Improving Hierarchical Critical Path Analysis Performance

Chris Louie

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

Abstract

Hierarchical critical path analysis is capable of localizing parallelism to varying granularities in a program, but is computationally expensive. Consequently, the memory and performance overheads are major impediments towards mainstream acceptance and usage.

This paper examines two techniques to allocate and utilize memory more efficiently and compiler analyses and optimizations targeted at improving the performance of hierarchical critical path analysis. On the presented benchmarks, these techniques improve memory and performance overheads as high as 1.95× and 1.62× and on average 1.27× and 1.29× respectively.

1 Introduction

The shift to multicore processors has driven programmers to convert their serial programs to parallel programs if they want to continue to observe performance gains with the coming generations of processors. However, programmers continue to struggle with parallelizing programs for performance due to the increased complexity and additional knowledge required about the parallelization platform and system. Consequently, we have developed tools to alleviate some of the burdens involved with parallelizing code. While the majority of parallelization tools focus on code generation and runtime management, programmers still rely on trial and error for parallelism discovery and planning. We attempt to fill this void by providing Kremlin [2] and Parkour [3] that localize parallelism and give tight parallel speedup upperbounds.

Both of these tools utilize a novel metric of *self parallelism* and *hierarchical critical path analysis* (HCPA) [2], an extension to critical path analysis [5] that overcomes some problems; however, this analysis is expensive computationally. Our initial implementation incurred execution time slowdown factors around 366× and memory overheads of 19×. In order to make these tools feasible in practice, these overheads must be reduced since only tiny to small programs can be analyzed.

This paper focuses explaining the techniques used to improve the performance and memory overheads of HCPA and contributes following:

- This paper present a mechanism to reduce the memory overhead incurred in the initial implementation of HCPA and simplify the usage model. The memory overheads involved in using HCPA is prohibitively large to run any modest program. This mechanism increases the range of programs HCPA can analyze.

- This paper introduces compiler analyses and optimizations to improve the performance of HCPA. In order to be practical, HCPA should add minimal performance overheads so that programs that already run too slowly on current ar-

chitectures can be profiled and analyzed in reasonable time.

- This paper quantifies the memory savings and performance improvements of each of these techniques.

The remainder of this paper is organized as follows. Section 2 provides background and the original technique used to calculate HCPA. Section 3 and Section 4 explain the optimizations made to reduce memory and performance overheads respectively. Section 5 presents the results and evaluation of the presented techniques. Finally, Section 6 concludes.

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

Figure 1: **Localizing Parallelism.** In the code snippet from the feature-tracking benchmark of SDVBS, only the inner loop iterating over k contains parallelism. Traditional critical path analysis would report that all loops contain parallelism since they contain the innermost.

2 Hierarchical Critical Path Analysis

In order to quantify parallelism within programs, programmers can use critical path analysis [5] or CPA. CPA begins by identifying the longest chain of operations through a program as the critical path. The critical path gives a lower bound on execution time of the program since all other operations can be executed in parallel with the critical path. The length of the critical path combined with work can also be used to calculate the average amount of parallelism in using the equation $p = work/length_{cp}$.

CPA provides a generic platform that we can characterize programs since it can produce these metrics independently of parallelization platform, architecture, programming language. Additionally, because CPA bases its analysis on dependencies of the operations, it outputs results that are invariant of the serial expression of the program. That is, the results do not change if independent sections of the program are reordered.

CPA's Shortcoming. Although CPA provides a generic platform to characterize and quantify the parallelism within a program, it has a major drawback when applying it to actually parallelizing a program. Since programs tend to execute in phases as explained in [8], the average parallelism metric provided by CPA does not give the programmer insight about the amount of parallelism exhibited by each portion of their program. Additionally, since programs exhibit a hierarchical structure due to function and loops, CPA is unable to distinguish parallelism from nested function calls and loops. To exemplify this limitation, Figure 1 shows a code snippet from the feature-tracking benchmark in the San Diego Vision Benchmark Suite [4]. In the code snippet, only the inner most loop contains parallelism, but traditional CPA would erroneously report that the outer loops also contain parallelism since they contain the innermost loop.

Localizing Parallelism. In order to overcome the limitations of CPA, we must be able to localize parallelism within a program. To accomplish this, we extend CPA into hierarchical critical path analysis (HCPA) [2]. We first divide the dynamic execution of the program into regions that represent portions of the execution of the program. We then continue to subdivide these regions into smaller subregions such that no sibling subregion overlaps and they are contained within their parent region. All of these parent-child relationships between the regions and their subregions form a region tree hierarchy representing the program at different granularities. Although alternative delineations of regions are possible, we choose

function calls and loops as regions because they form a hierarchical structure and they produce profiling information that corresponds to the static structure of programs.

We then run CPA within the scope of each region in the region tree to get its local critical path and work. For operands that are used before the region begins, we say the availability time of this operation is immediate (i.e. time 0) because they were created before that region's start.

From these two pieces of information from every region and the region tree, we are able to calculate the approximate parallelism contained in every region. The average parallelism of a leaf region in the tree is the work divided by the critical path length. For the non-leaf regions, Kremlin and Parkour use the metric of self parallelism [2] which quantifies the amount of parallelism in a region excluding the parallelism from their children. This novel metric enables us localize parallelism within a region and to estimate the speedup of parallelizing a region without having also to parallelize its children.

2.1 Implementation

This section describes the original implementation of HCPA.

Hierarchical Shadow Memory. Calculating the critical path dynamically is already a daunting task because any referenceable operation as the program executes could grow into being part of the critical path. Consequently, we must maintain these timestamps for every referenceable operation. In order to track the timestamps, we implement shadow memory similar to [7, 9].

We make two optimizations for allocating shadow memory. We first analyze functions statically to determine the amount of shadow memory needed for local operations and allocate it at the beginning of every dynamic instance. call. Static analysis also enables perfect hashing into the table so access is quicker than accessing non-local memory timestamps. The second optimization is that we evenly divide the address space using a two level page table and only allocate shadow memory as necessary for dynamic operations. We use calls to `malloc` and `free` to determine when we should allocate and free shadow memory in our page tables.

When extending to HCPA, we also must keep timestamps for every level of the hierarchy because the critical paths between each of these levels do not necessarily intersect. We store the timestamps of all of the active regions for a particular operation in a fixed-size array. Each index is associated with the depth of an active region. For example, `main` would occupy index 0 and all children of `main` would occupy index 1 and so on. The size of the arrays and the number of depths is configurable during compile time. This implementation enables for efficient reuse of memory since we reuse the memory of the last deallocated region for the new one.

Updating Timestamps. In order to perform updates, we begin by transforming the code into SSA format using LLVM[6] to eliminate any false dependencies. We then perform our analysis on each function. We scan through the instructions in the function and allocate an entry in the shadow register table for every operation. This entry will contain the availability time of that operation as calculated at runtime. After we allocate the space for all of our operands, we start adding in calls to runtime library functions to calculate the availability times of every operation dynamically.

The general rule for setting the timestamp of an operation is to take the maximum timestamp of all of the input operands and the control dependence and add the latency of the operation. We apply this rule to all unary and binary operations except for induction and reduction variables.

Induction and Reduction Variables. Induction and reduction variables create easy-to-break dependencies that are typically eliminated when parallelizing programs. As a result, we also model breaking these dependencies. We identify induction variables using LLVM's static analysis passes. We classify variables that only have commutative operations applied to themselves with other operations in loops as reduction variables. Instead of performing our normal

update rules for these timestamps, we set the timestamp for induction variables to the control dependence time. Reduction variables are not fully implemented, so we choose a loose upperbound of the constant cost for these operation.

Function Calls. For function arguments, we need to communicate the availability times to our callees. We add calls that will push the timestamps onto our virtual emulation of the runtime stack. Upon being called, callees will read the timestamps and associate them in their own shadow register table. However, callees may sometimes not find that their argument timestamps are available in the case they are called by uninstrmeneted code (e.g. start calling main or library callbacks). In this case, we assume that the timestamps of the arguments are available at time zero or the control dependence timestamp. We choose this implementation since uninstrmeneted code tends to be library code which the programmer cannot modify anyway and our function will only be called once all the operands are available.

Similarly for return values, we insert a call to a helper function that will set up a location to place the return value's timestamp before the call. The callee needs to fill in the timestamp of the return value. When the callee returns, we save the timestamp if available into the caller's shadow register table. If it is not available meaning we called an uninstrumented function, we assign its timestamp to be a fixed increment of the latest available argument. For the benchmarks that we ran, we had all of the sources except for the C standard runtime library, so we only did not have timestamps for the return values of these calls. Although this is not a completely accurate solution, we find that it models library calls well enough since they only can be called once they have all their operands ready and the functions normally called take a fixed amount of time.

Control Dependence. As part of calculating the availability time for every operand, we must consider control dependencies. For every basic block, we identify if it has any control dependence and subsequently, the timestamp of the control dependence value. The availability time of any operand in the block will be the maximum of any operand or timestamp of the control dependence plus the latency of the operation.

However, control dependence cannot be completely resolved statically in the case of function calls. All of the operations in these calls are also control dependent. To handle this case, we push the control dependence timestamp onto a stack and all operations within the block consider the control dependence timestamp from the top of the stack as a live-in timestamp. We can skip checking the remainder of the stack because the timestamps on the stack are monotonically increasing. Upon reaching the end of the basic block, we pop the control dependence timestamp off the stack.

ϕ Instructions. Static single assignment uses ϕ instructions to choose between values. The ϕ instruction takes one value per incoming basic block and chooses its value depending on how the block containing the ϕ was reached during dynamic execution. These ϕ instructions are used to implement conditional and loop constructs.

Since ϕ instructions only choose between values, they do not add to the critical path because they perform no work; however, their value is unknown until their control dependences resolve. Consequently, the timestamp of a ϕ instruction is the maximum of the incoming value and its control dependences. The control dependences for the ϕ instruction can be found anywhere from the basic block containing the ϕ instruction and the immediate dominators of the incoming values. Executing an immediate dominator indicates us that a particular value is incoming, so we need to add the control dependence that block.

In the case of loops, these immediate dominators of the incoming blocks could be the block containing the ϕ instruction, so the incoming value's condition may not necessarily dominate the ϕ instruction. In these cases, we add the condition as it becomes available.

3 Memory Reduction

The initial implementation of HCPA does not use memory completely efficiently and has a time consuming and error prone usage model. The initial version uses fixed-length arrays to store the timestamps for each level of the region tree. The length of the arrays are set during compilation and every dynamic timestamp instance allocates an array of this length in anticipation of the level being used, but if the level is never used, the memory is wasted. This waste is a serious problem since these timestamps for each level typically consumes over 95% of the instrumented program's total memory footprint. This implementation was chosen for ease of implementation and to reduce performance overheads associated with memory allocation.

Statically setting the maximal depth causes frustration for users. Since the program can continue to add levels dynamically, The maximal depth is unknown at compile time. In the case that the dynamic execution exceeds the maximal set level, the levels that exceed the maximum are not be profiled and our program produces no localized parallelism information about these deeper regions. This limitation is incredibly unfortunate for users since they only discover this void in their results after waiting hours for their run to complete. Consequently, users would like set a large maximal level depth, but this would result in potentially exceeding their available memory.

To improve the memory efficiency and usage model, we change our fixed-size arrays into dynamic length arrays that reallocate themselves lazily. Only upon trying to write to a particular timestamp, we allocate memory for the levels required. Using this method, timestamps that do not require the maximal depth avoid wasting space. This new implementation also eases the usage model because it eliminates one parameter the user needs to tune.

Although this technique could potentially cause a call to `realloc` after every operation, we find that this does not happen in practice and its performance quantified in Section 5.

4 Static Optimizations

The initial implementation of HCPA calculates the dynamic timestamp for all operations; however, all of these timestamp updates are not required and result in extra performance overheads. Since we calculated the timestamps for every SSA operation, the timestamps are typically only used once in the next operation.

We can make three large improvements over this original implementation. First, We can partially evaluate some of the timestamp calculation statically so we perform them once during compilation instead of many times during the dynamic execution of the program.

To explain the second and third improvements, we draw from liveness analysis terminology. We define the *live-out timestamps* as the timestamps used externally by others. We also define the *live-in timestamps* as the timestamps calculated by others for us.

We then can optimize our calculations by only calculating the dynamic timestamps for only live-out timestamps. This optimization vastly reduces the number of updates we need to accomplish since we do not need to perform the updates for the numerous intermediate SSA operations.

Third, we can statically produce one update rule per live-out timestamp as a function of the live-in timestamps it depends upon. To produce these functions, we repeat the following for each live-out value as determined by traditional liveness analysis.

We begin by transforming the original update rule for timestamp t_i which is the maximum of the operands plus the latency of the operation l_i by distributing the addition within the max function:

$$\begin{aligned} t_i &= \max(t_0, .., t_n) + l_i \\ &= \max(t_0 + l_i, .., t_n + l_i) \end{aligned}$$

We then recursively apply the update rules to its operand timestamps $t_0, .., t_n$:

$$= \max(\max(t_{00} + l_0, .., t_{0n} + l_0) + l_i, ..)$$

Again, we distribute the latencies within the max and use constant folding to eliminate an addition:

$$= \max(\max(t_{00} + l'_0, .., t_{0n} + l'_0), ..)$$

Finally, we flatten the nested max functions into a single one:

$$= \max(t_{00} + l'_0, .., t_{0n} + l'_0, ..)$$

After performing these transformations, the only remaining timestamps inside the max function will be the live-in timestamps because we cannot recurse to identify their timestamps statically.

This method also must respect control dependencies. Upon recursing on to each operand, we also look if the operation is control dependent. If it is, we add the control dependence timestamp into the aggregation of timestamps plus the latency of the current operation. This method of adding control dependence works except for operations that their control dependence is unknown statically. For these operations, we still build our stack of the control dependence timestamps and we root all live-in timestamps with the maximum of their timestamp or the timestamp of the last control dependence. Since we only add and take the maximum of the timestamps, control dependences will be respected because the final timestamp will be equal to or greater than the operands' timestamps.

In addition to respecting control dependences, we also must obey region boundaries. We cannot recurse past these boundaries because regions only calculate the localized critical path. Recursing beyond them will incorrectly include a portion of the critical path of their parent. Consequently, increasing the types of delineation between regions will decrease the effectiveness of this method. We find that the current demarcations of regions provides a balance between granularities and performance.

Areas for Improvement. The proposed algorithm for improving the performance can be improved further. Since the algorithm produces update rules for each live-out timestamp as a function of all of the live-in timestamp it requires, the algorithm can actually worsen performance with redundant calculations. Although it is not yet implemented, the performance overhead can be avoided if certain intermediate timestamps are dynamically calculated and saved. The intermediate timestamps that should be calculated dynamically are the ones that writing and subsequently, reading the intermediate timestamp multiple times costs less than repeatedly reading all the dependencies and calculating the live-in timestamps. These costs depend on memory system and computation overheads.

5 Results

In this section, we present the results of the memory and performance optimizations and evaluate their efficiency.

5.1 Experimental Setup

We use the initial implementation of HCPA used in Kremlin and Parkour as our baseline comparison of these techniques since they were built on top of it. We heavily optimized the baseline implementation by hand customizing each function, accessing memory sequentially instead of randomly, increasing cache reuse, and even using our custom function inliner. We run all eight programs in the NAS Parallel Benchmarks (NPB) [1] using the class S inputs. For gathering memory statistics, we run using each of the benchmarks through `massif` counting page references as memory use. For gathering timing results, we run each benchmark multiple times on idle nodes multiple times and take the minimum observed time.

5.2 Memory Reduction

In order to aggressively tune the baseline implementation, we performed the memory measurements using the minimal fixed array size required by the benchmark. Figure 2 shows the memory savings by allocating memory on demand. The amount of memory saved varies from benchmark due to their dynamic behavior. Benchmarks that tend to perform all of their operations near the maximal levels do not

Bench	Fixed (MB)	Lazy (MB)	Savings
bt	81.43	59.21	1.38
cg	160.40	82.37	1.95
ep	46.21	26.15	1.77
ft	641.50	493.40	1.30
is	46.72	24.19	1.93
lu	20.08	13.28	1.51
mg	642.10	592.20	1.08
sp	26.49	19.75	1.34
		mean	1.27

Figure 2: **Lazy Memory Allocation Savings.** Fixed shows the amount of memory used for each benchmark in NPB using the baseline implementation and Lazy shows the memory used when allocating on demand. In all cases, lazily allocating memory reduces memory overheads.

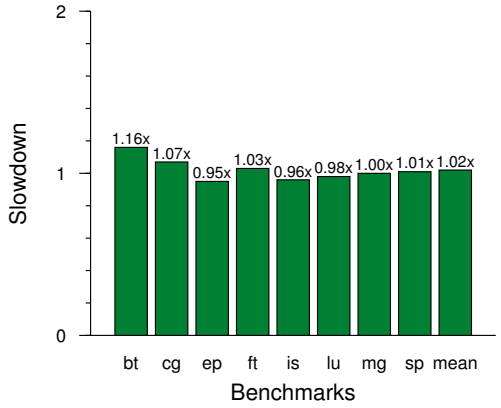

Figure 3: **Lazy Allocation Slowdown.** Lazily allocating memory for the vector timestamps hardly affects performance on all of the NPB programs. Surprisingly, some of the benchmarks observe marginal speedup.

benefit much from this technique since they all stay close to the fixed array size. Benchmarks that vary the number of active regions through their execution greatly benefit more from this technique because many of the timestamps do not require the largest allocation.

Figure 3 shows the effect of lazily allocating memory as needed on execution time. The performance impact is minimal in all cases since reallocations occur infrequently. For all of the benchmarks except bt, reallocations occur in less than 1% of the updates. bt incurs the worst slowdown of only 1.16× as it performs reallocations on 4.3% of its updates. Surprisingly, ep, is and lu actually observe speedup after making reducing memory. This is likely due to improved caching effects since lazily allocating memory removes the wasted space between the timestamp vectors. Consequently, cache lines can be filled with useful data instead of the unused padding.

5.3 Static Optimizations

Figure 4 shows the effects of partial evaluation, only calculating live-out timestamps and folding timestamp update rules into one. The variance in performance is likely to be caused by the size of basic blocks and the number of live-in timestamps. The optimizations perform better with longer basic blocks since intermediate values do not need to be calculated during every dynamic execution. Greater number of live-in timestamps degrades performance since it increases the number of live-in timestamps needed to be examined at runtime. All benchmarks observe speedup that demonstrates that the optimization causes less additional redundant work than operations saved. Additionally, the reported performance numbers are lower than expected since the dynamic timestamp calculations of the statically optimized version is not as heavily optimized as the baseline implementation.

6 Conclusion

HCPA appears to be a promising technique to localize parallelism and provide speedup upper bounds, but still needs to have minimal memory and computational overheads to be successful in practice. This paper presents two techniques to reduce these overheads and quantifies their results. The results demonstrate that the techniques can reduce memory and perfor-

Bench	Baseline(s)	Optimized(s)	Speedup
bt	393.376	340.527	1.16
cg	186.773	164.624	1.13
ep	949.825	587.188	1.62
ft	623.854	612.386	1.02
is	5.072	3.327	1.52
lu	110.489	75.121	1.47
mg	30.185	25.997	1.16
sp	172.123	138.393	1.24
		mean	1.29

Figure 4: **Static Optimization Performance.** Baseline and optimized show the execution time in seconds for each of the given benchmarks. Static optimizations improves performance over all benchmarks and still has room for further optimizations.

mance overheads on average by 1.27× and 1.29× on the presented benchmarks. Nevertheless, the implementation containing the optimizations still has room for improvement and hopefully will be useful in practice.

Acknowledgements

I would like to thank my advisor Michael Bedford Taylor for providing me the opportunity to work on novel research and his guidance throughout, Saturnino Garcia and Donghwan Jeon for their amazing work on the Kremlin project and Anshuman Gupta and Sravanthi Kota Venkata for their feedback through the years.

References

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

[2] S. Garcia, D. Jeon, C. Louie, and M. B. Taylor. "Kremlin: Rethinking and rebooting gprof for the multicore age." In *PLDI*, 2011.

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

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

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

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

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

[8] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. "Automatically characterizing large scale program behavior." *SIGOPS Oper. Syst. Rev.*, October 2002.

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