Optimizing Recursive Task Parallel Programs

Suyash Gupta*
Purdue University
gupta283@purdue.edu

Rahul Shrivastava
IIT Madras
rahuls@cse.iitm.ac.in

V Krishna Nandivada
IIT Madras
nvk@iitm.ac.in

ABSTRACT

We present a new optimization DECAF that optimizes recursive task parallel (RTP) programs by reducing the task creation and termination overheads. DECAF reduces the task termination (join) operations by aggressively increasing the scope of join operations (in a semantics preserving way), and eliminating the redundant join operations discovered on the way. Further, DECAF extends the traditional loop chunking technique to perform load-balanced chunking, at runtime, based on the number of available worker threads. This helps reduce the redundant parallel tasks at different levels of recursion. We also discuss the impact of exceptions on our techniques and extend them to handle RTP programs that may throw exceptions. We implemented DECAF in the X10v2.3 compiler and tested it over a set of benchmark kernels on two different hardwares (a 16-core Intel system and a 64-core AMD system). With respect to the base X10 compiler extended with loop-chunking of Nandivada et al. [**?**] (LC), DECAF achieved a geometric mean speed up of 2.14× and 2.53× on the Intel and AMD system, respectively. We also present an evaluation with respect to the energy consumption on the Intel system and show that on average, compared to the LC versions, the DECAF versions consume 71.2% less energy.

KEYWORDS

data parallel, recursive task parallel, useful parallelism

ACM Reference format:
Suyash Gupta, Rahul Shrivastava, and V Krishna Nandivada. 2017. Optimizing Recursive Task Parallel Programs. In *Proceedings of ICS '17, Chicago, IL, USA, June 14-16, 2017,* **??** pages.
DOI: http://dx.doi.org/10.1145/3079079.3079102

1 INTRODUCTION

The onset of multi-core architectures has brought forth a shift in programming paradigm from sequential programs to task parallel programs. The task parallel languages allow the programmer to express the desired amount of parallelism (a.k.a *ideal* parallelism), while delegating the task of extracting the *useful* parallelism to the compiler and/or runtime. Recursive Task Parallel (RTP) programs constitute an important subset of task parallel programs written in popular languages like Cilk [**?**], X10 [**?**], Chapel [**?**], OpenMP [**?**], HJ [**?**], and so on. In RTP programs, each task can recursively create newer tasks and wait for those respective tasks to terminate.

```
def find_queens() {...; nqueens (n, 0, ...);}
def nqueens(val n:Int, val j:Int, ...) {...
 finish {
   for(var i:Int=0; i<n; i++) {
     async { ... /* Check for conflicts */
             nqueens(n, j+1, ...); } } } }
```

(a)

```
def nqueens(val n:Int, val j:Int, ...) {...
 var nChunks:Int=Runtime.retNthreads();
 var chunkSize:Int=(n+nChunks-1)/nChunks;
 finish {
   for(var ii:Int=0; ii<n; ii+=chunkSize) {
     val ni = ii;
     async { var kx:Int = ni+chunkSize;
             if(kx>n) kx=n;
             for(var i:Int=ni; i<kx; i++) {
               ... /* Check for conflicts */
               nqueens(n, j+1, ...); } } } } }
```

(b)

Figure 1: BOTS Nqueens kernel in X10: (a) Unoptimized version (b) Loop Chunked version of the nqueens function.

This leads to the execution of a large number of redundant task-creation and -termination operations. Importantly, the structure of RTP programs makes it quite challenging to identify and eliminate such redundant operations. We will use an example to illustrate the problem.

Redundant task termination operations: Figure 1(a) shows the snippet of the BOTS [**?**] Nqueens kernel, in X10. The `async` construct spawns a new child task to execute the statement within its body, in parallel with the parent task. The `finish` construct acts as a join point for all the tasks spawned in its body. The code in Figure 1(a) (due the presence of recursive task parallelism) may lead to the execution of a large number of `finish` operations at runtime (for example, when n=14, it executes 27 million `finish` operations). Nandivada et al. [**?**] show that eliminating unnecessary `finish` operations can lead to significant performance improvements. However, in this code, their proposed technique does not reduce the number of `finish` operations. Interestingly, we observe that each task spawns new child tasks, and waits at the join point for the spawned tasks to terminate. After that the task simply returns from the procedure. Hence, the `finish` construct can be pulled out of the `nqueens` method and placed around its non-recursive call site (in `find_queens`), without altering the semantics of the code. In other words, the scope of the `finish` construct can be expanded to surround the first call to `nqueens`. Such an optimization brings down the number of dynamic `finish` operations to just one (compared to 27 million), for the code shown in Figure 1(a). This can lead to significant performance gains.

Redundant task creation operations: Further analysis of Figure 1(a) shows that at recursion level k, `nqueens` creates $\mathtt{n}^k$ number of `async`s (tasks) leading to an explosion of tasks (e.g., when n=14, it

[*] Work done while at IIT Madras

ICS '17, Chicago, IL, USA

DOI: http://dx.doi.org/10.1145/3079079.3079102

creates a total of 377 million tasks) — results in large performance overheads. The powerful Loop Chunking [**?**] scheme (henceforth referred to as LC) extracts useful parallelism from the ideal. LC splits the iterations of a parallel loop into a set of chunks, where each chunk (containing a set of serial iterations) runs in parallel.

Figure 1(b) presents the LC version of the `nqueens` function. Here, the `Runtime.retNthreads` function returns the initial count of the worker threads. Hence, the useful parallelism is bound by `nChunks`. Considering this, LC ensures that at most `nChunks` number of tasks are created in any invocation of this function. Thus, at level k of recursion, it creates $\texttt{nChunks}^k$ number of tasks (e.g., when `n`=14 and `nChunks`=8, it creates 189 million tasks). This chunked program runs faster than the unoptimized version, but still incurs a large task creation and termination overhead. This is because the chunking algorithm is oblivious to the recursive call inside the loop, and hence, permits the spawning of a large number of tasks.

We have observed such trends in a number of RTP kernels present in two open-source benchmark suites: IMSuite [**?**] and BOTS. In general, it is quite challenging to address the dual problems of reducing redundant task-termination and task-creation operations in RTP programs. For example, it is non-trivial to expand the scope of `finish` constructs nested deep inside some `if/while` constructs, and there may be dependencies between the code inside the `finish` block and the code outside. This problem becomes further challenging, in the presence of exceptions. Similarly, to avoid the creation of large number of redundant tasks it is imperative that tasks are created based on the available "free" workers at runtime.

In this paper, we present a new optimization DECAF that handles both of these challenges. DECAF aggressively expands the scope of `finish` operations and helps elide a large number of dynamic `finish` operations. This is in contrast to the finish-elimination algorithm of Nandivada et al [**?**] that mainly focusses on reducing the scope of `finish` operations. To handle the redundant tasks created in loops, in recursive functions, DECAF modifies the chunking algorithm of Nandivada et al [**?**] to generate code that spawns new tasks based on the number of "idle" workers available, at runtime. If no idle workers are available, the current worker executes the loop serially. During the serial execution, if some workers become idle, the remaining iterations can be executed in parallel (by the idle workers). For the example shown in Figure 1(a), DECAF significantly reduces the tasks creation operations – for n=14, DECAF leads to the creation of 6 million tasks (≈ 30× less, Vs LC).

We introduce a new optimization practice, in the context of RTP programs, of expanding (instead of contracting) the scope of task termination constructs (such as `finish`) from the procedural definitions to their respective call-sites, in a semantics preserving manner, even in the presence of exceptions. Furthermore, we extract the useful parallelism by utilizing the key insight of available idle workers at runtime, as part of a mixed compiler+runtime based optimization. We believe that such a principled strategy helps to define and extract the maximum useful parallelism in case of RTP programs.

Our Contributions

• We propose a new optimization DECAF for improving the performance of RTP programs that reduces the redundant task creation and termination operations. DECAF can also be extended to other task parallel languages (such as HJ, Chapel and OpenMP) that have similar constructs for task creation and task termination operations. We also implemented DECAF on top of the X10v2.3 compiler.

1. Loop-Finish Interchange

`for(S1;c;S2)` $\Longrightarrow$ `S1;` **`finish`** `{`
`{` **`finish`** `S3 }`   `for(;c;S2) {S3}}`

// Say E_s = set of e-asyncs in `S3`
// $\neg\exists e \in E_s$: `c` *has dependence on e.*
// $\neg\exists e \in E_s$: e has loop carried dependence on `S2`, `c` *or* `S3`

2. Finish Fusion

`finish``{S1}` **`finish`**`{S2}` $\Longrightarrow$ **`finish`**`{S1; S2}`
// `S2` *has no dependence on any e-async of* `S1`.

Figure 2: Existing transformation rules.

• We extend DECAF to perform semantics preserving code transformation even in the presence of exceptions.

• We evaluated DECAF over eight benchmarks (drawn from two benchmark suites: IMSuite and BOTS) on two different hardware systems (a 16-core Intel system and a 64-core AMD system). We show that DECAF leads to improved execution times (geometric mean of 2.14× on the Intel and 2.53× on the AMD system, with respect to the LC version. We also show that DECAF leads to lower energy consumption.

2 BACKGROUND

X10: We briefly describe some relevant X10 constructs here (see the X10 manual [**?**] for details). '`async S1`' spawns a new asynchronous task to execute `S1`. A task can be registered on one or more clocks. For example, '`async clocked(c1,c2) S`' registers the new spawned task on the clocks `c1` and `c2`. Two tasks with at least one common registered clock can synchronize by executing `Clock.advanceAll()`. '`finish S`' waits for all the tasks spawned in `S` to terminate. Each `async` has a unique Immediately Enclosing Finish (IEF), at runtime. Note: statically an `async` may have multiple IEFs.

The escaping asyncs or *e-asyncs* [**?**] of a statement `S` are the `async` statements (within `S`) whose IEF is not enclosed within `S`.

X10 runtime is built around the notion of *workers*. Each worker is assigned a task to execute and can be seen as a software thread. The initial count for workers can be set (typically to the number of available cores) at runtime, using the environment variable `X10_NTHREADS`. During execution, X10 runtime also tracks the number of *idle-workers* – workers which are assigned no task.

Finish Elimination: The 'Finish Elimination' optimization of Nandivada et al. [**?**] repeatedly applies a series of transformation rules to eliminate the redundant `finish` constructs. Two of their proposed set of rules are relevant to this work and for the sake of completeness, are reproduced in Figure 2. Each transformation includes a set of pre-conditions (shown as comments) necessary to ensure semantics preserving transformation. *Loop-Finish Interchange* is applicable when, neither there is a loop carried dependence between the iterations of the loop, nor the loop condition depends on the e-asyncs of `S3`. This rule can be trivially extended for other looping constructs such as, *while* and *do-while*. *Finish Fusion* merges two `finish` statements, if `S2` has no dependence on the e-asyncs of `S1`.

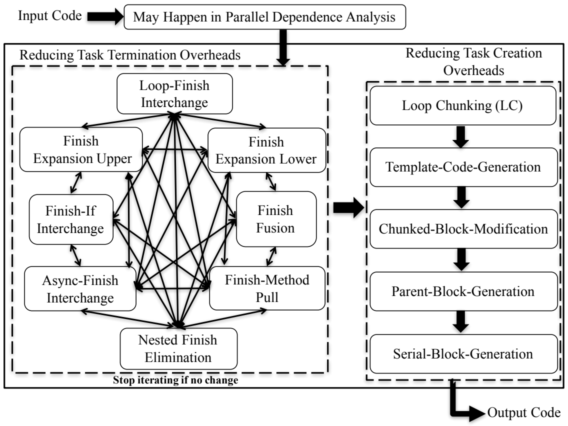


Figure 3: Block diagram of DECAF

3 DETAILS OF DECAF

In this section, we discuss a novel optimization to reduce the task creation and termination overheads in recursive task parallel programs; we call it DECAF[1]. Figure 3 shows the block-diagram of DECAF. We first compute the *may-happen-before dependence* (MHBD) [**?**] information, before invoking DECAF that generates optimized code. DECAF aggressively expands the scope of `finish` operations to eliminate redundant task termination operations (described in Section 3.1). To eliminate redundant task creation operations, DECAF extends the chunking algorithm of Nandivada et al [**?**] to perform loop chunking in a load-balanced manner, by taking into consideration the idle workers at runtime. For the sake of simplicity, in this section, we assume that the programs do not throw exceptions. In Section **??**, we extend our proposed techniques to do handle X10 programs that may throw exceptions.

3.1 Reducing Redundant Task Termination Ops

DECAF applies a set of eight transformation rules that aim to expand the scope of `finish` operations. Two of these rules have been proposed by Nandivada et al [**?**] (Figure 2). The rest six rules (*Nested Finish-Elimination*, *Async-Finish Interchange*, *Finish-If Interchange*, *Finish Expansion Upper*, *Finish Expansion Lower*, and *Finish-Method Pull*), shown in Figure 4, are new. The necessary pre-conditions (checked by the compiler) for any rule are specified as comments. Note: Since our goal of expanding the scope of `finish` operations and the goal of finish-elimination optimization [**?**] are opposite in nature, naturally some of the rules used here do the opposite of those used in the finish-elimination optimization. Accordingly, the pre-conditions and the transformations will vary.

Nested Finish Elimination eliminates the trivially redundant `finish` constructs. *Finish-If Interchange* pulls out a `finish` construct from the surrounding `if` construct. The if-then-else statement is handled as a special case: `if (cond) {finish S1} else {finish S2}`⟹`v=cond;finish{if(v) S1 else S2}`. The switch-case statement is also handled similarly. *Finish Expansion Upper* expands the `finish` scope by pulling a preceding statement `S1` in its scope. It requires that `S1` does not have any e-asyncs registered on clocks. *Finish Expansion Lower* expands the scope of the `finish` construct by pulling in a succeeding statement `S1`. It requires that (i) there is no dependence (MHBD) between `S2` and the e-asyncs of `S1`, and (ii) `S2` does not have any e-asyncs registered on clocks. The *Async-Finish Interchange* interchanges the surrounding `async` and the inner `finish`. In conjunction with other transformation rules, this rule helps to increase the scope of `finish`. *Finish-Method Pull* rules lift a `finish` construct from a method to all its possible callers (obtained by a conservative flow analysis), by taking into consideration the possible recursive calls. The rule (b) precludes the possibility of leaving `S` surrounded by a `finish` construct, in case `S` contains a recursive call to `f3`.

3. Nested Finish Elimination

```
finish finish S1       ⟹    finish S1
```

4. Finish-If Interchange

```
if(e) {finish S1}      ⟹    v=e; finish{if(v) S1}
```

5. Finish Expansion Upper

```
S1; finish {S2}        ⟹    finish {S1; S2}
```

// *If* `S1` *has no e-asyncs registered on clocks.*

6. Finish Expansion Lower

```
finish {S1}; S2        ⟹    finish {S1; S2}
```

// `S2` *has no e-asyncs registered on clocks. Say* E_s*=set of e-asyncs in* `S1`.
// $\neg\exists e \in E_s$*:* `S2` *has dependence on e.*

7. Async-Finish Interchange

```
async finish S1        ⟹    finish { async S1}
```

// `S1` *has no e-asyncs registered on clocks.*

8. Finish-Method Pull

(**a**) *// finish-method pull hasn't already been applied on* `f3()`.

```
def f2(){S1;f3();S2} ⟹ def f2(){S1;finish f3();S2}
def f3(){finish S}      def f3() { S }
```

(**b**) *// finish-method pull has already been applied on* `f3()`.

```
def f3(){finish S}     ⟹    def f3() { S }
```

Figure 4: Rules to help expand the scope of `finish` operations.

Note I: The pre-conditions on the e-asyncs like those specified on Rules 5, 6, and 7 ensure that the translated code does not deadlock [**?** , Section 15.2]. Note II: the order in which the set of eight rules are applied has no effect on the final generated code. These eight rules are iteratively applied till we reach a fix point.

The rules (#1-#8) listed in Figures 2 and 4 can be categorized under two heads (a) eliminating rules: transformations to eliminate redundant `finish` constructs, and (b) expanding rules: transformations to expand the scope of `finish` constructs. For example, Rule #2, and #3 reduce the static `finish` operations; and Rule #1 and Rule #8 can reduce the dynamic `finish` operations; these rules fall in the category of eliminating rules. The Rules #4−#8 are examples of expanding rules. Note: Rule #8 is both an 'eliminating' rule and an 'expanding' rule.

Sample Transformation:
We now discuss the working of the eight rules of DECAF for reducing redundant task termination operations, on the input code shown in Figure 5(a). Assume that `S1`, `S2`, `S3`, and `S4` have no e-asyncs registered on clocks. Further, `S4` has no dependence on the e-asyncs of `S2` or `S3`, and `S2` has no dependence on the e-asyncs of `S1`. DECAF starts by applying *Finish Fusion*, followed by *Finish/If Interchange* (Figure 5(b)). Then, it applies *Async/Finish Interchange*

[1] DECAF (decaffeinated coffee) is like regular coffee, except that at least 97% of the caffeine has been removed. – source authoritynutrition.com

```
// Example Code
S1;
finish {
 for(i in 0..n){
  async {        ⟹
   if(cond) {
    finish S2;
    finish S3;
   }}}} S4;
```

(a)

```
// Applying rules #2, #4
S1;
finish {
 for(i in 0..n){
  async {        ⟹
   finish {
    if(cond){S2;S3
    }}}}}
S4;
```

(b)

```
// Applying rule #7
S1;
finish {
 for(i in 0..n) {
  finish {    ⟹
   async {
    if(cond){S2;S3;
    }}}}}
S4;
```

(c)

```
// Applying rules #1, #3
S1;
finish {
 for(i in 0..n) {
  async {        ⟹
   if(cond){S2;S3;
   }}}}
S4;
```

(d)

```
// Applying rules #5, #6
finish {
 S1;
 for(i in 0..n) {
  async {
    if(cond){S2;S3;
    }}}
 S4; }
```

(e)

Figure 5: Using DECAF to expand the scope of `finish` operations on the code in (a). The modifications are highlighted in `bold`.

(Figure 5(c)), *Loop/Finish Interchange* followed by *Nested Finish Elimination* (Figure 5(d)), and finally *Finish Expansion Upper* and *Finish Expansion Lower* to obtain the code in Figure 5(e).

3.2 Reducing Redundant Task Creation Ops

The existing loop-chunking (LC) optimization [**?**] suffers from a drawback that it may create tasks even when there are no idle workers at runtime. This may lead to significant overheads (especially in case of RTP programs, where it is common to have many tasks created at each level of recursion). DECAF takes inspiration from the lazy-binary-splitting scheme [**?**] and reduces these overheads through two simple, yet effective strategies: (i) dynamic task creation based on the number of idle workers and load balancing among the workers, and (ii) serial execution if no idle workers are available. For the ease of explanation, we first discuss the details of our scheme for input codes having no synchronization operations. In Section 3.2.1, we extend our scheme to handle code with synchronization operations.

We first discuss a modification to the chunking policy of LC and a minor extension to the X10 runtime, before going over the scheme of DECAF to reduce the number of redundant task creation operations.

Modified chunking policy. We make two simple modifications to the chunking policy to balance the load: a) divide the iterations equally among all the idle workers (not all the workers), and b) spare some iterations for the *current* worker (executing the current task).

To highlight the unbalanced load distribution inherent in LC consider the code shown in Figure 1(b) (obtained after invoking LC on Figure 1(a)). Say, `n=9` and number of total workers = `nChunks` = 4. Thus, `chunkSize` is equal to `3`, and LC creates three tasks (to execute three iterations each). Say, excluding the current worker, the other three workers are currently idle. In such a scenario, the three idle workers execute one task each, while the current worker waits at the join point for the spawned tasks to terminate. In contrast, our chunking policy creates three tasks (with three, two and two iterations each), that are executed by the three idle workers. The remaining two iterations are executed by the current worker. Thus, DECAF ensures that the current worker not only does some useful work, but also gets the smallest chunk of iterations. This leads to better load-balancing (the "critical path" length remains unchanged).

Let us consider another case where `n=12` and `nChunks=4`. LC creates four tasks (to execute three iterations each). In such a scenario, the three idle workers execute one task each, and the last task is placed in the task queue. Say, the current worker reaches the join point (finish end), and then picks up the last task (from the task queue) to execute. In this process, the current worker switches its current task, executes the new task and then switches back to its old task. In contrast, instead of creating the "last task" and inserting it in the job queue, our chunking policy executes the iterations corresponding to the "last task" directly. Thereby avoiding the cost of task creation, and switching operations.

Extending the X10 runtime (XRX). We expose a field in XRX (`Runtime.retIdleWorkers`) to obtain the current count of idle workers. In a transformed RTP program, it is possible that two tasks may query the same value of the count of idle workers, at the same instant. Which may lead to creation of more tasks than the idle workers. This inaccuracy can be addressed by the use of 'atomic' sections, but we avoid this solution considering the associated performance overheads. Despite this inaccuracy, we show that our approach leads to significant reduction in task creation.

Dynamic task creation and overhead reduction. DECAF reduces redundant task creation operations in five substeps (see Figure 3). It starts by invoking LC on parallel loops in canonical form [**?**]. The next step is to introduce some template code that computes the current count of the idle workers and a set of five helper variables: i) `totWorkers`: # idle workers+1, ii) `eqChunk`: minimum number of iterations executed by any worker, iii) `actualn`: number of iterations of the parallel loop to be executed. iv) `newN`: total number of iterations to be executed by the idle workers, and v) `rem`: a temporary variable. This substep also introduces an outer while loop, which is used to avoid unstructured control flow. The third substep (Chunked-Block-Modification) modifies the chunked code to enforce the load balancing scheme discussed above. Similarly, the fourth step (Parent-Block-Generation) introduces code to be executed by the parent thread.

The final substep Serial-Block-Generation is more involved. DECAF aims to create tasks only if there are idle workers. Ideally, if there are no idle workers then we should neither execute a join operation nor spawn new tasks. In such cases the current task can be asked to complete the remaining job serially. DECAF handles this scenario, by using a simple heuristic: If at the time of task creation, no idle workers are available (`workers` = 0), then the loop under consideration should be executed serially. This heuristic is enforced by invoking the Serial-Block-Generation substep. Considering the possibility that some workers may get freed up during the life-time of this serial loop, the generated serial-code checks for available idle workers, after each iteration. And if idle workers are available, the rest of the iterations are divided into `totalWorkers` (= `workers` + 1) number of chunks to be executed in parallel.

Example: For the input code of Figure 1(a), DECAF would (i) perform aggressive finish expansion and pull the `finish` construct

```
1 def find_queens() {
2  finish { nqueens (n, 0, ...); } }
3 def nqueens(val n:Int,val j:Int, ...) {
4  var ii:Int=0;
5  var workers:Int = Runtime.retIdleWorkers;
6  outer: while(true) {
7    if(workers>0) {
8      val totWorkers:Int = workers+1;
9      val actualn:Int=n-ii;
10     val eqChunk:Int=actualn/totWorkers;
11     val newN:Int=actualn-eqChunk;
12     var rem:Int=actualn%totWorkers+workers
13     for( ; ii<newN; ) {      // ``chunked block"
14       val kx = ii+eqChunk+rem/totWorkers;
15       val ni=ii; rem--; ii = kx;
16       async {
17         for(var i:int=ni ; i<kx; i++) { ...
18           nqueens(n, j, ...);
19         } }/* async */}/* outer-for */
20     {                        // ``parent block"
21       for(var i:int=newN;i<size;i++){ ...
22         nqueens(n, j, ...);
23       } } } /* if */
24   else {                     // ``serial block"
25       for(i=0; i<n; i++) { ...
26       nqueens(n, j, ...);
27       workers = Runtime.retIdleWorkers;
28       if(workers>0  && i<n-2) {
29         ii=i+1;  continue outer;
30       }}} break; } /*while */ } /*nqueens */
```

Figure 6: DECAF applied on BOTS Nqueens kernel

to the nqueens call-site (find_queens), and (ii) reduce redundant task operations by invoking the five substeps described above – to generate code shown in Figure 6. The code computes the number of idle workers, and if workers>0, the execution continues at line 8. The if body includes a chunked parallel loop (chunked-block: executed by the idle workers), and a for-loop (parent-block: executed by the current worker). The "serial block" depicts the code generated by the 'Serial-Block-Generation' substep. At the end of each serial iteration, we check the count of the idle workers. If the count is greater than zero (and at least two iterations are left to execute, to account for the work available for the current worker and at least one of the idle workers), we create parallel tasks to execute the remaining iterations. To do so, ii is set to the number of iterations that have already been executed, and the control is transferred to line 6; at line 9, this updated value of ii is used to compute the value of actualn.

3.2.1 Impact of synchronization operations on reduction of redundant tasks . We now extend the techniques presented above to handle loops that may contain synchronization operations. Like before the code generation happens in five steps: (1) Loop Chunking: The LC substep chunks the parallel loops with N synchronization operations, such that the body of the outer chunked loop consists of a series of inner serial-for-loops ($N + 1$ of them) separated by Clock.advanceAll statement; the serial-for-loop bounds are guarded by a condition. For example, for the input (synthetic) code shown in Figure 7(a), Figure 7(b) shows the code generated by LC. (2) Template-Code-Generation: same as before. (3) Chunked-Block-Modification: The generated "chunked block" skips the first phase number of serial-for-loops (have already been executed in the "serial block"), and execute the rest of the serial-for-loops. This selective execution of the serial-for-loops is performed using a switch statement. (4) Parent-Block-Generation: The strategy for the "parent block" is similar to that followed for the "chunked block". (5) Serial-Block-Generation: To keep a tab on the complexity of the generated code, we choose a scheme in which (a) we keep track of the number of inner serial-for-loops executed (using a variable called phase, that is initialized to 0), which is incremented after the execution of each Clock.advanceAll statement. (b) we check for the availability of free workers after executing each Clock.advanceAll statement (instead of checking it after each iteration – can become too complex). Figure 7(c), shows the code corresponding to "chunked block", "parent block" and the "serial block", for the loop chunked code shown in Figure 7(b).

```
finish {
 for(var i:Int=0; i<n; i++){
  async clocked(c){S1;Clock.advanceAll();S2;}}}
                      (a)
var workers:Int = Runtime.retNthreads();
var chunkSize:Int=(n+workers-1)/workers;
finish {
  for(var ii:Int=0;ii<n;ii+=chunkSize) {
    val ni = ii;
    async clocked(c) {
      var kx:Int=ni+chunkSize; if(kx>n)kx=n;
      for(var i:Int=ni; i<kx; i++) S1;
      Clock.advanceAll();
      for(var i:Int=ni; i<kx; i++) S2; } } }
                      (b)
//``chunked block":
     for( ; ii<newN; ) {
       val kx:Int=ii+eqChunk+rem/totWorkers;
       val ni=ii; rem--; ii = kx;
       async clocked(c) {
         switch(phase) {
           case 0:for(var i:int=ni;i<kx;i++) S1;
                Clock.advanceAll();
           case 1:for(var i:int=ni;i<kx;i++) S2;
         } } /* async */ } /* outer-for */
//``parent block":
      switch(phase) {
        case 0:for(var i:Int=newN;i<n;i++) S1;
             Clock.advanceAll();
        case 1:for(var i:Int=newN;i<n;i++) S2;}
//``serial block":
    for(i=0 ; i<n; i++) S1;
    Clock.advanceAll(); phase++;
    workers = Runtime.retIdleWorkers;
    if(workers>0) { continue outer; }
    for(i=0;i<n;i++) S2;
                      (c)
```

Figure 7: Reducing task creation operations in the presence of synchronization operations. (a) Input unoptimized version, (b) LC version, and (c) DECAF version.

3.3 Discussion

Overheads due to DECAF: The code generated by DECAF may incur overheads due to possible reduction in parallelism: Consider the code transformation shown below:

```
def f1() {f2();f3();}      def f1() {
def f2() {            ⟹*     finish {f2();} f3();}
  async finish S1 }        def f2() {async S1 }
```

It can be seen that the shift of `finish` construct from the method `f2()` to its call site, inhibits the parallel execution of `S1` and the function `f3`. This overhead gets easily fixed if the scope of the `finish` can be further expanded later to include the call to `f3`.

During our evaluation, we have found that async-finish interchange is an important transformation and is invoked for each RTP benchmark under consideration. But we did not encounter any case leading to such an overhead – the scope of `finish` could always be expanded after applying async-finish interchange.

DECAF Vs a fully runtime approach: To reduce the task termination operations, DECAF involves elaborate dependence analysis and code transformation schemes that are non-local in nature (even in the absence of exceptions). Re-casting these as a runtime optimization may seem attractive, but is both non-trivial and can be expensive. Similarly, to reduce the task creation overheads, DECAF generates serial-code from the input parallel code. Doing this at runtime is non-trivial, especially in the presence of deeply nested barriers. Overall DECAF is a whole program optimizations that has intuitive compile-time implementation and reaps runtime benefits.

4 EXTENSIONS FOR EXCEPTIONS

We now discuss the extensions to the rules of DECAF discussed in Section 3, in the presence of exceptions. Though the rules are discussed in the context of X10 exception model (inspired by that of Java), the general idea can also be applied to other languages with exception semantics (for example, HJ and C++).

As per X10 semantics [**?**] when an exception is thrown in an `async`, at runtime, the exception is caught by its IEF (see Section 2). The IEF waits for termination of the remaining tasks, packages all the thrown exceptions into a new object of type `MultipleExceptions`, and throws this exception object. Note: an exception thrown in one task does not terminate the sibling tasks.

To motivate the impact of exceptions on the presented transformation rules, consider the *finish expansion upper* rule of Figure 4, being applied on the following example, where `S1` can throw an exception (of type `Ex`).

```
try{ S1; finish S2 ⟹  try{ finish { S1; S2;
} catch(e:Ex){...}      } } catch(e:Ex){...}
```

In the LHS, the exception thrown by `S1` is caught by the `catch` block. However, in the RHS, the `finish` block catches this exception and in turn throws an object of type `MultipleExceptions` – not caught by the catch block and hence not semantics preserving.

To ensure semantics preservation, we now present the required modifications to DECAF. Overall, DECAF still follows the same block diagram shown in Figure 3, but some of the individual transformation rules are modified. Note that since LC is semantics preserving in the presence of exceptions [**?**], our rules to reduce task creation operations do not alter the program semantics (even in the presence of exceptions). We now discuss the changes to the different rules, used by DECAF (to reduce the task termination overheads), to make them semantics preserving in the presence of exceptions.

1. Loop-Finish Interchange
for(S1;cond;S2) {
finish { S3
}<exlist>
}
// ++
// *e-asyncs in* cond/S2/S3
// *do not throw exceptions*
	S1; var e:Exception=null;
	var me:ME=null,v:Boolean;
	finish {
	for(; ;){ try {v=cond;}
	catch(ex:Exception)
	{e = ex; break; }
	if(e==null && v){
	try{S3}
	catch(ex:Exception){
	me=new ME(ex);break;}
	if(me==null) {
	try { exlist }
	catch(ex:Exception)
	{ e = ex; break; }
	if(e==null){
	try{S2}
	catch(ex:Exception)
	{e=ex; break;}}}}}}
	<if(e!=null) throw e;
	if(me!=null) throw me;>

2. Finish Fusion **finish**{S1}<exlist$_1$> **finish**{S2}<exlist$_2$> // ++ // *e-asyncs in* S1 *and* S2 // *do not throw exceptions.*	**finish** { S1 exlist$_1$ S2 }<exlist$_2$>

Figure 8: Rules of Figure 2, in the presence of exceptions.

Figures **??** and **??** present the extensions to the rules presented in Figures 2 and 4. As it can be seen the new rules are a lot more complicated, which underscores the importance of the compiler based automatic (in contrast to hand transformation by the programmer).

To aid the translation process, we use a temporary `finish` construct of the form "`finish {S1}<exlist>`", where `exlist` represents a sequence of conditional throw statements. Each entry in `exlist` is of the form "`if (ex != null) throw ex;`". We call `exlist` the list of *pending exceptions*. This temporary construct is translated away, at the end, using the following rule:

```
    finish{S}<exlist> ⟹ finish{S}; exlist;
```

Figure **??** presents the extensions for the two rules of Figure 2, in the presence of the exceptions. We use `ME` to refer to the X10 class `MultipleExceptions`. For brevity, we avoid re-stating the old rules specified in Figures 2 and 4 and use "// ++" to refer to them. Rule#1 ensures that `S3` is executed only if no exceptions are thrown by `cond`, `S2` and `exlist`. Rule#2 ensures that `S2` is executed only if no exception is thrown in exlist$_1$.

Figure **??** presents the extensions to the rules of Figure 4 in the presence of exceptions. Rule#3 uses a try-catch block to capture the exceptions thrown by the inner `finish` and `exlist`$_1$, and rethrow it later. The Rule #4 is similar to that shown in Figure 4. Rule #5 requires that no exceptions are thrown by the e-asyncs in `S1`. The transformed code catches the exception (if any) thrown in `S1` and throws the exception outside the `finish`. The execution of `S2` occurs only if `S1` throws no exceptions. Similarly, Rule #6 requires that no exceptions are thrown by the e-asyncs of both `S1`

```
3. Nested Finish Elimination  try { finish { S1 }
finish {                        exlist_1;
 finish {                      } catch(e:Exception) {
  S1}<exlist_1>                 val me = new ME(e);
}<exlist_2>                     throw me; }<exlist_2>
```

```
4. Finish-If interchange
                                 v = cond;
if(cond) {                       finish {
  finish { ⟹                     if(v) S1}<exlist>
    S1}<exlist>}
```

```
5. Finish Expansion Upper     var e:Exception=null;
S1;                           finish { try { S1 }
finish{S2}<exlist> ⟹           catch(e1:Exception){e=e1;}
// ++                          if(e == null) S2
// e-asyncs in S1 do not      }<if(e!=null)throw e;
// throw exceptions.          exlist>
```

```
6. Finish Expansion Lower     var e:Exception=null;
finish {  ⟹                   finish { S1;
  S1 }<exlist>                 try { exlist }
S2                             catch(e1:Exception){e=e1;}
// ++                          if(e==null){ try {S2}
// e-asyncs in S1 and S2        catch(ex:Exception){e=ex;}}
// do not throw exceptions.   }<if(e!=null)throw e;>
```

```
7. Async-Finish Interchange
async{finish {S1}}<> ⟹ finish{async{S1}}<>
// S1 throws no exceptions.
```

Figure 9: Rules of Figure 4, in the presence of exceptions.

and S2; execution of S2 occurs only if S1 and exlist throw no exceptions. Rule #7 requires that S1 does not throw exceptions. It also requires the finish has no pending exceptions. For the ease of explanation, we explain the modifications to the *Finish-Method Pull* transformation (Rule #8, Figure 4), using the following example:

```
def b() {f();}                  | var g:Ex;
def f() {                    ⟹ | def b() { var e:Ex;
 var e:Ex;                      |  finish { f(); e=g;
 finish S                       |  }<if(e!=null)throw e;>}
 <if(e!=null)throw e;>}         | def f(){var e:Ex;S;g=e;}
```

Here a new instance field g stores the exception e, inside the method f, and this exception is thrown in the callee of f.

Besides the extensions to the rules from Figure 4, in the presence of exceptions, we need another transformation to expand the scope of finish constructs – *Try-Finish Exchange*. This transformation requires that no exceptions are thrown by e-asyncs in S1.

```
9. Try-Finish Exchange       var e:Ex=null;
try {                        finish {try {try {S1}
 finish {                      catch(e1:Exception)
  S1 }<exlist> ⟹               {throw new ME(e1);}
} catch(e:Ex) { S2 }           exlist
// e-asyncs in S1 do not      }catch(e1:Ex){e=e1;}
// throw exceptions.         } if (e!=null){S2}
```

5 EVALUATION

In this section we evaluate our proposed optimization DECAF. We analyze DECAF on two different systems – a 16 core Intel system (2 Intel E5-2670 2.6GHz processors × 8 cores per processor) and a 64 core AMD system (4 AMD Abu Dhabi 6376 processors × 16 cores per processor).

We implement DECAF, as a whole program optimization, in the x10-2.3.0 compiler and present an evaluation of DECAF using the Native X10 (C++) backend. We found the compilation overheads to be negligible (< 0.05%). Each execution time reading is reported by taking an average over ten runs.

To evaluate DECAF, we used the following criteria to select kernels from the IMSuite [?] and BOTS [?] benchmark suite: a) presence of recursive task parallelism, and b) creation of asynchronous tasks only via parallel loops. Figure ?? lists all the benchmarks satisfying the selection criteria (first five from IMSuite, and the rest three from BOTS). Note that, *BFS*, *DST*, and *MST* also have their non-clocked versions in IMSuite. But we chose the clocked versions owing to their added complexity related to barriers.

Figure ?? (first two columns) provides a brief overview of the benchmarks and their respective input data sets. For each BOTS benchmark, we list the input type (e.g., Large, Medium) and for each IMSuite benchmark, we list the input size and a note if we are using the standard input or a modified one. For all the benchmarks (except *DST* and *MST*), we have used one of the standard inputs provided. For *DST* and *MST*, we found that the default inputs were not leading to much recursion (as the diameter of the input graph was around 2 or 3), thereby rendering the program nearly non-recursive. To overcome this challenge, we used their respective input generators (provided by IMSuite) to generate larger and denser graphs. For all the benchmarks, the chosen input size was the largest input such that the corresponding input program takes not more than an hour, when run on our 16-core Intel system.

5.1 Dynamic characteristics

We executed the chosen kernels on the specified inputs and collected the dynamic counts for the task creation (async) and task termination (finish) operations (with X10_NTHREADS=16). The last two columns of Figure ??, show these dynamic characteristics for the unoptimized (UnOpt), Loop Chunking (LC) and DECAF versions.

It can be seen that in comparison to both the UpOpt and LC versions, DECAF achieves a significant reduction in the number of async and finish constructs, for *BFS*, *NQ* and *BY* kernels. For *DR*, *HL* and *FL* there is a significant reduction in the number of async operations but DECAF is not able to expand the scope of any finish constructs (due to MHBD), and the reduction in the finish operations is a consequence of reduction of async operations. In case of *DST* and *MST* as the numbers of finish and async operations are low (for the UnOpt and LC versions), the reduction in their counts (because of DECAF) is also less.

5.2 Evaluation of DECAF

For varying number of cores (in the powers of two), Figure ?? compares the speedup of DECAF with respect to LC; higher the better. Figure ?? presents the speedups on the Intel system. We vary the number of cores and X10_NTHREADS from 2 to 16, in sync (i.e., for a *k* core setup, we set X10_NTHREADS to *k*). The speedup amount, for each kernel depends on a varied set of factors – the behavior of the kernel, the scope for reducing the task creation and

Kernel	Input	Type	#Finish	#Async
Bellman and Ford Breadth First Search*(*BFS*)	256	UnOpt	58k	930k
		LC	29k	343k
	(Standard)	DECAF	1	64
Byzantine (*BY*)	128	UnOpt	276k	3869k
		LC	276k	3308k
	(Standard)	DECAF	34	18k
Dijkstra Routing (*DR*)	512	UnOpt	28k	631k
		LC	28k	338k
	(Standard)	DECAF	17k	23k
Dijkstra Breadth First Search*(*DST*)	2048	UnOpt	3.2k	26k
		LC	3.2k	1k
	(Modified)	DECAF	18	338
Minimum Spanning Tree*(*MST*)	512	UnOpt	3.1k	6.3k
		LC	3.1k	2k
	(Modified)	DECAF	1.1k	1.5k
Nqueens (*NQ*)	(Large)	UnOpt	26993k	377901k
		LC	26993k	377901k
		DECAF	1	3460k
Health (*HL*)	(Large)	UnOpt	17516k	630575k
		LC	17516k	210192k
		DECAF	1636k	2851k
Floorplan (*FL*)	(Medium)	UnOpt	3678k	19244k
		LC	3657k	19193k
		DECAF	3619k	1650k

Figure 10: Benchmark statistics; starred(*) ones have barriers.

the task termination operations, the nature of the input, runtime/OS related factors and the hardware characteristics.

It can be seen that for kernels *BFS*, *DR*, *NQ* and *HL*, our technique achieves significant speedups on increasing the number of cores (and thus increasing values of X10_NTHREADS). These speedups can be attributed to the varied effects of increased parallelism on LC and DECAF. As the number of X10_NTHREADS increases, LC creates more number of tasks at each level. In contrast, DECAF creates tasks, only if idle workers are available, and thereby is able to take advantage of the increased number of cores. Hence, comparatively DECAF has low overheads and synchronization costs, which improve its relative performance. This is one of the main reasons for the sudden peak in case of *NQ* at 16 cores: the execution time for LC increases sharply due to excessive task creation, while the DECAF version maintains its scalable nature (uniform decrease in execution time), as the number of cores are increased. In case of *HL*, we observe a dip in the speedup on moving from 2 to 4 cores. This is due to the sudden improvement in performance of the LC version for four cores from two cores. We hypothesize this behavior of the LC versions to the system specific scheduling policies.

A general observation is that when the number of cores are less, the obtained speedups are less. This can be attributed to the fewer opportunities for expressing parallelism and the smaller value of X10_NTHREADS. Here, both the DECAF and the LC create few tasks at each level. Thus, DECAF is not able to record significant reduction in task creation/termination operations and show gains.

For kernels *DST* and *MST*, DECAF is unable to achieve significant speedups over LC. This behavior is due to the fewer opportunities for reduction of task creation and termination operations (number of async and finish operations < 3k, see Figure **??**).

FL is an interesting kernel where, at times, DECAF performs worse than LC. In *FL* the task creation occurs inside a doubly nested loop, while the finish construct is outside the nested loops. Also, the finish construct cannot be eliminated due to dependencies. Importantly, the inner loop does not spawn enough tasks to optimize (to see visible gains). Consequently, the DECAF has less scope for improvement, which in turn affects the comparative performance.

In case of *BY*, although DECAF decreases the number of task creation and termination operations by a good measure, the speedup is minimal. This is because, in case of *BY*, the work done by the majority of the recursively spawned tasks is negligible in comparison to the work done in the non-recursive data-parallel loops present in *BY*; the latter is optimized well by both LC and DECAF.

Figure **??** shows the behavior for the eight kernel benchmarks on the 64 core AMD system. In these plots, we vary the number of cores and X10_NTHREADS from 2 to 64, in sync. On increasing the cores from 2 to 16, we observe that the obtained speedups are similar to that of Figure **??**. Except in case of HL, where the dip in speedup discussed in the context of the Intel system, is not seen here. Thus giving credence to the hypothesis that the dip is related to some system level scheduling issues.

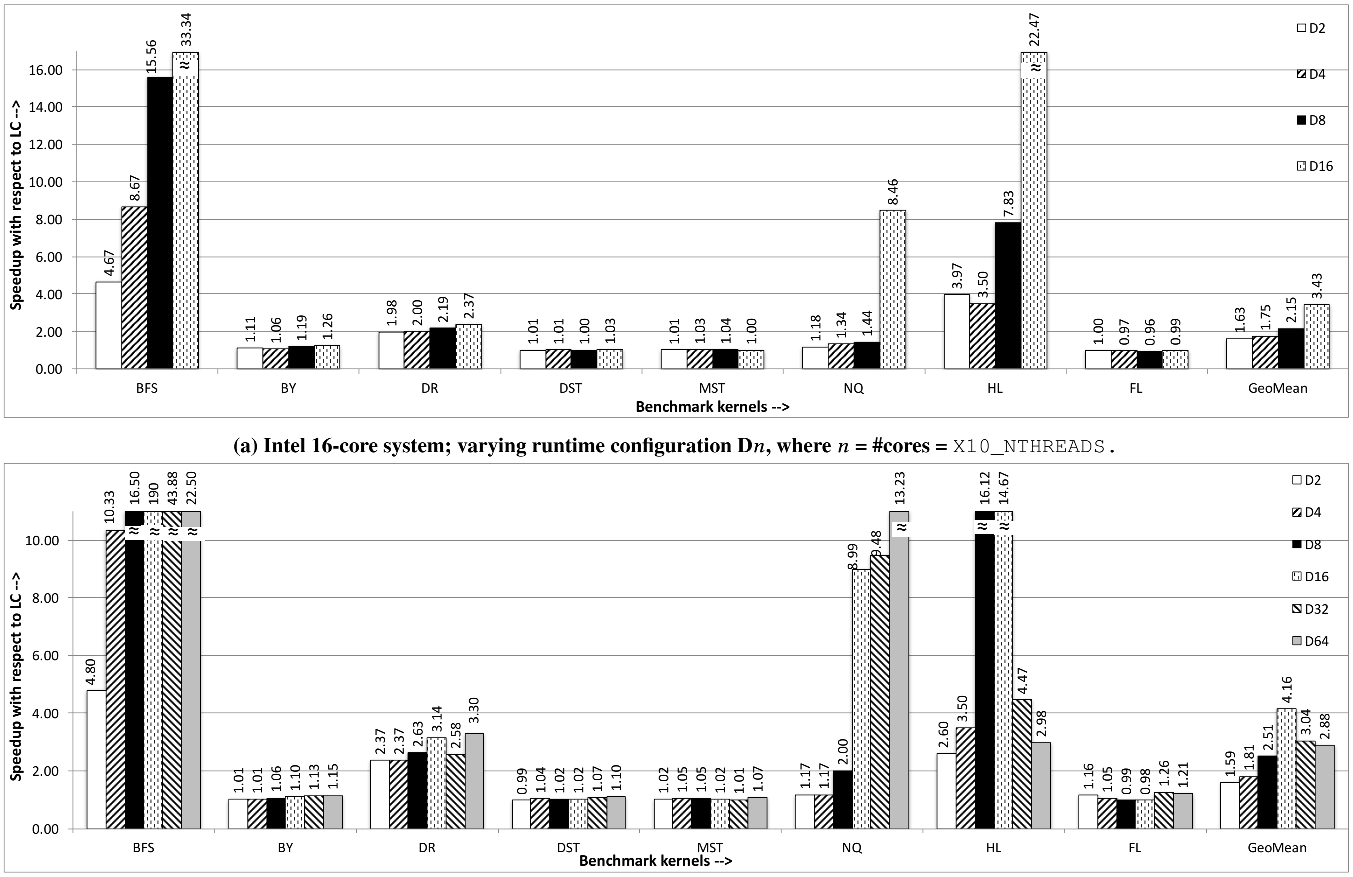
On moving from 16 - 32 - 64 cores, the improvements derived from DECAF (in comparison to LC) varied. This is mainly because of the chosen input sizes that effect (limit) the amount of parallelism in the programs. In such limiting scenarios, the gains may not be proportional to the increase in the number of cores.

For kernels *DST* and *MST*, as discussed earlier (for the Intel system), the speedups are not substantial due to less opportunities for exploiting parallelism.

Overall, with respect to the LC versions, the DECAF versions achieve speedup in the ranges of 0.96× – 33.34× (geometric mean of 2.14×), on the Intel system, and 1.07× – 22.5×, (geometric mean of 2.53×), on the AMD system. We have also compared the DECAF versions with respect to their serial counterparts and found that the speedups of the DECAF versions scale well with increasing number of cores. For example, for 2, 4, 8, and 16 cores on the 16-core Intel cluster the geomean speedup of the DECAF versions with respect to their serial counterparts are 1.25, 2.4, 4.45 and 5.4, respectively. Due to lack of space we skip the reporting of the detailed readings.

Individual impact of reduction in task creation and task termination operations. DECAF improves the performance of RTP applications along two dimensions: reducing the task creation operations and reducing task termination operations. The impact of each of these dimensions depend on the individual application and the specific input. To understand these impacts better, we implemented two other variations of DECAF: DECAF-light (DECAF that only reduces redundant task creation operations – does not apply the techniques of Section 3.1) and DECAF-LC (DECAF that uses only LC to reduce task creation operations instead of all the techniques discussed in Section 3.2). We exclude the *DR*, *HL* and *FL* kernels from this discussion, as for these three kernels, the DECAF-light versions match the DECAF versions (see Section **??**). For the remaining kernels, Figure **??** presents the execution times of the UnOpt, LC, DECAF-LC, DECAF-light, and DECAF versions, run on the AMD system (#cores = X10_NTHREADS = 64).

It can be see that for all of the five listed kernels, DECAF-light performs better than LC – the exact impact depends on the number of async operations eliminated (without creating load imbalance among the threads). To understand this consider *BY*, *BFS* and *NQ*,



(a) Intel 16-core system; varying runtime configuration Dn, where n = #cores = X10_NTHREADS.

(b) AMD 64-core system; varying runtime configuration Dn, where n = #cores = X10_NTHREADS.

Figure 11: Speedups for varying number of cores; Speedup = (execution time of LC version / execution time of DECAF version)

Kernel	UnOpt	LC	DECAF-light	DECAF-LC	DECAF
NQ	3911	4500	444	2150	340
BFS	89	90	6	87	4
BY	392	397	356	363	346
DST	2000	541	525	526	493
MST	820	293	291	287	273

Figure 12: Execution times (in seconds) on the AMD system (#cores = X10_NTHREADS = 64).

where we see that LC performs slightly worse than UnOpt. This is mainly because the LC scheme is oblivious of the load-imbalance among the created chunks – thereby increasing the critical path in some cases. In contrast, DECAF-light addresses this issue well, by assigning iterations to the free workers only.

Similarly, we see that DECAF-LC performs better than LC; the exact impact depends on the number of `finish` operations eliminated. For example, DECAF is able to reduce a good number of `finish` operations in the BY kernel and the impact can accordingly be seen. Similarly, in *NQ* where DECAF reduces a large number of `finish` operations, the impact is more visible.

Thus we find that the performance of DECAF-light and DECAF-LC are better than that of LC. And the numbers of DECAF (last column) indicate that the individual gains are adding up to realize overall bigger gains.

5.3 Energy Consumption

Considering the importance of reduction in energy consumption, we also compared DECAF and LC in terms of the energy consumed (on the Intel system). We used the Intel *Running Average Power Limit* (RAPL) [**?**] interface for this purpose. We couldn't find a similar interface for our AMD system.

Figure **??** depicts the energy consumed by the DECAF versions, normalized with respect to their LC counterparts. It can be seen that, in general, DECAF consumes less energy in comparison to LC. The amount of reduction in energy consumption varies with respect to the kernel under consideration. Overall, compared to the LC versions, the DECAF versions consume energy in the range of 0.025× – 0.997× (geometric mean 0.287×). Thus, on average DECAF consumes 71.2% less energy than LC.

We observe that the maximum energy savings is achieved for kernels *BFS*, *DR*, *NQ* and *HL*. These savings directly follow the significant reduction in the execution time, which in turn is due to the reduction in task-creation and -termination operations for these kernels. On the other hand, for *DST*, *MST* and *FL*, there isn't a

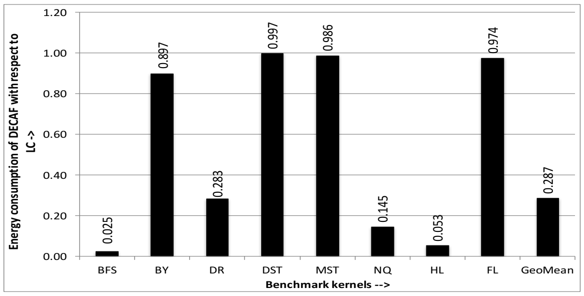


Figure 13: Energy consumption normalized to LC.

significant reduction in the energy consumption, which is due to the less task reduction opportunities available in these kernels.

6 RELATED WORK

There have been several works [**? ? ? ? ? ? ?**] that aim to reduce the overheads resulting from useless synchronization and join operations. Cytron et al. [**?**] propose reduction of synchronization constructs by translating input fork-join code to SPMD code with reduced number of barriers. Heinz and Philippsen [**?**] perform source to source transformations to reduce the barrier synchronization operations in data parallel programs. Their optimizations target the redundant synchronization operations present in the synchronous FORALL statements by converting them into simplified asynchronous FORALL statements with reduced synchronization overheads. Tseng [**?**] extends the work of Cytron et al. by using a combined fork-join and SPMD model to reduce synchronization overheads. Ferrer et al. [**?**] exploit the loop unrolling transformation in the presence of task parallel constructs. The authors try to aggregate multiple fine-grained tasks (by unrolling loop) into the larger ones to achieve performance. Noll and Gross [**?**] propose task reduction and synchronization optimizations for the JIT compilers. The authors propose an optimization that allows merging of small concurrent tasks into a large task. Compared to these, our optimizations eliminate redundant task creation and termination operations in recursive task parallel programs. Further, we present a scheme to do the transformations in a semantics preserving manner, even in the presence of fine grain synchronization (like clocks) and exceptions.

Yonezawa et al. [**?**] aim at reducing the barrier synchronization operations, by generating efficient communication code for data transfer operations in a distributed application. Similarly, Bikshandi et al. [**?**] propose methods to efficiently execute outer-most finish operations. Nagarajan and Gupta [**?**] use speculative execution to reduce the overheads associated with barriers. We believe that these techniques can be used in conjunction with our proposed DECAF, to further increase the performance gains.

Loop scheduling [**?**] has been one of the most popular techniques to efficiently execute loop nests. Some of the popular schemes of loop scheduling are static (dividing the all the iterations equally among the declared workers), dynamic (the iterations are divided into many small chunks and added to a work queue and each free worker takes a chunk from this work queue to execute), and guided (similar to dynamic, but the size of the chunks vary dynamically). Cilk [**?**] and TBB [**?**] both implement specialised mechanisms of loop scheduling at runtime, to achieve load balancing, by controlling the number of worker threads and the division of tasks among the workers. The scheme used by DECAF to reduce task creation operations, can be seen as a specialization of loop scheduling where i) iterations scheduled to be executed by the same processor are executed sequentially (no task creation overheads), ii) some iterations of the parallel loop may be executed sequentially, before dividing the rest of the loop iterations among the idle workers. Importantly, we handle codes with synchronization and exceptions.

There have been many works [**? ? ?**] that compute and assign the optimal number of processors / workers to execute a given loop nest and parallelize the loop accordingly. In contrast, we use a simple scheme of chunking parallel loops based on the number of idle worker threads (number of chunks = number of idle workers). It would be interesting to extend DECAF with more sophisticated mechanisms to compute the optimal number of worker threads.

Voss and Eigenmann [**?**] propose an inspector-executor model that at runtime decides whether to execute a loop in parallel or serially. The main emphasis behind this scheme is that benefits of executing a loop in parallel may be amortized if the overheads of parallel execution are significant. The authors first try to run a loop in parallel and measure its execution time. They next compare this execution time with that of the serial version of the loop and decide whether to run the next versions of this loop in parallel or not.

There have been several prior works that control the parallelism based on different kinds of thresholds (all measured at runtime). For non RTP programs, some of the popular threshholds are system load [**? ?**], size of the data structures [**? ?**] giving an estimation of the time the code to be parallelized may take to execute, and profile based estimated workload in different iterations [**?**]. For RTP programs, Duran et al. [**?**] show the use of a static value of recursion depth as a cut-off for parallelization. In general it is non-trivial to introduce cut-off related code, especially in programs with barriers and exceptions, as it may require significant amount of rewriting of the "serial" part (a serious drawback for programs written in higher level languages like X10). Further, even for programs with no barriers and exceptions, obtaining judicious cutoffs for RTP programs is quite challenging. The best value of cutoff depends on the nature of the benchmark, input, and the number of hardware cores. Consequently, the scheme of Iwasaki and Taura [**?**] does not identify the terminating conditions (required for static identification of cut-offs) for RTP kernels like those from IMSuite. Similarly, dynamic cut-offs based on runtime parameters [**?**] have also been used for RTP programs. Their approach requires additional monitoring threads which can impact the overall performance. Further, it is unclear how to extend their scheme to programs with synchronization and exceptions.

Recently there have been attempts to aggregate kernel launches [**?**] and consolidate workloads [**?**] in the context of GPUs. These in turn leads to reduction in the associated runtime overheads. Our proposed approach reduces task creation and termination overheads in RTP programs, even in the presence of synchronization and exceptions.

Thoman et al. [**?**] present a scheme that emits multiple versions of the code, one of which is executed at runtime, based on the availability of resources. Their idea revolves around the notion of task unrolling that is akin to inlining one or more invocations of a recursive function executed by the task. The resulting code will most likely contain join operations. In complete contrast, DECAF lifts redundant join operations outside the method. Further, their procedure to eliminate intra-procedural redundant join operations does not handle clocks and exceptions. The absence of a rule like async-finish interchange further limits the scope of their optimization. Although DECAF also generates multiple versions of the code, it has three fundamental differences: (i) handles parallel loops, (ii) conditional serialization in the presence of barriers, and (iii) switching to parallel execution, if idle workers are found during serial execution.

Lifflander et al. [**?**] present a runtime approach to improve data locality and load balance. The authors advocate coarser work steals which are identified by analysing a steal tree. In contrast, DECAF creates coarser tasks by combining a set of iterations, and executing that cluster sequentially. Further, our proposed scheme can switch from serial to parallel execution based on the available idle workers.

Flattening [**?**] transforms irregular nested (data-parallel) computation over nested data structures (for example, arrays of arrays) to regular computation on flat arrays; it also reduces synchronization-operations [**?**]. In contrast, DECAF reduces task-termination operations in recursive-task-parallel programs (for example, IMSuite kernels), where it is non-trivial to apply flattening.

Our idea of task creation based on worker availability and "serial block" can be seen as a compiler based extension of lazy-binary splitting (LBS) scheme [**?**] for RTP programs and programs with synchronization operations. the existing *eager* work-stealing algorithms. It would be interesting to evaluate the effect of DECAF on an LBS based runtime scheduler.

We are not aware of any past work that supports optimizing RTP programs in the presence of synchronization, and exceptions as in this paper, for languages that support dynamic parallelism with fine grain synchronization.

7 CONCLUSION

We present a new optimization DECAF to reduce the task creation and termination overheads in recursive task parallel (RTP) programs. This optimization improves the performance, both in terms of execution time and energy consumption. We implemented DECAF in the X10v2.3 compiler and performed experiments on two different hardware systems (a 16-core Intel system and a 64-core AMD system). Compared to the loop chunking scheme of Nandivada et al. [**?**], DECAF achieved significant improvements in execution time (geomean of 2.14× and 2.53×, on the Intel and AMD system, respectively), and substantial reduction in the energy consumption (geomean 71.2% on the Intel system). These significant improvements attest to the scope of the proposed optimizations. Though our results are shown in the context of X10, we believe that DECAF can be applied (with similar effect) to other task parallel languages like OpenMP, Chapel, Cilk and HJ that admit RTP programs.

Acknowledgements: This work is partially supported by DAE research grant 2012/36/54-BRNS/2943 and DST Fasttrack grant SB/TP/ETA-166/2012.

REFERENCES

[] G. Aharoni, D. G. Feitelson, and A. Barak. 1992. A Run-Time Algorithm for Managing the Granularity of Parallel Functional Programs. *JFP* 2, 4 (Oct 1992), 387–405.

[] L. Bergstrom and J. H. Reppy. 2012. Nested data-parallelism on the GPU. In *ICFP*. 247–258.

[] G. Bikshandi, J. G. Castanos, S. B. Kodali, V. K. Nandivada, I. Peshansky, V. A. Saraswat, S. Sur, P. Varma, and T. Wen. 2009. Efficient, portable implementation of asynchronous multi-place programs. In *PPoPP*. ACM, 271–282.

[] G. E. Blelloch and G. Sabot. 1990. Compiling Collection-Oriented Languages onto Massively Parallel Computers. *J. Parallel Distrib. Comput.* 8, 2 (1990), 119–134.

[] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. 2011. Habanero-Java: The New Adventures of Old X10. In *PPPJ*. ACM, 51–61.

[] O. Certner, Z. Li, P. Palatin, O. Temam, F. Arzel, and N. Drach. 2008. A Practical Approach for Reconciling High and Predictable Performance in Non-Regular Parallel Programs. In *DATE*. 740–745.

[] B.L. Chamberlain, D. Callahan, and H.P. Zima. 2007. Parallel Programmability and the Chapel Language. *IJHPCA* 21, 3 (Aug 2007), 291–312.

[] R. Cytron, J. Lipkis, and E. Schonberg. 1990. A Compiler-Assisted Approach to SPMD Execution. In *SC*. IEEE, 398–406.

[] A. Duran, J. Corbalán, and E. Ayguadé. 2008. An Adaptive Cut-off for Task Parallelism. In *SC*. IEEE Press, Article 36, 11 pages.

[] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade. 2009. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In *ICPP*. IEEE Computer Society, 124–131.

[] R. Ferrer, A. Duran, X. Martorell, and E. Ayguadé. 2010. Unrolling Loops Containing Task Parallelism. In *LCPC*. 416–423.

[] M. Frigo, C. E. Leiserson, and K. H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI*. 212–223.

[] Y. Guo, R. Barik, R. Raman, and V. Sarkar. 2009. Work-first and help-first scheduling policies for async-finish task parallelism. In *IPDPS*. 1–12.

[] S. Gupta and V. Krishna Nandivada. 2015. IMSuite: A Benchmark Suite for Simulating Distributed Algorithms. *JPDC* 75, 0 (Jan 2015), 1 – 19.

[] I. E. Hajj, J. Gómez-Luna, C. Li, L. Chang, D. S. Milojicic, and W. W. Hwu. 2016. KLAP: Kernel launch aggregation and promotion for optimizing dynamic parallelism. In *MICRO*. 1–12.

[] M. W. Hall and M. Martonosi. 1998. Adaptive Parallelism in Compiler-Parallelized Code. *Concurrency-Pract Ex* 10, 14 (1998), 1235–1250.

[] E. A. Heinz and M. Philippsen. 1993. *Synchronization Barrier Elimination in Synchronous FORALL Statements*. Technical Report No. 13/93. University of Karlsruhe, Department of Informatics.

[] L. Huelsbergen, J. R. Larus, and A. Aiken. 1994. Using the Run-time Sizes of Data Structures to Guide Parallel-thread Creation. In *LFP*. ACM, 79–90.

[] Intel. 2014. Intel 64 and IA-32 Architectures Software Developer's Manual. (2014).

[] S. Iwasaki and K. Taura. 2016. A Static Cut-off for Task Parallel Programs. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. ACM, 139–150.

[] K. Kennedy and J. R. Allen. 2002. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc.

[] D. A. Kranz, R. H. Halstead, Jr., and E. Mohr. 1989. Mul-T: A High-performance Parallel Lisp. In *PLDI*. ACM, 81–90.

[] J. Lifflander, S. Krishnamoorthy, and L. V. Kale. 2014. Optimizing Data Locality for Fork/Join Programs Using Constrained Work Stealing. In *SC*. 857–868.

[] S. S. Muchnick. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann.

[] V. Nagarajan and R. Gupta. 2010. Speculative Optimizations for Parallel Programs on Multicores. In *LCPC*. 323–337.

[] V. K. Nandivada, J. Shirako, J. Zhao, and V. Sarkar. 2013. A Transformation Framework for Optimizing Task-Parallel Programs. *ACM Trans. Program. Lang. Syst.* 35, 1 (April 2013), 3:1–3:48.

[] A. Noll and T. R. Gross. 2012. An Infrastructure for Dynamic Optimization of Parallel Programs. In *PPoPP*. ACM, 325–326.

[] OpenMP. 2008. OpenMP Application Program Interface, ver 3.0. (May 2008). http://www.openmp.org/mp-documents/spec30.pdf

[] L. Prechelt and S. U. Hánssgen. 2002. Efficient Parallel Execution of Irregular Recursive Programs. *IEEE TPDS* 13, 2 (Feb. 2002), 167–178.

[] J. Reinders. 2007. *Intel Threading Building Blocks*. O'Reilly Media.

[] V. Saraswat, B. Bard, P. Igor, O. Tardieu, and D. Grove. 2012. *X10 Language Specification Version 2.3*. Technical Report. IBM.

[] P. Thoman, H. Jordan, and T. Fahringer. 2014. Compiler Multiversioning for Automatic Task Granularity Control. *Concurr. Comput. : Pract. Exper.* 26, 14 (Sept. 2014), 2367–2385.

[] Chau-Wen Tseng. 1995. Compiler Optimizations for Eliminating Barrier Synchronization. In *PPoPP*. ACM, 144–155.

[] A. Tzannes, G. C. Caragea, R. Barua, and U. Vishkin. 2010. Lazy Binary-Splitting: A Run-time Adaptive Work-stealing Scheduler. In *PPPoP*. ACM, 179–190.

[] M. Voss and R. Eigenmann. 1999. Reducing Parallel Overheads through Dynamic Serialization. In *IPPS/SPDP*. 88–92.

[] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S. Liao, C. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. 1994. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *SIGPLAN Not.* 29, 12 (Dec. 1994), 31–37.

[] H. Wu, D. Li, and M. Becchi. 2016. Compiler-Assisted Workload Consolidation for Efficient Dynamic Parallelism on GPU. In *IPDPS*. 534–543.

[] N. Yonezawa, K. Wada, and T. Aida. 2006. Barrier Elimination Based on Access Dependency Analysis for OpenMP. In *ISPA*. 362–373.

[] K.K. Yue and D.J. Lilja. 1996. Efficient Execution of Parallel Applications in Multiprogrammed Multiprocessor Systems. In *IPPS*. 448–456.