A Static Cut-off for Task Parallel Programs

Shintaro Iwasaki
Graduate School of Information Science and Technology
The University of Tokyo
Tokyo, Japan
iwasaki@eidos.ic.i.u-tokyo.ac.jp

Kenjiro Taura
Graduate School of Information Science and Technology
The University of Tokyo
Tokyo, Japan
tau@eidos.ic.i.u-tokyo.ac.jp

ABSTRACT

Task parallel models supporting dynamic and hierarchical parallelism are believed to offer a promising direction to achieving higher performance and programmability. Divide-and-conquer is the most frequently used idiom in task parallel models, which decomposes the problem instance into smaller ones until they become "trivial" to solve. However, it incurs a high tasking overhead if a task is created for each subproblem. In order to reduce this overhead, a "cut-off" is commonly used, which eliminates task creations where they are unlikely to be beneficial. The manual cut-off typically enlarges leaf tasks by stopping task creations when a subproblem becomes smaller than a threshold, and possibly transforms the enlarged leaf tasks into specialized versions for solving small instances (e.g., use loops instead of recursive calls); it duplicates the coding work and hinders productivity.

In this paper, we describe a compiler performing an effective cut-off method, called a static cut-off. Roughly speaking, it achieves the effect of manual cut-off, but automatically. The compiler tries to identify a condition in which the recursion stops within a constant number of steps and, when it is the case, eliminates task creations at compile time, which allows further compiler optimizations. Based on the termination condition analysis, two more optimization methods are developed to optimize the resulting leaf tasks in addition to replacing them with function calls; the first is to eliminate those function calls without exponential code growth; the second transforms the resulting leaf task into a loop, which further reduces the overhead and even promotes vectorization.

The evaluation shows that our proposed cut-off optimization obtained significant speedups of a geometric mean of 8.0x compared to the original ones.

Keywords

compilers; performance optimization; task parallelism; cut-off

PACT '16, September 11-15, 2016, Haifa, Israel
© 2016 ACM. ISBN 978-1-4503-4121-9/16/09…$15.00
DOI: http://dx.doi.org/10.1145/2967938.2967968

1. INTRODUCTION

Task parallel models, which support dynamic creation of fine-grained tasks and their dynamic load balancing, offer a promising prospect for programming processors with an increasing number of cores, as they achieve both productivity and scalability for a range of programs. One notable feature of task parallelism is hierarchical parallelism which is particularly suitable for divide-and-conquer algorithms. A number of parallel systems and libraries including widely-used ones such as Cilk [4], Intel Threading Building Blocks [27], and OpenMP [25] incorporate task parallelism.

While task parallel systems provide good programmability and scalability, naive task parallel divide-and-conquer programs creating tasks until the problems become literally "trivial" to solve suffer from a high runtime overhead of managing tasks. In order to reduce the overhead, it is necessary to enlarge the granularity of tasks by replacing creation of tasks with function calls. This practice is commonly referred to as a "cut-off" and is typically applied manually by the programmer.

Replacing task creations with function calls is often not enough to achieve high absolute performance; the resulting code still has recursive function calls to subproblems, rendering further compiler optimizations difficult to apply. Addressing this problem requires not only removing creation of tasks, but also aggressively optimizing the resulting code. Several studies focusing on an automatic cut-off have been proposed in the literature [3, 6, 34]. Nevertheless, to the best of our knowledge, most of them are *runtime* techniques, deciding whether to create a task or not by observations made at runtime (e.g., the number of tasks and/or processor utilization) and/or a depth from the root. Though the runtime techniques are effective in reducing a task creation overhead in many cases, they reveal limited opportunities for further *compile-time* optimizations. Moreover, runtime techniques have a risk of seriously reducing parallelism by stretching the critical path of computation, as we discuss in Section 5.

To tackle these problems, we propose an effective cut-off technique for divide-and-conquer task parallel programs based on a static analysis of a condition in which the recursion stops within a constant height from leaves. The obtained termination condition is useful not only to generate the cut-off threshold as humans do, but also perform further compile-time optimizations to the programs obtained after eliminating task creations, including inline expansion of function calls and transformations into vectorizable loops. In contrast to the runtime approaches, our static cut-off method has a smaller risk of adversely reducing parallelism,

```
void fib(int n, int* ret){
  if(n < 2){
    *ret = n;
  }else{
    int a, b;
    spawn fib(n-1, &a);
    spawn fib(n-2, &b);
    sync;
    *ret = a + b;
  }
}
```

Figure 1: Original task-parallel Fibonacci

because it guarantees that tasks applied the cut-off are within a constant height from leaves and thus are likely to be small (see Section 3.3). We implemented the algorithms as an optimization pass in LLVM [18].

This paper makes the following contributions:

- We propose a static cut-off technique based on an analysis on a termination condition of recursive task functions. The static cut-off requires no additional cost at runtime and guarantees the adequate parallelism of programs.
- We develop methods for further static optimizations on several types of task parallel programs based on the termination condition analysis, including inline expansion of subproblem calls in a code-bloat-free manner (*code-bloat-free inlining*), and transforming tasks into vectorizable loops (*task loopification*).
- We show the proposed static cut-off technique can be easily combined with runtime cut-off techniques, which are useful as a fallback strategy when none of the static optimizations applies.
- We evaluated performance of the proposal and show a substantial speedup of a geometric mean of 5.0x with the static cut-off, 6.2x with the code-bloat-free inlining, and 21x (up to 220x) with loopification over unoptimized task parallel programs. The performance obtained by the loopification was comparable to, or faster in some cases than the loop-based programs written in OpenMP with GCC and Polly [11].

The rest of this paper is organized as follows. Section 2 presents the overview of our method and the motivation of this research. Section 3 first describes the static analysis of the cut-off condition, and proposes static cut-off transformation composed of three cut-off methods based on the analysis: static task elimination, code-bloat-free inlining, and loopification. Section 4 shows the experimental results of the proposed optimizations. Section 5 discusses related work and Section 6 concludes this paper.

2. OVERVIEW

Before entering into details, this section overviews our cut-off methods using a simple running example. Consider a program in Figure 1, which calculates an nth Fibonacci number.

The original version suffers from a large overhead to create tasks due to its too fine-grained task granularity. A cut-off is a very common method to enlarge the granularity, but making the granularity too large has a risk of reducing parallelism significantly. As is commonly used in a manual cut-off, our compiler applies a cut-off to a function when the call tree rooted from it has a height within a constant

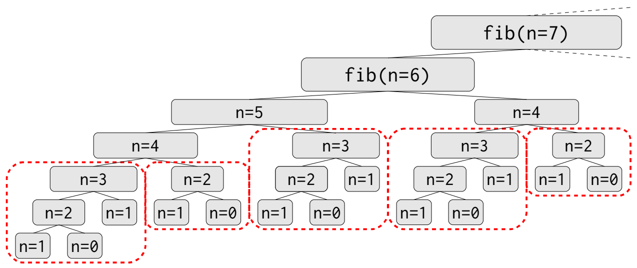

Figure 2: Task tree of `fib` in Figure 1. Tasks enclosed by a dotted line are the cut-off targets under the 2nd termination condition ($n < 4$).

threshold. With an appropriately chosen threshold, tasks under such conditions will perform a small amount of work, so serializing them will have little impact on the parallelism. Section 3.3 describes how we determine the cut-off threshold. The key of our static analysis is, therefore, to find a condition on function's arguments in which the height of the task tree from leaves to that task is within a constant height H. We call this condition the *Hth termination condition*. In particular, a 0th termination condition is a condition in which a function itself is a leaf and a 1st termination condition a condition in which a function is either a leaf or calls only leaves. Section 3.1 details how to obtain these conditions.

For example, Figure 2 shows cut-off targets (encircled by a dotted line) within a height 2 in the task tree of `fib`. The 2nd termination condition is $n < 4$.

If the compiler successfully identifies the termination condition, it tries to apply one of the transformations presented in Figure 3.

Figure 3a shows the simplest transformation which just replaces task creations with function calls. We call this optimization *static task elimination* (see Section 3.2.1). It reduces a tasking overhead while guaranteeing serialized tasks are small. The overhead still remains high for extremely fine-grained tasks such as `fib` due to the function calls, however.

A traditional method to alleviate function call overheads is inline expansion. It becomes powerful under an Hth termination condition since we can eliminate the innermost recursive calls by inlining H times. However, a usual inlining strategy does not work well in our setting. It is desirable to completely eliminate function calls within a coarsened task (`fib_seq` in Figure 3a), but doing so would increase code size significantly. Typical divide-and-conquer algorithms have multiple recursive call sites, so repeatedly expanding them increases code size exponentially with the depth of inline expansion. To completely inline-expand divide-and-conquer functions without exponential code growth, we develop a method to aggregate the recursive call sites into *a single static call site*. We call the method *code-bloat-free inlining* (see Section 3.2.2). The resulting code after transformation is shown in Figure 3b. Code-bloat-free inlining in general transforms recursive functions into nested loops as deeply nested as the depth of inline expansions.

There are cases in which we can do even better by transforming recursive functions into more natural, flat or shallowly nested loops. A divide-and-conquer vector addition program shown in Figure 3c is such a program which can be represented as a flat loop instead of a deeply nested loop obtained by *code-bloat-free inlining*. Our compiler is able to transform such a function to a flat loop as shown in Fig-

```
void fib(int n, int* ret){
  if(n < 4){
    fib_seq(n, ret);
  }else{
    int a, b;
    spawn fib(n-1, &a);
    spawn fib(n-2, &b);
    sync;
    *ret = a + b;
  }
}
void fib_seq(int n, int* ret){
  if(n < 2){
    *ret = n;
  }else{
    int a, b;
    fib_seq(n-1, &a);
    fib_seq(n-2, &b);
    *ret = a + b;
  }
}
```

(a) Static task elimination

```
void fib_seq(int n, int* ret){
  if(n < 2){
    *ret = n;
  }else{
    int a, b;
    for(int i = 0; i < 2; i++){
      int n2, *ret2;
      switch(i){
      case 0:
        n2=n-1; ret2=&a; break;
      case 1:
        n2=n-2; ret2=&b; break;
      }
      if(n2 < 2)
        *ret2 = n2;
      else
        [...]
    }
    *ret = a + b;
  }
}
```

(b) Code-bloat-free inlining

```
void vecadd(float* a,
            float* b, int n){
  if(n == 1){
    *a += *b;
  }else{
    spawn vecadd(a, b, n/2);
    spawn vecadd(a+n/2, b+n/2,
                 n-n/2);
    sync;
  }
}
```

(c) Task-parallel vector addition

```
void vecadd_seq(float* a,
                float* b, int n){
  for(int i = 0; i < n; i++)
    *(a+i) += *(b+i);
}
```

(d) Loopification

Figure 3: Overview of our methods

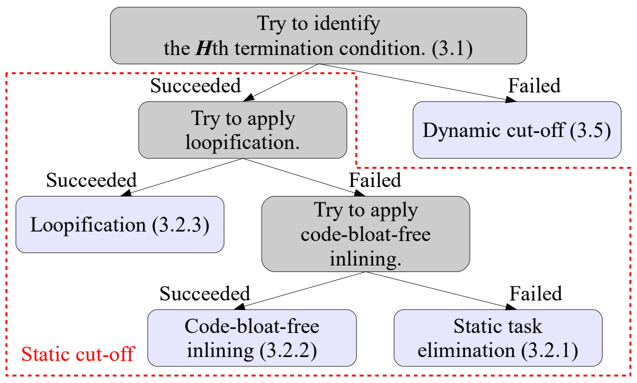
Try to identify
the *H*th termination condition. (3.1)
Succeeded
Failed
Try to apply loopification.
Dynamic cut-off (3.5)
Succeeded
Failed
Loopification (3.2.3)
Try to apply code-bloat-free inlining.
Succeeded
Failed
Code-bloat-free inlining (3.2.2)
Static task elimination (3.2.1)
Static cut-off

Figure 4: Flow of optimization selection

ure 3d, which then may be vectorized by the backend code generator. We call this optimization *loopification* (see Section 3.2.3). Loopification succeeds typically when the original program is a dense array-based computation such as stencil kernels and dense matrix multiplication. One might argue such loops could have easily been manually written in a parallel loop nest in the first place. We note that, however, the divide-and-conquer version can achieve the effect of cache blocking at all levels [10] with uniform and simple code, whereas loops must be manually and explicitly tiled to enjoy cache blocking. Our goal is to help programmers write cache-friendly parallel programs with high performance.

3. METHOD

This section details the proposed methods. Our compiler first runs an analysis to identify an Hth termination condition for an appropriately chosen threshold H. When the analysis succeeds, a serialized version is generated and the original version is made to call it when the condition is satisfied. For further optimizations, the compiler tries to apply loopification if applicable; otherwise, it attempts code-bloat-free inlining. If the termination condition analysis fails, the compiler applies a dynamic cut-off proposed by Thoman et al. [34] as a fallback strategy. Figure 4 summarizes the flow of the algorithm selection described above.

All of these methods were designed to receive the LLVM intermediate representation (LLVM IR). We believe that these methods can be implemented for any framework based on Static Single Assignment (SSA) form.

3.1 Termination condition analysis

We first show the termination condition analysis, which is essential to applying our static cut-off methods. While it provides useful information for a cut-off, we note that this analysis is applicable to any recursive functions. For a divide-and-conquer task parallel function, its Hth termination condition C_H is a (sufficient) condition on its incoming arguments under which the call tree created by the function call has a height not greater than H.

Based on this definition, a 0th termination condition C_0 is a condition in which it creates no child tasks. We compute C_0 as a condition in which the execution never reaches basic blocks containing self-recursive calls by analyzing the control flow graph and branch conditions of each basic block.

Specifically, let EC_b be a condition in which basic block b may be executed. EC_b can be calculated by solving the following set of equations for all basic blocks ("⇒" means an implication).

$$\bigvee_{p:p\to b} (EC_p \land BC_{p\to b}) \quad \Rightarrow \quad EC_b \qquad (1)$$

$$EC_{b_{\text{entry}}} \quad = \quad \texttt{true} \qquad (2)$$

where b_{entry} is an entry block of the function, $b_1 \to b_2$ denotes a control transfer edge from b_1 to b_2, and $BC_{b_1\to b_2}$ is a condition at the end of b_1 in which the control branches to b_2. Using an SSA representation, they can be obtained by tracing definitions of variables from the branch instruction at the end of b_1.

Given EC_b's, C_0 is calculated as a condition satisfying:

$$C_0 \Rightarrow \bigwedge_{r\in R} \neg EC_{b(r)} \qquad (3)$$

where R is the set of self-recursive task creation sites and $b(r)$ the basic block containing $r \in R$. This expression means that C_0 is a condition in which no execution conditions of basic blocks which contain self-recursive task creations are satisfied.

We can then calculate an Hth termination condition re-

```
void nqueens(board b, int row, int col, int* ret){
  put(&b, row, col);
  if(valid(b, row)){
    if(row == N){
      *ret = 1;
    }else{
      int ans[N]; zeroclear(ans, N);
      for(int i = 0; i < N; i++)
        spawn nqueens(b, row+1, i, ans+i);
      sync;
      *ret = sum(ans, N);
    }
  }
}
```

Figure 5: Task parallel N-Queens solver

cursively as follows. The recursion stops within a height H if, for each self-recursive task creation site, either the control never reaches there or its arguments to the recursion satisfy the condition C_{H-1}. Thus, C_H is a condition satisfying:

$$C_H \Rightarrow \bigwedge_{r \in R} \left(\neg EC_{b(r)} \vee (\text{arguments to } r \text{ satisfying } C_{H-1})\right) \tag{4}$$

Let us apply the algorithm to our `fib` example in Figure 1. By applying the expression (1), the compiler is able to find, as a 0th termination condition:

$$C_0 \equiv n < 2.$$

Let r_1 and r_2 denote the two self-recursive task creation sites and b_{else} the basic block containing them (the else block in Figure 1). Based on the expression (4), C_1 can be obtained as follows:

$$\begin{aligned} C_1 &\equiv \bigwedge_{r \in \{r_1, r_2\}} (\neg EC_{b_{\text{else}}} \vee (\text{arguments to } r \text{ satisfying } C_0)) \\ &\equiv (\neg EC_{b_{\text{else}}} \vee (\ (n-1) \text{ satisfying } C_0)) \\ &\quad \wedge (\neg EC_{b_{\text{else}}} \vee (\ (n-2) \text{ satisfying } C_0)) \\ &\equiv (n < 2 \vee n-1 < 2) \wedge (n < 2 \vee n-2 < 2) \\ &\equiv n < 3. \end{aligned}$$

By continuing this, the Hth termination condition C_H is recursively calculated as $n < 2 + H$.

There are cases when we fail to find a termination condition; it happens when we cannot express the branch condition that appears in the expression (1) in terms of the arguments of the function. Finding the expression for $BC_{p \to b}$ begins with an expression of SSA variables that directly appear in the branch condition; it transitively replaces these variables with their definitions, until the expression involves only the incoming parameters of the function and constants. When we encounter memory references, phi functions, or function calls as a definition along the way, we always overestimate the reachability condition to identify a sufficient termination condition, leading to missing optimization opportunities. If the obtained termination condition is always false, the analysis effectively fails and no static optimizations are applied.

This strategy can detect a useful condition for programs involving a complex control flow. Figure 5 shows an `nqueens` program, which counts solutions for the N-Queens problem; reachability to recursive call sites depends on the return value of a function call `valid(b, row)`, which is unknown. Our analysis assumes the control *may* transfer, from the basic block containing `valid(b, row)`, to the "then" block of the if statement. Our analysis can nevertheless infer a sufficient condition, $N - H \leq \texttt{row} \leq N$, as the Hth termination condition.

3.2 Static cut-off transformations

As shown in Figure 4, when the termination condition analysis succeeds, we choose an appropriate cut-off parameter H (discussed later in Section 3.3) and then try to apply further cut-off optimizations: static task elimination, code-bloat-free inlining, and loopification. We assume the inputs of these transformations are task parallel programs, but they are potentially applicable to non-task self-recursive functions if dependence between multiple recursive calls is properly analyzed.

If the termination condition cannot be obtained, our compiler applies a dynamic cut-off to the task as a fallback.

3.2.1 *Static task elimination*

If the termination condition for a function is successfully obtained, our compiler generates a serialized version in which all task creations are replaced with serial function calls. The transformation always succeeds if the compiler calculates the termination condition successfully. We call this method *static task elimination*, which cuts off tasks and reduces a tasking overhead without a risk of seriously degrading parallelism. Since this naive method alone is insufficient for extremely fine-grained tasks, the compiler tries to apply two further optimizations to the serialized version, which we discuss below.

3.2.2 *Code-bloat-free inlining*

Though static task elimination is effective in reducing the cost of task creation, a serialized function generated by the elimination still confronts a overhead of function calls. Inlining is a well-known remedy; in our setting, inlining is so powerful that applying it H times can even remove all recursive calls derived from tasks in an Hth termination condition. Simply expanding recursive function calls, however, grows code size exponentially when there are multiple recursive call sites, so the simple inline expansion is unlikely to yield the desired result.

Our compiler overcomes this problem by transforming recursive functions so that there is only a single recursive call site prior to inlining. A basic observation is that a divide-and-conquer function often calls itself multiple times in series. In that case, it is usually possible to transform the series of recursive calls into a loop, containing a single recursive call site in its body. If self-recursive calls are made to appear only once, the inlining technique can be safely applied to it repeatedly, increasing the code size linearly, not exponentially.

This method first searches for recursive task creations in the function. If they appear only once, we need no further transformation (e.g., `nqueens` shown in Figure 5); otherwise, it checks if all of them appear in a single basic block. If this is the case, it tries to move other self-recursive calls to the last one in the basic block unless it violates dependence, and replace the last one into a loop calling all of them. This dependence restriction is alleviated by the task semantics that allow execution of spawned tasks to delay to the corresponding `sync`. If all recursive calls cannot be gathered at the same line, it gives up. The transformation is schematically shown in Figure 6a and 6b.

```
A₀;
f(a₀, b₀, ...);
A₁;
f(a₁, b₁, ...);
[...]
A_{K-1};
f(a_{K-1},b_{K-1},...);
A_K
```

(a) A basic block containing multiple recursive calls. Each A_i is an arbitrary code sequence without branches.

```
A₀;
A₁;
[...]
A_{K-1};
for(int i = 0; i < K; i++){
  a=phi(a₀,...,a_{K-1});
  b=phi(b₀,...,b_{K-1});
  [...]
  f(a,b,...);
}
A_K;
```

(b) Prior to inlining recursive calls, recursive call sites are merged into a single call site.

Figure 6: Transformation of code-bloat-free inlining

Once the transformed function has only one static recursive call site, the compiler inlined it H times. Importantly, the innermost loop containing the recursive call can be omitted since it is never executed in the Hth termination condition. This code-bloat-free inlining can therefore completely remove the self-recursive function calls derived from task creations. For example, consider the serialized `fib` function generated by static task elimination with the 1st termination condition shown as `fib_seq` in Figure 3a. It can be converted into a function presented in Figure 7a, and then into a fully inlined version shown in Figure 7b. Note that, in Figure 7b, the innermost loop is removed since it is unreachable in the given termination condition ($n < 3$).

3.2.3 *Loopification*

While code-bloat-free inlining eliminates all recursive calls, the resulting deeply nested loop is not always ideal outcome. Some functions can be converted into more natural (flat or shallowly nested) loops under the cut-off conditions. For example, it is obvious for humans that our `vecadd` example shown in Figure 3c is equivalent to a flat loop shown in Figure 3d. Loopification attempts to recognize just that. The target of loopification is a recursive function whose control flow graph has the following properties, schematically shown in Figure 8.

1. All recursive call sites are in a single basic block. Let us call it *a recursion block* of the function.[1]
2. The recursion block does not contain any side effects of the function besides calling recursions.
3. There are no control flows executing these blocks twice or more in the recursion block (e.g., loop).

We call a part of function including other basic blocks *a leaf function*, which is obtained by substituting a termination condition into the original function. Such a function ultimately performs all its side effects in the leaf function, thus is hopefully equivalent to a loop whose body is the leaf function. Our analysis tries to find such a loop in the following two steps.

[1]The assumption that there is only a single recursion block is mainly for simplifying the exposition and implementation.

```
void fib_seq(int n, int* ret){
  if(n < 2){
    *ret = n;
  }else{
    int a, b;
    for(int i = 0; i < 2; i++){
      int n2, *ret2;
      switch(i){
      case 0:
        n2 = n-1; ret2 = &a; break;
      case 1:
        n2 = n-2; ret2 = &b; break;
      }
      fib_seq(n2, ret2);
    }
    *ret = a + b;
  }
}
```

(a) `fib_seq` which has a single recursive call site.

```
void fib_seq(int n, int* ret){
  if(n < 2){
    *ret = n;
  }else{
    int a, b;
    for(int i = 0; i < 2; i++){
      int n2, *ret2;
      switch(i){
      case 0:
        n2 = n-1; ret2 = &a; break;
      case 1:
        n2 = n-2; ret2 = &b; break;
      }
      // Inline-expand fib_seq(n2, ret2);
      if(n2 < 2){
        *ret2 = n2;
      }else{
        // The innermost loop can be removed
        // in the 1st termination condition.
        //
        //int a2, b2;
        //for(int i2 = 0; i2 < 2; i2++){
        //  int n3, *ret3;
        //  switch(i2){ ... }
        //  fib_seq(n3, ret3);
        //}
        //*ret2 = a2 + b2;
      }
    }
    *ret = a + b;
  }
}
```

(b) Fully inlined `fib_seq` in Figure 7a

Figure 7: Code-bloat-free inlining for `fib` under $n < 3$

1. **Candidate generation:** It selects a few constant values that satisfy the appropriate termination condition and propagates these constants throughout the body of the function, until recursive calls are completely expanded. The resulting code contains a set of leaf functions with arguments. Based on this information, it tries to synthesize a candidate loop.
2. **Induction:** It inductively verifies that the candidate loop is equivalent to the recursive code in an arbitrary termination condition.

We explain the details by following our `vecadd` example showing in Figure 3c. Its leaf function consists of the "then" block containing the assignment `*a += *b;` We write it $L(a, b)$ below.

1. Candidate generation:.

Let us say we have thus far identified the 2nd termination condition, $1 \le n \le 4$. By assigning a value satisfying the condition, say 4, to n, and by propagating it, serialized

```
void f(a, b, c, ...){
  if(...){
    // Leaf function.
    L(a, b, c, ...);
  }else{
    // Recursion block.
    ...
    f(a0, b0, c0, ...);
    ...
    f(a1, b1, c1, ...);
    ...
  }
}
```

Figure 8: Loopification target

vecadd (vecadd_seq) becomes:

```
void vecadd_seq(float* a, float* b, int n/*=4*/){
  vecadd_seq(a,   b,   2);
  vecadd_seq(a+2, b+2, 2);
}
```

By traversing recursive functions with constant propagation, it ends up with:

```
  L(a, b);
  L(a+1, b+1);
  L(a+2, b+2);
  L(a+3, b+3);
```

It then observes the set of argument expressions:

$$\{(\mathtt{a},\mathtt{b}),(\mathtt{a}+1,\mathtt{b}+1),(\mathtt{a}+2,\mathtt{b}+2),\ldots,(\mathtt{a}+3,\mathtt{b}+3)\ \}$$

and tries to express the set by an affine transformation of a rectangle. If it cannot be expressed so, the analysis fails. In this example, it finds:

$$\{(\mathtt{a},\mathtt{b})+i(1,1)\mid i\in[0,4)\}.$$

By assigning a few other values to n and unifying the obtained results, it yields:

$$\{(\mathtt{a},\mathtt{b})+i(1,1)\mid i\in[0,n)\}.$$

The candidate loop thus becomes:

```
  for (i ∈ [0,n))
    L(a + i, b + i);
```

2. Induction:.

Induction phase verifies that the candidate loop is equivalent to the original function by the usual induction.

Base case checks if the generated loop matches the candidate loop in the 0th termination condition.

Induction substitutes the candidate loop for the recursive calls and checks if they can be fused into a single candidate loop.

In our vecadd example, the compiler derives that the base case, under the 0th termination condition $n = 1$, becomes

```
  L(a, b);
```

by propagating $n = 1$ throughout the function body. It is then easy to see this is equivalent to the candidate loop.

For induction, it proves that a recursive block can be translated into the loop if recursive calls can be converted into that loop. We first replace the recursive calls in the recursion block:

```
  vecadd(a,     b,     n/2);
  vecadd(a+n/2, b+n/2, n-n/2);
```

with the candidate loops, obtaining:

```
  for (i ∈ [0,n/2))
    L(a+i,     b+i);
  for (i ∈ [0,n-n/2))
    L(a+n/2+i, b+n/2+i);
```

We can shift the iteration space of the latter loop by n/2, to obtain:

```
  for (i ∈ [0,n/2))
    L(a+i, b+i);
  for (i ∈ [n/2,n))
    L(a+i, b+i);
```

and fuse them into a single loop:

```
  for (i ∈ [0,n))
    L(a+i, b+i);
```

which is equivalent to the candidate. As a general strategy, we try to shift the iteration space of each loop so that its body becomes equivalent to the candidate loop and see if the union of their iteration spaces becomes equivalent to that of the original loop. The order of loops is changed if necessary as long as task parallelism admits reordering.

It is notable that this induction only proves the equivalence in the case where the recursive function terminates, namely in an arbitrary termination condition. Our transformation guarantees that the loopified function is only called in a termination condition which is explicitly introduced by the static cut-off.

In practice, multiple loop candidates are generated for particular tasks because of their arbitrariness of loop nesting order (e.g., a loop with an index i, then a loop with an index j, or vice versa and loop iterations order (ascending or descending)). Either of them calculates the same computation, however the performance is sensitive to the loop representation, especially loop order. The compiler tries to choose one mostly accessing continuous memory in the innermost loop with a smaller number of terms of lower/upper boundaries as possible. Lastly, it attaches metadata conveying dependence information exposed by task parallelism for promoting further loop optimizations including loop vectorization.

3.3 Determining a cut-off height

Determining a height H for the static cut-off is essential to balancing between parallelism of programs and sequential performance; smaller H's allow less effective cut-off, while larger H's might decrease parallelism. Appropriate thresholds are values that mask a task creation overhead; i.e., values that give granularity larger than a constant factor of a task creation overhead. If, for example, we want to keep the tasking overhead lower than 2% of the execution time, we would choose granularity at least 49 times larger than the task creation overhead. Considering that the state-of-the-art runtime systems create a task in roughly a hundred cycles, our compiler estimates the number of cycles of a function under various heights and chooses the minimum height that makes the granularity larger than 5000 cycles. In order to avoid choosing a too small number due to inaccuracy of cycle estimations, we ensure that the chosen height is at least four. For tasks to which loopification is applied, the H effectively determines cache blocking size of the loop in the leaf, so we instead use the largest H whose resulting task is estimated to access at most 256KB of data, a typical L2 cache size, which we experimentally found balances the advantage of cache blocking and straightforward control flows in a loop.

```
void taskWithSizeCheck(arg1,arg2,...){
  if(terminationCondition_{H+H_min}(arg1,arg2,...)){
    // Input is too small.
    taskWithoutCutoff(arg1,arg2,...);
  }else{
    // Input is adequately large.
    taskWithCutoff(arg1,arg2,...);
  }
}
```

Figure 9: Task interface from external calls

To estimate the number of cycles of a function, our current implementation simply sums up the cost of all instructions in the function, obtained by an LLVM's cost function, and then multiplies it by the number of children (or 2 if failed to identify the number) to the power of the cut-off height. There have been several studies on finding an optimal task granularity by autotuning [2] or memory hierarchy-aware task mapping [9], which may improve our rough estimation. In reality, however, as shown in Section 4.4, our experiences so far indicate performance in most programs plateaus with a modest cut-off height (four).

3.4 Guaranteeing a minimum parallelism

Our heuristics to choose the cut-off height tries to ensure the tasking overhead is within a small constant of the execution time. While desirable in most cases, when the height of the whole task tree is smaller or only slightly greater than the cut-off height, it makes more sense to use a smaller cut-off height, even if it is known to have a non-negligible impact on the overhead. To address this issue, we could generate several versions each using a different cut-off height and select an appropriate one when the function is called from outside. We currently adopt a simplified method using only two versions, one that employs a cut-off at the determined height and the other that does not use cut-off at all. We introduce a constant H_{min}, the minimum height from the root up to which tasks are guaranteed to be created. Figure 9 shows the resulting function that serves as an entry from outside the task.

3.5 Dynamic cut-off

A dynamic cut-off is a method which calls functions instead of creating tasks when tasks are estimated to be abundant based on runtime information such as the depth of the task and the number of ready tasks. While we mainly focus on the static cut-off, we also employ a dynamic cut-off technique as an effective fallback method when our static analysis fails.

Our system adopts a state-of-the-art dynamic cut-off technique proposed by Thoman et al. [34], which generates multiple versions and selects one at runtime. They are the original task, ones that unroll recursion a few times, and a completely serialized version (similar to our static task elimination). Since we were unable to implement the "simplification" algorithm [34] in the unrolling step due to lack of details in the paper. We applied inline-expansion and the maximum `-O3` optimization level of LLVM as our best effort.

4. EVALUATION

4.1 Evaluation settings

We implemented the proposed static cut-off algorithms as an optimization pass in LLVM 3.6.0 [18]. All the programs were run on MassiveThreads [22], a lightweight task library employing a child-first random work-stealing scheduler [21]. The task library was modified to implement the dynamic cut-off.

Fifteen benchmarks were prepared to demonstrate the efficacy of our cut-off methods. The benchmarks are as follows:

1) **fib** calculates a 45th Fibonacci number with the naive double recursion.
2) **nqueens** computes all solutions of the N-Queens problem, with $N = 14$. Both of our **fib** and **nqueens** adopt the same task creation patterns in the benchmarks of BOTS [7].
3) **fft** performs a Fast Fourier Transform. The input array has 2^{25} single-precision complex numbers.
4) **sort** is a merge sort with a parallelized merge step [1]. The input is an array of $100M$ single-precision floating-point numbers.
5) **nbody** directly calculates forces between all N to N pairs. The input array consists of $30K$ particles each of which has its mass, position, velocity, and a temporal variable to accumulate the force. Positions, velocities, and forces are 3D vectors.
6) **strassen** multiplies two matrices using the Strassen algorithm. The inputs are two matrices of 1024×1024 single-precision floating-point numbers. Though it is ordinarily written to switch to a normal matrix multiplication by hand when the matrix size gets smaller, this benchmark uses Strassen algorithm all the way until the matrix size becomes 1.
7) **vecadd** is similar to the task shown in Figure 3c, but it adds two float arrays and stores the result into another array. Each array has 10^9 elements.
8) **heat2d** is a stencil computation solving a 2-dimensional thermal diffusion equation on a $30K \times 30K$ mesh.
9) **heat3d** is a 3-dimensional version of **heat2d**, on a $1K \times 1K \times 1K$ mesh.
10) **gaussian** is an image processing kernel that applies a 5×5 Gaussian filter to an array of $30K \times 30K$ single-precision floating-point numbers.
11) **matmul** multiplies two matrices and stores the result into another matrix. All three matrices have 2000×2000 single-precision floating-point numbers.
12) **trimul** multiplies two upper triangular matrices with the size of 2000×2000. It is similar to **matmul**, except that it omits some multiplications with matrices either of which is zero.
13) **treeadd** traverses a binary tree structure and updates values (floating-point numbers) in the leaves. The input is a balanced tree with the height of 30, containing $2^{30} - 1$ elements.
14) **treesum** traverses a binary tree structure and sums up the values (floating-point numbers) in the leaves. The input is a balanced tree with the height of 30 (the same as the input to **treeadd**).
15) **uts** runs Unbalanced Tree Search [24] with the "T1XL" input in the official samples, which generates a geometric tree with 1,635,119,272 elements in total.

Table 1 presents the larger input we used to examine the performance achieved by our proposed methods and those of loop parallel programs shown in Figure 12, which made execution time longer than 0.2 seconds.

Table 1: Data size for an evaluation shown in Figure 12

nbody	$40K \times 40K$	vecadd	$2G$
heat2d	$40K \times 40K$	heat3d	$(1.2K)^3$
gaussian	$40K \times 40K$	matmul	$5K \times 5K$
trimul	$8K \times 8K$		

Table 2: Applicability of cut-off methods ((X/Y) means a method was applied to X out of Y tasks in the benchmark.)

	dynamic	static	cbf	loop
fib	✓	✓	✓	
nqueens	✓	✓	✓	
fft	✓	✓	✓	
sort	✓	(1/2)	(1/2)	
nbody	✓	✓	✓	
strassen	✓	✓	(4/5)	(4/5)
vecadd	✓	✓	✓	✓
heat2d	✓	✓	✓	✓
heat3d	✓	✓	✓	✓
gaussian	✓	✓	✓	✓
matmul	✓	✓	✓	✓
trimul	✓	✓	(1/4)	(1/4)
treeadd	✓			
treesum	✓			
uts	✓			

We wrote the benchmarks in C language, so we first translated them into the LLVM IR with Clang, a frontend C/C++ compiler for LLVM, and then applied our methods to the intermediate representation. Finally, we compiled them into binary programs with the LLVM compiler with optimization flags `-O3 -ffp-contract=fast` and a machine-specifying option. None of them use any manually written cut-off.

Table 2 shows which optimizations were applied to these benchmarks.

We evaluated seven versions of each benchmark:

- **base**: An original version without any optimization.
- **dynamic**: A version with the adaptive multiversioning method presented by Thoman et al. [34]. It contains no optimizations we propose.
- **static**: A version with static task elimination.
- **cbf**: A version with code-bloat-free inlining.
- **loop**: A version with loopification.
- **proposed**: A version automatically selected by the algorithm flow shown in Figure 4; it applies **loop**, **cbf**, and then **static** in this order and selects the first one that succeeds, falling back to **dynamic** when all fail.
- **seq**: A fully serialized version, created by replacing all task creations with function calls.

Some benchmarks have multiple tasks to which applicable optimizations differ. When the transformation specified by the version is inapplicable to the task, the compiler tries to apply **cbf** and then **static**. If both fail, it employs **dynamic**. For example, **sort** has two tasks: a sorting task and a merging task. Whereas code-bloat-free inlining applies to the former for the **cbf** version, neither **cbf** nor **static** to the latter, which is thus optimized by **dynamic**.

The programs were run on a machine running Linux 3.16, which has dual sockets of Intel Xeon E5-2699 v3 (Haswell) processors (36 cores in total). We ran all the benchmarks with `numactl --interleave=all` to balance physical memory across sockets. All results reported are the average of five measurements. Error bars in the charts indicate the 95% confidence intervals.

4.2 Single-threaded performance

We first examine improvement of single-threaded performance, shown in Figure 10. The baseline is a task parallel program with no cut-off (**base**). All programs were parallelized but executed on a single core except for **seq**, which simply serializes all task creations. The degree of improvement depends on applications characteristics (e.g., the size of each leaf calculation and the number of division of the divide-and-conquer strategies), but our optimizations consistently achieved a significant speedup compared to the baseline (**base**). Static task elimination (**static**) achieved from 1.1x to 24.8x speedup (geometric mean of 5.3x), and code-bloat-free inlining (**cbf**) elevated it to 1.1x - 33.5x (geometric mean of 6.6x) where it is applicable. Performance of static task elimination (**static**) was comparable to that of fully serialized ones (**seq**) except **sort** and **strassen**. It presents that it successfully reduced a tasking overhead in most cases.

Loopification (**loop**) further boosted performance, sometimes spectacularly, up to 333x, achieved by removing control flows and by exposing vectorization opportunities to the backend compilers. Though the dynamic cut-off (**dynamic**) has a wider applicability, the achieved speedup was a moderate 0.9x to 5.9x (geometric mean of 2.6x); they were smaller than those of static task elimination (**static**) in all cases except **heat2d**, and those of code-bloat-free inlining (**cbf**) and loopification (**loop**) in all cases where applicable.

Importantly, our proposed optimization (**proposed**) offered the best performance in all cases, showing the effectiveness of the algorithm selection criteria shown in Figure 4.

4.3 Multi-threaded performance

Figure 11 shows the improvement of multi-threaded executions. The same programs we used in the previous experiment were this time executed by 36 threads. The baseline is the parallel performance of the task parallel programs with no cut-off (**base**).

The result was overall similar to the single-threaded one; by using the geometric mean metric, a speedup of 5.0x was achieved by static task elimination (**static**), 6.0x by code-bloat-free inlining (**cbf**), and 21x (up to 220x) by loopification (**loop**), while dynamic cut-off (**dynamic**) yielded a modest 1.8x speedup.

In comparison to the single-threaded performance shown in Figure 10, scalability (speedup from single-threaded execution to multi-threaded execution) tended to be lower in optimized versions than in the base versions, especially in **vecadd** and **heat2d**. It was presumably because they became memory bandwidth-limited after removing task creation overheads.

Finally, for programs naturally expressible in loops, we loop-parallelized them by hand and compared them with task parallel programs optimized by our compiler. We loop-parallelized **nbody**, **vecadd**, **heat2d**, **heat3d**, **gaussian**, **matmul**, and **trimul**. They performed the same calculations with the corresponding task parallel versions, with the only differences in load partitioning and execution order.

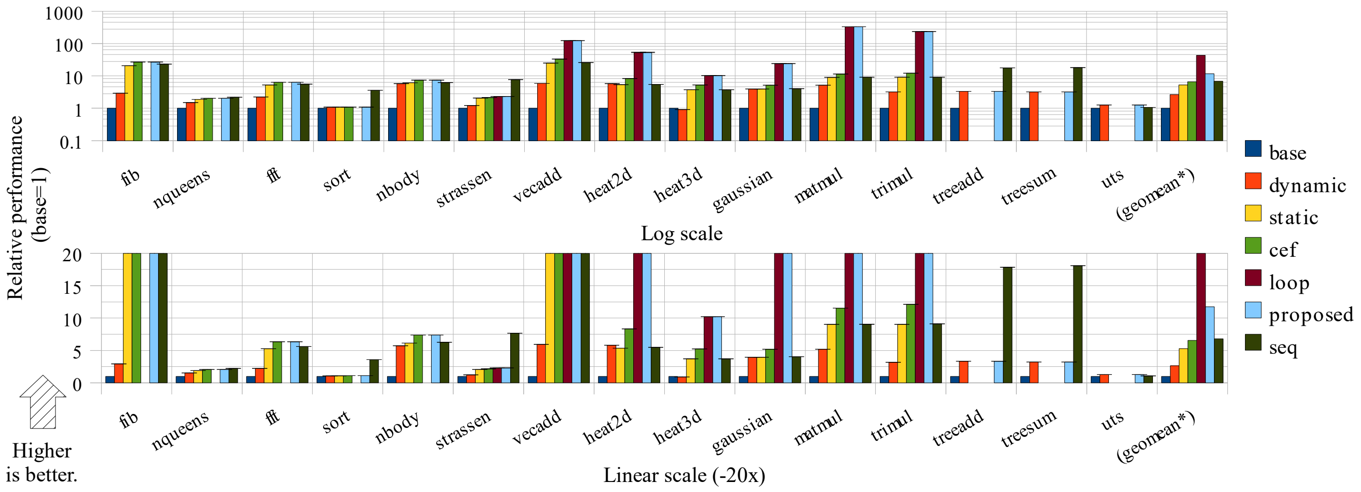

Figure 10: It shows relative single-threaded performance of dynamic cut-off [34] (**dynamic**), static task elimination (**static**), code-bloat-free inlining (**cbf**), loopification (**loop**), our proposal (**proposed**), and a sequential version (**seq**), using the original task parallel program with no cut-off (**base**) as the baseline. The missing results are excluded when we calculate the geometric mean of the overall speedups (**geomean***). Both charts show the same results, different in the scale of y-axis.

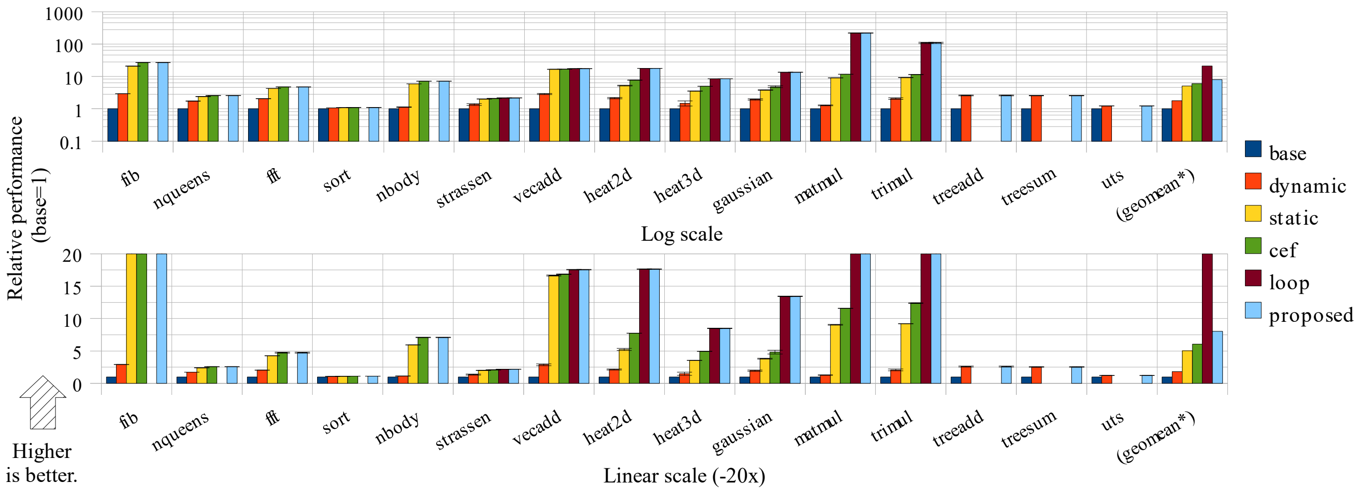

Figure 11: Multi-threaded performance. See Figure 10 for the meaning of labels. The baseline is the original (**base**).

The loop programs were parallelized in two ways: OpenMP with GCC and Polly [11] with LLVM. One was an OpenMP version parallelized by `omp parallel for` pragma. It was compiled by GCC 4.8.4 with `-O3 -ffp-contract=fast` and a machine-specifying option. The other one was automatically parallelized by Polly [11], a locality-optimizer for LLVM based on the polyhedral models with an automatic parallelization feature. The Polly version was compiled by the Clang 3.8.0 with `-O3 -ffp-contract=fast` and the machine-specifying option, and its Polly with `-polly -polly-parallel` and `-polly-vectorizer = stripmine` if it made the program faster. We gave `restrict` keywords appropriately to help automatic parallelization and vectorization in both cases.

Figure 12 shows the relative performance of loop parallel programs, using 36 cores. The baseline is the task parallel version optimized by **proposed** (**task**). **omp** and **polly** are the performance of loop parallel programs optimized by OpenMP with GCC or Polly with Clang. They used default scheduling algorithm and applied no manual locality optimizations such as blocking. We also made **omp_optimized**, a tuned OpenMP program by changing block sizes, scheduling strategies, scheduling parameters (chunk size, etc.), and collapse clauses for nested loops. Since we found that **nbody** and **vecadd** were not parallelized by **polly**, these results missing in the graph are excluded to calculate the geometric mean of speedup (**geomean***). Figure 12 shows the overall performance of **task** was faster than **omp** or **polly**; the geometric mean of **omp**'s relative performance was 0.59x, and that of **polly** was 0.60x. It indicates **task** utilized the cache blocking, especially for **matmul** and **trimul**. However, **omp_optimized** boosted performance significantly to show that of 1.4x; careful manual optimizations (especially blocking) enhanced performance overall. One disadvantage of the divide-and-conquer strategy is that, even if we can choose the best cut-off height, our cache blocking of **task** is not so flexible as to fit the cache size exactly since the problem size is basically evenly divided by the number of recursive calls in a divide-and-conquer strategy.

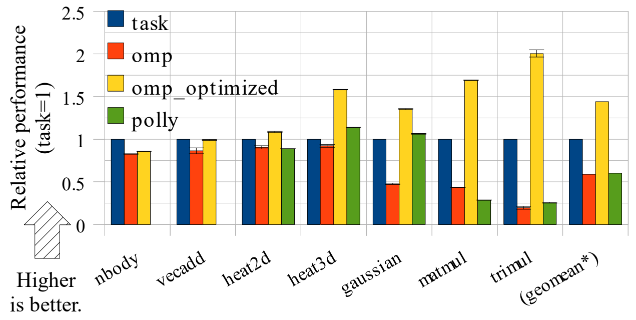

Figure 12: Performance comparison task parallel programs optimized by our methods to loop parallel programs. The baseline is task parallel programs (**task**). **vecadd** and **nbody** are excluded to calculate the (**geomean***) of **polly**

4.4 Cut-off threshold

Figure 13 shows the performance of static task elimination (**static**) using 36 threads, by changing the fixed height parameter H in order to evaluate our current automatic height selection. The baseline is the parallel performance of tasks without cut-off (**base**). Crosses in the figure indicate the heights our compiler automatically selected. Note that some benchmarks (e.g., **fft**) have multiple tasks whose chosen heights are different, thus have multiple crosses. While a few benchmarks had the room for improving performance by larger h (especially for **fib**), the figure overall shows that our algorithm chose good heights that achieved nearly the best performance, without unnecessarily aggressive cut-off heights.

5. RELATED WORK

Automatic cut-off.

Since task granularity is one of the most important factors for the performance of task parallel programs [20], there have been several studies on automating manual cut-off [3, 6, 34]. Duran et al. [6] proposed a heuristic cut-off utilizing the runtime information such as depth of the task and the number of ready tasks. Bi et al. [3] extends their adaptive cut-off to efficiently deal with irregular task parallel programs. Thoman et al. [34] proposed generating multiple versions including inlined versions and fully serialized version, and switching between them based on the similar criteria. These proposals, or runtime-based approaches in general, have advantages of being widely applicable and not requiring substantial compiler development efforts. However, besides the difficulty of applying aggressive optimizations after a cut-off, they have a risk of adversely decreasing parallelism. To see the problem, consider a task creation site:

```
spawn f(x);
```

and say the system decides to serialize it (i.e., not to create a task for it). The decision is made because all or most processors are busy when the execution reaches this statement and they will remain so at least for a while. The true condition that deems this task creation unnecessary is that, however, processors are busy *during the entire span of f(x)*, a condition that is difficult to predict at the time of task creation. Were it not the case, an idle processor would lose the opportunity to steal the continuation of *f(x)*, decreasing

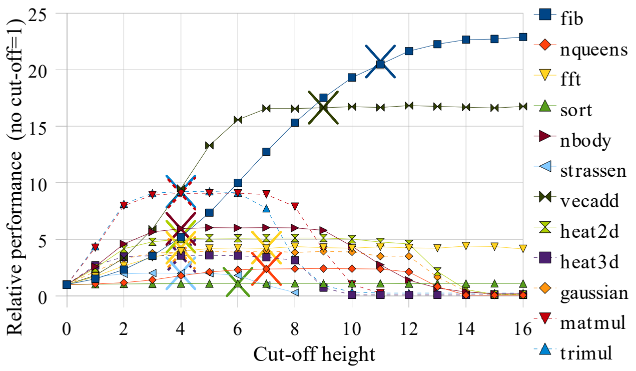

Figure 13: Performance of various cut-off heights. The baseline is task parallel programs with no cut-off (**base**). Crosses denote the heights our compiler automatically selected.

parallelism.[2] In general, serialization of a task is equivalent to inserting an artificial dependence from the end of the created task to the continuation of the task creation. It may thus lengthen the critical path (the longest dependent chain) of the computation, which determines an achievable speedup [8] and there seems no obvious bounds on the factor it is stretched.

Our static cut-off approach differs from these prior efforts in that we *statically identify small tasks* and specifically target them for cut-off and further aggressive *compile-time* optimizations, such as inlining and loopification.

Optimization of recursive functions.

A number of studies have sought effective techniques to remove recursions, especially in a field of functional programming [5]. Tail call elimination [30] is a well-known technique to transform recursive procedures into jumps (a control flow without stacks). Particularly, it cannot handle most divide-and-conquer algorithms, which perform two or more recursive calls one of which is necessarily not tail-recursive. Tang [32] proposed *complete inlining*, which can be thought of a generalization of tail call elimination. It recognizes, more broadly, recursive calls that can be transformed into a jump. The recursive call does not have to be tail-recursive, yet the technique still cannot handle procedures that perform recursive calls twice or more. In contrast, our approach fully expands such recursions under the condition in which the recursion stops in a certain number of steps.

There have been numerous techniques primarily targeting divide-and-conquer algorithms [13, 29, 31]. Rugina et al. [29] proposed function unrolling and rerolling, which together transform a recursive procedure to another recursive procedure. The success of the transformation relies on "conditional fusions," which essentially require the depths of the two recursions be identical. This condition seems very restrictive; most programs we applied our techniques to do not satisfy it. Herrman and Lengauer [13] have shown transformation of divide-and-conquer Haskell programs to parallel loop nest in C. It assumes Haskell programs written in a specific skeleton (template) and vectorizes them. The vectorizable skeleton essentially assumes that the recursion stops at the same depth everywhere, which seems restrictive for

[2]Here we assume a child-first execution policy [21], but a similar argument holds for help-first policies as well.

practical purposes. An idea of *recursion flattening* proposed by Stitt and Villarrealm [31] is similar to ours, which completely eliminates recursive calls by unrolling. Their algorithm, however, only targets recursive functions whose maximum recursion depth can be determined at compiling time (e.g., constant arguments).

Transformation algorithms translating loops into recursions have been studied previously [15, 35] and some studies [33, 35] aimed at achieving the effect of cache blocking at many different levels [10]. The opposite conversion has also been developed [12, 14, 19], while most of them are targeting functional programming languages and their main purpose is recursion removal. Technical differences are as follows. Harrison's approach [12] cannot convert divide-and-conquer programs. The technique proposed by Himpe [14] preserves an execution order, so the generated loop is less optimal. Liu's method [19] requires a stack for multiple recursive calls in general, or divide-and-conquer algorithms. The target of our loopification, on the other hand, is divide-and-conquer task functions and its resulting code can utilize the cache blocking at all levels.

Vectorization of recursive programs.

Most previous researches on auto-vectorization targeted loops [23] or basic blocks [17]. There have been a few targeting recursive programs. Auto-vectorization on recursive functions written in Haskell has been proposed by Petersen et al. [26], which focuses on a simple recursive function, but a divide-and-conquer function with multiple recursive call sites. Jo et al. [16] have presented an effective auto-vectorization method for programs traversing the same tree many times and it is those multiple traversals to which a vectorization is applied. In a recent work, Ren et al. [28] proposed a general vectorization technique for divide-and-conquer programs, which is very different from ours. In their framework, a compiler assigns multiple tasks created by a single task to different SIMD lanes. This strategy is applied to the root of the task tree, effectively executing subtrees *near the root of the task tree* in different SIMD lanes. In contrast, our compiler tries to convert tasks *near the leaves of the task tree* into loops, which may then be vectorized. Their vectorization technique can be applied to a wider range of programs than our loopification technique. On the other hand, we believe the program generated by our loopification technique still has a number of performance advantages when solely looked at vectorization techniques. It has a simpler control flow and preserves the temporal locality of the original program. Moreover, our loopification straightforwardly co-exists with thread level parallelism.

6. CONCLUSION

This paper proposed an effective static cut-off method for divide-and-conquer task parallel programs and two further optimizations; code-bloat-free inlining and loopification. Our proposed algorithm statically determines a condition in which the recursion reaches near leaves and cut off only those tasks. It then applies either code-bloat-free inlining or loopification of the resulting coarsened tasks. Our static cut-off can be easily combined with the dynamic cut-off, which can widen the applicable range. Compared to the original task parallel programs with no cut-offs, our evaluation shows good performance improvement.

Acknowledgment

We thank the anonymous reviewers for their helpful and constructive comments. This work was in part supported by Grant-in-Aid for Scientific Research (A) 16H01715.

7. REFERENCES

[1] S. G. Akl and N. Santoro. Optimal parallel merging and sorting without memory conflicts. *IEEE Trans. Comput.*, 36(11):1367–1369, Nov. 1987.

[2] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. PetaBricks: A language and compiler for algorithmic choice. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 38–49, Jun. 2009.

[3] J. Bi, X. Liao, Y. Zhang, C. Ye, H. Jin, and L. T. Yang. An adaptive task granularity based scheduling for task-centric parallelism. In *Proceedings of the 2014 IEEE International Conference on High Performance Computing and Communications*, HPCC '14, pages 165–172, Aug. 2014.

[4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '95, pages 207–216, Jul. 1995.

[5] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, Jan. 1977.

[6] A. Duran, J. Corbalán, and E. Ayguadé. An adaptive cut-off for task parallelism. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 36:1–36:11, Austin, Texas, USA, Nov. 2008.

[7] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade. Barcelona OpenMP Tasks Suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In *Proceedings of the 2009 International Conference on Parallel Processing*, pages 124–131, Sept. 2009.

[8] D. L. Eager, J. Zahorjan, and E. D. Lozowska. Speedup versus efficiency in parallel systems. *IEEE Trans. Comput.*, 38(3):408–423, Mar. 1989.

[9] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, Nov. 2006.

[10] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 285–, Oct. 1999.

[11] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet. Polly - Polyhedral optimization in LLVM. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques*, IMPACT '11, Apr. 2011.

[12] P. G. Harrison and H. Khoshnevisan. A new approach to recursion removal. *Theoretical Computer Science*, 93(1):91 – 113, 1992.

[13] C. A. Herrmann and C. Lengauer. *Transformation of Divide & Conquer to Nested Parallel Loops*, pages 95–109. PLILP '97. Springer-Verlag, 1997.

[14] S. Himpe, F. Catthoor, and G. Deconinck. Control flow analysis for recursion removal. In *Proceedings of the 7th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '03, pages 101–116. Springer, Sept 2003.

[15] D. Insa and J. Silva. Automatic transformation of iterative loops into recursive methods. *Information and Software Technology*, 58:95 – 109, 2015.

[16] Y. Jo, M. Goldfarb, and M. Kulkarni. Automatic vectorization of tree traversals. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 363–374, Sept. 2013.

[17] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 145–156, Jun. 2000.

[18] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Mar. 2004.

[19] Y. A. Liu and S. D. Stoller. From recursion to iteration: What are the optimizations? In *Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '00, pages 73–82, Jan. 2000.

[20] H.-W. Loidl and K. Hammond. On the granularity of divide-and-conquer parallelism. In *Proceedings of the 1995 Glasgow Workshop on Functional Programming*, GWFP '95. Springer-Verlag, Jul. 1995.

[21] E. Mohr, D. A. Kranz, and R. H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 185–197, Jun. 1990.

[22] J. Nakashima and K. Taura. MassiveThreads: A thread library for high productivity languages. In *Concurrent Objects and Beyond*, volume 8665 of *Lecture Notes in Computer Science*, pages 222–238. 2014.

[23] D. Nuzman and A. Zaks. Autovectorization in GCC - two years later. In *Proceedings of the 2006 GCC Developers' Summit*, pages 145–158, Jun. 2006.

[24] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng. UTS: An unbalanced tree search benchmark. In *Proceedings of the 19th International Conference on Languages and Compilers for Parallel Computing*, LCPC '06, pages 235–250, 2007.

[25] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 3.0, May 2008.

[26] L. Petersen, D. Orchard, and N. Glew. Automatic SIMD vectorization for Haskell. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 25–36, Sept. 2013.

[27] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism.* O'Reilly Media, 2007.

[28] B. Ren, Y. Jo, S. Krishnamoorthy, K. Agrawal, and M. Kulkarni. Efficient execution of recursive programs on commodity vector hardware. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 509–520, Jun. 2015.

[29] R. Rugina and M. C. Rinard. Recursion unrolling for divide and conquer programs. In *Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing-Revised Papers*, LCPC '00, pages 34–48, Aug. 2001.

[30] G. L. Steele, Jr. Debunking the "expensive procedure call"; myth or, procedure call implementations considered harmful or, lambda: The ultimate goto. In *Proceedings of the 1977 Annual Conference*, ACM '77, pages 153–162, Jan. 1977.

[31] G. Stitt and J. Villarreal. Recursion flattening. In *Proceedings of the 18th ACM Great Lakes Symposium on VLSI*, GLSVLSI '08, pages 131–134, May 2008.

[32] P. Tang. Complete inlining of recursive calls: Beyond tail-recursion elimination. In *Proceedings of the 44th Annual Southeast Regional Conference*, ACMSE '44, pages 579–584, Mar. 2006.

[33] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The Pochoir stencil compiler. In *Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 117–128, Jul. 2011.

[34] P. Thoman, H. Jordan, and T. Fahringer. Adaptive granularity control in task parallel programs using multiversioning. In *Proceedings of the 19th International Conference on Parallel Processing*, Euro-Par'13, pages 164–177, Aug. 2013.

[35] Q. Yi, V. Adve, and K. Kennedy. Transforming loops to recursion for multi-level memory hierarchies. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 169–181, Jun. 2000.