Task Static Single Assignment (T-SSA) and Its Use in Safe Program Parallelization (short paper)

Chen Ding and Lingxiang Xiang
University of Rochester
cding@cs.rochester.edu

Abstract

The mechanism of copy-on-write is recently used in speculative and deterministic parallel systems. In these systems, copy-on-write happens when a task modifies a variable for the first time. At the program level, the copying is a form of renaming. This paper extends the formalism of static single assignment (SSA) to introduce a program representation called task static single assignment (T-SSA). The new representation integrates the copy-on-write renaming across tasks with the conventional SSA renaming within a task. This position paper defines the T-SSA form and demonstrates its use in making copy-on-write explicit, automatic, and automatically enhanced for safe program parallelization.

1. Introduction

Many parallel languages and interfaces such as p-threads, OpenMP, and Cilk provide mechanisms to create concurrent tasks that communicate through shared memory. A fundamental difficulty is the data race. Some recent systems adopted a two-step strategy to eliminate data races. The first step is data copy-on-write when a (concurrent) task is executing. The new copy stores the changed data locally within each task. After the task finishes, the second step is data commit, which merges the local changes with those made by other tasks. The data commit happens sequentially, one task at a time. The two steps, copy-on-write and sequential commit, eliminate data races because there is no concurrent updates to the same memory location.

The strategy has been used for speculative parallelization in behavior-oriented parallelization (*BOP*) [4, 5] and copy-or-discard tasks (Cord) [9]. Speculative parallelization is safe in that the parallelization does not alter the program output. *BOP* provides an interface that allows a programmer to suggest a possible-parallel region (*PPR*). In this paper we refer to a program annotated with *PPR* as *a safe parallel program*. To ensure safety, *BOP* implements *PPR* tasks as Unix processes, monitors their data access using program annotation or OS paging support, checks for conflicts and re-runs the tasks in the sequential order if needed. In this paper, we focus on the compiler implementation of scalar variables—their copy-on-write and commit operations—to support this type of speculative parallelization.

Copy-on-write is a form of renaming. For sequential programs, a compiler performs systematic renaming using the static single assignment (SSA) form [2, 3]. SSA removes false dependences and permits the full freedom of code motion. It encodes precise def-use relations as well as the exact point in control flow when two values "merge". Most major compilers use a form of SSA as the intermediate code representation as a common basis to implement various program transformations. SSA is most effective for scalar variables. The effect is limited for heap and array data because of the problems of aliasing and data indirection.

This paper augments SSA to define task SSA (T-SSA) to represent a safe parallel program. Task SSA makes three additions to SSA. First, T-SSA adds task-based renaming to remove false dependences (write-after-read and write-after-write) on scalar variables. Second, it uses static analysis to identify the joint point of a *PPR* task based on the true dependence (on scalar variables). Finally, it introduces a special merge function called omega merge to commit renamed data (at a join point).

T-SSA implements renaming at the program level, removing the need for OS-based copy-on-write for scalar variables. More importantly, it makes inter-task value flow explicit and may support compiler optimization of safe parallel code for the same reasons that SSA supports compiler optimization of sequential code.

2. Task Static Single Assignment (T-SSA)

We first describe the interface of safe parallelization and then defines T-SSA as its implementation.

2.1 Safe Parallel Code

We augment a sequential programming language with the primitive PPR (possibly parallel region), which brackets a block of code we refer to as a PPR *block*. The

annotation suggests parallelism between the *PPR* block and its subsequent code. A suggestion does not change semantics—the execution with suggestions always produces the same output as the execution without suggestions. A suggestion may be inserted manually by a user or automatically by a tool. It can express partial parallelism where a *PPR* task is parallel in some but not all executions. A suggestion can be wrong. Operationally, the *PPR* annotation can be viewed as an optimistic spawn or fork. As a hint, the user is not required to insert the matching join points for the spawn (although a suggestion can be made through a dependence hint [5]). In its general form, *PPR*s may be nested.

The task annotation creates a task language rather than a parallel language. We define tasks using a canonical, sequential execution. A *PPR* block is made into a task each time it is executed. Moreover, the code outside a *PPR* block is also made into a task. If *PPR*s do not overlap, the canonical execution is a series of *PPR* and non-*PPR* tasks. With nested *PPR*s, the execution is divided into a tree of tasks, whose linear relationship can be encoded [7]. In a canonical execution, every instruction belongs to a task with a unique id, and the task ids have a sequential order. We refer to "earlier" and "later" tasks by their canonical order. The goal of a parallel implementation is to maximize the parallelism between each task and its subsequent execution, that is, its later tasks.

2.2 The T-SSA Form

In the SSA form, variables are renamed so that each assignment defines a new name, and each use refers to a single definition [2, 3]. The task SSA extension is three-fold:

- PPR *id and* PPR *join.* Each *PPR* block has a unique static id p_x. The join function *PPR*_join(id) waits for all earlier p_x tasks (before the current task, so p_x can "join itself", i.e. call *PPR*_join(p_x)).
- *Task subscript.* Each variable has the task id, t, as the second subscript (the first being the SSA subscript). The new subscript symbolizes the renaming in different tasks.
- *Omega merge.* The omega merge takes a set of renamed versions of a variable and chooses one from the *latest task*, i.e. the one with the largest task subscript. The symbol ω comes from the same vernacular as ϕ in the original SSA, and its shape visually resembles the merge of parallel sources.

As a simple example, consider the program in Listing 1. Variable x is first assigned in a *PPR* to compute y and then assigned again after the *PPR* to compute z. In the implementation shown by the T-SSA form in Listing 2, two assignments are run in parallel writing to two renamed xs. The data race is removed by renaming. After the parallel execution, the surviving x is determined by the $omega$ merge, which chooses the x from the second task.

Listing 1: *PPR* code

```
task {
    x = 1
    y = x + 2
}
x = 3
z = x + y
```

Listing 2: The T-SSA form

```
p1 = ppr_spawn {
    x1,t1 = 1
    y1,t1 = x1,t1 + 1
}
x2,t2 = 3
ppr_join( p1 )
x3,t3 = ω(x1,t1, x2,t2) = x2,t2
y2,t3 = ω(y1,t1) = y1,t1
z1,t3 = x3,t3 + y2,t3
```

Figure 1: An example safe parallel program and its task SSA form, which adds the *PPR* spawn, join, task subscript, and ω merge.

As a parallel form, T-SSA has to go beyond SSA. First, it provides task-based renaming (in addition to assignment-based renaming in SSA), so it removes races and false dependences. Second, it determines when a task should be joined based on the def-use relation. Third, at the join, it merges T-SSA names based on the task subscript.

T-SSA is not entirely static in the first and the third step. The task subscript is dynamic. Still to analyze it in a compiler, we give static ids assigned differently for each static *PPR* code block and inter-*PPR* code region. In contrast, the ϕ merge in SSA is static evaluated entirely at compile time (through SSA deconstruction [2]). In general, the $omega$ merge may be a dynamic operation. In the 2-task example in Listing 2, the omega function is statically evaluated, demonstrating a benefit of compiler analysis.

2.3 T-SSA Construction

T-SSA construction follows the 4-step process below. It is similar to SSA construction [2] except for the middle two steps of inserting join points, omega merges, and task subscripts.

1. ϕ function insertion, same as in SSA construction by computing the dominance frontiers [2].
2. ω function and task join insertion.
3. renaming variable assignments with the task subscript
4. renaming variable uses through reaching definition analysis, as in SSA construction [2].

The second step is the most complex part of the T-SSA extension. The basic problem is join-point insertion, with the following requirements.

- *Safety.* For each variable read, *use(x)*, in task t, all tasks spawned before t containing a *def(x)* must be joined before *use(x)* in t.
- *Parallelism.* A task should not be forced to join until one of its results is used by another task.
- *Efficiency.* A task should not be joined again if it is already requested by a previous join.

The algorithms are omitted for lack of space. The insertion uses maybe-parallel analysis and solves a data-flow problem in ways similar to methods used in the compilers for explicitly parallel code (e.g. [1, 8]).

3. T-SSA based PPR Hint Improvement

T-SSA makes data flow explicit in the parallel code, in particular the def-use relations within and across *PPR* tasks. The information can be analyzed statically to transform a program to increase its parallelism.

***Moving dependent operations outside a* PPR** Consider the example in Listing 3. The loop uses a pointer variable n to iterate through a list of nodes and a counter variable c to store the node count. Because the *PPR* block includes the pointer move and the counter increment, there is no parallelism since these operations are mutually dependent, as shown by the T-SSA form in Listing 4.

A compiler can inspect the def-use of n, c, move their assignments outside the *PPR* block and enable the parallelism among the computations on different nodes. There is no run-time merge (ω is replaced by ϕ). Neither there is a join after the loop, so the node count c can be printed in parallel when the *PPR* tasks of the loop are still executing. The resized *PPR* block is shown in Listing 5. The inter-task value flow is changed, as reflected by the T-SSA names.

4. Implementation

The T-SSA system is implemented in two separate parts, a T-SSA compiler pass in GCC 4.2.2 and a runtime library. The compiler pass takes GIMPLE intermediate representations generated by GCC's front-end as input, and transforms all *PPR* blocks into their T-SSA form using the algorithm in Section 2.3. Each *PPR* block is replaced by an equivalent anonymous function so that it can be executed as a thread at runtime. Task creations, task joins (ϕ functions), and omega merges are implemented as corresponding ABI calls to the runtime library.

The runtime uses pthread as backend, mapping an active task to a running thread. Each thread has its own local *PPR* stack, where the variables with inter-task dependence are kept (the compiler pass decides the layout of task stack), so modifications to these variables are temporarily buffered in local task stack. When a task thread is joined by a ϕ function, the dirty part of its local stack will be merged to a global stack. Since at a join point, the runtime joins a *PPR*'s threads in their creation order, the last modification to a same variable in the task stack is always visible. In other words, the sequential semantics of original code is maintained in its T-SSA equivalence.

The runtime also limits the number of concurrent threads for a *PPR* so as not to spawn redundant threads. In addition, thread pool technique is used to minimize the overhead of thread creation.

5. Preliminary Results

A simple matrix manipulation microbenchmark is used for preliminary testing of the T-SSA system. For each row in the matrix, the microbenchmark (Listing 6) first fills it with random values, then sorts it in order to find out its medium value, which is compared with a constant ($THRESHOLD$). A boolean array ($flags$) keeps comparison results for all rows.

Listing 6: T-SSA microbenchmark

```
int data[N_ROW][N_COL];
bool flags[N_ROW];
for (int r=0; r < N_ROW; r++) {
  task {
    random_fill(row);
    sort(row);
    int medium = data[r][N_COL/2];
    flags[r] = medium < THRESHOLD;
  }
}
```

The preliminary tests were conducted on a machine with a dual-core Intel Core2 Duo E4400 CPU and a machine with a quad-core Intel Xeon E5520 CPU. The latter supports hyper-threading, so 8 hardware threads are available. On both machines, the maximum number of concurrent threads is limited to the number of hardware threads. Compared to sequential code, T-SSA code results in 1.90x speedup on Core2 Duo E4400, and 6.33x speedup on Xeon E5520.

6. Related Work

SSA and its use are expounded in the textbook by Cooper and Torczon [2]. Most prior work on SSA is concerned with optimizing scalar variables in sequen-

Listing 3: A loop

```
n = head
c = 0
while
   n ≠ nil
 task {
   foo( n )
   n = n → nxt
   c = c + 1
 }
end
print c
```

Listing 4: T-SSA

```
n_{1,t} = head
c_{1,t} = 0
while
   ppr_join(p_1)
   n_{2,t} = ω(n_{1,t}, n_{3,p1})
   c_{2,t} = ω(c_{1,t}, c_{3,p1})
   n_{2,t} ≠ nil
 p_1 = ppr_spawn {
   foo( n_{2,t} )
   n_{3,p1} = n_{2,t} → nxt
   c_{3,p1} = c_{2,t} + 1
 }
end
print c_{2,t}
```

Listing 5: *PPR* resizing

```
n_{1,t} = head
c_{1,t} = 0
while
   n_{2,t} = φ(n_{1,t}, n_{3,t})
   c_{2,t} = φ(c_{1,t}, c_{3,t})
   n_{2,t} ≠ nil
 p_1 = ppr_spawn {
   foo( n_{2,t} )
 }
 n_{3,t} = n_{2,t} → nxt
 c_{3,t} = c_{2,t} + 1
end
print c_{2,t}
```

Figure 2: T-SSA construction and *PPR* resizing for a while-loop. The original *PPR* permits no parallelism while the resized one does.

tial code [2] or arrays in data parallel code [6]. T-SSA addresses the copying and merging of scalar variables in task parallel code, which requires maybe-parallel analysis and new types of program transformations to improve parallelism and reduce overhead especially in copying and merging.

The CorD system of speculative parallelization has addressed similar issues of copying and merging using compiler analysis [9]. The paper did not mention an SSA-based formulation. Other types of compiler analysis have been developed to improve parallel or transactional code e.g. in race checking and sandboxing, but they do not target the particular strategy of copy-on-write and sequential commit. Moving code across a task boundary is rarely permitted since it affects the parallel semantics. T-SSA eliminates the race condition by renaming. Being more restrictive than parallel semantics, sequential equivalence permits dependence-based automatic transformation, which the new task SSA form is designed to support.

7. Discussion

T-SSA targets only scalar variables and hence does not fully guarantee safety of a program execution. The run-time speculation support of conflict checking and error recovery are still needed for the rest of program data [4, 5].

References

[1] S. Agarwal, R. Barik, V. Sarkar, and R. K. Shyamasundar. May-happen-in-parallel analysis of X10 programs. In *Proceedings of PPOPP*, pages 183–193, 2007.

[2] K. Cooper and L. Torczon. *Engineering a Compiler, 2nd Edition*. Morgan Kaufmann, 2010.

[3] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Prog. Lang. and Sys.*, 13(4):451–490, 1991.

[4] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *Proceedings of PLDI*, pages 223–234, 2007.

[5] C. Ke, L. Liu, C. Zhang, T. Bai, B. Jacobs, and C. Ding. Safe parallel programming using dynamic dependence hints. In *Proceedings of OOPSLA*, pages 243–258, 2011.

[6] K. Knobe and V. Sarkar. Array SSA form and its use in parallelization. In *Proceedings of POPL*, San Diego, CA, Jan. 1998.

[7] J. M. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of the ACM/IEEE conference on Supercomputing*, pages 24–33, 1991.

[8] M. M. Strout, B. Kreaseck, and P. D. Hovland. Data-flow analysis for MPI programs. In *Proceedings of ICPP*, pages 175–184, 2006.

[9] C. Tian, M. Feng, V. Nagarajan, and R. Gupta. Copy or Discard execution model for speculative parallelization on multicores. In *Proceedings of ACM/IEEE MICRO*, pages 330–341, 2008.