Kismet: Parallel Speedup Estimates for Serial Programs

Donghwan Jeon   Saturnino Garcia   Chris Louie   Michael Bedford Taylor

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

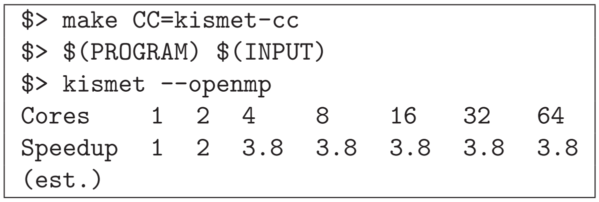
Abstract

Software engineers now face the difficult task of refactoring serial programs for parallel execution on multicore processors. Currently, they are offered little guidance as to how much benefit may come from this task, or how close they are to the best possible parallelization.

This paper presents Kismet, a tool that creates parallel speedup estimates for unparallelized serial programs. Kismet differs from previous approaches in that it does not require any manual analysis or modification of the program. This difference allows quick analysis of many programs, avoiding wasted engineering effort on those that are fundamentally limited. To accomplish this task, Kismet builds upon the *hierarchical critical path analysis* (HCPA) technique, a recently developed dynamic analysis that localizes parallelism to each of the potentially nested regions in the target program. It then uses a parallel execution time model to compute an approximate upper bound for performance, modeling constraints that stem from both hardware parameters and internal program structure.

Our evaluation applies Kismet to eight high-parallelism NAS Parallel Benchmarks running on a 32-core AMD multicore system, five low-parallelism SpecInt benchmarks, and six medium-parallelism benchmarks running on the fine-grained MIT Raw processor. The results are compelling. Kismet is able to significantly improve the accuracy of parallel speedup estimates relative to prior work based on critical path analysis.

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



```
$> make CC=kismet-cc
$> $(PROGRAM) $(INPUT)
$> kismet --openmp
Cores     1  2  4    8    16   32   64
Speedup   1  2  3.8  3.8  3.8  3.8  3.8
(est.)
```

Figure 1: **Kismet's User Interface.** After compiling and executing the program, Kismet produces estimated upper bounds on parallel speedups for the program.

General Terms Measurement, Performance

Keywords Hierarchical Critical Path Analysis, Expressible Self-Parallelism, Performance Estimation, Parallel Software Engineering

1. Introduction

Software engineers currently face the enormous task of refactoring their programs to take advantage of multi-core processors. These multi-core processors provide extensive parallel resources, providing the potential for greatly improved performance. However, this improved parallel performance is typically unlocked only after extensive engineering efforts that rely on the individual engineer's experience rather than automated tools. Frequently, the parallelization process results in difficult-to-diagnose bugs and a murky understanding of the connection between code modifications and resulting performance.

The performance of refactored serial programs often falls short of optimistic speedups derived from raw hardware resources. Worse-than-expected performance can be caused by several factors. First, the implementation may be poor—the result of missed parallelism opportunities or poorly executed parallelization attempts. Second, the program may have inherently low amount of parallelism—possibly the result of choosing an algorithm without considering its parallelizability. Finally, the target system may be a poor choice for that program—the result of a mismatch between the structure of the parallelism in the program and the ability of the system to efficiently exploit it.

OOPSLA'11, October 22–27, 2011, Portland, Oregon, USA.

An assortment of tools have been developed to help in the task of refactoring for parallelism. Some of these tools [6, 18, 47] help the programmer debug performance problems in their parallel implementation. While these tools help the programmer overcome poorly executed parallelization, they fail to uncover missed parallelism opportunities or determine if other factors in poor performance are the limiting reagents. Other tools have looked into measuring the parallelism available in a program [27, 30] but often look only at abstract models of execution and do not provide realistic estimates on the speedup of the refactored program. Yet other tools examine the scalability of a parallel program [10, 53] but look only at an existing implementation and therefore do not provide insight into the fundamental scalability of a program. Furthermore, a vast majority of existing tools assume that there is already a mature parallel implementation. Relying on these tools greatly increases the probability of undesirable sunk costs: programmers can waste significant time and money in refactoring code only to determine a fundamental limitation to its performance.

Kismet's Purpose In this paper we introduce Kismet, a parallel speedup estimation tool. Kismet performs dynamic program analysis on an unmodified, serial version of a program to determine the amount of parallelism available in each region (e.g. loop and function) of the program. Kismet then incorporates system constraints to calculate an approximate upper bound on the program's attainable parallel speedup. These constraints include the number of cores, synchronization overhead, cache effects, and *expressible parallelism* types (i.e. loop and task parallelism for multicore chips; instruction level parallelism for VLIW-style chips; and data level parallelism for vector machines).

Kismet provides a simple usage model in the style of `gprof`, as shown in Figure 1. The program is first compiled with a drop-in compiler replacement called `kismet-cc`. The program is run with a representative input, which produces as a side-effect an output file containing profile information. The user then runs `kismet`, which analyzes the output file and generates a table of estimated speedup upper-bounds for a spectrum of core counts.

Kismet's Basic Structure In order to estimate the parallel performance of a serial program, Kismet uses a *parallel execution time model*. Kismet's parallel execution time model is based on the major components that affect parallel performance, including the amount of parallelism available, the serial execution time of the program, parallelization platform overheads, synchronization and memory system effects which contribute in some cases to super-linear speedups.

To determine available parallelism, Kismet extends the recently developed *hierarchical critical path analysis*, or *HCPA* [16, 21]. HCPA measures the critical path and work across many nested regions of the program hierarchy [15] in an efficient manner, which allows the average parallelism of region (including its children) to be determined. The key to leveraging HCPA is the *self-parallelism* metric [16], which provides a mechanism for separating the parallelism of parent regions from their children. With this metric, we can assign an average parallelism quantity to each region in the program, which quantifies the potential speedup if only that region were targeted. This is useful, for instance, in determining which loop in a triply-nested loop is most promising for parallelization.

HCPA is an extension of critical path analysis (CPA) and therefore is able to identify many subtle forms of parallelism that potentially are available only after significant code refactoring that spans many loops and independent function calls. In cases where parallelism is exploitable only after major refactoring across many region levels, HCPA reports this parallelism at the highest level region. This self-parallelism value results in a prediction which is consistent with the major refactoring. Thus Kismet can postulate transformations that greatly exceed the capabilities of today's parallelizing compilers. Kismet's optimistic view of speedup attempts to take into account the programmer's greater ability to perform code transforms that would be unsafe in automatic parallelizing compilers.

Kismet improves upon HCPA so that it can be used to estimate attainable parallel speedups from serial code. In particular, it employs a summarizing variant of HCPA called *summarizing hierarchical critical path analysis*, or *SHCPA*, which improves the scalability while maintaining the accuracy of the analysis by aggregating the data between sibling nodes in the region graph in a context-sensitive manner.

Compelling Results To demonstrate the effectiveness of Kismet in creating realistic upper bounds on parallel performance, we evaluated Kismet in three contexts. First, we looked at six medium-parallelism benchmarks running on the MIT Raw processor [35, 36]. Second, we examined the performance of 6 low-parallelism SpecInt benchmarks relative to results reported in the literature. Finally, we examined Kismet's accuracy on high-parallelism benchmarks from the NAS Parallel Bench [9] on a 32 core AMD system. In all contexts, our results show that Kismet is able to create strong approximate upper bounds on the actual parallel performance.

The remainder of this paper proceeds as follows. Section 2 overviews the architecture of the Kismet tool. Section 3 continues with a description of the SHCPA implementation, including the extensions that Kismet implements for speedup estimation. Section 4 examines how the SHCPA data is processed to account for machine and parallelization system properties. Section 5 overviews the Kismet-based tools we built for the two platforms. Section 6 presents results, Section 7 presents related work, and Section 8 concludes.

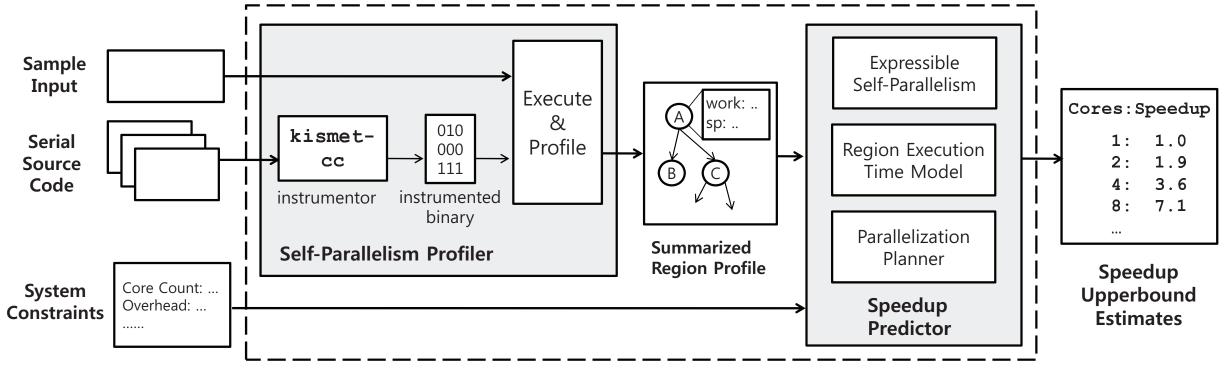


Figure 2: **Kismet System Architecture.** Starting with a program's source code, Kismet produces an instrumented binary by inserting profiling code. Running the instrumented binary on the sample input outputs a trace file containing both program structure and self-parallelism. Finally, the speedup predictor estimates the speedup upper bound based on the profile data. The parallelism classifier filters unexpressible parallelism for realistic speedup estimates (via the *expressible self-parallelism* filter) and the parallel execution time model incorporates hardware constraints and parallelization overhead for accurate performance prediction.

2. Kismet Overview

In this section, we provide a high-level overview of the Kismet system architecture, as shown in Figure 2.

To use Kismet, programmers need only supply the unmodified serial source code and a sample input data for a program. The output is the upper bounds on parallel program speedup as the number of cores is varied, as shown in Figure 1. Internally, Kismet also makes use of a set of input files that describe the targeted machine. Since speedup results are often quite machine-dependent, this serves to improve Kismet's accuracy.

Kismet operates in two phases; the first phase is a profiler that collects the self-parallelism data and the second phase is a speedup predictor which applies the machine and system constraints.

Self-Parallelism Profiler Gathering of self-parallelism information in Kismet starts with a static instrumentation phase which instruments the target program with code that implements the SHCPA dynamic analysis, and ends with running the instrumented program.

The static instrumentation phase transforms the input code to support SHCPA during execution of the sample input. The inserted instrumentation consists of calls to a special SHCPA library, which will perform the dynamic analysis during execution. In addition to adding instrumentation for calculating critical paths, Kismet also inserts instrumentation to clearly delineate the regions of the code. Three types of regions—loops, functions, and sequence—are used, allowing SHCPA to calculate each region's self-parallelism and to determine the type of parallelism in each region.

The Kismet code instrumentator utilizes LLVM's [31] static instrumentation infrastructure to perform a deeper level of analysis than is available with dynamic instrumentation tools such as Valgrind [41]. This allows Kismet to easily uncover the program structure and account for false dependencies introduced by loop induction variables and reduction variables. Static instrumentation also provides greater opportunity for optimizing the instrumented code in order to reduce the overhead associated with profiling.

The dynamic analysis phase begins when the instrumented binary is run with the sample input to produce per-region statistics. For each dynamic region that is executed, the dynamic analysis computes three key pieces of data: the critical path length, the amount of work done, and the self-parallelism. Section 3 describes how this data is produced in more detail. The data produced for each region is relatively small but the number of dynamic regions grows quickly, leading to a possibly unmanageable amount of data. Kismet improves the manageability of region data by summarizing the data as it profiles, creating *summarized region profiles*. The summarized region profiles reduce the number of recorded regions by orders of magnitude, leading to much smaller log sizes and allowing for more efficient processing in later stages of Kismet. While the reduced log size from summarization is desirable, summarization should not compromise the quality of self-parallelism information. Kismet's SHCPA provides rich call context-sensitive region information, helping the speedup predictor not to underestimate the potential speedup from parallelization. We'll examine more details about this context-sensitive approach in Section 3.

Speedup Predictor After running the instrumented binary on the sample input, Kismet has captured the underlying structure of the application in the form of the summarized region profile. The next step is to combine this information with machine and parallelization system properties in order to make a prediction.

Performance strongly depends on the target system. Kismet accepts a list of target-dependent parallelization constraints and utilizes this information to provide more accurate predictions. Typically constraints include a simple hardware specification (e.g. the number of available cores), the types of expressible parallelism by that system, and functions that quantify parallelization overheads such as synchronization. We have found that only a small number of

```
for (i=1 to N) {
  foo(1);   // callsite A
  foo(N);   // callsite B
}

void foo(int size) {
  for(i=1 to size) {
    ...
    // loop body
    ...
  }
}
```

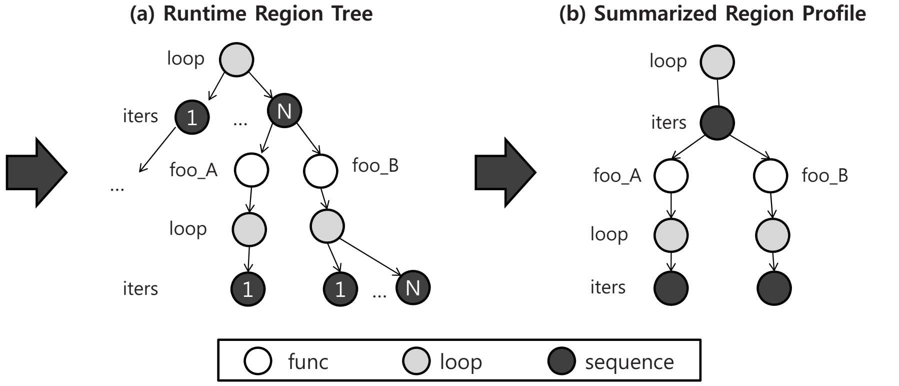


Figure 3: **Kismet's Program Representation.** (a) Kismet's self-parallelism profiler builds a hierarchical region structure from source code, consisting of three types of regions. At runtime, it forms a region tree consisting of dynamic regions. (b) Kismet radically reduces the output file size by compressing runtime region tree into a call context-sensitive region tree. The summarized tree preserves context-sensitive parallelism information, exposing more parallelization opportunities.

constraints are needed to accurately predict performance. This simplifies the process of extending Kismet to new platforms. We were surprised at the ease with which our model could support two very different parallel systems—an MIT Raw tiled processor and a 32-core AMD multicore system.

The speedup predictor contains three sub-components. The first sub-component is a modified self-parallelism metric called *expressible self-parallelism*, or *ESP*, which filters out parallelism unexpressible by the specified target system.

The second sub-component is the *parallel execution time model*. The parallel execution time model allows the speedup predictor to estimate the parallel execution time of each program region and the whole program based on a given parallelism plan. This model incorporates self-parallelism, number of allocated cores, and parallelization overhead. Kismet also provides an extended, cache-aware parallel execution time model that considers the impact of caching on parallel execution. The parallel execution time model is used by the resource allocator to evaluate between completing parallelization plans and determines the final speedup numbers reported by Kismet. Section 4 describes these models further.

The final sub-component is the *speedup planner*. An ideal parallel system will take advantage of all the expressible parallelism in a program. This desirable property is not available on most existing systems. These systems have other constraints–such as limited hardware resources, synchronization overhead, or poor support for nested parallelism – that make *expressible parallelism* not be *exploitable parallelism*. The speedup planner creates a mapping from regions to parallel resources, modeling at a high-level what the execution of the parallelized program would look like; we refer to this mapping as the parallelization plan. In Section 5, we describe how a parallelization plan is created, using two widely different systems as case studies.

3. Self-Parallelism Profiler

Kismet extends the HCPA-based profiler introduced in [16] to quantify the parallelism in each region of the program in a summarized fashion. Additionally, it modifies the algorithm in a number of ways to facilitate speedup prediction and calculation of expressible self-parallelism. In this section, we describe Kismet's implementation of SHCPA and self-parallelism.

3.1 Summarizing Hierarchical Critical Path Analysis

Summarizing hierarchical critical path analysis extends traditional critical path analysis to incorporate the hierarchical region structure of a program. SHCPA calculates the critical path of each dynamic region of the program, unlike CPA which looks only at the critical path of the whole program. This per-region calculation provides the basis for improved localization of parallelism information.

Types of Regions Kismet demarcates region boundaries at static instrumentation time. Kismet includes all loops and functions in the list of regions but introduces the concept of a *sequence* region, an important extension of prior work in HCPA. A sequence region can be any single-entry piece of code but Kismet restricts sequences to two important cases: loop bodies and self-work sequences. Loop body regions form a child region for each iteration of a loop region, allowing Kismet to identify loop-level parallelism. Self-work sequence regions are sequences of code that are contained in non-leaf regions and do not have any function calls or loops. These regions may seem unintuitive but they address a concern in prior work on HCPA: the separability of different types of parallelism. Self-work sequences factor out the instruction level parallelism in regions that would otherwise contain a mix of task-level parallelism (from its other children) and instruction level parallelism. Figure 3 (a) shows how regions are dynamically formed from a sample piece of code.

SHCPA Implementation SHCPA uses shadow memory to track the earliest time that an instruction is available. When each operation executes, it reads from shadow memory the availability times of all of its dependencies. SHCPA adds the maximum time amongst these dependencies to the latency of the operation being performed and then stores this value to the shadow memory location corresponding to the operation. Dependencies that are not true dependencies are filtered out using two main mechanisms. First, Kismet operates on LLVM's SSA form IR. This eliminates false output (i.e. write-after-write) dependencies. Next, Kismet detects induction and reduction variables then breaks the false dependencies that result from them.

Kismet tracks control dependencies through the use of control dependence analysis and a dynamic control dependence stack. Control dependence times are pushed to and popped from the control dependence stack whenever a control dependent region is entered and exited. This stack has monotonically increasing values from the bottom to the top, allowing Kismet to include only the topmost entry in the list of dependencies for each instruction.

Each active region effectively has its own shadow memory, enabling Kismet to independently calculate the region's critical path length. All times in the region's shadow memory are logically initialized to zero upon entry so that a reference to an instruction outside the region will be assumed be available immediately at the beginning of the region (i.e. time 0). Regions track the largest time that was written to their view of shadow memory; this value is the *critical path length*. Each region also records the combined latencies of all operations performed in that region; this summation is the amount of *work* in that region. The ratio of work to critical path length is the *total parallelism* of that region.

SHCPA can also record other useful runtime information. For example, load and store counts are also included for cache-aware performance estimation that will be explained in Section 4.

Identifying Independent Children Kismet also determines if all of the children of a non-leaf region are independent and therefore can be executed in parallel. This information is stored as the region's "P bit" and is calculated according to the following equation:

$$P = CP(parent) == MAX(CP(child_1), ..., CP(child_n))$$

where $CP(parent)$ is the critical path length of the parent and $CP(child_i)$ is the critical path length of the i^{th} child.

If all children can be executed in parallel, then the length of the critical path will simply be the length of longest critical path of all of the children. In this case, the "P bit" will be 1. Kismet uses the information in the "P bit" to help identify expressible parallelism. Section 4 will describe this in more detail.

SHCPA Optimizations Kismet's implementation of shadow memory is based on a two-level table that equally divides the memory address space. Kismet contains several optimizations of shadow memory for increased performance and reduced memory usage, most notably shadow register tables and dynamic allocation of shadow memory. Shadow register tables optimize the common case of writing to local variables by creating a direct access shadow register table. This avoids the overhead of indirectly accessing the two-level page table. Dynamic allocation of shadow memory reduces the overhead that would result from keeping all shadow entries in memory.

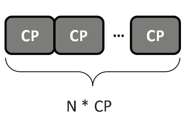
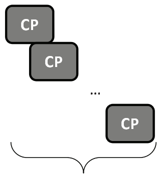
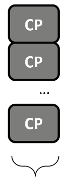
Type	Serial	DOACROSS	DOALL
CP (R)	 N * CP	 (N/2) * CP	 CP
SP (R)	$\frac{N * CP}{N * CP} = 1.0$	$\frac{N * CP}{(N/2) * CP} = 2.0$	$\frac{N * CP}{CP} = N$

Figure 4: **Self-Parallelism Calculation on Regions with Varying Parallelism.** Self-parallelism computes the amount of parallelism in a parent region that is attributable to that region and not its children. The figure above shows that Kismet's self-parallelism calculation successfully quantifies parallelism across a spectrum of loop types, ranging from totally serial to partially parallel (DOACROSS) to totally parallel (DOALL). The shaded boxes are child regions, corresponding to separate iterations of the loops. The relative scheduling of child regions is indicated spatially, with time running from left to right. The self-parallelism calculation correctly quantifies parallelism in non-loop region hierarchies as well.

3.2 Calculating Self-Parallelism

Although SHCPA produces a total parallelism value for each region of the program, this alone is not enough to localize parallelism to specific regions of the program. Total parallelism is computed without knowledge of the region hierarchy and thus incorporates parallelism that originates from child regions. Kismet's self-parallelism metric takes the next step in localizing parallelism.

To determine the self-parallelism of a region R, $SP(R)$, Kismet employs the following equation:

$$SP(R) = \begin{cases} \frac{\sum_{k=1}^{n} cp(child(R,k))}{cp(R)} & \text{R is a non-leaf} \\ \\ \frac{work(R)}{cp(R)} & \text{R is a leaf} \end{cases}$$

Here n is the number of children of R, $child(R, k)$ is the k^{th} child of R, $cp(R)$ is the critical path length, and $work(R)$ is the amount of work in R.

In contrast to earlier work on self-parallelism [16], Kismet treats leaf and non-leaf regions differently because they contain different types of parallelism. Leaf regions contain no children and therefore will contain only instruction level parallelism (ILP). Non-leaf regions have children that may provide opportunities for either loop-level or task-level parallelism. Prior work in HCPA [16] employed self-work in the calculation of SP for non-leaf regions but this led to a mixture of ILP and task-level parallelism inside of a single region. Kismet includes the new self-work sequence region to factor out the ILP in order to compute expressible self-parallelism.

Figure 4 demonstrates the calculation of SP in three non-leaf regions, one totally serial, one partially parallel (DOACROSS), and the other totally parallel (DOALL). For simplicity, in the example, each iteration's critical path length cp is the same. For the serial loop, the measured $cp(R)$ will be equal to $n * cp$ and the computed self-parallelism will be $\frac{n*cp}{n*cp} = 1$, which is expected since serial dependences prevent overlapped execution of regions. For the DOACROSS loop shown, where half of an iteration can overlap with the next iteration, $cp(R)$ will be the half of the $cp(R)$ for the serial loop. Thus $SP(R)$ is $\frac{n*cp}{(n/2)*cp} = 2$. For the DOALL loop, $cp(R)$ will be equal to cp, so $SP(R)$ is $\frac{n*cp}{cp} = n$. Although we show three relatively simple cases here, this method is a good approximation of self-parallelism even with more sophisticated child region interaction.

3.3 Summarizing Hierarchical Critical Path Analysis

The number of dynamic regions quickly grows as nested loops with many iterations are executed. This large amount of regions poses practical challenges not only in the size of the profile output but also in the runtime of algorithms that need to analyze this data. Garcia et al [16] developed a dictionary-based compression algorithm to reduce the profile size. However, this type of compression performs poorly on programs with loop iterations that vary in their work or critical path length. In this section we will describe a new technique that is used by Kismet for managing the number of regions.

Region Summarization Kismet combines all dynamic regions that have the same region context into a single summarized region. Figure 3 depicts how the runtime region tree (a) becomes a summarized region profile (b). In this method, all loop iterations collapse to a single node, greatly reducing the number of regions. Each node calculates weighted averages for self-parallelism, work, and other profiled data across all dynamic regions corresponding to that node.

Kismet maintains a 'current' pointer that tracks the summary node that corresponds to the current dynamic region. When a new region is entered, it updates the 'current' pointer to one of its children node based on statically assigned callsite ID information. If there is no corresponding node, it creates a new summary node and updates the 'current' pointer. When a region exits, the region's profiled information is added to the current node and the pointer returns to the parent node. This process is similar to the call context tree described in [5] but modified for Kismet's region hierarchy.

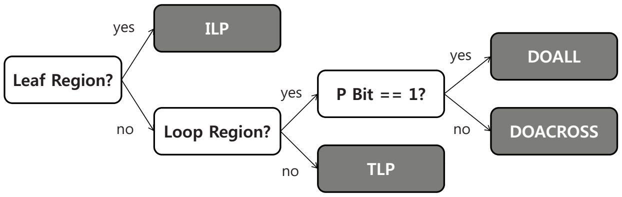


Figure 5: **Parallelism Identification Logic.** Kismet uses the program structure and parallelism information provided by SHCPA to help classify parallelism. This figure shows the simple classification process. Kismet then uses the classification result to calculate the expressible self-parallelism (ESP). ESP quantifies the amount of expressible parallelism within a specific region of the program.

Utilizing Context Sensitivity The example summarized region profile shown in Figure 3(b) contains two nodes for the same function (`foo`) from what appears to be the same context. This corresponds to two separate calls from the same loop. While this increases the number of nodes in the summarized profile, it allows Kismet to uncover new parallelism opportunities.

To understand the merit of context-sensitive representation, consider the code in Figure 3. When the loop in function `foo` is parallel and N is large, the parallelism of this loop significantly differs between callsites A and B. Callsite A's loop will always have a self-parallelism of 1, providing no benefit to parallelism and likely causing slowdown due to synchronization overhead. Callsite B's loop will have a self-parallelism of N and would likely be a good candidate for parallel refactoring. Kismet can capitalize on the split contexts, incorporating the speedup from callsite B into its estimates while ignoring callsite A.

4. Speedup Predictor

Kismet's speedup predictor attempts to find the upper bound on parallel speedup of a program by examining a spectrum of candidate parallelizations of the program on the target machine. Kismet's self-parallelism profiling provides the groundwork for calculating this speedup but it alone is not enough to determine a tight bound on speedup. In this section we will describe how Kismet processes the self-parallelism data to predict the maximum parallel speedup.

4.1 Expressible Self-Parallelism (ESP)

While Kismet's self-parallelism profile quantifies the parallelism in each region of the program, there is no guarantee that the parallelism will be expressible. Many systems have limitations on the type of parallelism that can effectively be expressed. Kismet transforms self-parallelism into expressible self-parallelism (ESP) in two steps. First, it classifies the type of parallelism found in each region. Second, it uses this classification to conditionally adjust self-parallelism into ESP, as follows. Regions that have self-parallelism that is unexpressible are assigned an ESP of 1. Regions with self-parallelism that is expressible have an ESP that is equivalent to their SP.

Figure 5 illustrates the Kismet's decision process when classifying parallelism. As described in Section 3, Kismet's region hierarchy has been designed to ensure that only leaf regions have instruction level parallelism (ILP) and that ILP is found only in leaf regions. The first step in Figure 5 is thus to check if the region is a leaf. If the region is not a leaf then the parallelism is either of the form of loop- or task-level parallelism. Kismet checks the region type to determine if there is a loop or a function.

Kismet further classifies loop parallelism based on whether there are cross-iterations dependencies. Loops *without* cross-iteration dependencies are classified as DOALL while those *with* cross-iteration dependencies are classified as DOACROSS. While Kismet's profile output does not contain statistics on the number of cross iteration dependencies, it does contain the information needed to quickly distinguish DOALL and DOACROSS loops. Namely, the "P bit" described in Section 3 indicates if all iterations are independent. Kismet examines the "P bit" for the region, classifying the region as DOALL if $P == 1$ and DOACROSS otherwise.

As with any dynamic analysis tool, Kismet's identification of parallelism is subject to differences across multiple inputs. In practice we have found that while the amount of speedup may vary slightly across multiple inputs, the Kismet classification is consistent across these same inputs.

4.2 Parallel Execution Time Model

Although self-parallelism is a major factor that affects the realizable speedup of a region, there are other major factors such as allocated core counts and parallelization overhead. Kismet uses a parallel execution time model that captures major factors that affect parallel execution time. With the parallel execution time model, Kismet's speedup predictor can evaluate the effectiveness of parallelization plan it produces, and reports the plan that would bring the highest speedup. We also show a cache-aware parallel execution time model that incorporates changed cache miss rates after parallelization.

Base Model The base parallel execution time model incorporates region structure, core count, and parallelization overhead in addition to self-parallelism. This model uses the following equation to determine the execution time of region R:

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

While there are different equations for leaf and non-leaf regions, they follow the same general model. The first term represents the time needed to execute the parallelized, assuming that $A(R)$ cores are allocated to that region. The top of the fraction represents the serialized execution time—the work of a leaf region, or the sum of the children's work of a non-leaf region. This time is divided by the minimum of the self-parallelism of the region ($SP(R)$) and $A(R)$. Intuitively, this means that the speedup is either fundamentally limited by the parallelism available—when $SP(R)$ is the limiting factor—or by the amount of parallel resources allocated to the region—when $A(R)$ is the limiting factor. Note that the execution time of the non-leaf regions depend on the execution time of their children; this forces a bottom-up approach to calculating the execution time of the program.

The second term, $O(R)$ models target-dependent parallelization overhead. Parallel execution typically involves overhead from several sources: thread management, synchronization, communication, etc.. As a result, the overhead factor is highly target dependent. For example, the synchronization operation takes less than 20 cycles in the MIT Raw processor but takes several thousand cycles on shared memory multicore processors. As such, Kismet allows target-dependent customization of $O(R)$ by accepting parallelization constraints. This overhead function directly impacts the parallelization granularity as the amount of work in a region should offset parallelization overhead for a profitable parallelization.

Cache-Aware Model While the base model is able to accurately model benchmarks that have up to linear speedup, our results showed that some benchmarks resulted in super-linear speedup when parallelized. For example, the cg benchmark from the NAS Parallel Bench [9] showed significant super-linear speedup when using between 4 and 16 cores on 32-core AMD Opteron system. We found that this was a result of increasing cache size with a larger number of cores on this system, prompting us to include a cache-aware model of parallel execution time.

Kismet's cache-aware model extends the base model by including the memory service time (MST) in the calculation of $ET(R)$. MST represents time spent in memory accesses that resulted in a cache miss; it is calculated using the following equation:

$$MST(R) = \begin{cases} \dfrac{\sum_{k=1}^{n} MST(child(R,k))}{A(R)} & \text{non-leaf} \\ \\ \dfrac{\sum_{i=1}^{depth} CMT_i(R)}{A(R)} & \text{leaf} \end{cases}$$

For both leaf and non-leaf equations, MST sums the time spent and for cache misses—either in that region for a leaf region, or among all children of a non-leaf region—in the level i cache, CMT_i, and divides this by the number of cores allocated to the region, $A(R)$. This optimistically assumes that the memory system of the target is scalable, distributing memory accesses evenly across cores so that they may be simultaneously serviced without penalty. Although it is possible to model more complicated behaviors of memory systems, this simple cache model appears to do a reasonable job of predicting superlinear speedup effects due to the memory systems.

To calculate the cache miss time at level i, Kismet uses the following equation:

$$CMT_i(R) = \sum_{i=1}^{n} MemCnt(R) * Miss_i(R, conf) * Penalty_i$$

where $MemCnt(R)$ is the number of memory accesses in region R, $Miss_i(R, conf)$ is the cache miss rate for level i, $conf$ is a specific memory configuration, and $Penalty_i$ represents the penalty for a level i cache miss. As more cores are allocated, the total cache size of $conf$ increases, potentially leading to a decrease in $Miss_i(R, conf)$.

5. Case Studies - Raw and Multicore

In this section, we demonstrate how Kismet can be configured to a specific platform by examining two very different platforms: the MIT Raw tiled multicore processor ("Raw") [14, 35, 36, 48] and a conventional multicore processor ("Multicore"). Table 1 shows details of these two targets. We also model specific software platform because software platforms also create constraints in parallelization, affecting the speedup even on the same hardware. Specifically, we model the automatically parallelizing compiler RawCC [4, 32] for Raw, and manual OpenMP parallelization for Multicore. For each platform, we first introduce hardware characteristics and parallelization constraints, and describe how we model parallelization overhead in parallel execution time model, and then finally describe the target-specific planning algorithm.

5.1 Targeting Raw in Kismet

Platform Description MIT Raw is an early tiled multicore processor [17, 46, 49] featuring a fast, 1-cycle per hop, fine-grained scalar operand network [50]. Although many different forms of parallelism (ILP, TLP, DLP, etc) are expressible on the Raw ISA, we model the RawCC [4, 32] parallel compiler, for which only ILP is expressible.

Platform	Raw	Multicore
Core Type	Modified MIPS	AMD Opteron
L1 Size	32KB / Core	64KB / Core
L2 Size	-	512KB / Core
L3 Size	-	6MB / Four Cores
SW Platform	RawCC	OpenMP
Expressible Parallelism	ILP	DOALL
Non Reduction Parallelization Overhead (cycles)	$2+2\sqrt{N}$	250 * N
Reduction Parallelization Overhead (cycles)	$2+2\sqrt{N}$	500 * N

Table 1: **Overview of Two Platforms** - Raw and Multicore. These two targets have different constraints in parallelization, expressible parallelism, and parallelization overhead.

RawCC finds ILP in each basic block and performs space-time scheduling to exploit it. For each instruction, the space-time scheduling determines which core executes the instruction for minimum total execution time. Inter-core data dependencies are resolved utilizing Raw's low latency network, and control flow information is broadcast across cores to ensure all cores execute the same basic block. If needed, RawCC performs loop unrolling to increase the amount of exploitable ILP in a loop.

Modeling Parallelization Overhead Two sources can incur parallelization overhead in Raw: control dependencies and data dependencies.

To ensure control dependencies are respected, RawCC broadcasts the control dependency information to all cores via Raw's static network. At the end of every basic block, each core waits for the control dependence information and branches to the specified basic block when the information arrives. The broadcast cost is $2+2\sqrt{N}$, where N is the number of cores. In our parallel execution time model for Raw, we approximate this overhead based on [50]: an injection latency of 2 cycles, a network diameter of $2\sqrt{N}$, and a per-hop latency of 1 cycle.

Data dependences between two instructions on different cores also incur communication overhead. Unlike with broadcasts, the cost can be hidden if the communication is not on the critical path of the execution by RawCC. As Kismet aims to bound the achievable highest speedup, Kismet does not model this overhead.

Planner Algorithm The planner algorithm takes as input the summarized region profile which includes a region tree where each node is a summarized region and each edge represents "reachable" relationship between them. As RawCC can express only ILP in a program, the planner first filters nodes with non-ILP parallelism and sets their ESP value to 1, effectively eliminating them from consideration. After ILP regions are identified, producing the plan with highest speedup is straightforward. For each ILP region R, decide A(R) that minimizes ET(R) with the given parallel execution time model. For non ILP regions, A(R) is simply set to one, representing serial execution. When A(R) is determined, parallel execution time model calculates the estimated parallel execution time of the root node with given A(R) function, and will compute the speedup against serial execution time.

5.2 Targeting Multicore with OpenMP in Kismet

Platform Description The multicore platform represents conventional multicore processor systems such as Intel's Nehalem or AMD's Opteron line or processors. They use shared memory for inter-core communication and as a result the latency is significantly higher compared to Raw's low-latency networks. For the software platform, we target the popular OpenMP platform that exploits mainly DOALL parallelism. In addition to the restricting expressible parallelism to DOALL, we also disallow nested parallelization – although OpenMP supports nested parallelization, the feature is rarely used in practice as synchronization overhead is typically too large.

Modeling Parallelization Overhead OpenMP parallelization involves overhead in several aspects: thread creation, thread scheduling, reduction operations, and barrier cost. We found that thread creation cost is typically amortized with a thread pool implementation and scheduling cost is negligible when static scheduling is used. We model barrier and reduction costs since they significantly impact performance. The values chosen in Table 1 was taken from running the EPCC micro-benchmark [11] on 32-core AMD Opteron machine.

Planner Algorithm Once regions with unexpressible parallelism are filtered out, the main constraint in the Multicore planner is prohibited nested parallelization. When nested parallelization is disallowed, the planner cannot choose more than one region among regions in the path from the root node to any node in summarized region profile.

To find the optimal solution with the constraint, Kismet uses a dynamic programming algorithm. The core intuition of the algorithm is that a region should be parallelized only when the benefit of parallelization is greater than the benefit from parallelizing any set of descendant regions. The planner traverses the region profile in a bottom-up fashion, from leaf nodes up to the root node, while saving the optimal plan P(R) at each region R. When the planner processes a new region, it compares the expected benefit of parallelizing the region against the cumulative benefit of the optimal plans of its child regions. If the benefit of parallelizing R exceeds the cumulative benefit of child regions, P(R) is set to R; otherwise P(R) is set to the union of child regions' optimal plans.

5.3 Kismet Usage

In this case study, we also address four commonly asked usability issues about the Kismet tool.

How Sensitive Is Kismet to Changing Inputs? Since Kismet's analysis is dynamic, it can take advantage of information that can only be extracted by observing the runtime execution of the program. This allows Kismet to find opportunities for speedup that would be undiscovered by the more conservative analyses found in parallelizing compilers. The sensitivity of a potential speedup of a program to the input varies by the underlying algorithms in the program. Although Kismet could mirror parallelizing compilers and provide more "worst-case" speedup estimates, this fails to expose the opportunities that might be available in taking advantage of input-dependent parallelization strategies. As a result, our recommended usage model is that the user run Kismet on the application multiple times, across a spectrum of representative inputs, in order to gain a deeper knowledge of this issue.

What Tasks are Performed by the User from Program to Program? Our expectation is that the maintainer of Kismet would "ship" Kismet with a library of representative machine models and planners. When the user runs Kismet, they would select via commandline parameter the machine model which most closely matches the target architecture. Thus, from the user's perspective, the tool is "push-button." Although this is clearly future work, we have also envisioned the possibility of using auto-tuner techniques (i.e. as in FFTW) to automatically calibrate these components to a new architecture, which alleviates the Kismet maintainers of the need to update the library. Finally, as last resort, the user could extend the machine model and planner library themselves.

What is Kismet's Utility in Providing Refactoring Assistance? To be clear, Kismet does not try to make specific recommendations about how the programmer should refactor the program. Rather, it provides advanced information that helps the programmer decide a) whether it may not be worth the effort to parallelize the piece of code and b) what kind of speedup might be reasonable to aim for. The latter item may also influence the programmer's choice of transformations, but only in an indirect fashion. Although Kismet's speedup upperbounds are indeed approximate, our results show that they are rarely exceeded by actual parallelized code. A consistently low estimated speedup upperbound is a strong signal to the user that attaining speedup of the existing serial program is likely to be very challenging.

What is Kismet's Benefit Over Parallelizing Compilers? Kismet derives its key advantages over parallelizing compilers through an extension of CPA, which is a dynamic analysis not commonly used in today's parallelizing compilers. Speedup estimates provided by Kismet are likely to be higher than those attainable by a parallelizing compiler, because they are determined by empirical measurements about program parallelism rather than the ability of an automatic tool to prove properties about the program. Kismet's optimistic view of speedup attempts to take into account the programmer's greater ability to perform code transforms that would be unsafe in automatic parallelizing compilers.

6. Experimental Results

This section evaluates Kismet as follows. We first outline our evaluation methodology, including our selection of benchmarks and target machines. Using this methodology, we then quantify Kismet's accuracy by comparing both predicted and measured speedups from parallelization of three benchmark suites on three machine classes. Finally, we analyze the impact of novel techniques featured in Kismet: expressible self-parallelism, cache-aware prediction, and summarization.

6.1 Methodology

Kismet's goal is to provide realistic upper bounds on the parallel performance of serial programs. Our results will therefore focus on examining the tightness of these upper bounds on a wide range of benchmarks on several different platforms, both real and theoretical.

In our evaluation, we worked hard to address threats to validity by evaluating Kismet's performance across three very different architectures and by comparing against third-party parallelized codes from three benchmark suites, including both low and high parallelism applications.

We selected benchmarks using two primary criteria. First, the set of benchmarks needed to display a range of parallelism: from super-linear speedup down to very limited speedup. Second, the benchmarks needed to have either 1) a parallel implementation that could be used to gather real results or 2) published performance results from a variety of sources. Programs that are highly parallel tend to have a parallel implementation available while those with low amount of parallelism tend not to have parallel implementations available, possibly for reasons of vanity.

The selected benchmarks came from three benchmark suites, each targeting a different platform. Here we overview these suites, describing the amount and types of parallelism available and describing the steps necessary to obtain our results.

- **Raw.** We modeled RawCC's ILP exploitation on Raw as described in Section 5. Kismet's estimates are compared against speedup numbers reported in [35]. These benchmarks range from non-scalable to scalable.

  As mentioned before, RawCC utilizes loop unrolling to increase the amount of ILP. Unrolling also enables serial optimizations such as constant propagation and common sub-expression elimination. To control for these factors during profiling, Kismet uses LLVM to unroll the loops before static instrumentation.

- **SpecInt2000.** SpecInt2000 benchmarks are widely known to have extremely limited parallelism. Luckily, a wide range of proposed parallelization systems—especially those using speculative parallelization—have attempted to parallelize these benchmarks, providing a fertile source of published results. We chose to examine the benchmarks from this suite that have most frequently been the target of parallelization, namely bzip2, gzip, mcf, twolf, and vpr.

  In general, these benchmarks are hard to parallelize due to complex dependence patterns in DOACROSS loops. The speedup numbers reported in literature typically required heroic code transformations, and often involved special speculative hardware support or simulation-only experiments [24, 43, 44, 59, 60]. To approximate the machine models in those aggressive scenarios, we modified the Multicore-OpenMP model described in Section 5 so that it allows the exploitation of both DOALL and DOACROSS with zero parallelization overhead. Even with these permissive settings, Kismet is able to create strong bounds.

- **NAS Parallel Bench (NPB).** In contrast to SpecInt2000, NPB [9] generally consists of benchmarks with large amounts of easy-to-exploit parallelism. We use the Multicore-OpenMP predictor targeting only DOALL parallelism with parameters for a 64-core system. We measured speedup with third-party parallelized version [2] of NPB, running these parallel versions on the 32-core AMD system described in Table 1. For all NPB benchmarks, we used the 'A' input data set during both profiling and execution of the parallel versions.

What are "Correct" Speedup Predictions? In our evaluation, we employ benchmarks that were parallelized by third-party experts. To the extent that the benchmarks have been widely used in the research community, we have a reasonable expectation that these parallelization efforts are not too far off from optimal. To us, "correctly predict" means 1) that the actual speedup did not exceed the predicted speedup upperbound (i.e., Kismet's results correspond to actual empirical upperbounds) and 2) that the speedup experienced is close to Kismet's predictions (i.e. Kismet provides relative tight bounds.) To the extent that Kismet's bounds are not tight, it could be either due to insufficient modeling of machine constraints, or that there is remaining attainable speedup in the application.

6.2 Prediction Results

Raw Figure 6 shows predicted and measured speedup on RAW. In all benchmarks, Kismet correctly predicts the speedup trend in both high parallelism benchmarks [8]

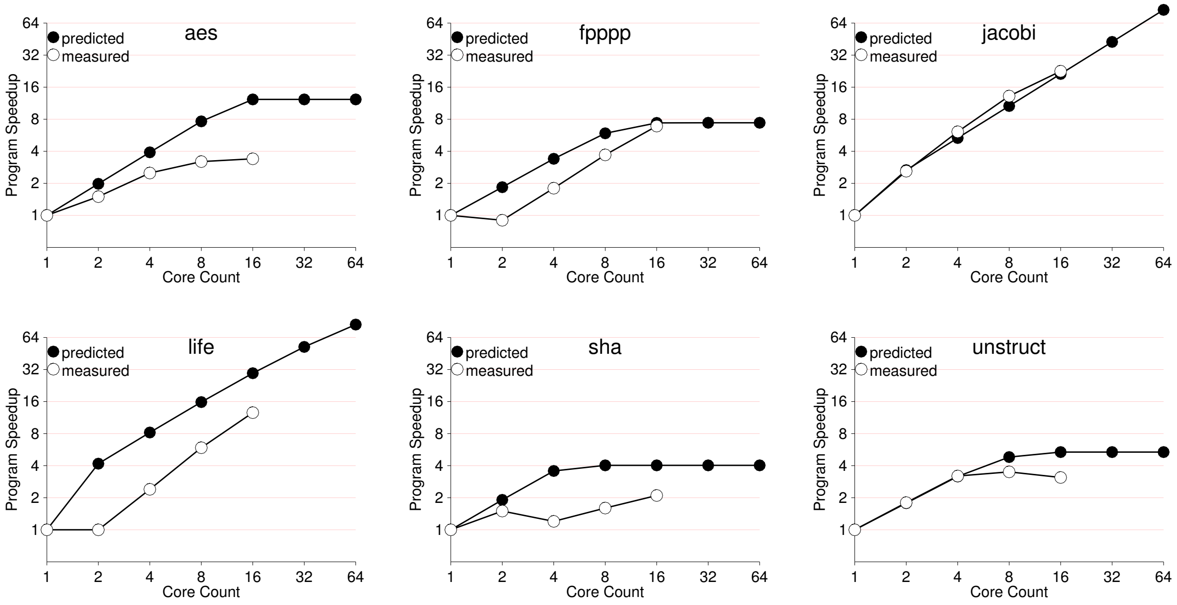


Figure 6: **Predicted and Measured Speedup for RAW Benchmarks on RAW hardware.** Kismet models the MIT Raw processor and RawCC, targeting the exploitation of ILP. From low- to high-parallelism benchmarks, Kismet provided appropriate upper bounds. This successful speedup prediction results from Kismet's ability to isolate ILP from other forms of parallelism based on summarizing hierarchical critical path analysis.

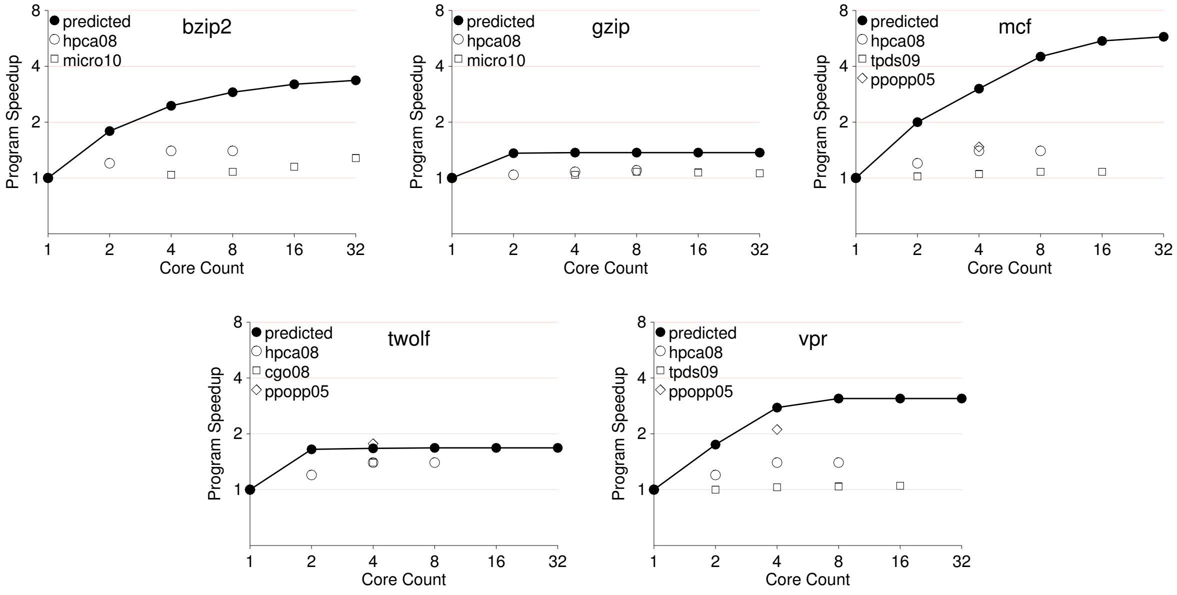


Figure 7: **Predicted and Reported Speedup in Low-Parallelism SpecInt2000 Benchmarks using third-party published results.** Kismet correctly captures the low parallelism in SpecInt2000 benchmarks, providing tight speedup upper bounds. Reported speedup numbers are from multiple sources that applied aggressive hardware/software techniques to extract parallelism from these benchmarks [24, 43, 44, 59, 60]. To model those experimental systems, Kismet is configured to exploit loop-level parallelism (DOALL and DOACROSS) with zero overhead. Kismet's expressible self-parallelism (ESP) and parallel execution time model enables similar modeling for a wide range of systems.

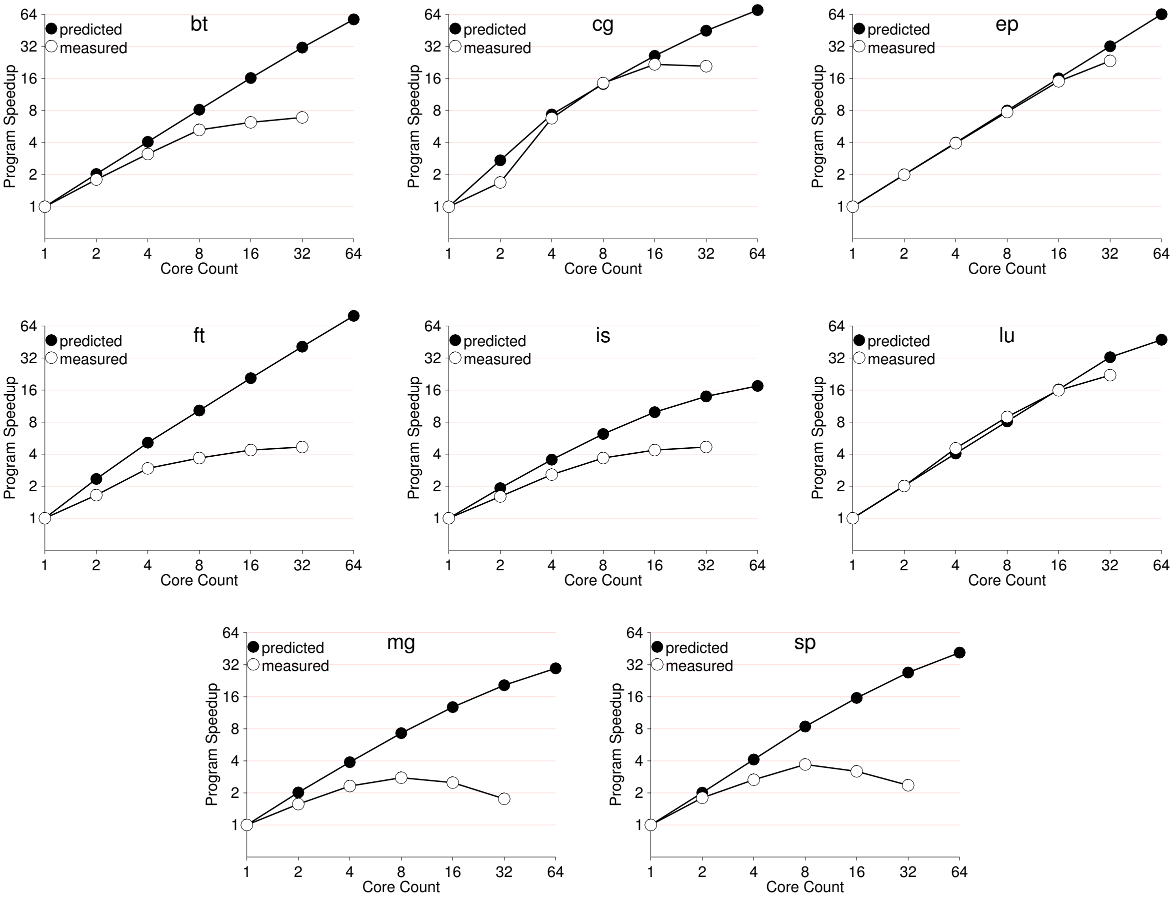


Figure 8: **Estimated and Measured Speedup of NAS Parallel Bench on 32-core AMD Multi-core System.** The NAS benchmarks are generally much higher parallelism than the other benchmarks considered in the paper. Kismet's cache-aware prediction is able to bound the speedup of the benchmarks reasonably well, including for the cg benchmark which observed superlinear speedup due to cache effects. The limited scalability of memory system becomes the bottleneck in several benchmarks when using 16+ cores.

(jacobi, life) and low parallelism benchmarks (aes, fpppp, sha, unstruct).

Super-linear speedup is predicted and measured in both jacobi and life but only the former had actual super-linear speedup. These benchmarks consist mainly of DOALL loops, allowing unrolling to linearly increase the amount of ILP. In contrast, unstruct also benefits from unrolling and serial optimizations, but its loops are DOACROSS, limiting unrolling's effects and limiting scalability. For the remaining benchmarks, aes, fpppp, and sha, unrolling was ineffective as the parallelized regions were functions rather than loops.

Kismet correctly bounded the speedup for all benchmarks except jacobi, which slightly outperformed Kismet's estimates. This anomaly can be attributed to the fact that including more cores from the Raw processor increases the number of registers, leading to decreased memory system delays; Kismet did not incorporate this effect into its basic estimation model as its effect is generally negligible.

SpecInt2000 Figure 7 shows Kismet's speedup estimates and speedup numbers gathered from third-party efforts running on aggressive hypothetical hardware [24, 43, 44, 59, 60]. These results confirm the generally-held belief that SpecInt benchmarks are fundamentally limited in their parallelism. Kismet predicted low speedups, plateauing at a speedup of 2 to 4 for all benchmarks except mcf. The reported results conform to Kismet's upper bounds.

NAS Parallel Bench (NPB) Figure 8 shows predicted and measured speedups for the benchmarks in NPB. As expected,—based on the abundant, easily-exploitable DOALL parallelism of these benchmarks—Kismet estimated relatively high speedups in all benchmarks except is. The lower amount of speedup in is results from it having only a limited amount of execution spent in parallel regions.

For ep and lu, measured speedup was very close to predicted speedup. Even though the communication cost on multicore processors typically limit the scalability of bench-

Benchmark		Estimated Speedup		
Suite	Name	Without ESP	With ESP	Ratio
RAW	jacobi	8649	53.81	160.7X
	life	26840	153.73	174.6X
	sha	4.81	4.71	1.0X
	fpppp	1190	98.74	12.1X
	aes	39547	150.95	262.0X
	unstruct	4416	8.22	537.2X
SpecInt2000	bzip2	17.4	3.39	5.1X
	gzip	4.27	1.37	3.1X
	mcf	67.12	5.92	11.3X
	twolf	11.35	1.68	6.8X
	vpr	15.77	3.1	5.1X
NPB	bt	161650	64.46	2507.8X
	cg	275	171.06	1.6X
	ep	93.69	38.67	2.4X
	ft	10709	151.92	70.5X
	is	565	37.53	15.1X
	lu	43845	52.98	827.6X
	mg	2478	87.35	28.4X
	sp	147873	65.18	2268.7X
Total	mean	23592	61	**363.2X**
	geomean	878	25	**34.5X**

Table 2: **Estimated Speedup with and without Expressible Self-Parallelism.** ESP helps the tightening of speedup estimates by providing only expressible parallelism to Kismet. In these benchmarks, ESP successfully reduced the speedup estimates by 363.2X, showing that it is indeed a central component in speedup estimation.

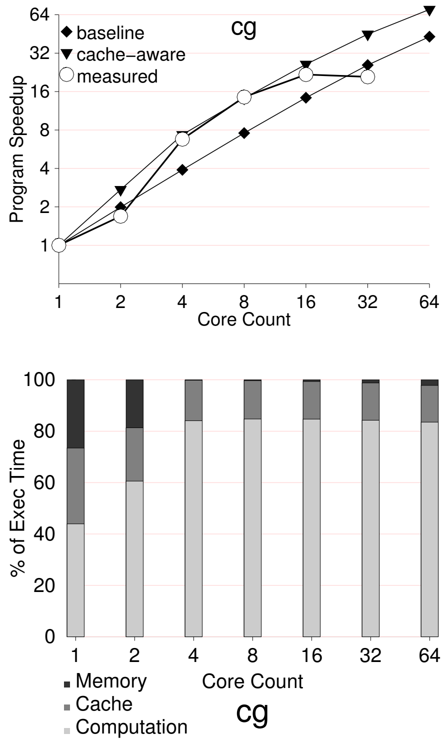


Figure 9: **Impact of Cache-aware Estimation in cg Benchmark.** The baseline estimation fails to predict the super-linear speedup of cg. By incorporating potentially reduced cache miss rates in a parallel execution, cache-aware estimation successfully predicts the super-linear speedup. Execution time breakdown clearly shows the time spent in cache and memory is considerably reduced from two-core to four-core execution.

marks, these benchmarks' speedup continued to scale as they do not rely on inter-core communication.

cg is an interesting benchmark that exhibits super-linear speedup in both predicted and measured speedup, thanks to Kismet's cache-aware performance model. We will examine cg in more detail later in the results section.

mg and sp scale up to 8 cores, but their speedup starts to decrease from that point. The drop in performance can be attributed to shared-memory related overhead that is not captured by Kismet's parallel execution time model. These benchmarks share data across cores and a data location is written by multiple cores, greatly increasing the sharing overhead. The gap between predicted and measured performance in these benchmarks might be closed when innovations in parallel computer architecture reduce the cost of shared-memory based communication. Alternately, more advanced modeling of coherence traffic in Kismet could be of assistance.

6.3 Impact of Expressible Self-Parallelism (ESP)

One of HCPA's major advantages over traditional CPA is its ability to localize parallelism using the self-parallelism metric. Kismet further improves the utility of self-parallelism by introducing the concept of expressible self-parallelism (ESP), a filtering step that removes self-parallelism that is unexpressible by the target system. To quantify the impact of ESP, we compared the estimated speedup with and without ESP in all benchmarks. We assumed zero overhead and infinite cores in the speedup estimation, in order to isolate the impact from ESP from other speedup limiting factors.

Table 2 shows the estimated speedup number with and without ESP. By honoring only unexpressible parallelism, Kismet tightens the speedup upper bound by up to 2508X, with an average reduction in speedup of 363.2X. The results confirm that ESP is an essential part in speedup estimation system.

6.4 Impact of Cache-aware Speedup Estimation

Cache-aware time estimation model incorporates potentially reduced cache service time caused by increased cache sizes when additional cores are used in execution. The top part of Figure 9 demonstrates the effectiveness of cache-aware estimation shown on the cg benchmark. Without cache-awareness Kismet predicts linear speedup, but measured speedup exhibits super-linear speedup. In cache-aware prediction, Kismet incorporates varying cache miss rates gathered from Cachegrind [41] for each core configuration, correctly predicting super-linear speedup of cg.

The lower part of Figure 9 shows the breakdown of execution time on different number of cores. As the core count switches from one to two and from two to four, the portion of cache and memory service time is significantly reduced. When the cache miss rate does not change, the portion for cache and memory should remain the same. Indeed, switching from 1 to 4 cores, L1 cache miss rate drops from 23.3% to 6.5%, and the last level cache miss rate drops from 6% to 0.1%.

6.5 Effectiveness of the Summarization Technique

To examine the effectiveness of Kismet's summarization technique, we ran NPB and SpecInt2000 benchmarks [1] with two different input sizes ('S' and 'A' for NPB, 'test' and 'ref' for SpecInt2000) and examined dynamic region counts as well as output file sizes. Figure 3 shows the results.

The results show that Kismet's summarization technique scales well with increasing input sizes and is effective at reducing the output file size. As expected, the dynamic region count significantly increases when we switch from small input to larger input – 463X on average. With the larger input sets, dynamic region profile data runs as large as several terabytes, clearly too large to be conveniently stored to disk. With SHCPA, there is virtually no difference in the output file size between small and large input sets. Moreover, the summarization technique results in very modest file sizes – only 85KB on average.

7. Related Work

This section examines Kismet's related work according to four themes: parallelism profiling, performance prediction, parallel performance debugging, and optimizations for reducing memory and execution overheads of dynamic program analyses.

Parallelism Profiling Approaches for parallelism-related profiling have generally fallen into two categories: critical path analysis and dependence testing.

Critical path analysis (CPA) dates back several decades, with early important works including Kumar and Austin [7, 28]. CPA approaches seek to measure the number of concurrent operations at each time step along the critical path of the program. In contrast to these approaches, Kismet's hierarchical critical path analysis is able to localize parallelism within nested program regions, and provide concrete guidance on which program regions to target. Recently, Kulkarni et al [27] used a critical path based analysis to bring insight into the parallelism inherent in the execution of irregular algorithms. In contrast to Kismet's focus on estimating speedup in 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. Other works have used CPA to perform limit studies for processors that target instruction-level parallelism (ILP) [29, 52].

Bench	Dynamic Region Count (Mega Regions)			Output File Size (Kilo Byte)		
Input	S	L	Ratio	S	L	Ratio
bt	4	2665	666×	102	102	1.0×
cg	38	830	22×	15	15	1.0×
ep	50	805	16×	4	4	1.0×
ft	40	1526	38×	50	50	1.0×
is	0.7	104	149×	3	3	1.0×
lu	2	2208	1104×	45	45	1.0×
mg	2	969	485×	79	79	1.0×
sp	10	7452	745×	166	167	1.0×
bzip2	846	4086	5×	62	63	1.0×
gzip	141	4477	32×	96	137	1.4×
mcf	7.8	4758	595×	19	20	1.1×
twolf	11.4	23023	2093×	260	309	1.2×
vpr	42.1	3020	72×	104	107	1.0×
mean	92	4302	**463**×	77	85	**1.1**×

Table 3: **Impact of Summarization Technique on File Size in NPB.** Switching from the small (S) to large (L) inputs causes 463× more dynamic regions to execute on average, but the output file size increases only 1.1× on average, from 77KB to 85 KB. Thus, the summarization technique is very effective in keeping output file size manageable even with large inputs.

Dependence testing is another parallelism profiling approach that strives to uncover the dependencies between different regions in the program. pp [30] is an early important work that proposed hierarchical dependence testing to estimate the parallelism in loop nests. Similar techniques are used in Alchemist [54] and Prospector [25]. Although dependence testing and Kismet's HCPA share similar goals, HCPA 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 if significant code transformation would be required to

[1] Raw benchmarks have only a single input set.

exploit it. Dependence testing is generally more pessimistic and sensitive to existing program structure.

Performance Prediction CilkView [18] and Intel Parallel Advisor's Suitability Tool [1] are recent tools whose motivation is similar to Kismet. Like Kismet, they also predict parallel performance on a target with arbitrary number of cores. Unlike Kismet, however, CilkView and Parallel Advisor rely on the user's parallelized code—or annotations—to predict speedup. Kismet minimizes user's efforts in prediction by automatically detecting parallelism in the serial program.

Simulation has been used to predict the performance of processors and systems that are still in development. In this case, a parallel version of the program exists, but the machine itself is not available to run it. ManySim [56] is one such simulator that was designed to evaluate the performance potential and scalability of large-scale multicore processors. GEMS [37] is a full-system functional simulator for multiprocessors. It separates the simulation from the timing models, allowing them build a detailed memory system timing simulator rather than focus on basic functional simulation. However, simulators still require code that has been parallelized for these systems, unlike Kismet.

A number of works have looked at the limits of parallelism and their impact on performance. Theobald et al [51] examined the "smoothability" of a program's parallelism, i.e. the ability to which a program's parallelism could be equally spread throughout the program's entire execution to ensure high utilization on a constrained multiprocessor. Rauchwerger et al [45] also looked at the ability to map ideal parallelism to a constrained processor, introducing the concept of *slack* to describe the ability of parallelism to be pushed to later parts of the program. Kismet improves upon these works by using HCPA's ability to localize parallelism; Kismet can examine the effect of parallelizing specific regions of the program in order to gain a better estimate of the program's parallel performance.

There have been several efforts to predict serial performance [20, 23, 34, 42]. In theory, these predictions could be combined with Kismet's speedup predictions to predict the parallel execution time of a program.

Several works have looked at predicting the scalability of parallel programs based on their performance on a small number of processors [10, 53]. Barnes et al [10] looked at several techniques for extrapolating performance of MPI programs, including one that measured the global critical path. Zhai et al [53] avoid performance extrapolation to predict performance; instead, they use deterministic replay to measure sequential time of each process using only a single node. Again, these systems differ from Kismet in that they predict performance based on an existing parallel implementation.

Hill and Marty [19] recently proposed a simple performance analytical model, extending Amdahl's law. Their model assumes future processors include different types of cores and each program region can choose the more appropriate core based on its workload. Chung and Mai [12] further improved Hill and Marty's model with heterogeneous chip including ASIC, FPGA, and GPU. Although we kept Kismet's analytical model relatively simple, Kismet can easily incorporate these sophisticated models if needed.

Parallel Performance Debugging Tools Several systems have been developed in order to help debug the performance of pre-existing parallel programs [3, 13, 39]. SvPablo provided an integrated viewing and instrumentation environment that allowed performance debugging of MPI programs. Adve et al [3] performed similar analysis on data parallel FORTRAN. Paradyn [39] automatically searches for performance problems in long running programs by dynamically instrumenting the program. Martonosi et al [38] were able to examine the performance of the cache system with very little overhead by integrating performance monitoring into existing cache-coherence mechanisms. These systems could be used in concert with Kismet to help determine why actual performance does not match the predicted bound on program performance. SUIF Explorer [33] uses static and dynamic analyses to understand parallel-execution related properties, much like Kismet; however, Kismet does not require user interaction, and uses a simplify hardware specifications to give reasonable speedup predictions of post-parallelized code.

Reducing Dynamic Program Analysis Overheads Dynamic program analyses often have huge memory and storage requirements as they can produce data for each dynamic instruction in a program that easily could run billions or trillions of instructions. To alleviate the severe memory requirements of dynamic program analysis, compression techniques have been used in whole program analysis [55], dependence analysis [26], and HCPA [16]. Initially Kismet used a compression technique similar to [16], but we found that handling more irregular programs like SpecInt necessitated the creation of Kismet's summarization-based HCPA variant, SHCPA.

In addition to memory overhead, runtime overhead is also important for practical use. Specifically for program analysis that uses shadow memory, the implementation of shadow memory significantly impacts the overall runtime as each load and store instruction will access the shadow memory. Valgrind [41]'s shadow memory implementation is described in [40]. Umbra [58] and EMS64 [57] proposed efficient shadow memory implementation for 64-bit address space, exploiting the sparse usage of memory space in 64-bit systems and cached shadow memory. Although techniques introduced in these papers can be incorporated in Kismet, Kismet's shadow memory implementation differs from other tools as it needs to efficiently store and retrieve multiple timestamps for each memory address to track the critical path of multiple region levels.

Prior HCPA-Based Work Kismet extends the HCPA region hierarchy proposed in [22] and [16] to include *sequence regions*. Sequence regions allow HCPA-based planners to separate ILP from other classes of parallelism. This is important for modeling performance of both superscalar-based out-of-order systems, where the ILP is likely already exploited by the base core (and thus is paradoxically unexpressible as far as the parallel programming is concerned), and also Raw-like systems with ILP compilers, where ILP is the primary source of parallelism. Sequence regions enable the implement of Kismet's expressible self-parallelism (ESP) metric, which allows Kismet to filter regions that have parallelism unexpressible by the target platform, which greatly enhances accuracy. Kismet also adds a cache-aware execution time model and models the effects of loop unrolling; both of these enhancements enable prediction of super-linear speedup. Kismet introduces a new HCPA variant with region summarization, SHCPA, which is important for handling irregular applications for which trace compression is not effective.

8. Conclusion

This paper presents Kismet, a tool that estimates the parallel speedup of serial programs. Kismet automatically localizes the parallelism available throughout nested program regions and combines this with user-specified constraints to provide approximate upper bounds for the parallel speedup attainable on a specified system. Our preliminary results on 19 benchmarks and two classes of machines (AMD Opteron and a tiled processor) demonstrate Kismet's effectiveness at providing accurate upper bounds across diverse programs and machine architectures.

Acknowledgment

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

References

[1] "Intel Parallel Advisor 2011." `http://software.intel.com/en-us/articles/intel-parallel-advisor`.

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

[3] V. Adve, J. Mellor-Crummey, M. Anderson, J.-C. Wang, D. A. Reed, and K. Kennedy. "An integrated compilation and performance analysis environment for data parallel programs." In *SC '95: Proceedings of the ACM/IEEE conference on Supercomputing*, 1995.

[4] A. Agarwal, S. Amarasinghe, R. Barua, M. Frank, W. Lee, V. Sarkar, D. Srikrishna, and M. Taylor. "The RAW compiler project." In *Proceedings of the Second SUIF Compiler Workshop*, 1997.

[5] G. Ammons, T. Ball, and J. R. Larus. "Exploiting hardware performance counters with flow and context sensitive profiling." In *PLDI '97: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1997.

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

[7] T. Austin, and G. S. Sohi. "Dynamic dependency analysis of ordinary programs." In *ISCA '92: Proceedings of the International Symposium on Computer Architecture*, 1992.

[8] J. Babb, M. Frank, V. Lee, E. Waingold, R. Barua, M. Taylor, J. Kim, S. Devabhaktuni, and A. Agarwal. "The raw benchmark suite: computation structures for general purpose computing." In *FCCM '97: Proceedings of the IEEE Symposium on FPGA-Based Custom Computing Machines*, 1997.

[9] Bailey et al. "The NAS parallel benchmarks." In *SC '91: Proceedings of the Conference on Supercomputing*, 1991.

[10] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. de Supinski, and M. Schulz. "A regression-based approach to scalability prediction." In *ICS '08: Proceedings of the International Conference on Supercomputing*, 2008.

[11] J. M. Bull, and D. O'Neill. "A microbenchmark suite for OpenMP 2.0." *SIGARCH Computer Architecture News*, Dec 2001.

[12] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai. "Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus?" In *MICRO '10: Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, 2010.

[13] L. De Rose, and D. Reed. "Svpablo: A multi-language architecture-independent performance analysis system." In *ICPP '99:International Conference on Parallel Processing*, 1999.

[14] E. Waingold et al. "Baring It All to Software: Raw Machines." *IEEE Computer*, Sept 1997.

[15] S. Garcia, D. Jeon, C. Louie, S. Kota Venkata, and M. B. Taylor. "Bridging the parallelization gap: Automating parallelism discovery and planning." In *HotPar '10: Proceedings of the USENIX workshop on Hot Topics in Parallelism*, 2010.

[16] S. Garcia, D. Jeon, C. Louie, and M. B. Taylor. "Kremlin: Rethinking and rebooting gprof for the multicore age." In *PLDI '11: Proceedings of the Conference on Programming Language Design and Implementation*, 2011.

[17] N. Goulding, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, J. Babb, M. Taylor, and S. Swanson. "GreenDroid: A Mobile Application Processor for a Future of Dark Silicon." In *Hotchips*, 2010.

[18] Y. He, C. Leiserson, and W. Leiserson. "The Cilkview Scalability Analyzer." In *SPAA '10: Proceedings of the Symposium on Parallelism in Algorithms and Architectures*, 2010.

[19] M. D. Hill, and M. R. Marty. "Amdahl's law in the multicore era." *IEEE Computer*, July 2008.

[20] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. De Bosschere. "Performance prediction based on inherent program similarity." In *PACT '06: Parallel Architectures and Compilation Techniques*, 2006.

[21] D. Jeon, S. Garcia, C. Louie, S. Kota Venkata, and M. B. Taylor. "Kremlin: Like gprof, but for Parallelization." In *PPoPP '11: Principles and Practice of Parallel Programming*, 2011.

[22] D. Jeon, S. Garcia, C. Louie, and M. B. Taylor. "Parkour: Parallel speedup estimates for serial programs." In *HotPar '11: Proceedings of the USENIX workshop on Hot Topics in Parallelism*, May 2011.

[23] T. S. Karkhanis, and J. E. Smith. "A first-order superscalar processor model." In *ISCA '04: Proceedings of the International Symposium on Computer Architecture*.

[24] H. Kim, A. Raman, F. Liu, J. W. Lee, and D. I. August. "Scalable speculative parallelization on commodity clusters." In *MICRO '10: Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, 2010.

[25] M. Kim, H. Kim, and C. Luk. "Prospector: A dynamic data-dependence profiler to help parallel programming." In *HotPar '10: Proceedings of the USENIX workshop on Hot Topics in parallelism*, 2010.

[26] M. Kim, H. Kim, and C.-K. Luk. "SD3: A scalable approach to dynamic data-dependence profiling." *MICRO '10: Proceedings of the International Symposium on Microarchitecture*, 2010.

[27] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Casçaval. "How much parallelism is there in irregular applications?" In *PPoPP '09: Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2009.

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

[29] M. S. Lam, and R. P. Wilson. "Limits of control flow on parallelism." In *ISCA*, 1992.

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

[31] C. Lattner, and V. Adve. "LLVM: A compilation framework for lifelong program analysis & transformation." In *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*, 2004.

[32] 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 '98: International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct 1998.

[33] S.-W. Liao, A. Diwan, R. P. Bosch, Jr., A. Ghuloum, and M. S. Lam. "SUIF Explorer: an interactive and interprocedural parallelizer." In *PPoPP '99: Proceedings of the ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*.

[34] G. Loh. "A time-stamping algorithm for efficient performance estimation of superscalar processors." In *SIGMETRICS*, 2001.

[35] M. B. Taylor et al. "Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams." In *ISCA '04: Proceedings of the International Symposium on Computer Architecture*, Jun 2004.

[36] M. B. Taylor et al. "The Raw Microprocessor: A Computation Fabric for Software Circuits and General-Purpose Programs." In *IEEE Micro*, Mar/Apr 2002.

[37] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. R. Alameldeen, K. Moore, M. Hill, and D. Wood. "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset." *SIGARCH Comput. Archit. News*, Nov 2005.

[38] M. Martonosi, D. Felt, and M. Heinrich. "Integrating performance monitoring and communication in parallel computers." In *SIGMETRICS*, 1996.

[39] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. "The Paradyn Parallel Performance Measurement Tool." *IEEE Computer*, 1995.

[40] N. Nethercote, and J. Seward. "How to shadow every byte of memory used by a program." In *VEE '07: Proceedings of the 3rd international conference on Virtual Execution Environments*, 2007.

[41] N. Nethercote, and J. Seward. "Valgrind: A framework for heavyweight dynamic binary instrumentation." In *PLDI '07: Proceedings of the Conference on Programming Language Design and Implementation*, 2007.

[42] D. Ofelt, and J. L. Hennessy. "Efficient performance prediction for modern microprocessors." In *SIGMETRICS*, 2000.

[43] M. K. Prabhu, and K. Olukotun. "Exposing speculative thread parallelism in spec2000." In *PPoPP '05: Proceedings of the ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, 2005.

[44] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August. "Parallel-stage decoupled software pipelining." In *CGO '08: Proceedings of the International Symposium on Code Generation and Optimization*, 2008.

[45] L. Rauchwerger, P. K. Dubey, and R. Nair. "Measuring limits of parallelism and characterizing its vulnerability to resource constraints." In *MICRO '93: Proceedings of the international symposium on Microarchitecture*, 1993.

[46] S. Bell et al. "TILE64 - Processor: A 64-Core SoC with Mesh Interconnect." In *ISSCC '08: IEEE Solid-State Circuits Conference*, 2008.

[47] N. R. Tallent, and J. M. Mellor Crummey. "Effective performance measurement and analysis of multithreaded applications." In *PPoPP '09: Proceedings of the ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2009.

[48] M. B. Taylor. *Design Decisions in the Implementation of a Raw Architecture Workstation*. Master's thesis, Massachusetts Institute of Technology, Sept 1999.

[49] M. B. Taylor. *Tiled Microprocessors*. Ph.D. thesis, Massachusetts Institute of Technology, 2007.

[50] M. B. Taylor, W. Lee, S. P. Amarasinghe, and A. Agarwal. "Scalar operand networks." *IEEE Transactions on Parallel and Distributed Systems*, Feb 2005.

[51] K. B. Theobald, G. R. Gao, and L. J. Hendren. "On the limits of program parallelism and its smoothability." In *MICRO '92:*

Proceedings of the International Symposium on Microarchitecture, 1992.

[52] D. W. Wall. “Limits of instruction-level parallelism.” In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[53] J. Zhai, W. Chen, and W. Zheng. “Phantom: predicting performance of parallel applications on large-scale parallel machines using a single node.” In *PPoPP ’10: Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2010.

[54] X. Zhang, A. Navabi, and S. Jagannathan. “Alchemist: A transparent dependence distance profiling infrastructure.” In *CGO ’09: Proceedings of the International Symposium on Code Generation and Optimization*, 2009.

[55] Y. Zhang, and R. Gupta. “Timestamped whole program path representation and its applications.” In *PLDI ’01: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2001.

[56] L. Zhao, R. Iyer, J. Moses, R. lllikkal, S. Makineni, and D. Newell. “Exploring Large-Scale CMP Architectures Using ManySim.” *IEEE Micro*, July 2007.

[57] Q. Zhao, D. Bruening, and S. Amarasinghe. “Efficient memory shadowing for 64-bit architectures.” In *ISMM ’10: Proceedings of the International Symposium on Memory Management*, Jun 2010.

[58] Q. Zhao, D. Bruening, and S. Amarasinghe. “Umbra: Efficient and scalable memory shadowing.” In *CGO ’10: Proceedings of the IEEE/ACM international symposium on Code Generation and Optimization*, 2010.

[59] H. Zhong, M. Mehrara, S. Lieberman, and S. Mahlke. “Uncovering hidden loop level parallelism in sequential applications.” In *HPCA ’08: Proceedings of the International Symposium on High Performance Computer Architecture*, 2008.

[60] D. A. Zier, and B. Lee. “Performance evaluation of dynamic speculative multithreading with the cascadia architecture.” *IEEE Transactions on Parallel and Distributed Systems*, Jan 2010.