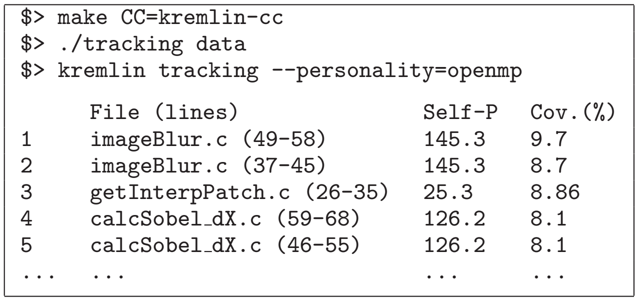
2.4 Limitations of Kremlin and Other Profile-Based Tools

Needless to say, Kremlin is naturally affected by the same limitations that apply to other profiling tools, like gprof, that make use of dynamic information that is input-dependent and does not necessarily predict the program's execution with other inputs. Our experience in varying inputs suggests that in many cases, the additional information gained by dynamic analysis outweighs the drawbacks. Further, Kremlin supports aggregation of data from multiple runs, which reduces these risks. Another limitation is that Kremlin cannot predict the enhanced parallelism that might be attained by changing algorithms, although a report of low parallelism may surprise the user into realizing they need a more parallel algorithm.

CPA-based tools can also be limited by unnecessary dependencies that are still true dependencies. Although no tool can handle all such cases, Kremlin handles many common cases by breaking such dependencies via induction and reduction variable analysis.

3. Kremlin Overview

In this section, we overview Kremlin's user interface and its high-level system architecture. As shown in Figures 3 and 4, Kremlin takes in unmodified source code and produces an instrumented binary. The user executes this binary as they would the uninstrumented program. This binary produces a parallelism profile that Kremlin's parallelism planner uses, along with the specified planner personality, to produce a parallelism plan.



```
$> make CC=kremlin-cc
$> ./tracking data
$> kremlin tracking --personality=openmp

     File (lines)                Self-P   Cov.(%)
1    imageBlur.c (49-58)         145.3    9.7
2    imageBlur.c (37-45)         145.3    8.7
3    getInterpPatch.c (26-35)    25.3     8.86
4    calcSobel_dX.c (59-68)      126.2    8.1
5    calcSobel_dX.c (46-55)      126.2    8.1
...  ...                         ...      ...
```

Figure 3. Kremlin's User Interface. After compiling and executing the program, Kremlin produces an ordered parallelism plan based on the selected planner personality. Regions are ordered by their estimated program speedup. In addition to location, Kremlin displays the average amount of region-specific parallelism (self-parallelism) and the execution coverage. Results shown are for the feature_tracking benchmark [20].

Figure 3 also shows the parallelism plan for the feature_tracking benchmark as it would be displayed to the user. The plan presents an ordered list of regions for the programmer to parallelize along with the self-parallelism and work coverage of those regions. Kremlin sorts the list according to decreasing whole-program potential speedup. The list contains only those regions that are expected to meet a minimum speedup threshold; the programmer can expect to obtain nearly all the performance benefits possible if they parallelize all the regions in the list.

Once the programmer has the plan, the basic usage model is that they visit these regions of code in the specified order and determine how to expose the underlying parallelism that was detected by Kremlin. In the event that the user is unable or unwilling to exploit the parallelism in a region, they can rerun the planner with a list of excluded regions and receive an updated plan. The results in this paper were generated without using the exclusion list feature.

Having explained the user's view of the tool, we now examine the Kremlin System Architecture, shown in Figure 4.

Static Instrumentation Kremlin's discovery components are split into two instrumentation steps: critical path instrumentation and region instrumentation. The first step helps quantify parallelism via critical path analysis while the second step helps uncover the program's structure and localize parallelism to specific regions.

Both stages of the discovery phase utilize LLVM's [25] static instrumentation infrastructure. Static instrumentation has two important benefits over dynamic instrumentation. First, it allows for a deeper analysis of the program since the full program source is available. In our experience, tasks such as identifying induction variables, reduction variables, and region boundaries are challenging in dynamic infrastructures such as Valgrind [30] but are easy when performed statically. Second, by statically inserting instrumentation, Kremlin can heavily optimize the code to produce a more efficient instrumented binary. This helps to lower the overhead associated with the heavyweight analysis infrastructure required for hierarchical critical path analysis. Kremlin performs this optimization after instrumentation occurs so that it does not taint the analysis.

During the critical path and region instrumentation stages, Kremlin inserts calls to instrumentation functions that calculate the critical paths of the program and track region entries and exits. These instrumentation functions are implemented inside the

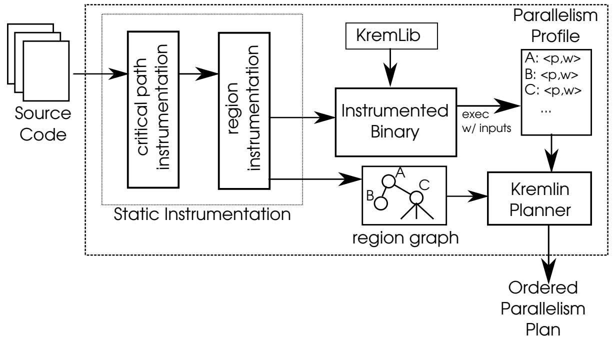


Figure 4. Overview of Kremlin System Architecture. Starting with a program's source code, Kremlin statically instruments the code to insert the proper profiling code and extract the region structure (i.e. region graph) from the program. Running the instrumented binary produces a parallelism profile for each of the program regions. Combined with the region graph, the parallelism profile is used by the parallelism planner to provide the user with a specific list of regions to parallelize (the parallelism plan).

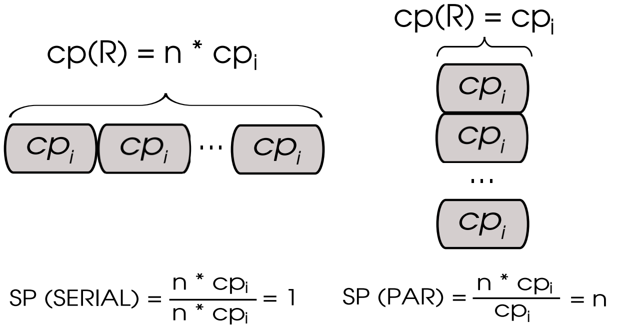

cp(R) = n * cp_i
cp_i
cp_i
...
cp_i
cp(R) = cp_i
cp_i
cp_i
...
cp_i
SP (SERIAL) = (n * cp_i) / (n * cp_i) = 1
SP (PAR) = (n * cp_i) / cp_i = n

Figure 5. Examples Demonstrating Self-Parallelism (SP) Metric. SP identifies the parallelism local to a region by relating its critical path to the sum of its children's critical paths and its self-work. Shown in the example are SP calculations for two regions; one whose children must execute serially, and one whose children can execute in parallel.

KremLib library. Section 4 provides more details on Kremlin's hierarchical critical path analysis.

Linking and Execution Kremlin next links in the KremLib instrumentation library to produce the instrumented binary. When run, the instrumented binary also produces a parallelism profile output file in addition to its normal outputs that contains parallelism information for each dynamic instance of a program region. Since regions in a program may be executed many times, Kremlin performs an online, dictionary-based compression to greatly reduce the size of the parallelism profile.

Kremlin Planner With the parallelism profile and region graph produced by the discovery phase, Kremlin can begin to create an effective plan to utilize the parallelism in the program. To begin, Kremlin combines the profile data with the region graph information to calculate, among other things, the self-parallelism and work metrics for regions in the program, as described in Section 4.3.

After calculating the self-parallelism for each region, Kremlin designs an ordered plan for the programmer that describes which regions should be parallelized. Kremlin uses both coverage and self-parallelism to estimate the speedup associated with parallelizing specific regions, using this speedup information to order the regions. Kremlin uses planning personalities that incorporate both target- (e.g. OpenMP) and machine-specific parameters in order to improve accuracy; detailed discussion resides in Section 5.

4. Parallelism Discovery

As we discussed earlier, traditional critical path analysis is poor at localizing parallelism to specific code regions. The following subsections describe how Kremlin performs an enhanced form of analysis, hierarchical critical path analysis, and how our new self-parallelism metric is calculated. They also describe a compression technique that reduces the resource requirements of the system.

4.1 Critical Path Analysis

This section describes aspects of hierarchical critical path analysis, including how Kremlin calculates critical path lengths using shadow memory and how it handles both false dependencies and control dependencies.

Calculating Critical Path Length With Shadow Memory Critical path analysis calculates parallelism by quantifying both the amount of work done and the minimum time needed to do that work (i.e. the length of the critical path). The ratio of work to critical path length indicates the average number of instructions that can be executed in parallel in the ideal case. Kremlin efficiently determines both of these values through the use of shadow memory [30, 45].

Kremlin's shadow memory infrastructure works by tracking the earliest possible time that a value stored in a memory location can be calculated. This "availability time" is a function of the longest string of operations required to calculate that value. Kremlin calculates the availability time of each operation by looking up the times of all instructions it depends upon, finding the maximum value, and then adding the operation's latency. Kremlin tracks true data and control dependencies but factors out false dependencies such as anti and output dependencies.

Kremlin utilizes two techniques to limit the overhead associated with shadow memory: dynamic allocation of shadow memory and shadow register tables for local variables. Dynamic shadow memory allocation is enabled by the use of a two-level table that equally splits the whole address space. Kremlin allocates table entries only when they are needed, utilizing calls to `malloc` and `free` as signals of the need to allocate or deallocate. Kremlin uses directly addressable shadow register tables for all local variables. This reduces access time for the common case of reading from local rather than global variables.

Resolving False and Easy-to-Break Dependencies A major challenge for any critical path analysis infrastructure is to mitigate the effects of false and easy-to-break dependencies. Many of these false dependencies, such as unnecessary reuse of a variable, are eliminated by the use of SSA form in LLVM's IR. However, the easy-to-break dependencies associated with induction and reduction variables are more challenging. These types of dependencies can create the false impression of seriality in an otherwise parallel region. Kremlin statically identifies these dependencies and breaks them by using a special shadow memory update rule that ignores the dependency on their old value.

Managing Control Dependencies Kremlin performs static control dependence analysis to identify which values a basic block is control dependent upon. Unfortunately, static analysis cannot fully resolve all control dependencies. Kremlin handles control dependencies through the use of a control dependence stack similar to one proposed in [42]. Kremlin pushes a dependency onto the control stack at the beginning of a control dependent region, popping it off when exiting that region. Availability times in the control stack can only increase. Therefore, Kremlin incorporates control dependencies by checking only the top of the stack.

4.2 Hierarchical Shadow Memory

Hierarchical analysis introduces a new requirement to the shadow memory infrastructure described earlier: we must run separate critical path analyses across each nested dynamic region, and thus must effectively maintain many versions of the shadow memory. When we enter a new region, for the purposes of that region, we start at $time = 0$, $work = 0$, ignoring any dependencies that the parent region is aware of and is tracking. As we execute instructions in the program, we update the dependency times in the shadow memory and the work count for each active region. When we exit a new region, we record a summery of a dynamic region. This summary contains the static region ID, the total work in the region, and the critical path length.

To implement shadow memory for the evolving set of dynamically nested regions as the program runs, each location in the shadow memory and register tables is associated with a set of availability times rather than just one. This set expands when a region is entered and shrinks when a region is exited.

Kremlin optimizes hierarchical shadow memory performance by maintaining a fixed-sized array of shadow data for every memory location. Kremlin assigns a region to a slot based on its depth in the region graph. For instance, the `main` function (the root in the graph) might occupy slot 0, while its children will use slot 1, and so on. A command line flag can vary the range of region depths that are collected, facilitating parallel data collection for the HCPA.

This assignment method leads to multiple regions at the same depth being mapped to the same slot—although they will never use it at the same time—and therefore Kremlin must provide a mechanism to avoid data reuse across region boundaries. Kremlin solves this problem by assigning a unique ID to every region instance and tagging every shadow memory write with the ID of the writer. Kremlin compares the tag with the ID of readers, discarding the data if there is a mismatch and assuming time 0 instead.

4.3 Self-Parallelism

Kremlin incorporates both the parallelism information and the program's structure to calculate a region's self-parallelism. Self-parallelism factors out the parallelism that comes from subregions to determine the amount of parallelism specific to a region.

Kremlin uses the following equation to determine $SP(R)$, the self-parallelism of a region R:

$$SP(R) = \frac{\sum_{k=1}^{n} cp(child(R,k)) + SW(R)}{cp(R)} \qquad (1)$$

where n is the number of children of R, $child(R,k)$ is the k^{th} child of R, and $cp()$ is the critical path length of the region. $SW(R)$ represents the amount of work that is performed exclusively in region R (i.e. self-work) and is calculated by the equation:

$$SW(R) = work(R) - \sum_{k=1}^{n} work(child(R,k)) \qquad (2)$$

Self-parallelism factors out the children's parallelism by summing the critical path lengths of the children rather than their work, thereby eliminating the parallelism that would otherwise come from the children. Self-parallelism adds work exclusive to the region to capture any remaining parallelism outside of the children.

The self-parallelism metric also allows Kremlin to estimate the speedup from parallelizing a region, R. If the execution time of a R is $ET(R)$, then the execution time after the region is parallelized will be bounded by $ET(R)/SP(R)$. This lower bound on execution time of a parallelized region is used by the planner when determining which regions will bring the largest speedup.

To illuminate the effectiveness of self-parallelism, we will examine the self-parallelism in two cases, shown in Figure 5: a region with n children that can execute in parallel, and a region where all n children must execute serially due to dependencies. For simplicity, we assume that parent regions have no self work and that all children have the same measured critical path length, cp_i.

For the parallel region, its measured critical path will be equal to a single child (i.e. $cp(R) = cp_i$). Thus, the computed self-parallelism, will be $\frac{n*cp_i}{cp_i} = n$; this is as expected because its parallelism is equal to the number of children. Now consider a parent whose child regions must be executed completely serially. In this case, the measured $cp(R)$ will be equal to $n * cp_i$ and therefore the computed self-parallelism will be $\frac{n*cp_{avg}}{n*cp_{avg}} = 1$; again, this is expected because it cannot overlap execution of the regions.

In addition to resolving simple cases like these, the SP heuristic also computes reasonable upper-bounds estimates on self-parallelism for cases where there is self-work in the parent node, and where there are dependencies between child regions that allow partial overlap in execution.

4.4 Compressed Representation

Performing whole-program shadow memory tracing and region analysis is nominally a compute- and memory- intensive endeavor. In order to manage both execution time and storage requirements, Kremlin employs a novel compressed trace representation that enables the planning algorithm to operate on the data without actually decompressing it (analogous to [36] but in a different domain). In this subsection, we describe our implementation and quantify its benefits.

Runtime Compression Technique Kremlin produces a parallelism profile for each dynamic region that is executed. The number of regions is based on the program structure – deeply nested loops can lead to a large number of regions – and the input to the instrumented program and can quickly grow to multiple gigabytes for some programs. Kremlin takes advantage of the fact that many of the region summaries are identical and therefore compression can be used. This reduces not only the amount of data that needs to be written to disk during the instrumented program's execution but also the amount of data that needs to be processed by the planner.

Kremlin utilizes a dictionary-based compression algorithm. When the flow of execution exits a region, Kremlin checks the tuple of ⟨`static region, critical path, work, children`⟩ for the region against the current alphabet of unique regions. If there is no match, a new character is added to the alphabet. If there is a match, the associated character is used. The alphabet necessarily starts with leaf regions. For `children`, Kremlin summarizes child regions via a sorted list of characters, representing compressed regions, and their frequencies. Thus, the children used in the tuple are defined in terms of the existing alphabet rather than the raw region info. Starting from the leaves, the alphabet expands to regions that contain only leaves for children and so on to the root (i.e. `main`).

When the Kremlin planner makes use of the trace files to calculate self-parallelism, it does not need to decompress the data; instead, it operates on each character in the dictionary's alphabet directly. Kremlin exploits the dictionary representation's ability to summarize the critical path and work across recursively nested regions. This can accelerate the calculation of self-parallelism because each dictionary entry essentially summarizes repetitive sequences of large numbers of dynamic regions; processing each character therefore corresponds to processing thousands of dynamic regions.

Benefits We have found that this approach is quite effective. Our original log sizes for the NPB benchmarks [6], using the W inputs, ranged from 750 MB to 54 GB, and averaged 17.9 GB. After compression, they were reduced to between 5 KB and 774 KB, with an average of 150 KB. This was a net average reduction of ~119,000x for log size, and a typically proportional factor for planning time – from minutes to small fractions of a second. Code instrumented with our infrastructure, which is not heavily optimized, is currently about 50× slower than `gprof`-instrumented code; however the parallelism discovery process can be sped up by running instrumentation of disjoint code regions in parallel.

5. Parallelism Planning

In this section we will discuss Kremlin's OpenMP planning personality, and briefly overview a Cilk++ planning personality.

5.1 OpenMP Planner

OpenMP is a popular parallel programming environment with a strong focus on parallelization of loops. Programmers insert pragma statements into their source code and the OpenMP compiler generates the necessary threaded code for them to run. While it does support nested parallelism, the overhead is often too high for it to be effective: the number of execution contexts available is often not enough to handle the extra threads that are spawned and thus the cost of spawning new threads is never amortized. Furthermore, OpenMP requires the programmer to transform loops into parallel (i.e. DOALL) loops in order to achieve good performance.

Kremlin contains a planner that takes into account the major constraints associated with OpenMP. The planner disallows nested parallel regions to avoid the performance penalty we observed on our experimental setup. OpenMP supports reduction variables in parallel loops, but they have significant overheads [8]. We found that the amount of work in a region should be large enough to amortize these costs. For instance, reduction-based loops in the SPEC OMP2001 [2] benchmarks art and ammp have too little work to overcome overheads. On the other hand, ep, from the NAS Parallel Benchmarks [6] ("NPB"), has a reduction-based main function that should be parallelized because it has ample work.

Based on these constraints for OpenMP planning, we can formulate the problem as follows. Given a region graph G, select a set of regions to parallelize, R, such that in any multi-node path, P, in G there is at most one node in R (i.e. $|P \cap R| \leq 1$). The optimal solution will minimize the time required to execute the program.

OpenMP Planning Algorithm A naive algorithm for determining which regions to parallelize would be to repeatedly select the region with the largest potential speedup among all regions considered for parallelization. When a region is selected by the planner, any region that can reach or can be reached from a selected region would then be excluded from consideration to avoid nested parallelization. In some cases, this algorithm may lead to optimal results but in many cases it is suboptimal. For example, a parent region might have the highest single potential speedup, but collectively, a set of its child regions could offer a higher combined speedup. A greedy algorithm would select the parent, precluding the more optimal solution of selecting the set of child regions. Specifically, this problem was observed in two of the NPB benchmarks: ft and lu.

Kremlin's OpenMP planner employs a bottom-up dynamic-programming algorithm. At each level, if we have an optimal plan for parallelizing all of the children, we can make an optimal plan for parallelizing the current node by comparing the expected speedup of the parent versus the cumulative expected speedup of parallelizing the children according to their optimal plan. At the leaves, we select the node by default if it has expected positive speedup.

Additional Constraints for OpenMP and Machine Overheads As part of Kremlin's support for planner personalities, Kremlin can be configured with a few parameters that attempt to capture the system's ability to exploit parallelism. In shared memory and large-scale NUMA machines using runtime schedulers, the cost of data movement and synchronization can be relatively high compared to finer-grained research machines, which impairs their ability to exploit regions with fine-grained parallelism or synchronization.

Clearly many parameters could be incorporated to try to more accurately model the underlying system. However, we would generally prefer to have relatively simple parameters that are useful across a range of systems. As a result, we strove for parameters that are expressed in a relatively architecture-independent fashion.

For the OpenMP planner personality, we found that it was effective to use a set of three threshold parameters. The first is a threshold that determines a minimum level of self-parallelism for a region to be exploited. This mechanism indirectly accounts for the overhead of the the OpenMP scheduler and also the cost of migration between nodes. The second and third thresholds are the minimum required ideal whole-program speedup due to a DOALL region and due to a DOACROSS region. We differentiate between these two kinds of regions because DOACROSS regions are much more synchronization intense (and thus less likely to pay off), and require more programmer effort, and so require a higher ideal whole-program speedup to be justified. As discussed in Section 4.3, Kremlin identifies DOALL loops by checking for equivalence between self-parallelism and iteration count. In this paper, we employ 5.0 as the cutoff for self-parallelism, 0.1% speedup for DOALL loops and 3% speedup for DOACROSS loops. Our sensitivity analysis suggests that Kremlin is not particularly sensitive to minor variations in the settings of these parameters.

Our initial prototype of the OpenMP planner also included the number of cores as a constraint. This initial version capped the speedup of any region at the number of cores available. However, we found that including this constraint had a negative impact on plan quality. With this cap in place, the planner could not differentiate between regions with self-parallelism of N (the number of cores) and those which much higher self-parallelism. Our results suggest that high self-parallelism is correlated with large speedup: a region with much higher self-parallelism likely has more opportunities to amortize parallelization overhead and more flexibility in restructuring to improve memory locality.

5.2 Cilk++ Planning Personality

Kremlin also contains a Cilk++ [27] planner that accounts for Cilk++'s support for nested- and finer-grained parallelism. It employs the same self-parallelism metric as the OpenMP planner but with lower thresholds for self-parallelism and ideal speedup, accounting for Cilk++'s lower underlying costs; it also uses a nesting-aware planning algorithm.

Kremlin was originally developed with a Cilk++ planner, which was used by a group of graduate students in a recent parallel architecture class. Although students found the tool useful, the relative lack of a large, established Cilk++ benchmark suite (and Intel's acquisition of CilkArts) barred us from performing a quantitative evaluation for publication, motivating the development of the OpenMP planner.

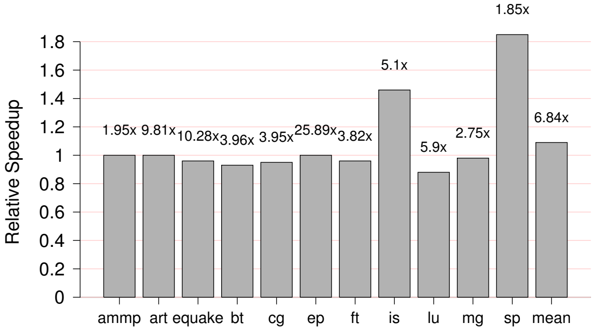
5.3 Developing Planning Personalities

Planning personalities provide an avenue for the user to tailor planning recommendations to different systems. Underlying the development of new planning personalities is a fundamental tension between accuracy and portability. The designer of a planning personality must decide the level of architectural independence that is part of the personality. Architectural independence is a desirable property for portability, allowing the planning results to be useful over a wide range of systems, but may need to be sacrificed in order to attain sufficient accuracy.

The development of the OpenMP and Cilk++ personalities provided some insight into the portability-accuracy trade-off. These

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

(a) Plan Size Comparison



(b) Relative Speedup of Kremlin Compared to MANUAL with Absolute Speedup

Figure 6. Evaluation of Kremlin-Based Parallelization. Table (a) shows MANUAL plan sizes are significantly larger (1.57× on average) than Kremlin plan sizes. Surprisingly, the majority of regions in Kremlin plans are overlapping with MANUAL. Even though Kremlin proposes substantially smaller number of regions to a user, (b) demonstrates that the resulting performance is generally quite close to the manually-parallelized versions, ranging from 12% slower to 85% faster. Note that Kremlin formulated its plans solely by examining the execution of the unmodified serial code. In order to reduce the experimental effects of different effort levels and different programmers for hand tuning, we evaluated the plans for Kremlin by using the parallelized code regions in the manually-parallelized version. In the case of SP and IS, Kremlin's recommendations were significantly different, so we had to manually apply those optimizations.

personalities required that we model only fundamental parameters of the parallel machines: synchronization costs, loop type, and region granularity. These parameters are likely to port well to other parallelization systems, reducing the work necessary to develop new planners for these other systems. We found that while machine-specific parameters such as cache size, page size, and memory bandwidth do influence parallel performance, and influence how code should be transformed, they have limited impact on the set of regions that should be parallelized. These machine-specific parameters therefore are of greater import during the *Enabling Transform* stage in Figure 1 than during the *Planning* stage.

6. Experimental Evaluation

We evaluated Kremlin using all 8 programs in the NAS Parallel Benchmarks (NPB) [6] and all 3 C-language programs in the SPEC OMP2001 [2] benchmark suite. For NPB, we used the third-party OpenMP manually-parallelized version of these programs [1] as a point of comparison for Kremlin's ability to create an effective parallelization plan. For SPEC OMP2001, we ran our tool on the corresponding serial versions of the programs in the SPEC 2000 benchmark suite, and then compared Kremlin's plans against those parallelized by humans in the SPEC OMP2001 versions. For art and ammp, SPEC OMP versions benefit from serial optimizations compared to their SPEC 2000 counterparts [38]. To exclude the effect of serial optimizations, we applied those optimizations on the SPEC 2000 code before running Kremlin. Our evaluation included only third-party benchmarks that have preexisting parallel versions to facilitate comparison and to make our results more credible. The programs vary greatly in terms of speedup (1.5x to 25.89x, Figure 6(b)), but low coverage, low parallelism, parallelization overhead, and other factors significantly reduce the percentage of regions that are good candidates for parallelization (Figure 9).

One might expect that iterative, trial-and-error manual parallelization would do significantly better than Kremlin, because the user has the benefit of performing iterative runtime measurements as they incrementally parallelize the program. We found that parallelization with Kremlin came surprisingly close in terms of performance on all but two benchmarks, and in those cases, it did much better. At the same time, it achieved these results with substantially smaller numbers of regions that needed to be parallelized.

6.1 Methodology

We first ran Kremlin on the unmodified, serial versions of the benchmarks to generate a parallelism plan for each program. The resulting plan was used to create a parallelized version of the serial program. In cases where Kremlin's parallelism plans recommended regions that had also been parallelized in the third-party, manually-parallelized version of the benchmark ("MANUAL"), we reused the parallelized regions from the MANUAL version. This allowed us to control for variances in performance that could result from slightly different parallel implementations of the same region.

To generate and evaluate parallelism plans, the 'W' input set was used for NPB benchmarks while the 'train' input was used for SPEC OMP. Kremlin relies on dynamic analysis and therefore may be affected by varying inputs. To test for input-related sensitivities, we reused the parallelized program based on the 'train' input parallelism plan to measure the speedup numbers for SPEC OMP benchmarks with the larger 'ref' input. We found that Kremlin-based parallelization remained equally competitive on both input sizes, despite requiring a much smaller set of parallelized regions.

Program performance was tested on 32-core system (8 × AMD 8380 Quad-core processors) with 256GB of memory running on the Linux 2.6.18 Kernel. Programs were compiled with gcc version 4.1 with OpenMP and -O3 flags specified. We executed the programs using configurations of 1, 2, 4, 8, 16, and 32 cores. As is typical for these kinds of systems, performance can decline as locality effects start to trump the benefits due to parallelization. For each parallel version, we determined the configuration with the best performance and report that number.

6.2 Evaluation of Parallelism Planning

Kinds of Parallelism Found and Transformations Implied Kremlin detects parallelism of all forms: nested-loop-parallelism including DOACROSS and DOALL, pipelined parallelism between loops and functions, ILP, and thread and task-level-parallelism. All of these exist in our benchmark suite, although Kremlin only recommends regions that are predicted profitable to parallelize according to the planner and personality in use.

Exposing the parallelism detected by Kremlin required user transformations such as: privatization; loop restructuring, fusion and interchange; insertion of OpenMP constructs; and refactoring of code and data structures to eliminate false sharing and contention. These transformations range in difficulty from trivial to difficult, that is, from requiring less than an hour of work to requiring several hours. As we will show in the following sections, Kremlin is able to significantly reduce the number of regions that must be parallelized, thereby significantly reducing the total effort needed to parallelize the program. Kremlin also provides a mechanism whereby the user can specify a set of regions that are too difficult to parallelize and rerun the planner, which recomputes the optimal plan excluding that region.

Plan size comparison Kremlin seeks to focus the programmer's efforts on a small subset of regions that have the most potential for speedup from parallelization. To test Kremlin's effectiveness in this regard, we compared Kremlin's recommended regions ("plans") to the set of regions that were parallelized in the third party-parallelized version of the benchmark suite, referred to as MANUAL. Figure 6(a) provides this plan size comparison.

Across all of the regions in the benchmarks, the MANUAL version included 1.57× more regions than the plan provided by Kremlin. For small benchmarks (e.g. ep, ft, and is) there was little room for improvement, but larger, more complex benchmarks showed larger savings compared to the average. At the extreme end, lu's manually-parallelized plan size was 2.55× the size of Kremlin. The programmer using Kremlin would have had far fewer regions to parallelize than the original third party parallelizers[2].

Performance comparison with MANUAL Next, we evaluated the speedup of parallelized versions based on Kremlin's parallelism plan against the MANUAL version. Figure 6(b) shows the results of this comparison. The Kremlin version of sp and is performed significantly better (1.85×, 1.46×) than MANUAL as Kremlin was able to identify parallelism that was missed in the MANUAL version. In this case, Kremlin recommended a coarse-grained parallelization, requiring privatization and refactoring. Other benchmarks saw a slight degradation in performance, averaging about 3.8%. Kremlin generally selected the same regions as MANUAL, but decided to stop earlier because of the diminishing returns.

To gain additional insight, Figure 7 shows the marginal benefit attained by applying each of the recommendations, in order, from Kremlin's plans. Also shown in the graphs are the marginal benefits of regions parallelized in MANUAL but *not* recommended by Kremlin (regions to the right of the dotted line).

In a large majority of cases, regions not recommended by Kremlin but parallelized by MANUAL provide negligible benefit. Additionally, we can see that, although Kremlin's plans are well-prioritized overall, the incremental contribution of a parallelizing a region can be somewhat noisy. For instance, in several cases, the second recommended region attains a much higher incremental speedup than the first recommended region—this is because as more of the program is parallelized, less data migration happens in the NUMA machine. Often it is groups of regions that must be parallelized before any speedup is observed.

Overall, Kremlin does an excellent job of eliminating regions that offer little benefit. Even for those few regions that were eliminated by Kremlin but had some marginal benefit, the benefits are slight. Given the savings in the number of regions parallelized by Kremlin, we suspect that the programmer could easily make up the difference by applying serial optimizations rather than attempting to parallelize the additional regions.

	Fraction of Kremlin Plan Applied			
Benchmark	**First 25%**	**First 50%**	**First 75%**	**All 100%**
ammp	74.7 %	100.0 %	100.0 %	100.0 %
art	100.0 %	100.0 %	100.0 %	100.0 %
equake	82.5 %	89.2 %	99.0 %	100.0 %
bt	48.9 %	85.8 %	92.2 %	100.0 %
cg	84.9 %	86.7 %	93.5 %	100.0 %
ep	100.0 %	100.0 %	100.0 %	100.0 %
ft	44.7 %	78.9 %	100.0 %	100.0 %
is	100.0 %	100.0 %	100.0 %	100.0 %
lu	45.8 %	84.0 %	95.4 %	100.0 %
mg	35.6 %	73.0 %	79.5 %	100.0 %
sp	9.6 %	62.1 %	94.5 %	100.0 %
average benefit	56.2 %	86.4 %	95.6 %	100.0 %
marginal average benefit	**56.2 %**	**30.2 %**	**9.2 %**	**4.4 %**

Figure 8. Marginal Benefit of Region Parallelization. A well-prioritized parallelism plan will show decreasing marginal benefits as more of the recommended regions are parallelized. This table shows the average marginal benefit of 25% increments in the fraction of regions parallelized. The final row shows that a majority (56.2%) of benefit comes from the first 25% of regions with the following intervals showing decreasing average marginal benefits. This suggests that Kremlin's parallelism planner is effective at region prioritization.

Effectiveness of Region Prioritization An important aspect of planning is to ensure not only that the regions with the most benefit are selected but also that they are prioritized correctly. The planner attempts to place regions with the largest benefit at the beginning of the plan. Meeting this goal maximizes the productivity of the programmer by focusing their efforts where they are most valuable.
To evaluate the effectiveness of the ordering produced by Kremlin, we measured the fraction of total realized execute time reduction attained by following increasing portions of Kremlin's plans, including the first 25%, first 50%, first 75%, and all 100% of the plan. We would expect well-prioritized plans to generally produce monotonically decreasing benefits for each additional fraction that is added. As shown in Figure 8, Kremlin's plans are well-prioritized. The first 25% of the plans average 56.2% of the benefit, the next 25% averages 30.2% of the benefit, while the following 25% yields 9.2%, and the last 25% yields 4.4% of the benefit.

Influences on Plan Size Next, we evaluated how plan size is reduced as additional information is taken into account. The factors that we looked at were work coverage, self-parallelism, and usage of the full OpenMP planner personality. Figure 9 illustrates the impact of each of these factors on the programs. Programmers that take into account only work information (e.g. a gprof-based approach) would be left with an average of approximately 59% of the total regions to analyze and attempt to parallelize. With the addition of self-parallelism information for each region, the average plan size is cut to 25.4% of all regions. Finally, when using the full planner an average of only 3.0% of the regions are included in the plan. As we have shown in Figure 6, despite only parallelizing

[2] Programmer effort metrics for the *Enabling Transforms* part of parallelization is clearly a hard problem. We have also explored other metrics, like lines of code, as proxies for programmer effort in Kremlin, since it could perhaps be a better proxy for parallelization complexity. However, our impression from the benchmarks is that, at least, for OpenMP, region count is a better, albeit imperfect, approximation of programmer effort.

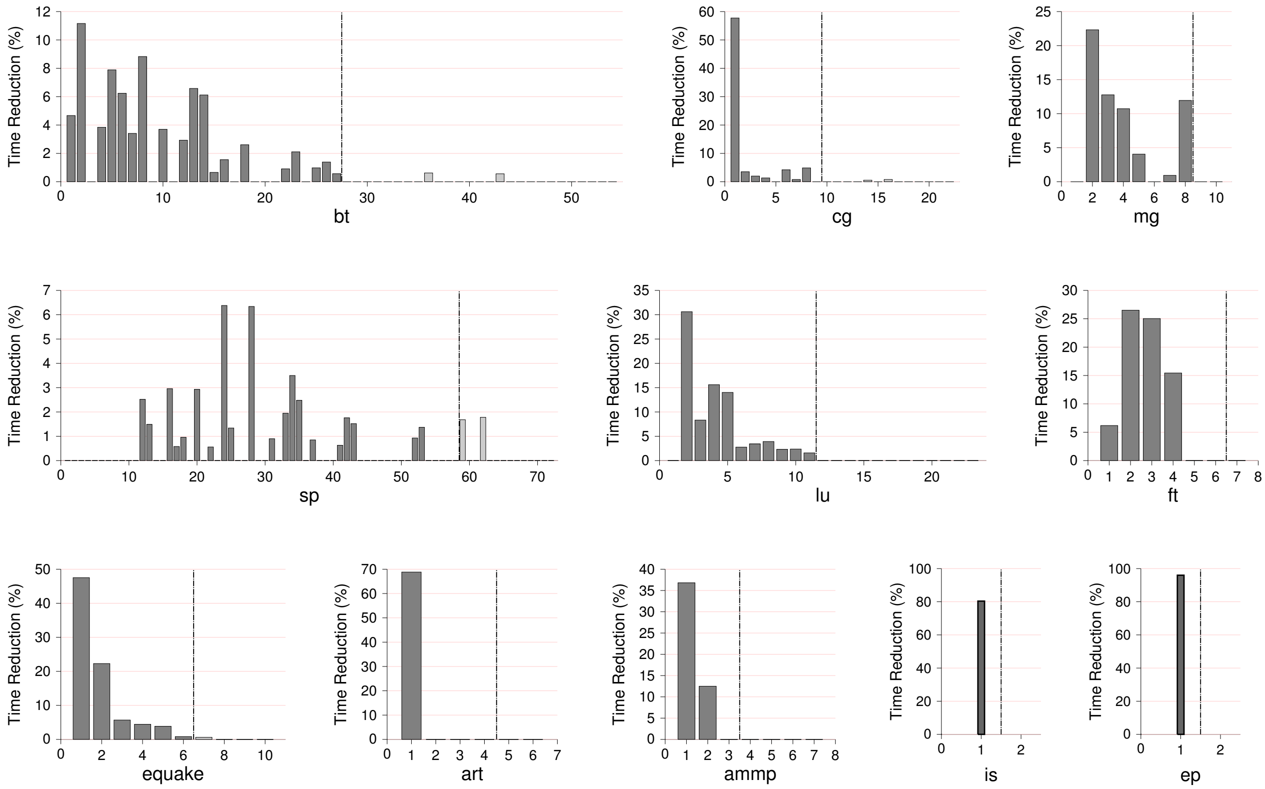


Figure 7. Effectiveness of Region Prioritization. Kremlin provides a list of regions prioritized by their estimated speedup so that users can maximize their productivity. The graphs above show the marginal decrease in execution time, relative to the original program run time, as each region in Kremlin plan is parallelized. We also included regions that were filtered out in the Kremlin plan but were chosen to be parallelized by the expert third party (MANUAL). These regions are shown to the right of the dotted line. As the graphs illustrate, little benefit came from regions that were parallelized by the third-party but that were not suggested by Kremlin.

a fraction of the regions, Kremlin achieves performance that is comparable to the highly-tuned MANUAL version.

Effectiveness of Self-Parallelism Metric To determine the ability of self-parallelism's ability to reduce the number of parallelism false positives (i.e. serial regions that are reported as being parallel), we calculated the self-parallelism and total-parallelism numbers for all 2535 regions that appear in the benchmarks, and classified them according to whether their parallelism is greater than or less than ("high" or "low") a threshold value of 5.0.

The total-parallelism metric identified 25.8% of regions as having low parallelism, while the self-parallelism metric identified 58.9% of regions as having low parallelism, a reduction of 2.28×.

7. Related Work

Parallelism Discovery and Dependence Testing Approaches for parallelism-related profiling have generally fallen into two categories: critical path analysis (CPA) and dependence testing. Critical path analysis dates back several decades, with early important works including [5, 23]. These approaches measured the number of concurrent operations at each time step along the critical path of the program. More recent work includes application of CPA to Java [13] as well as a modified CPA for the purpose of function-level parallelism in Java programs [32]. Unlike these approaches, Kremlin's hierarchical critical path analysis is able to localize parallelism within nested program regions, and provide concrete guidance on which program regions to parallelize.

Allen et al. [3] performed static analysis of Fortran programs in an attempt to automatically identify the correct granularity of parallelism for a target architecture. Kremlin is also able to identify the proper granularity of parallelism through the use of self-parallelism and planning personalities. However, the work in [3] was limited to structured, Fortran code; Kremlin is able to work with unstructured code that contains pointers which cannot be analyzed statically. Furthermore, Kremlin focuses on enabling the user to parallelize complicated code with which automatic parallelizing compilers have traditionally struggled.

Kulkarni et al. [22] used a critical path based analysis to bring insight into the parallelism inherent in the execution of irregular algorithms. In contrast to Kremlin's focus on localizing parallelism to concrete code regions via HCPA, Kulkarni's approach attempts to transcend the details of the implementation and to quantify the amount of latent parallelism in irregular programs that exhibit amorphous data parallelism.

Cilkview [14] is a recent tool that takes an already-parallelized Cilk++ program and estimates how that program's performance will change as the number of cores is increased. Similar to Kremlin, Cilkview leverages runtime information, and analyzes runtime dependencies in the program. However, Cilkview examines dependencies between pre-parallelized threads in a work-queuing runtime system rather than between instructions.

Another approach to parallelism-related profiling has been to use dependence testing to uncover the dependencies between different regions in the program. pp [24] is an early important work that proposed hierarchical dependence testing to estimate the parallelism in loop nests. Notable recent works include Alchemist [43] and SD3 [19], which reduces runtime and memory overhead of dependence testing through the use of parallelization and compression. Although dependence testing and Kremlin's HCPA share similar goals, Kremlin focuses on localizing and quantifying parallelism across many different, nested program regions rather than establishing independence of pre-existing regions. As a result, it can identify more nuanced forms of parallelism even though significant transformation is required to expose it. Dependence testing is generally more pessimistic and sensitive to existing program structure.

A number of works have used dependence testing to determine the probability that specific dependencies will occur [39, 41]. DProf [41] uses a compiler to identify may dependencies and then determine the probability that these dependencies will occur. von Praun et al. [39] introduced the *dependence density* metric to describe the probability that two random tasks would have a dependency. Both of these approaches target optimistic concurrency such as TLS or transactional memory.

The main difference between Kremlin's parallelism discovery and dependence testing frameworks is in the stage of parallelization (Figure 1) that profiling targets. Kremlin's parallelism discovery is meant to quantify the parallelism in a fashion that is not as strongly tied to the program's current structure, exposing hidden sources of parallelism. In contrast, dependence testing-based approaches are more aligned with the enabling transforms stage of parallelization as they enable identifying specific changes that need to be made to enable parallelism. In the absence of discovery and planning tools, [43] orders regions by total execution time. An interesting possibility would be to augment [43]'s approach with the improved analysis provided by Kremlin. ParaScope [18] used static analysis to expose difficult-to-analyze dependencies to the user so that they could circumvent them via refactoring.

The discovery phase takes advantage of a compression scheme that resembles whole-program path compression schemes [44]. We achieve much higher compression levels because we do not need to store information about the relative ordering of child subregions.

Parallelism Planning The task of parallelism planning has been mostly overlooked in the context of manual parallelization. Outside of manual parallelization, automatic parallelizing compilers such as SUIF [12] and Polaris [7] implicitly perform planning. Because these tools do not target user-assisted parallelization, their planning phases focus on finding thresholds for profitable exploitation.

Speculative parallelization systems [31, 34] have created new opportunities for compilers to exploit parallelism even in the face of difficult-to-analyze code, or infrequent dependencies that result in overly conservative execution. These systems typically have a memory speculation system, often in special hardware but sometimes in software, which removes the burden of proving the correctness of potential parallelizations, allowing the compiler to focus on selecting the parallelizations that maximize performance. TLS compilers [9, 11, 29, 31, 37, 46] also benefit from detecting parallelism and often use dynamic critical path or dependence testing analyses in order to establish regions which are likely to be profitable for TLS-style execution. Kremlin's HCPA can be used in a complementary manner by providing a way to guide programmers in restructuring their code to improve parallelism for execution on TLS. This can enable an even larger class of transformations than these systems natively support.

Recent work by Tournavitis et al. [38] provides a semi-automated approach to parallelization. This approach automates parallelism discovery using a form of dependence testing and uses machine learning to pick a set of regions to be parallelized. The selected regions are automatically annotated with OpenMP pragmas. The presumption is that the user will verify the correctness of the parallelization. While this approach has promise, it is limited by the compiler's ability to perform the *Enabling Transforms* phase of parallelization. In contrast, Kremlin has a more optimistic view of parallelism and is able to report regions with parallelism even if the compiler is not up to the task of exploiting it automatically.

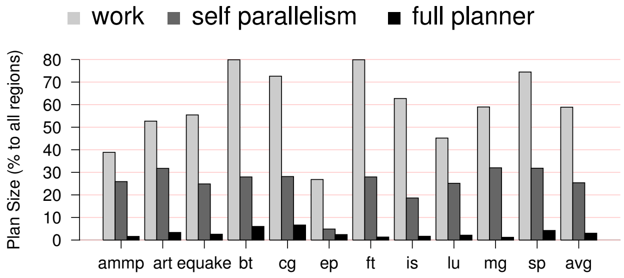


Figure 9. Evaluating Plan Size Reduction Due to Each of Kremlin's Planning Components. Plans based only on work coverage comprised 58.9% of all regions on average. Using self-parallelism to eliminate low parallelism regions cut this more than half (25.4%, on average). Finally, using the full OpenMP planner personality, the plan size was reduced to only 3.0% of the total regions.

Systems such as SUIF Explorer [28] and CAPO-Paraver [16] share Kremlin's focus on empowering the user during parallelization. SUIF Explorer's novelty focuses around its use of static interprocedural program analysis including pointer analysis and slicing; its use of dynamic analysis is very briefly described but appears to detect the absence or presence of memory dependencies within loops, and to provide time profiles for regions. CAPO-Paraver extends the CAPO parallelizing compiler to allow it to insert instrumentation that helps the user understand the load balancing properties of parallelized code. A short paper on Kremlin appeared in PPoPP [15].

8. Conclusion

Kremlin strives to simplify the task of parallelization by addressing the question: *what parts of the program should I spend time parallelizing?* Beginning with unmodified serial source code and sample inputs, Kremlin produces a list of regions in the order that they should be parallelized.

At the heart of Kremlin is an extension of the critical path analysis technique. Critical path analysis, which was invented in 1988, is rarely used in current-day performance profiling tools. In this paper, we extend that work by proposing the hierarchical critical path analysis (HCPA) technique, the self-parallelism metric, and the idea of using a planner algorithm that models the execution of the program on the machine. Given how conceptually straightforward HCPA appears to be, it seems surprising that it has not been proposed earlier, closer to the inception of CPA in the 80's. Although HCPA appears in retrospect to be straightforward, the "hard part" of the research is making the jump from the output of the HCPA algorithm to the creation of an effective parallel plan. The self-parallelism metric and planner concept are the key steps in doing this, and took many people-years of iteration for us to arrive at. One of the challenges with this research is figuring out exactly what information is needed to make meaningful recommendations, and how to decimate the information at each stage of the process. A valid approach must work without the collection of traces that are proportional to the execution time of the program, which can easily amount to terabytes. Further optimization detail is required for creating the multi-versioned shadow memory data-structures and rules that enable HCPA to run quickly and with low overhead.

Kremlin's results are very strong. Typically, by examining only the original serial code, the tool beats iterative parallelization by experts. It reduces the number of unnecessarily parallelized regions by 1.57x, attains better average performance, and in two out of the eleven benchmarks, improves speedups substantially by factors of 1.86x and 1.46x. Our results also show that Kremlin is effective at prioritizing regions: 56.2% of the benefit is captured in the first quarter of recommendations, and 86.4% comes from the first half of recommendations. All of these results suggest that Kremlin is likely to be highly effective at reducing programmer effort.

Acknowledgement

This research was funded by the US National Science Foundation under CAREER Award 0846152, and under Awards 0725357 and 1018850.

References

[1] "NAS Parallel Benchmarks 2.3; OpenMP C." `www.hpcc.jp/Omni/`.

[2] "Spec OMP2001 Benchmarks." `http://www.spec.org/omp`.

[3] F. Allen, M. Burke, R. Cytron, J. Ferrante, W. Hsieh, and V. Sarkar. "A framework for determining useful parallelism." In *Proceedings of the 2nd international conference on Supercomputing*, ICS '88, 1988.

[4] T. E. Anderson, and E. D. Lazowska. "Quartz: A tool for tuning parallel program performance." In *SIGMETRICS*, vol. 18, 1990.

[5] T. Austin, and G. S. Sohi. "Dynamic dependency analysis of ordinary programs." In *ISCA*, 1992.

[6] Bailey et al. "The NAS parallel benchmarks." In *SC*, 1991.

[7] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, W. Paek, Y. Pottenger, L. Rauchwerger, and P. Tu. "Parallel programming with Polaris." *IEEE Computer*, Aug 2002.

[8] J. M. Bull, and D. O'Neill. "A microbenchmark suite for openmp 2.0." *SIGARCH Comput. Archit. News*, December 2001.

[9] M. K. Chen, and K. Olukotun. "The Jrpm system for dynamically parallelizing Java programs." In *ISCA*, 2003.

[10] D. Dig, J. Marrero, and M. D. Ernst. "Refactoring sequential java code for concurrency via concurrent libraries." In *ICSE*, 2009.

[11] Z. H. Du, C. C. Lim, X. F. Li, C. Yang, Q. Zhao, and T. F. Ngai. "A cost-driven compilation framework for speculative parallelization of sequential programs." In *PLDI*, 2004.

[12] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and E. Bu. "Maximizing multiprocessor performance with the SUIF compiler." *IEEE Computer*, Aug 1996.

[13] C. Hammacher, K. Streit, S. Hack, and A. Zeller. "Profiling java programs for parallelism." In *Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering*, IWMSE '09, 2009.

[14] Y. He, C. Leiserson, and W. Leiserson. "The Cilkview Scalability Analyzer." In *SPAA*, 2010.

[15] D. Jeon, S. Garcia, C. Louie, S. Kota Venkata, and M. Taylor. "Kremlin: Like gprof, but for Parallelization." In *Principles and Practice of Parallel Programming*, 2011.

[16] G. Jost, H. Jin, J. Labarta, and J. Gimenez. "Interfacing computer aided parallelization and performance analysis." In *Computational Science ICCS 2003*, vol. 2660 of *Lecture Notes in Computer Science*, 715–715. 2003.

[17] K. Kelsey, T. Bai, C. Ding, and C. Zhang. "Fast track: A software system for speculative program optimization." In *CGO*, 2009.

[18] K. Kennedy, K. S. McKinley, and C. W. Tseng. "Interactive parallel programming using the parascope editor." *IEEE TPDS*, 1991.

[19] M. Kim, H. Kim, and C.-K. Luk. "SD3: A scalable approach to dynamic data-dependence profiling." *Microarchitecture, IEEE/ACM International Symposium on*, 2010.

[20] S. Kota Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. Taylor. "SD-VBS: The San Diego Vision Benchmark Suite." In *IISWC*, 2009.

[21] D. Kuck, Y. Muraoka, and S.-C. Chen. "On the number of operations simultaneously executable in fortran-like programs and their resulting speedup." *IEEE Transactions on Computers*, Dec. 1972.

[22] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Casçaval. "How much parallelism is there in irregular applications?" In *PPoPP*, 2009.

[23] M. Kumar. "Measuring parallelism in computation-intensive scientific/engineering applications." *IEEE TOC*, Sep 1988.

[24] J. R. Larus. "Loop-level parallelism in numeric and symbolic programs." *IEEE Trans. Parallel Distrib. Syst.*, 1993.

[25] C. Lattner, and V. Adve. "LLVM: A compilation framework for lifelong program analysis & transformation." In *CGO*, Mar 2004.

[26] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe. "Space-time scheduling of instruction-level parallelism on a Raw machine." In *ASPLOS*, October 1998.

[27] C. E. Leiserson. "The Cilk++ concurrency platform." In *DAC*, 2009.

[28] S.-W. Liao, A. Diwan, R. P. Bosch, Jr., A. Ghuloum, and M. S. Lam. "Suif explorer: an interactive and interprocedural parallelizer." In *Proceedings of the ACM SIGPLAN symposium on Principles and practice of parallel programming*, 1999.

[29] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. "POSH: a TLS compiler that exploits program structure." In *PPoPP*, 2006.

[30] N. Nethercote, and J. Seward. "Valgrind: A framework for heavyweight dynamic binary instrumentation." In *PLDI*, 2007.

[31] L. Rauchwerger, and D. Padua. "The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization." In *PLDI*, 1995.

[32] A. Rountev, K. Van Valkenburgh, D. Yan, and P. Sadayappan. "Understanding parallelism-inhibiting dependences in sequential java programs." In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, Sept 2010.

[33] V. A. Saraswat, V. Sarkar, and C. von Praun. "X10: concurrent programming for modern architectures." In *PPoPP*, 2007.

[34] G. Sohi, S. Breach, and T. Vijaykumar. "Multiscalar processors." In *ISCA*, 1995.

[35] N. R. Tallent, and J. M. Mellor Crummey. "Effective performance measurement and analysis of multithreaded applications." In *PPoPP*, 2009.

[36] W. Thies, S. Hall, and S. Amarasinghe. "Manipulating lossless video in the compressed domain." In *ACM Multimedia*, 2009.

[37] C. Tian, M. Feng, V. Nagarajan, and R. Gupta. "Copy or discard execution model for speculative parallelization on multicores." In *MICRO*, 2008.

[38] G. Tournavitis, Z. Wang, B. Franke, and M. F. P. O'Boyle. "Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping." In *PLDI*, 2009.

[39] C. von Praun, R. Bordawekar, and C. Cascaval. "Modeling optimistic concurrency using quantitative dependence analysis." In *PPoPP*, 2008.

[40] J. Wloka, M. Sridharan, and F. Tip. "Refactoring for reentrancy." In *FSE*, 2009.

[41] P. Wu, A. Kejariwal, and C. Caşcaval. "Compiler-driven dependence profiling to guide program parallelization." In *LCPC*, 232–248. 2008.

[42] B. Xin, and X. Zhang. "Efficient online detection of dynamic control dependence." In *ISSTA*, 2007.

[43] X. Zhang, A. Navabi, and S. Jagannathan. "Alchemist: A transparent dependence distance profiling infrastructure." In *CGO*, 2009.

[44] Y. Zhang, and R. Gupta. "Timestamped whole program path representation and its applications." In *PLDI*, 2001.

[45] Q. Zhao, D. Bruening, and S. Amarasinghe. "Umbra: Efficient and scalable memory shadowing." In *CGO*, 2010.

[46] H. Zhong, M. Mehrara, S. Lieberman, and S. Mahlke. "Uncovering hidden loop level parallelism in sequential applications." In *HPCA*, 2008.