SPARTAN: A <u>S</u>oftware Tool for <u>Par</u>allelization Bo<u>t</u>tleneck <u>An</u>alysis

Mayank Agarwal, Matthew I. Frank
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
{magarwa2,mif}@illinois.edu

Abstract

The multicore era has brought to an end the trend of doubling single-thread performance with each generation of microprocessors. To continue scaling programmers must parallelize existing single-threaded applications. A lack of tools makes identifying program features and data-dependence relationships that bottleneck parallel performance a cumbersome ad-hoc task.

This paper presents SPARTAN, a tool that analyzes single-threaded applications, and points out the data-dependences that are likely to pose bottlenecks to parallel performance. In addition SPARTAN ranks the bottlenecks by giving an estimate of their impact on performance, so that these can be fixed in an appropriate order. We validate SPARTAN by showing improved parallel speedup on a benchmark application when the identified bottlenecks are removed.

1 Introduction

The emergence of multicore architectures has halted the trend of increasing single-thread performance with successive generations of microprocessors. Current architectures are optimized for throughput or power-performance rather than raw single-thread performance [11].

Therefore to attain higher performance levels, existing single-threaded applications need to be parallelized to utilize the multiple available cores. This can be a daunting task. The prevalent (and manually intensive) approach is to understand the program in enough detail to decide how to partition it into threads. The next (and even harder) step is performance debugging. A major challenge in this step comes from inter-thread data-dependences that pose bottlenecks to parallel performance in unforeseen (and usually unintended) ways. Performance debugging requires time and effort to weed out such bottlenecks. The lack of tools to aid in these two steps makes parallelizing single-threaded applications a very time-consuming and ad-hoc process.

This paper argues for a more structured methodology to the parallelization process. We have developed two tools to aid in the process:

- a **Thread Partitioner** tool: This tool identifies a good partitioning of the application into threads, using models of both application parallelism and architectural communication and synchronization costs.
- a **Parallelization Bottleneck Analyzer (SPARTAN)** tool: This tool identifies the data-dependence relationships in the application that are likely to constrain parallelization, and therefore pose bottlenecks to parallel performance. The tool also quantifies the estimated performance benefit of alleviating a particular bottleneck (or of specified combinations).

We developed the Thread Partitioner tool in previous work [4, 2]. This paper describes the key ideas involved in the development of the SPARTAN tool. SPARTAN is a software tool built on top of a trace generator. It analyzes a run of a single-threaded application and lists the data-dependences that will pose performance bottlenecks when the application is parallelized. Internally the tool performs an abstract dependence height analysis on the application trace. This allows it to identify limiting data-dependences independently of any particular thread partitioning of the application. Next, SPARTAN breaks down the program critical path to isolate the top bottleneck data-dependences in the program. SPARTAN can also estimate the improvement in parallel performance potential from removing a particular bottleneck dependence (or a combination of dependences).

Next, this paper presents the results of running SPARTAN on a set of single-threaded benchmark programs from the SPEC2000 suite [1]. We divide the most limiting bottleneck data-dependences into two classes:

- Essential dependences: These are related to the choice of high-level algorithm used by the application. Parallelization would require a major rewrite.
- Accidental dependences: These come from the specific way in which an otherwise parallelizable task was coded, perhaps because the programmer was unaware of, or didn't originally care about, the impact on parallelizability.

We find a surprisingly large number of accidental dependences in our chosen applications, including calls to several library functions such as random number generation, memory allocation, garbage collection, etc. This trend shows encouraging ways forward by using parallel libraries.

Finally, the paper demonstrates that removing the bottlenecks identified by SPARTAN actually improves parallel performance of the benchmark applications consistent with the tool's predictions. For this purpose, we implicitly parallelize a benchmark application on a speculative parallelization system. We find that the dependences identified by SPARTAN are indeed bottlenecks on this system, and that removing these bottlenecks can substantially improve the measured parallel performance.

2 Background

SPARTAN is built around two key concepts: first, it does an abstract dependence height study [13] to estimate the potential of parallelizing an application(section 2.1); and second it does a critical path analysis [8] to identify the bottlenecks posed by data-dependences in parallelizing the application (sections 2.2 and 2.3).

2.1 Abstract Dependence Height Analysis

An abstract dependence height study is a useful tool to get an upper bound on the performance potential of parallelizing the application. One such study was carried out by Lam and Wilson [13]. Their study relaxes constraints on program execution imposed by von Neumann architectures, by allowing multiple flows to execute different (control-independent) parts of the programs simultaneously. Within a flow, branches can be speculated upon. However, a mispredicted branch must be executed before instructions control-dependent upon it can execute.[1] Only true control- and data-dependences are enforced in their study.[2]

There are latencies associated with the execution of instructions which constrains the earliest execution time of instructions data-dependent upon them. This leads to a completion time of the program. This is the absolute minimum time required for completing the program in its current form given the execution latencies, since the control- and data-dependences must be respected in any execution. In particular, the completion time is decided by the longest chain of control- and data-dependences in the program, and we refer to its length as the dependence height of the program.

The dependence height of the program is the absolute best any parallelization of the program could achieve. This is because the control-independent flows represent a superset of the threads that could be simultaneously executing instruction in parallel. On the other hand, the above study is hugely optimistic in many ways, including allowing for an infinite fetch bandwidth to each flow, and the ability to buffer an infinite number of instructions in each flow. This is something that cannot be achieved by a real thread unit.

We can inject a bit of realism in the study by imposing some constraints on individual flows, which correspond to the constraints imposed on a real thread unit (limited fetch bandwidth, buffer constraints, etc). But we still allow for infinitely many flows to be executing simultaneously. This is because limiting the number of flows would require us to make an optimal resource allocation decision to calculate the upside potential, which is a hard problem. In addition, we also don't impose any cost to flow creation or inter-flow data communication. Thus, we obtain the benefits of each flow without incurring the costs seen in a real system. Again, finding the upside potential that includes these costs would require us to make an optimal flow selection to balance the benefits and costs of flow creation, which is a hard problem.

In spite of the idealizations that remain, this version of the dependence height study is a more useful tool to understand the amount and nature of parallelism in the application, and the bottlenecks to further parallel performance. The next sections describe how to identify these bottlenecks. Note that for the rest of the paper, we will not differentiate between a thread and a flow for easier reading.

2.2 Dependence Graph of Execution

The execution of instructions on an out-of-order superscalar processor can be visualized using a dependence graph [8] built on a trace of program instructions. Each instruction is represented by a set of nodes in the graph: the "D" node which represents *dispatch*, address generation, fetch, decode and renaming; the "E" node which represents (out-of-order) issue and *execution* of the instruction and the "C" node which represents *commit*.

The graph also has edges that represent dependences between nodes. True **data-dependences** are captured through EE edges, from the E node of producer to that of consumer instructions. Other edges model **microarchitectural dependences**. An instruction needs to be dispatched before it can execute, and execute before being committed. Thus, within each instruction's nodes there is a DE edge and an EC edge. A superscalar processor fetches and retires instructions in-order, so DD and CC edges flow between successive instructions. The processor's reorder buffer can contain only N instructions so fetch must be stalled whenever there are more than N uncommitted instructions. This leads to CD edges from each instruction i's commit to fetch node of instruction i+N. Finally there are **dynamic dependences**, such as that imposed by a mispredicted branch. The incorrect prediction is detected upon branch execution, and the

[1]This corresponds to the SP-CD-MF configuration described in the Lam-Wilson study.

[2]Lam and Wilson did not enforce reassociatable data-dependences such as those due to loop index variables. SPARTAN does enforce these dependences and identifies them as bottlenecks if they have an impact on performance.

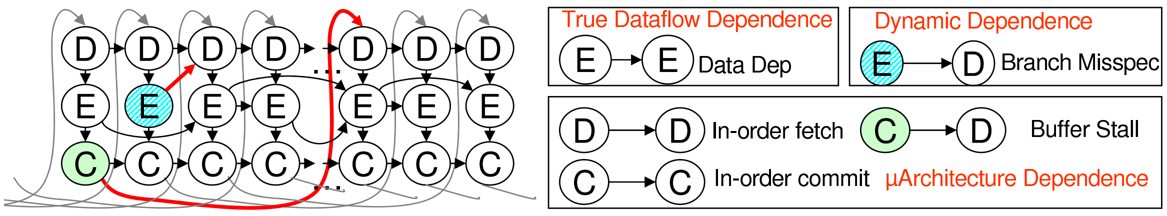

Figure 1: An example of a Dependence Graph of Program Execution [8].

machine rolls back state and restarts fetching from the correct target. This is represented by an ED edge from the E node of the branch to the D node of the succeeding (correct) instruction. Figure 1 gives an example graph.

Edges in the graph are labeled with the delay imposed by the dependence. For example, EE edges are labeled with the latency due to issue contention and functional unit latency.

In previous work [4], we have shown how to construct a dependence graph for a situation similar to the abstract dependence height study of section 2.1.

2.3 Program Critical Path

The **program critical path** is the longest path from the *start node* (D node of the first instruction) to the *end node* (C node of the last instruction) in the program dependence graph. This path can be computed can be done in two passes of the dependence graph [8].

The program critical path decides the running time of the program. In addition, edges that lie on the critical path have a zero slack, that is, increasing the latency of any such edge will delay the completion of the program by the same amount. Another way to interpret the situation is that knowing the dependences that lie on the critical path can help us identify bottlenecks to performance.

3 Design of SPARTAN

3.1 Functionality

SPARTAN works in two modes: bottleneck identification mode, and bottleneck quantification mode. In the identification mode, the tool takes in a single-threaded application, and outputs a list of the data-dependences that it believes will pose important bottlenecks to parallel performance, ranked by their relative importance. In the quantification mode, it takes in, along with the single-threaded application, a data-dependence (most likely identified in the identification phase) or a list of such dependences, and outputs the expected improvement in parallel performance potential when the bottleneck posed by that dependence is removed. Thus repeated queries to SPARTAN can help identify the order in which bottlenecks should be removed.

3.2 Bottleneck Identification

SPARTAN is a software tool built on top of a trace generator. Figure 2 illustrates the steps involved in bottleneck

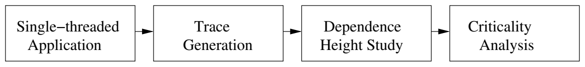

Figure 2: Bottleneck identification steps.

identification process. We describe these steps in detail.

The trace generation step outputs the trace for the single-threaded application running on a given input. This step can use a trace generator similar to Intel's PIN [14]. This trace is passed on to the dependence height analysis step, which assigns a timestamp to each instruction for its dispatch (D), execution (E) and commit (C), following the techniques of trace-based simulation of Wall [19] and modeling the constraints of the dependence height study described in section 2.1. As described earlier, the dependence height study assigns timestamps to each instruction to estimate the potential of the best-case parallelization of the application in its current form. For the next steps, we analyze chunks of trace at a time for good performance.

These timestamps can then be used in the construction of the program dependence graph for the dependence height study. This construction is done using the concepts of section 2.2. Once the dependence graph has been constructed, SPARTAN computes the longest path from the *start node* of the graph to its *end node*. This path represents the *program critical path*. Of particular interest are the data-dependences on the critical path that cross flow boundaries. These represent a superset of dependences that could pose a bottleneck to parallel performance, since the flows represent a superset of threads that could be created. The tool records all such dependences. At the end, SPARTAN outputs the list of such observed critical data-dependences, sorted by the number of times we see a particular dependence appear on the critical path through the trace.

Note that SPARTAN's data-dependence list is not tied to any particular selection of threads to parallelize the application. This makes it particularly suited for refactoring a single-threaded application for which we do not yet know an appropriate thread partitioning.

SPARTAN could be modified to identify the bottleneck dependences for a specific choice of threads. Our previous work shows how to do the timestamp assignment for a given thread selection [4].

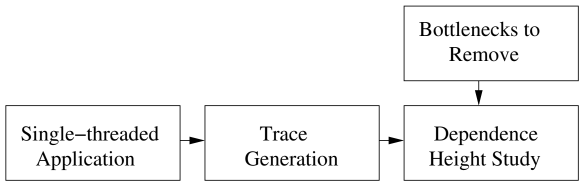

Figure 3: Bottleneck quantification steps.

Table 1: SPEC Benchmarks chosen

Benchmark	Description
175.vpr	FPGA Circuit Placement and Routing
300.twolf	Place and Route Simulator
197.parser	Word Processing
164.gzip	Compression

Not also that instruction criticality analysis is typically used to identify architectural bottlenecks, such as cache ports, branch prediction, etc. SPARTAN is a novel use of the critical path analysis to learn about the application structure and data-dependences.

3.3 Bottleneck Quantification

The bottleneck quantification mode allows SPARTAN to estimate the improvement in parallel performance potential from removing a particular bottleneck dependence (or a combination of dependences). The steps involved are illustrated in figure 3. The trace generation step proceeds as before. In this mode, however, the programmer provides a list of the data-dependences that are to be ignored. This information is incorporated in the dependence height analysis, where the specified dependence constraints are not enforced. That is, the consumers of these dependences are allowed to execute even before the specified producers have completed execution (other dependences still need to be satisfied for the consumers). The result of this study gives the upside potential of parallelization when the specified bottleneck is removed.

4 Results of Identifying Bottlenecks

This section describes the results of running SPARTAN on a set of single-threaded benchmark applications from the SPEC2000 suite, and gives some insights about the type of bottlenecks we have observed. For the purpose of this study, we analyzed the following SPEC Integer applications: vpr.place, twolf, parser and gzip. They are all single-threaded C applications. Table 1 gives brief descriptions. We run these benchmarks on the Minnesota reduced input sets [10] for a representative interval of 100M instructions. Next we present the output from running SPARTAN on these applications, along with descriptions and code snippets to illustrate the nature of bottlenecks.

Table 2: Bottleneck dependences in VPR Place sorted by the count of the number of times a particular dependence appeared on the critical path.

PC→PC	From	To	Count
2688→263c	my_irand	my_irand	32611
9604→9604	try_swap	try_swap	26129
2710→263c	my_frand	my_irand	25048
9604→967c	try_swap	try_swap	13360
26bc→92cc	my_irand	try_swap	12729
d39c→95fc	net_cost	try_swap	11679
92cc→9330	try_swap	try_swap	11272
2688→26d4	my_irand	my_frand	10775

```
int my_irand(int imax) {
  /* Create random integer between 0 and imax */
  int ival;
  current_random = current_random * IA + IC;
  ival = current_random & (IM-1); /* Modulus */
  ival = (int)((float)ival * (float)(imax+0.999) /
             (float)IM);
  return (ival);
}
```

Figure 4: VPR Place Random number generator with bottleneck dependence highlighted.

4.1 VPR Place

Table 2 lists the output of running SPARTAN on VPR Place. The tool identifies the address of the producer and consumer instructions of the critical dependences. The tool annotates this information with the function names of these instructions. This information can also be used to trace back the lines in the code causing this dependence.

For VPR Place, we find, quite surprisingly, that a major hindrance to parallelization comes from the dependence between successive calls to random number generator functions: `my_irand` and `my_frand`, the integer and floating point versions. These account for 3 of the top 8 bottlenecks.

Figure 4 illustrates the code that causes this problem. We see that the variable `current_random` is first read, and then written in each call to `my_irand` (and also in `my_frand`). This forces a sequentialization of successive executions of the function, which becomes a bottleneck to parallel performance.

4.2 Twolf

The story in Twolf is quite similar to that of VPR Place. Table 3 lists the top bottlenecks identified by SPARTAN. As with VPR Place, random number generation is a major bottleneck to parallelization. In addition, we investigated the bottleneck in the functions `new_dbox` and `new_dbox_a`. These functions are nearly identical, as is the bottleneck, which comes from an integer reduction illustrated in figure 5. The variable `costptr` is an integer location that is read and written in each iteration of a doubly nested loop.

Table 3: Bottleneck dependences in Twolf sorted by the count of the number of times a particular dependence appeared on the critical path.

PC→PC	From	To	Count
1cfa4→1cf54	Yacm_random	Yacm_random	70238
aaf8→ ab0c	new_dbox	new_dbox	28795
9dc8→ 9ddc	new_dbox_a	new_dbox_a	23161
1cfa4→1cfc4	Yacm_random	Yacm_random	8031
ab10→ aaf8	new_dbox	new_dbox	7687
9de0→ 9dc8	new_dbox_a	new_dbox_a	5908

```
new_dbox(antrmptr, costptr)
int *costptr;
{
  for (termptr=atnrmptr;
       termptr;
       termptr=termptr->nextterm){
    ...
    for (netptr=dimptr->netptr;
         netptr;
         netptr=netptr->nterm){
      oldx = netptr->xpos;
      if (netptr->flag == 1) {
        ...
      } else {
        ...
      }
      *costptr += ABS(newx-new_mean) -
                  ABS(oldx-old_mean);
    }
  }
  return;
}
```

Figure 5: new_dbox **function in Twolf highlighting the bottleneck integer reduction.**

This prevents parallelization of both the inner and the outer loop. This variable can easily be reassociated with large improvements in the potential.

4.3 Parser

Table 4: Bottleneck dependences in Parser sorted by the count of the number of times a particular dependence appeared on the critical path.

From	To	Count
xfree	xfree	462577
xfree	xfree	21548
build_clause	build_clause	7776
free_disjuncts	free_disjuncts	5652
right_table_search	right_table_search	4322
power_prune	power_prune	4223

Parser highlights another type of bottleneck dependence, arising from the allocation and freeing up of memory in programs. This benchmark has its own memory management functions: xalloc and xfree, and we find that the top two dependences (where the third bottleneck is very distant) come from the dependence within the function xfree. Again, there is a dependence between successive calls to xfree which prevents any meaningful parallelization of the application.

4.4 Gzip

Table 5: Bottleneck dependences in Gzip sorted by the count of the number of times a particular dependence appeared on the critical path.

PC→PC	From	To	Count
b3f8→b400	updcrc	updcrc	766811
b3ec→b3f4	updcrc	updcrc	766811
b5d8→b5e0	flush_window	flush_window	425562
b5cc→b5d4	flush_window	flush_window	425553
10ec→110c	deflate	deflate	216305
10fc→1110	deflate	deflate	215441
1110→1118	deflate	deflate	215305
110c→10c8	deflate	deflate	209197

```
/* Run a set of bytes through the crc
   shift register. */
ulg updcrc(s, n) {
  /* temporary variable */
  register ulg c;
  /* shift reg contents */
  static ulg crc = (ulg)0xffffffffL;
  if (s == NULL) {
    c = 0xffffffffL;
  } else {
    c = crc;
    if (n) do {
      c = crc_32_tab[((int)c ^ (*s++)) & 0xff] ^
          (c >> 8);
    } while (--n);
  }
  crc = c;
  return c ^ 0xffffffffL;
}
```

Figure 6: updcrc **function in Gzip highlighting the sequential nature of algorithm.**

Finally, we find a different class of dependences that limit parallelization in Gzip, listed in table 6. These dependences have more to do with the choice of the algorithm, which is inherently sequential in nature, and less with the choice of library functions or implementation details. We illustrate this through the updcrc function in figure 6. The crc variable introduces sequentialization between successive calls to updcrc and parallelizing this requires domain expertise and knowledge of the algorithm used.

4.5 Discussion

We can broadly classify the observed bottleneck dependences into two categories:

Essential dependences: These arise from the choice of high-level algorithm used that is inherently sequential in nature, such as the case of Gzip. Parallelization would require domain expertise. SPARTAN might be used as an aid to figure out the algorithms to redesign.

Accidental dependences: These are related to the specific way in which an otherwise parallelizable task was coded up, therefore constraining parallelization. Examples are random number generation in VPR and Twolf, memory allocation in parser, etc.

The latter case is the more interesting one, and is surprisingly frequent in our case studies. The fact that this is observed in the SPEC Integer benchmarks, that are traditionally believed to not be amenable to parallelization, makes it even more impressive. Encouragingly, the trend of accidental dependences from within library functions shows a clear path forward, in the form of parallel library versions that can be widely used to ease the task of the parallel programmer.

5 Quantifying Bottlenecks and Validation

In this section, we do a case study of the VPR Place application, and validate that the predictions of SPARTAN are relevant for an actual parallelization of the application. Due to the preliminary nature of the results and shortage of space, we plan to carry out this study on a larger number of applications as future work.

5.1 Quantifying Bottlenecks in VPR

We run SPARTAN in the bottleneck quantification mode for VPR and we get the following result (table 6):

Table 6: Impact of removing the bottleneck from random-number generation on upside potential of parallelization in VPR Place.

Upside potential in original form	Upside potential with bottleneck removed
8.3X	154.2X

The potential is shown as times speedup (X) over the performance (measured in instructions per cycle or IPC) on a single aggressive 4-wide out-of-order superscalar processor. We find a great increase, from 8.3X to 154.2X, almost a 20-fold improvement in the potential. This indicates that removing this dependence (perhaps through parallelizable or parallelized) random-number generation could lead to huge rewards.

5.2 Experimental Methodology

Next we try to test the applicability of the results from SPARTAN on an actual parallelized version of VPR Place. Rather than explicitly parallelizing VPR for this purpose, we instead carry out implicit parallelization of the application on a speculative parallelization system [9, 12, 17, 18]. The system that we use for this purpose models a multicore architecture with support for speculative parallelization. Each core can be running at the most one thread at a time. Threads are created in-order, once a thread has spawned another thread, it cannot create another thread in its lifetime.

The system imposes costs for inter-thread data communication (5 cycles) and dependence synchronization, thread initiation and completion (5 cycles). Inter-thread data-dependences can be trained by predictors or identified off line by a compiler analysis, and such dependences are synchronized. In our modeled system, synchronized consumer instructions must wait until producer (and all intermediate) threads have fetched all instructions and all branches correctly executed before it can start execution, which places a significant cost to synchronization. Other instructions that do not depend on earlier threads for data values can proceed in an out-of-order fashion.

Since the parallelization is (data) speculative in nature, processor state is regularly checkpointed and the system can do a roll-back if an inter-thread data-dependence relationship is found not to have been enforced in a timely manner. This mechanism ensures correctness, maintaining sequential semantics externally, thus making this a case of implicit parallelization. However, such a roll-back is costly, and the system tries to minimize such actions by using data-dependence predictors to identify and enforce inter-thread data-dependences.

5.3 Potential for Parallel Performance on Experimental System

Since our implicit parallelization system needs to buffer processor state until it can be committed to the system, this limits the scope of parallelization. In particular, the system cannot create arbitrarily large threads, since these would exceed the allowed buffer. Our evaluation system therefore allows threads that have been profiled to have an average length of at the most 1K instructions.

Table 7: On the evaluation system, impact of removing the bottleneck from random-number generation on potential of parallelization .

Upside potential in original form	Upside potential with bottleneck removed
7.0X	14.9X

This reduces the upside potential of application parallelization. We can do another dependence height study to estimate the new upside potential of parallelization under this constraint (table 7). We find that the upside potential drops from 8.3X to 7X when we restrict thread sizes to capture the constraint on our evaluation system. In addition, the impact of removing the bottleneck from random number generation is also considerably lower, from almost 20-folds down to 2-folds improvement. This points to the fact that removing the bottleneck especially benefits large threads where earlier they were of little use (due to this dependence).

This result also indicates a bottleneck from the architecture (buffer size). However, alleviating the architectural bottleneck is scope of future work. This study focusses on the application bottleneck from the data-dependence.

5.4 Speculatively Parallelization of VPR

Having obtained an idea of the upside potential of implicit parallelization from SPARTAN from removing the bottleneck, we carry out an actual parallelization on the evaluation multi-core system.

First we partition the application into threads. For this, we analyze the application on our *Thread Partitioner* tool. The tool takes the application and a list of the potential threads to be analyzed as input, and outputs a good thread selection (which is a subset of the input threads) for the underlying architecture after doing a cost-benefit analysis incorporating the cost of thread creation, synchronization, etc. For this study, we allowed the tool to choose from an exhaustive list of potential threads obtained by a compiler postdominance analysis. In previous work, we have shown that this subsumes thread choices from heuristics such as loops, procedures, and their continuations [3].

The output of the thread partitioner specifies how to break the application up into threads. We then measure the performance of the application in our evaluation environment. The system then internally parallelizes the application according to that thread selection, while giving the appearance of sequential execution externally. For this study, we don't constrain the number of cores available in the system, to allow a maximal exploitation of parallelism. Note that other costs and constraints are still imposed as described in section 5.2.

We get the following improvements in performance on the evaluation system:

Table 8: Measured performance on an implicit parallelization system, before and after removing the bottleneck in VPR Place.

Performance Improvement in original form	Performance Improvement with bottleneck removed
2.3X	4.5X

Note that when we remove the bottleneck, we generate a separate thread selection using *Thread Partitioner*, optimized for the new version of the application. Also the achieved improvements are significantly lower than the predicted upside potential. This is because the experimental system incurs significant cost from inter-thread data synchronization and other thread-related actions. Nevertheless, the achieved gains are quite encouraging, and we see almost a two-folds increase in performance from removing the bottleneck, quite similar to the gains predicted by SPARTAN.

We further explored this issue by identifying the top bottleneck dependences for our chosen implicit parallelization of VPR in its original form. We find that the top bottleneck dependence is indeed the same as identified by the tool, and listed in table 2. In fact, the top dependence lists match up quite well. This is indicative of two things. Firstly, our *thread partitioner* tool generates a selection that is optimized for the underlying architecture, and we get quite close to the application bottlenecks. Secondly, SPARTAN through an abstract dependence height analysis is able to make quite meaningful and useful predictions about the behavior of the application when parallelized on a real system.

6 Related Work

Static dependence analysis has been used to aid in parallelization of Fortran scientific applications. For example the Parascope [5] editor pointed out potential dependences or race conditions in a loop to be parallelized. In contrast SPARTAN analyzes dynamic traces allowing it to work on larger and more complex programs, and avoids the problems of imprecision of static pointer analysis.

Several studies have manually identified bottlenecks to implicit parallelization for several SPEC2000 applications [7, 16]. SPARTAN automates the detailed process of identifying bottlenecks and, in the cases where we have run SPARTAN on the programs used in those studies, SPARTAN finds a superset of the previously identified bottlenecks.

There has also been work on parallel library functions for the type of accidental dependences identified in this study. Work has been done on parallel memory allocation [6], parallel random number generation [15], etc.

In the sequential domain, there are large number of standardized tools for bottleneck analysis.

7 Conclusion

While parallelizing single-threaded applications is a relevant problem, there is a lack of tools to help in the task of parallelization and identification of performance bottlenecks. This paper describes SPARTAN, a tool that can identify bottlenecks to parallel performance, as well as quantify the importance of removing particular bottlenecks.

SPARTAN combines the concepts of abstract dependence height study and critical path analysis to identify the data-dependences that constrain program parallelization. Running the tool on benchmark applications finds two classes of bottleneck dependences: essential dependences, that arise from the choice of a sequential high-level algorithm; and accidental dependences, that are due to an implementation ill-suited to parallelization and could be easily fixed without requiring domain knowledge or significantly changing application behavior.

Validation of SPARTAN in an implicit parallelization environment for the VPR Place benchmark application shows that the bottlenecks identified by the tool actually constrain performance in this environment, and that removing the top bottleneck improves performance consistent with predictions.

Acknowledgements

We thank Steve Lumetta for valuable feedback on this work. The work reported in this paper was supported in part by the National Science Foundation under grant CCF-0746910 and by the Gigascale Systems Research Center, one of five research centers funded under the Semiconductor Research Corporation's Focus Center Research Program. Computational resources were provided by the Trusted ILLIAC Center at the Information Trust Institute and Coordinated Science Laboratory at the Univ. of Illinois.

References

[1] Spec CPU2000, 2001.

[2] M. Agarwal and M. I. Frank. Re-evaluating the potential of speculative parallelization. In *Currently under review*, 2009.

[3] M. Agarwal, K. Malik, K. M. Woley, S. S. Stone, and M. I. Frank. Exploiting postdominance for speculative parallelization. *Int'l Symp. High Performance Comp. Arch.*, (HPCA-13):295–305, 2007.

[4] M. Agarwal, N. Navale, K. Malik, and M. I. Frank. Fetch-criticality reduction through control independence. *Int'l Symp Comp Arch*, 35:13–24, 2008.

[5] V. Balasundaram, K. Kennedy, U. Kremer, K. McKinley, and J. Subhlok. The parascope editor: an interactive parallel programming tool. In *Supercomputing*, 1989.

[6] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. *Arch Support Prog Lang and Operating Sys*, ASPLOS-IX:117–128, 2000.

[7] M. J. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. I. August. Revisiting the sequential programming model for multi-core. In *MICRO 40*, 2007.

[8] B. Fields, S. Rubin, and R. Bodík. Focusing processor policies via critical-path prediction. *Int'l Symp Computer Architecture*, (ISCA-28):74–85, 2001.

[9] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun. The Stanford Hydra CMP. *IEEE Micro*, 20(2), 2000.

[10] A. KleinOsowski and D. Lilja. Minnespec: A new spec benchmark workload for simulation-based computer architecture research, 2002.

[11] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: a 32-way multithreaded sparc processor. *IEEE Micro*, 25, 2005.

[12] V. Krishnan and J. Torrellas. A chip multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers*, 47(9), September 1999.

[13] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. *Int'l. Symp. Comp. Arch.*, (ISCA-19), 1992.

[14] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Prog Lang Design and Impl*, (PLDI), June 2005.

[15] G. Marsaglia. Multiply-with-carry (mwc) generators, 1994. Included with dieHard package (http://stat.fsu.edu/pub/diehard/).

[16] M. K. Prabhu and K. Olukotun. Exposing speculative thread parallelism in spec2000. In *PPoPP '05: Principles and practice of parallel programming*, 2005.

[17] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. *Int'l Symp Computer Architecture*, (ISCA-22):414–425, 1995.

[18] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry. The stampede approach to thread-level speculation. *ACM Trans. Comput. Syst.*, 23(3):253–300, 2005.

[19] D. W. Wall. Limits of instruction-level parallelism. Research Report 93/6, DEC Western Research Laboratory, Nov. 1993.